

Chapter 2: Operating System Structures

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

Operating System Services

- User Interface (UI)
- Program execution:
- I/O operations
 - User programs cannot execute I/O operations directly, so the operating system must provide means to perform I/O
- File-system manipulation
- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network.
 - Implemented via shared memory or message passing

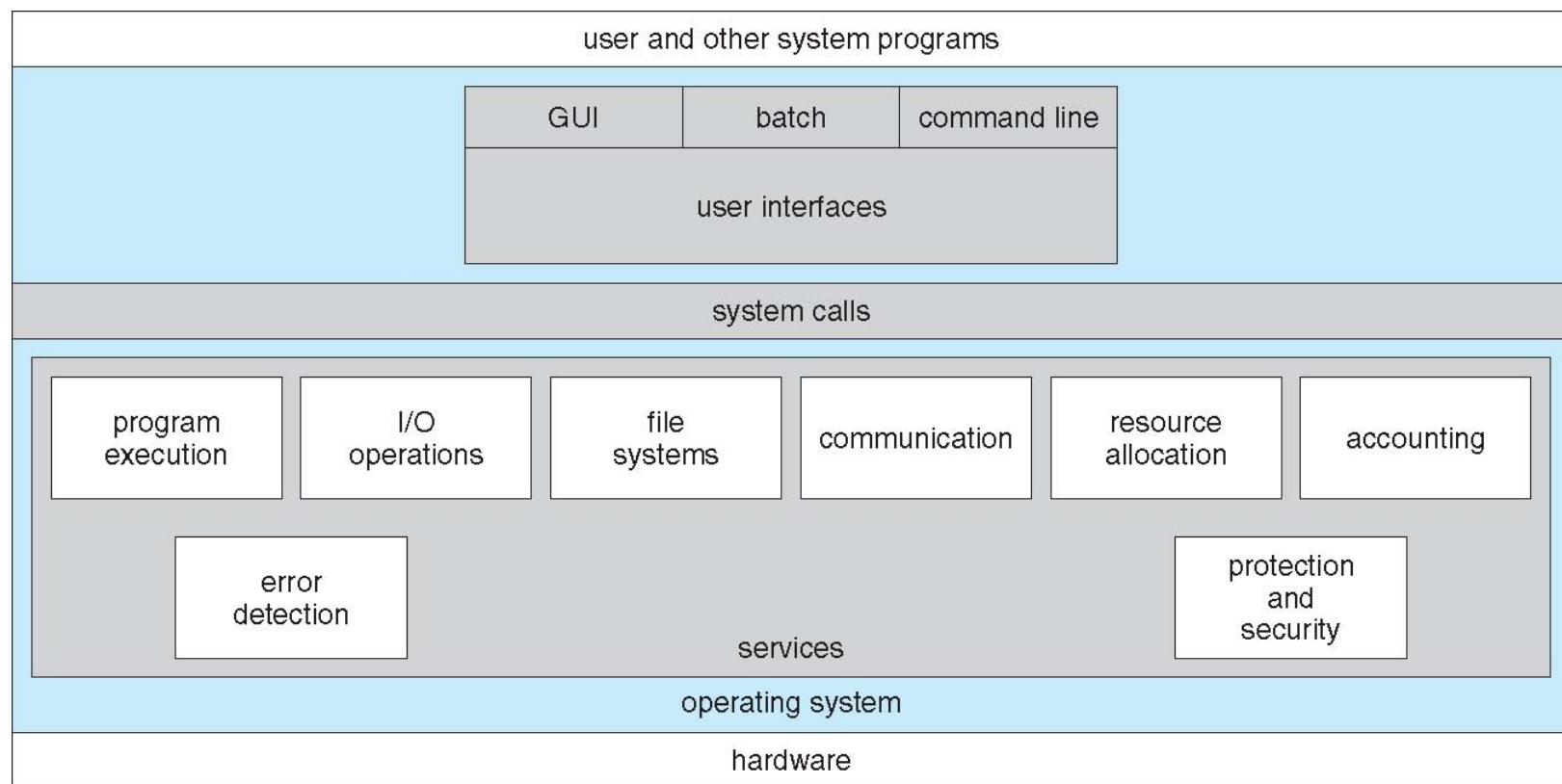
Operating System Services

- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.
 - provides debugging facilities to help track down bugs

Additional functions exist not for helping the user, but rather for ensuring efficient system operations:

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.
- Accounting – keep track of and record which users use how much and what kinds of computer resources

A View of Operating System Services



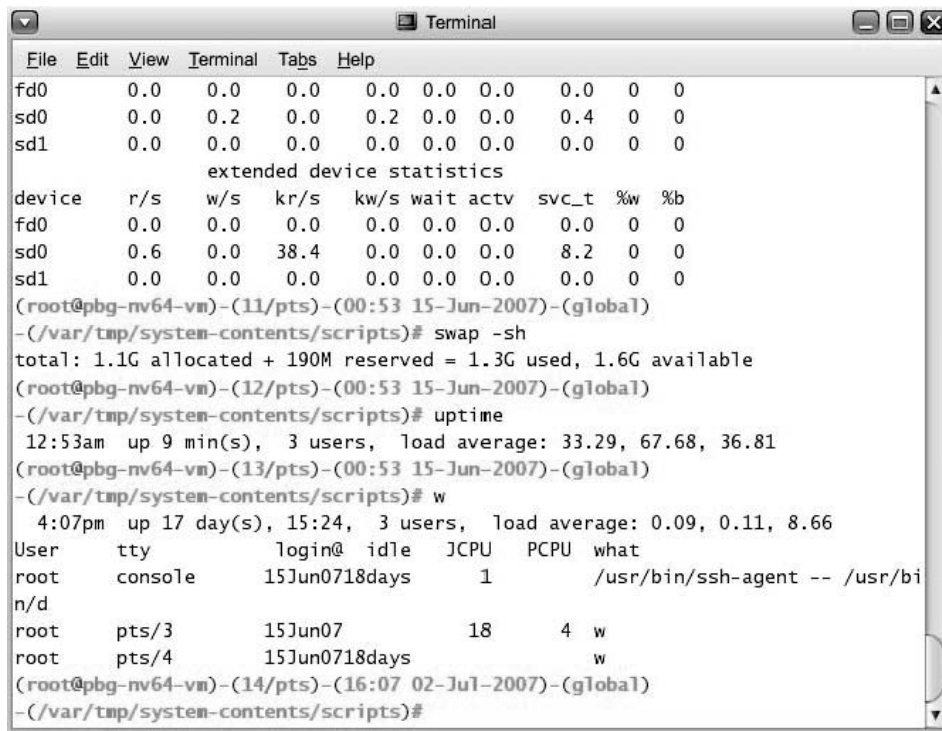
User Operating System Interface – Command-Interpreter System

- Many commands are given to the operating system by control statements typed at the keyboard (for example)
- The program that reads and interprets control statements is called variously:
 - command-line interpreter (CLI)
 - Shell (UNIX)
- Its function is to get and execute the next command statement

User Operating System Interface – Graphical User Interface (GUI)

- User-friendly desktop metaphor interface
 - Usually mouse, keyboard, and monitor
 - Icons represent files, programs, actions, etc
 - 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 has “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

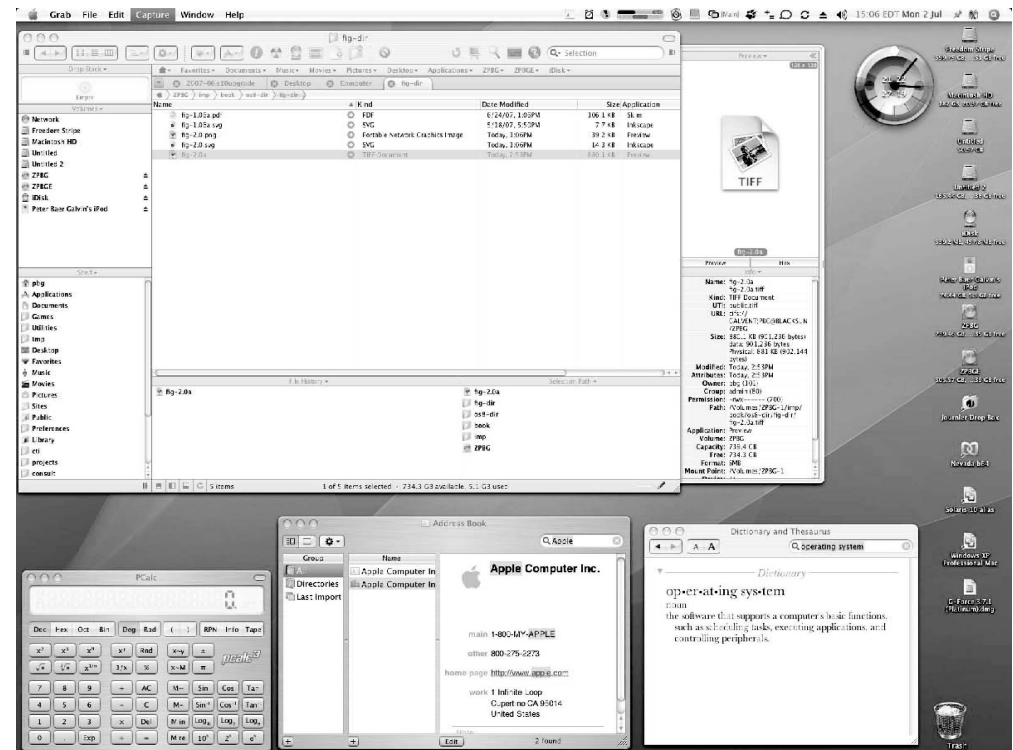
Bourne Shell CLI vs. Mac OS/X GUI



```
File Edit View Terminal Tabs Help
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.0 0.2 0.0 0.2 0.0 0.0 0.4 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0

extended device statistics
device r/s w/s kr/s kw/s wait actv svc_t %w %b
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.6 0.0 38.4 0.0 0.0 0.0 8.2 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0

(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User      tty      login@ idle JCPU PCPU what
root      console  15Jun0718days 1      /usr/bin/ssh-agent -- /usr/bi
n/d
root      pts/3    15Jun07      18      4 w
root      pts/4    15Jun0718days      w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#
```



System Calls

- System calls provide the interface between a running program and the operating system
- 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)
 - Java API for the Java virtual machine (JVM)

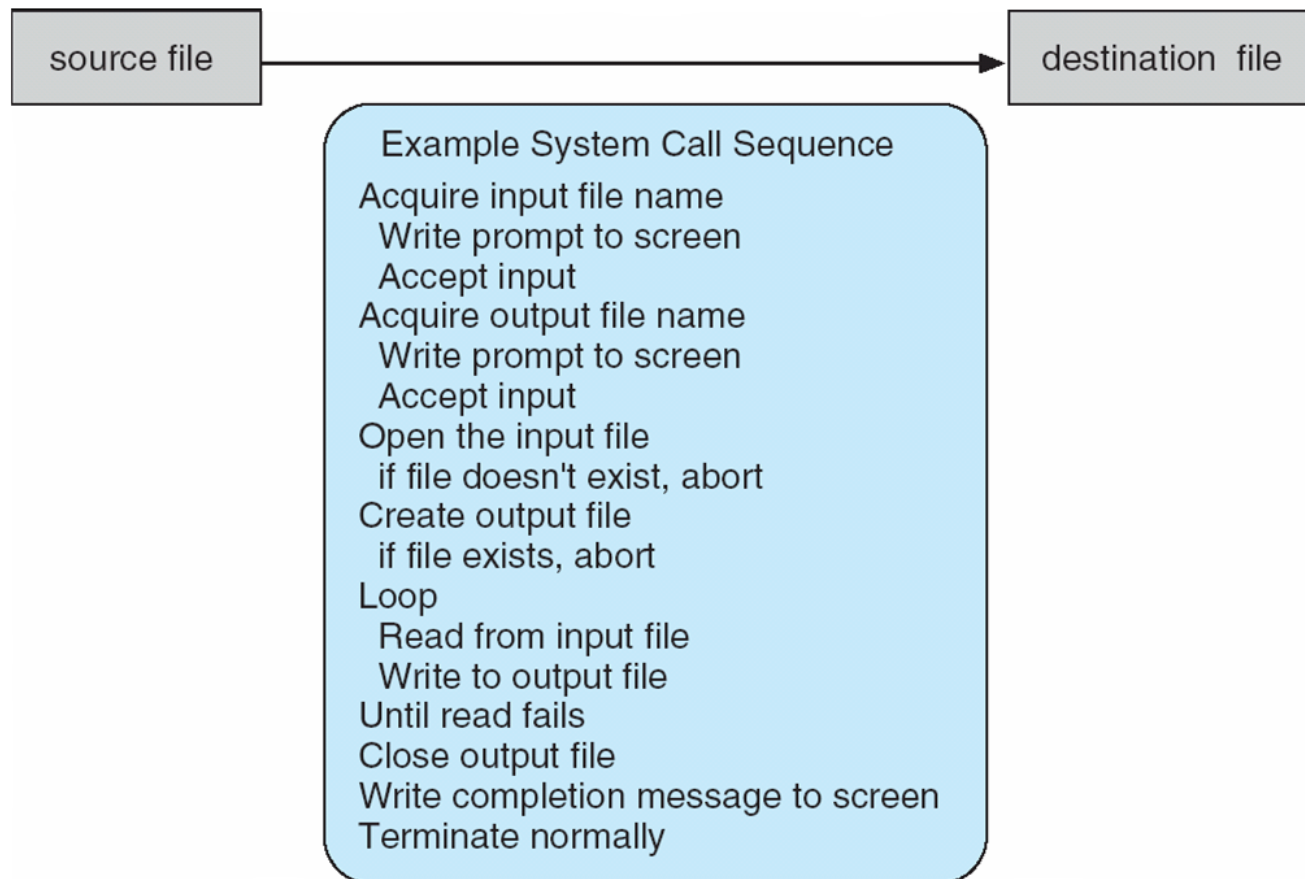
System Calls

- Why use APIs rather than system calls?
 - Allows programs involving system calls to work on multiple systems.

- Types of system calls:
 - Process control
 - File management
 - Device management
 - Information Maintenance
 - Communications
 - Protection

Example of System Calls

➤ System call sequence to copy the contents of one file to another file:



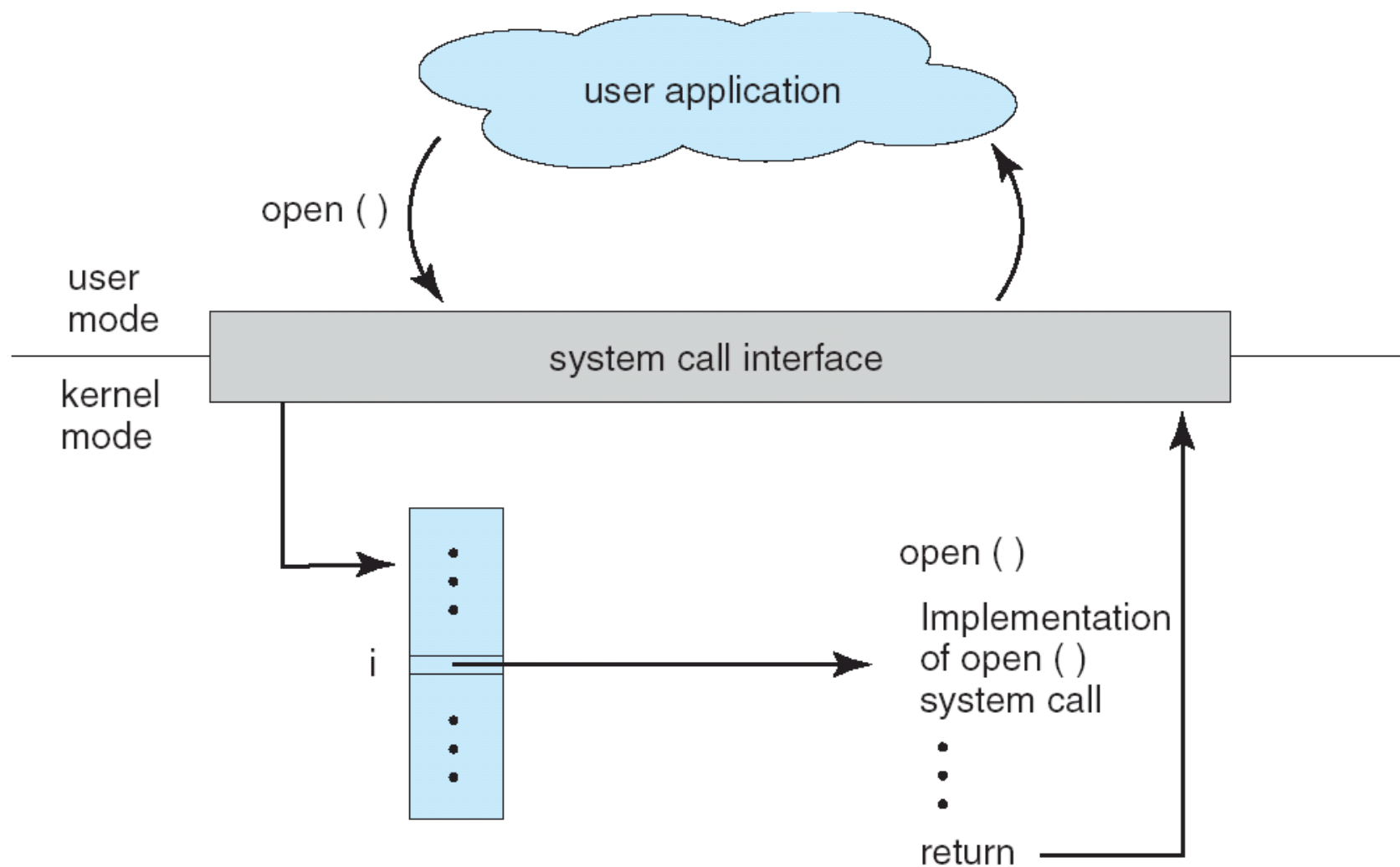
Examples of Windows and Unix System Calls

	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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Implementation

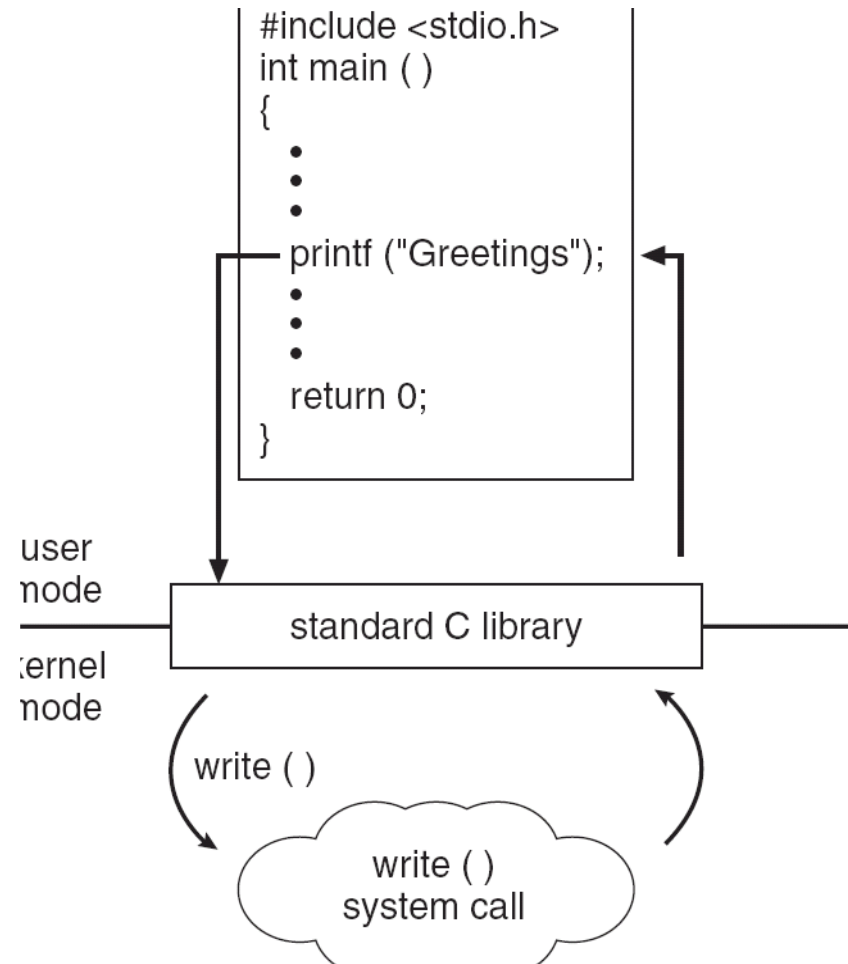
- Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how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

API – System Call – OS Relationship



Standard C Library Example

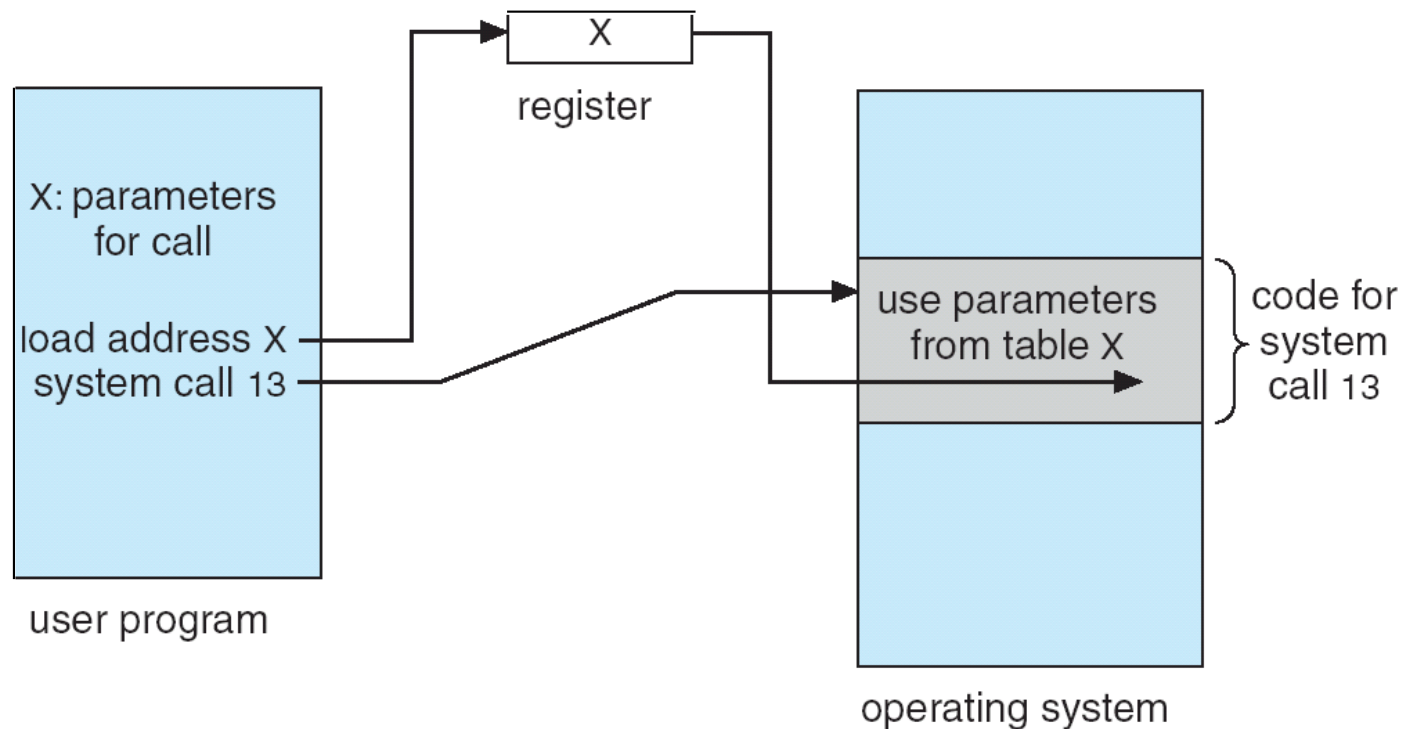
- C program invoking *printf()* library call, which calls *write()* system call



System Calls – Parameter Passing

- Three general methods are used to pass parameters between a running program and the operating system:
 - Pass parameters in **registers**
 - Store the parameters in a **table** in memory
 - Reserve a place in memory to store parameters in.
 - The table address is passed as a parameter via a register
 - Use a **stack**
 - push parameters onto the stack, and pop them off the stack in the system call function
- Table and stack methods do not limit the number of parameters

Parameter Passing via Table



System Programs

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

Operating System Design & Implementation

➤ *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
- flexible, reliable, error-free and efficient

Operating System Design & Implementation

➤ Important principle to separate

Policy: what will be done?

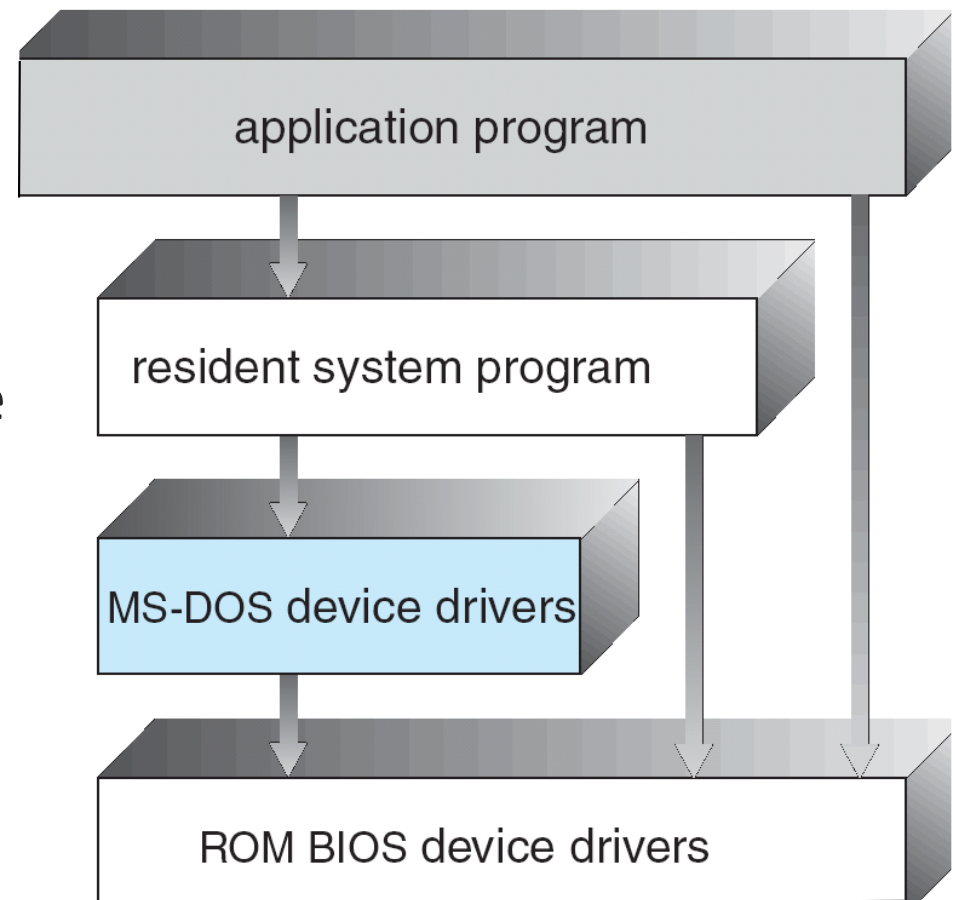
Mechanism: how to do it

➤ Why have this separation?

➤ allows flexibility if either policy or mechanism changes

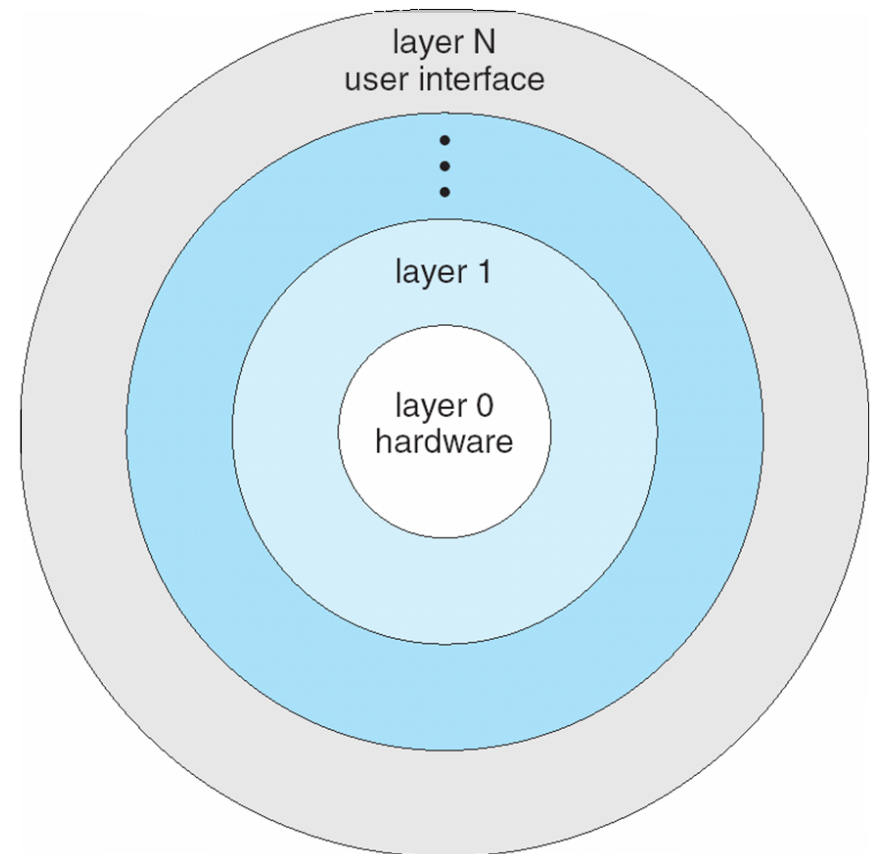
Simple Structure

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



Layered Approach

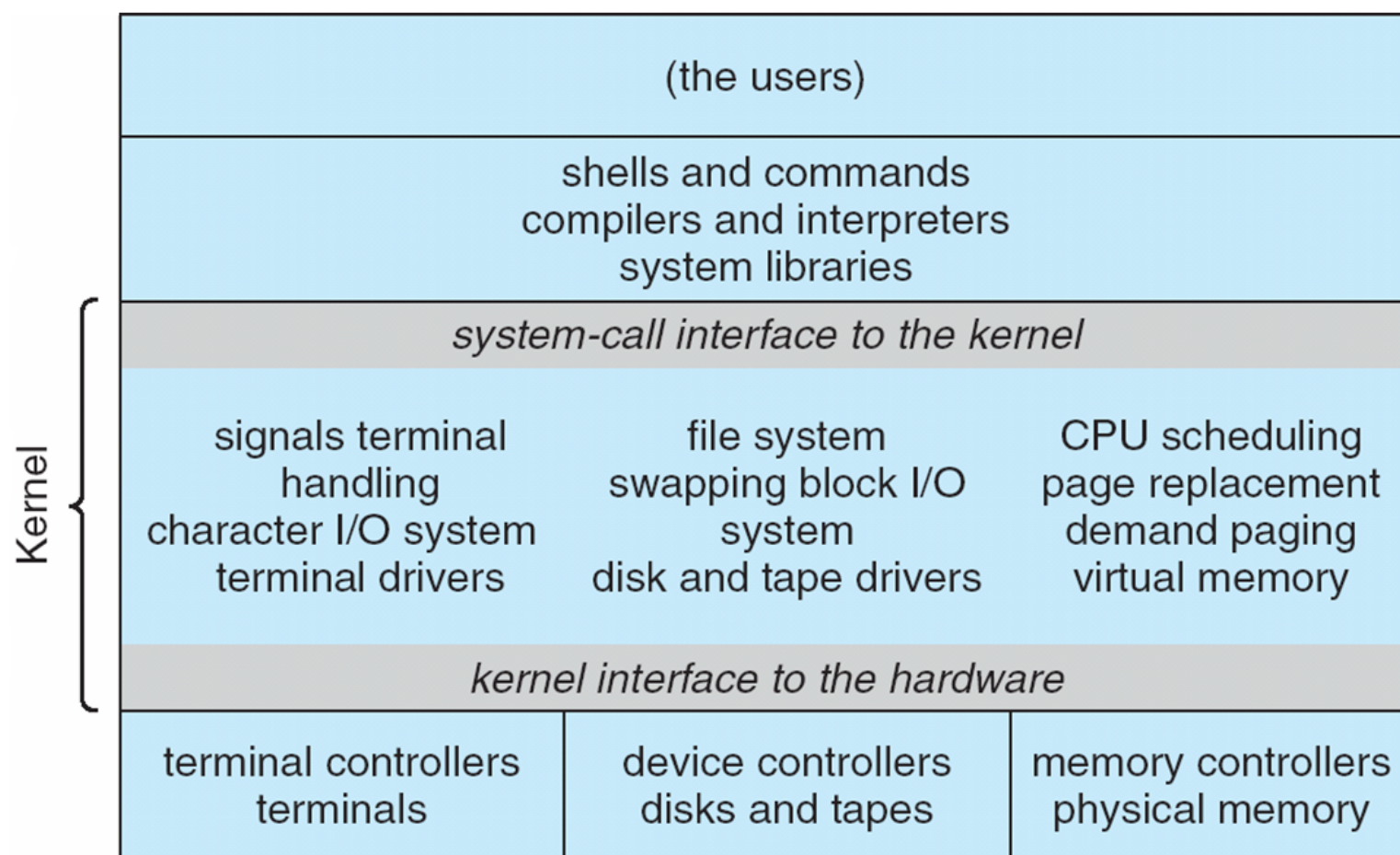
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions and services of only lower-level layers



UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system was a *monolithic kernel*.
- The UNIX OS consists of two separable parts:
 - Systems programs monolithic = most of the functionality within the kernel
 - 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

Traditional UNIX Structure



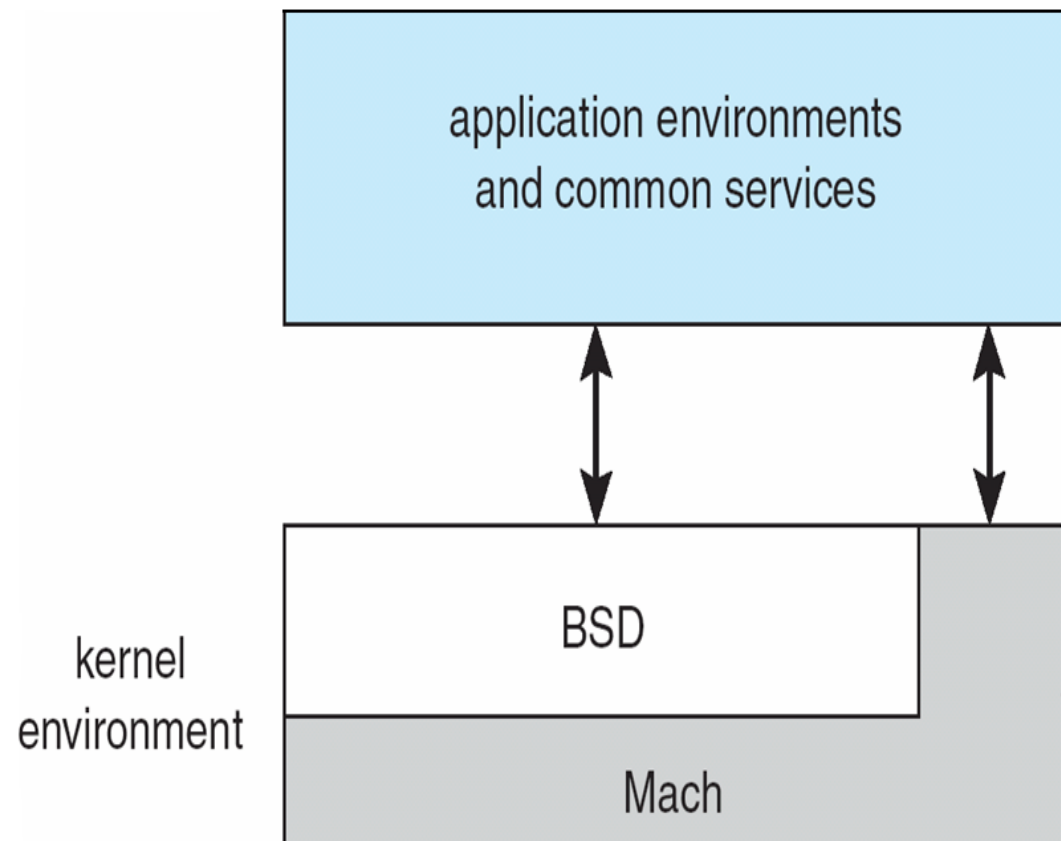
Microkernel System Structure

- Moves as much from the kernel into “*user*” space
- Benefits:
 - easier to extend OS (add functionality to it)
 - easier to port OS to a new architecture
 - More reliable (less code running in kernel mode)
 - More secure (most services run as user rather than kernel)
- Detriments:
 - Performance overhead of user space to kernel space communication

Monolithic vs. Microkernel Structure

➤ Most popular modern OSes are actually hybrids of the monolithic and microkernel structures:

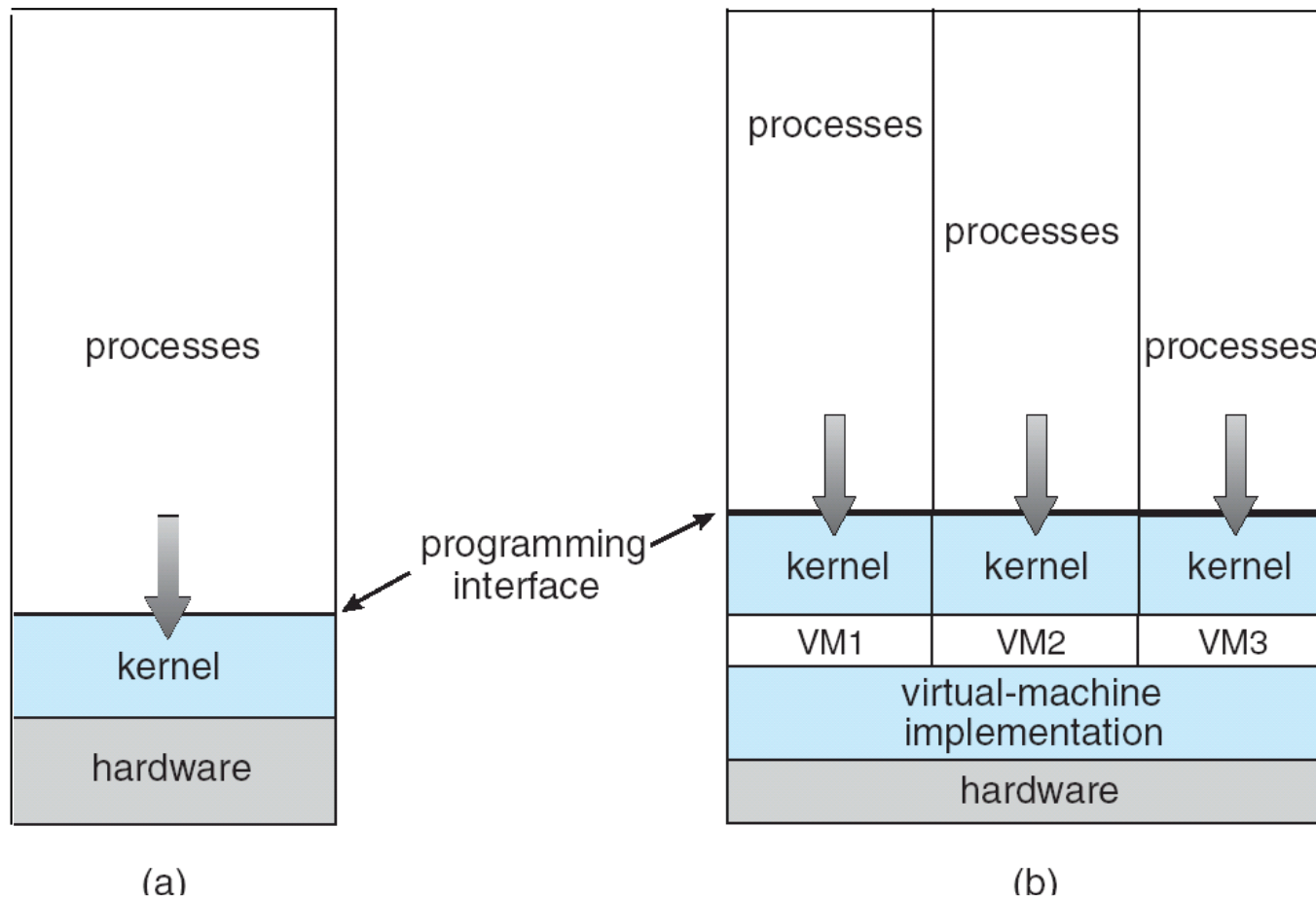
- Many functions are moved into “user” space
- Some are kept in the kernel for performance reasons
- E.g. Mac OS/X structure:



Virtual Machines

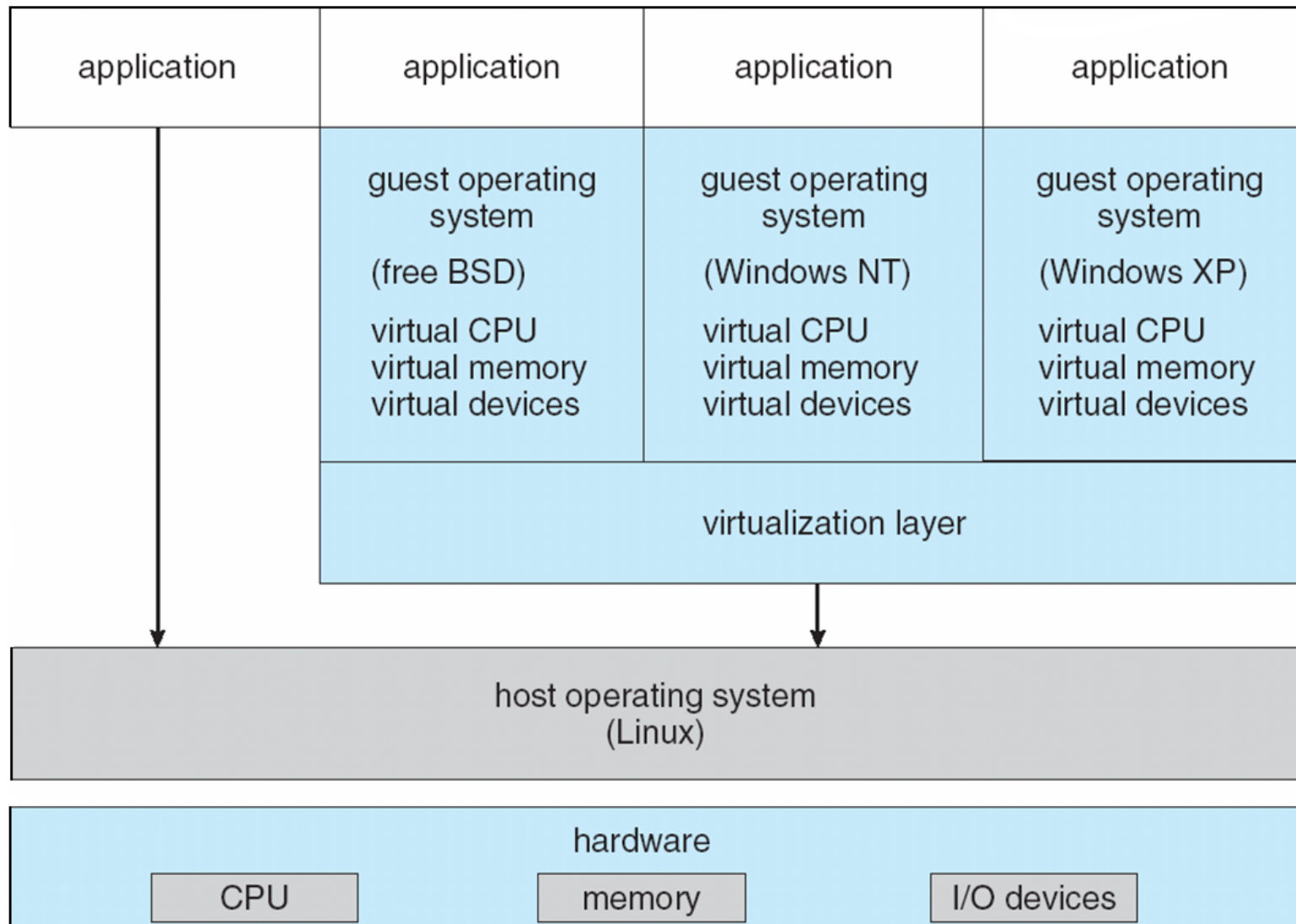
- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface identical to the underlying hardware
- The operating system host creates the illusion that a process has its own processor and memory
- Each guest is provided with a (virtual) copy of underlying computer

Virtual Machines

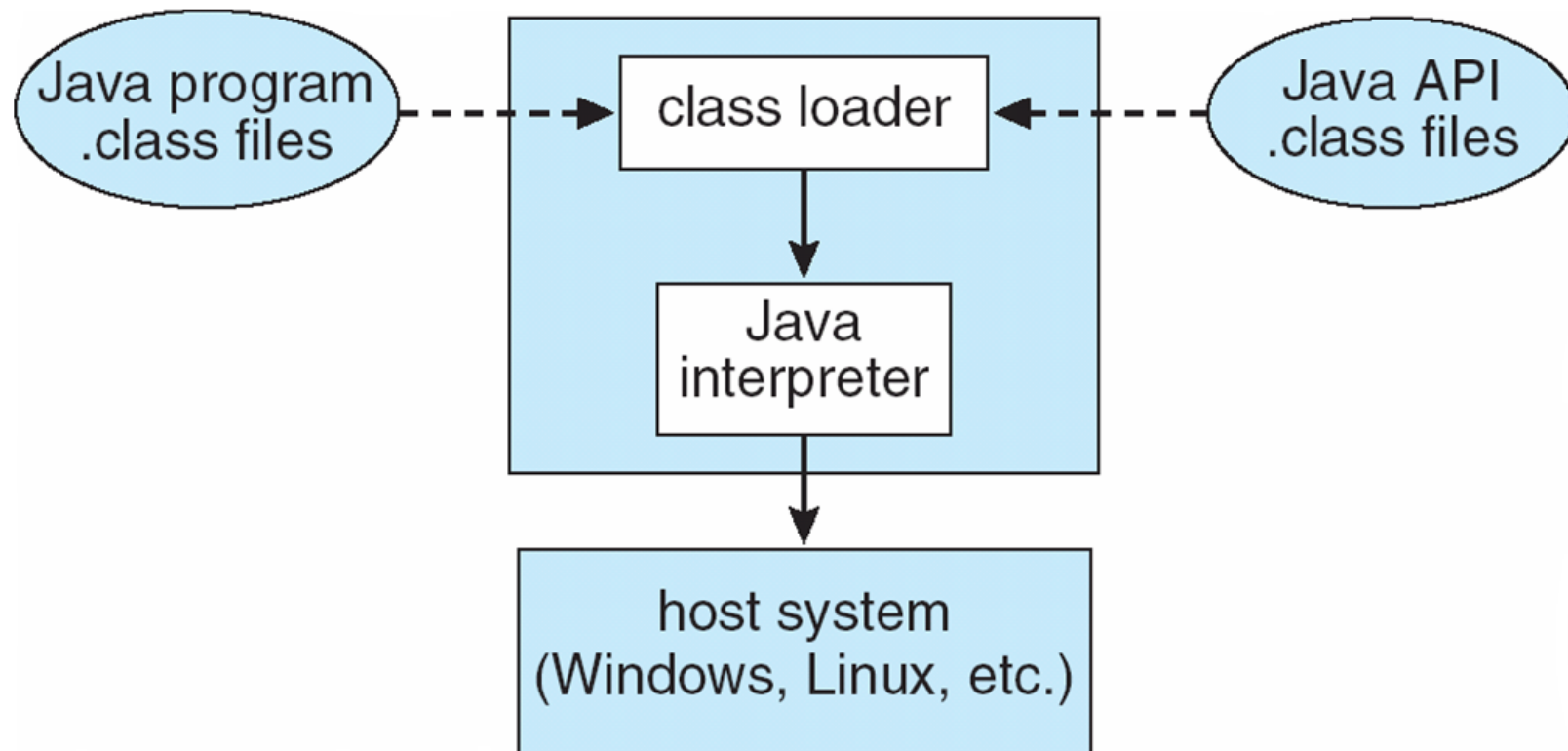


(a) Nonvirtual machine (b) virtual machine

VMWare Architecture



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
- 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."*

System Boot

- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM (firmware) that is able to:
 - locate the kernel,
 - load it into memory,
 - and start its execution