Ch 2 - Overview

1. Interacting with services provided by the OS
   - System calls - link between application programs and OS
   - System programs - users interact using programs

2. Ways of structuring the OS itself - OS can be a large code base that needs to be maintainable, portable etc.
   - Monolithic - aka spaghetti code
   - Layered
   - Micro-kernel - small and efficient kernel, functionality moved to user level
   - Virtual machine - make it look like multiple “machines” - very popular with data centers

3. Installation, customization etc.
   - booting

Recap OS:allows for program execution

- load a program into memory and run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program requires I/O, which may involve a file, an I/O device, shared with other programs or computers
  - Error detection – OS are constantly aware of errors
    - May occur in the CPU and memory hardware, in I/O devices and 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

Interfacing with OS

- User interface - Almost all operating systems have a user interface (UI). Varies between
  - **Command-Line (CLI)** (e.g., shells in UNIX, command.exe in Windows). The command line may itself perform functions or call other system programs to implement functions (e.g. in UNIX, /bin/rm to remove files) [more later]
  - **Graphics User Interface (GUI)** (e.g., MS windows, MAC OS X Aqua, Unix X & variants). point and click interface
  - **Batch**. Commands are given using a file/command script to the OS and are executed with little user interaction. Used in high performance computers. (e.g. .bat files in DOS, shell scripts, JCL interpreters for Main frames)

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C, 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?
  - Underlying systems calls (error codes) can be more complicated. API gives a uniform, portable interface

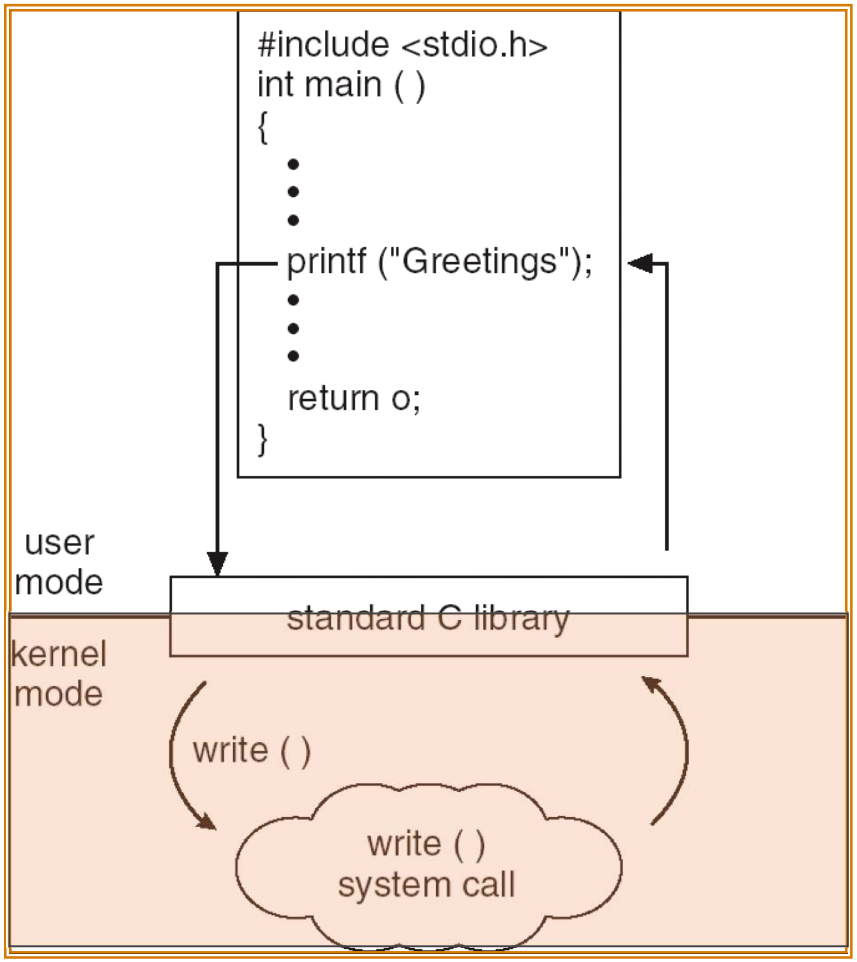
Example of System Calls

- System call sequence to copy the contents of one file to another file (POSIX like C pseudo code) (bold are API system calls)

```
write(1, “Input file\n”, 11);
read(0, &buffer, 100);
…..
Fd = open(buffer, O_RDONLY);
Outfd = open(buffer, O_WRONLY | O_CREAT | O_TRUNC,
    0666);
if (Outfd < 0) abort(“File creation failed”);
…..
close(fd);
```

Standard C Library Example

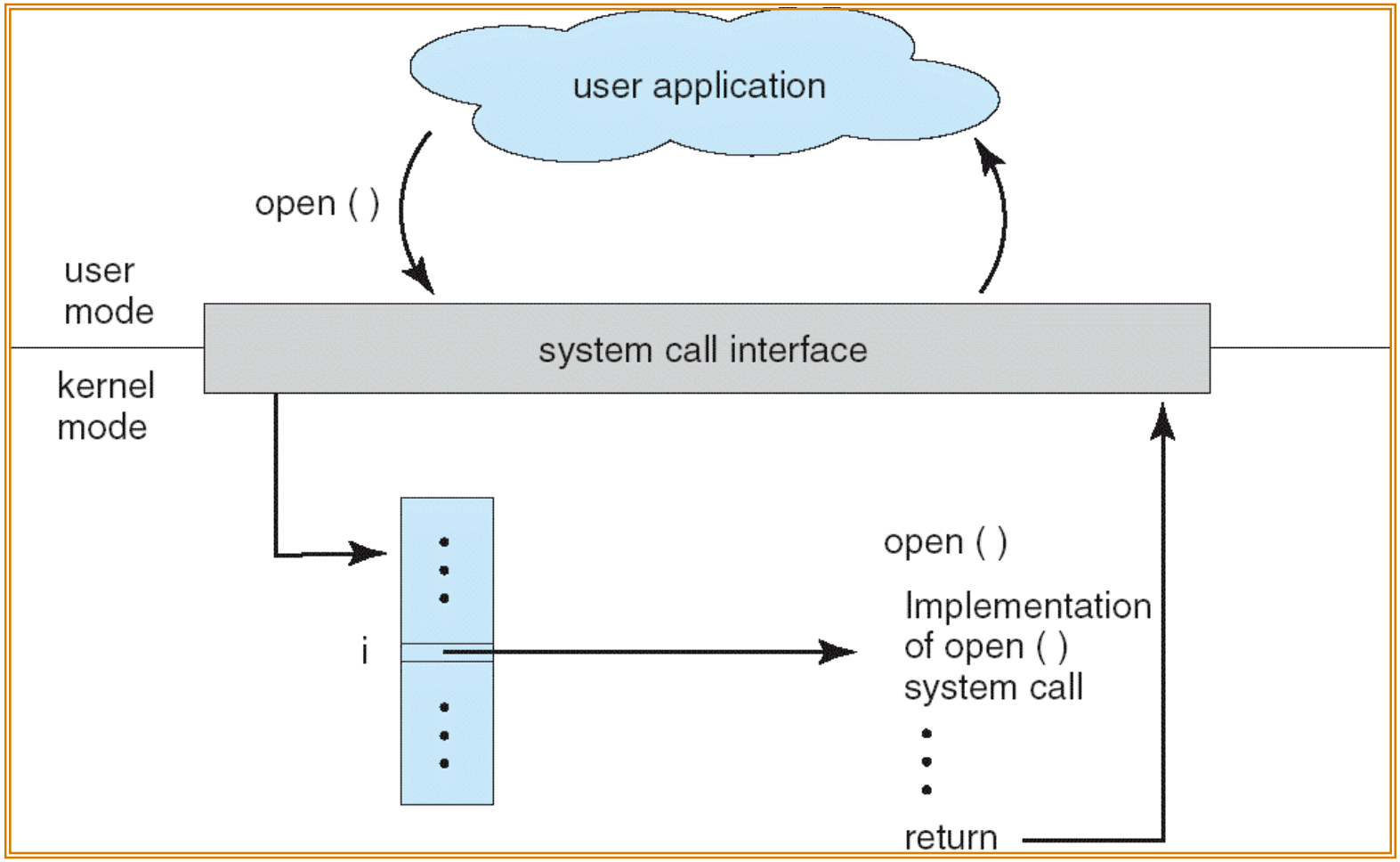
- C program invoking printf() library call, which calls write() system call



System Call Implementation

- A number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
  - Additional info: check /usr/include/sys/syscall.h
- 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
  - Details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

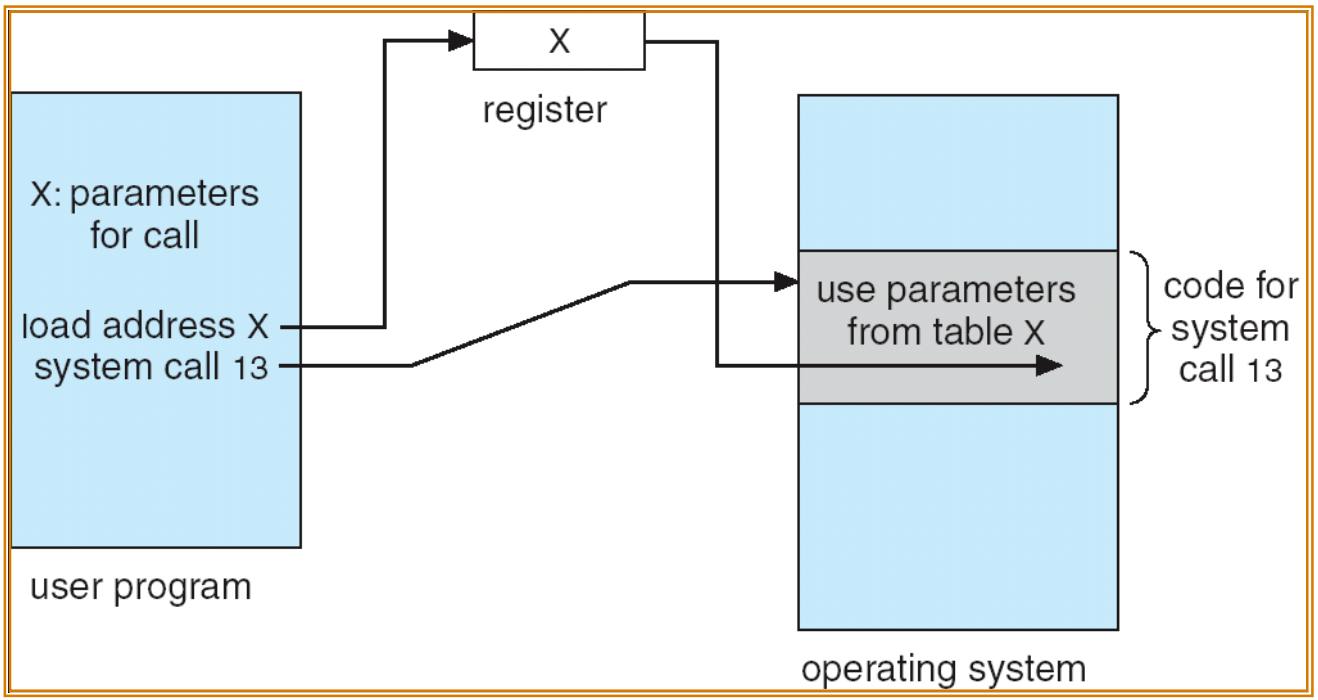
API – System Call – OS Relationship



System Call Parameter Passing

- 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 hardware *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



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, edit, rename, print, dump, list, and generally manipulate files and directories
  - 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 - chat, web browsing, email, remote login, file transfers
  - Status information - system info such as date, time, amount of available memory, disk space, number of users

Operating System Design and Implementation

- Design and Implementation of OS 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 (portable?)**, as well as flexible, reliable, error-free, and efficient
- Important principle to separate

  Policy: What will be done?
  Mechanism: How to do it?
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

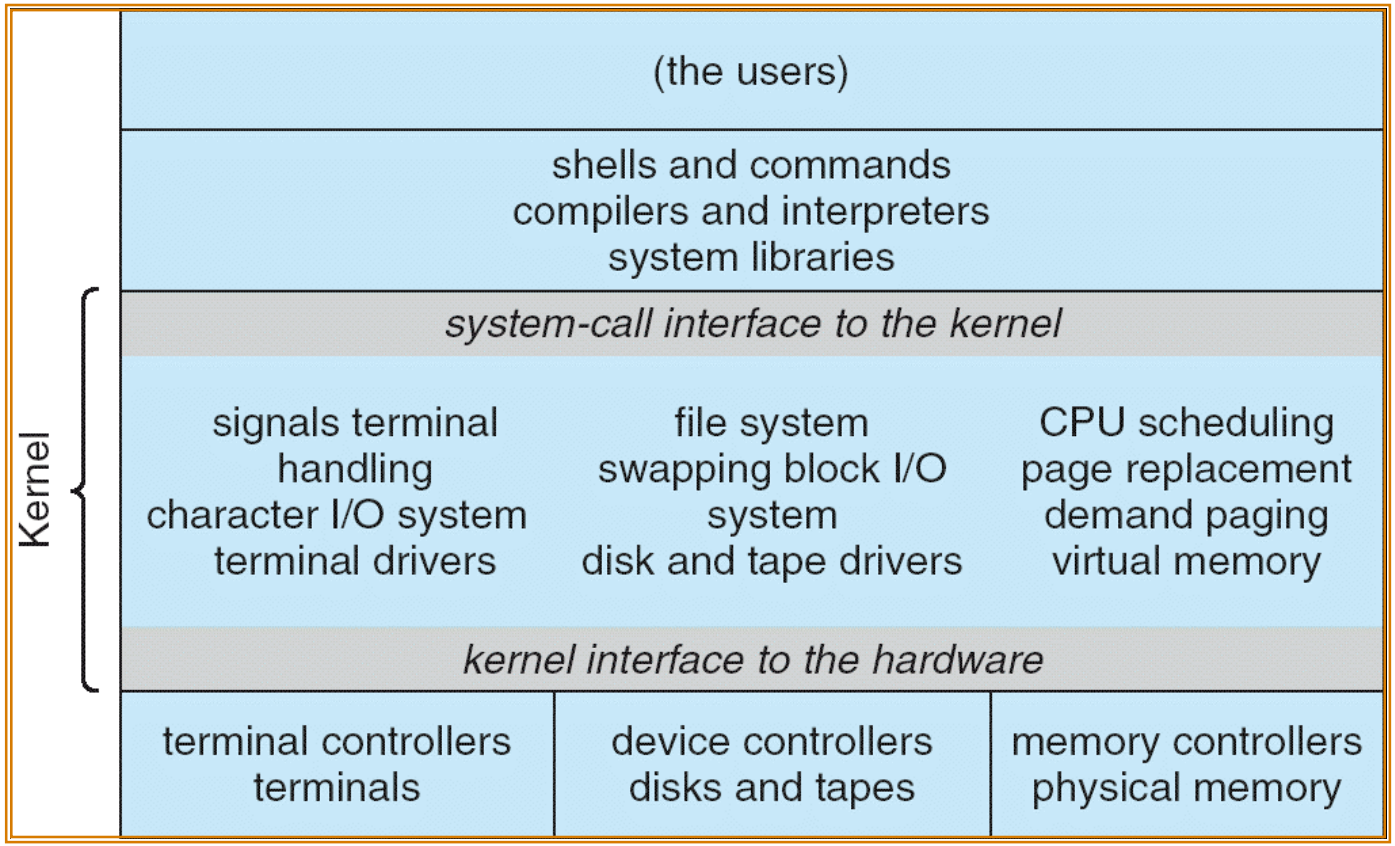
Simple Structure

- 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

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
- UNIX – the 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

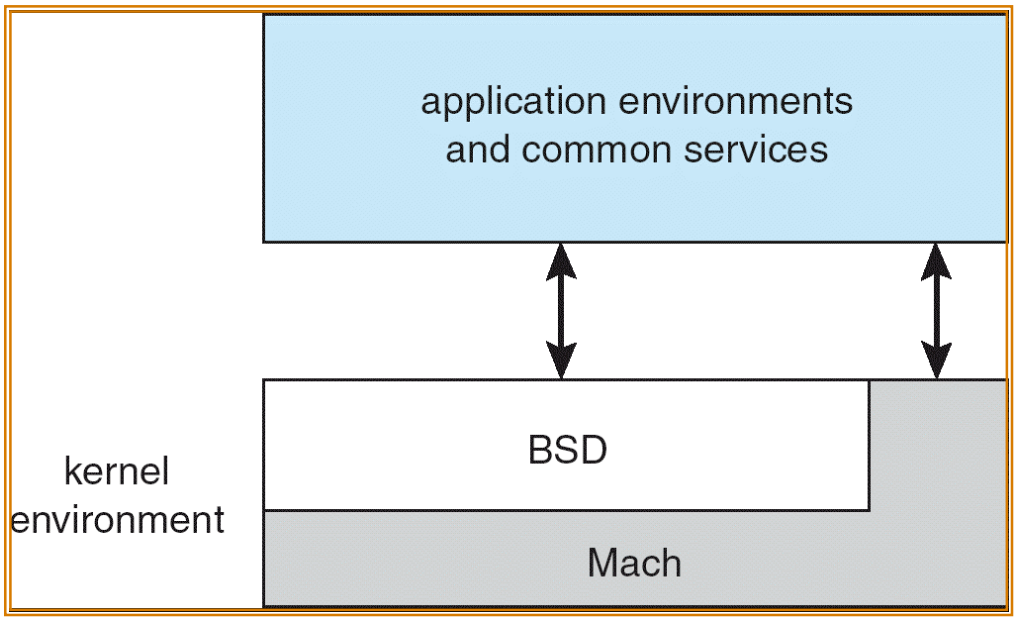
UNIX System Structure



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)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Mac OS X Structure



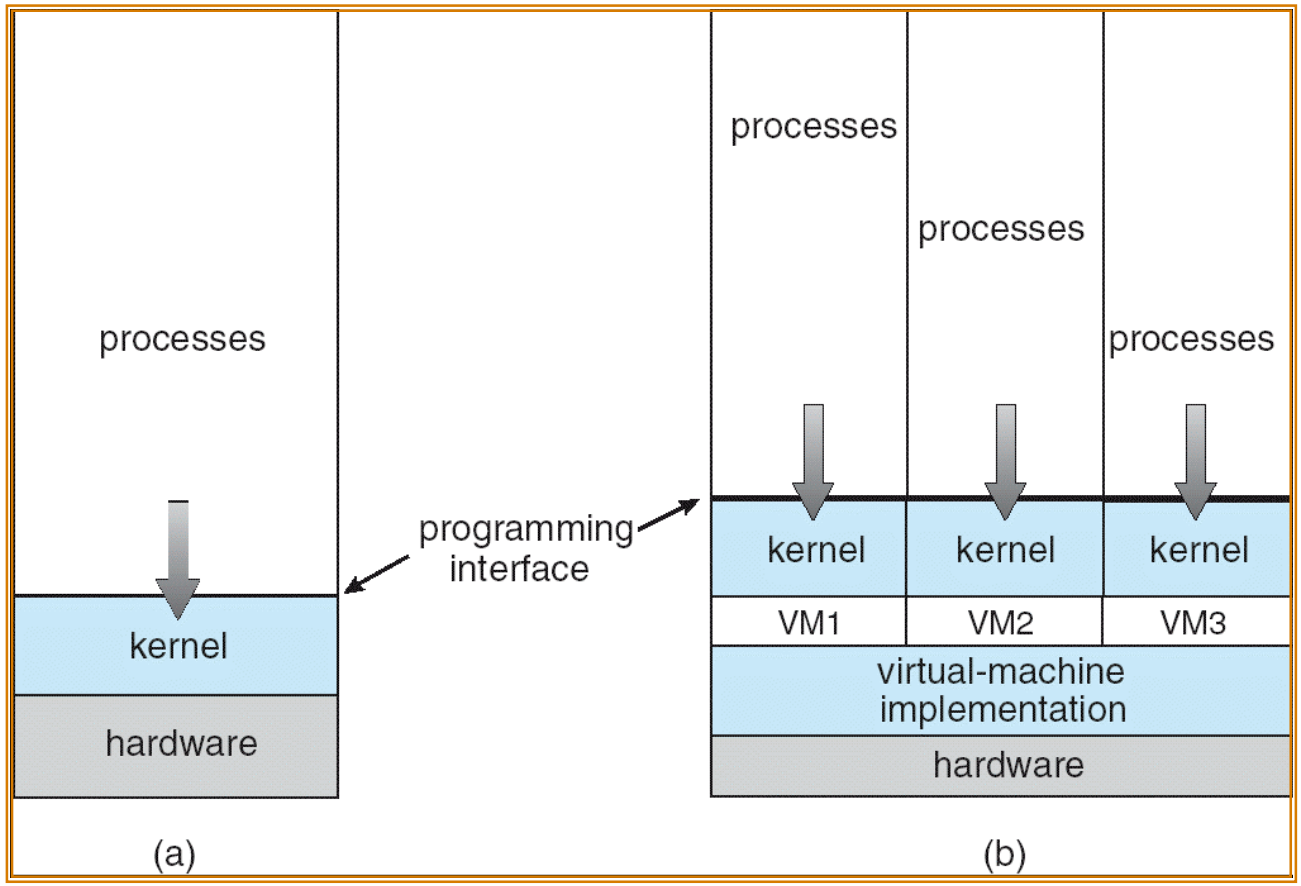
Modules

- 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 flexible

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

Virtual Machines (Cont.)

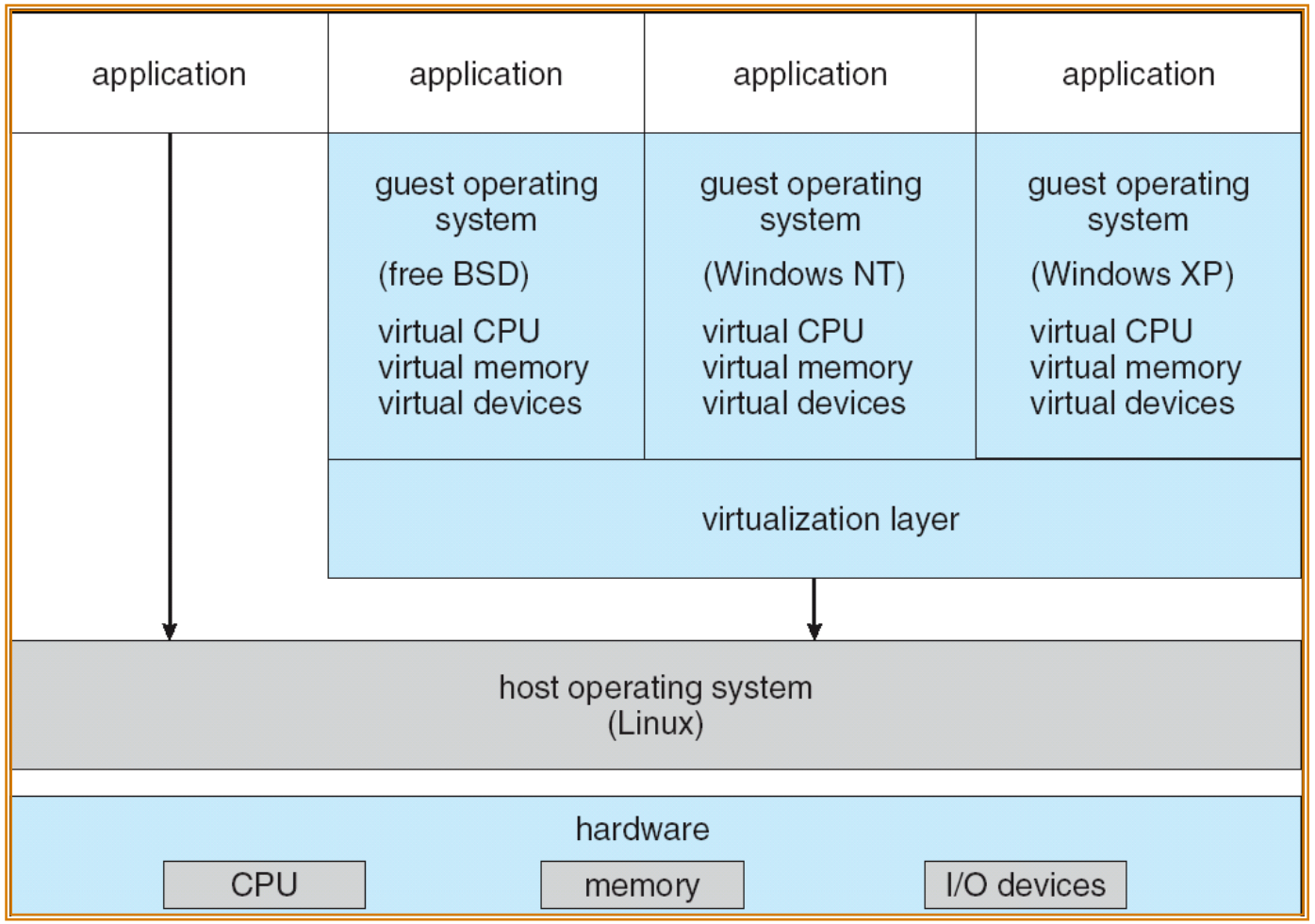


(a) Nonvirtual machine (b) virtual machine

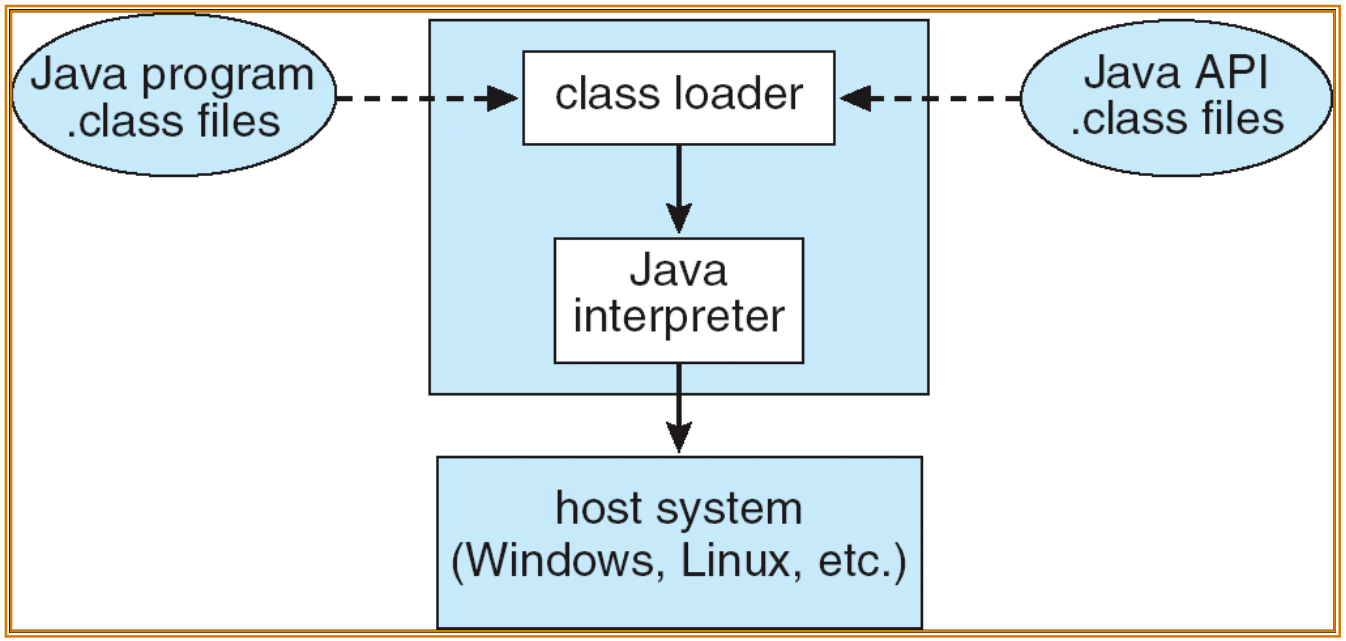
Virtual machines for data centers

- 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



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

System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, 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

Wrapup

- System calls provide a mechanism for user programs to access OS services
  - System programs use system calls to provide functionality to users
- Need to design the OS for maximum flexibility of the user and OS developer
- Other issues such as bootstrapping to initialize the OS