Chapter 2
System Structures

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

Outline

- Operating System Services
- Interface Provided by an Operating System
  - User Interface
  - Programming Interface of Operating System
    - System Calls
    - Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure

Operating System Services

- Functions that are **helpful to the user/applications**
  - User interface - Almost all operating systems have a user interface (UI)
    - Command-Line Interface (CLI), Graphics User Interface (GUI)
  - Program execution - The system must be able to load a program into memory and to run that program, terminate 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 - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

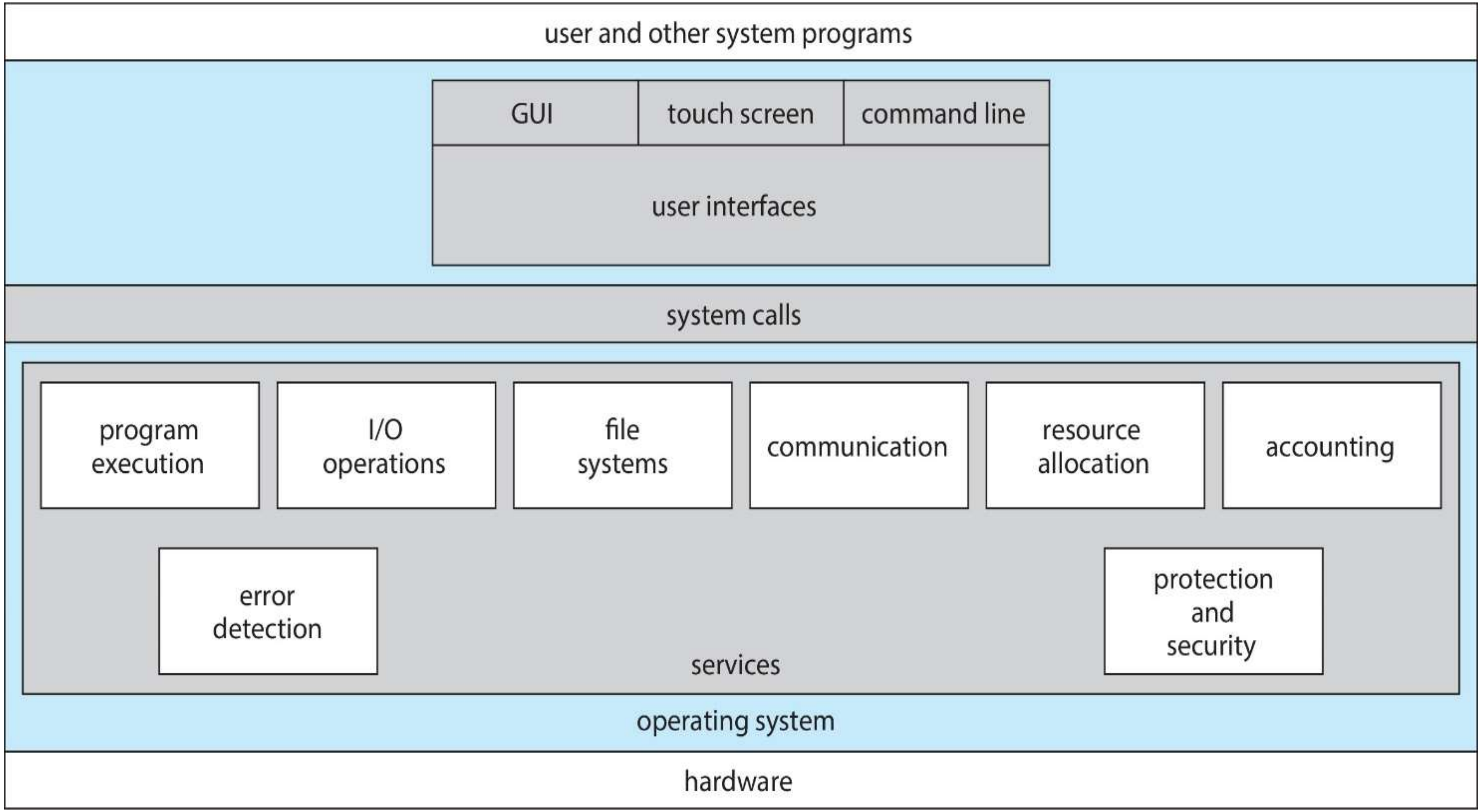
Operating System Services (Cont.)

- Functions that are **helpful to the user/applications (Cont.)**
  - Communications – Processes may exchange information, on the same computer or between computers over a network
  - Error detection – OS needs to be constantly aware of possible errors
    - Hardware errors (CPU, memory, I/O devices…)
    - Software errors (invalid memory access…)
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

Operating System Services (Cont.)

- Functions that ensures 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 - CPU cycles, main memory, file storage, I/O devices
  - Logging/Accounting **-** To keep track of which users use **what kinds of** and **how much** computer resources
  - Protection & Security issues raised from resource sharing
    - ensuring that all access to system resources is controlled
    - concurrent processes should not interfere with each other
    - user authentication
    - defending resources from invalid access attempts

A View of Operating System Services



User Interface Provided by an OS - CLI

CLI allows users to enter commands directly

- Sometimes implemented in kernel, sometimes by system program (e.g. **shells**)
- Sometimes multiple flavors implemented –
  - bash, C shell, TC shell, Korn shell…
- Primarily fetches a command from user and executes it
  - Some commands are built-in, and the others just names of executable programs
    - » If the latter, adding new features doesn't require shell modification
  - E.g., rm file.txt
    - » Execute the *rm* executable program
    - » Pass "file.txt" as the parameter

User Interface Provided by an OS - GUI

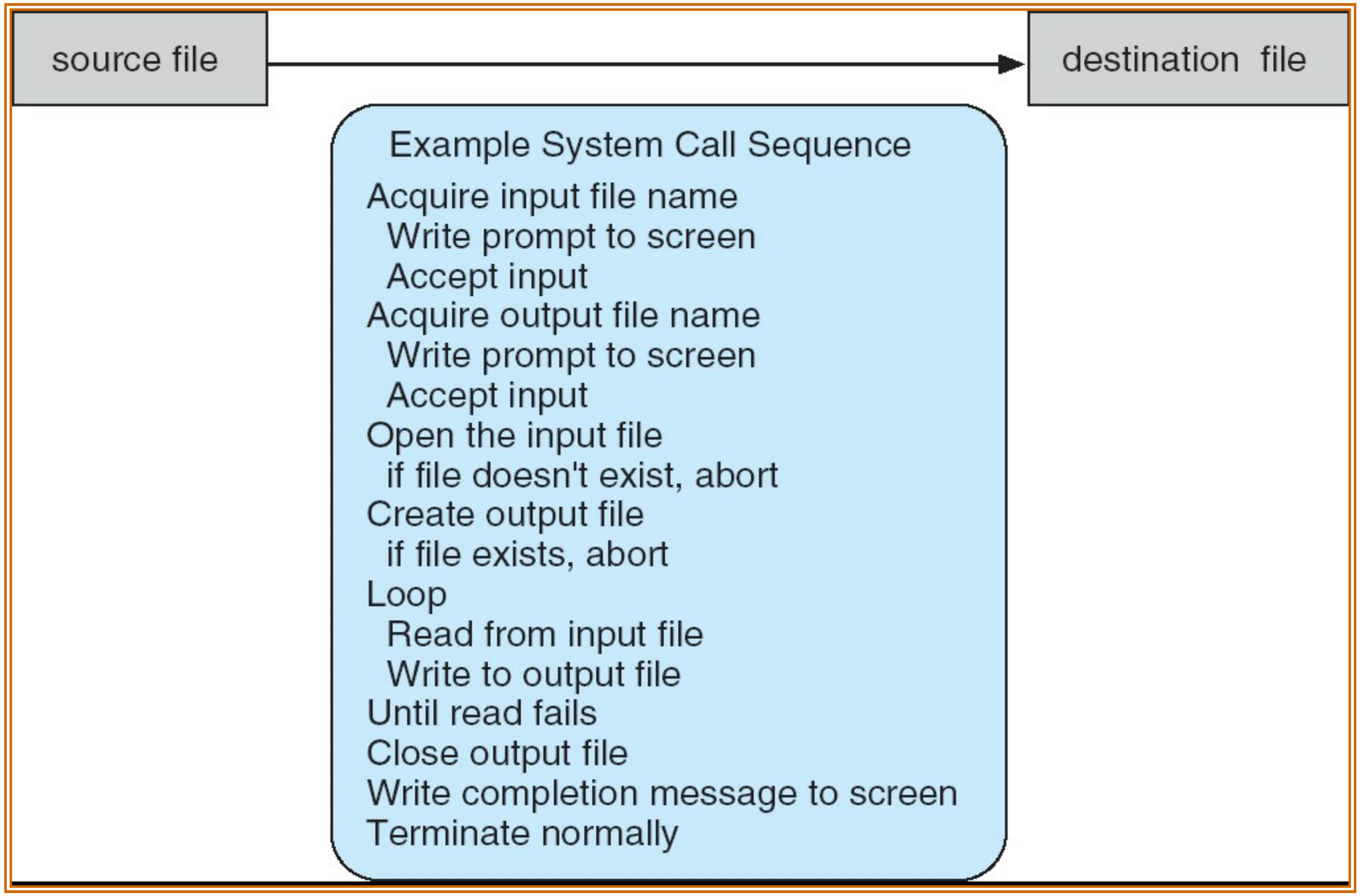
- User-friendly **desktop** metaphor interface
  - Desktops, windows, icons, …
  - **Icons** represent objects such as files, programs, actions, etc
  - Various actions can be performed on the objects, e.g., provide information, execute function, open directory (known as a folder)
  - Invented at Xerox PARC
    - The first GUI in 1973
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface and CLI available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Programming Interface of an OS - System Calls

- System call
  - the programming interface provided by the OS
  - Typically written in a high-level language (C or C++)
- In many cases, a user program calls a **higher-level** Application Program Interface (API) rather than directly calling the system calls
  - API: a set of functions that can be invoked by the programs
- Three most common APIs are Windows 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 higher-level APIs rather than system calls?
  - Easier to use
  - Portability
- Note that the system-call names used throughout this text are generic
  - Each OS has its own name for each system call

Example of System Calls

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



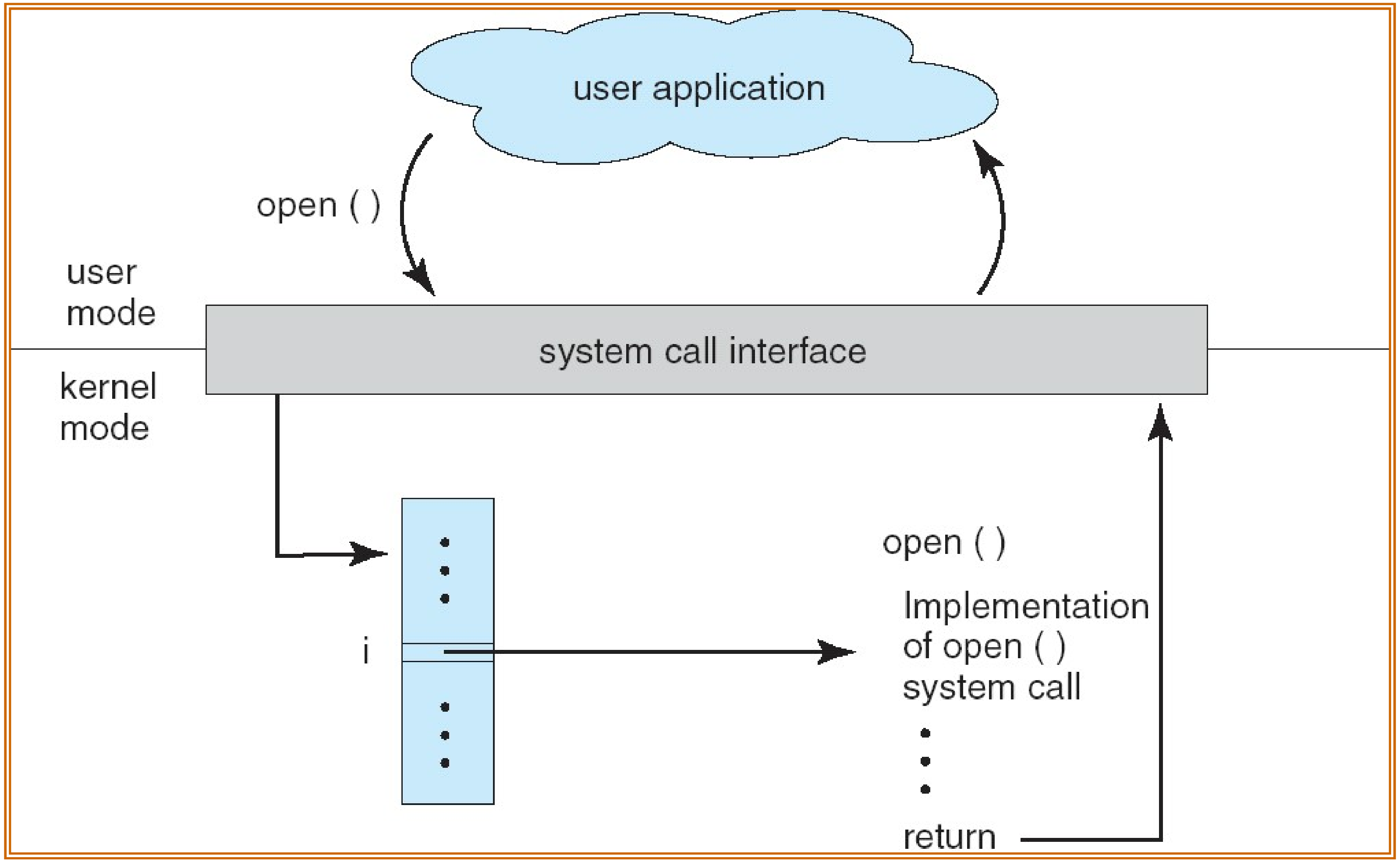
System Call Implementation

- 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 handler in the OS kernel
- After system call handler finishes, the following are returned
  - status of the system call
  - return values
- The caller knows Nothing about how the system call is implemented
  - Just needs to follow the high-level API or system call interface and know what OS will do

For programs using the high-level API, details of the OS system call interface are also hidden by the API

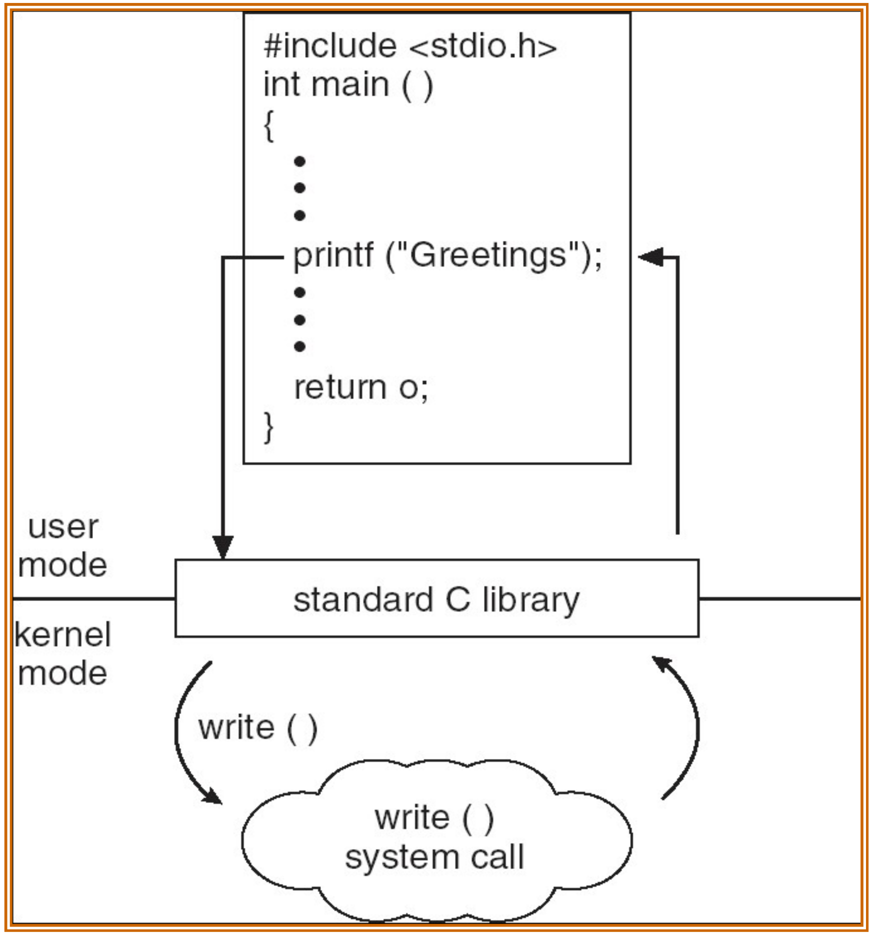
- Managed by run-time support library

System Call Implementation



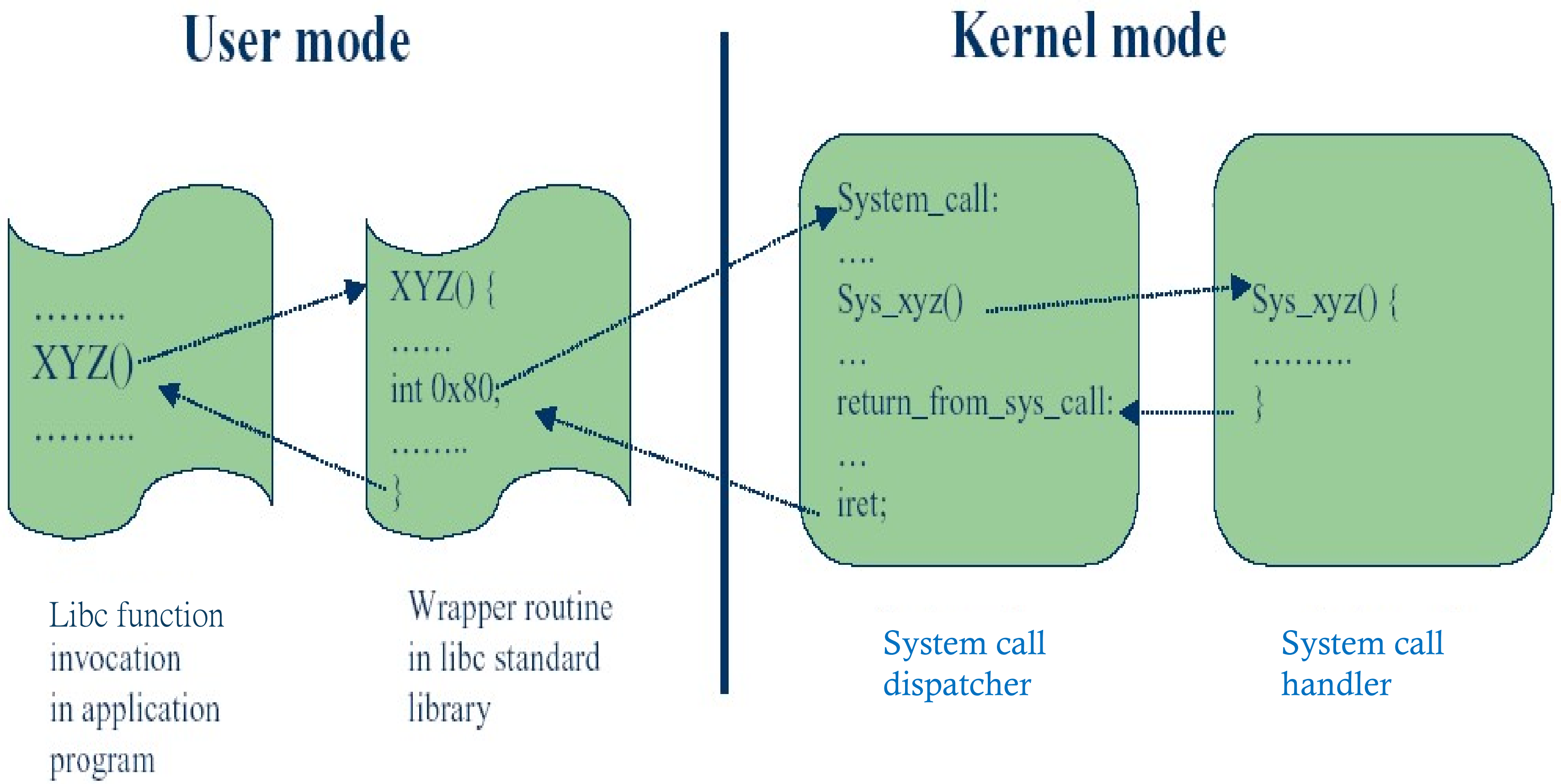
Standard C Library Example

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



🡐Incorrect!
C Lib. resides totally in user mode

System Call on Linux/x86



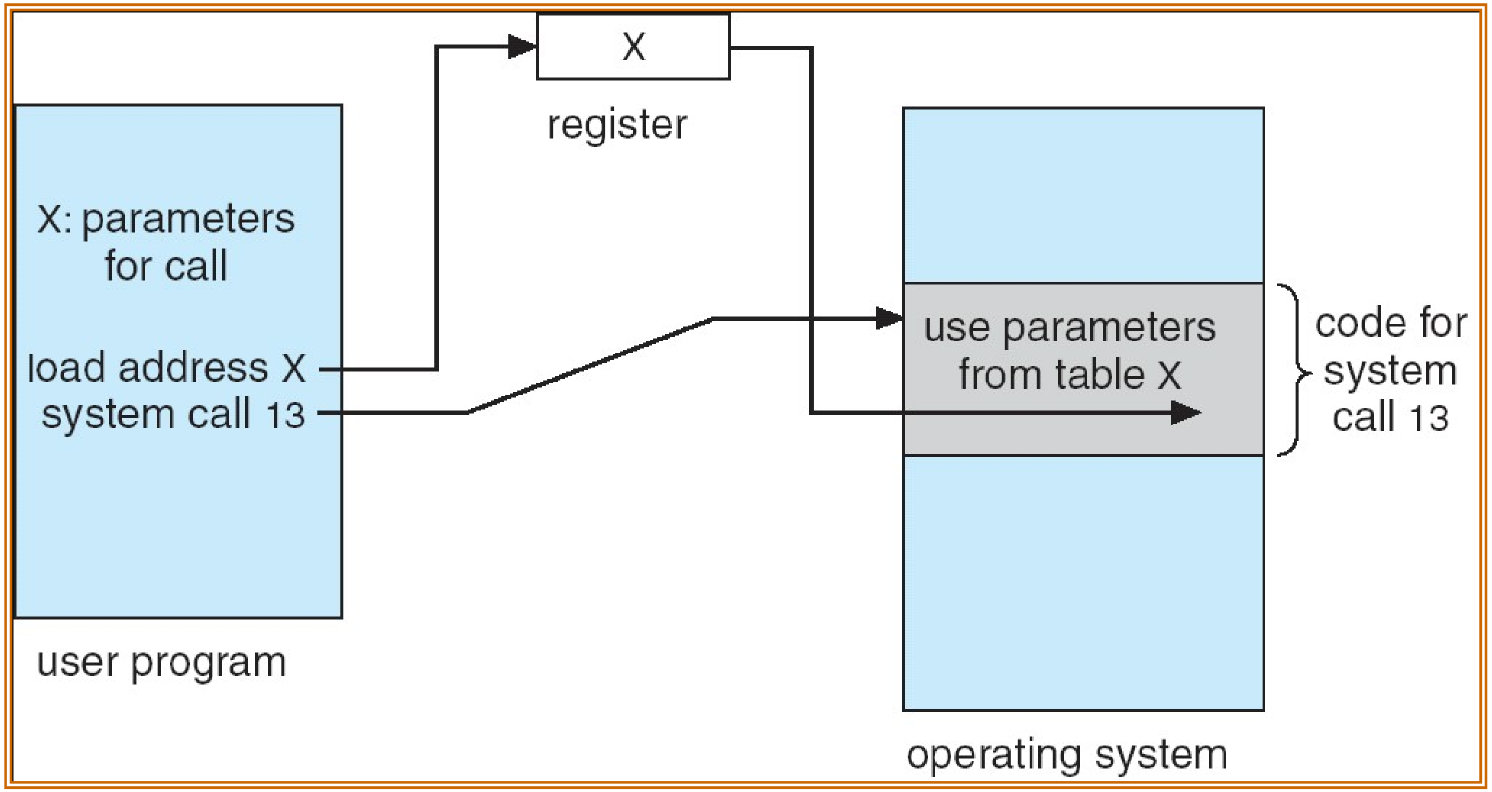
In modern x86, the *syscall* instruction can be used to perform faster system call
- A replacement of int 0x80

Passing Parameters in a System Call

- Often, more information is required than simply the ID/number of the desired system call
  - Type and amount of information/parameters vary according to OS and system call

- Three general methods used to pass parameters to the OS
  - **Registers**: the simplest way to pass the parameters
    - In some cases, there may be more parameters than registers
  - **Memory block/table**: parameters stored in a block/table in memory, and the address of the block passed as a parameter in a register
  - **Stack**: parameters placed, or *pushed,* onto the stack by the program and *popped* off the stack by the operating system
    - **Do you know "stack"?**
  - Memory block and stack methods allow large parameters or a large number of parameters to be passed

Parameter Passing via Block/Table



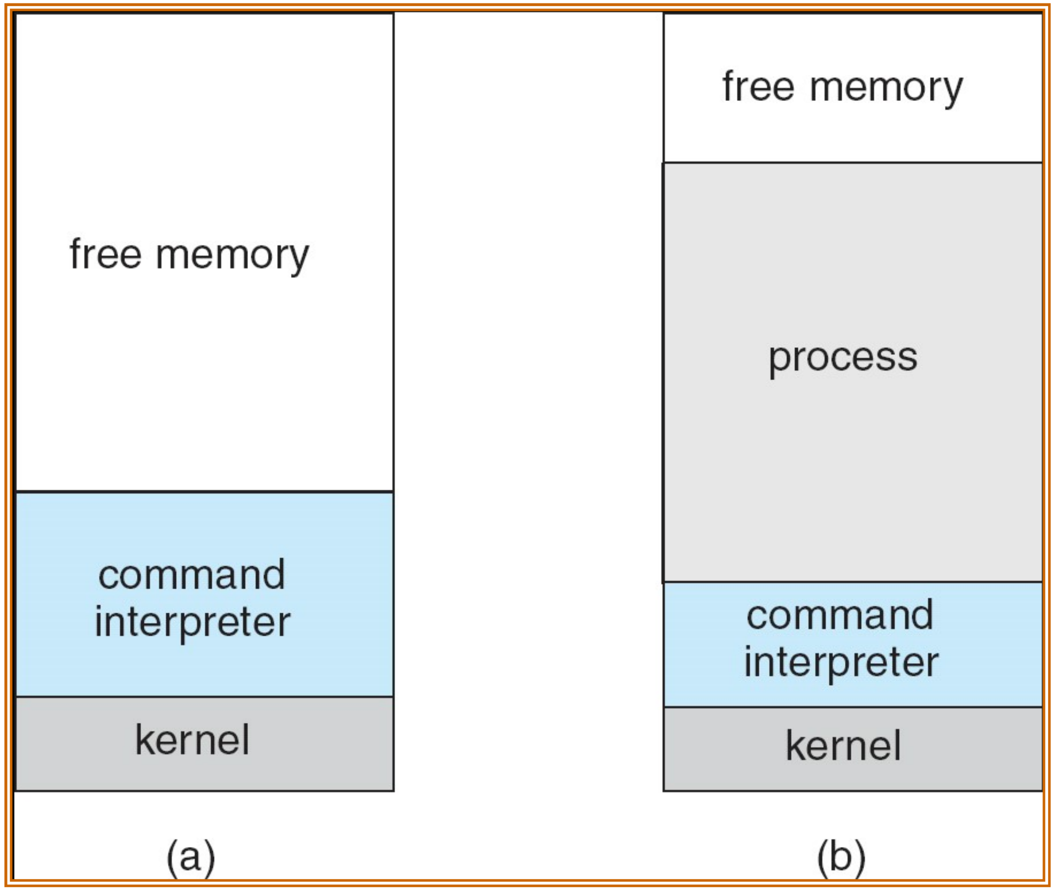
Types of System Calls

- Process control
  - Create/execute/terminate processes
  - Get/set process attributes
  - Wait for time
  - Wait signals/events
  - Allocate and free memory

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

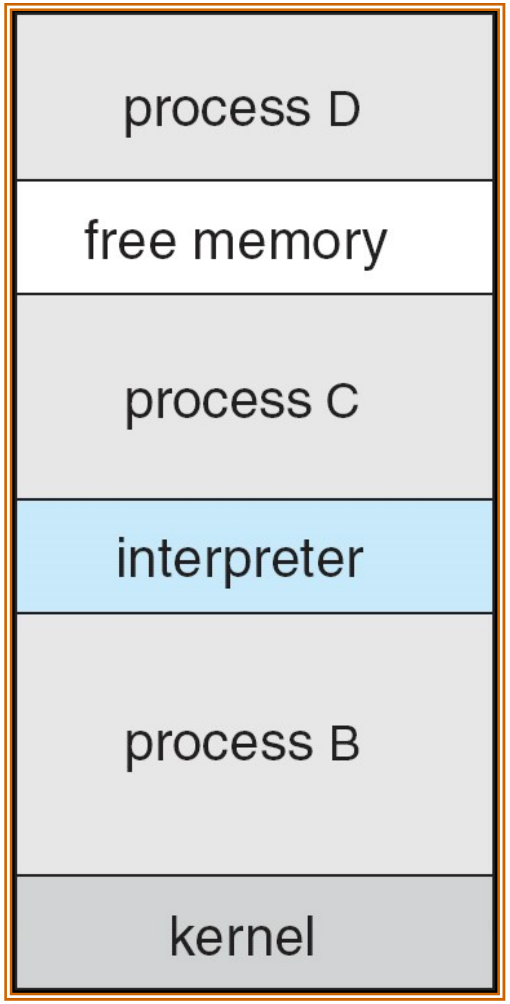
Described later

MS-DOS Execution – Single-Tasking



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

FreeBSD Execution – Multi-Tasking



Types of System Calls (cont.)

- File management
  - Create/delete files
  - Open/close files
  - read, write, reposition (seek)
  - Get/set attributes
- Device management
  - Request/release devices
  - read, write, reposition (seek)
  - Get/set attributes
  - Attach/detach devices

Types of System Calls (cont.)

- Information maintenance
  - Get/set date or time
  - Get/set process or system information
- Communications
  - Create/delete connections
  - Send/receive messages
  - Transfer status information

Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

List of Linux system calls: https://man7.org/linux/man-pages/man2/syscalls.2.html

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File management
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications

Described later

- **Most users' view of an OS** is defined by system programs, not the actual system calls
- Sometimes the above functions are provided by system utilities or application programs.

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. E.g., file manager
- Status information
  - Some provide general status 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
  - E.g., **task manager** and **registry** in Windows, **top** utility in Linux

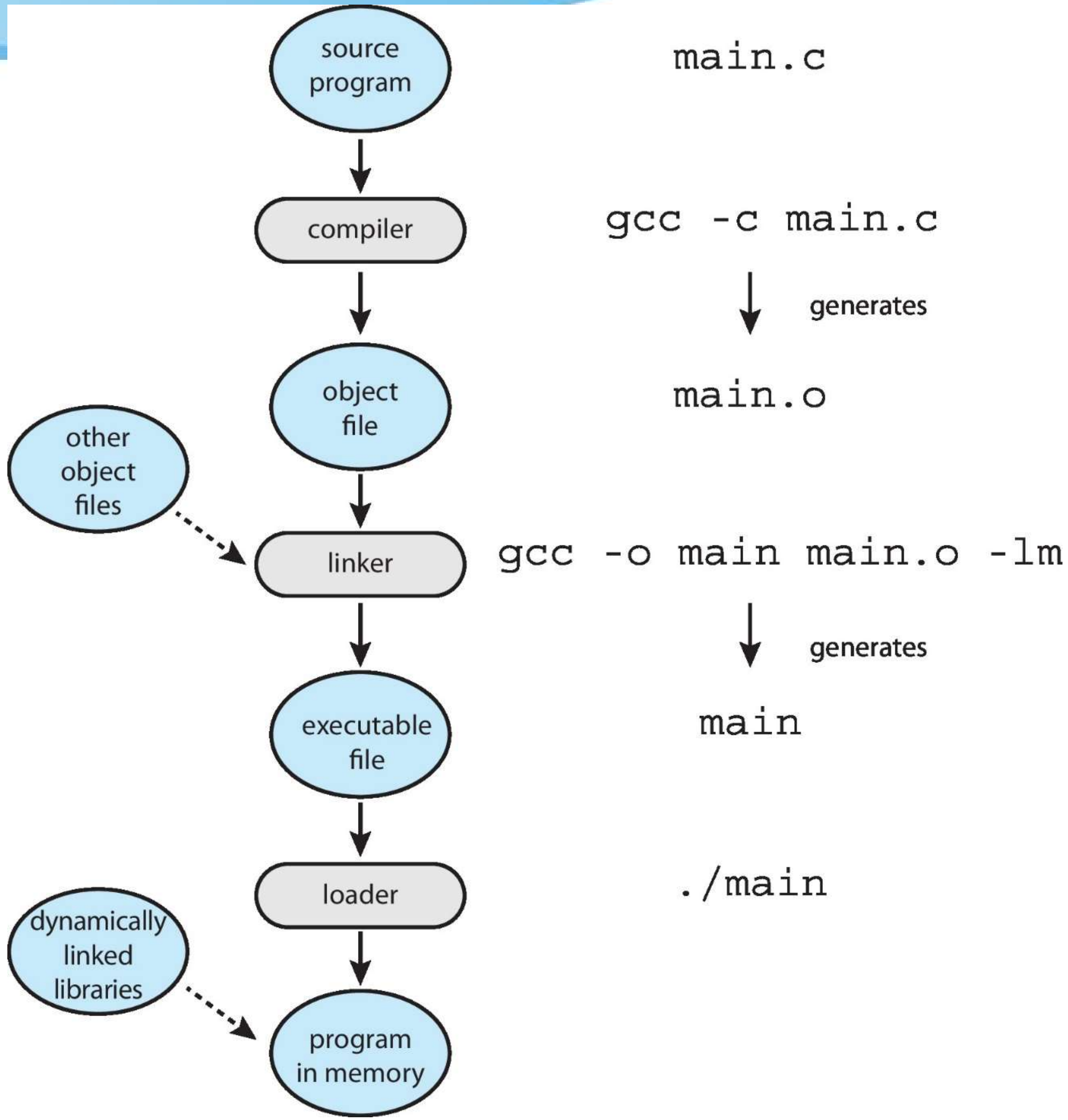
System Programs (Cont.)

- 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 - executable file loaders
  - ELF (Extensible Linking Format) loader
- Communications - mechanisms 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
  - E.g., Outlook, Chrome/Edge…

Linkers and Loaders

- Source code compiled into object files
- **Linker** combines object files into single binary **executable** file (program)
- Program resides on secondary storage as binary executable
- Program must be brought into memory by **loader** to be executed
  - Object, executable files have standard formats, so operating system knows how to load and start them
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)

The Role of the Linker and Loader



Operating System Design and Implementation

- OS Design
  - Internal structure of different operating systems can vary widely
  - Start by defining goals and specifications
  - Affected by choice of hardware, and types 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.)

- OS Design
  - Separation of *policy* and *mechanism*

    Policy: What will be done?

    Mechanism: How to do it?

    - e.g., policy and mechanism in CPU scheduling (CPU cycle allocation)?
  - 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 maximal flexibility if policy decisions are to be changed later

Operating System Design and Implementation (Cont.)

- OS Implementation
  - In early days, OS were written in assembly languages
    - MS-DOS was written in Intel 8088 assembly language
  - Most OS are now written in high-level languages
    - C, C++
    - > 90% of Linux code was written in C
    - Portability matters!
  - Major performance improvements in an OS comes from
    - Better data structures and algorithms
    - NOT from using assembly language
      - Because of **advanced compiler techniques**
  - although some performance-critical code are still assembly code…

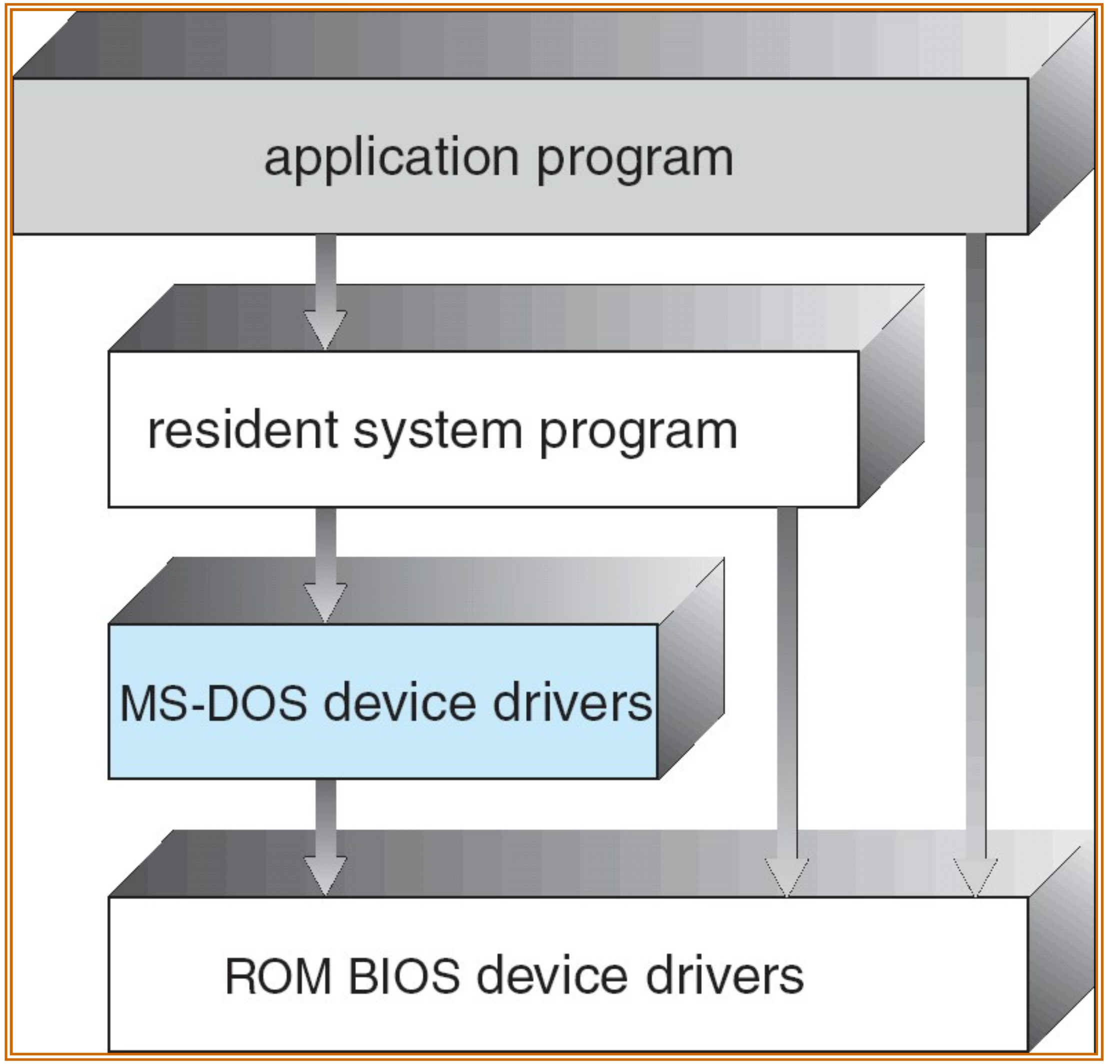
Operating System Structures

- Simple Structure
- Layered Approach
- Microkernels
- Modules

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

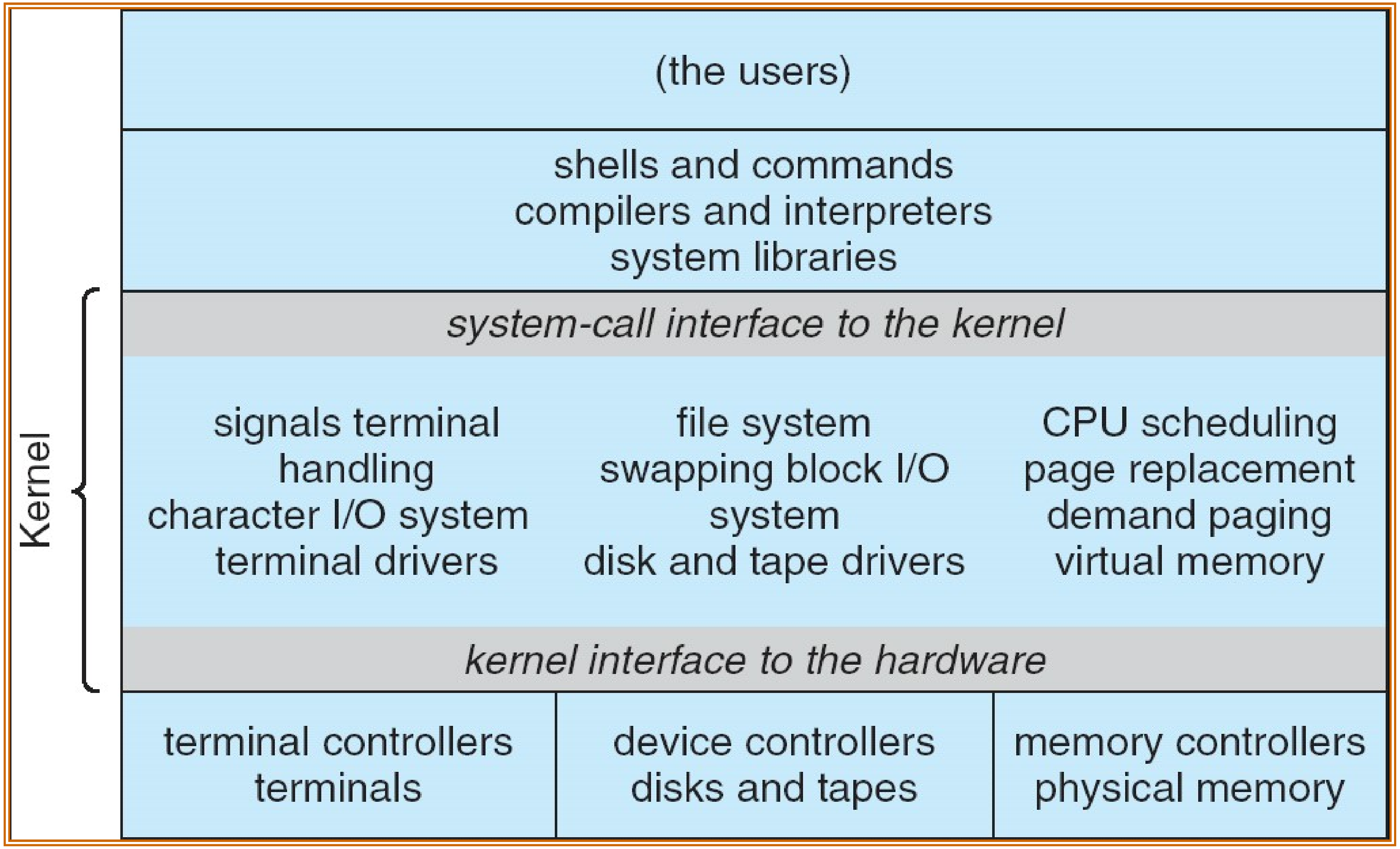
MS-DOS Structure



UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - System programs
  - The kernel
    - Consists of everything below the system call interface and above the physical hardware
    - Provides file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions in one level
      - Monolithic kernel

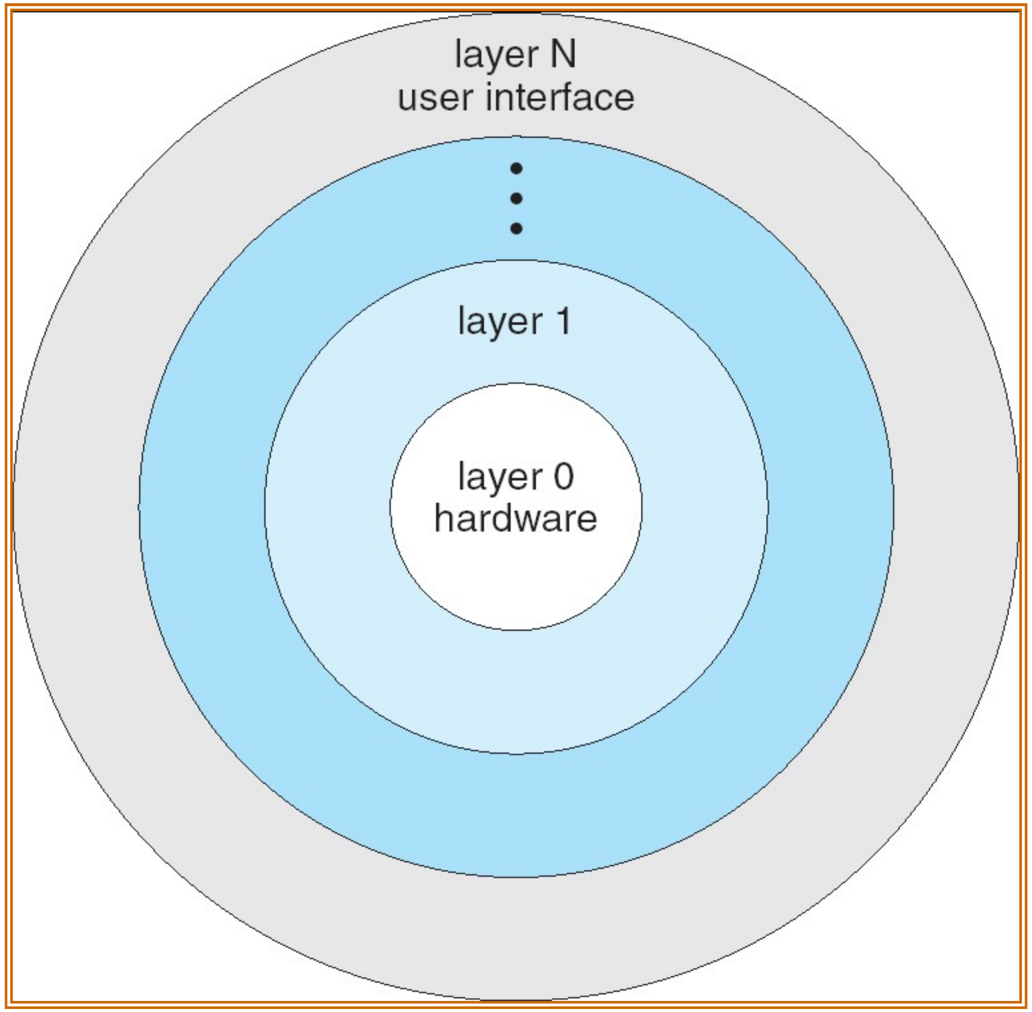
UNIX System Structure



Layered Approach

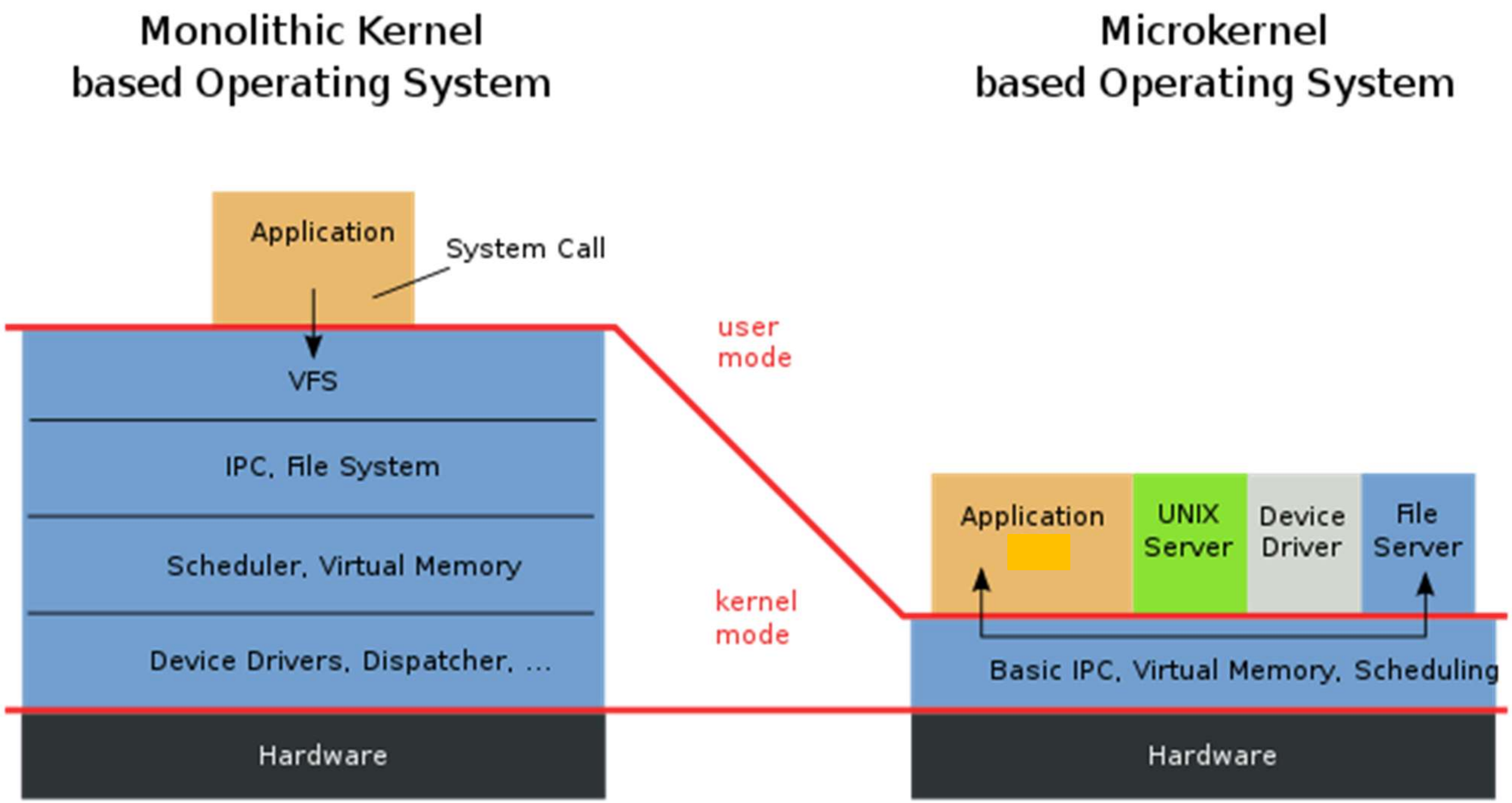
- Like the layers of network protocols, the operating system is divided into a number of layers (levels), each built on top of lower layers.
  - The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each layer invokes operations/services of only the lower-level layer
- Advantages
  - Simplicity of construction and debugging
    - Starts from the lowest layer
  - Do not have to know the details of the other layers
    - Hide the details from the other layers
- Difficulties
  - Hard to define the layers
  - Less efficient
    - A service may cause the crossing of multiple layers
- **Fewer layers** are applied

Layered Operating System



Microkernel System Structure

- Microkernel moves as much from the kernel into ***"user"*** space

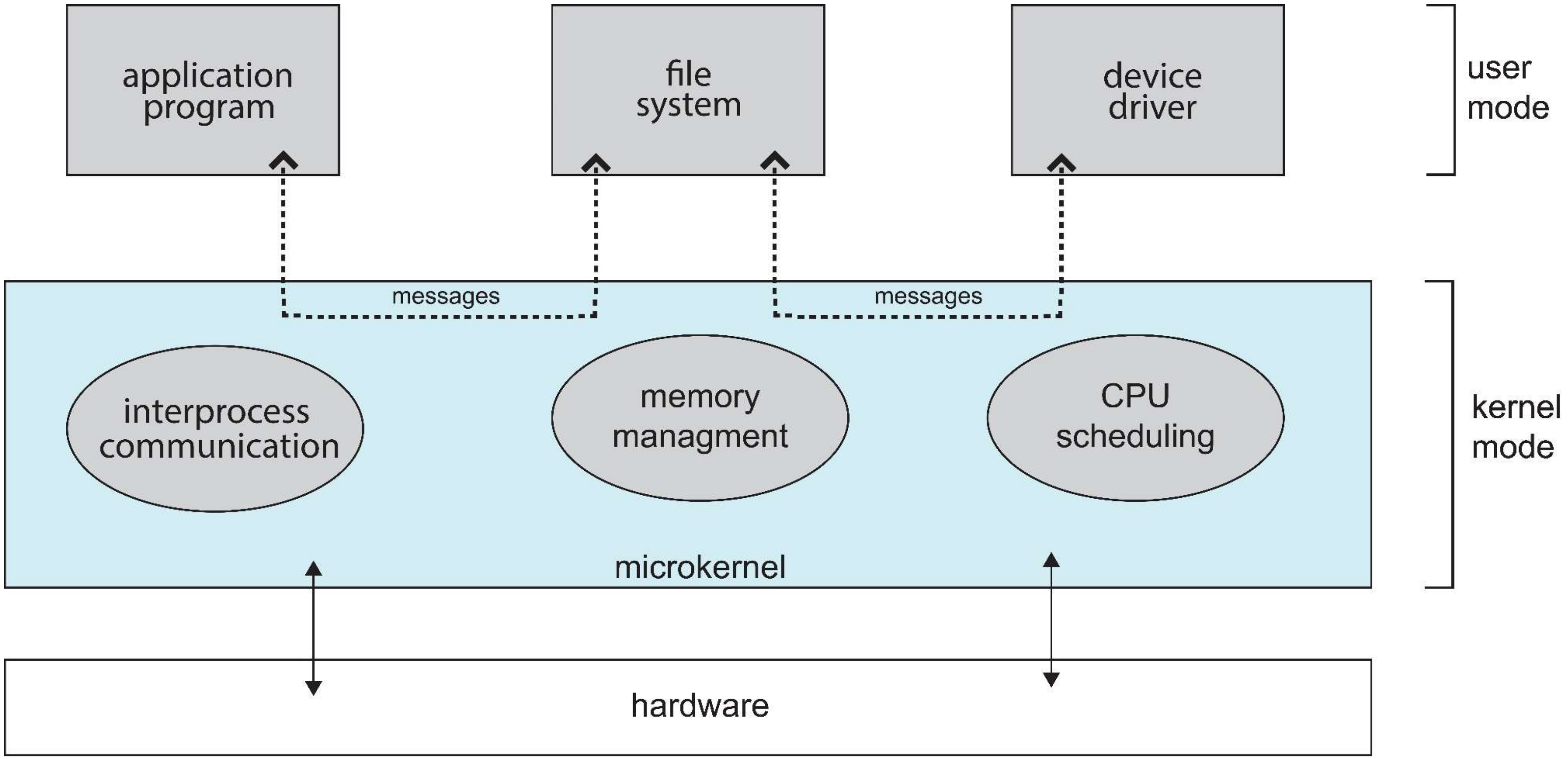


Source: https://en.wikipedia.org/wiki/Microkernel

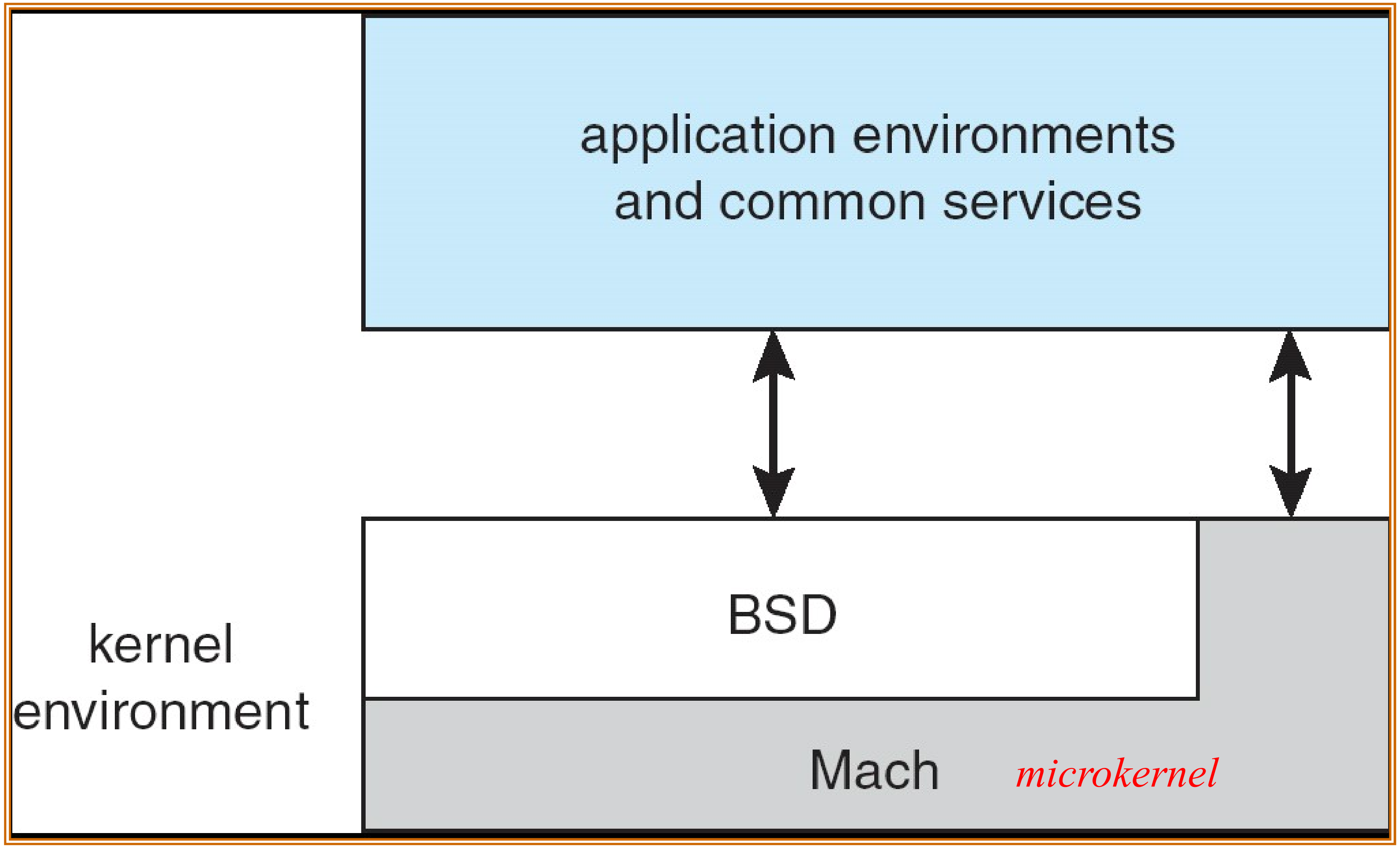
Microkernel System Structure

- Communication takes place between user modules using message passing (see next slide)
- 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
- Drawback
  - **Performance overhead** of user-kernel communication
    - **For inter-subsystem communication (e.g., file system invokes disk driver)**
      - 1 function call + return ➔ 2 system calls and 2 upcalls
    - **Fine grained components ➔ high overhead**

Microkernel System Structure



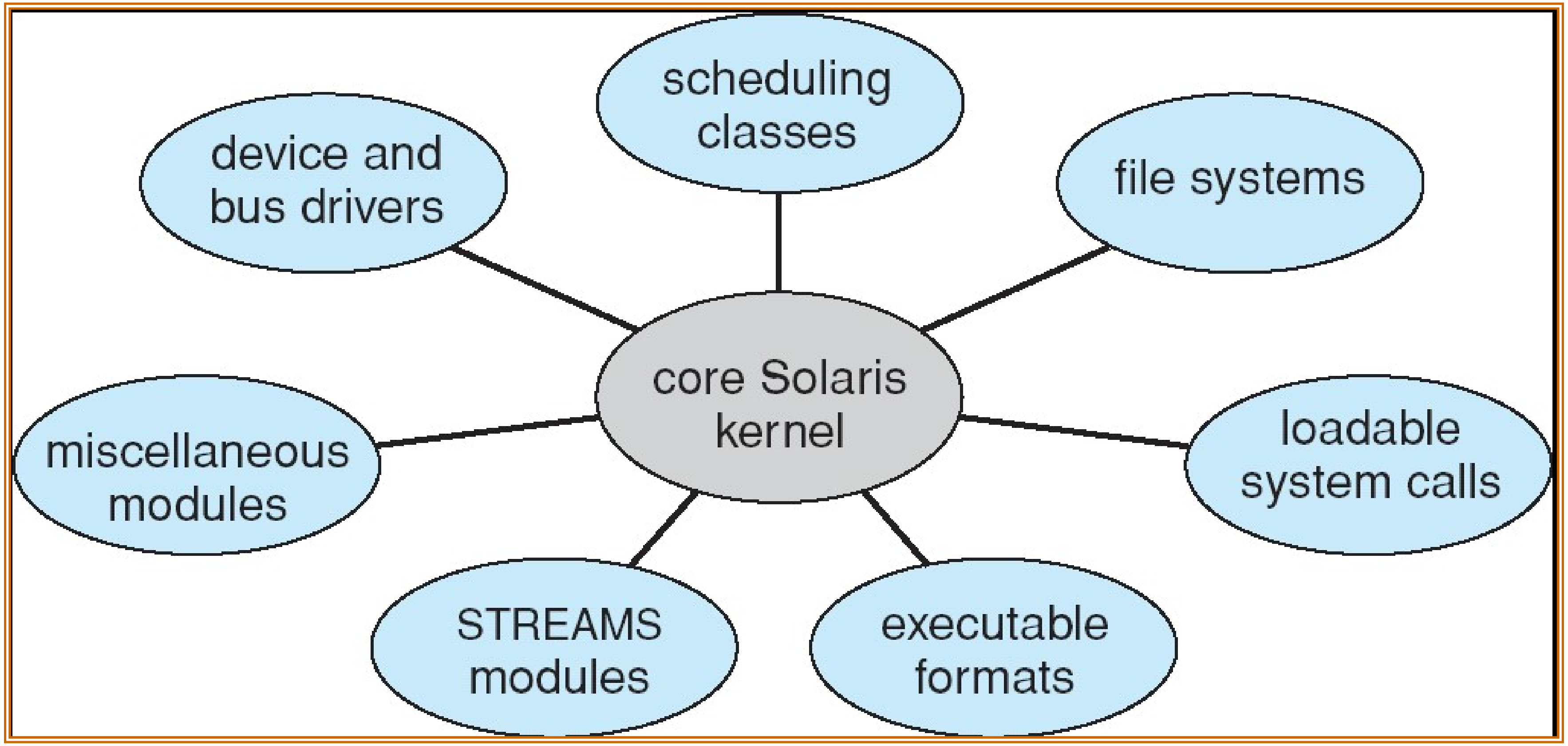
Mac OS X Structure



Modules

- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separated
  - Each talks to the others through known interfaces
  - Each is dynamically loadable as needed within the kernel
- Overall, similar to layers but more flexible

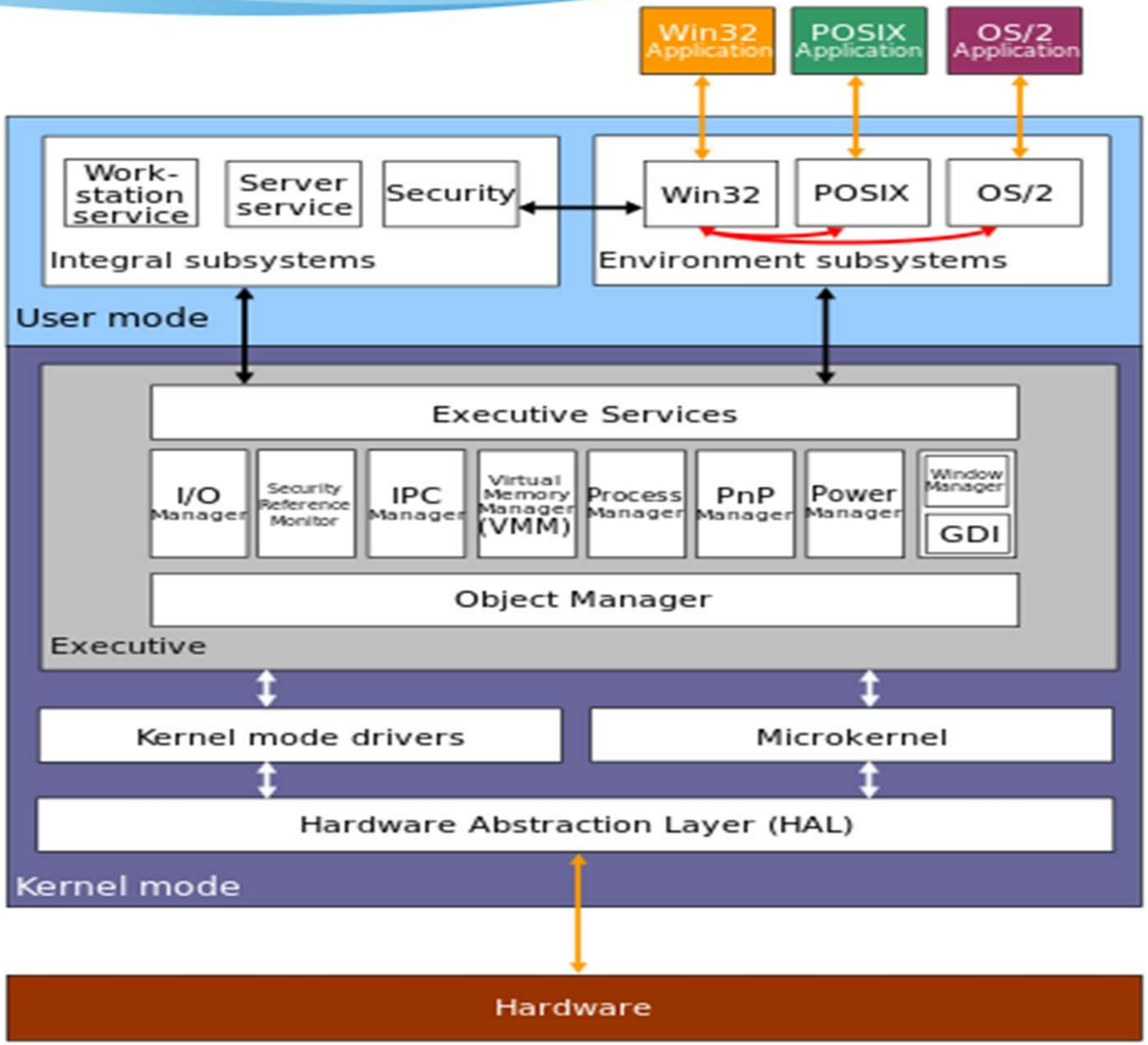
Solaris Modular Approach



Hybrid Systems

- Most modern operating systems do not adopt a single, strictly defined structure
  - **Hybrid** systems combine multiple structures to address *performance*, *usability, flexibility…*
  - Linux is **monolithic**,
    - + **modular** for dynamic loading of functionality
  - Windows is mostly **monolithic**
    - + **microkernel** for different ***personalities (user-level services)***
    - + **modular** for dynamic loading of functionality

Windows NT Kernel



Source : https://en.wikipedia.org/wiki/Hybrid_kernel#NT_kernel

Virtual Machines

- Multiple virtual machines on a physical machine

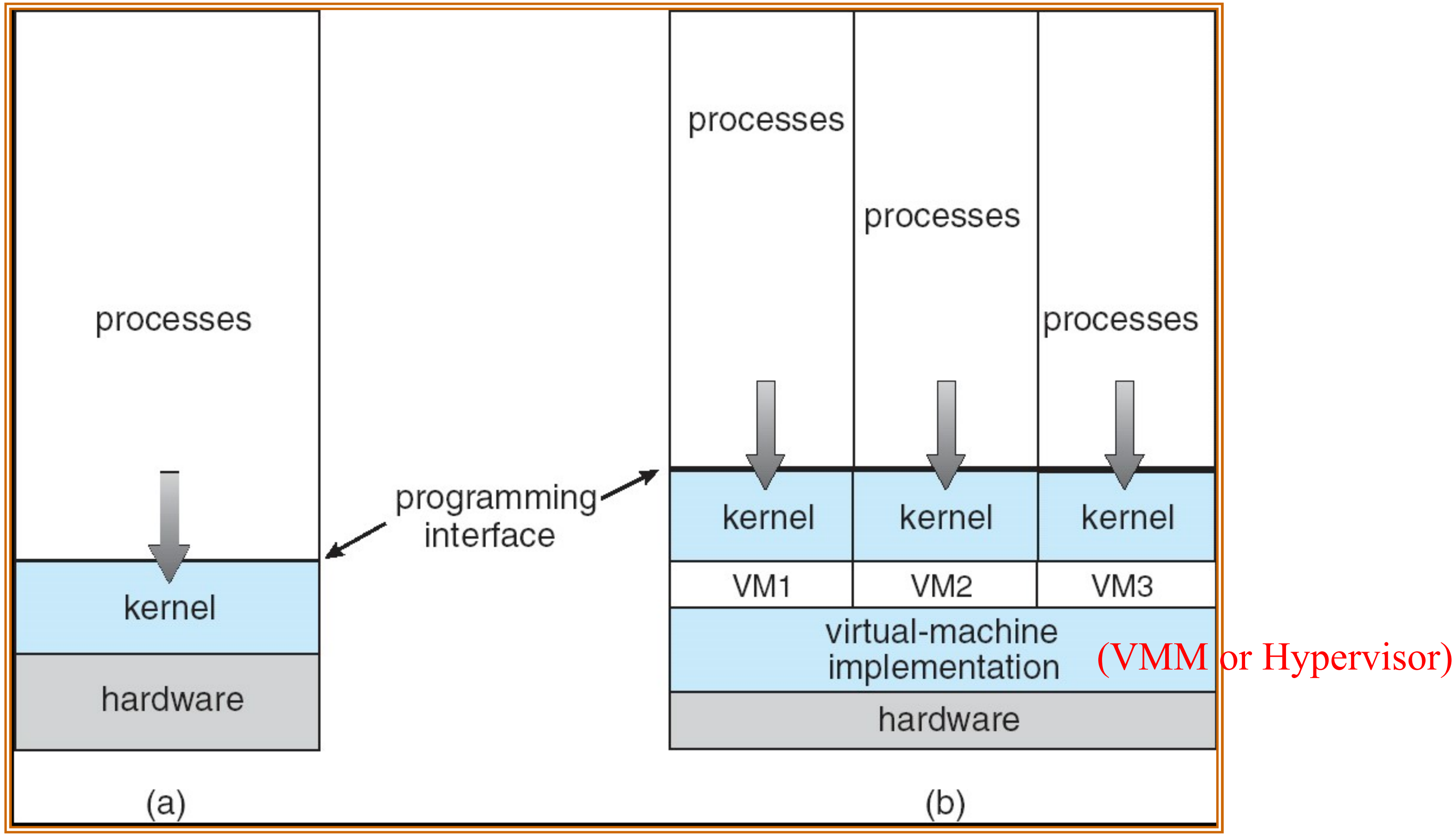
- A virtual machine (VM) provides an interface identical to the underlying bare hardware
  - Each VM can have its own operating system

- The resources of the physical computer are shared by the VMs
  - CPU scheduling can create the appearance that VMs have their own processors

Virtual Machines (Cont.)

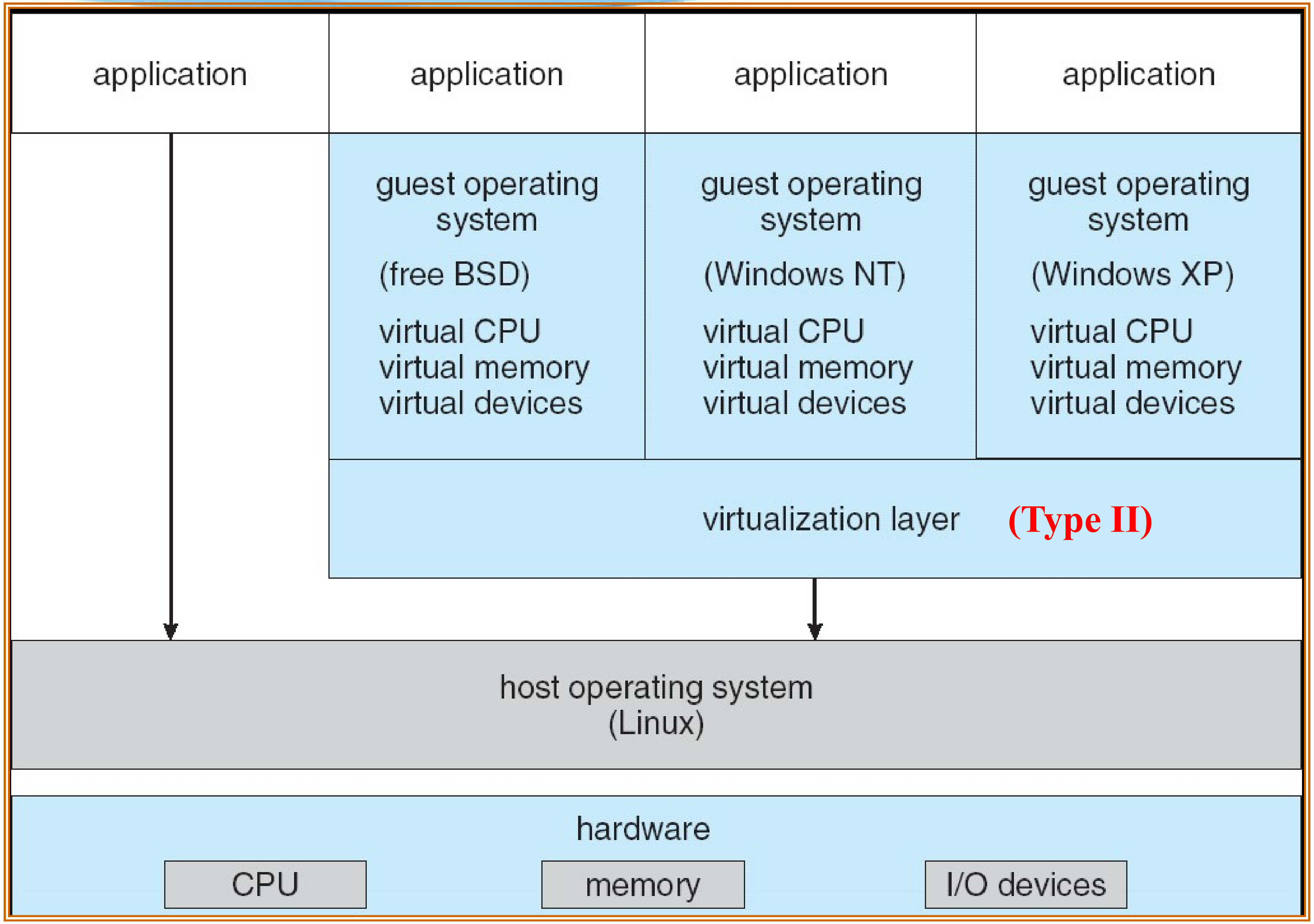
- **Benefits**
  - machine consolidation (ease management, reduce cost…)
  - useful for operating-systems research and development
    - System development can be done on virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
    - a VM provides complete protection of system resources. Each VM is isolated from the other VMs.
  - remain available during hardware maintenance/upgrade
    - The VM providing services can be migrated to another physical machine

Virtual Machines (Cont.)

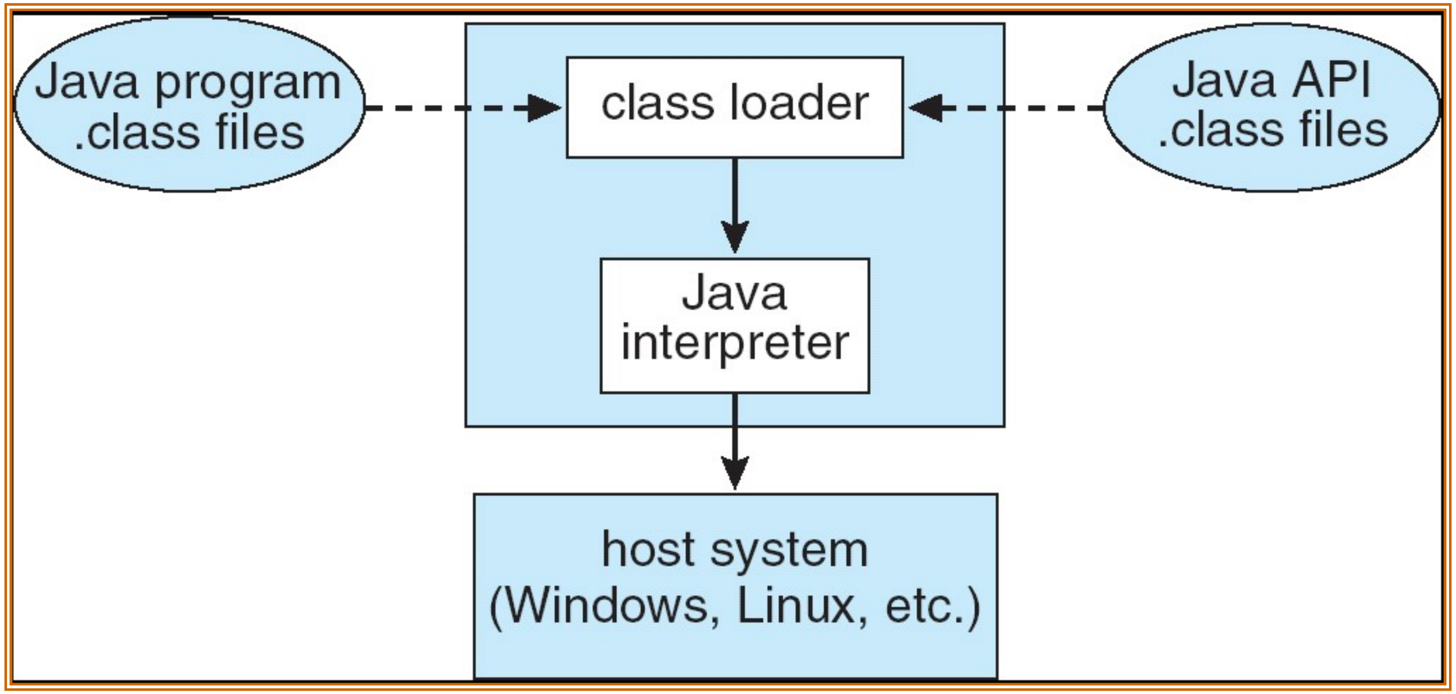


Non-virtual Machine **Virtual Machine (Type I)**

VMware Architecture - Workstation



The Java Virtual Machine



.JIT (Just-in-time) Compiler

.Automatic memory management
- Garbage collection