# Operating System Structures

Zhi Wang
Florida State University

# Content

- Operating system services

- User interface

- System calls

- System programs

- Operating system structures

- Virtual machines

# Objectives

- To describe services provided by an operating system

- To discuss various ways of structuring an operating system

# 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
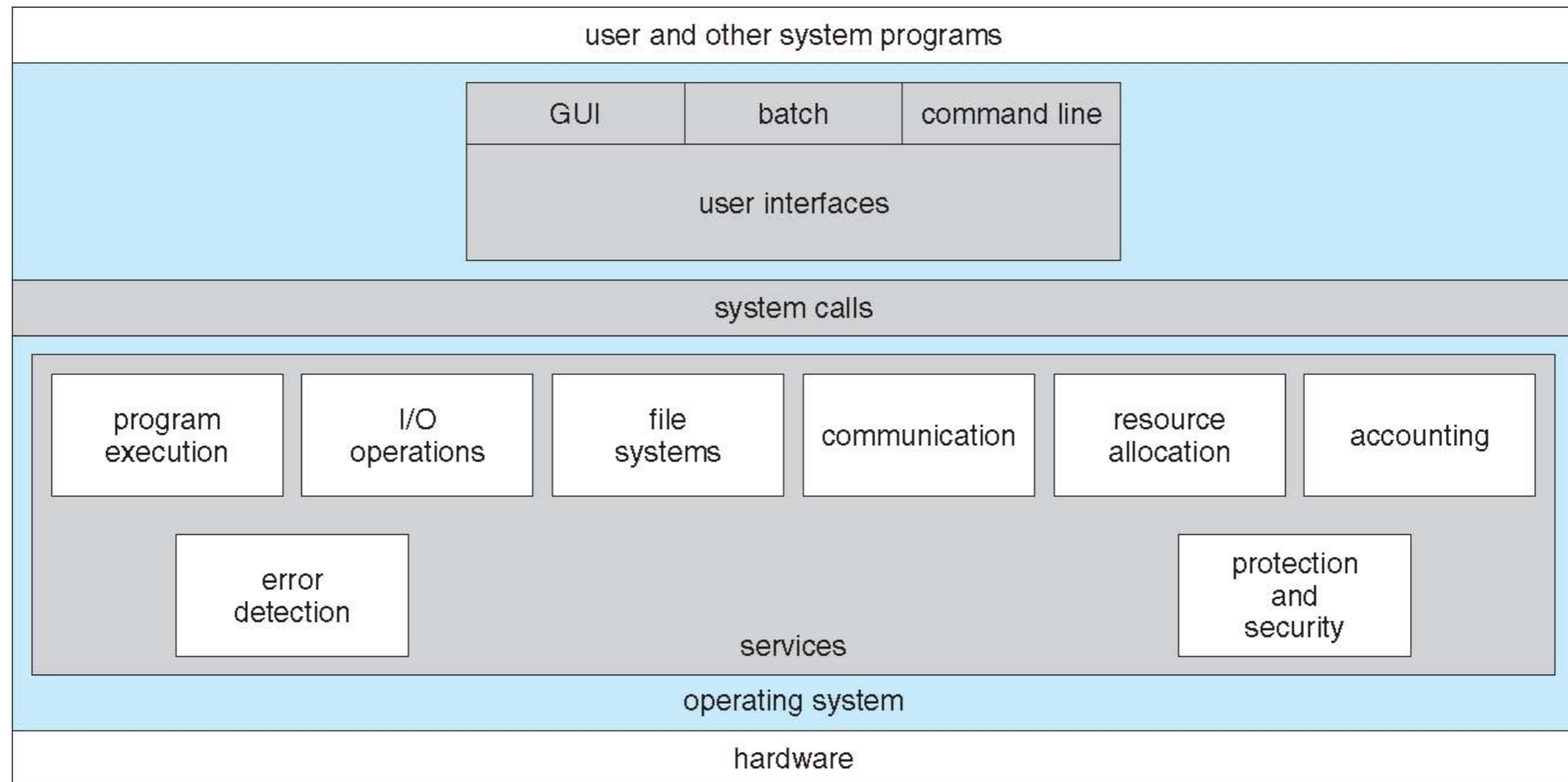
# Operating System Services (User-Visible)

- 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**

# 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)
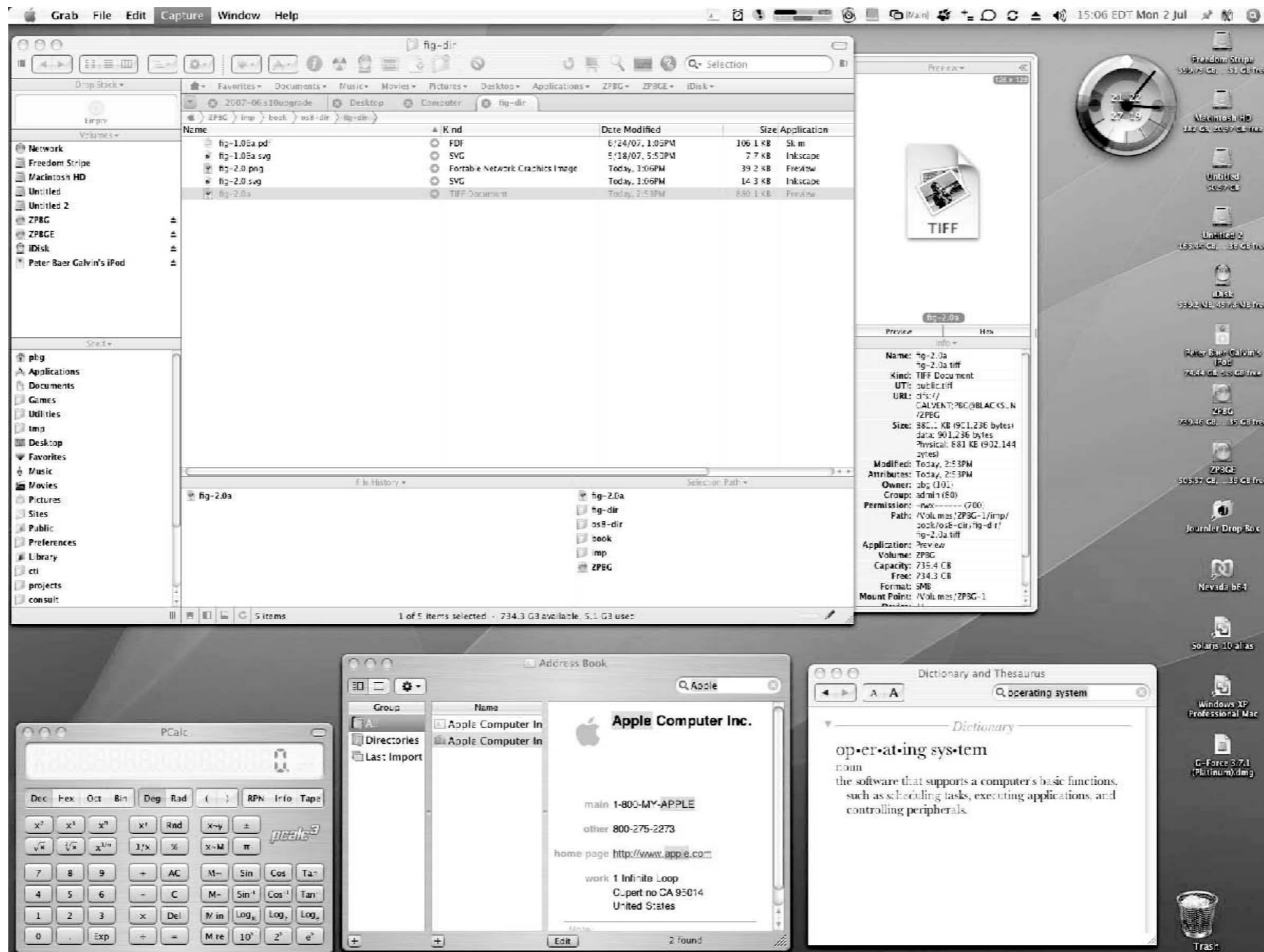
# Bourne Shell Command Interpreter

# 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?

# Example of System Calls

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

| source file | → | destination file |

Example System Call Sequence

Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

# Example of Standard API

- Consider the ReadFile( ) function in the Win32 API

```
                        return value
                            |
                            v
BOOL    ReadFile c  (HANDLE          file,
                     LPVOID          buffer,
                     DWORD           bytes To Read,   ] parameters
                     LPDWORD         bytes Read,
                     LPOVERLAPPED    ovl);
                ^
                |
        function name
```
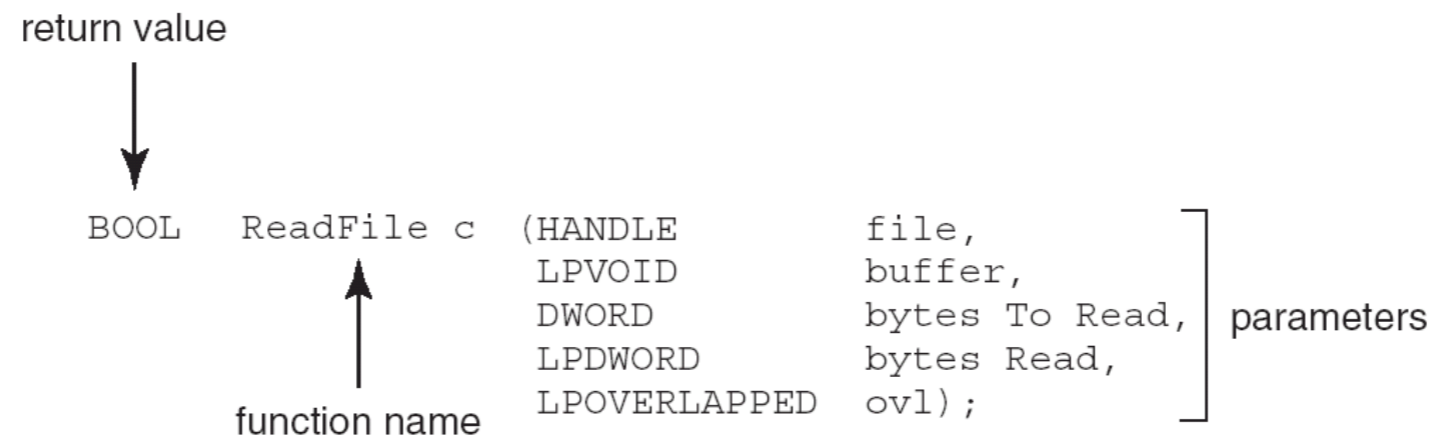
- A description of the parameters passed to ReadFile()

  - 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 actually read

  - LPOVERLAPPED ovl—indicates if overlapped (asynchronous) I/O is used

# System Call Implementation

- 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

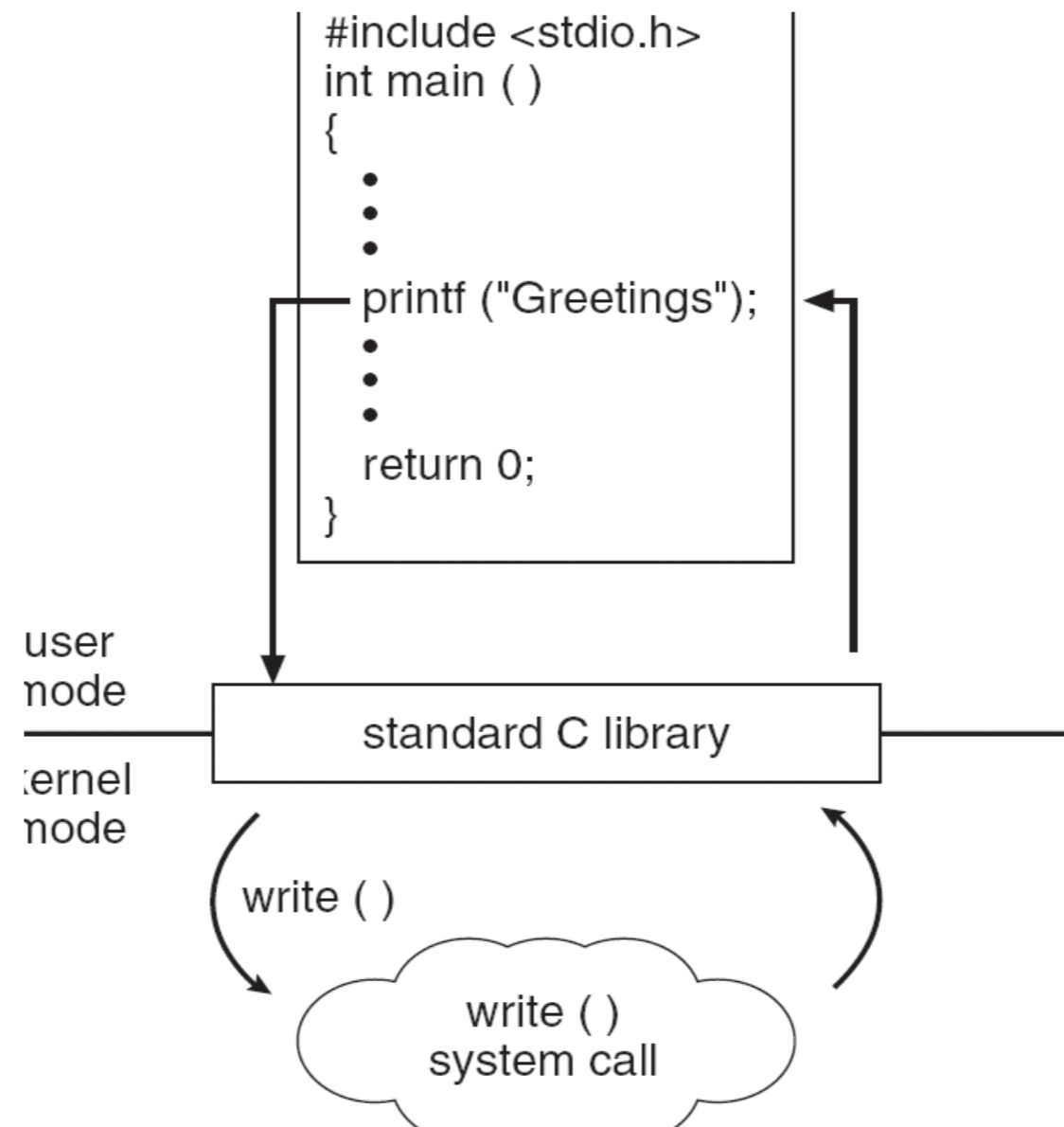  - most details of OS interface hidden from programmers by the API

# API – System Call – OS Relationship
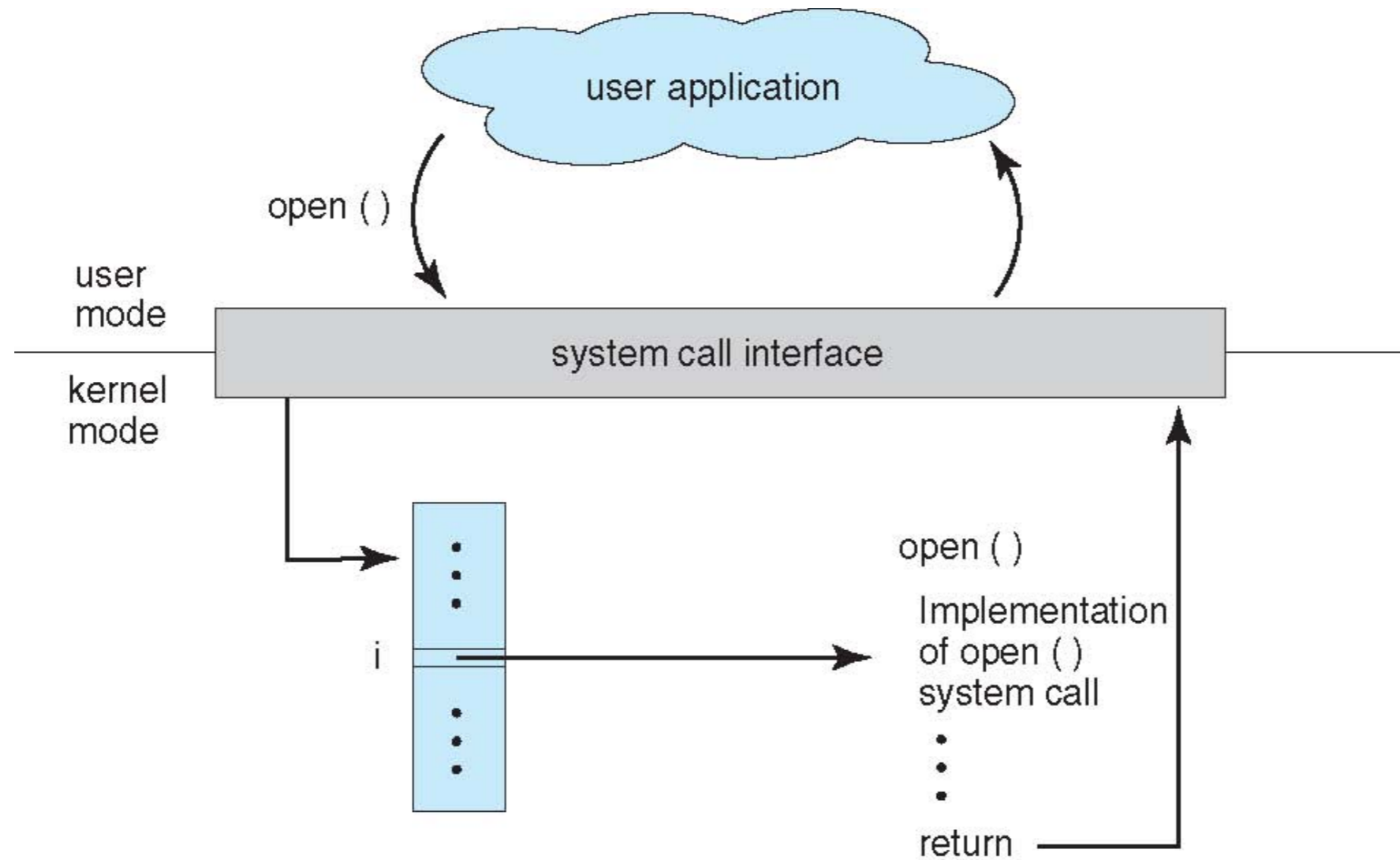
# Standard C Library Example

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

# 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 Table
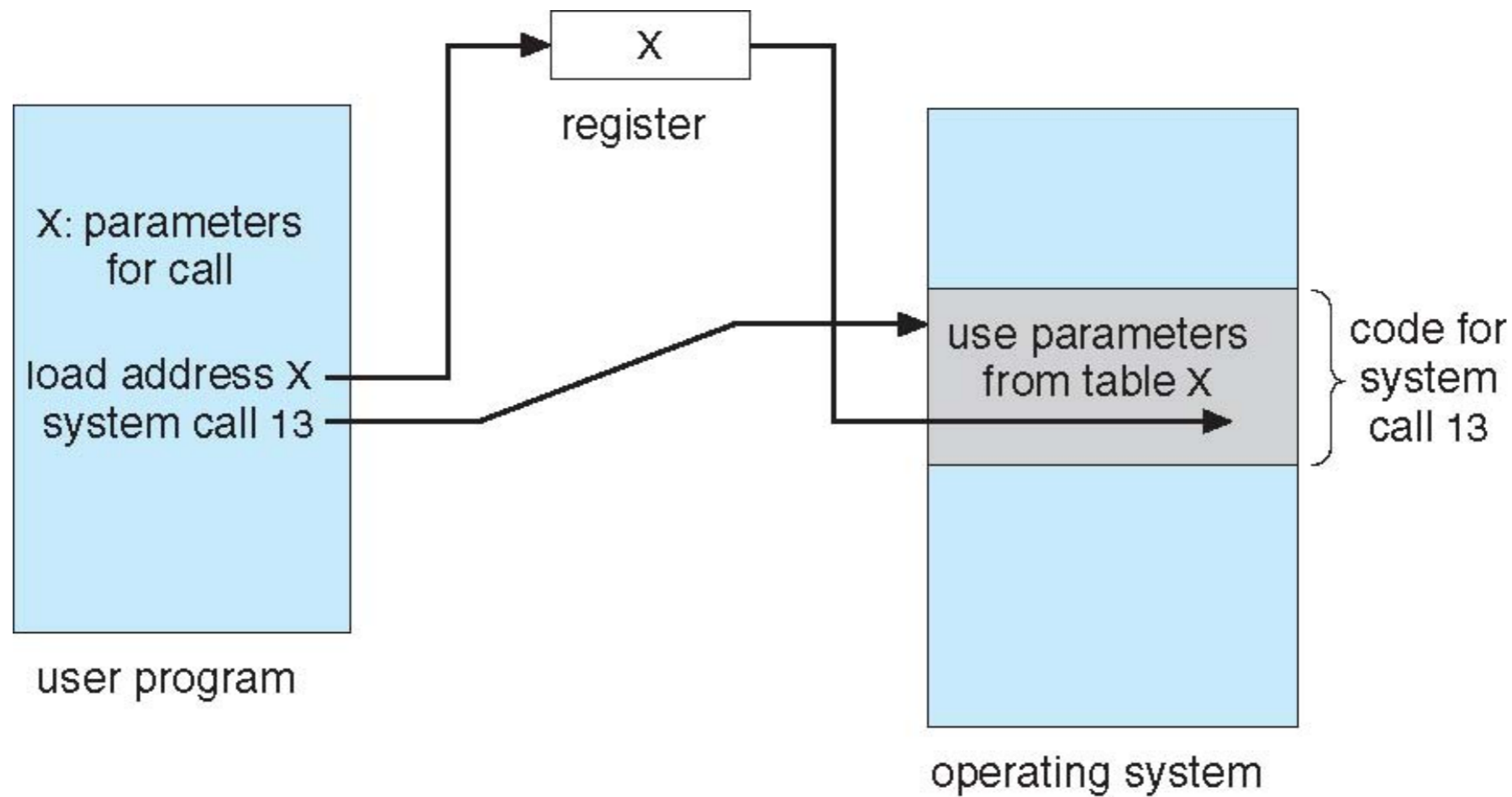
# Types of System Calls

- 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

# Examples of Windows and Unix System Calls

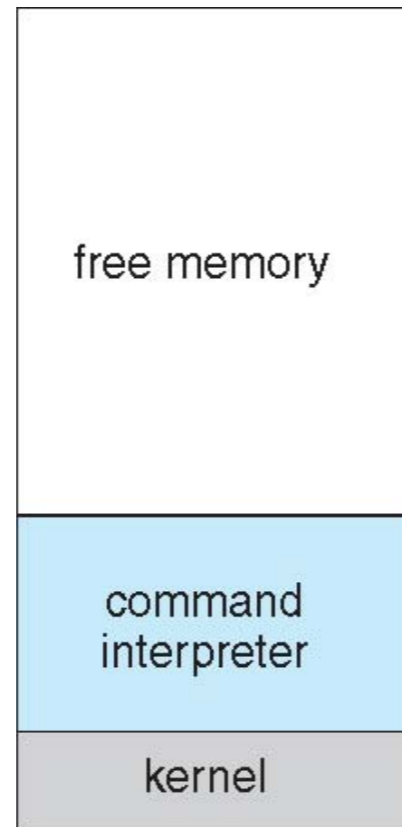|  | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()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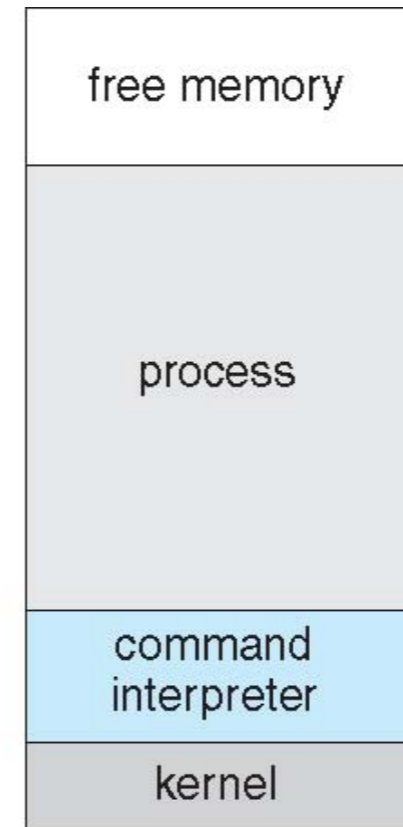
  - program exit -> shell reloaded

# MS-DOS Execution



at system startup

running a program

free memory

free memory

process

command
interpreter

command
interpreter

kernel

kernel

(a)

(b)

# 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
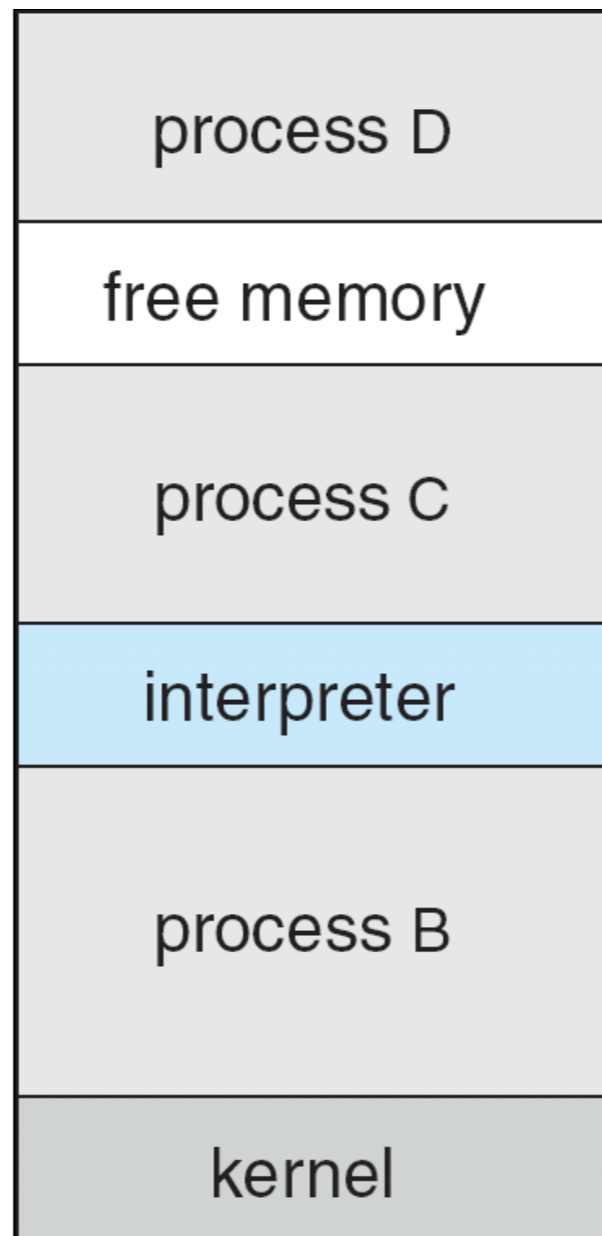
# System Programs

- 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

# System Programs

- File management

  - create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- Status information

  - many different types of status information

    - date, time, available memory, disk space, current users

    - performance, logging, and debugging information

  - some systems implement a registry, database of configuration information

- File modification

  - text editors to create and modify files

  - commands to search contents of files

  - programs to perform transformations of the text

# System Programs

- Programming-language support

  - compilers, assemblers, debuggers and interpreters

- Program loading and execution

  - absolute loaders, relocatable loaders, linkage editors, and overlay-loaders

  - debugging systems for higher-level and machine language

- Communications

  - virtual connections among processes, users, and computer systems

  - allow users to send messages to one another's program, log in remotely, transfer files from one machine to another…

# 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**

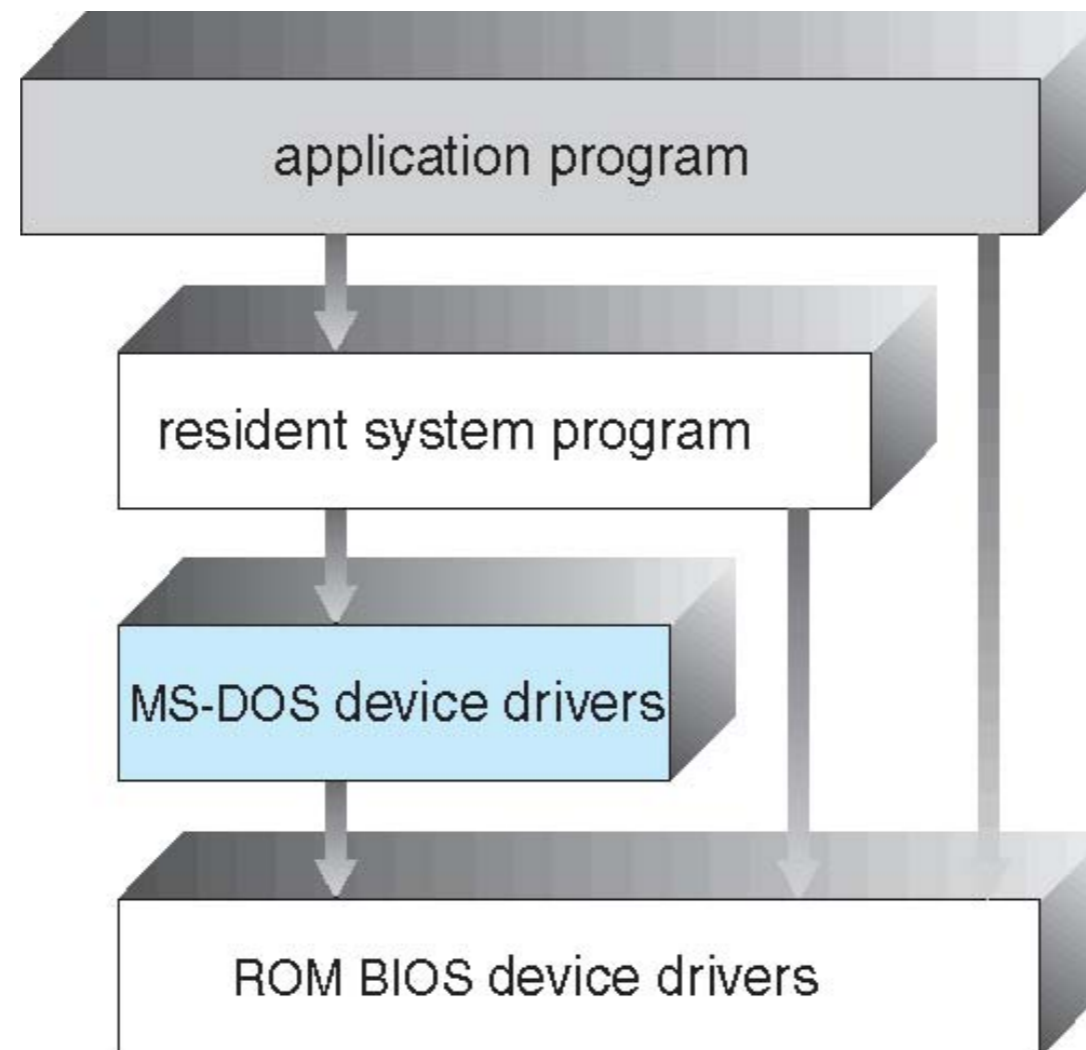    - research system: exo-kernel, multi-kernel…

# 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
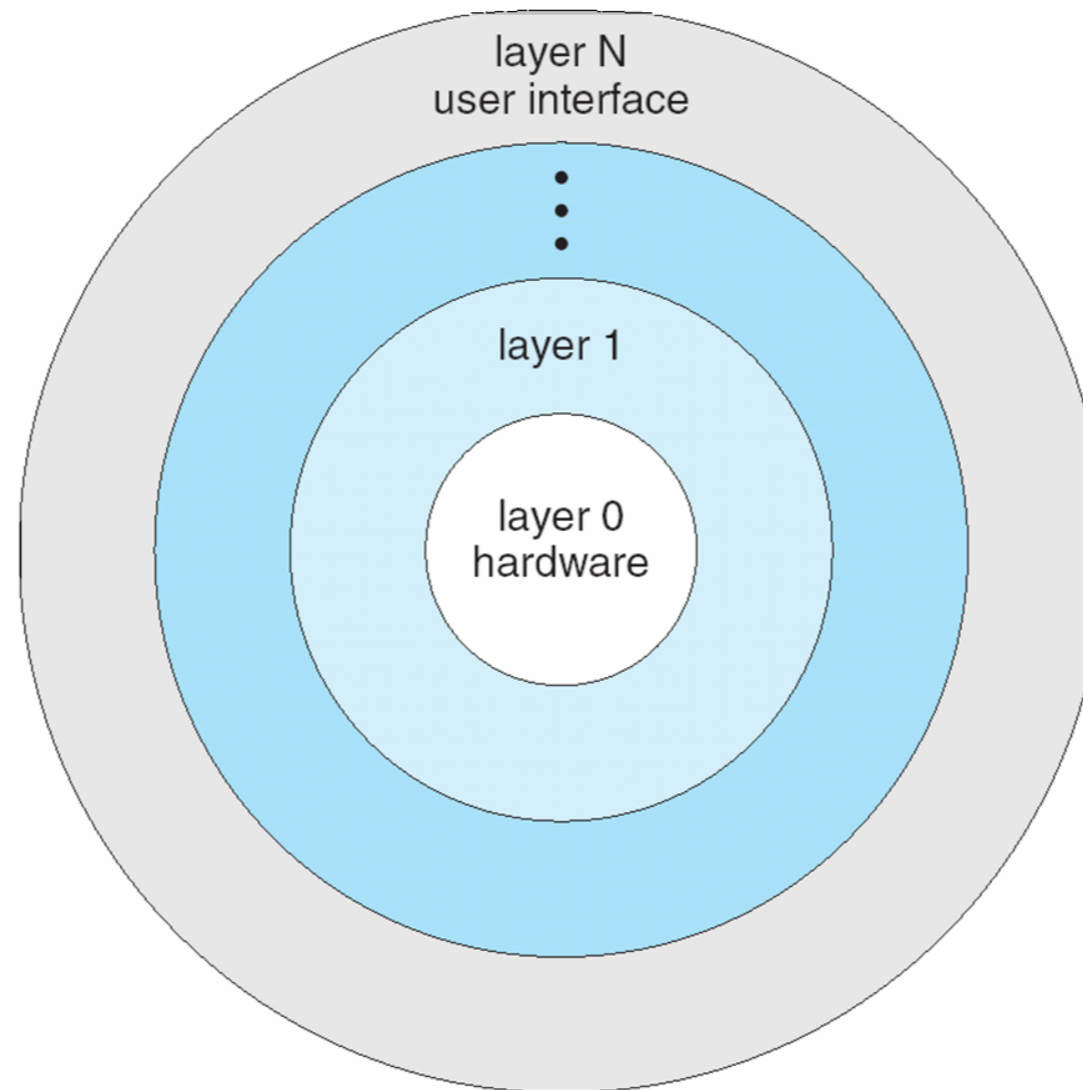
# MS-DOS Structure

# 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
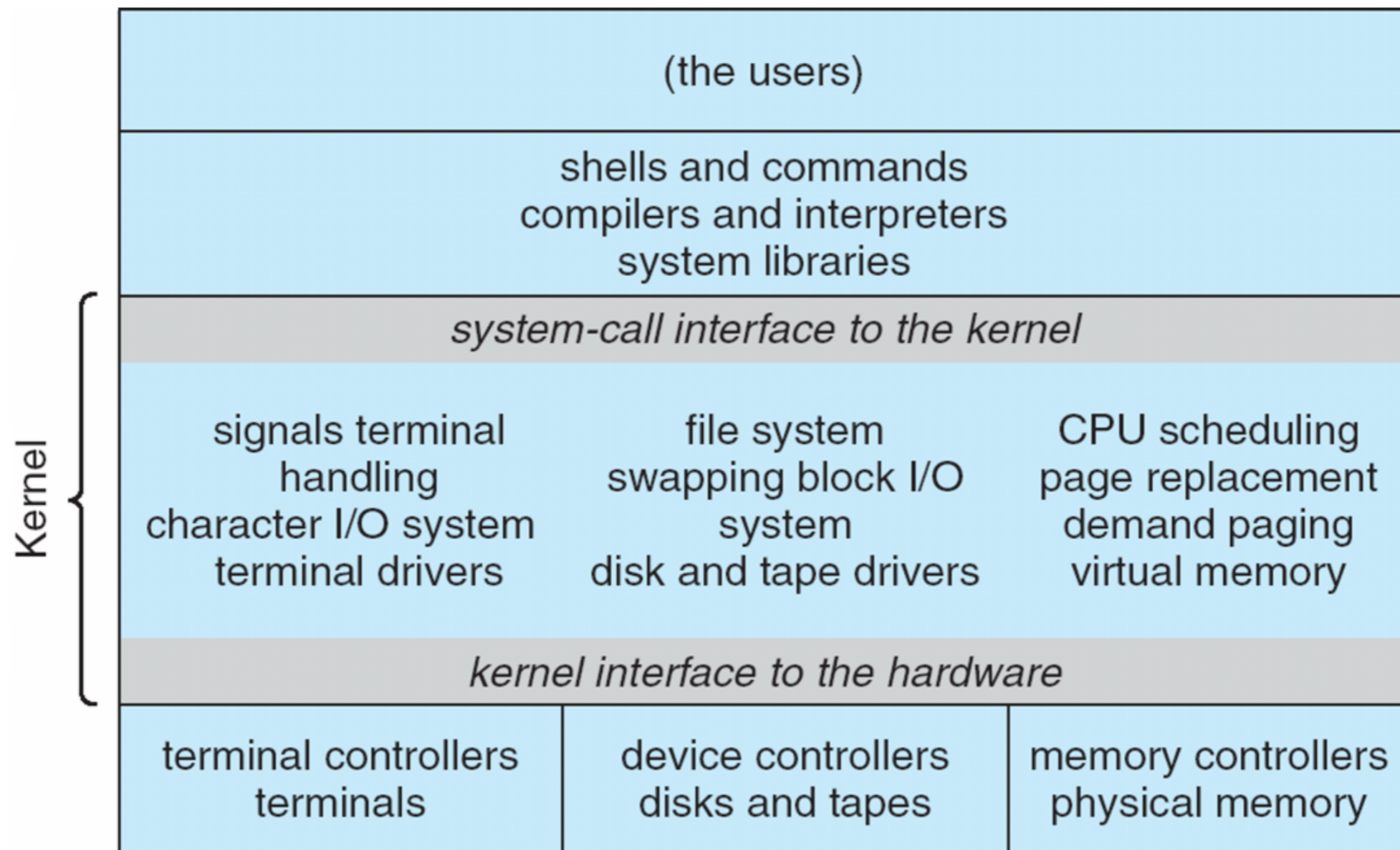
# Layered Operating System

# UNIX

- Limited by hardware functionality, the original UNIX operating system 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 between kernel components makes it impossible to structure kernel strictly in layers

  - 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
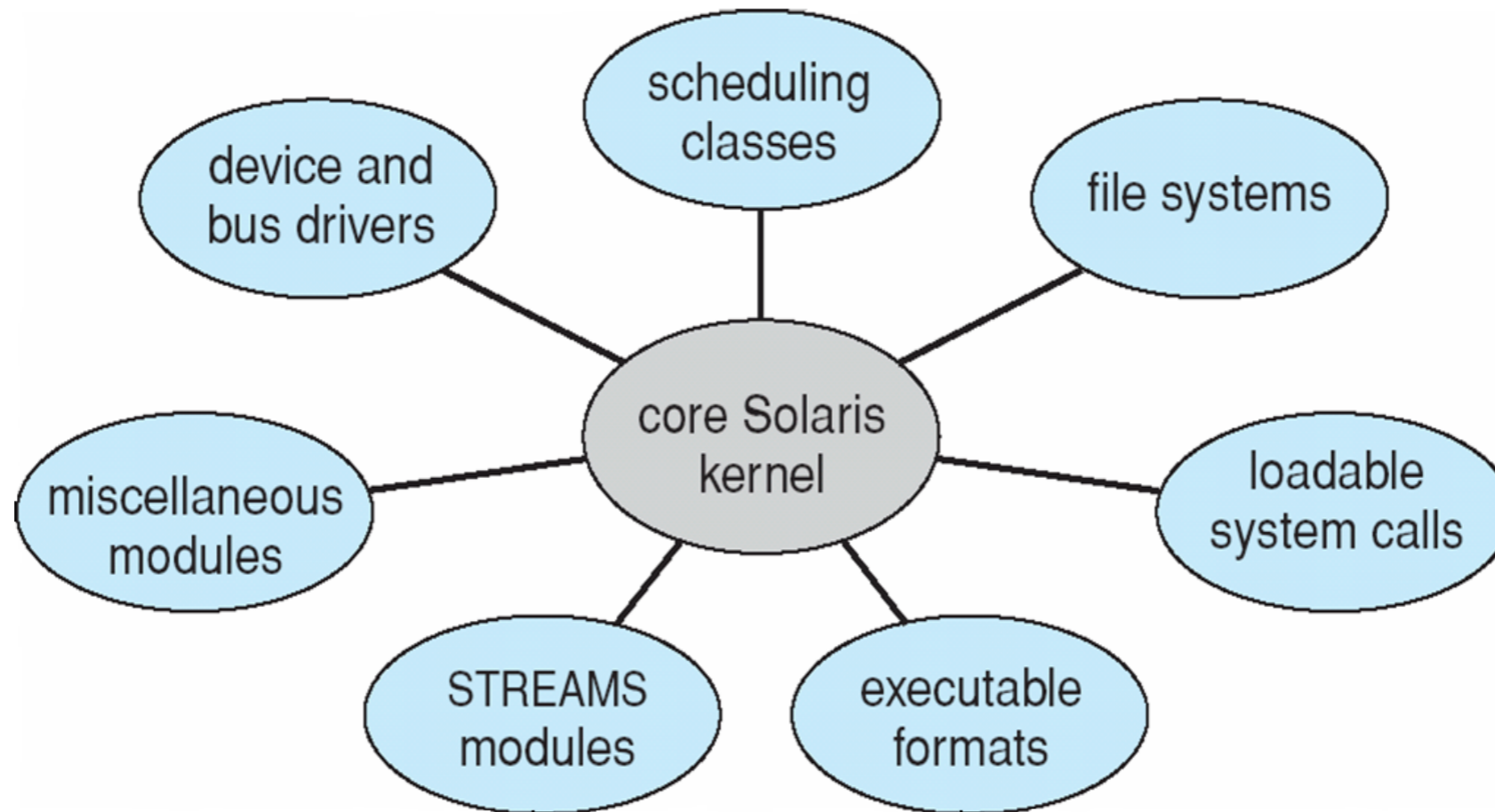
# Microkernel System Structure

- Moves as much from the kernel 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
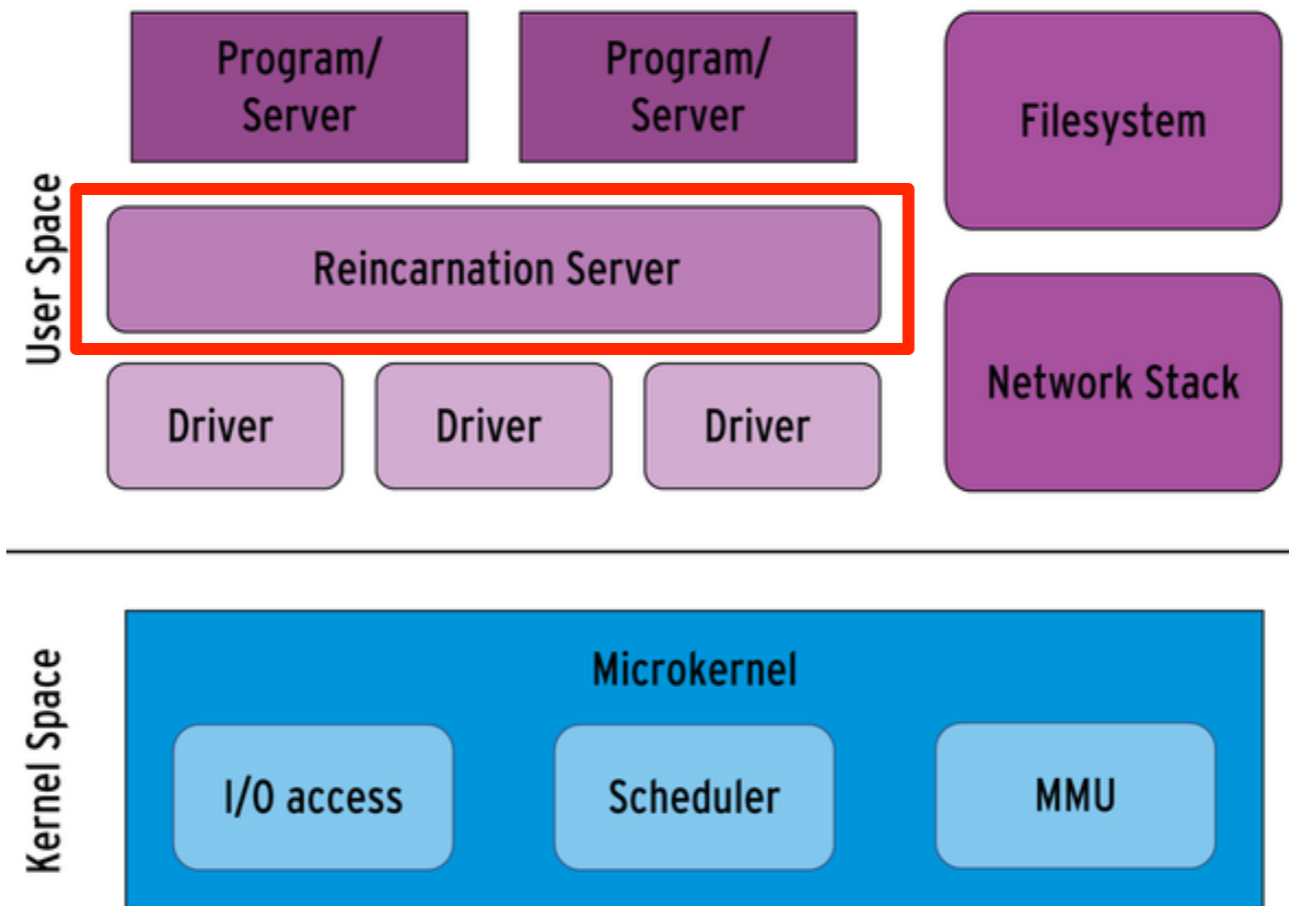
# Mac OS X Structure
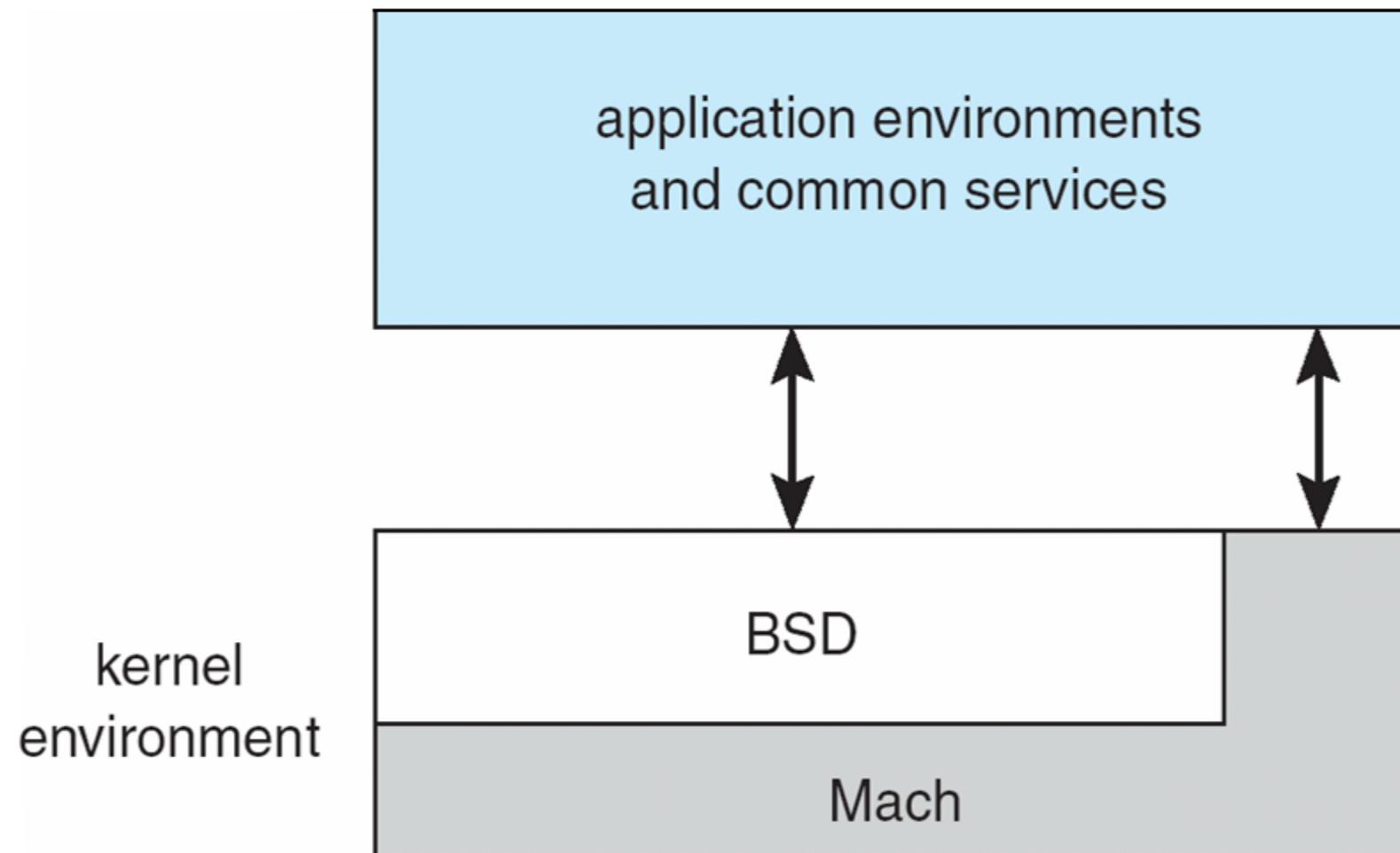
# Exokernel v.s. Normal Kernel
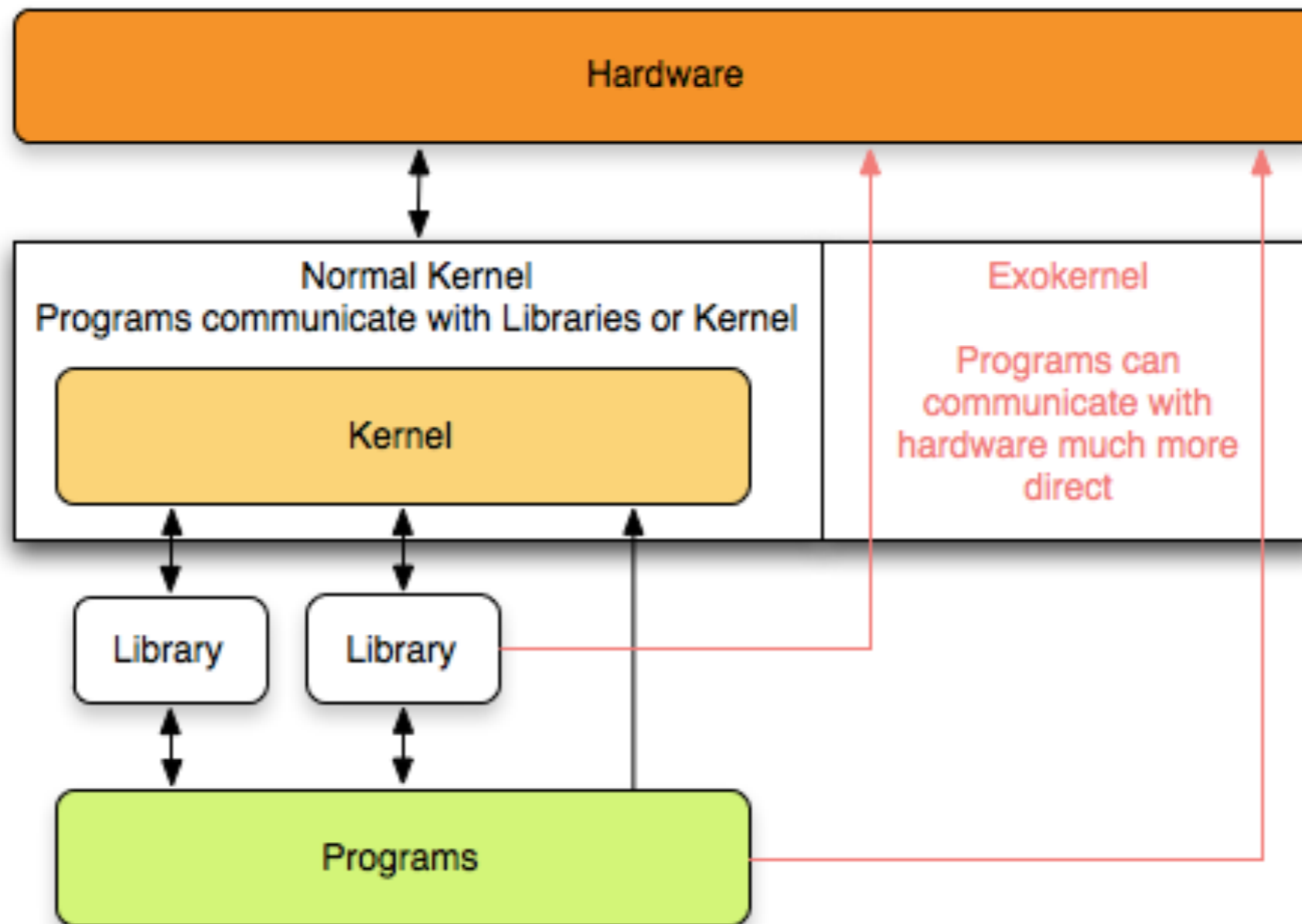
# Virtual Machines

- 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 History and Benefits

- 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

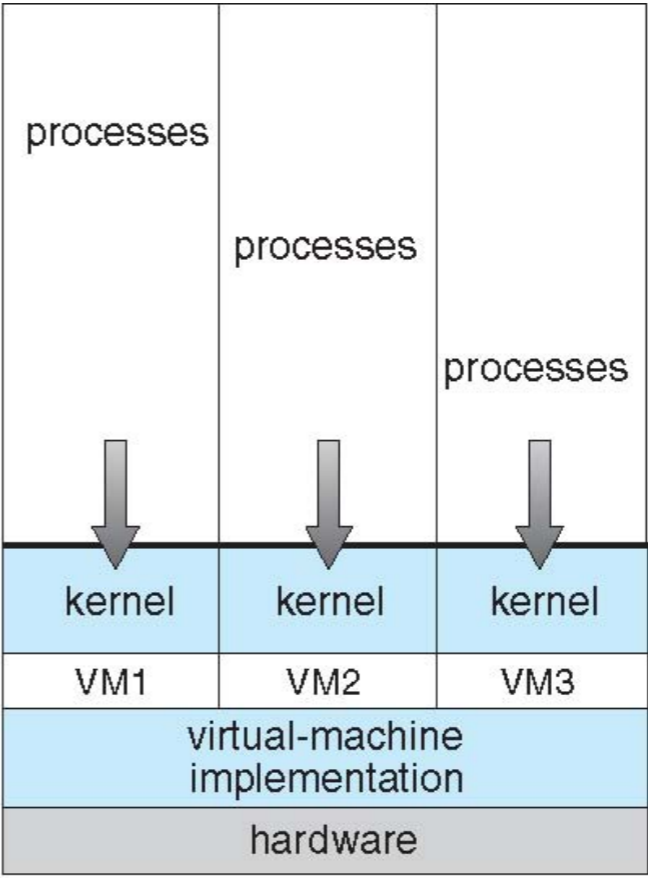# Virtual Machines

non-virtual machine                    virtual machine



programming interface

(a)                                          (b)

# Para-virtualization

- Virtual machine is similar but not identical to hardware

- Guest must be modified to run on paravirtualized hardware

- Guest can be an OS, or in the case of Solaris 10 applications running in containers

# VMware Architecture

| application | application | application | application |
| --- | --- | --- | --- |
| | guest operating system<br><br>(free BSD)<br><br>virtual CPU<br>virtual memory<br>virtual devices | guest operating system<br><br>(Windows NT)<br><br>virtual CPU<br>virtual memory<br>virtual devices | guest operating system<br><br>(Windows XP)<br><br>virtual CPU<br>virtual memory<br>virtual devices |
| | virtualization layer | | |

host operating system
(Linux)

hardware

| CPU | memory | I/O devices |

# Solaris 10 with Two Containers

# Java Virtual Machine
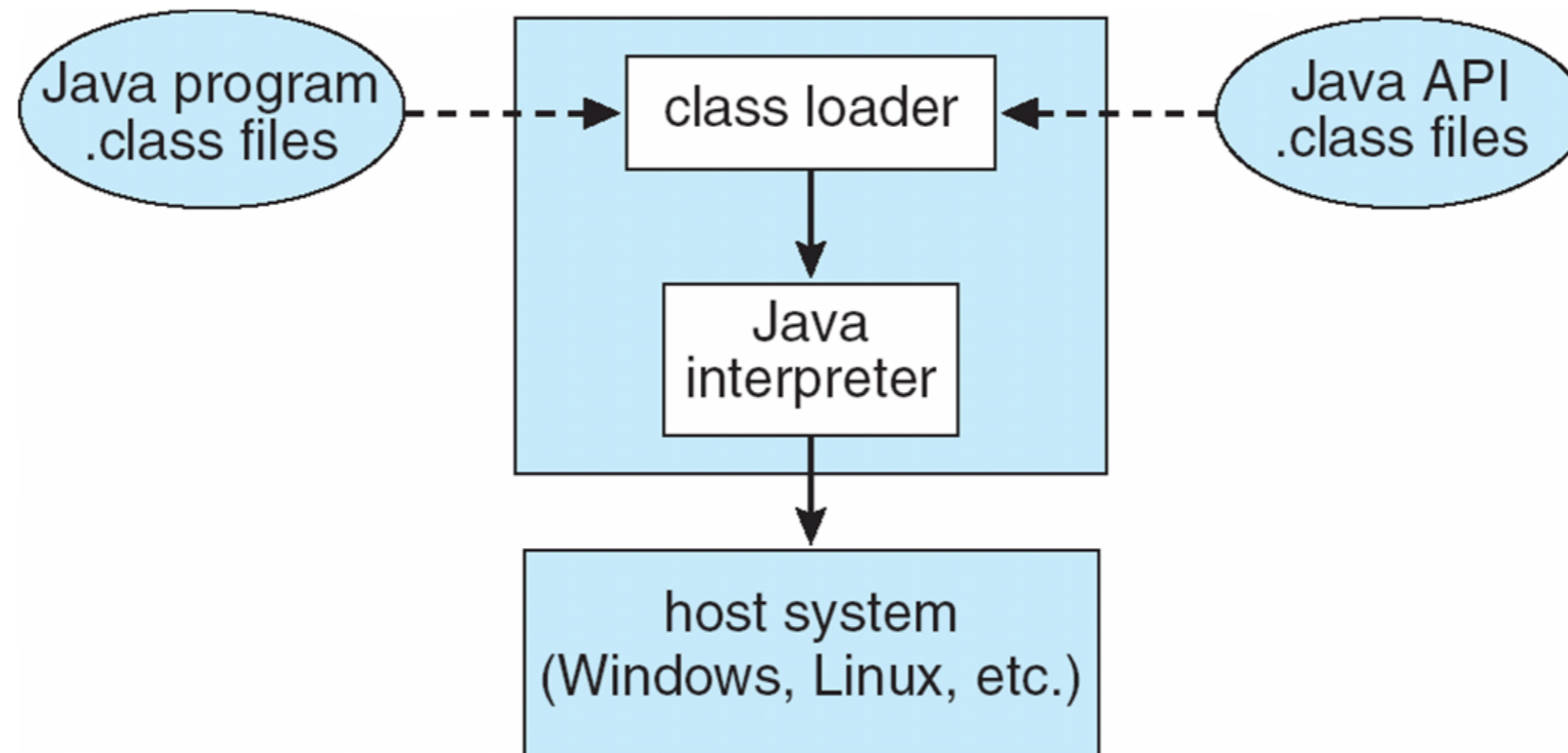
# Operating-System Debugging

- Debugging is to find and fix errors, or bugs

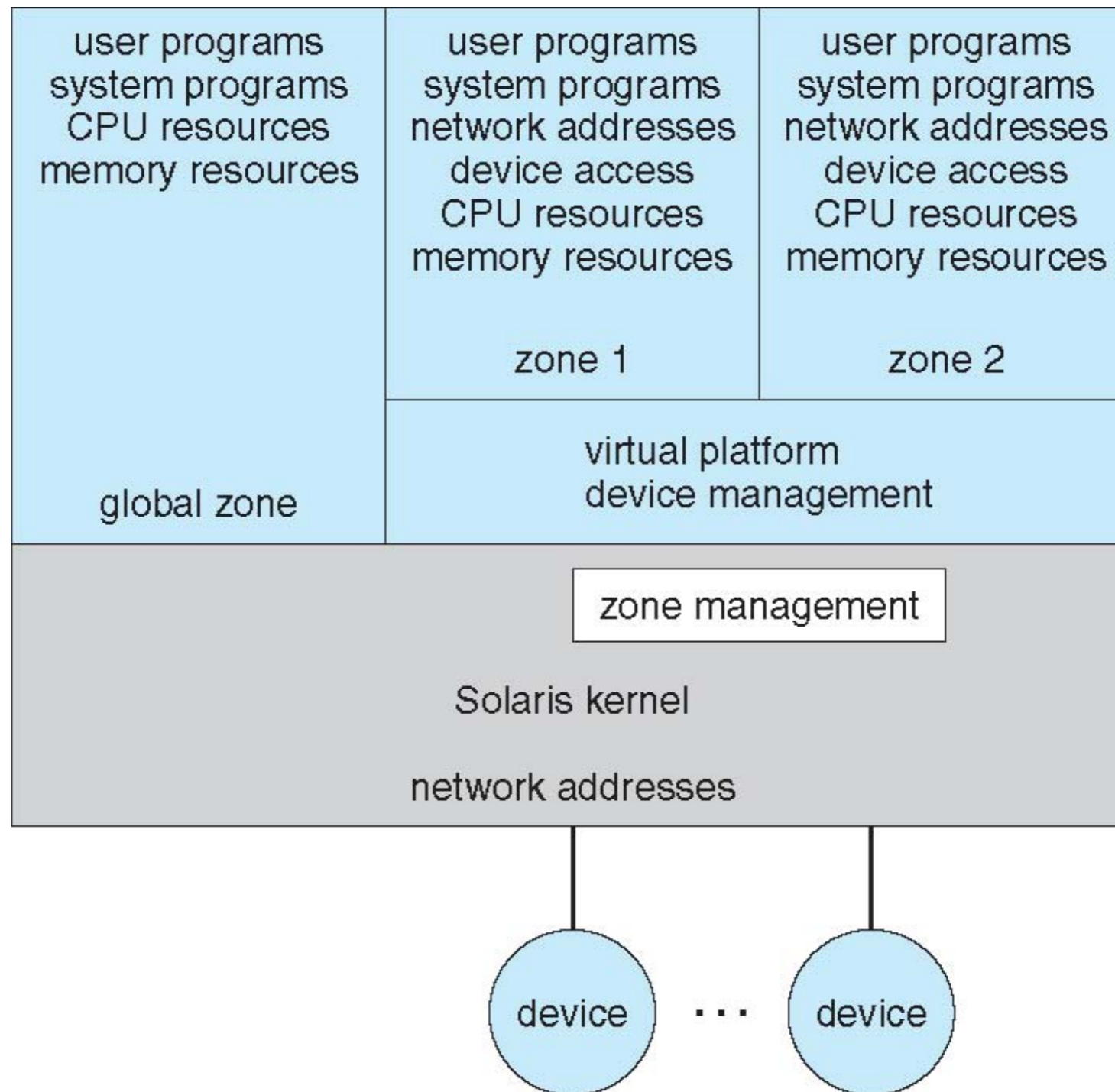  - OS generates log files containing error information

    - 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