UNIT I

OS STRUCTURE + API & DESIGN OF AN API + COMPONENTS

OS Abstraction

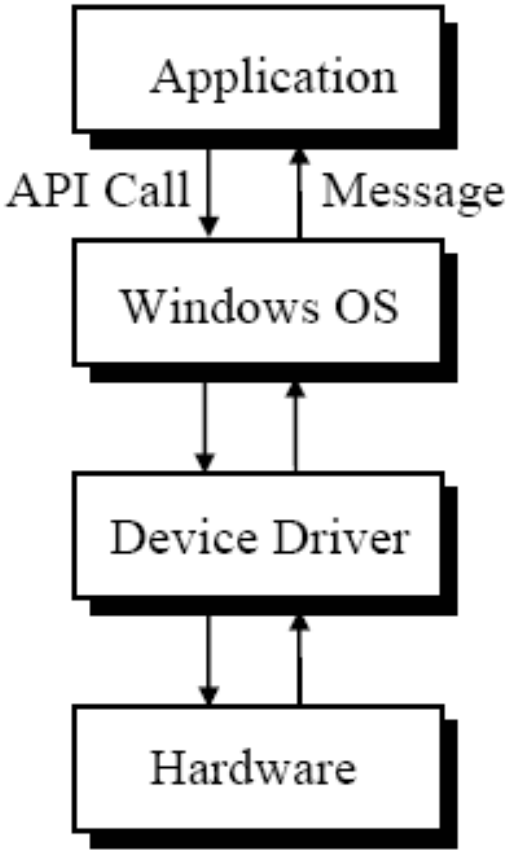
- The operating system provides a layer of abstraction between the user and the bare machine. Users and applications do not see the hardware directly, but view it through the operating system.
- This abstraction can be used to hide certain hardware details from users and applications. Thus, changes in the hardware are not seen by the user (even though the OS must accommodate them).
- This is particularly advantageous for venders that want offer a consistent OS interface across an entire line of hardware platforms.
- For example, certain operations such as interaction with 3D graphics hardware can be controlled by the operating system. When an instruction pertaining to the hardware is executed and if the hardware is present then all is fine. However, if the hardware is not present then a trap is generated by the illegal instruction. In this case the OS can emulate the desired instruction in software.
- Another way that abstraction can be used is to make related devices appear the same from the user point of view. For example, hard disks, floppy disks, CD-ROMs, and even tape are all very different media, but in many operating systems they appear the same to the user.
- Unix, and increasingly Windows NT, take this abstraction even further. From a user and application programmer standpoint, Unix is Unix regardless of the CPU make and model.

What is an API?

- API, an abbreviation of ***a****pplication* ***p****rogram* ***i****nterface*, is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

- Most operating environments, such as MS-Windows, provide an API so that programmers can write applications consistent with the operating environment. Although APIs are designed for programmers, they are ultimately good for users because they guarantee that all programs using a common API will have similar interfaces. This makes it easier for users to learn new programs.

- An API can be created for applications, libraries, operating systems, etc, as a way to define their "vocabularies" and resources request conventions (e.g. functions calling conventions). It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer program and the implementer program of the API


Application
API Call
Message
Windows OS
Device Driver
Hardware

What is an API?

- An API is an abstraction that describes an interface for the interaction with a set of functions used by components of a software system. The software providing the functions described by an API is said to be an *implementation* of the API.

An API can be:

- general, the full set of an API that is bundled in the libraries of a programming language, e.g. Standard Template Library in C++ or Java API.
- specific, meant to address a specific problem, e.g. Google Maps API or Java API for XML Web Services.
- language-dependent, meaning it is only available by using the syntax and elements of a particular language, which makes the API more convenient to use.
- language-independent, written so that it can be called from several programming languages.

WINDOWS API

- **Purpose**

  The Microsoft Windows application programming interface (API) provides services used by all Windows-based applications.

  You can provide your application with a graphical user interface; access system resources such as memory and devices; display graphics and formatted text; incorporate audio, video, networking, or security.

- **Where Applicable**

  The Windows API can be used in all Windows-based applications. The same functions are generally supported on 32-bit and 64-bit Windows.

- **Developer Audience**

  This API is designed for use by C/C++ programmers. Familiarity with the Windows graphical user interface and message-driven architecture is required.

Application Programming Interface (API)

API s are implement using 3 Libraries in windows.

- **KERNEL**
- **USER**
- **GDI**

Kernel

It is library named KERNEL32.DLL, which supports capabilities associated with OS such as

- **Process Loading.**
- **Context switching.**
- **File I/O.**
- **Memory Management.**

User

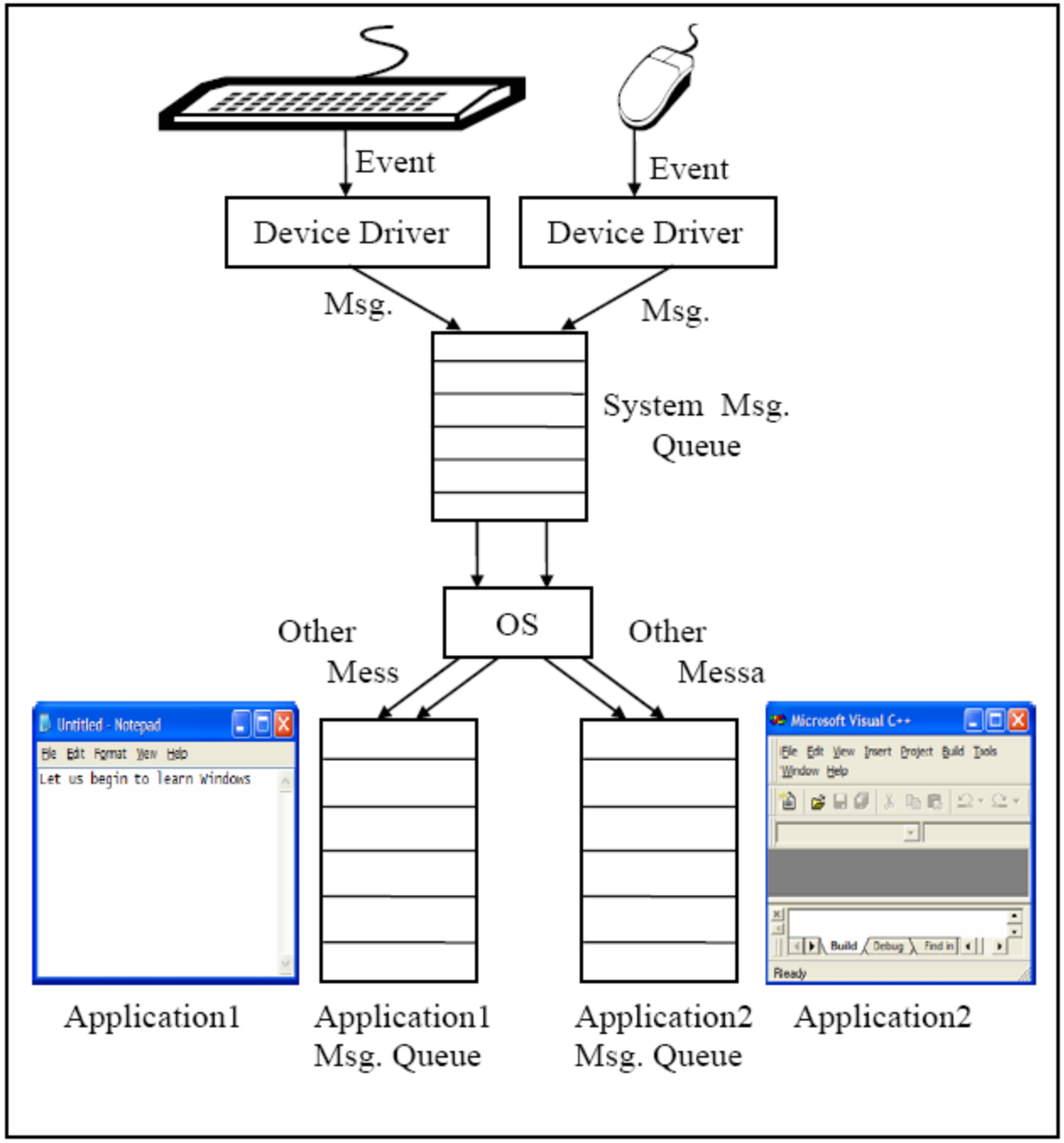
It is library named USER32.DLL, which allows managing the user interface such as

- **Windows.**
- **Menus.**
- **Dialog Boxes.**
- **Icon.**

GDI

It is library named GDI32.DLL, using GDI windows draws windows, menus and dialog boxes

- **It can create graphical output.**
- **Also use for storing Graphical Images.**


Event
Event
Device Driver
Device Driver
Msg.
Msg.
System Msg.
Queue
OS
Other
Mess
Other
Messa
Untitled - Notepad
File Edit Format View Help
Let us begin to learn Windows
Microsoft Visual C++
File Edit View Insert Project Build Tools
Window Help
Build
Debug
Find in
Ready
Application1
Application1
Msg. Queue
Application2
Msg. Queue
Application2

EXAMPLE WIN32 API

- **Application window is created by calling the API function CreateWindow().**

- **Create Window with the comments identifying the parameter.**

```
hwnd = CreateWindow
 (“classname”,                          // window class name
TEXT ("The First Program"),             // window caption
WS_OVERLAPPEDWINDOW,                    // window style
CW_USEDEFAULT,                          // initial x position
CW_USEDEFAULT,                          // initial y position
CW_USEDEFAULT,                          // initial x size
CW_USEDEFAULT,                          // initial y size
NULL,                                   // parent window handle
NULL,                                   // window menu handle
hInstance,                              // program instance handle
NULL) ;    // creation parameters, may be used to point some data for reference.
```

- **Overlapped window will be created, it includes a title bar, system menu to the left of title bar, a thick window sizing border, minimize, maximize and close button to the right of the title bar.**

- **The window will be placed in default x, y position with default size. It is a top level window without any menu.**
- **The CreateWindow() will returns a handle which is stored in hwnd.**

Why is API Design Important?

- APIs can be among a company's greatest assets
  - Customers invest heavily: buying, writing, **learning**
  - Cost to stop using an API can be prohibitive
  - Successful public APIs capture customers
- Can also be among company's greatest liabilities
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward
- Public APIs are forever - one chance to get it right

Why is API Design Important *to You*?

- If you program, you are an API designer
  - Good code is modular–each module has an API
- Useful modules tend to get reused
  - Once module has users, can't change API at will
  - Good reusable modules are corporate assets
- Thinking in terms of APIs improves code quality

Characteristics of a Good API

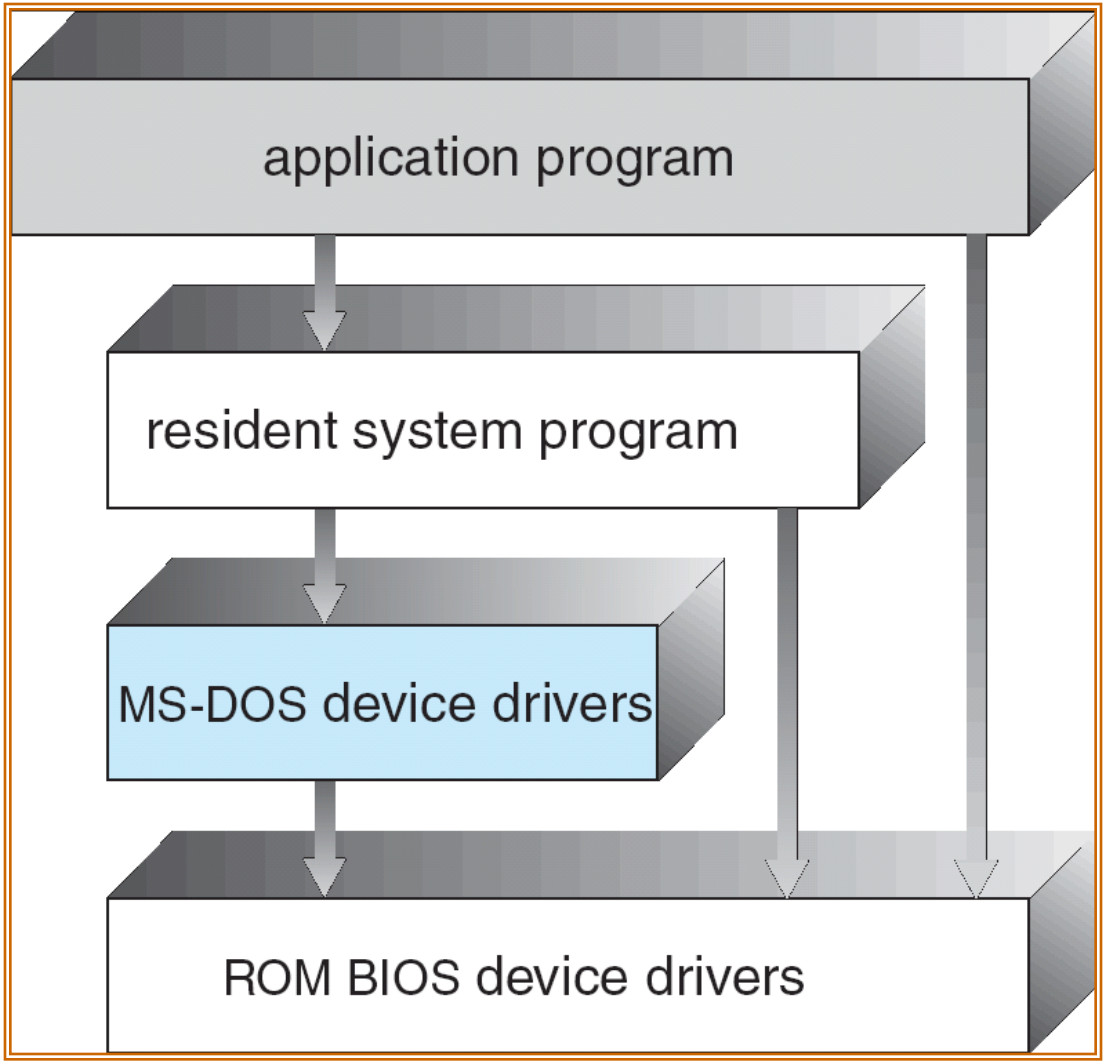
- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Operating system structures

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

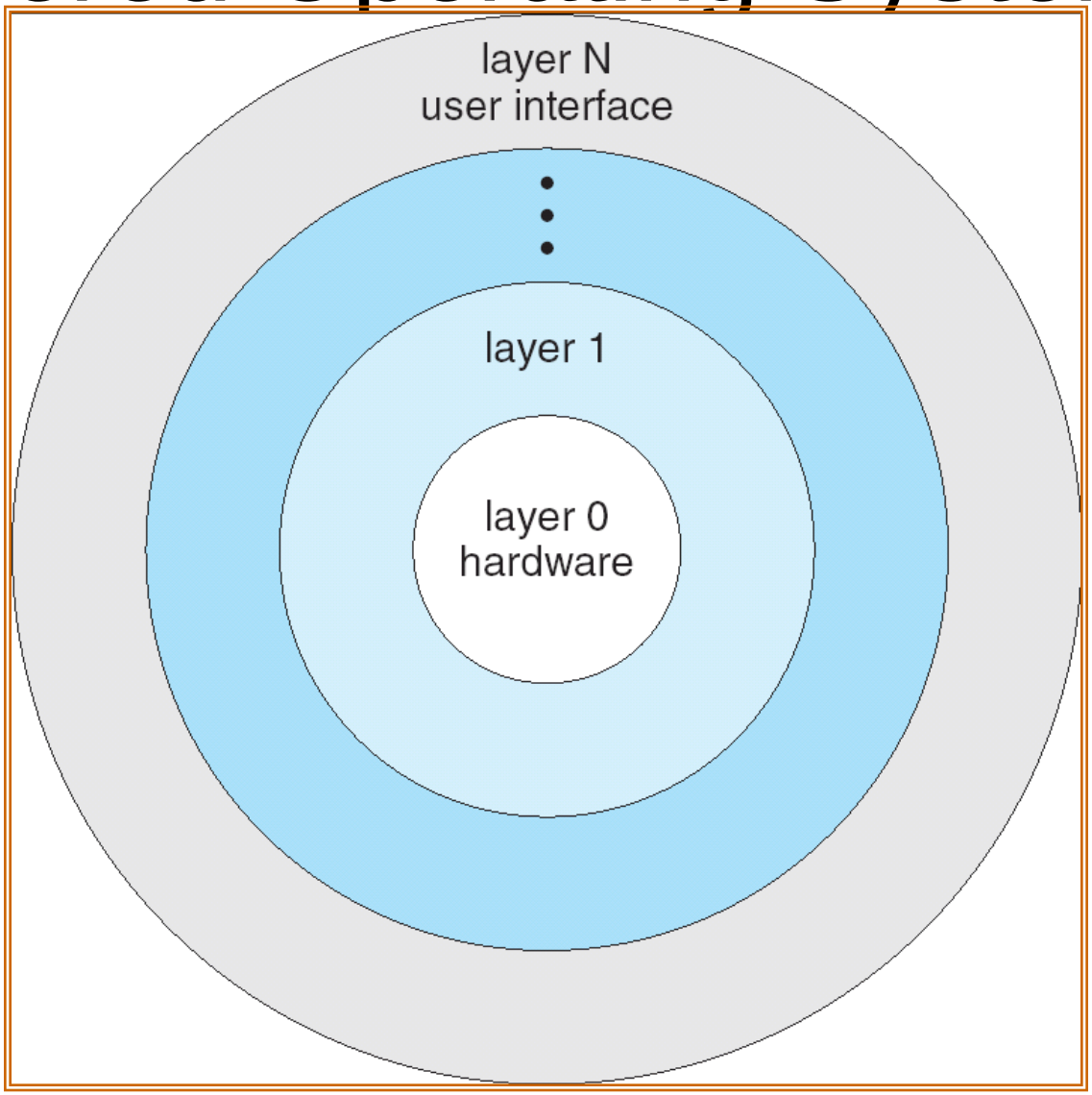
MS-DOS Layer Structure



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

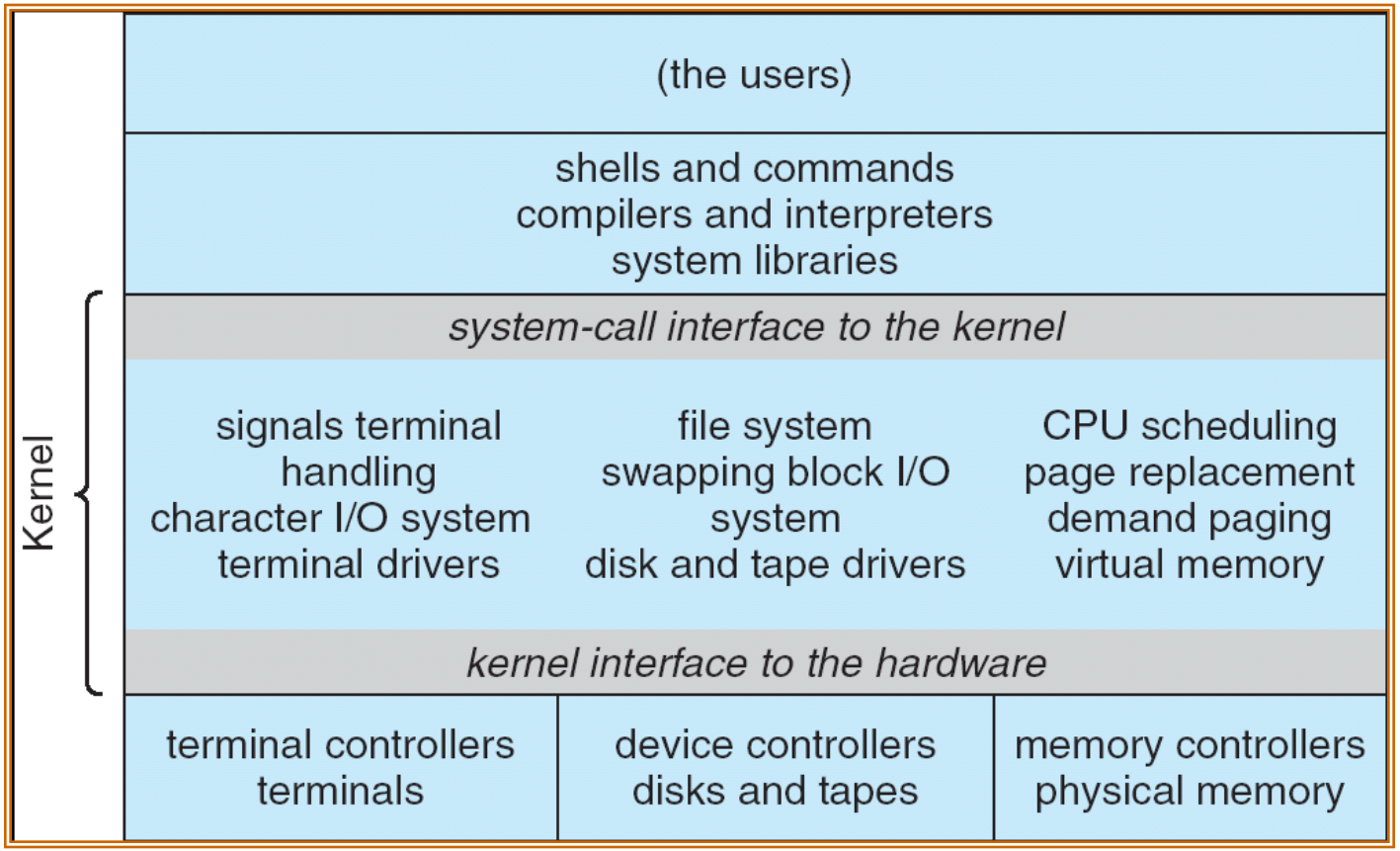
Layered Operating System



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

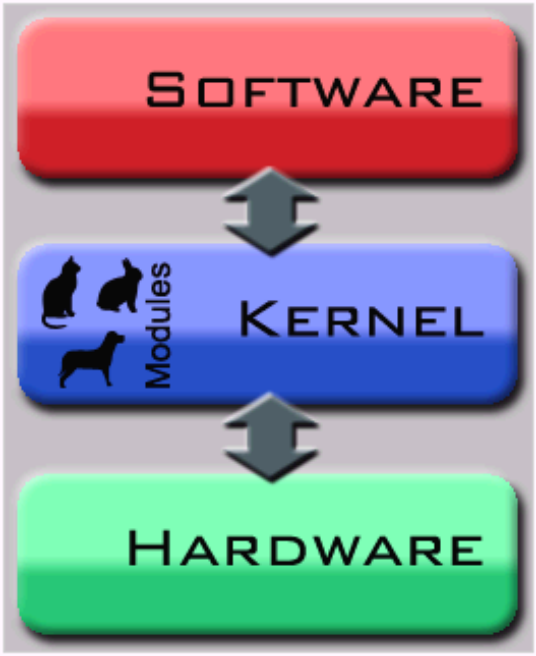
UNIX System Structure



Microkernel System Structure

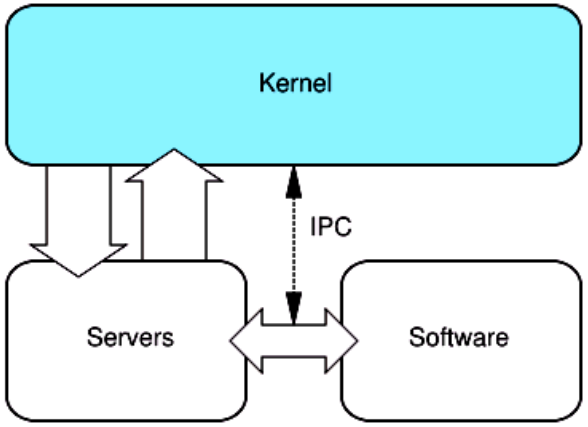
- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

- Monolithic Kernels
  - Execute all of their code in the same address space (kernel space)
  - Rich and powerful hardware access



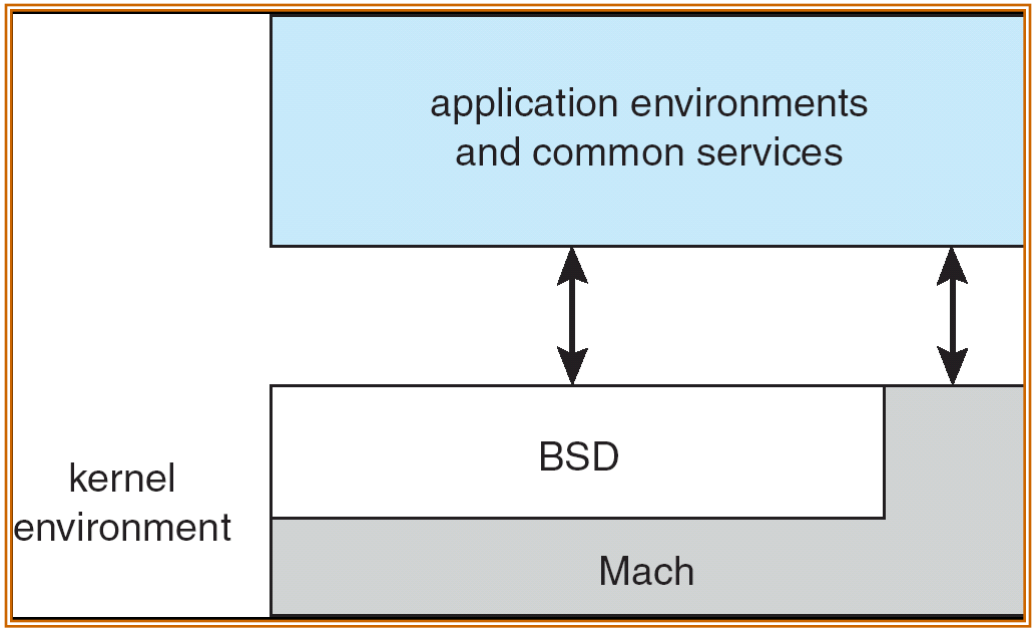
  - Disadvantages
    - The dependencies between system components
    - A bug in a driver might crash the entire system
    - Large kernels → very difficult to maintain

- Microkernels
  - Run most of their services in user space
    →improve maintainability and modularity
  - A simple abstraction over the hardware
  - A set of primitives or system calls
    - Memory management
    - Multitasking
    - IPC
  - Disadvantages
    - #(system calls) ↑
    - #(context switches) ↑



- Monolithic Kernels Versus Micro kernels
  - Most of the field-proven reliable and secure computer systems use a more microkernel-like approach
  - Micro kernels are often used in embedded robotic or medical computers where crash tolerance is important
  - Performances
    - Monolithic model is more efficient
      - IPC by: Shared kernel memory
        instead of message passing
        (Microkernel)

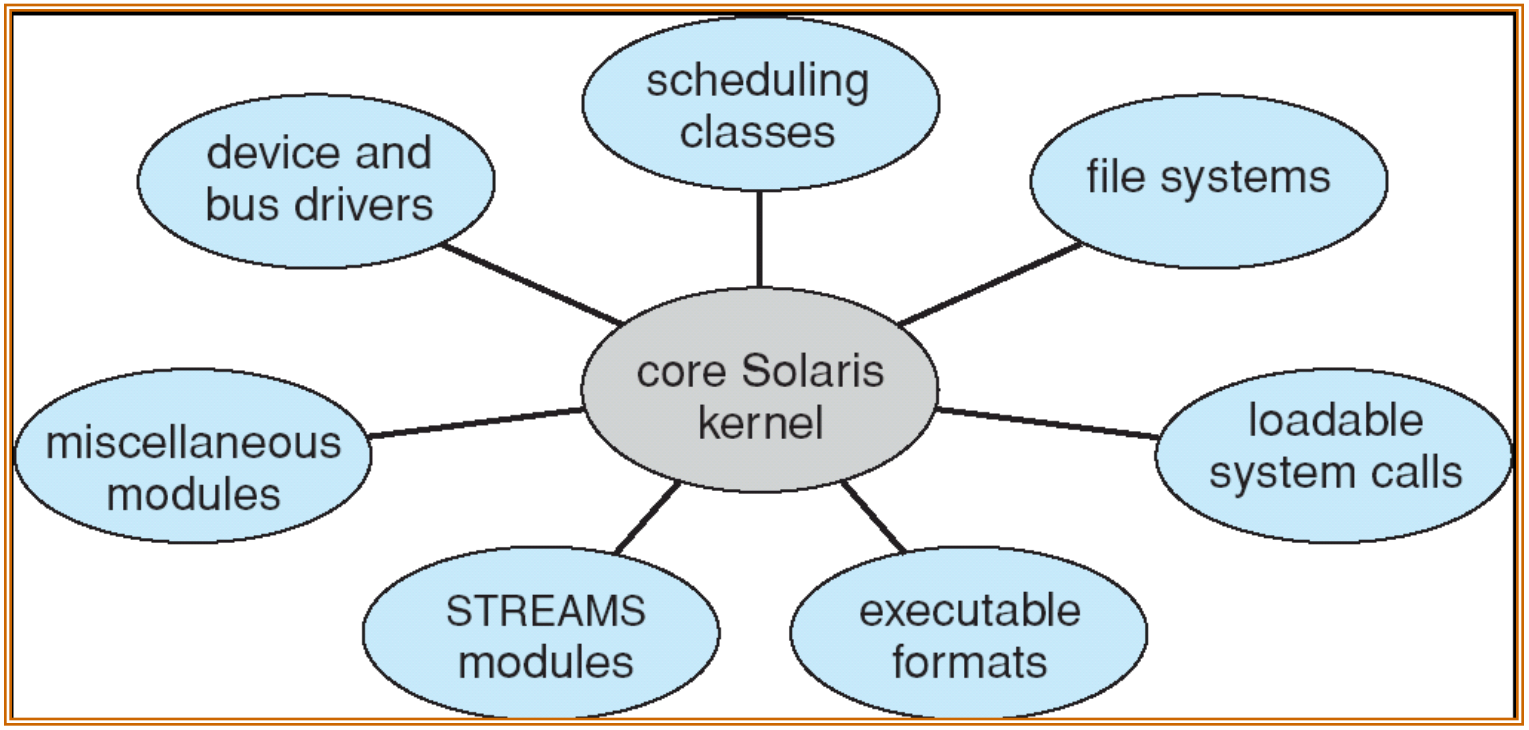
Mac OS X Structure



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

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

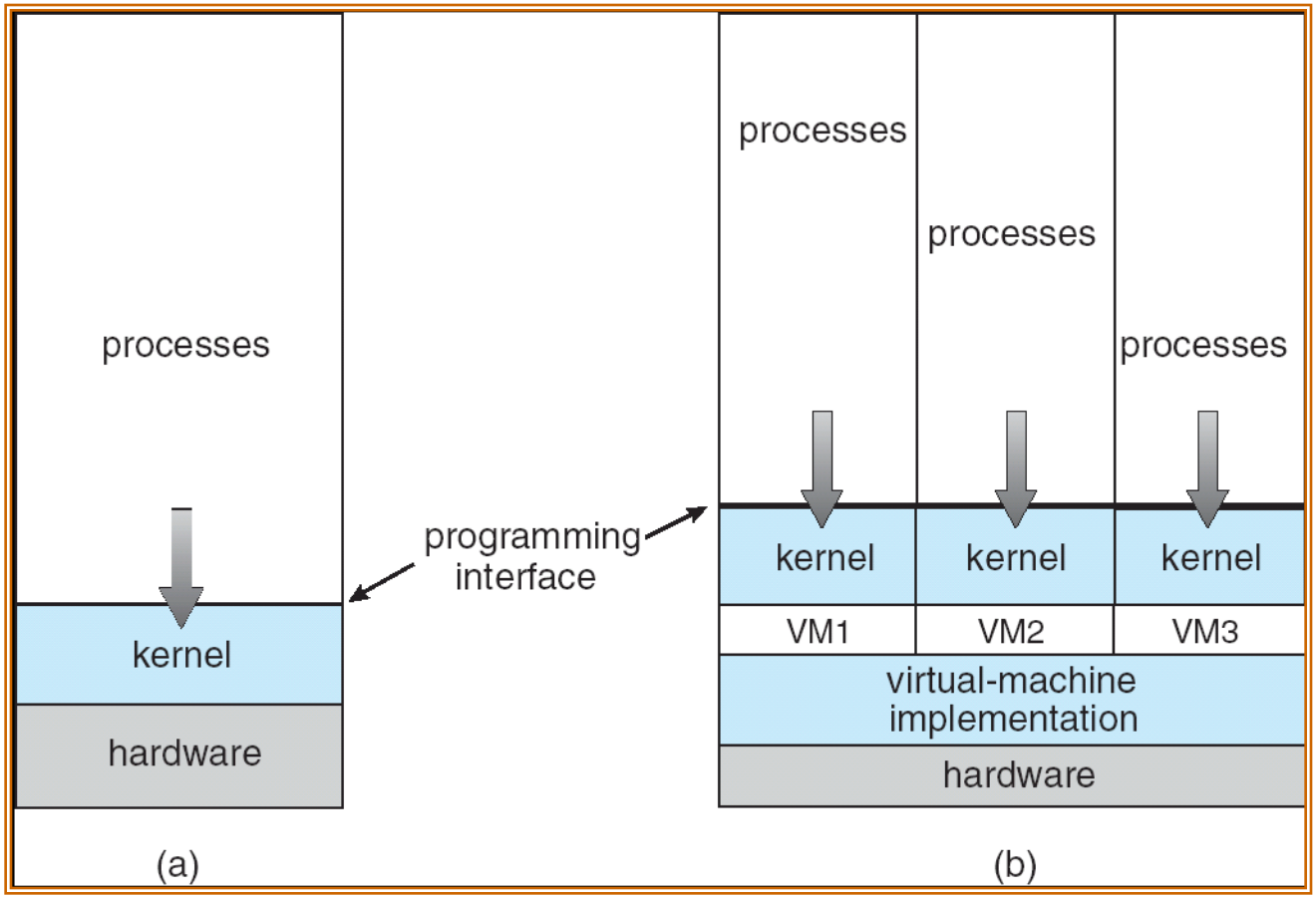
- A virtual machine provides an interface *identical* to the underlying bare hardware

- The operating system 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 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

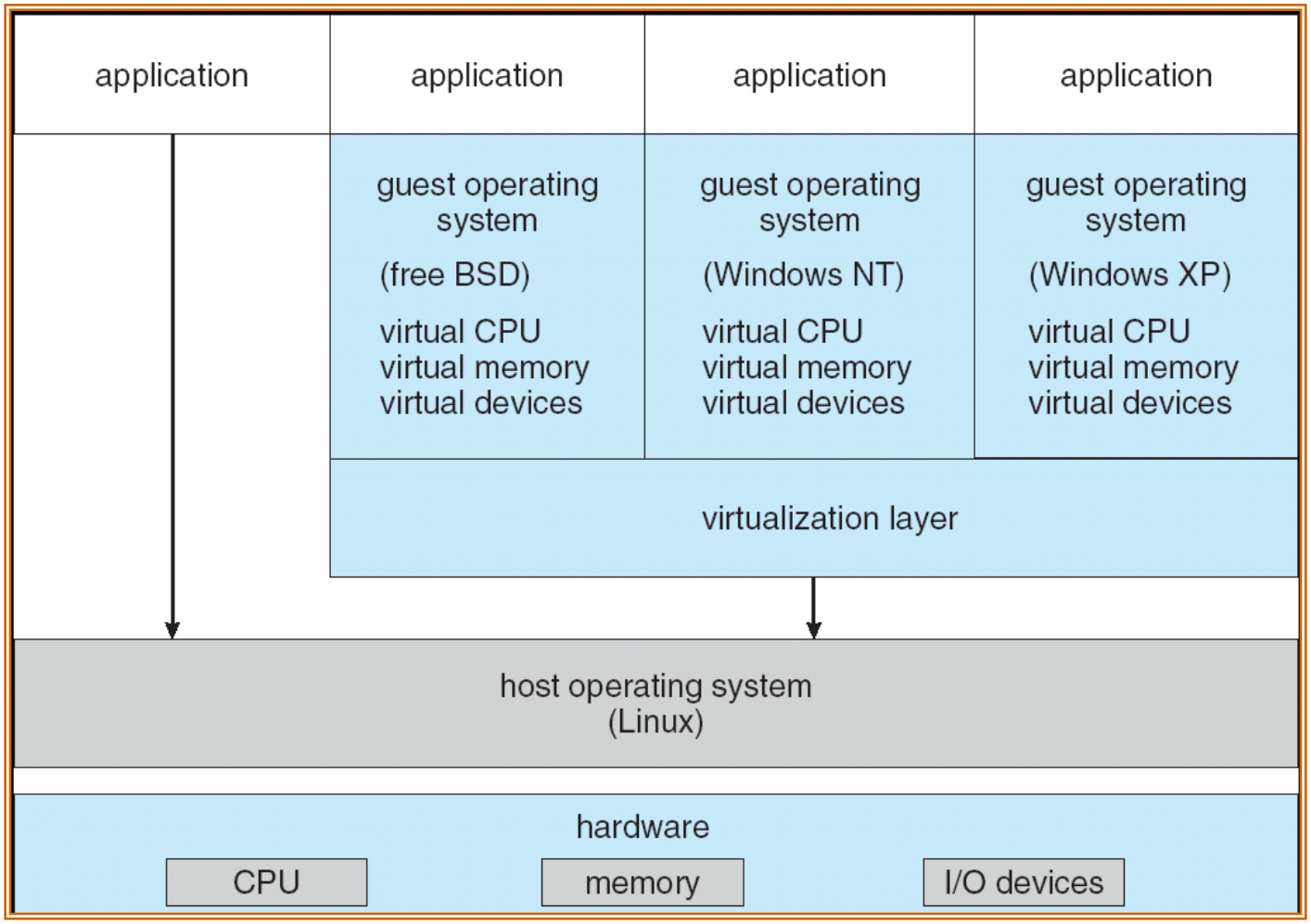
Virtual Machines (Cont.)



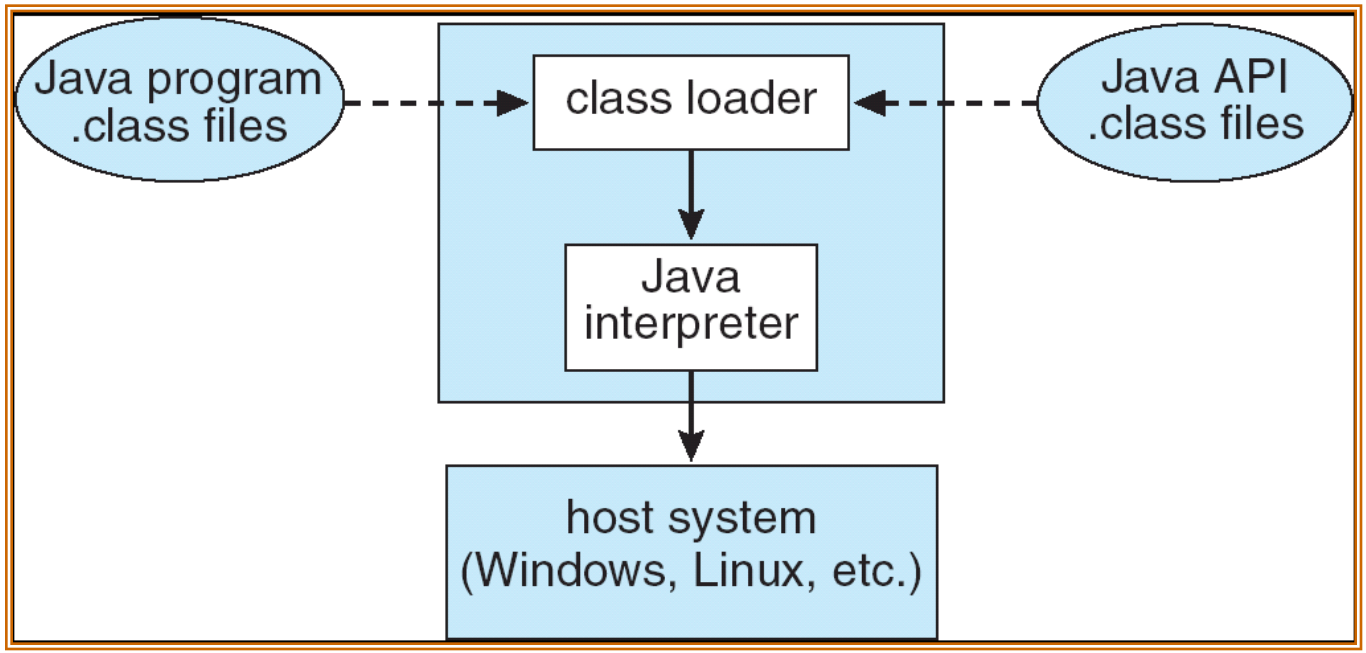
Virtual Machines (Cont.)

- 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



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

Operating system Components

OPERATING SYSTEM STRUCTURES

SYSTEM COMPONENTS

These are the pieces of the system we'll be looking at:

Process Management
Main Memory Management
File Management
I/O System Management
Secondary Management
Networking
Protection System
Command-Interpreter System

OPERATING SYSTEM STRUCTURES

SYSTEM COMPONENTS

PROCESS MANAGEMENT

A **process** is a **program** in execution: (A program is passive, a process active.)

A process has resources (CPU time, files) and attributes that must be managed.

Management of processes includes:

- Process Scheduling (priority, time management, . . .)
- Creation/termination
- Block/Unblock (suspension/resumption)
- Synchronization
- Communication
- Deadlock handling
- Debugging

OPERATING SYSTEM STRUCTURES

System Components

MAIN MEMORY MANAGEMENT

- Allocation/de-allocation for processes, files, I/O.
- Maintenance of several processes at a time
- Keep track of who's using what memory
- Movement of process memory to/from secondary storage.

FILE MANAGEMENT

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

The operating system is responsible for the following activities in connections with file management:

- File creation and deletion.
- Directory creation and deletion.
- Support of primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- File backup on stable (nonvolatile) storage media.

OPERATING SYSTEM STRUCTURES

System Components

I/O MANAGEMENT

- Buffer caching system
- Generic device driver code
- Drivers for each device - translate read/write requests into disk position commands.

SECONDARY STORAGE MANAGEMENT

- Disks, tapes, optical, ...
- Free space management (paging/swapping)
- Storage allocation (what data goes where on disk)
- Disk scheduling

OPERATING SYSTEM STRUCTURES

System Components

NETWORKING

- Communication system between distributed processors.
- Getting information about files/processes/etc. on a remote machine.
- Can use either a message passing or a shared memory model.

PROTECTION

- Of files, memory, CPU, etc.
- Means controlling of access
- Depends on the attributes of the file and user

How Do These All Fit Together?

In essence, they all provide services for each other.

SYSTEM PROGRAMS

- Command Interpreters -- Program that accepts control statements (shell, GUI interface, etc.)
- Compilers/linkers
- Communications (ftp, telnet, etc.)

OPERATING SYSTEM STRUCTURES

System Tailoring

Modifying the Operating System program for a particular machine. The goal is to include all the necessary pieces, but not too many extra ones.

- Typically a System can support many possible devices, but any one installation has only a few of these possibilities.
- **Plug and play** allows for detection of devices and automatic inclusion of the code (drivers) necessary to drive these devices.
- A **sysgen** is usually a link of many OS routines/modules in order to produce an executable containing the code to run the drivers.

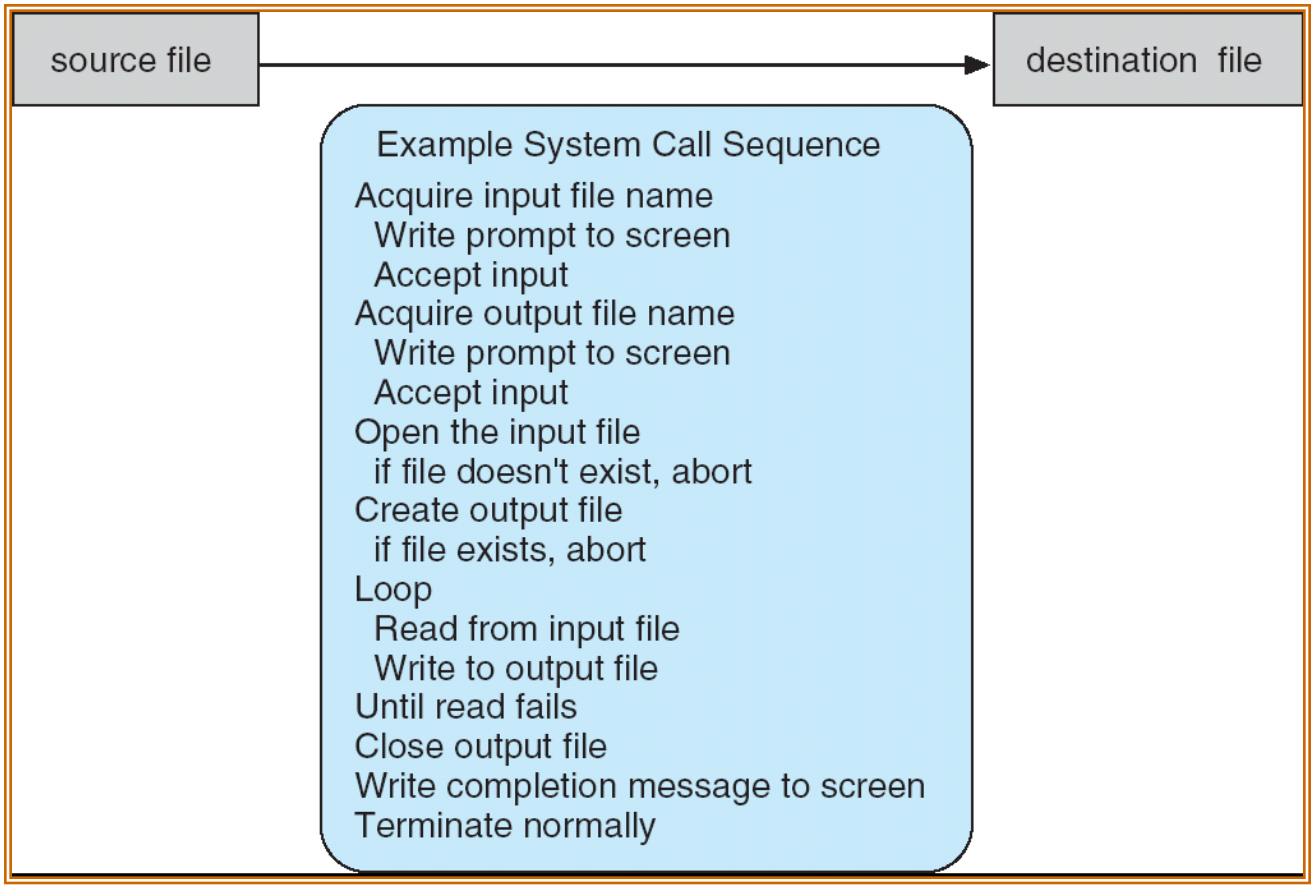
SYSTEM CALLS

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?

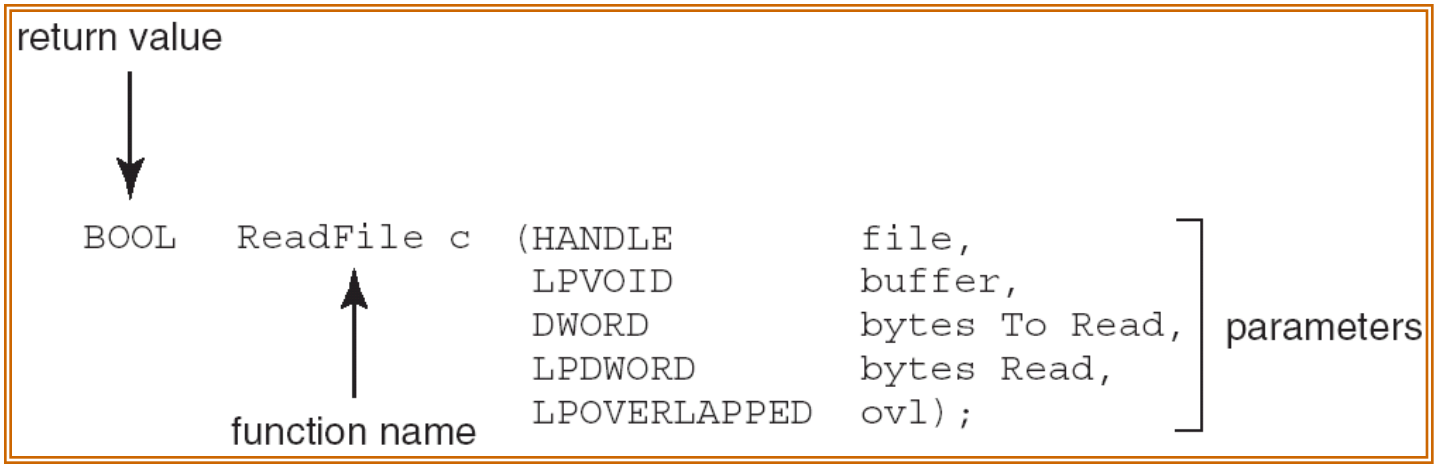
Example of System Calls

- 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

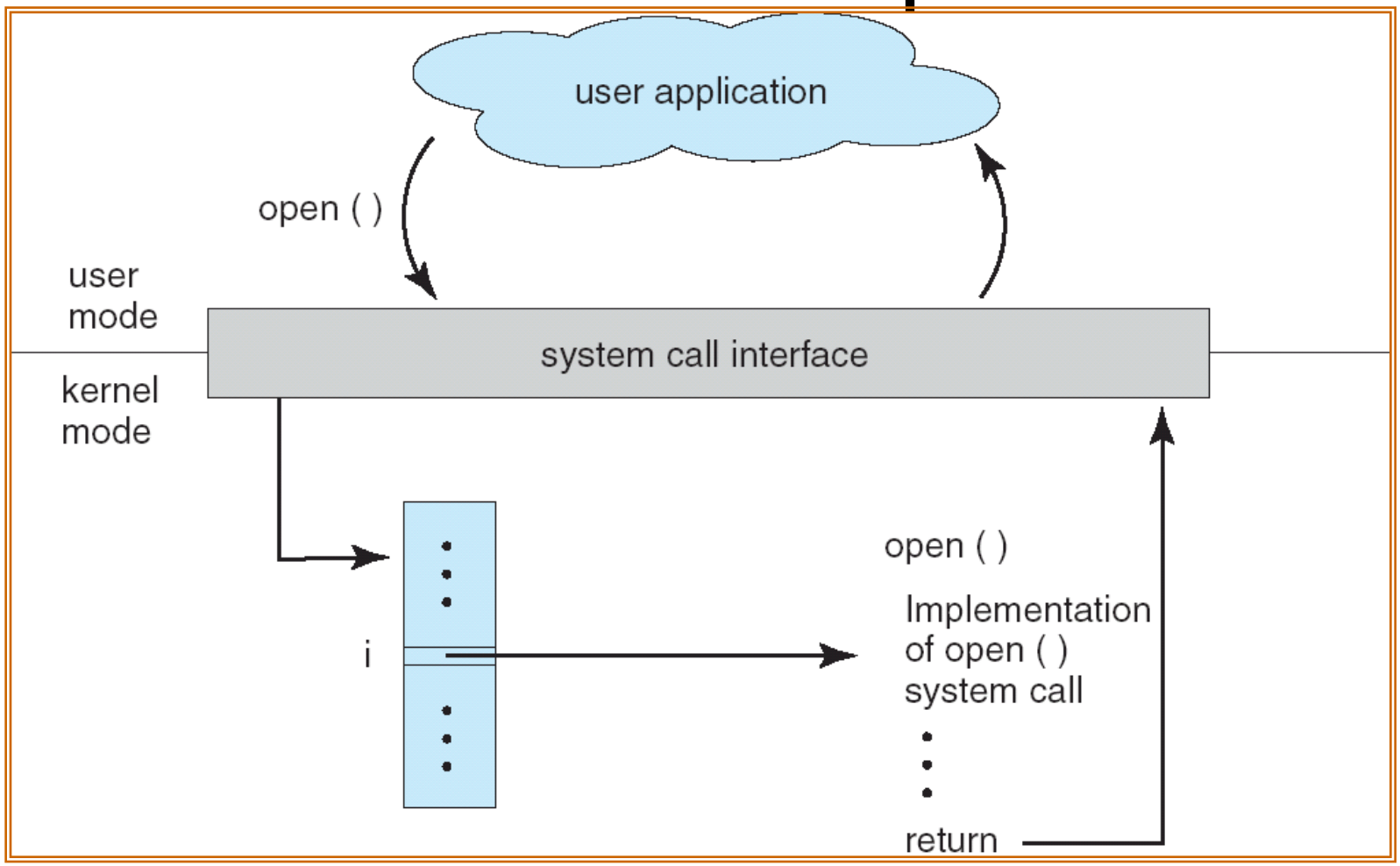


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read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

System Call Implementation

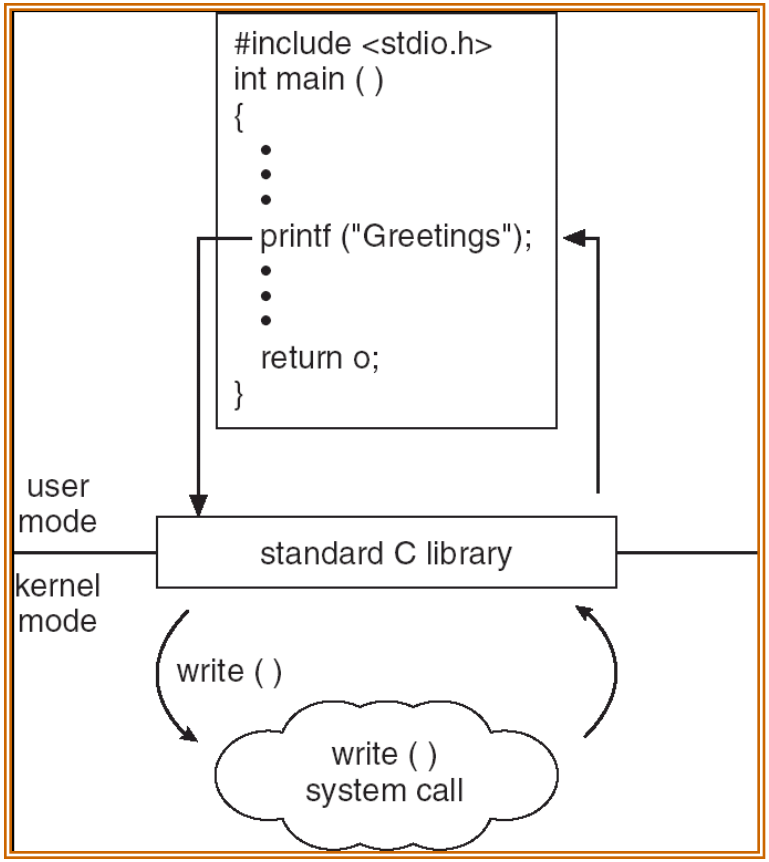
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about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

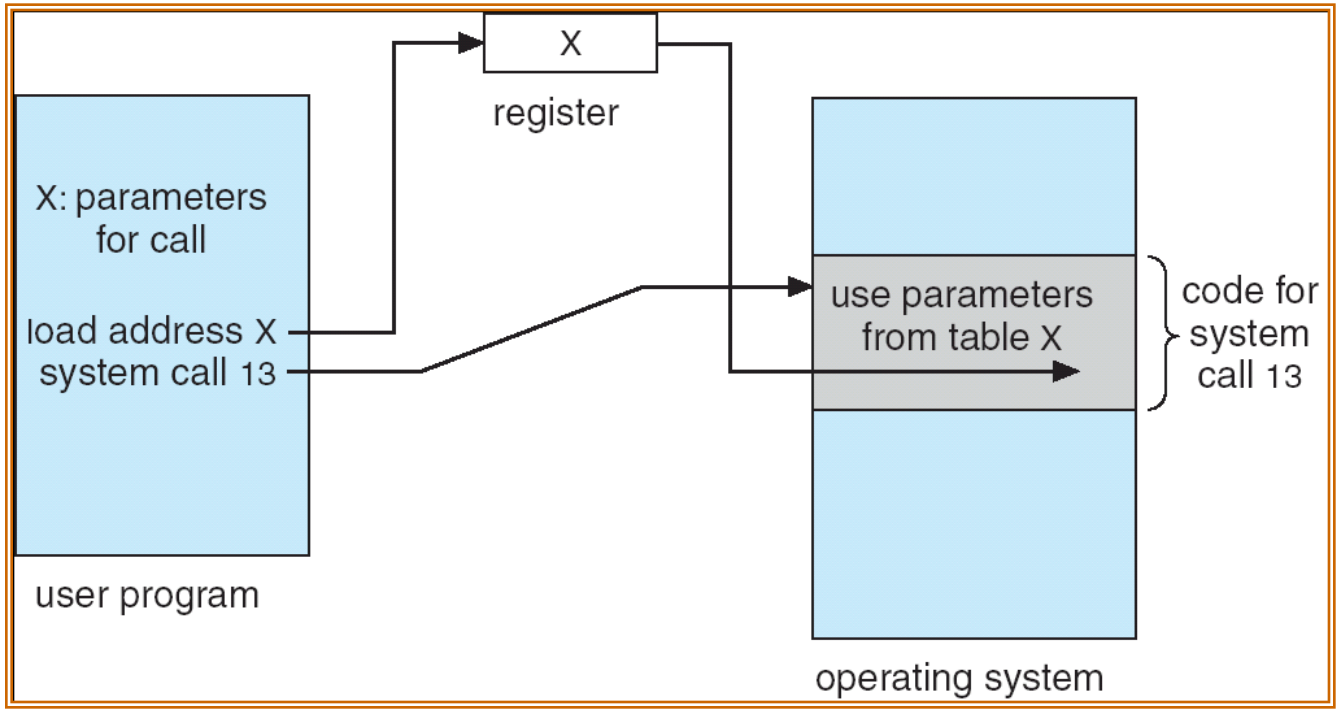
- C program invoking printf() library call, which calls write() system call



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

Reading Assignment

- What is the difference Between System Call & API?
- Can we interpret push & pop as an API in a stack program?
- Differentiate between Micro kernel & Monolithic kernel.