Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- System Boot

Objectives

- To describe the services an OS provides to users, processes, and other systems

- To discuss the various ways of structuring an OS

- To explain how OS boot

Operating System Services

- User interface
- Program execution
- I/O operations
- File-system manipulation
- Communications
- Error detection
- Resource allocation
- Accounting
- Protection and security

User Operating System Interface - CLI

- CLI allows direct command entry
  - Sometime implemented by kernel
  - Sometime implemented by program
  - Sometimes multiple flavors implemented – shells
  - Fetches a command from user and executes it

- Commands
  - Sometimes built-in
  - sometimes just names of programs
    - Adding new features doesn’t require shell modification

User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor/touchscreen
  - WIMP-Window, Icon, Menu, Pointing device
  - Invented at Xerox PARC in 1973
- Many systems include both CLI and GUI
  - Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI interface with shells
  - Linux and Solaris are CLI with optional GUI interfaces

System Calls（系统调用）

- Programming interface to the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API)
- Three most common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems
    - including all versions of UNIX, Linux, and Mac OS X
  - Java API for the Java virtual machine (JVM)

Example of Standard API

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

return value

```
BOOL   ReadFile c  (HANDLE        file,
                    LPVOID        buffer,
                    DWORD         bytes To Read,
                    LPDWORD       bytes Read,
                    LPOVERLAPPED  ovl);
```

function name

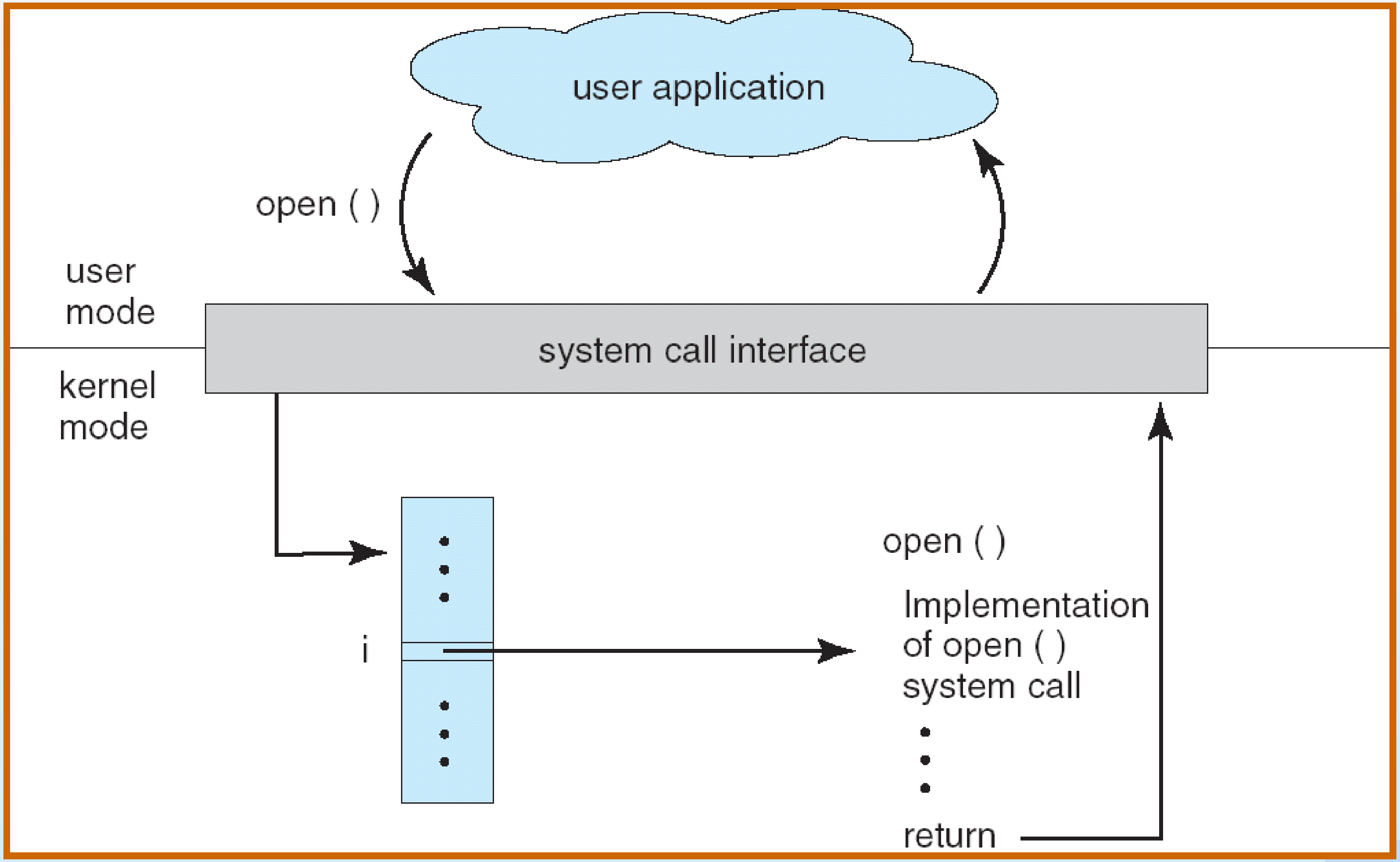
parameters

- 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

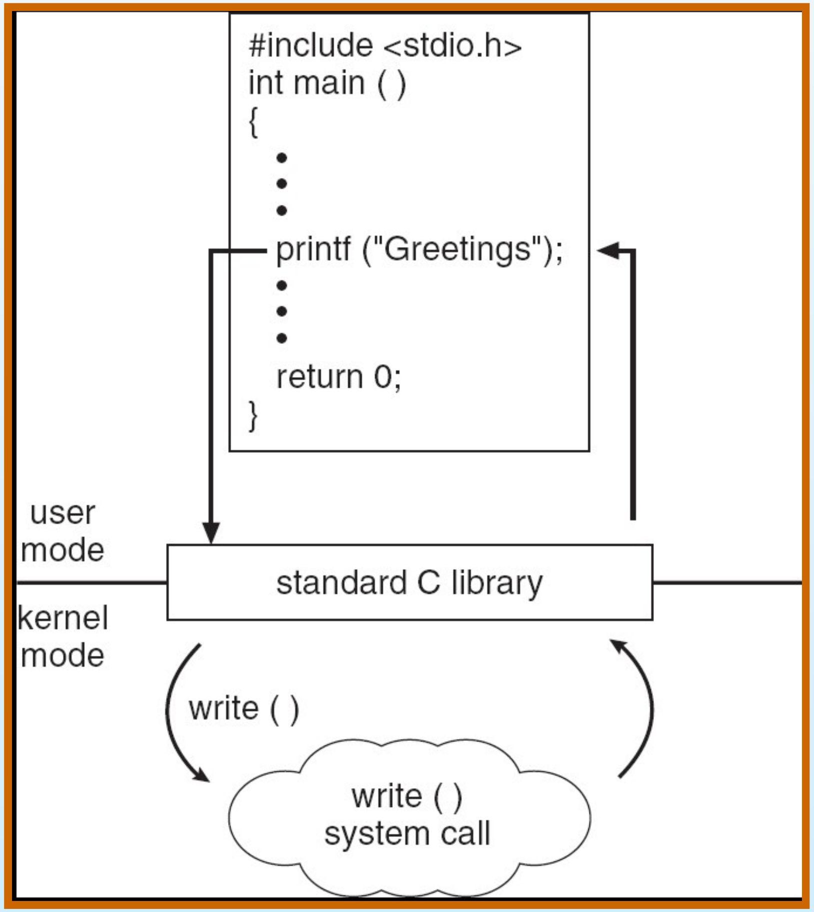
- A number associated with each system call
- System-call interface
  - Maintains a table indexed by system-call numbers
  - Invokes intended system call in OS kernel
  - Returns status of the system call and any return values
- The caller needn't know how the system call is implemented
  - Just obey API and understand what OS will do
  - APIs are managed by run-time library
    - 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



Linux strace Hello World

$ strace ./hello

System Call Parameter Passing

- Pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - May be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed in a register
    - This approach taken by Linux and Solaris
  - Parameters *pushed* onto the *stack* by the program and *popped* by the OS
  - Block and stack do not limit the number or length of parameters

Types of System Calls

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

System Programs

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

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different OS can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
  - User goals
    - Convenient to use, easy to learn, reliable, safe, and fast
  - System goals
    - 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?

- **The separation of policy from mechanism is a very important principle**
  - **it allows maximum flexibility if policy decisions are to be changed later**

Which language is your OS implemented by?

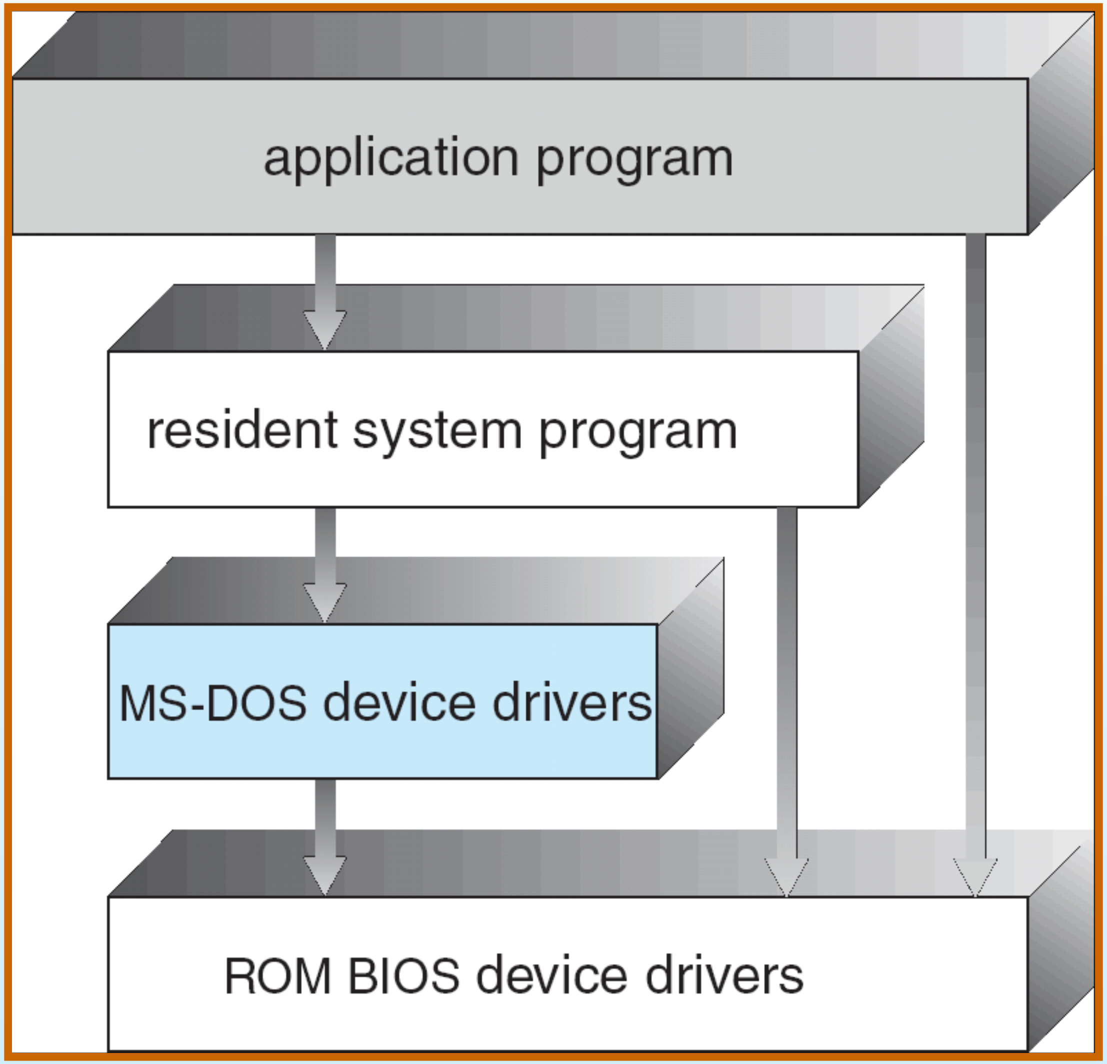
Operating System Implementation

- Long long ago, written by ASM
  - MS-DOS
- Now, written by C and C++, minor ASM
  - Unix, Linux, Windows…
- Advantages of higher-level language
  - How about ASM vs. C?
  - Portability
- Disadvantages of higher-level language
  - Performance

Operating System Structure

- Simple structure
- Layered approach
- Microkernel
- Modules
- Virtual machine

MS-DOS Layer Structure

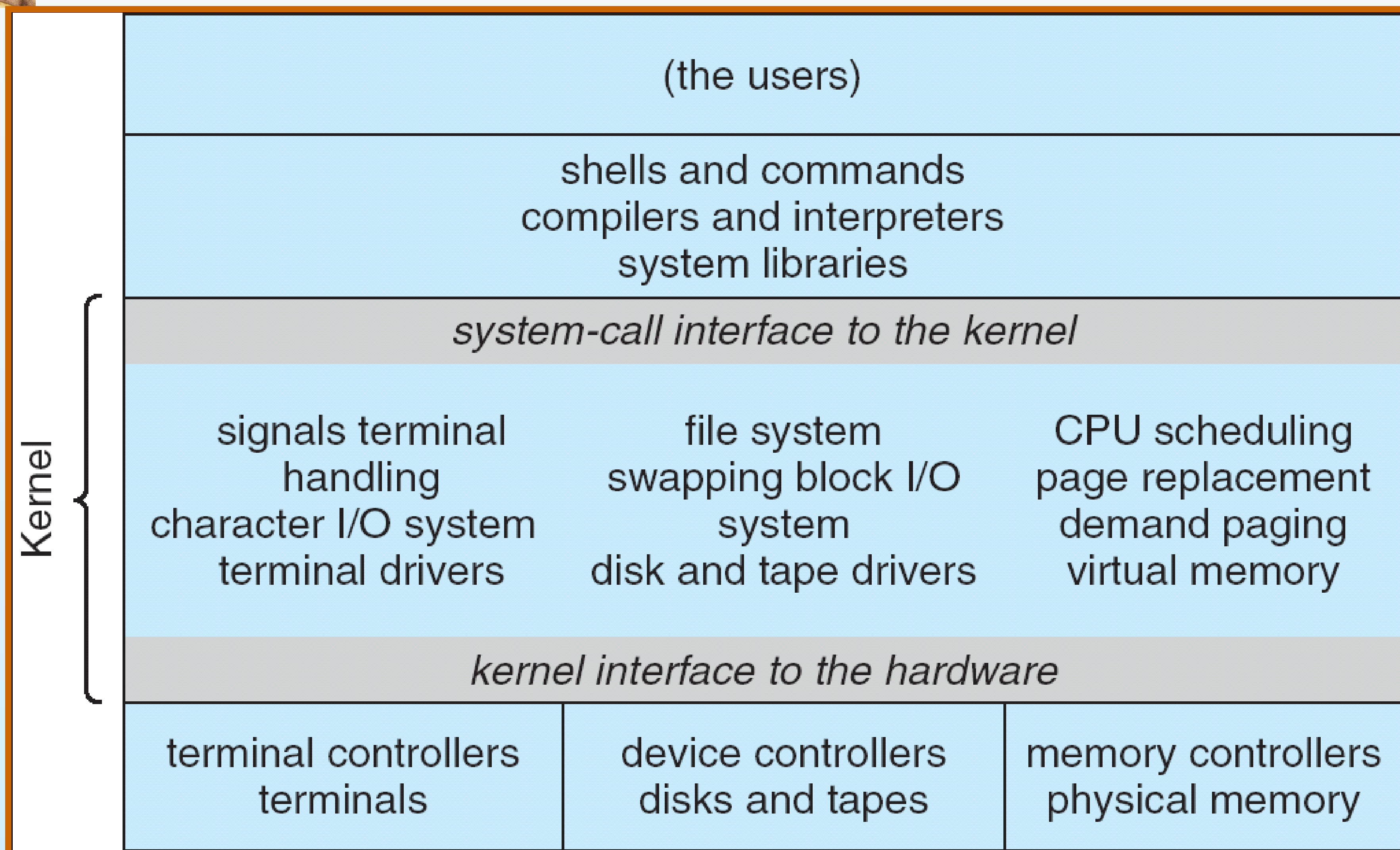


Simple Structure

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

- MS-DOS was limited by hardware of its era

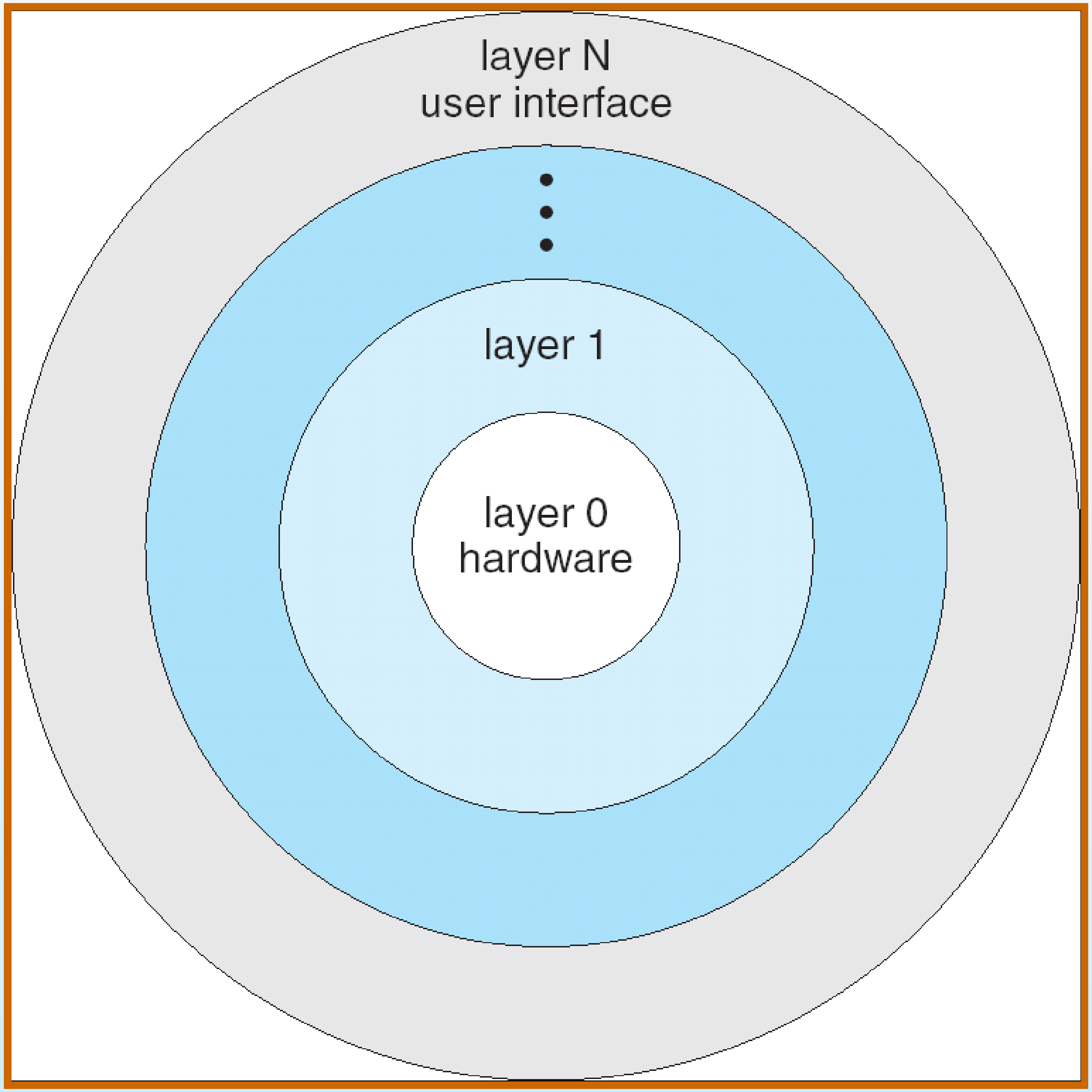
UNIX System Structure



UNIX

- The original UNIX OS had limited structuring
- Consists of two separable parts
  - System programs
  - The kernel
    - Everything below the system-call interface and above the hardware
      - The file system, CPU scheduling, memory management, and other OS functions
    - A large number of functions for one level

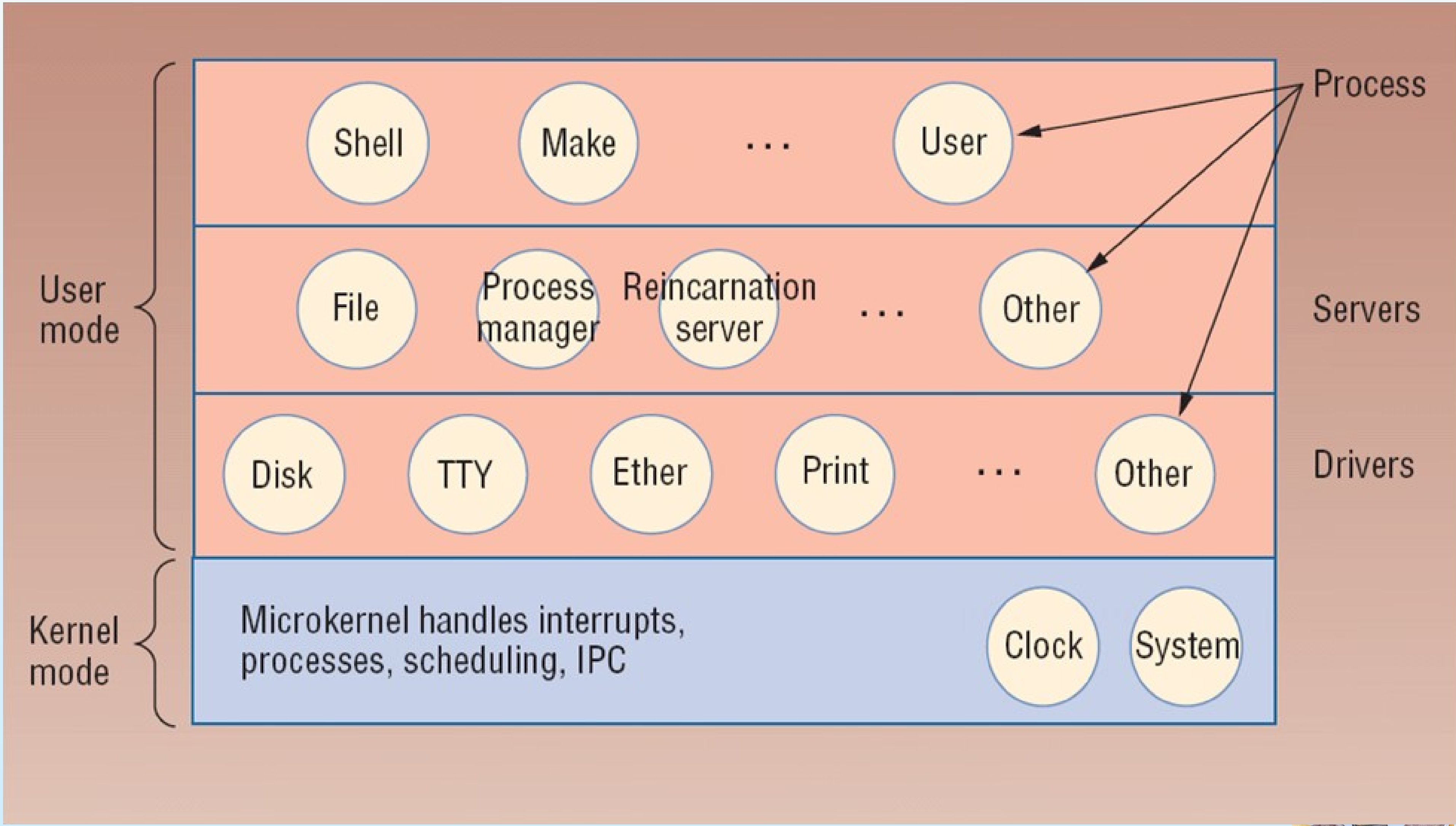
Layered Operating System



Layered Approach

- The OS is divided into a number of layers (levels)
  - Each layer 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

MINIX 3

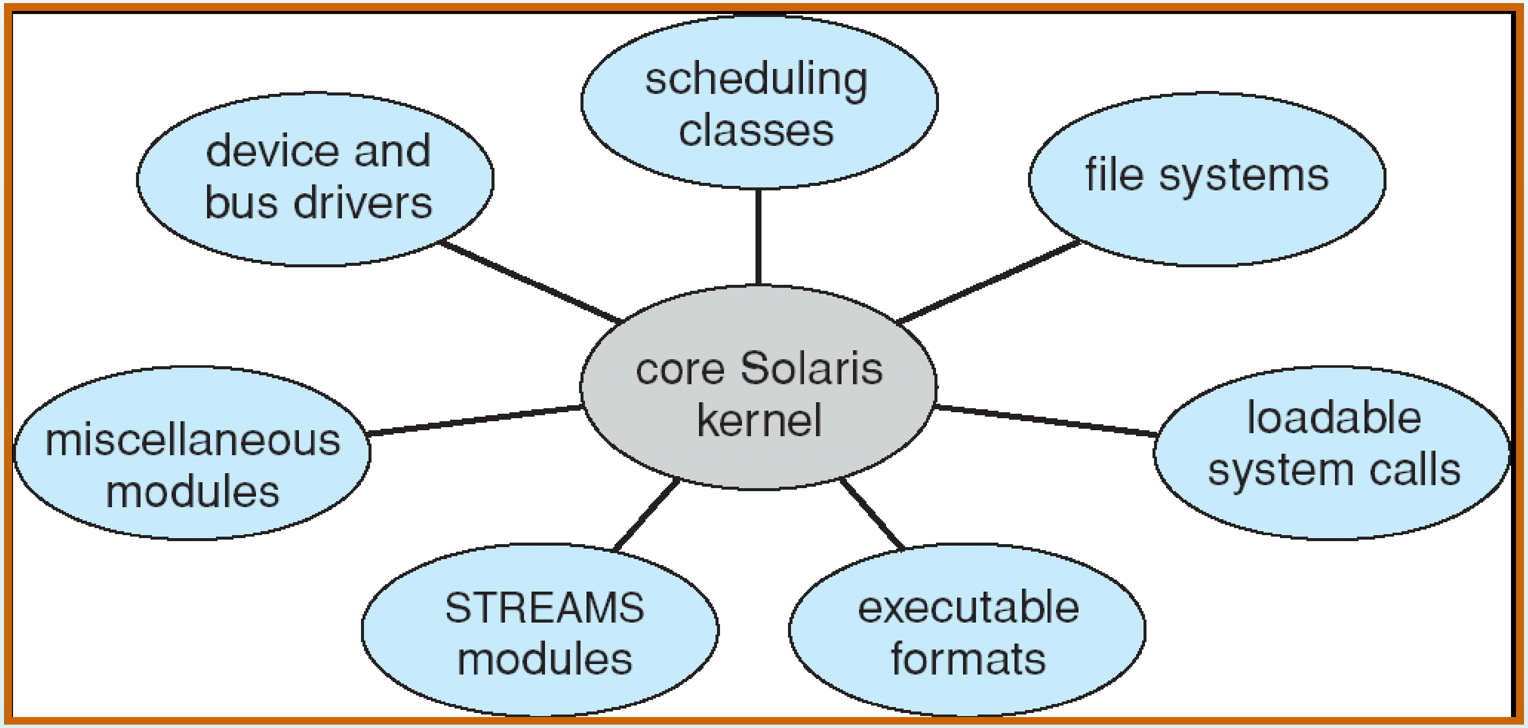


Shell
Make
. . .
User
Process
File
Process manager
Reincarnation server
. . .
Other
Servers
Disk
TTY
Ether
Print
. . .
Other
Drivers
User mode
Kernel mode
Microkernel handles interrupts, processes, scheduling, IPC
Clock
System

Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Communicate between user modules by message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More safe
- Detriments:
  - Performance overhead of user space to kernel space communication

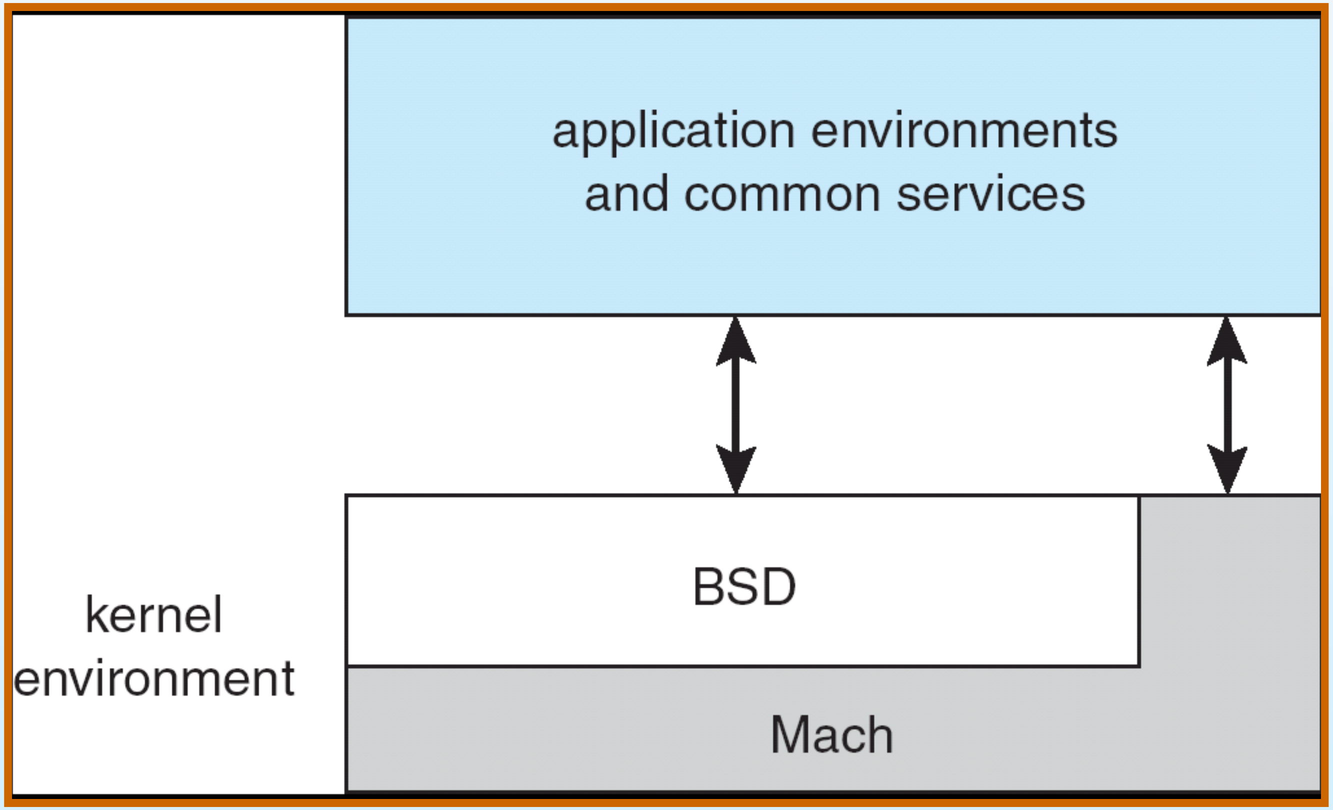
Solaris Modular Approach



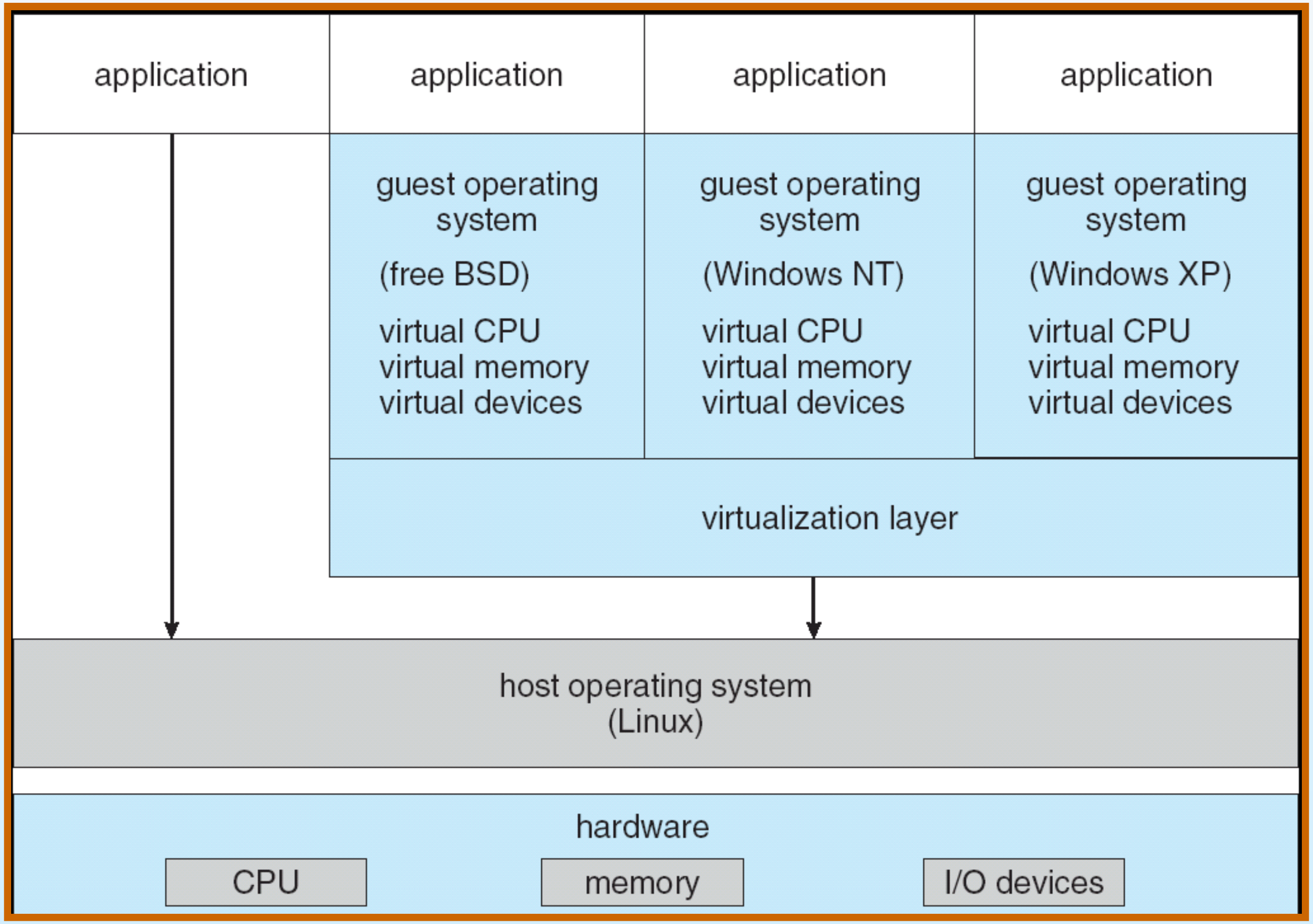
Modules

- Most modern OS 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

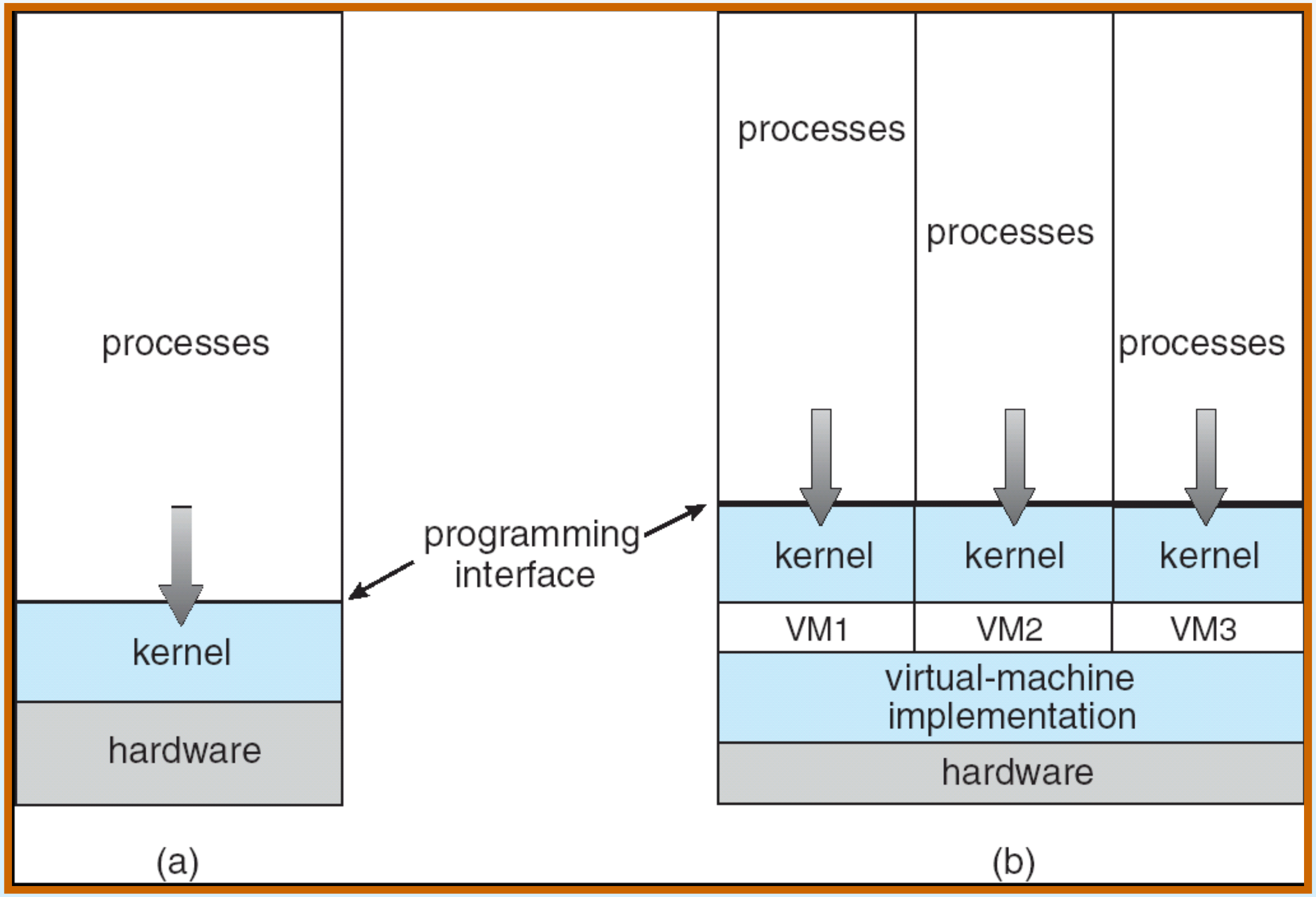
Mac OS X Structure



VMware Architecture



Virtual Machines



(a) Nonvirtual machine (b) virtual machine

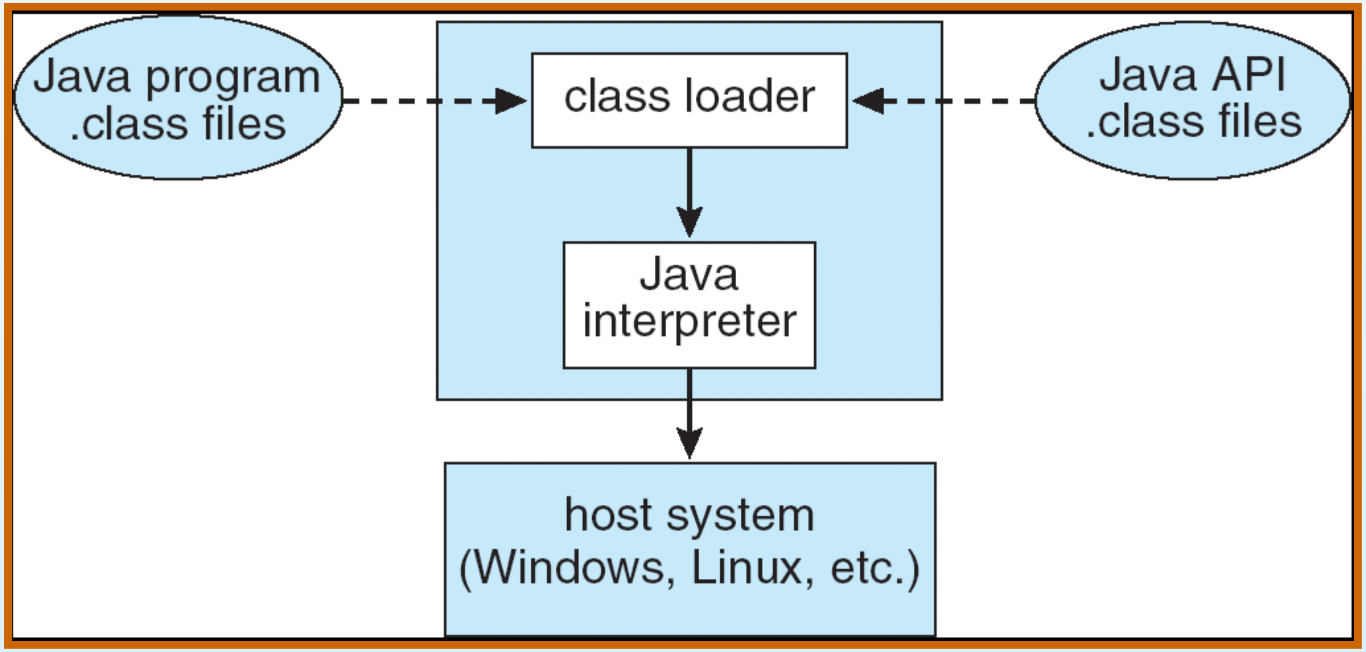
Virtual Machines (Cont.)

- A ***virtual machine*** **takes the layered approach to its logical conclusion.**
- **A VM provides an interface *identical* to the underlying bare hardware**
- **Each OS executes on its own processor with its own memory**

Virtual Machines (Cont.)

- The VM provides complete protection of resources
  - Each virtual machine is isolated from all other virtual machines.
- A VM system is a perfect vehicle for OS research and development.
  - System development is done on the VM, instead of on a physical machine
  - Does not disrupt normal system operation.
- It is difficult to implement
  - Required to provide an *exact* duplicate to the underlying machine

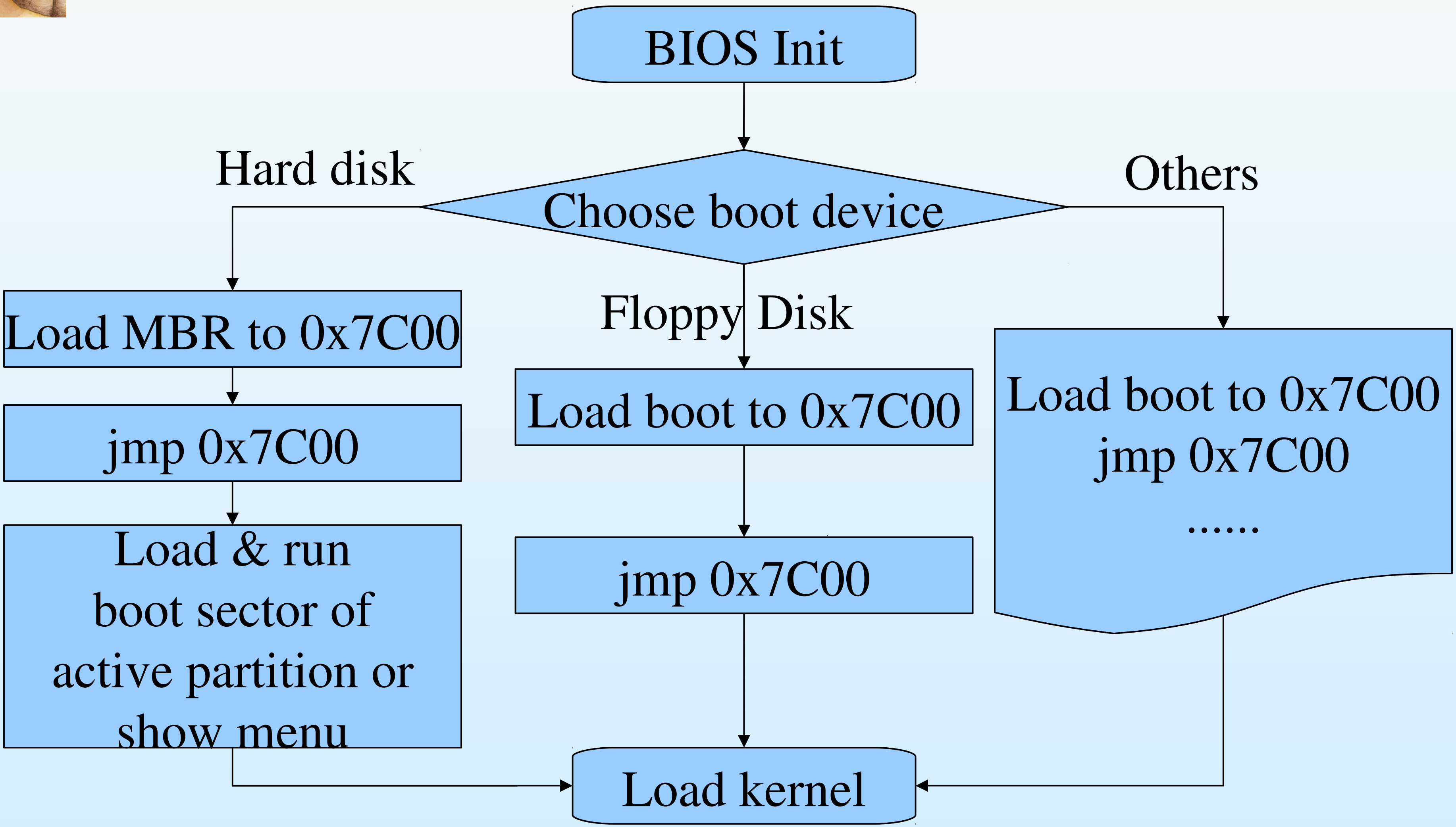
The Java Virtual Machine



System Boot

- *Booting*
  - starting a computer by loading the kernel
- *Bootstrap program*
  - Code stored in ROM that locates the kernel, load it into memory, and start its execution
- OS 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
  - Store entire OS in ROM and run it directly

X86 PC Boot


BIOS Init
Choose boot device
Hard disk
Floppy Disk
Others
Load MBR to 0x7C00
jmp 0x7C00
Load & run boot sector of active partition or show menu
Load boot to 0x7C00
jmp 0x7C00
Load boot to 0x7C00
jmp 0x7C00
……
Load kernel

End of Chapter 2