Introduction

Operating Systems

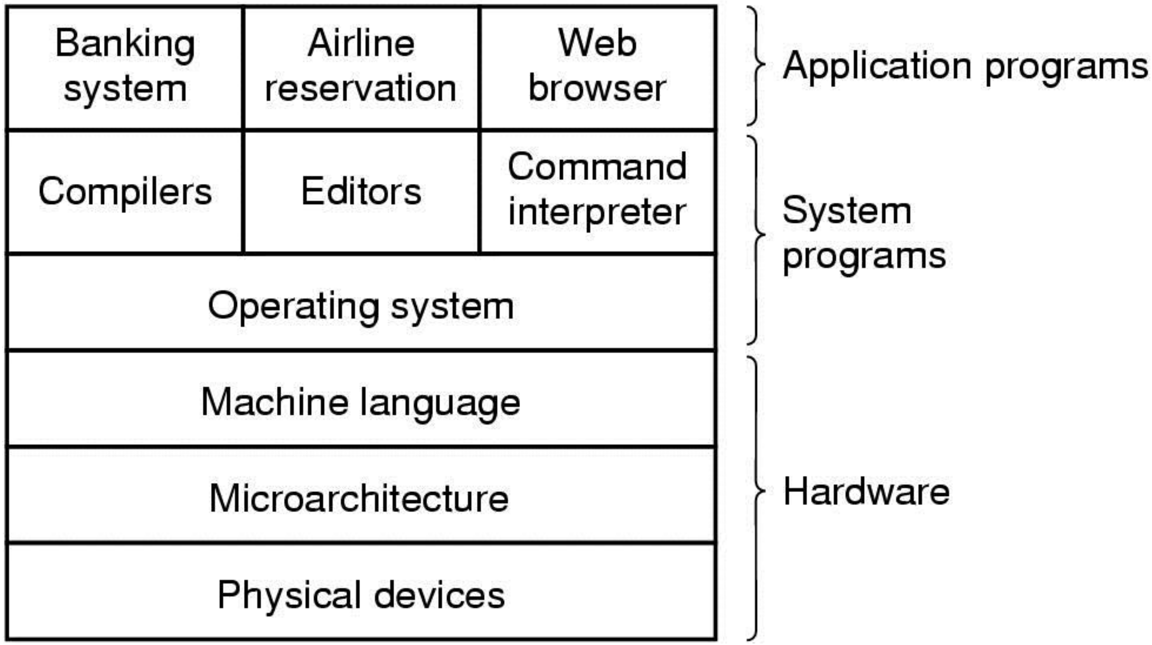
Chapter 1

- Read chapter 1…

Introduction

- Computer system can be divided into four components
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

Computer System



- Layers of a computer system
  - hardware
  - system programs
  - application programs

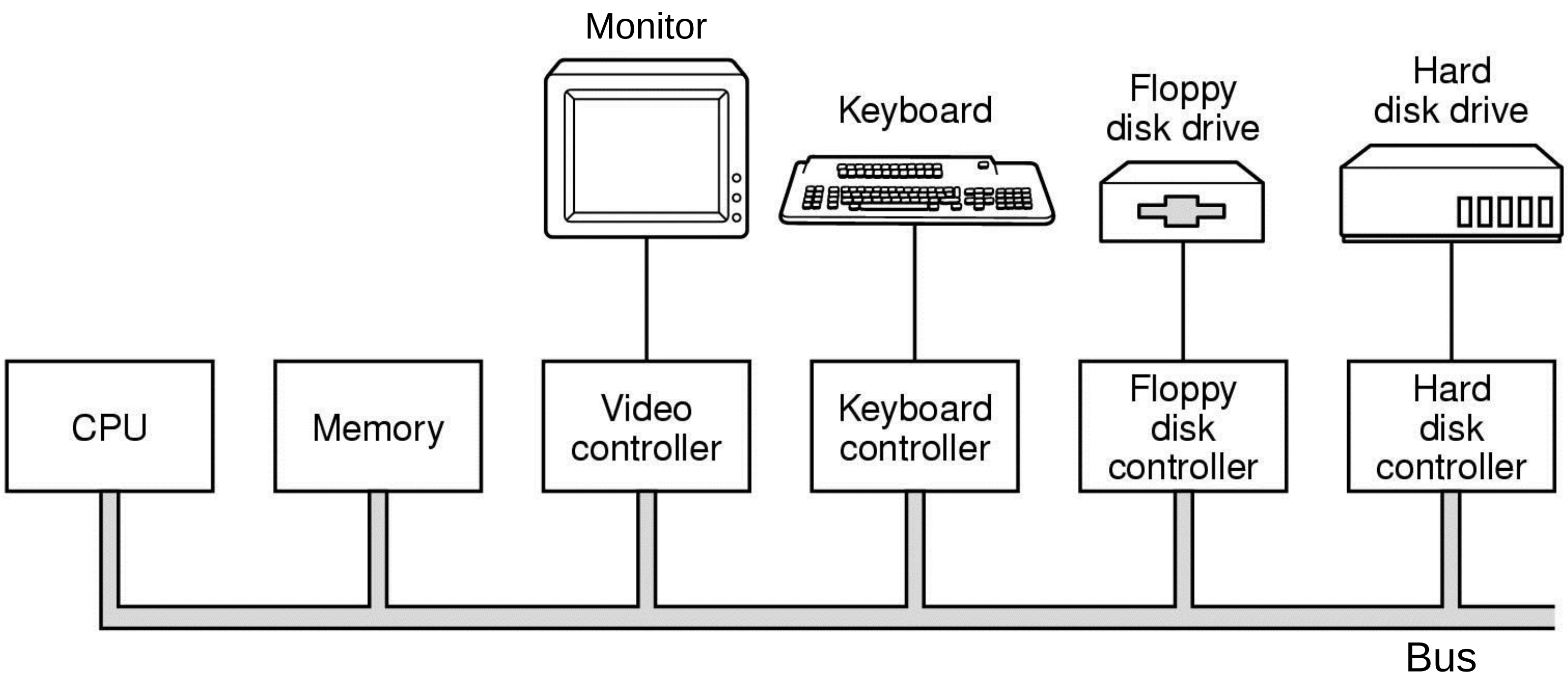
What is an Operating System

- OS is a program that acts as an intermediary between a user of a computer and the computer hardware
- It is a resource manager
  - Each program gets time with the resource
  - Each program gets space on the resource
- It is also an extended, virtual machine
  - Hides the messy details which must be performed
  - Presents user with a virtual machine, easier to use

Computer Hardware review

- Computer organization
  - Processor
  - Caching and storage hierarchy
  - I/O structure and devices
  - Hardware protection

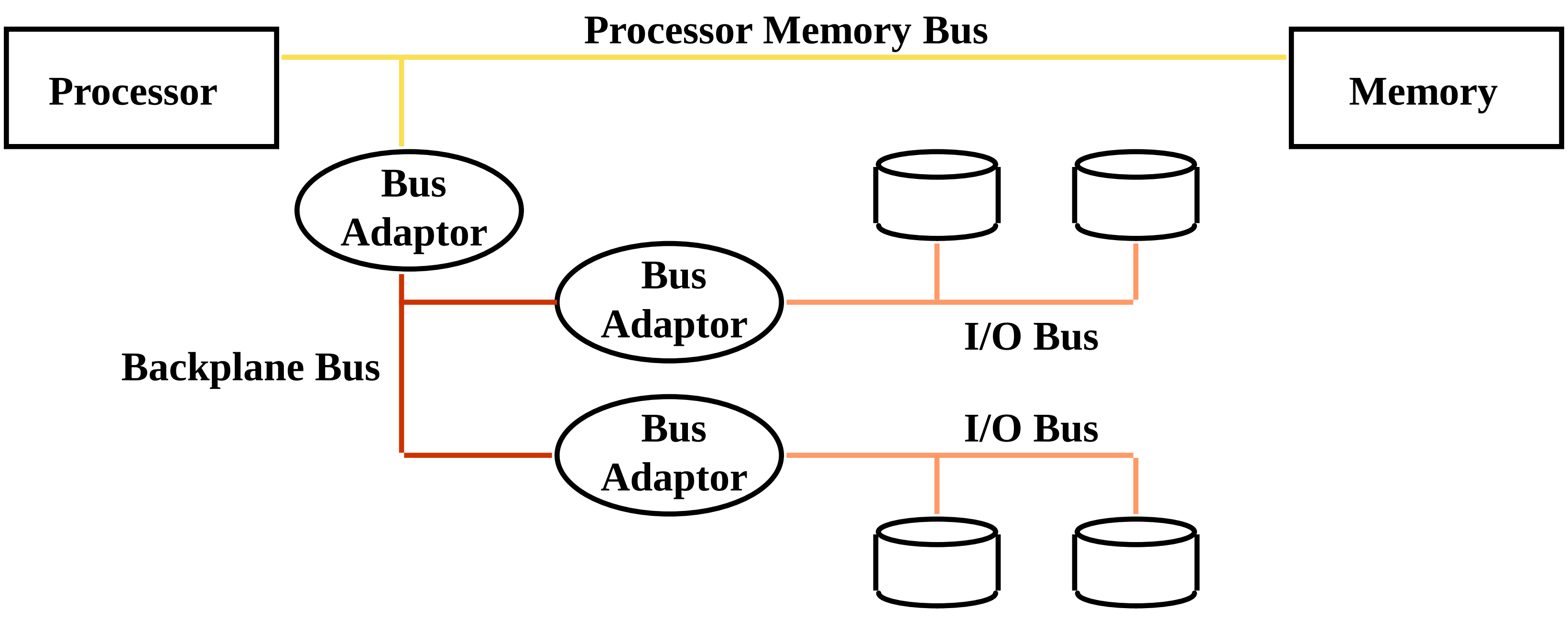
Computer Hardware Review



- Old system architectures: typical components of a simple personal computer (~1980’s)
  - Everything connected to a single bus.

Computer Hardware Review

- More modern computers have multiple levels of busses to accommodate the large variation in speeds and interfaces.

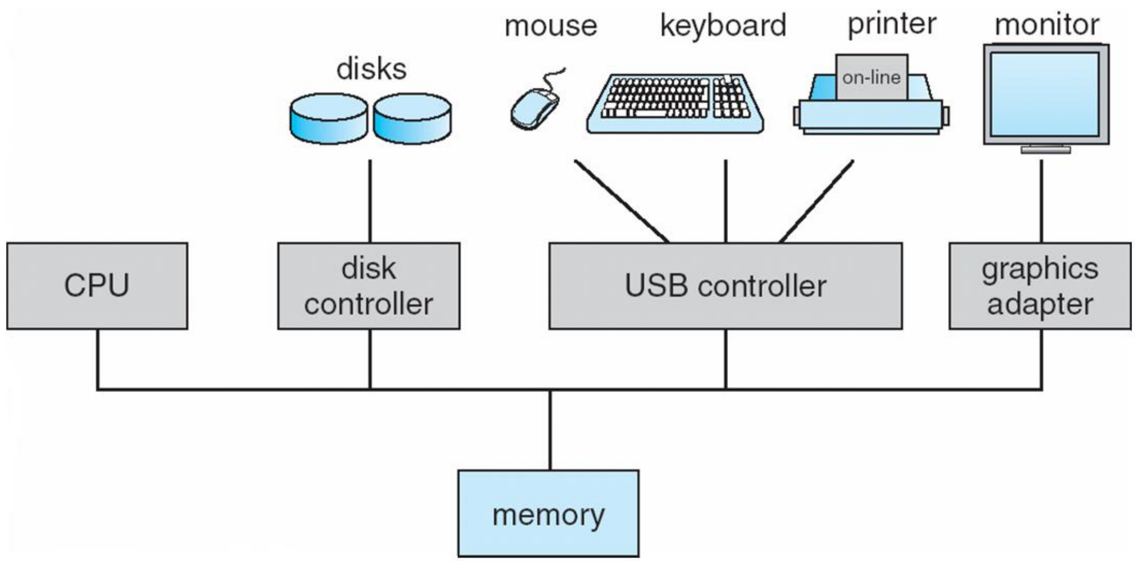


Computer System Operation

- Device controllers provide the interface between the low level details of the hardware and the OS
- Each particular device would have a device controller that the OS communicates with.

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupt Structure

- Interrupt structures were invented by computer scientists to switch between resources.
- Interrupt performs unconditional jump to “the interrupt service routine”
- Service routine located at a fixed address provided by an interrupt vector
- Hardware and software must save address of interrupted instruction
- Other important register contents also saved

Direct Memory Access (DMA) Structure

- Device controller transfers block of data to/from device from/to memory
- CPU initiates the transfer and stays out of it afterwards
  - Starting address, length, op code, device
  - no further intervention from CPU.
  - Transfer not just to local buffers but into memory directly.
  - Interrupt raised after DMA is complete
- Cycle stealing
  - As the controller is transferring data, it needs to use the bus, so it steals cycles from CPU to do this.
- Modern paged virtual memory systems rely on this heavily.

Basis for Hardware Protection

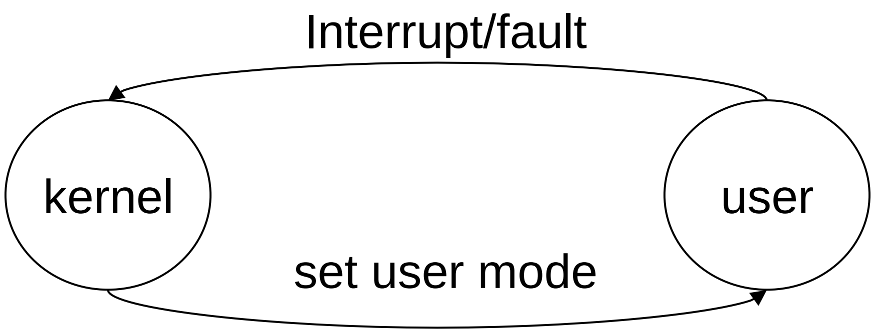
- Dual-mode operation with privileged instructions
  - User vs. kernel modes
- I/O operations, interrupts, interrupt vectors
  - Done by privileged instructions (in kernel mode)
- Memory protection, base/bounds or keys, page tables
  - Protected address spaces for various processes.
- CPU protection
  - By masking interrupts and having privileged modes.

Dual-mode Operation

- Sharing of system resources without corruption
- User mode — computation consumes input/produces results in memory. Execution of privileged instructions causes interrupt.
- Kernel mode — performs I/O and management activities in protected operating system mode. Executes privileged instructions.

Dual-Mode Operation (Basis for hardware protection)

- Mode bit added to computer hardware (usually in the program status word – PSW) to indicate the current mode: monitor/kernel/privileged mode (0) or user mode (1).
- When an interrupt or fault occurs hardware switches to monitor/kernel/privileged mode.



- Privileged instructions can be issued only in kernel mode.

I/O Protection

- All I/O instructions are privileged instructions.
- Must ensure that a user program could never gain control of the computer in kernel mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).
- Implemented via general **system call** architecture.
  - More later…

Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- Page tables protected in paged memory
- In segmented memory, base and limit registers determine the range of valid addresses.
- Modifying these data must be done in kernel mode, so user processes cannot change what parts of the memory they can access and modify.

CPU Protection

- *Timer* – interrupts computer after specified period to ensure operating system maintains control.
  - Timer is decremented every clock tick.
  - When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing.
- Timer also used to compute the current time.
- Modifying timer data is a *privileged instruction*.
- Modifying clock (upon which timer related activities may rely) is a *privileged instruction*.

Overview of Operating System Structures

Chapter 2

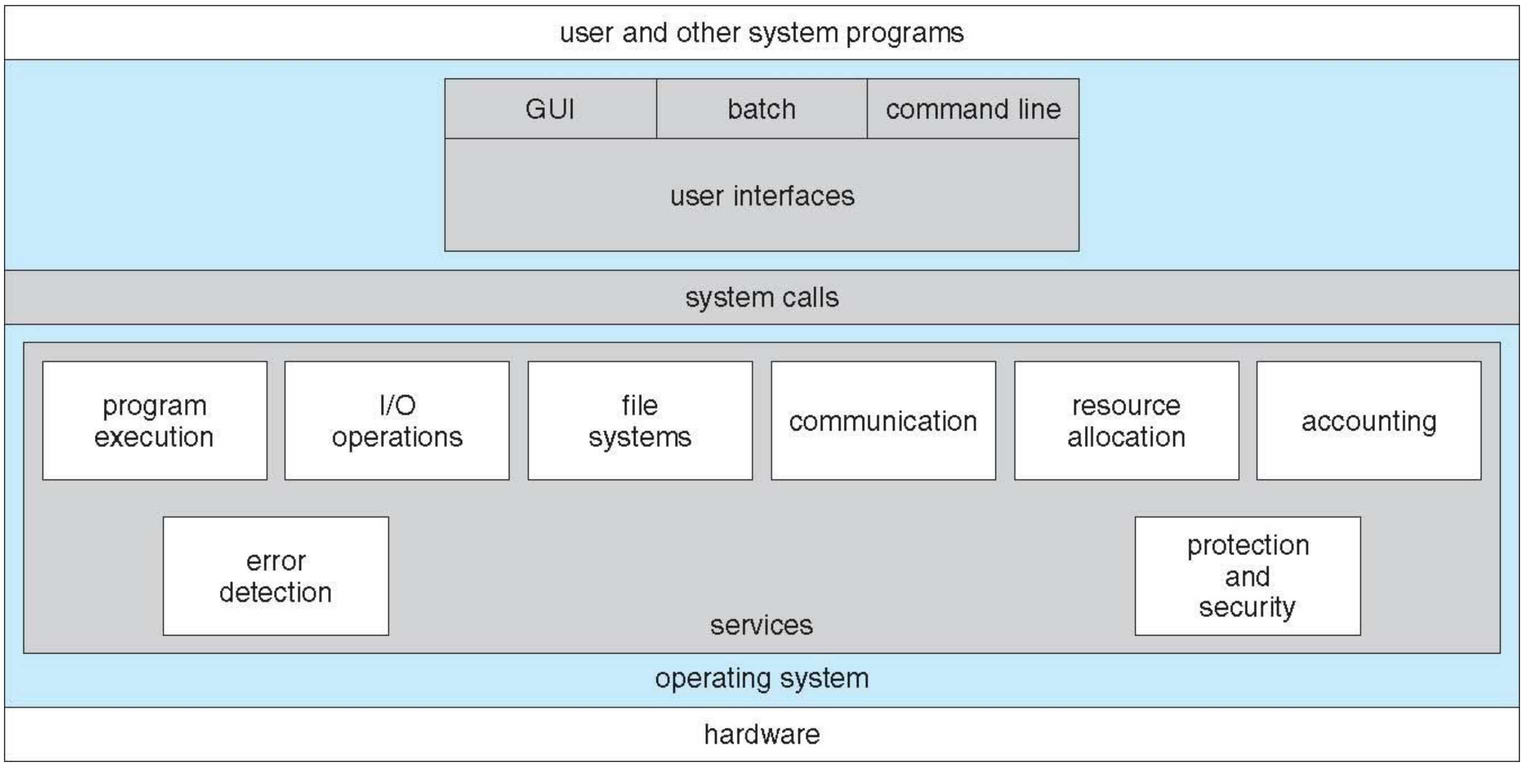
Operating-System Structures

- 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
- Operating System Debugging
- Operating System Generation
- System Boot

Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end 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 - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

A View of Operating System Services



Operating System Services (Cont)

- One set of operating-system services provides functions that are helpful to the user:
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system

Operating System Services (Cont)

- Other OS functions:
  - **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.
  - **Accounting -** Keep track of which users use how much and what kinds of computer resources
    - they may have to pay for services they use ➔ important to be accurate.
  - **Protection and security:**
    - **Protection** ensure that all access to system resources is controlled (e.g., memory access)
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

User Operating System Interface - CLI

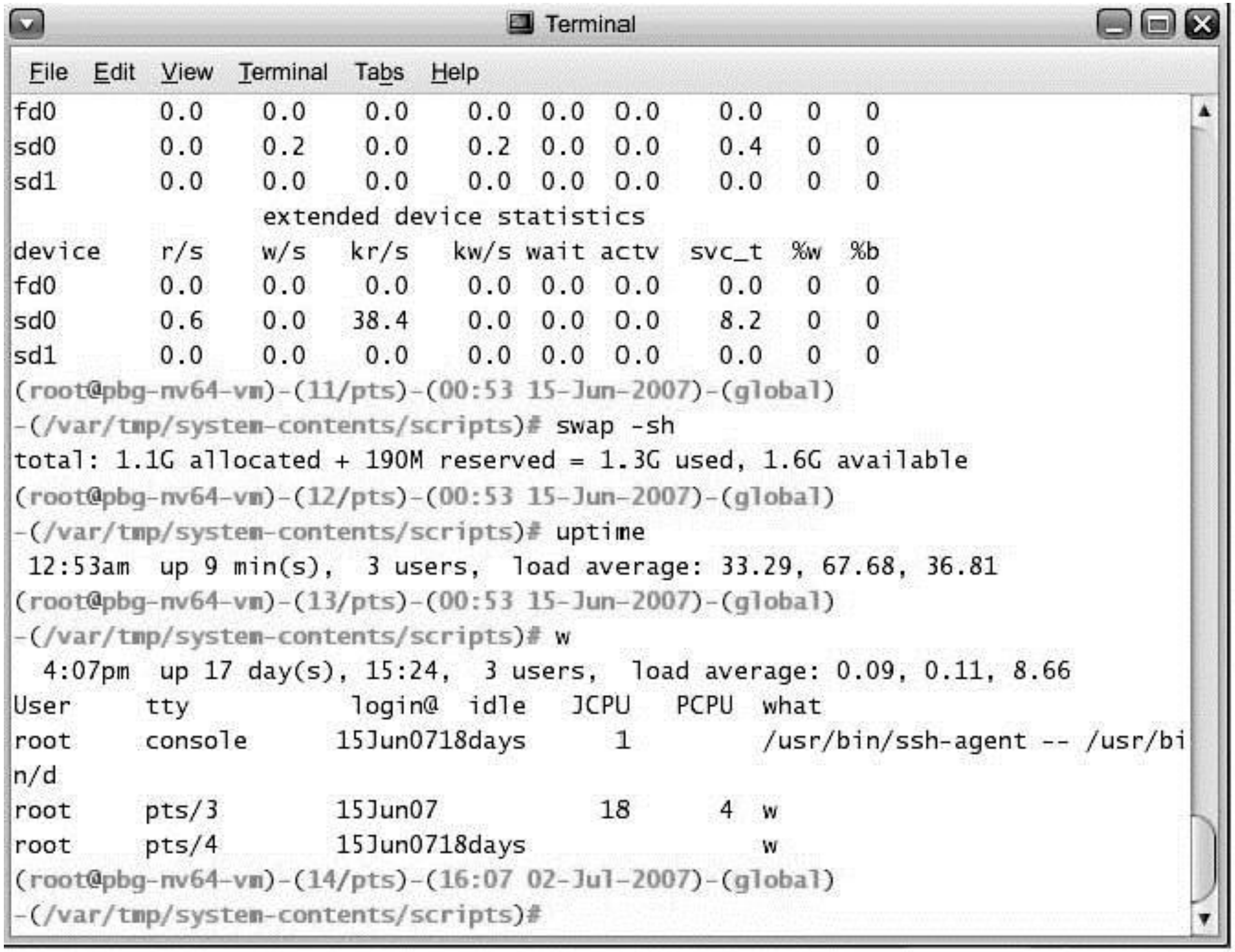
Command Line Interface (CLI) or command interpreter allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – shells
- Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes 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
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - 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 as “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)
  - Linux GUI and various shells

Bourne Shell Command Interpreter



Terminal

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)#
```

The Mac OS X GUI

System Calls

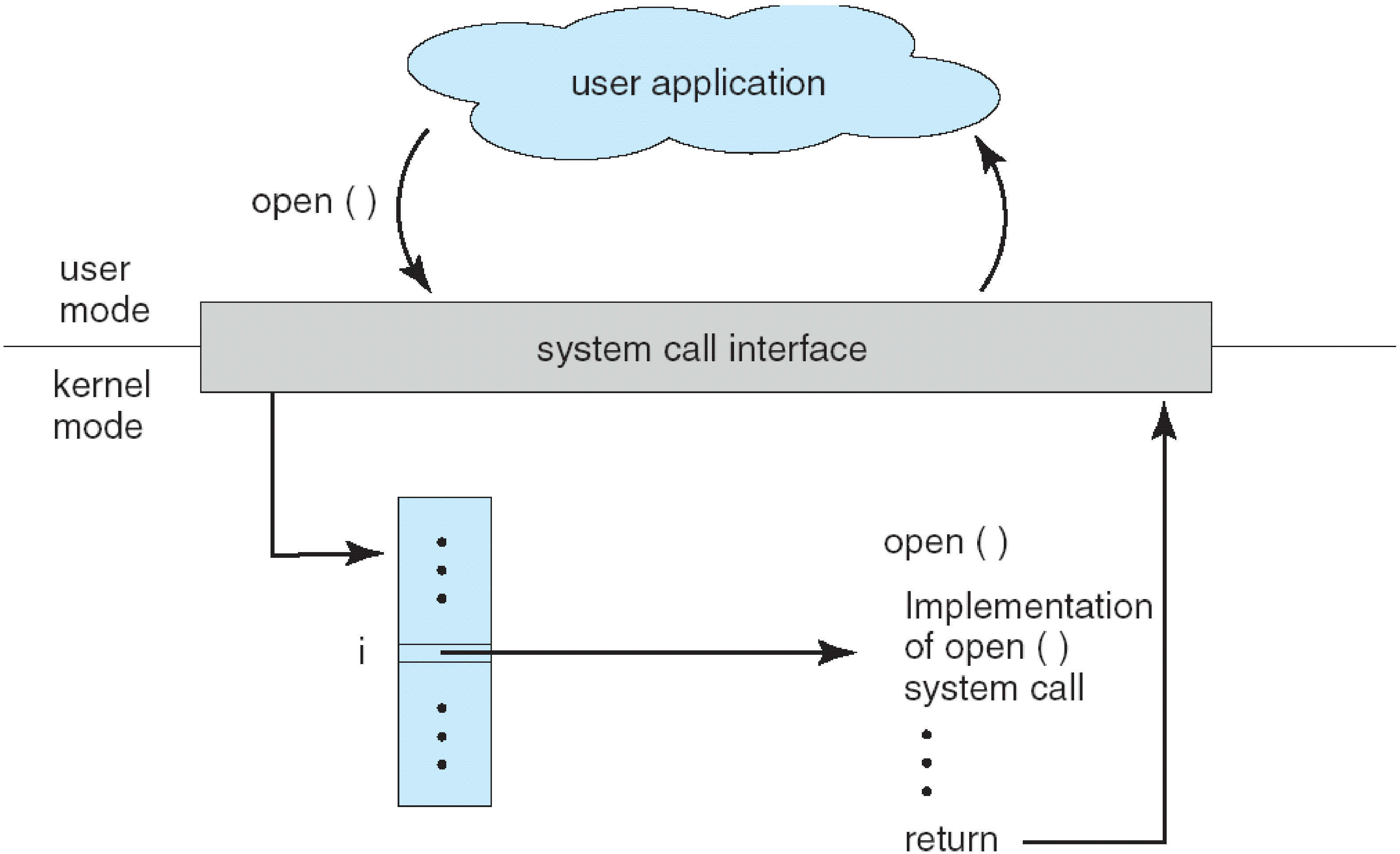
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

(Note that the system-call names used throughout this text are generic)

System Call Implementation

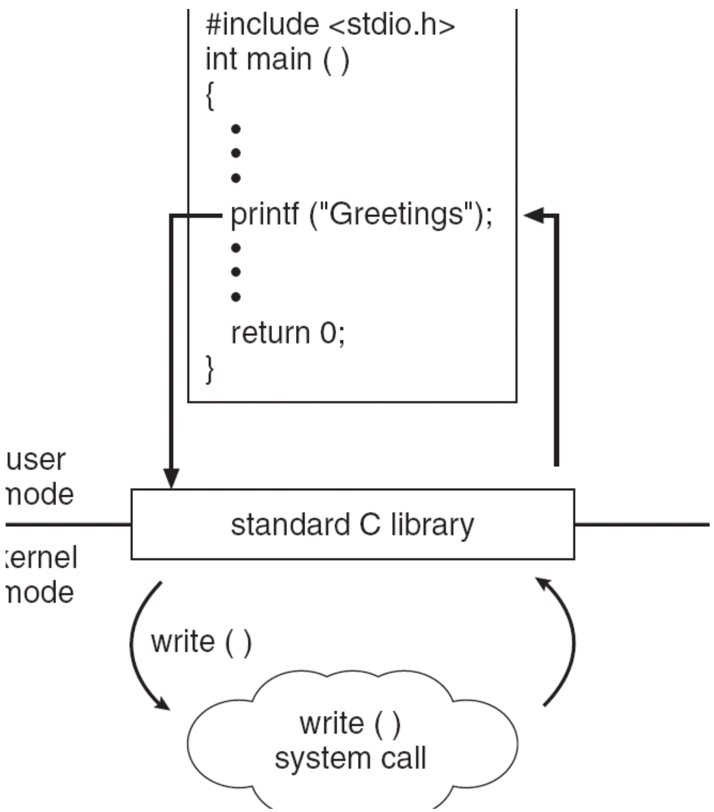
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of 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



Types of System Calls

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

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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Programs

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

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
- Status information
  - Some ask the system for 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

System Programs (cont'd)

- 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- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- Communications - Provide the mechanism 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

Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type 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)

- Important principle to separate

  Policy: What will be done?

  Mechanism: How to do it?

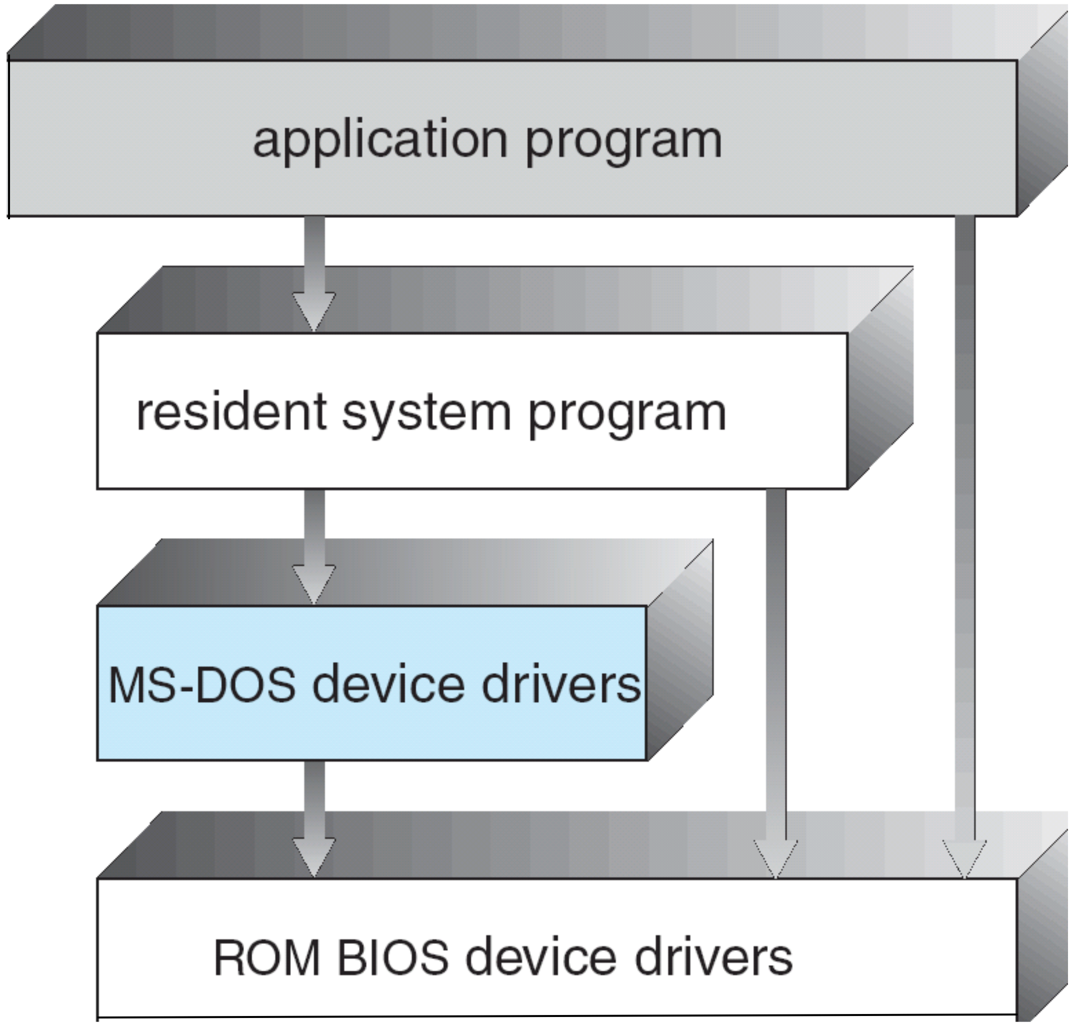
- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- Example:
  - Implementing a timer to protect CPU ➔ mechanism
  - How long to set the timer ➔ policy
  - Want to be able to change policy without changing mechanisms (i.e., without recoding and recompiling things)

How OS's can be organized

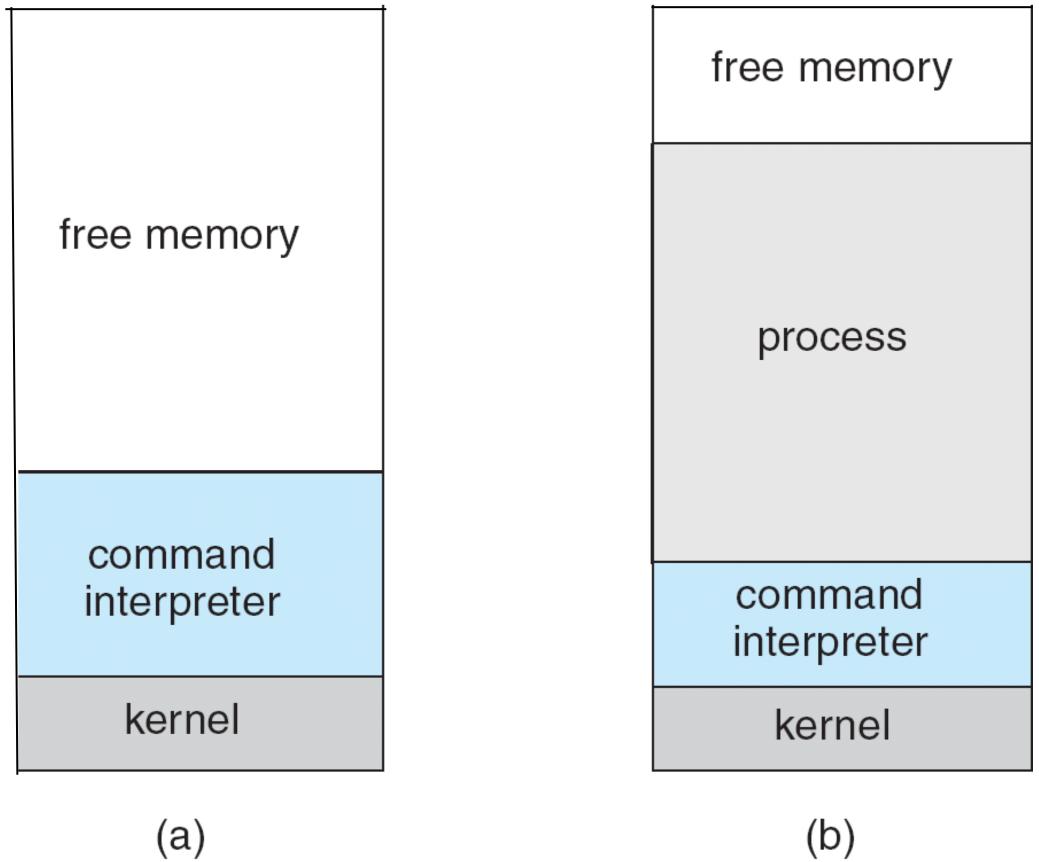
Simple Structure

- 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 Layer Structure

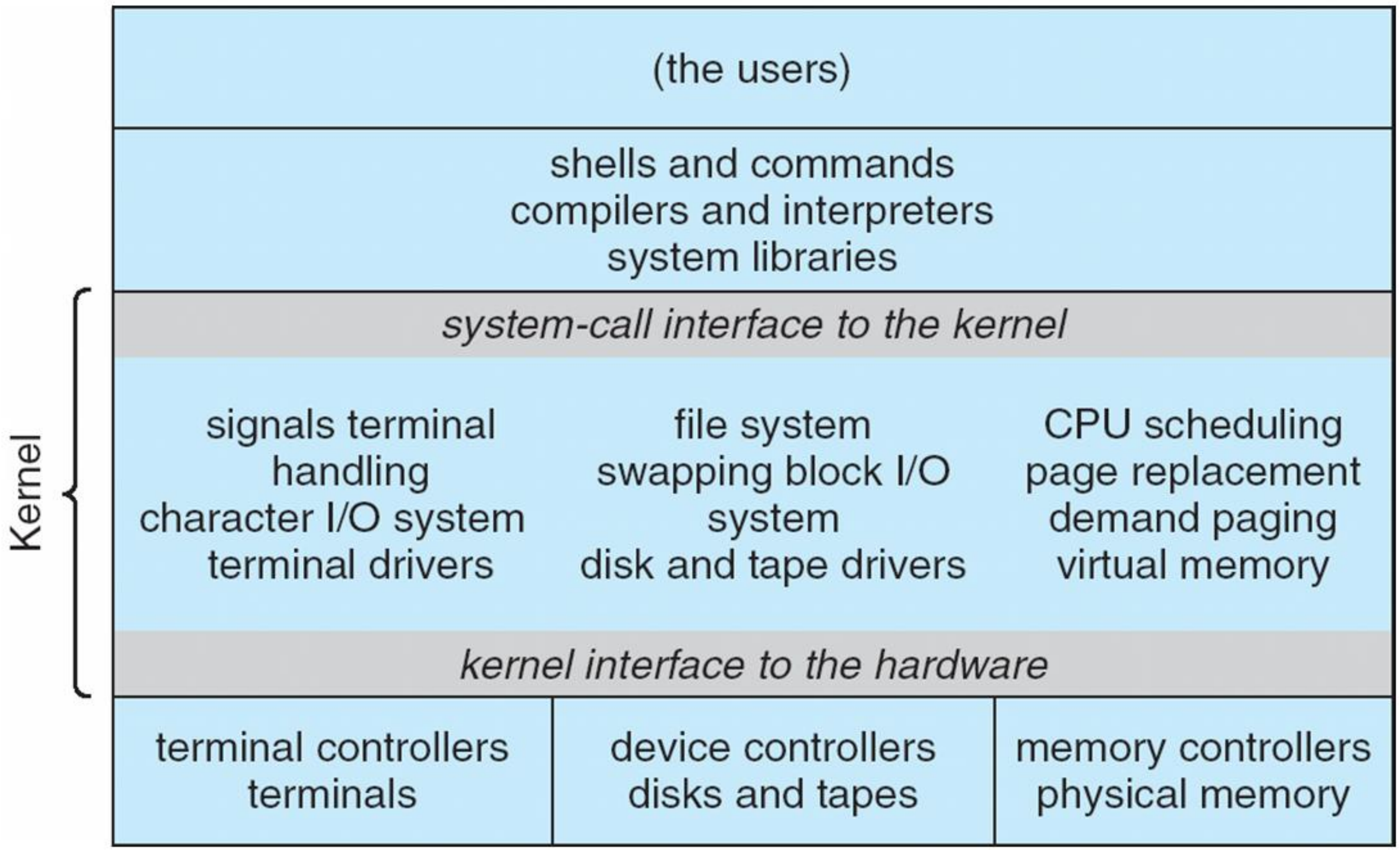


MS-DOS execution



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

Traditional UNIX System Structure

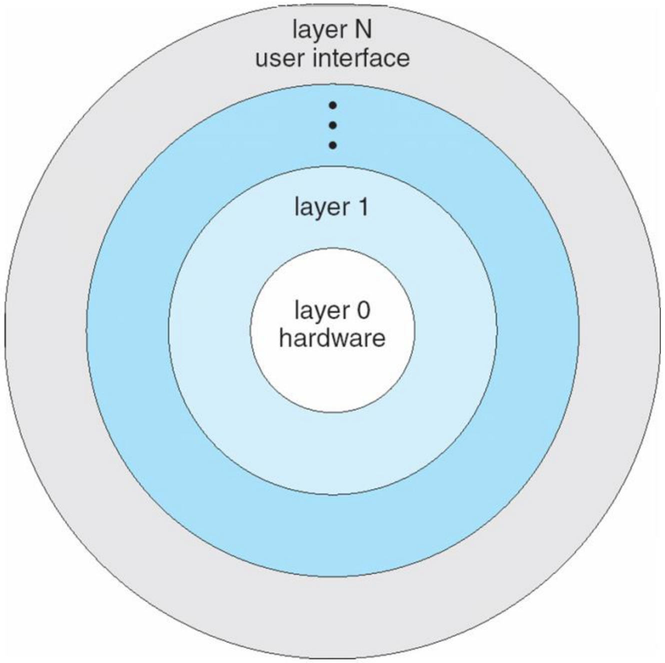


UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - 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

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 (operations) and services of only lower-level layers



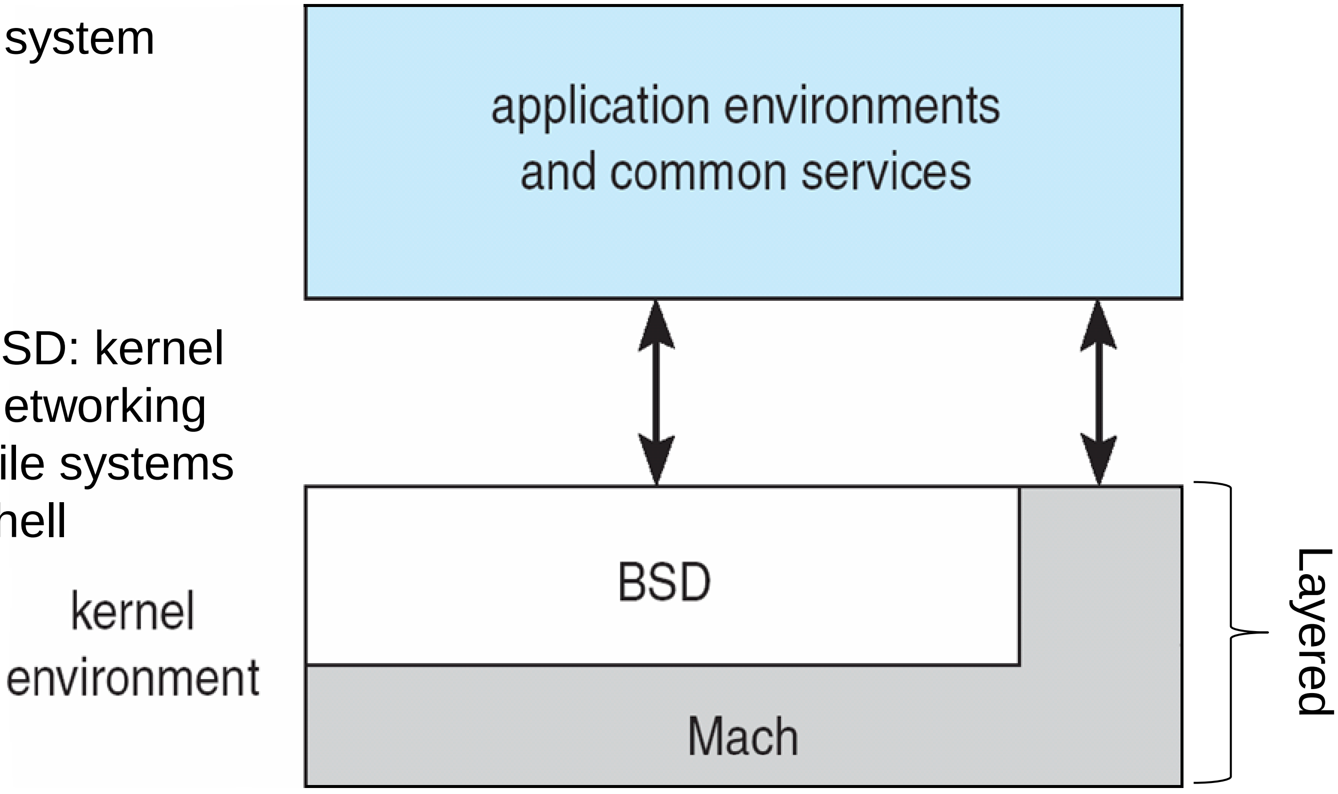
Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using 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)
    - Smaller kernel code → less chance to have serious bugs
  - More secure
    - Smaller kernel → critical/sensitive codes localized to small parts of the system.
- Detriments:
  - Performance overhead of user space to kernel space communication

Mac OS X Structure

Hybrid system

BSD: kernel
Networking
File systems
shell

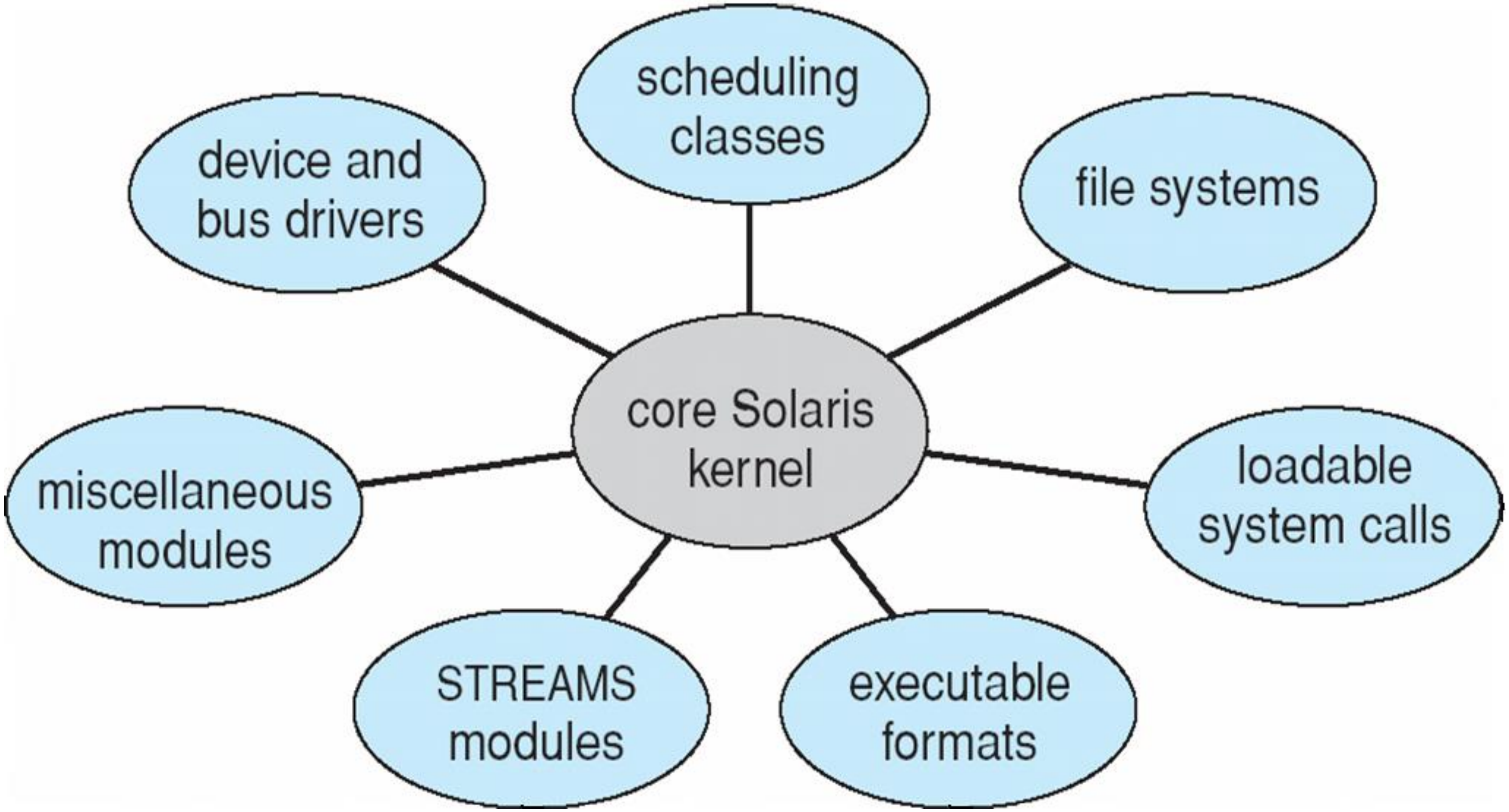


Mach: microkernel
Memory management
IPC, thread scheduling

Modules

- Most modern operating systems 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 flexibility

Solaris Modular Approach



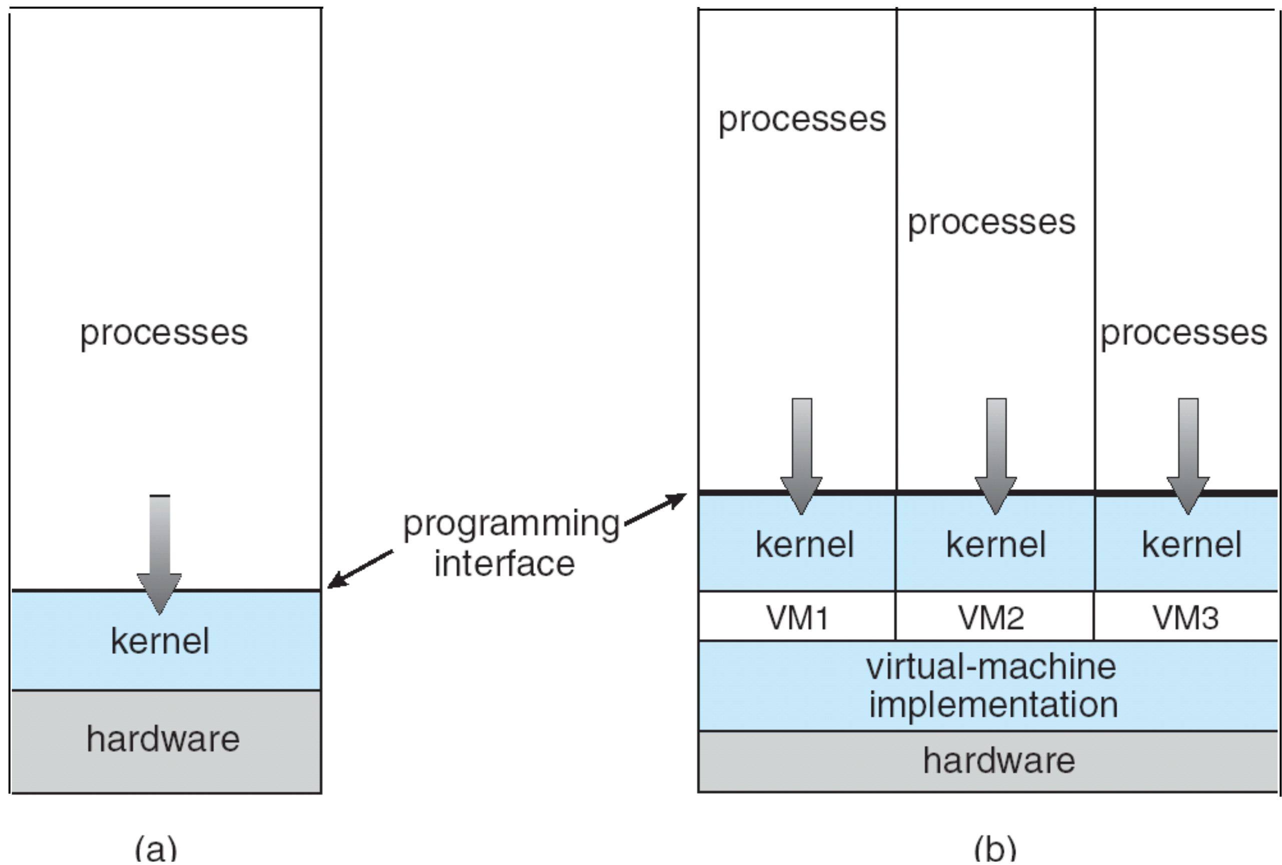
Virtual Machines

- 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 bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

Virtual Machines History and Benefits

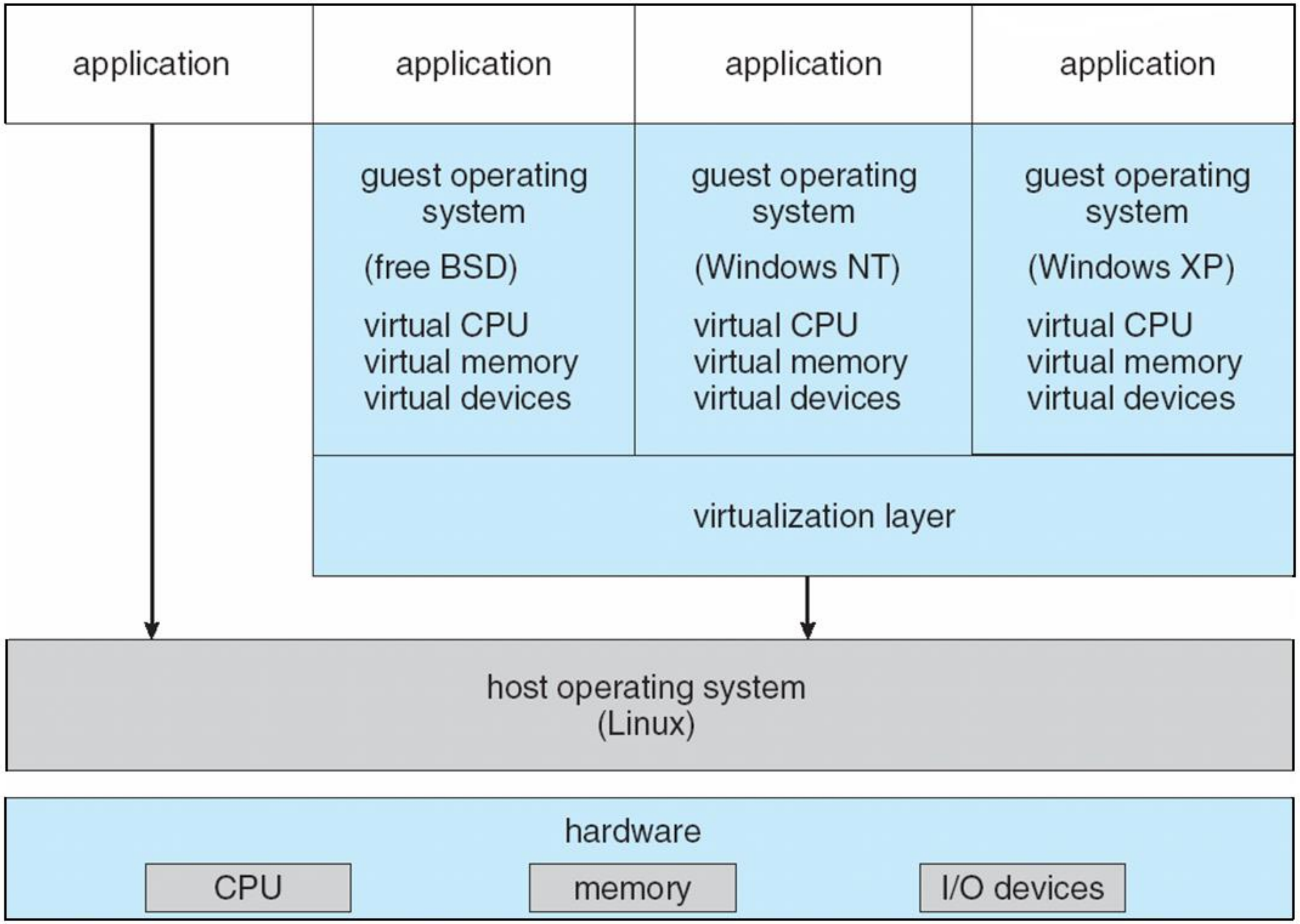
- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing
- Consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms

Virtual Machines (Cont)

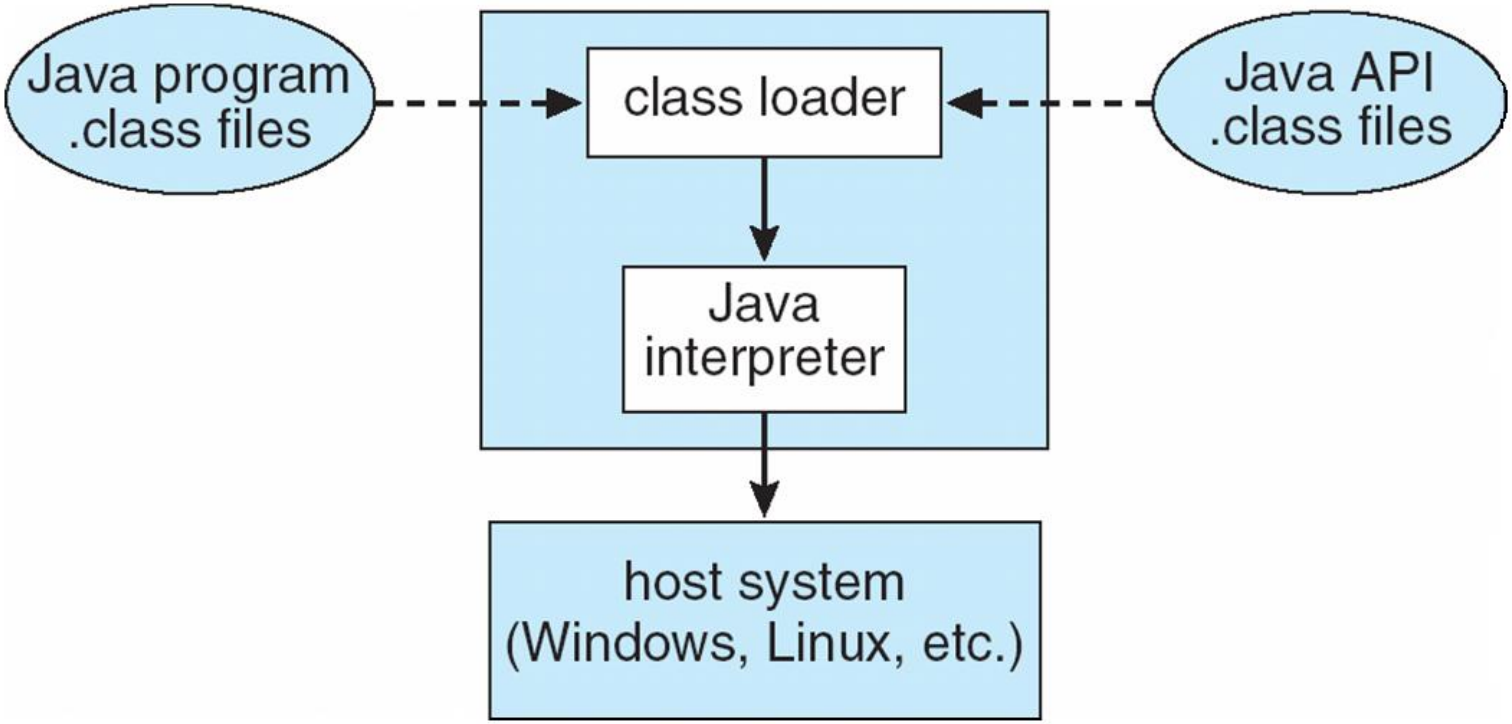


(a) Nonvirtual machine (b) virtual machine

VMware Architecture



The Java Virtual Machine



System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader (bootloader)**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code