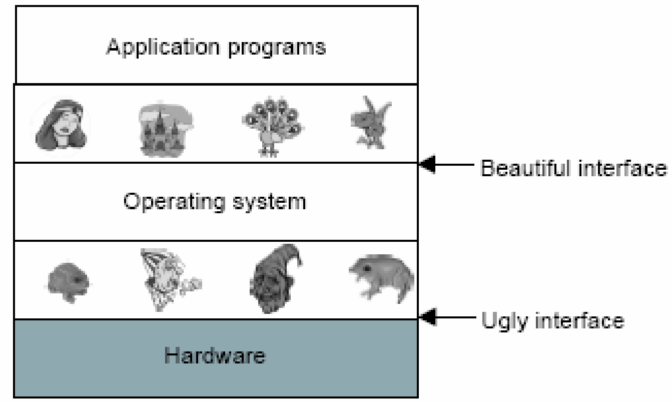
Chapter 2:

Operating-System Structures

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Recap



From: OS by Tanenbaum, 2008

Objectives

- Describe the services provided by an operating system to users, processes, and other systems
- Discuss the various ways of structuring an operating system
- Explain how operating systems are installed and customized, and how they boot

Outline

- 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 Generation
- System Boot

Operating System Services

- One set of OS services provides functions that are helpful to the user:
  - User interface (UI): CLI, GUI, Batch
  - Program execution: <u>Load</u> a program into memory, <u>run</u> that program, <u>end</u> execution either normally or abnormally (indicating error)
  - I/O operations: Provide a means to do I/O required for a running program (process)
  - File-system manipulation: Programs need to <u>read/write</u> <u>files</u> and <u>directories (folders)</u>, <u>create</u>/<u>delete</u> them, <u>search</u> them, <u>list</u> file information, manage access <u>permissions</u> (allow/deny access based on ownership).

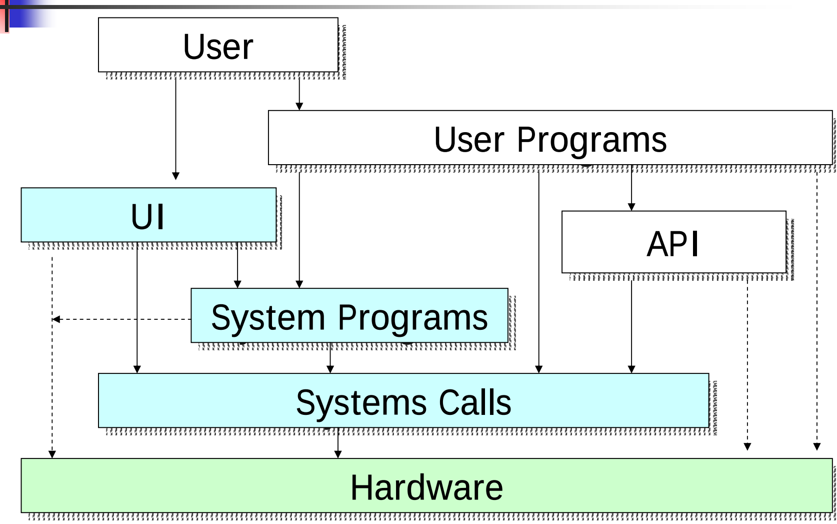
Operating System Services (Cont.)

- Communications:
  - Processes may exchange information, on the same computer or between computers over a network
  - via <u>shared memory</u> or through <u>message passing</u>
- Error detection:
  - OS needs to be constantly <u>aware</u> of possible errors (may occur in the CPU and memory hardware, I/O devices, user program)
    - E.g. power failure, lack of paper in the printer, arithmetic overflow
  - For each type of error, OS should <u>take the appropriate action</u> to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

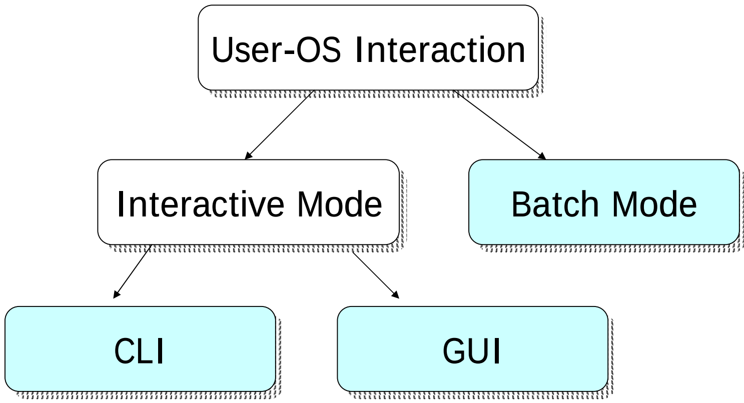
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring <u>efficient</u> operation of the system itself
  - Resource allocation
    - OS allocate resources to multiple users or jobs
    - Some resources may have special allocation code, e.g. CPU cycles, main memory, and file storage
    - Other resources may have general request and release code, e.g. I/O devices (such as printers, modems, USB storage drives)
  - Accounting
    - Keeping track of which users use how much and what kinds of computer resources
    - For billing users or accumulating usage statistics
  - Protection and security
    - <u>Protection</u> involves ensuring that all access to system resources is controlled
      - Not possible for a process to interfere with others or the OS itself
    - <u>Security</u> of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link!

How OS services are made available?



Operating-System User Interfaces



Command Line Interface (CLI)

- CLI is also called command interpreter
- CLI allows direct command entry
- Sometimes implemented in the kernel, sometimes by systems program
- Sometimes multiple interpreters with minor differences are implemented – called **shells**
  - E.g. in Unix/Linux: Bourne shell, C shell, Bourne-Again shell, Korn shell
- Primarily fetches a command from user and executes it
  - E.g. manipulating files: create, delete, copy, move, execute, print, etc
  - Two general ways to implement commands
    - built-in: the interpreter itself contains the code of the command
    - system programs: commands are separate external programs loaded into memory and executed; adding new features doesn't require shell modification

User Operating System Interface - GUI

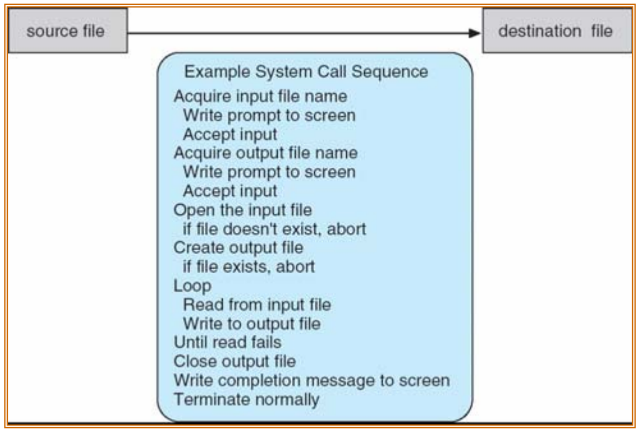
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - First appeared in the early 1970s at Xerox PARC on Xerox Alto computers; gain widespread use with Mac OS; then with Windows OS
- 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 (X-Windows, CDE, KDE, GNOME)

System Calls

- Programming interface to the services provided by the OS
- Routines generally written in a high-level language (C or C++); some low-level tasks are programmed in assembly
- 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?

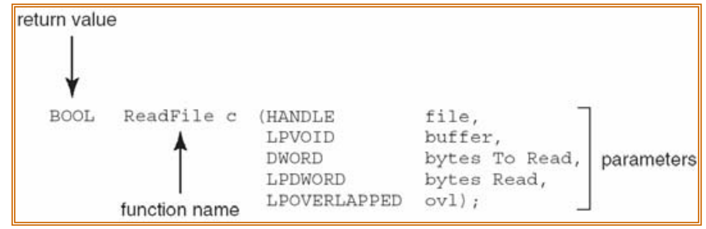
Example of System Calls

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



Example of Standard API

- Consider the ReadFile() function in the Win32 API—a function for reading from a file

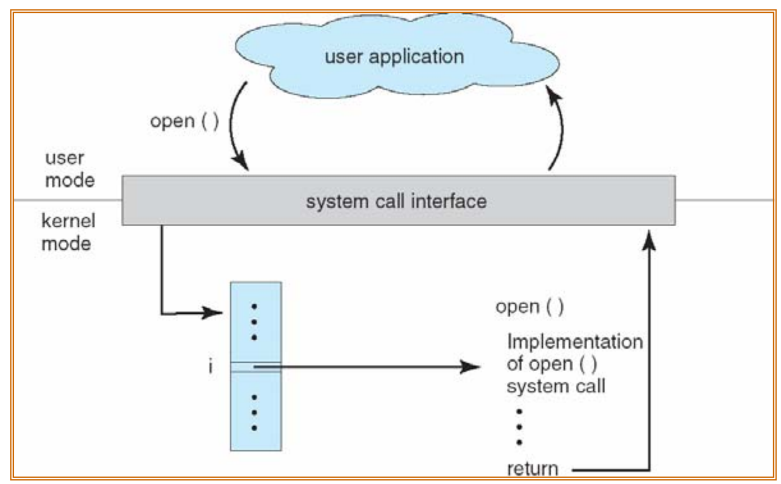


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

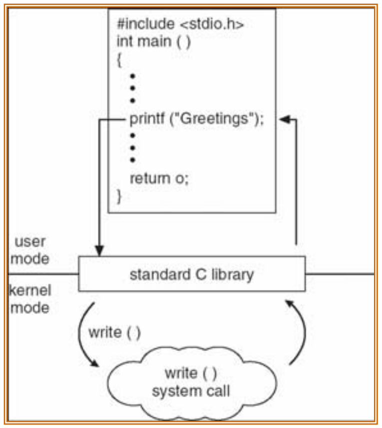
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about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

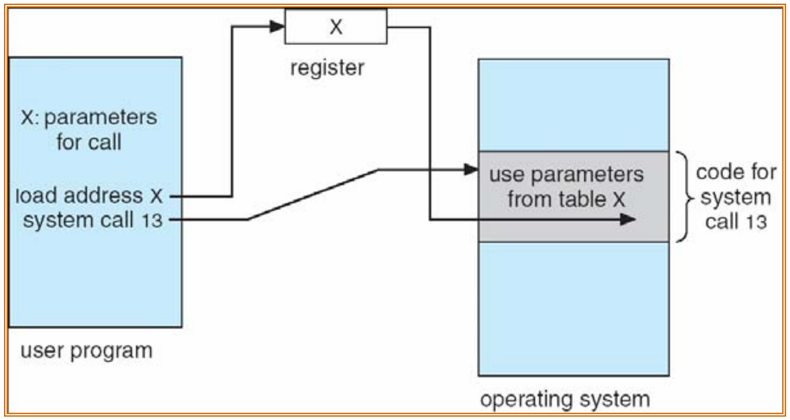
- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

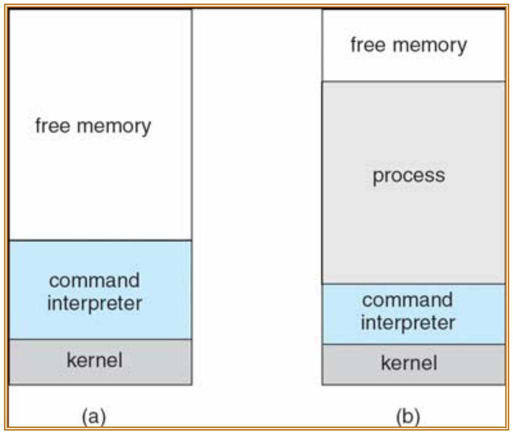
Parameter Passing via Table



Types of System Calls

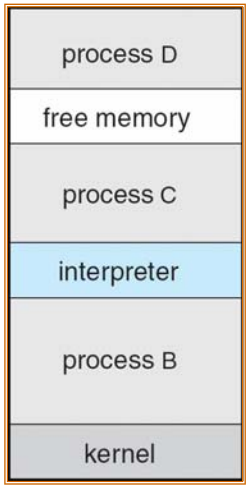
- Process control
- File management
- Device management
- Information maintenance
- Communications

MS-DOS execution



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

FreeBSD Running Multiple Programs



System 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
- Most users' view of the operation system is defined by system programs, not the actual system calls

System Programs (Cont.)

- Various commands that be divided into:
  - File management - generally manipulate files and directories, e.g. delete, copy, rename, print, etc
  - Status information, e.g. date, time, available disk space, detailed performance, configuration information (registry), etc
  - File modification, e.g. edit, modify, and search file content, etc
  - Programming language support, e.g. compilers, assemblers, interpreters, etc
  - Program loading and execution, e.g. absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
  - Communications, e.g. among processes, users, computer systems
  - System utilities (Applications programs), e.g. web browsers, word processors, games

OS Design and Implementation

- There are several challenges facing OS design and Implementation
- No complete solutions to such problems, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely for different environments
- Deign is affected by choice of hardware, and type of system (batch, time shared, single user, multi-user, distributed, real time, general purpose)
- Start design by defining goals and specifications (requirements)
  - User goals vs. System goals
    - OS should be convenient to use, easy to learn, reliable, safe, and fast
    - OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
  - Functional vs. non-functional requirements
- Specifying and designing an OS is highly creative – general principles have been developed in the field of Software Eng.

OS Design and Implementation (Cont.)

- An important principle to separate policies from mechanisms
- Policies decide what will be done
- Mechanisms determine how to do something
- Polices are likely to change across places or over time
  - Worst case – each policy change require a mechanism change
  - Best case (desirable) – mechanism is insensitive to changes in policy
- Separation of policy from mechanism is a very important principle for flexibility if policy decisions are to be changed later

OS Design and Implementation (Cont.)

- After design, the OS is implemented:
  - assembly language, high-level general-purpose languages (e.g. C, C++)
- Example
  - MS-DOS is written in Intel 8088 assembly language (hence can be used only for Intel family of CPUs)
  - Linux is written mostly in C and hence is available for a number of different CPUs (e.g. Intel 80X86, SPARC, MIPS RX000)
  - Windows XP is written mostly in C
- Q. Discuss the advantages and *potential* disadvantages of using a high-level language in implementing OS.

OS Design and Implementation (Cont.)

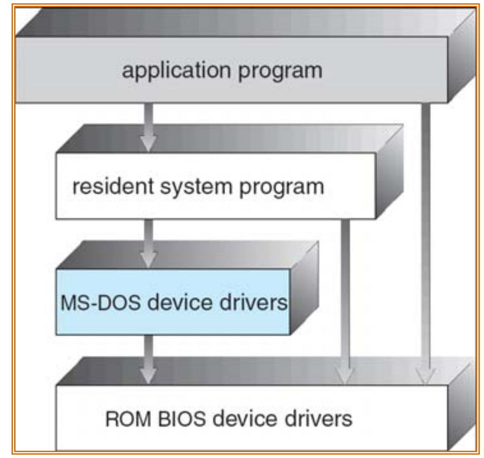
- Performance improvement
  - Better data structure and algorithms
  - Modern compilers can perform sophisticated analysis and optimization to produce excellent code
  - Modern processors have deep pipelining and multiple functional units that can handle complex dependencies (beyond human mind)
  - The most critical routines are probably memory manager and CPU scheduler
- Monitor system performance
  - Extra code must be added to compute and display measures of system behavior
  - Log files and trace lists can be used for further analysis to identify bottleneck and inefficiencies
- Identify and replace bottleneck routines

OS Structures

- OS is complex and large
- Must be carefully engineered to function properly and to be easily modified
- Monolithic vs. modular design
- Simple limited structures vs. well-defined structures to interconnect various components

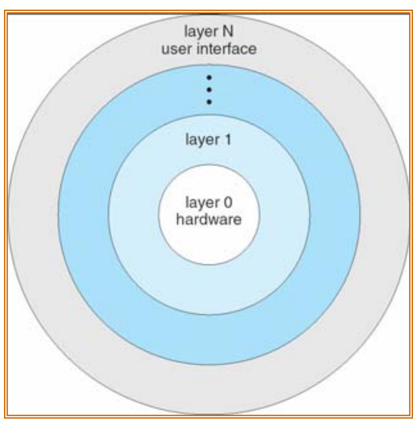
Simple Limited Structures

- 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



Layered Approach

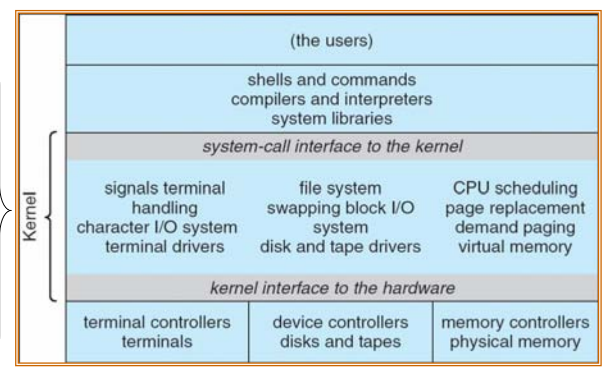
- OS is divided into a number of layers (levels), each built on top of lower layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Benefits: Simplicity of construction, debugging and upgrade
- Detriments:
  - Careful planning is necessary in defining layers
  - Tend to be less efficient



UNIX System Structure

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts: System Programs & Kernel

Kernel:
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 for one level



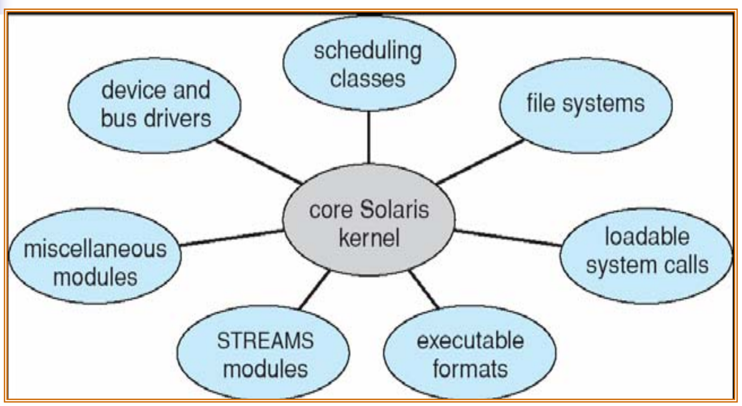
Microkernel System Structure

- Introduced by Carnegie Mellon University in Mach OS
- Moves non-essential components from the kernel to "*user*" level programs (which ones?)
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel based OS
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode; if a process fails, the rest of the OS will not be touched)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Modules

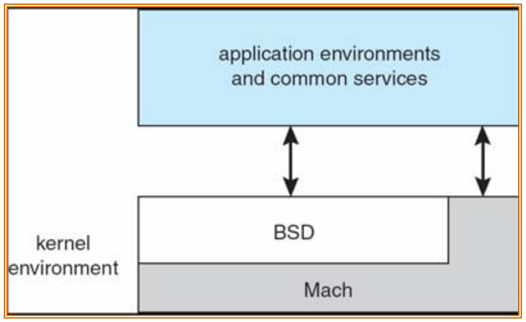
- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each is loadable as needed within the kernel
  - Each talks to the others over known interfaces
- Overall, similar to layers but with more flexible
- Examples: modern implementations of Unix such as Solaris, Linux, and Mac OS X

Example of Modular Kernel: Solaris Loadable Modules



Q. Discuss how it is similar and different from layered and microkernel approaches.

Hybrid Structure: Mac OS X



- Use microkernel (Mach) for memory management, remote procedure calls (RPCs), Interprocess communication (IPC) facilities
- BSD provides CLI, support for networking and file systems, implementation of POSIX APIs
- Kernel environment (extensions) provides I/O kit for development of device drivers and dynamically loadable modules

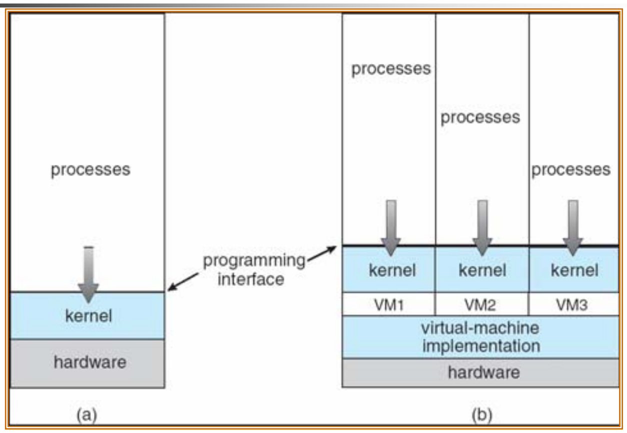
Virtual Machines (VMs)

- 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 creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

Virtual Machines (Cont.)

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling and virtual memory can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual card readers and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator's console

Virtual Machines (Cont.)

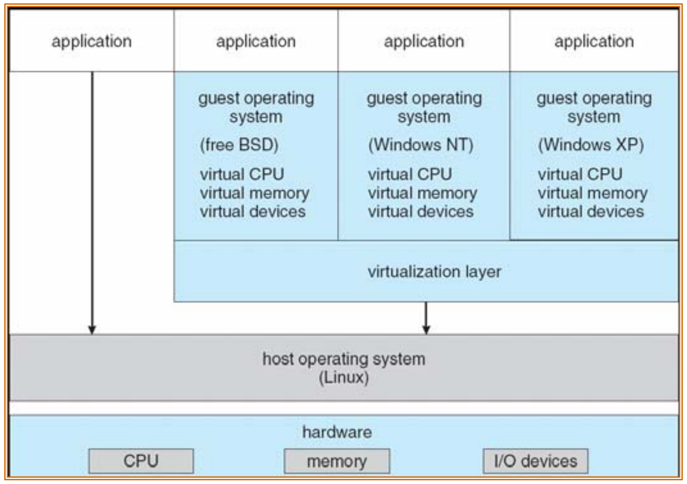


(a) Nonvirtual machine (b) virtual machine

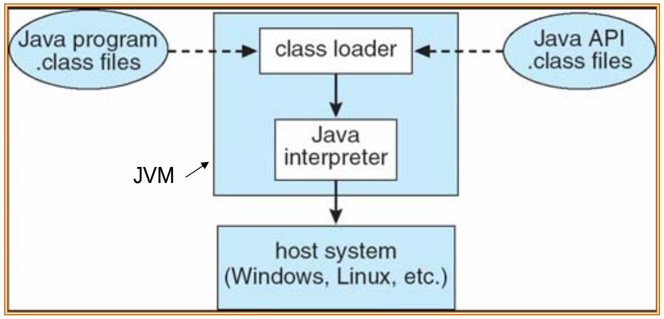
Virtual Machines (Cont.)

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine

VMware Architecture



The Java Virtual Machine (JVM)



- JVM runs on the top of a host OS or embedded in web browser
- Can be implemented as software (using interpreter or JIT compiler similar to .NET framework) or hardware (Java chip)

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
  - What CPUs is there? What options are installed?
  - How much memory is available?
  - What devices are available?
  - What OS options are desired?
- Implementation variations:
  - Completely tailored
  - Less tailored
  - Completely table driven
- Criteria: generality, size and ease of modification as the hardware change

System Boot

- Operating system must be made available to hardware so hardware can start it
- *Booting* – starting a computer by loading the kernel
  - **Bootstrap loader**: a small piece of code locates the kernel, loads it into memory, and starts its execution
    - Can also determine the state of the machine (diagnostic tests)
  - Sometimes use two-step process
    - Bootstrap starts code at a fixed location on the disk called **boot block**
    - Boot disk or system disk
  - When a CPU is powered up, the instruction register is loaded with a predefined memory location (which has the initial bootstrap program)
    - Firmware used to hold initial boot code
    - Changing the bootstrap requires changing the ROM or using EPROM

End of Chapter 2

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005