Operating System Structures

Zhi Wang
Florida State University

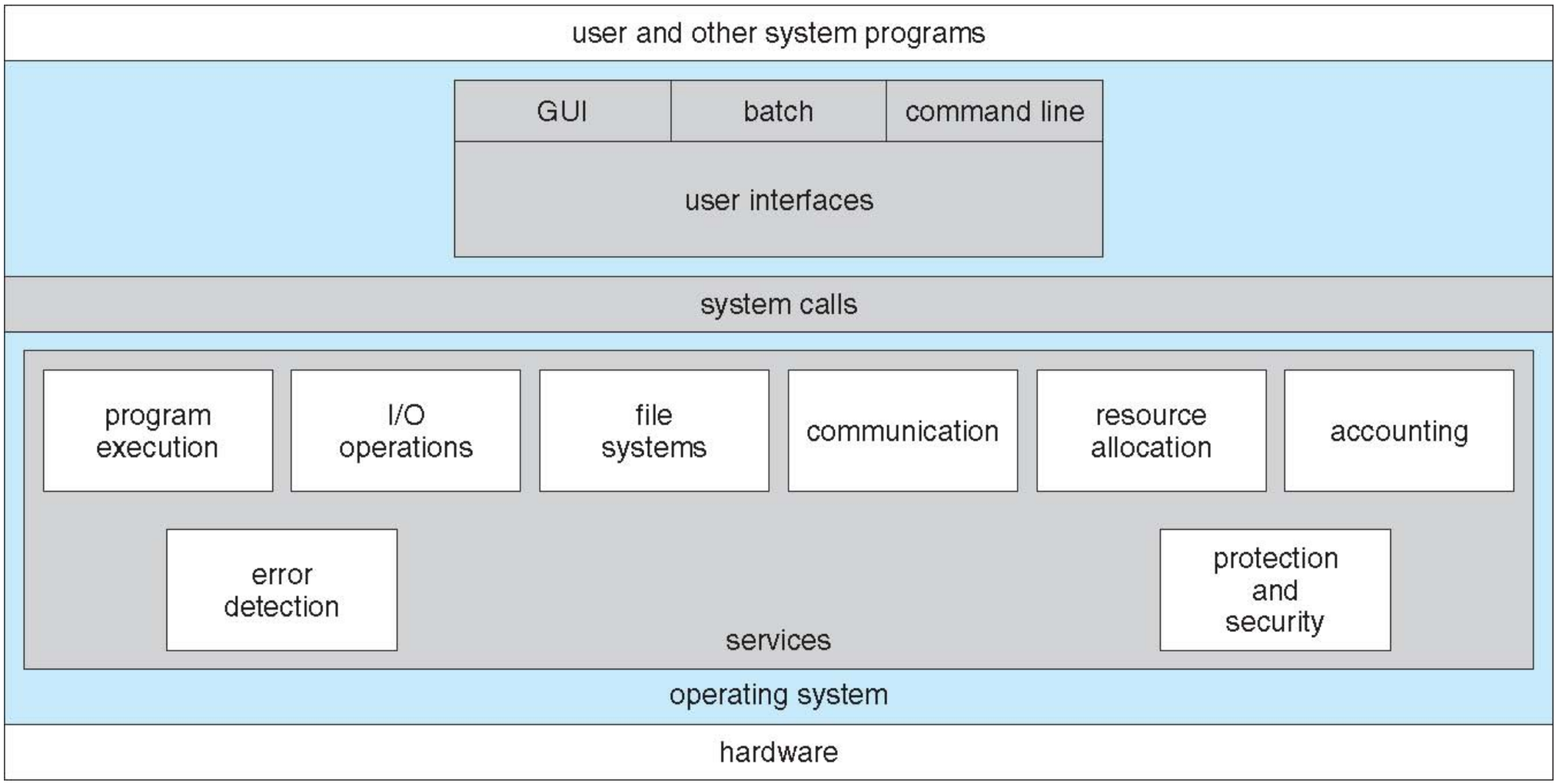
Content

- Operating system services
- User interface
- System calls
- Operating system structures
- Virtual machines

Operating System Services

- Operating systems provides an environment for program execution and services to programs and users
  - a set of services is helpful to (**visible to**) **users**:
    - user interface
    - program execution
    - I/O operation
    - file-system manipulation
    - communication
    - error detection
  - another set of services exists for ensuring **efficient operation of the system**:
    - resource allocation
    - accounting
    - protection and security

A View of Operating System Services



Operating System Services (User-Visible)

- **User interface**
  - most operating systems have a user interface (UI).
  - e.g., command-Line (CLI), graphics user interface (GUI), or batch
- **Program execution**
  - load and execute an program in the memory
  - end execution, either normally or abnormally
- **I/O operations**
  - a running program may require I/O such as file or I/O device
- **File-system manipulation**
  - read, write, create and delete files and directories
  - search or list files and directories
  - permission management

Operating System Services (User-Visible)

- **Communications**
  - processes exchange information, on the same system or over a network
  - via shared memory or through message passing
- **Error detection**
  - OS needs to be constantly aware of possible errors
  - errors in CPU, memory, I/O devices, programs
  - it should take appropriate actions to ensure correctness and consistency

Operating System Services (System)

- **Resource allocation**
  - allocate resources for multiple users or multiple jobs running concurrently
  - many types of resources: CPU, memory, file, I/O devices
- **Accounting**
  - to keep track of which users use how much and what kinds of resources
- **Protection and security**
  - protection provides a mechanism to control access to system resources
    - access control: control access to resources
    - isolation: processes should not interfere with each other
  - security authenticates users and prevent invalid access to I/O devices
    - a chain is only as strong as its weakest link
  - protection is the **mechanism**, security towards the **policy**

System Programs

- System programs provide a convenient environment for program development and execution
- They can be divided into:
  - file operations
  - status information
  - programming language support
  - program loading and execution
  - communications
- Most users' view of OS is defined by system programs, not the actual system calls

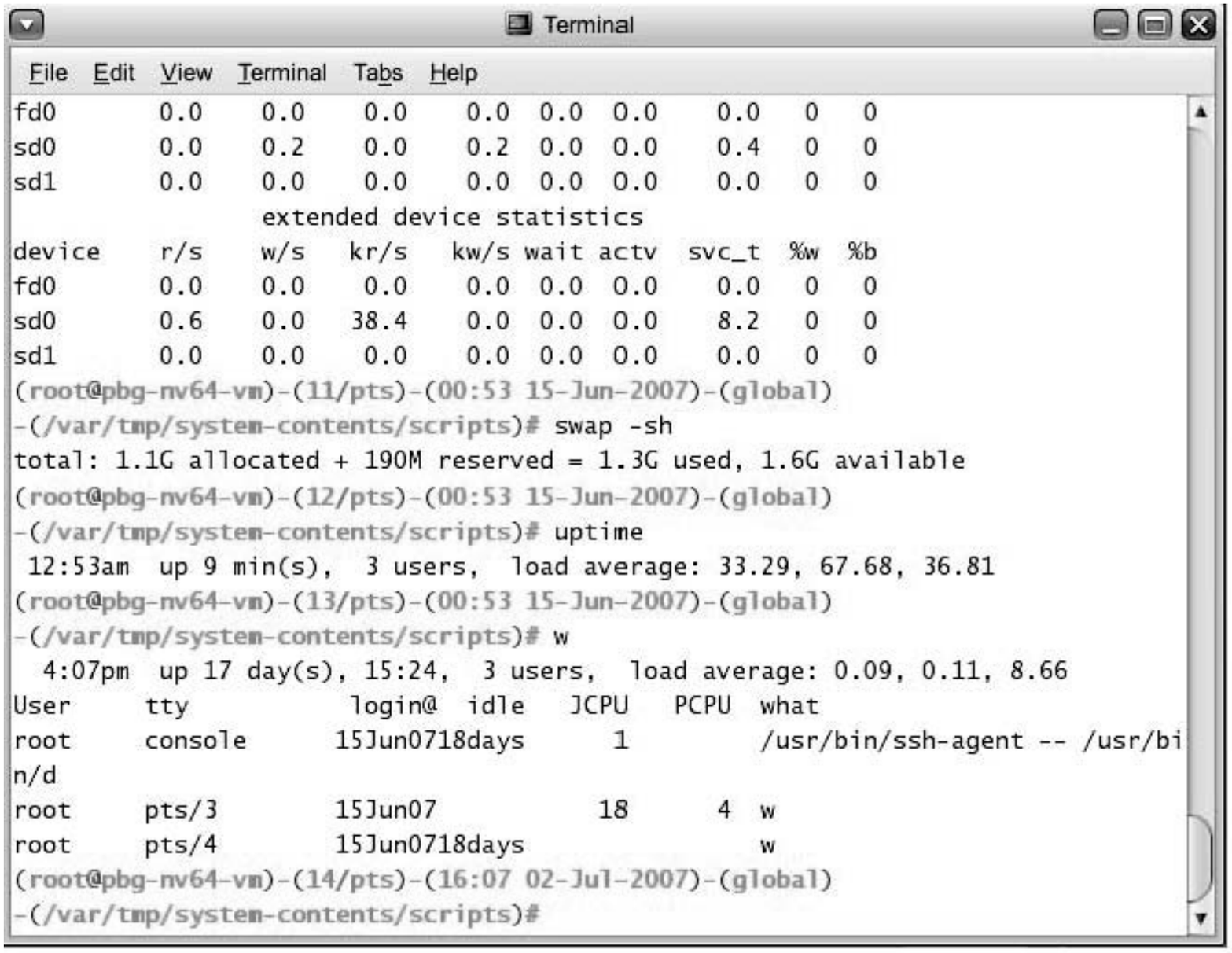
User Operating System Interface - CLI

- CLI (or command interpreter) allows direct command entry
  - a loop between fetching a command from user and executing it
- It can be implemented in the kernel or by a system program
  - In UNIX, it is usually called shells, there are many flavors of shells
- Commands are either built-in or just names of programs
  - if the latter, adding new features doesn’t require shell modification

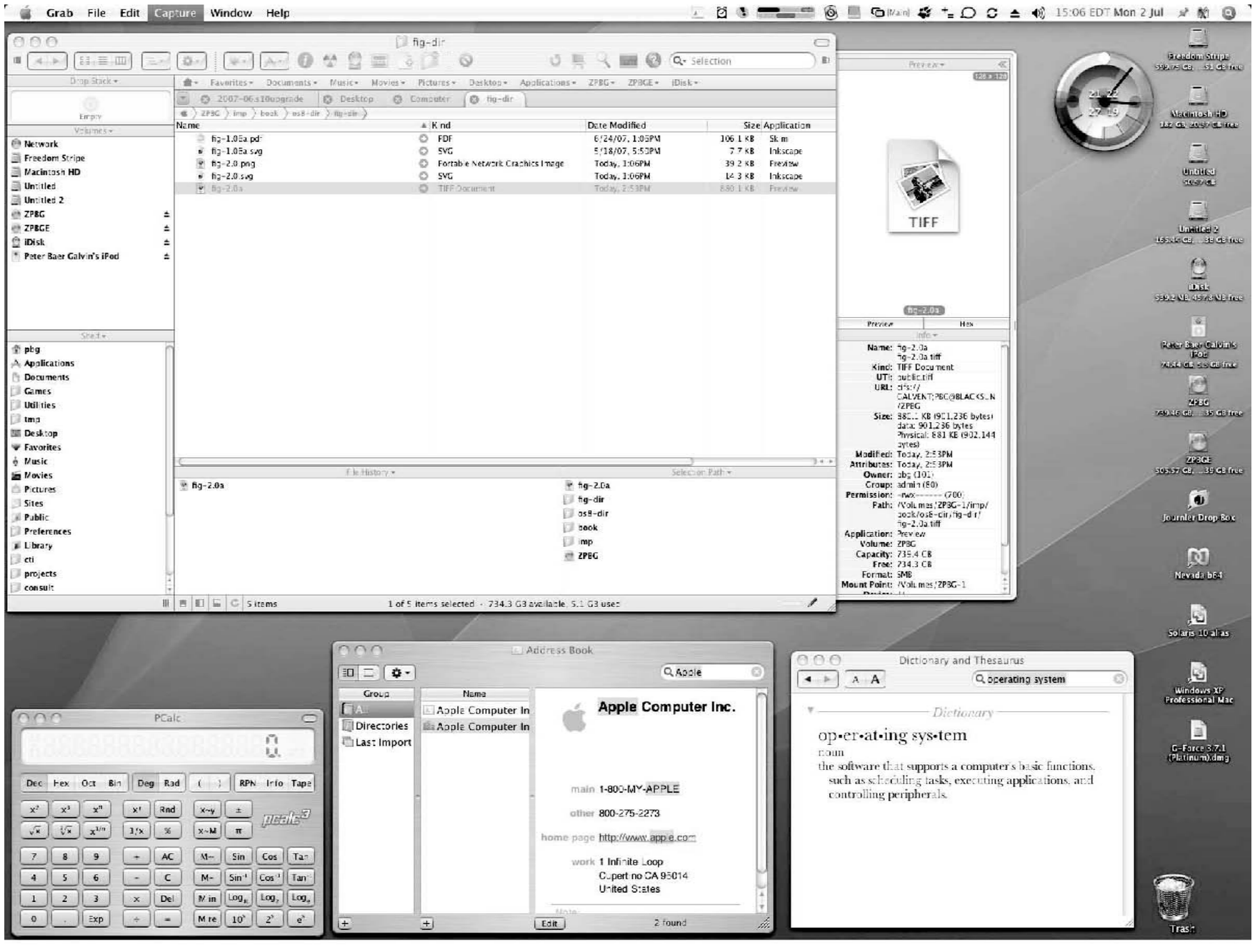
User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - users use mouse, keyboard, and monitor to interactive with the system
  - icons represent files, programs, actions, etc
  - mouse buttons over objects in the interface cause various actions
    - open file or directory (aka. folder), execute program, list attributes
  - invented at Xerox PARC
- Many systems include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI with UNIX kernel underneath
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Bourne Shell Command Interpreter



The Mac OS X GUI



System Calls

- System call is **a programming interface to access the OS services**
- Direct system call access usually requires to use assembly language
  - e.g., int 0x80 for Linux
- System call is typically wrapped in a high-level **Application Program Interface** (API)
  - three most common APIs:
    - **Win32** API for Windows
    - **POSIX** API for POSIX-based systems (UNIX/Linux, Mac OS X)
    - **Java** API for the Java virtual machine (JVM)
  - why use APIs rather than system calls?

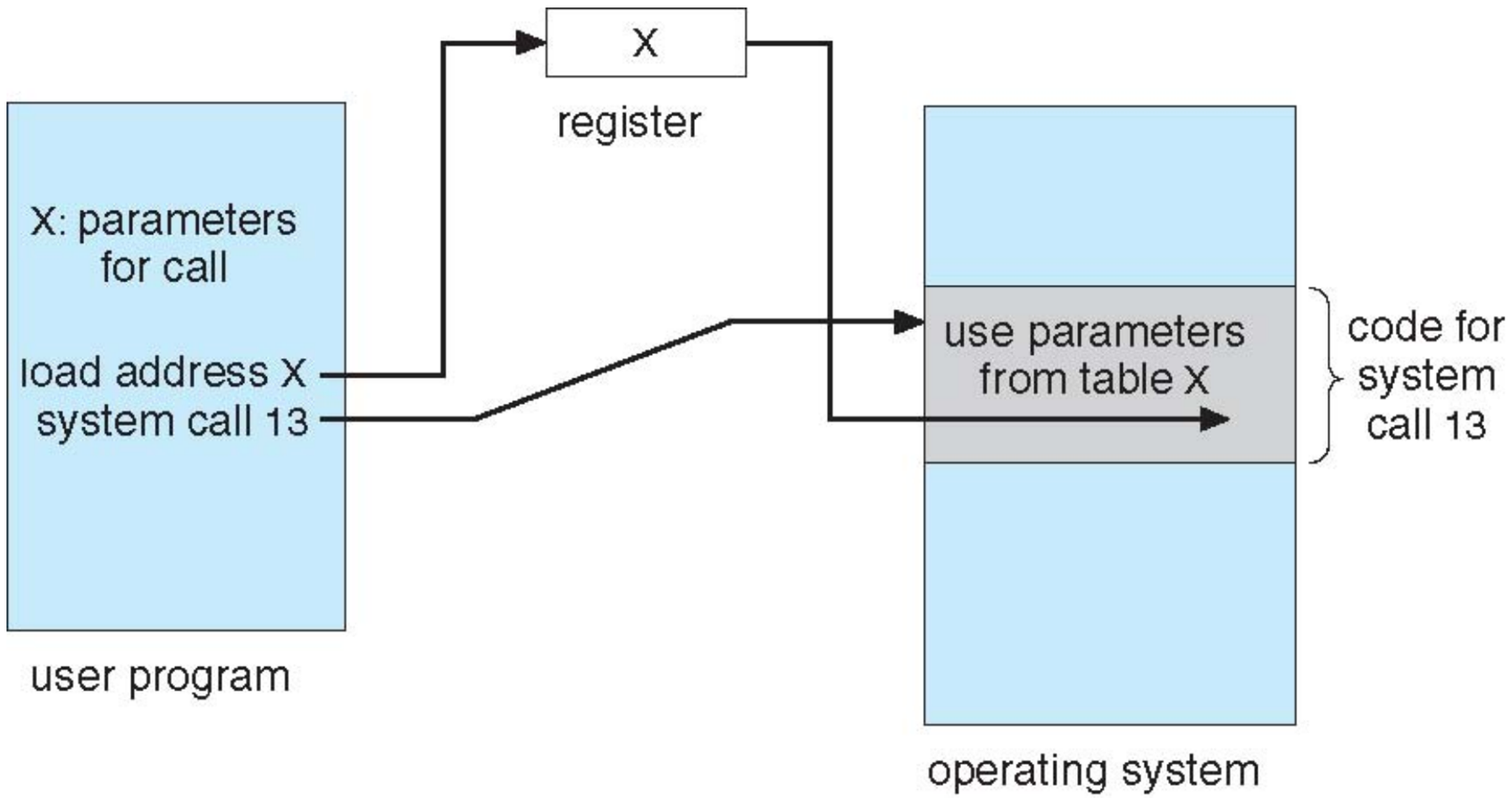
System Calls

- Typically, a **number** is associated with each system call
  - system-call interface maintains a table indexed by these numbers
  - e.g., Linux 3.2.35 for x86 has 349 system calls, number 0 to 348
- Kernel invokes intended system call and returns results
- User program needs to know nothing about syscall details
  - it just needs to use API and understand what the API will do
  - most details of OS interface hidden from programmers by the API

System Call Parameter Passing

- Parameters are required besides the **system call number**
  - exact type and amount of information vary according to OS and call
- Three general methods to pass parameters to the OS
  - **Register**:
    - pass the parameters in registers
    - simple, but there may be more parameters than registers
  - **Block**:
    - parameters stored in a memory block (or table)
    - address of the block passed as a parameter in a register
    - taken by Linux and Solaris
  - **Stack**:
    - parameters placed, or pushed, onto the stack by the program
    - popped off the stack by the operating system
- Block and stack methods don’t limit number of parameters being passed

Parameter Passing via Block



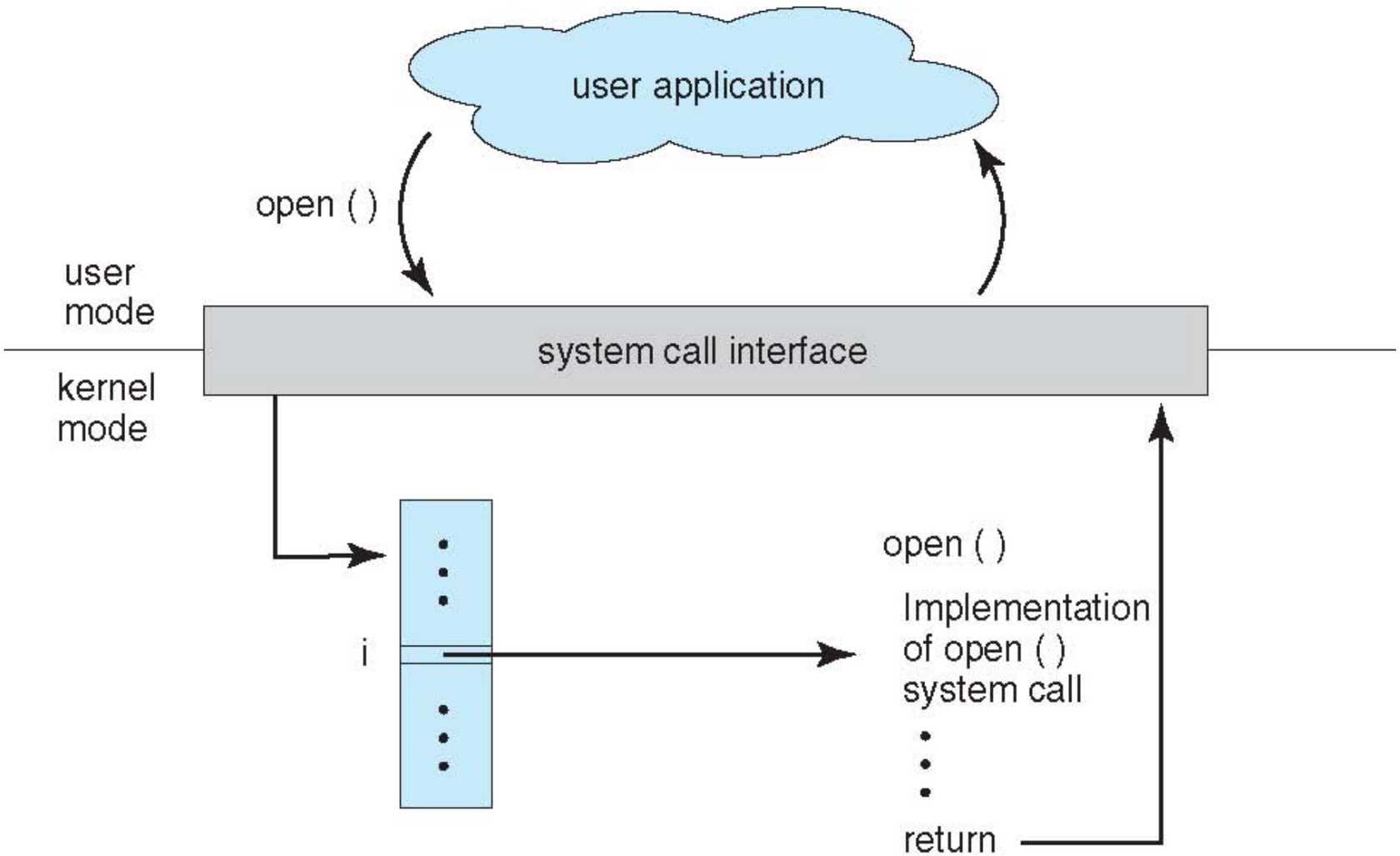
Execve System Call on Linux

- Store syscall number in eax
- Save arg 1 in ebx, arg 2 in ecx, arg 3 in edx
- Execute int 0x80 (or sysenter)
- Syscall runs and returns the result in eax

execve (“/bin/sh”, 0, 0)

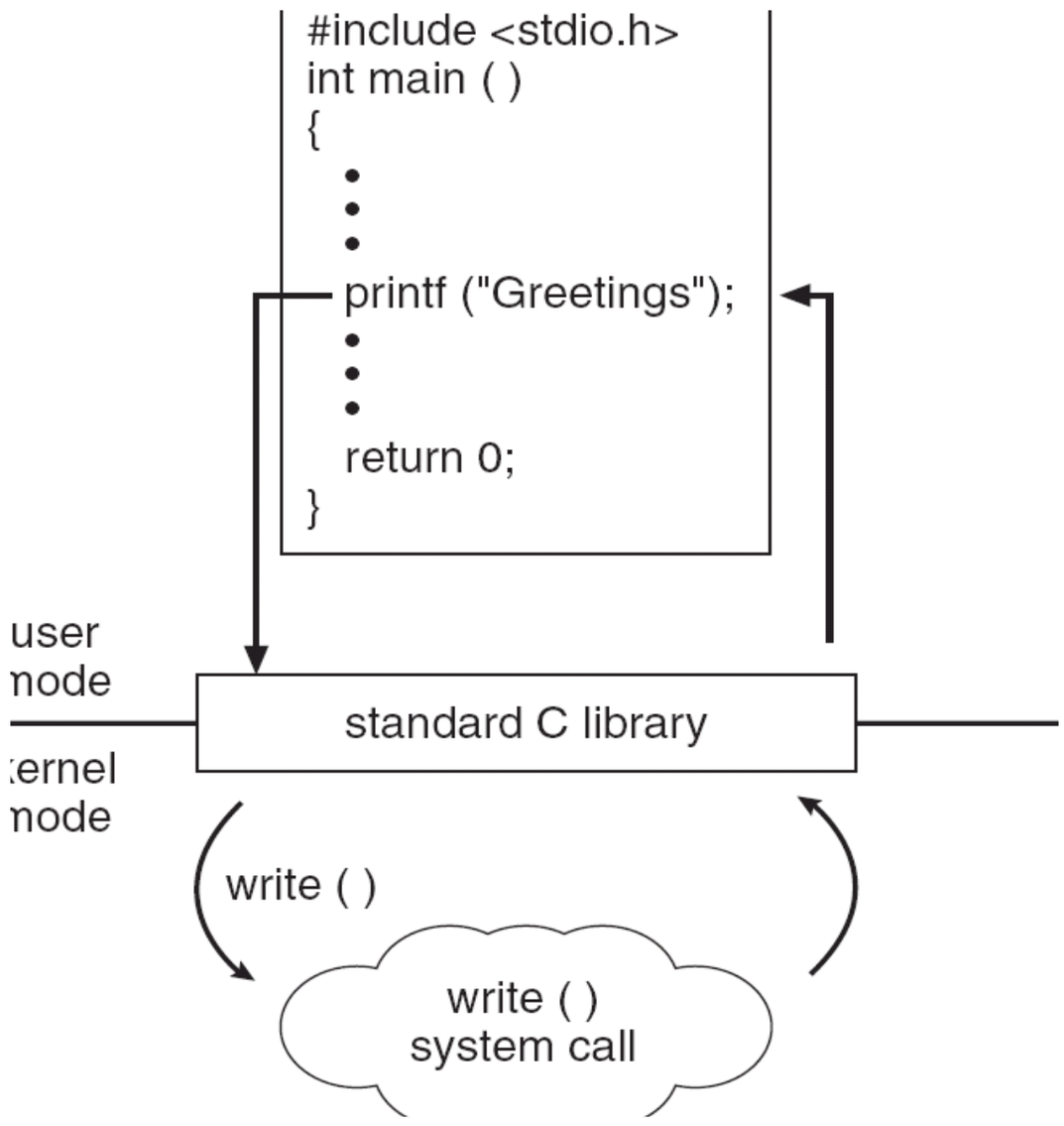
eax: 0x0b
ebx: addr of “/bin/sh”
ecx: 0

API – System Call – OS Relationship



Standard C Library Example

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



Types of System Calls

- **Process control**
  - create process, terminate process
  - load, execute, end, abort
  - get process attributes, set process attributes
  - wait for timer or event, signal event
  - allocate and free memory
- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls

- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- **Information maintenance**
  - get/set time or date
  - get/set system data
  - get/set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

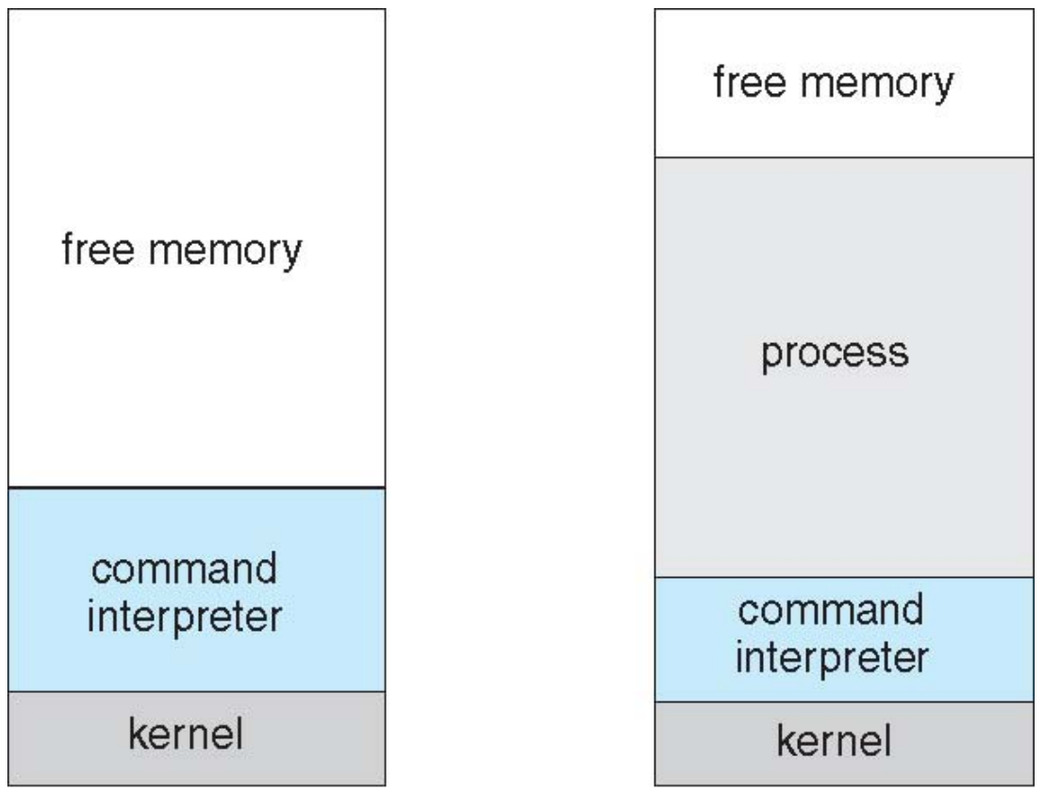
Windows and Unix System Calls

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

Example: MS-DOS

- 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

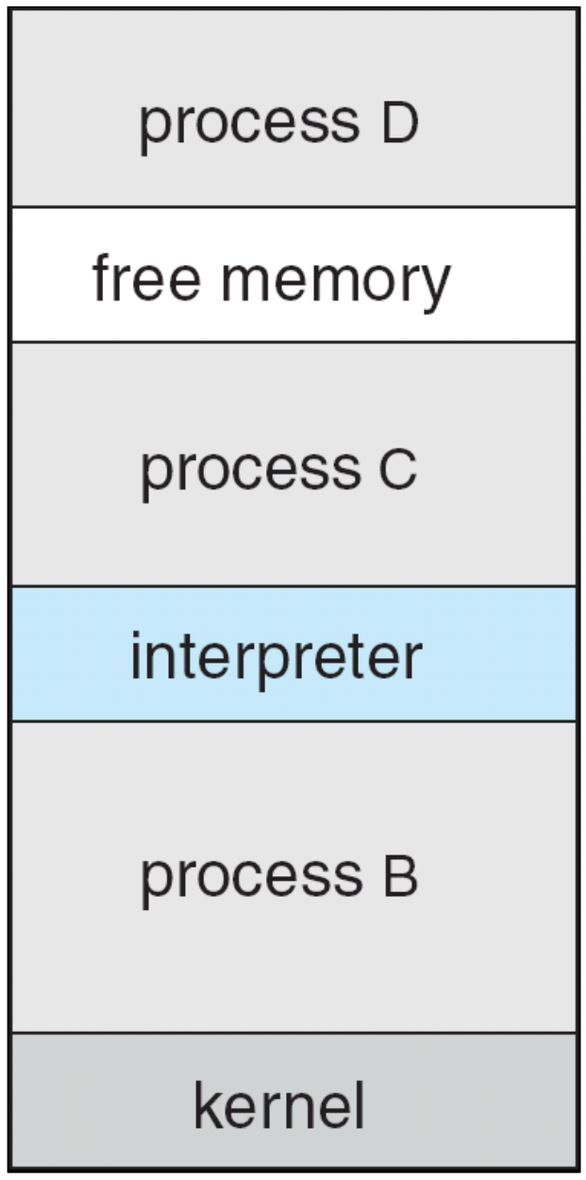
MS-DOS Execution



Example: FreeBSD

- A variant of Unix, it supports multitasking
- Upon user login, the OS invokes user's choice of shell
- Shell executes **fork**() system call to create process, then calls **exec**() to load program into process
  - shell waits for process to terminate or continues with user commands

FreeBSD Running Multiple Programs



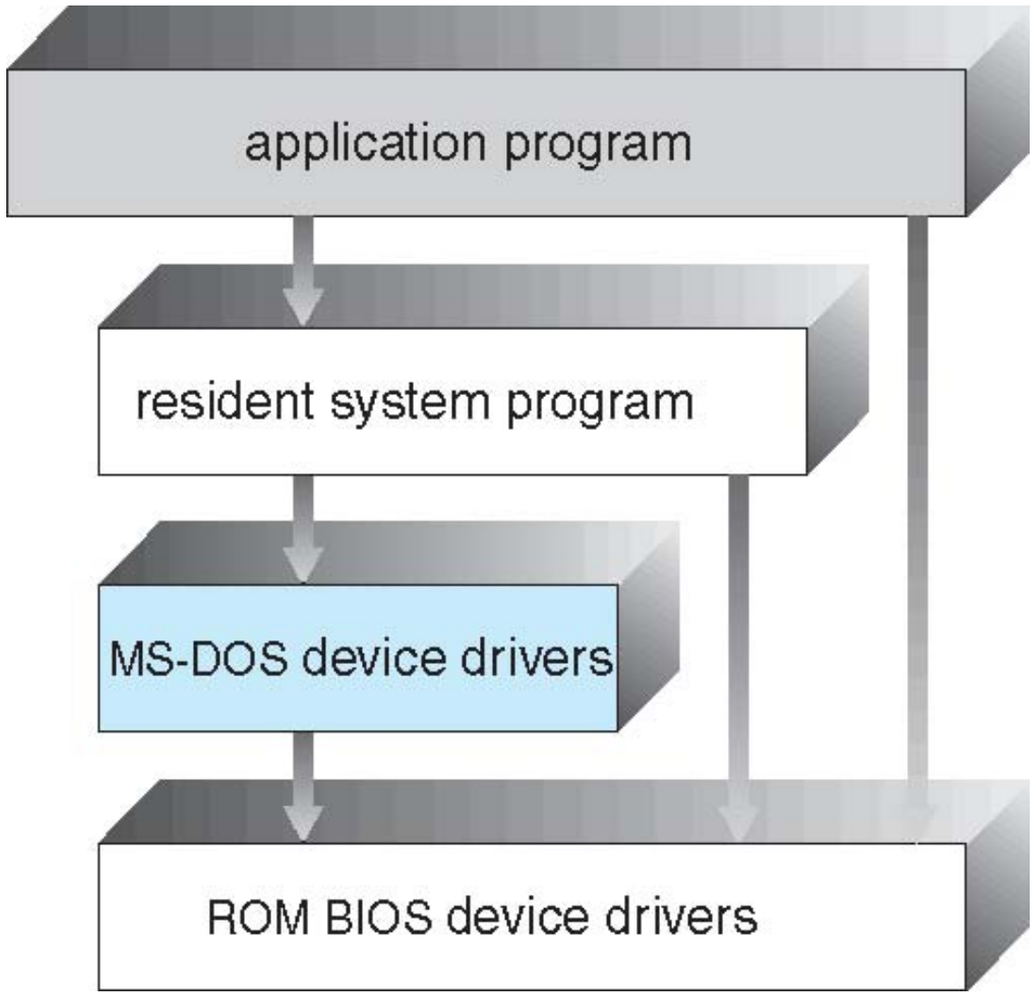
Operating System Structure

- Important principle: to separate mechanism and policy
  - **mechanism**: how to do it
  - **policy**: what will be done
- Many structures:
  - **simple structure**
  - **layered structure**
  - **modules**
  - **microkernel system structure**
  - research system: **exo-kernel**, **multi-kernel**…

Simple Structure

- No structure at all!
  - written to provide the most functionality in the least space
- A typical example: MS-DOS
  - its interfaces and levels of functionality are not well separated
  - the kernel is not divided into modules

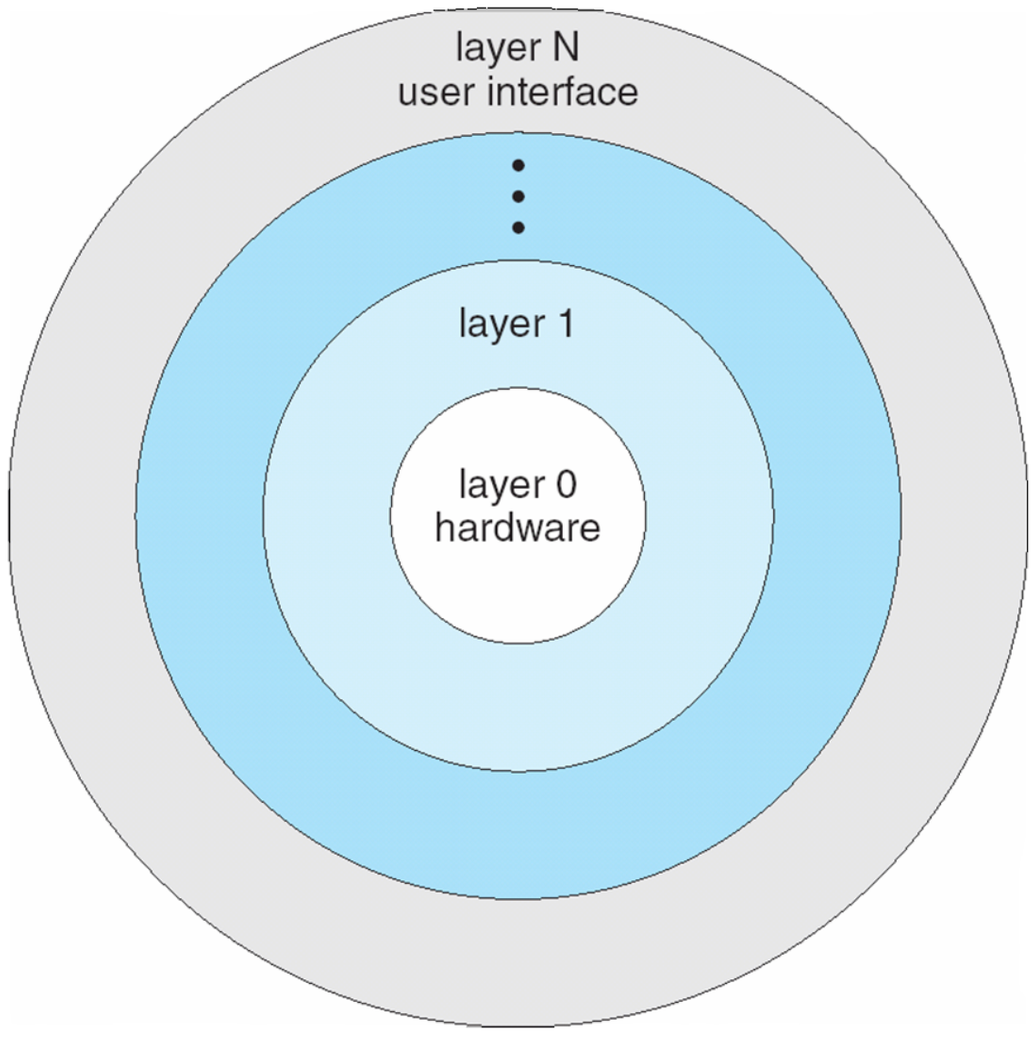
MS-DOS Structure



Layered Approach

- The OS is divided into a number of layers (levels)
- Each layer built on top of lower layers
  - is this strictly enforceable in the kernel?
- The bottom layer (layer 0), is the hardware; the highest (layer N) is UI

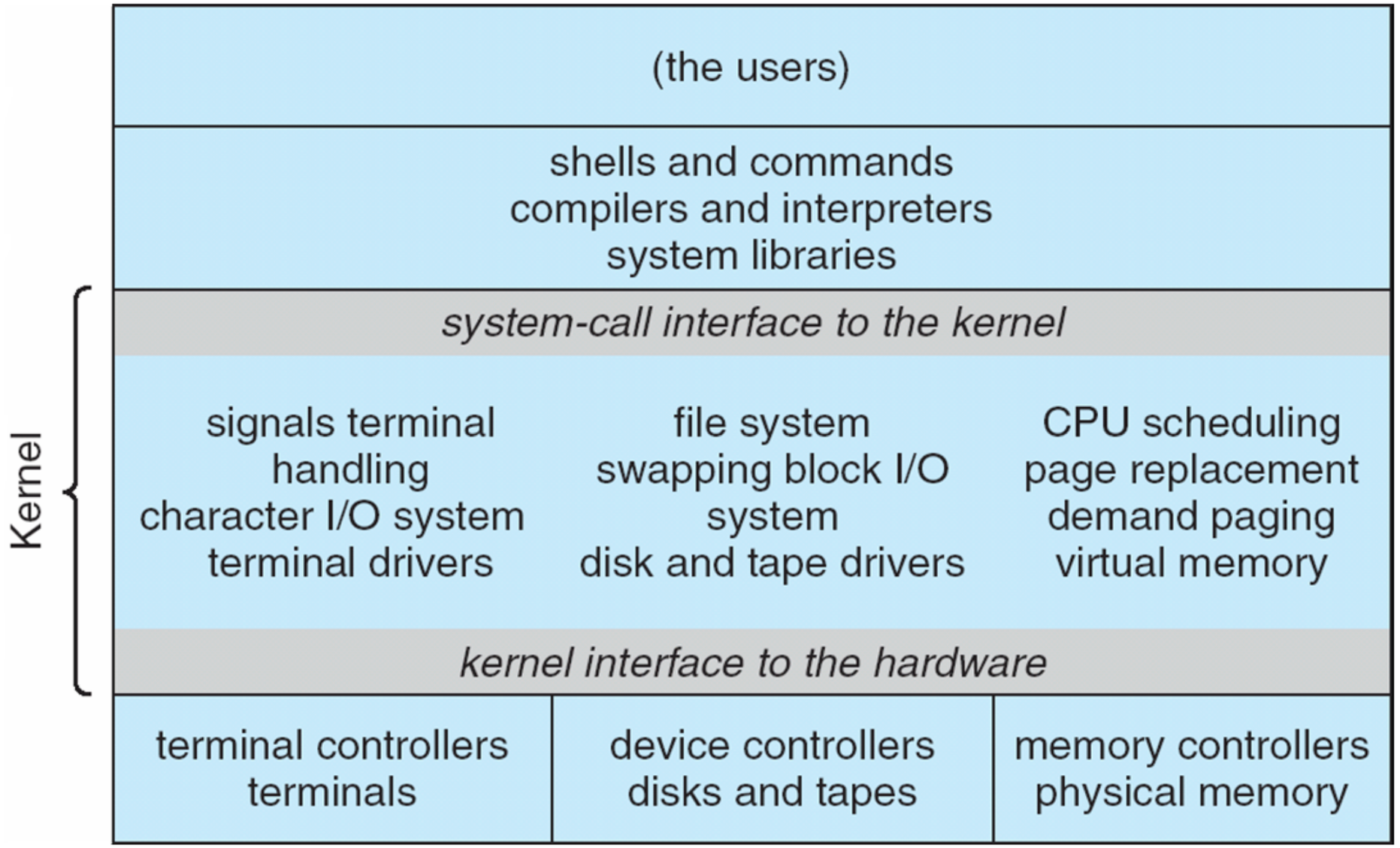
Layered Operating System



UNIX

- Limited by hardware functionality, the original UNIX had limited structure
- UNIX OS consists of two separable layers
  - systems programs
  - the kernel: everything below the system-call interface and above physical hardware
    - a large number of functions for one level: file systems, CPU scheduling, memory management …
- Interdependency of components makes it impossible to structure kernel strictly in layers
  - Example: memory manage and storage

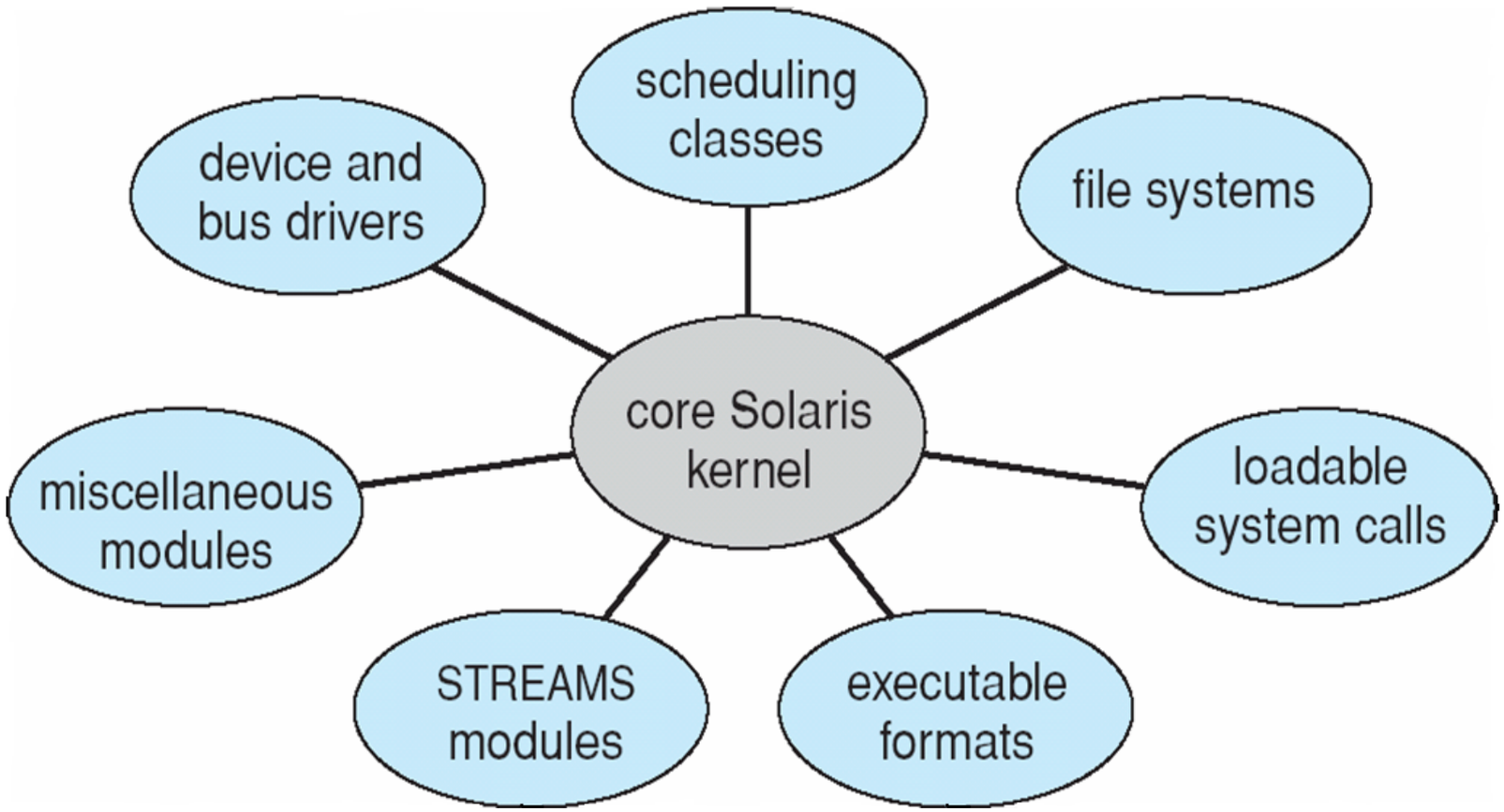
Traditional UNIX System Structure



Modules

- Most modern operating systems implement **kernel modules**
  - uses **object-oriented** design pattern
  - each core component is separate, and has **clearly defined interfaces**
  - some are loadable as needed
- Overall, similar to layers but with more flexible
- Example: Linux, BSD, Solaris
  - http://www.makelinux.net/kernel_map/

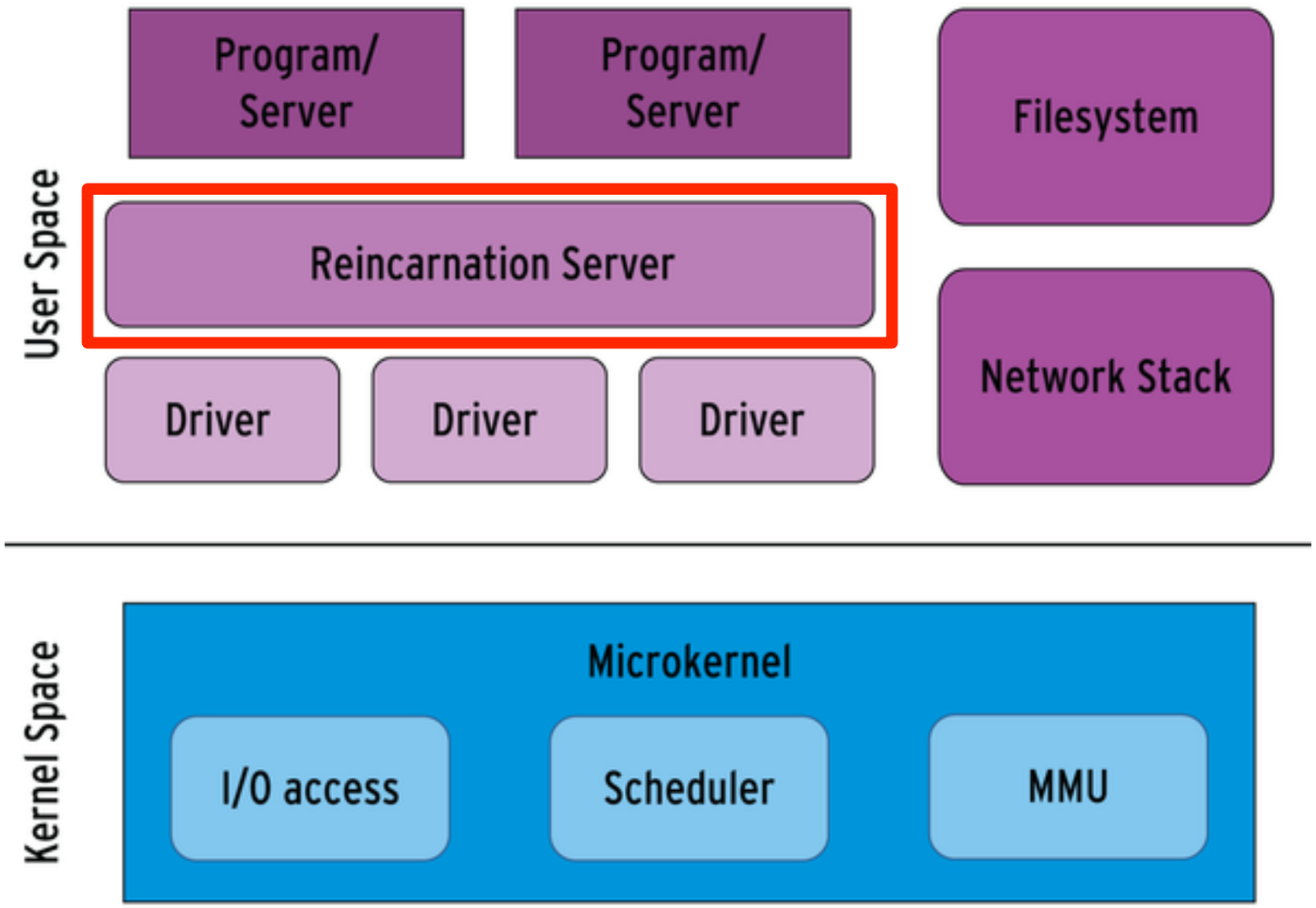
Solaris Modular Approach



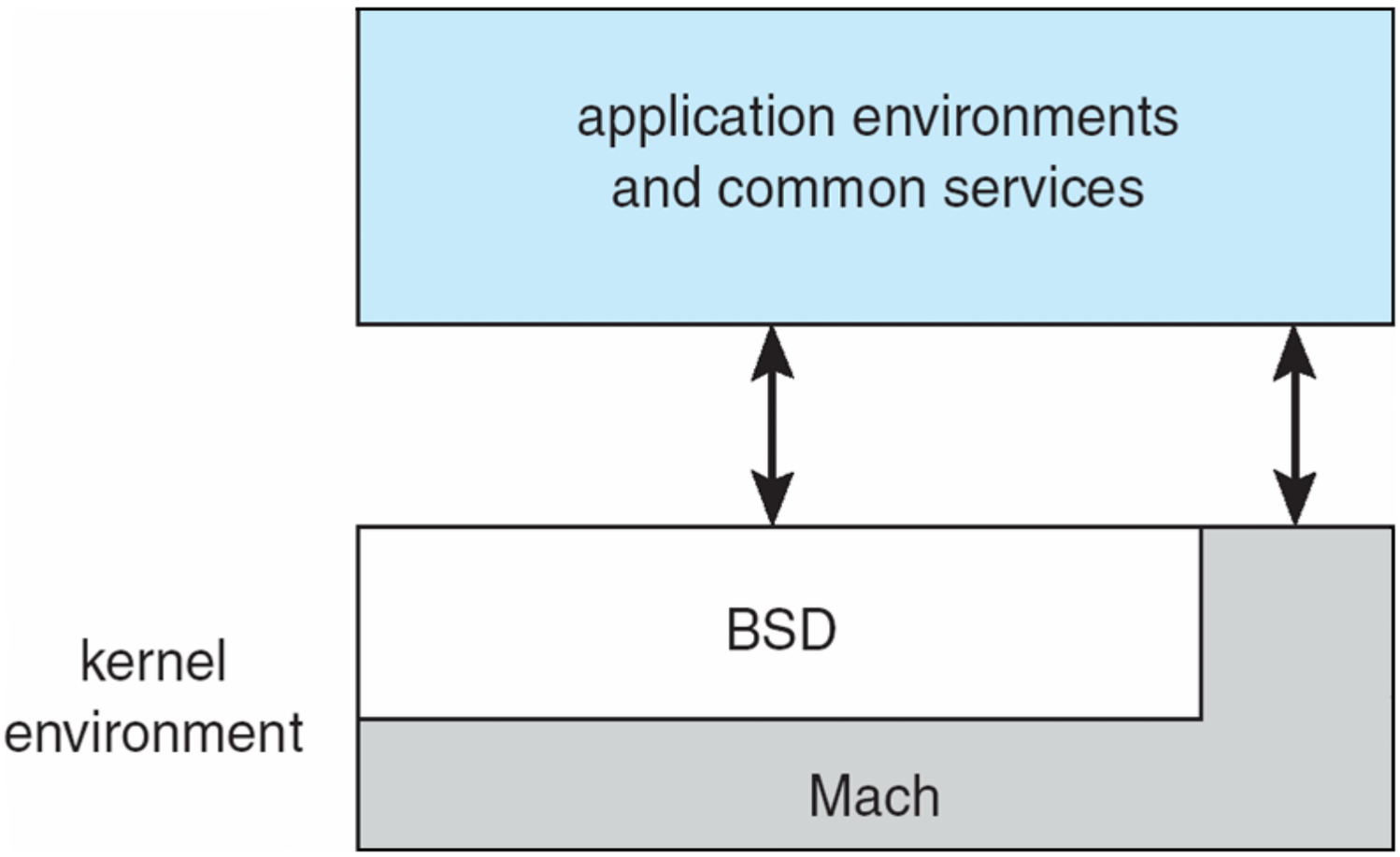
Microkernel System Structure

- Microkernel moves as much from the kernel (e.g., file systems) into “user” space
- Communication between user modules uses **message passing**
- Benefits:
  - easier to extend a microkernel
  - easier to port the operating system to new architectures
  - more reliable (less code is running in kernel mode)
  - more secure
- Detriments:
  - performance overhead of user space to kernel space communication
- Examples: Minux, Mach, QNX, L4…

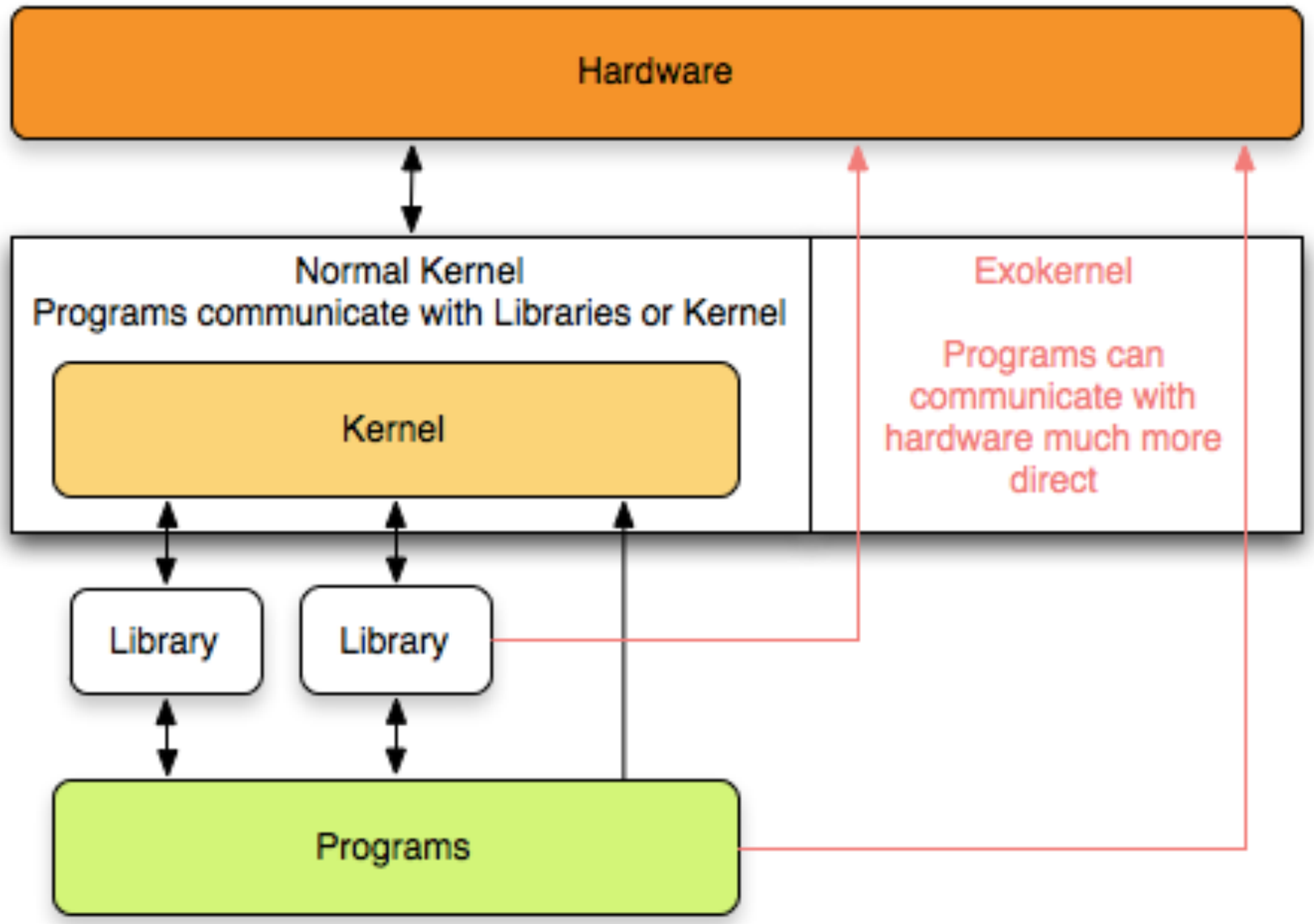
MINIX Layered Microkernel Architecture



Mac OS X Structure



Exokernel v.s. Normal Kernel



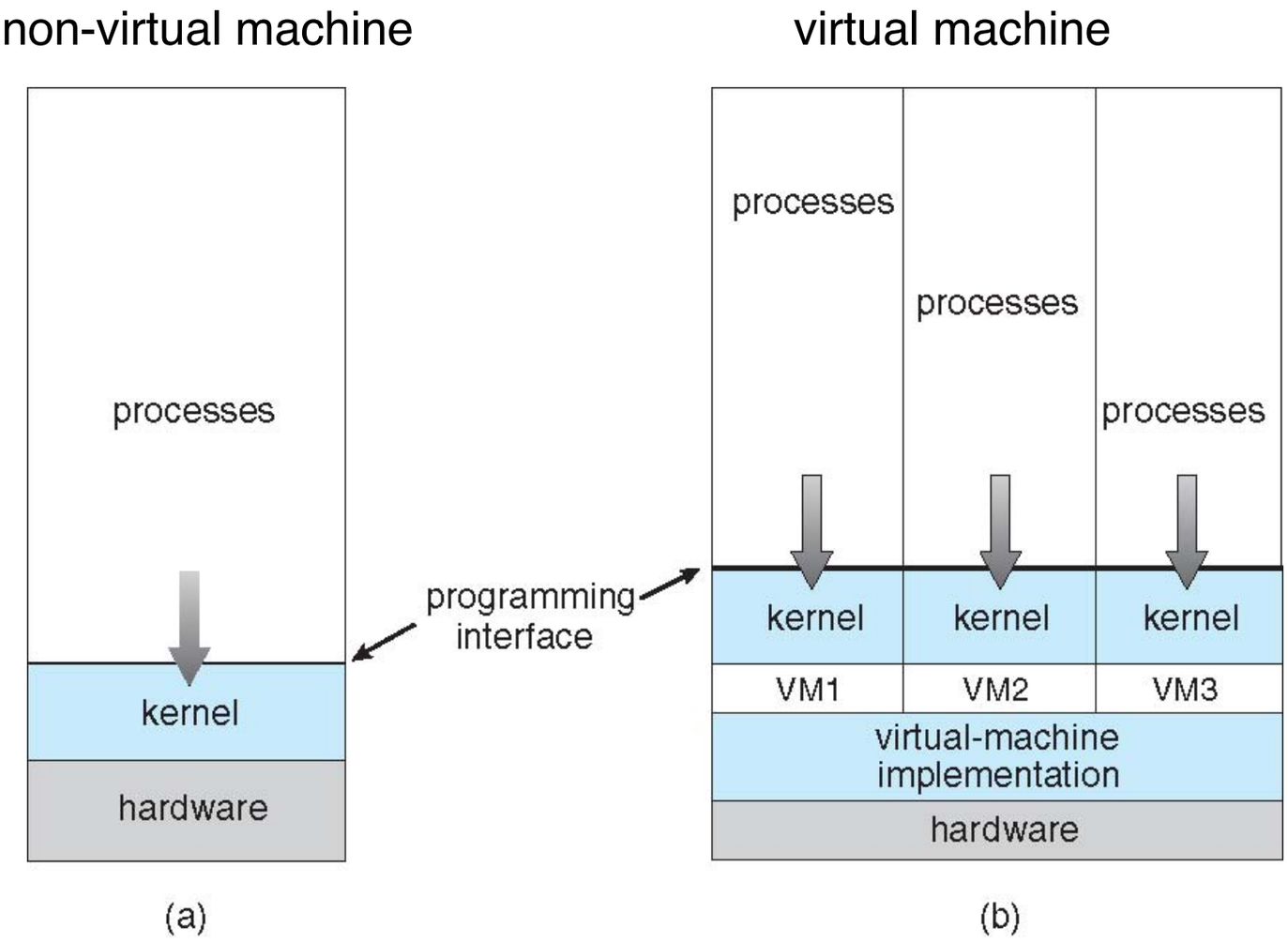
Virtual Machines

- A virtual machine takes **layered approach** to its logical conclusion
  - a virtual machine encapsulates the hardware and whole software stack
- VM provides an interface **identical to the underlying hardware**
  - Host creates the illusion that the guest has its own hardware
    - Each guest is provided with a (virtual) copy of underlying computer
- Example: VMware, VirtualBox, QEMU, KVM, Xen, Java, .Net

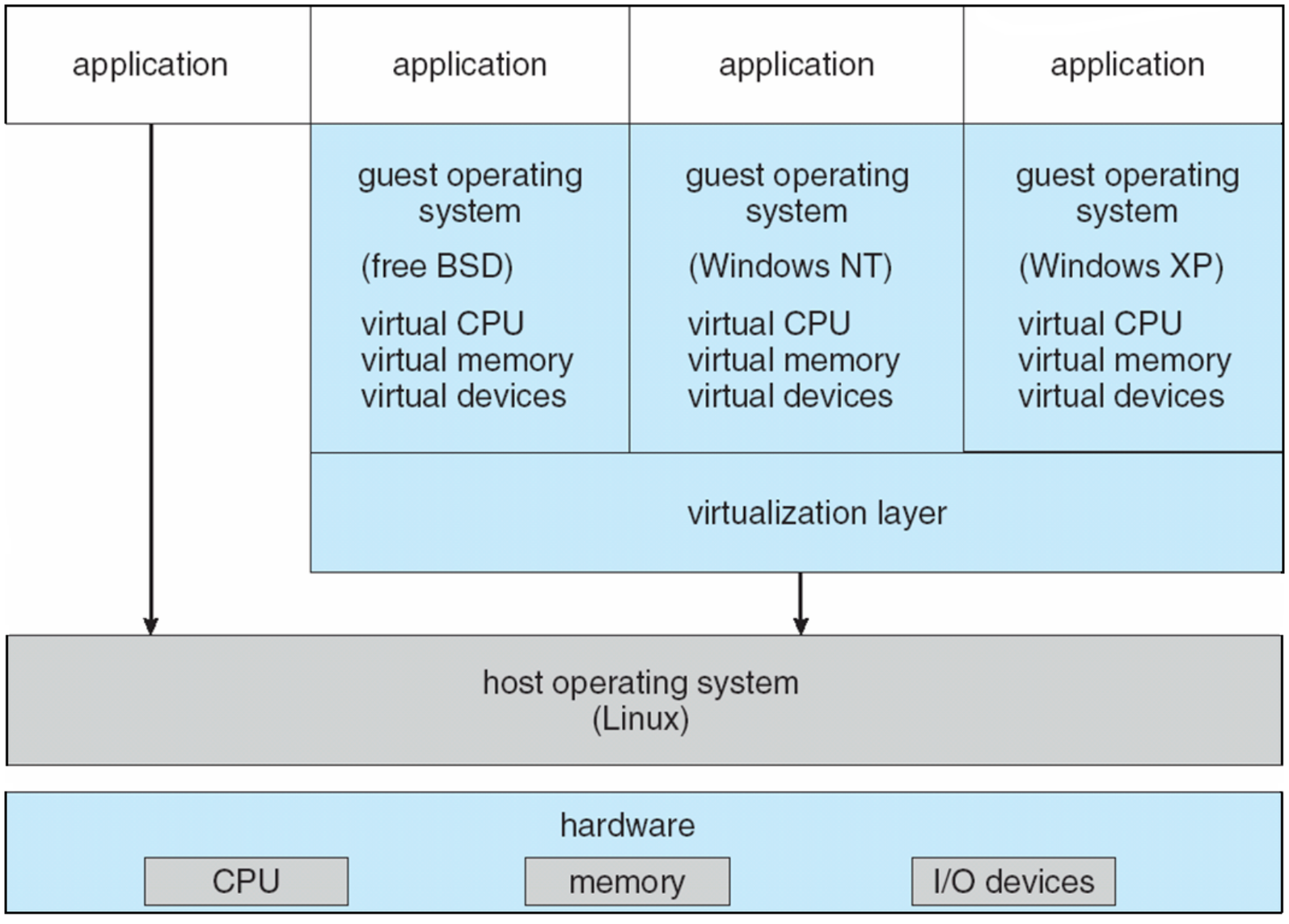
Virtual Machines

- First appeared commercially in IBM mainframes in **1972**
- Multiple (different) operating systems can share the same hardware
  - each VM is isolated from each other
  - sharing of resource can be permitted and controlled
  - commutate with each other and other physical systems via networking
- Benefit
  - consolidate low-resource use systems to fewer busier systems
  - strong isolation benefits security
  - useful for development, testing

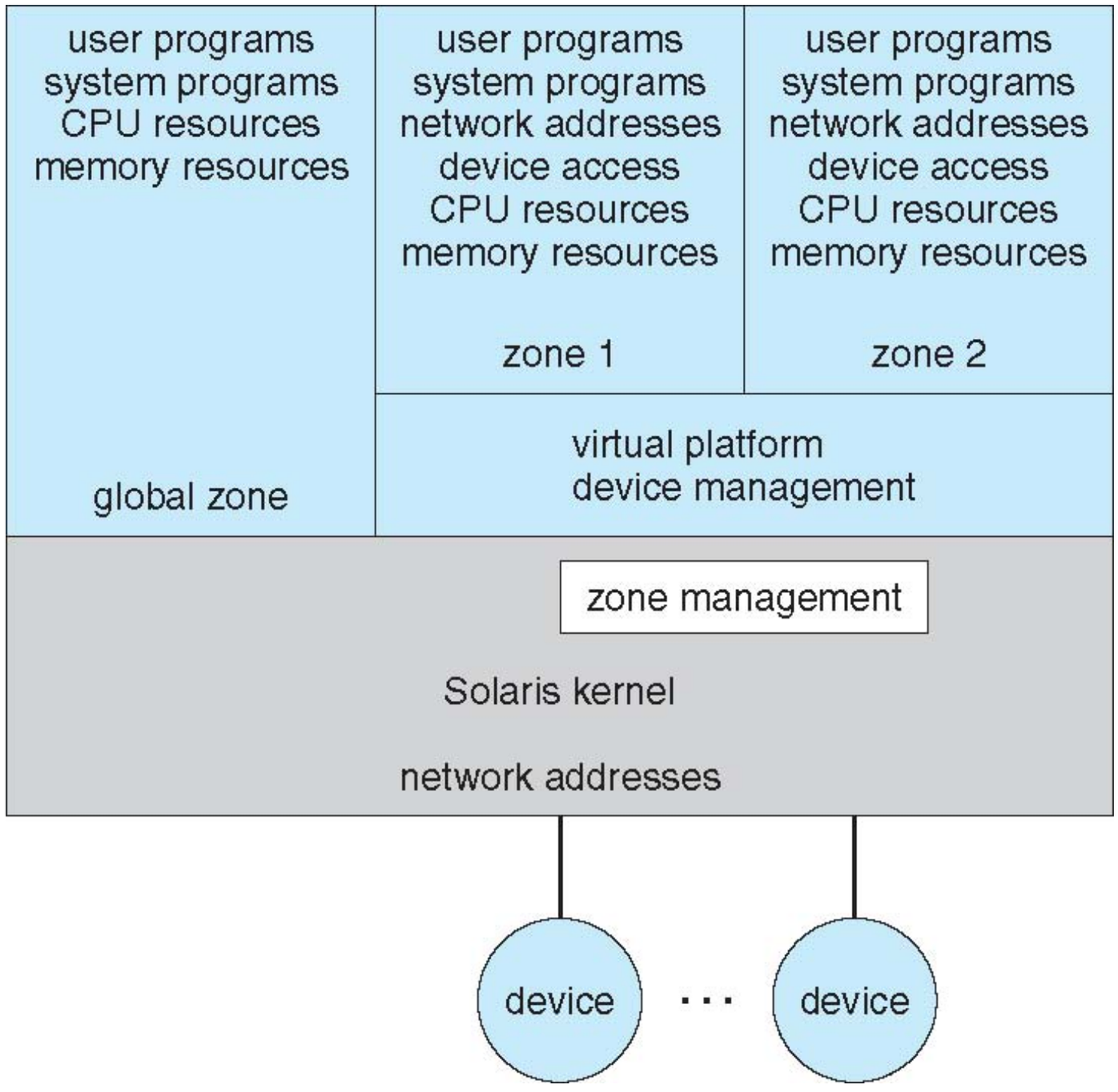
Virtual Machines



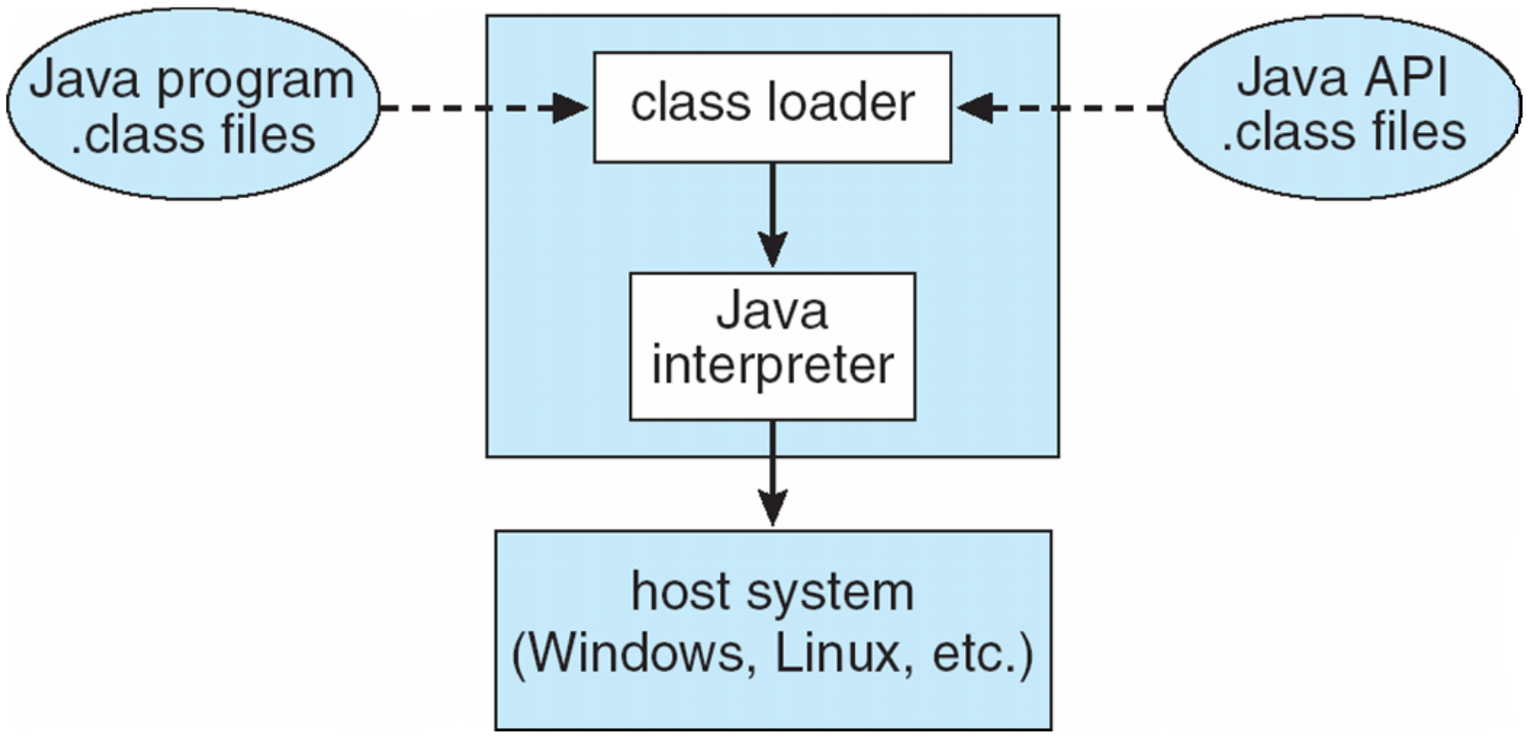
VMware Architecture



Solaris 10 with Two Containers



Java Virtual Machine



Operating-System Debugging

- Debugging is to find and fix errors, or bugs
  - OS generates log files containing error information
    - **dmesg** and **/var/log** in Linux
  - application failure can generate core dump file capturing process memory
  - OS failure can generate crash dump file containing kernel memory
    - security issues?

“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.”

– Kernighan’s Law

Solaris 10 dtrace Following System Call

- DTrace (SystemTap, Kprobes) allows live instrumentation of kernel
  - probes fire when code is executed, capturing state data and sending it to consumers of those probes

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   ->  XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable    U
  0     <- _X11TransSocketBytesreadable    U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```

End of Chapter 2