Chapter 2: 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
- System Boot

Objectives

- To describe the **services** an operating system provides to users, processes, and other systems
- To discuss the various ways of **structuring** an operating system
- To explain how operating systems are **installed and customized** and how they 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 interface
  - **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.

Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont):
  - **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

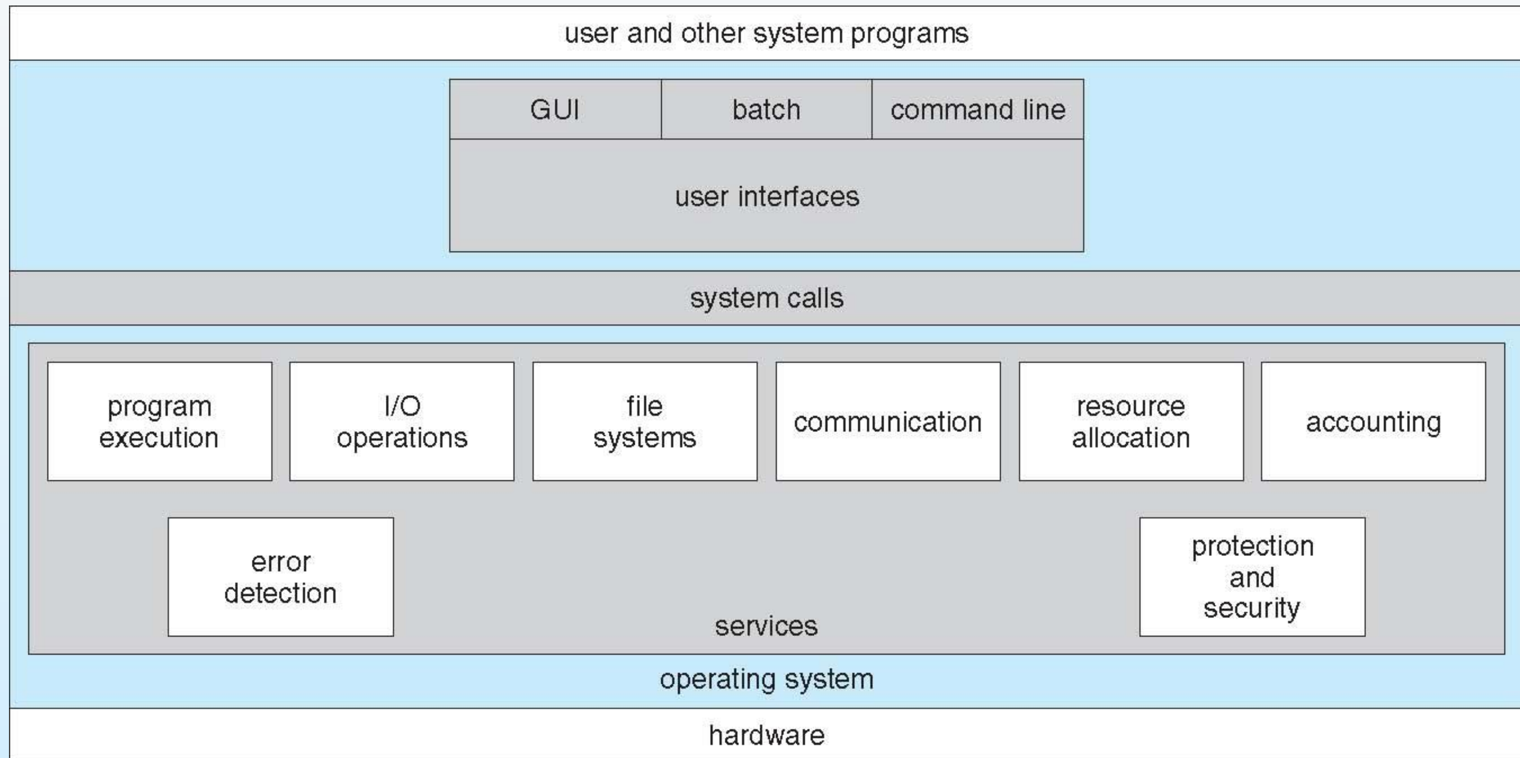
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - **Many types of resources** - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources

Operating System Services (Cont.)

- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** 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.

A View of Operating System Services



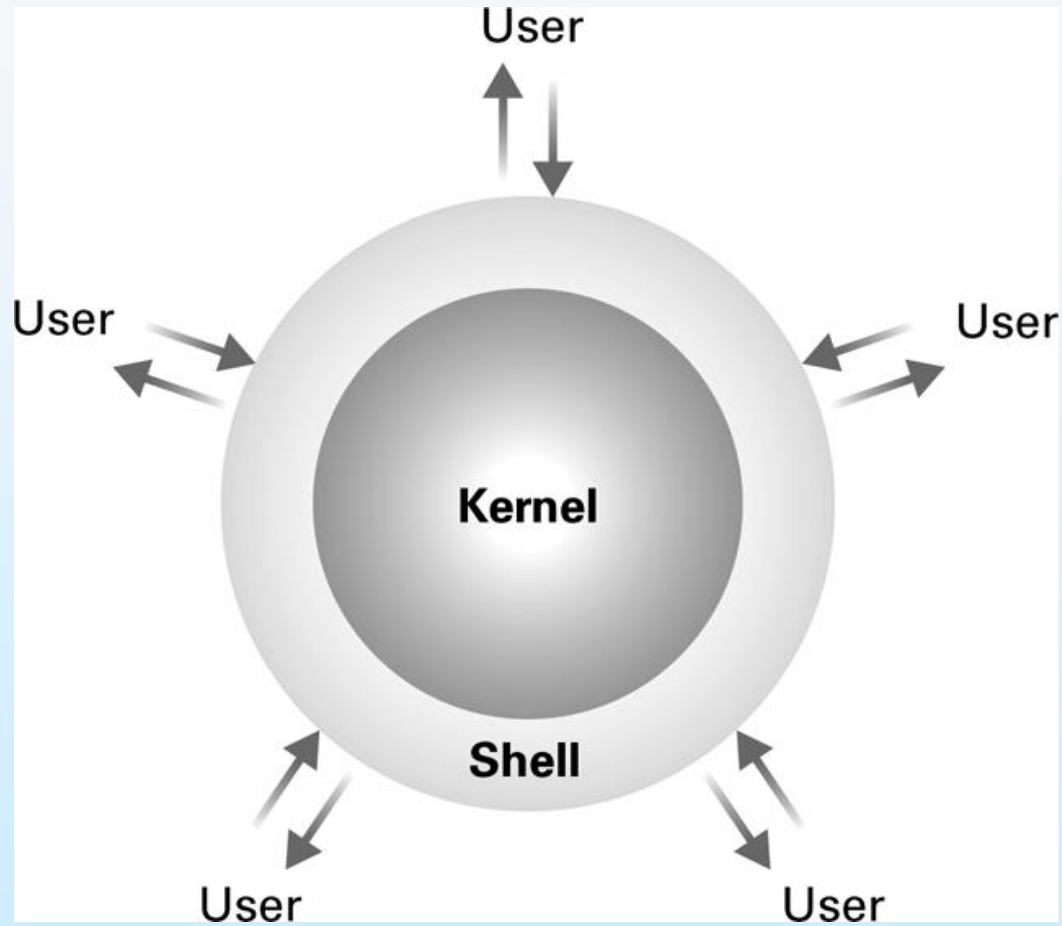
Chapter 2: Operating-System Structures

- 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
- System Boot

User Operating System Interface - CLI

- CLI is also called **command interpreter**
- CLI allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
    - Windows XP and Unix treat CLI as special programs
  - Sometimes multiple flavors implemented – **shells**
    - UNIX and Linux: Bourne Shell or C Shell
- 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
- UNIX command to delete a file
  - rm filename.txt (e.g. bankaccounts.txt)

Shell



Shells

```
52k CP/M vers 2.2.2.15
Micro Technology BIOS

Set for 8sec drive spin-time
A>dir
A: MOVCPM   COM : PIP      COM
A: SUBMIT   COM : XSUB     COM
A: ED       COM : ASM      COM
A: DDT      COM : LOAD     COM
A: STAT     COM : DUMP     COM
A: DUMP     ASM : SYSGEN   COM
A: BACKUP   COM : CMT      COM
A: CONFIG   COM : CONSOLE  COM
A: COPY     COM : DEL      COM
A: EJECT    COM : FILES    COM
A: FORMAT   COM : INIT     COM
A: IODEFS   ASM : TIME     COM
A: CPM780   COM : SPIN87   COM
A: SOFTCEN7 COM : DISCMOD3 COM
A: DISX     COM : REENTER  COM
A: UNERASE  COM : VERIFY   COM
A: DPB      COM
A>
```

C:\WINDOWS\system32\cmd.exe

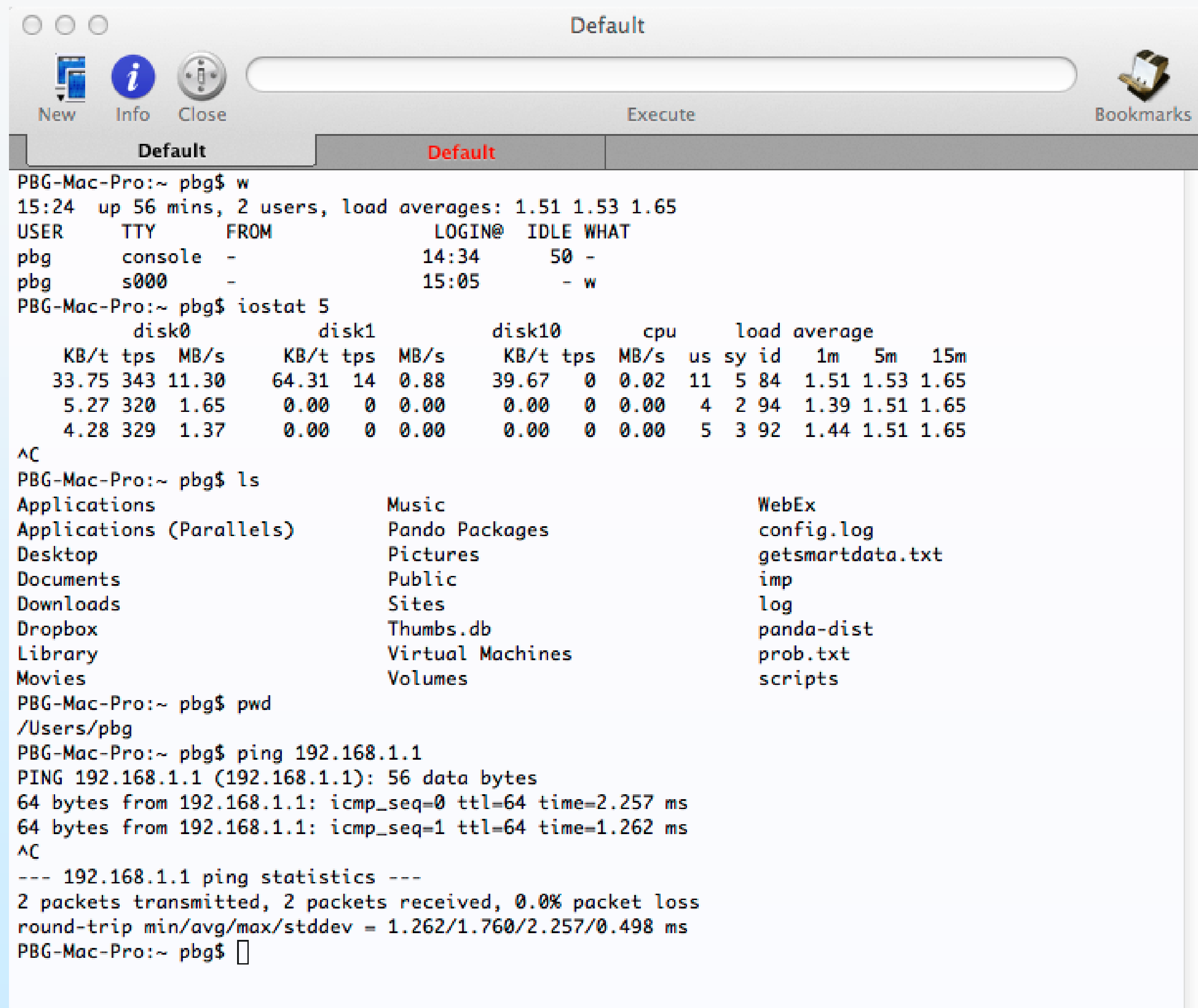
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Cay>
```

LXTerminal

File Edit Tabs Help

```
arvind[~]$
```

Bourne Shell Command Interpreter



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 1973
- 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)
  - GNU project: KDE and GNOME
    - Run on Linux and various versions of UNIX

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands.

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
- System Boot

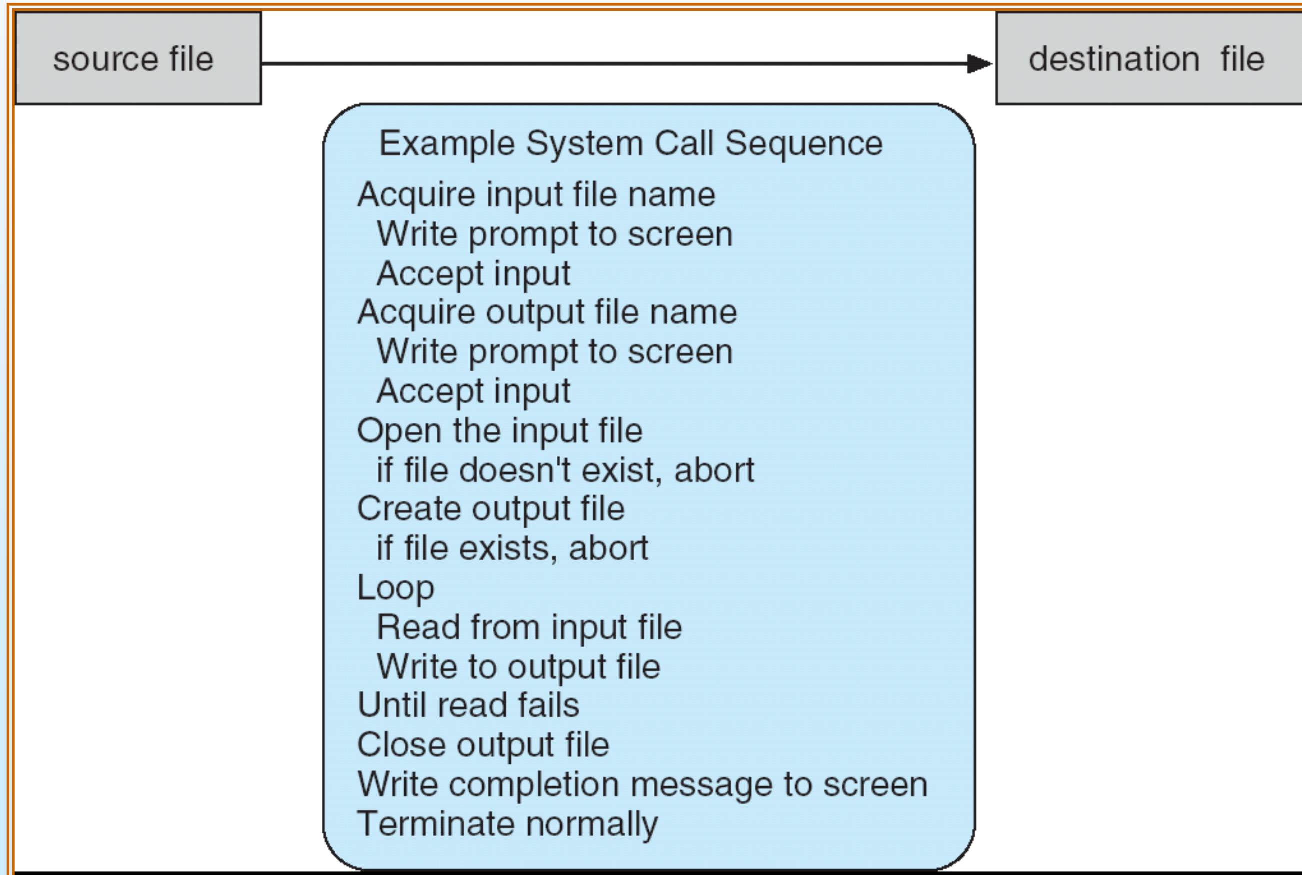
System Calls

- **Programming interface** to the services provided by the OS
- Typically written in a high-level language (C or C++)
  - Sometimes, if hardware access is necessary, these are written in assembly language.
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs:
  - 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)
- Why use APIs rather than system calls?

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

Example of System Calls

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



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

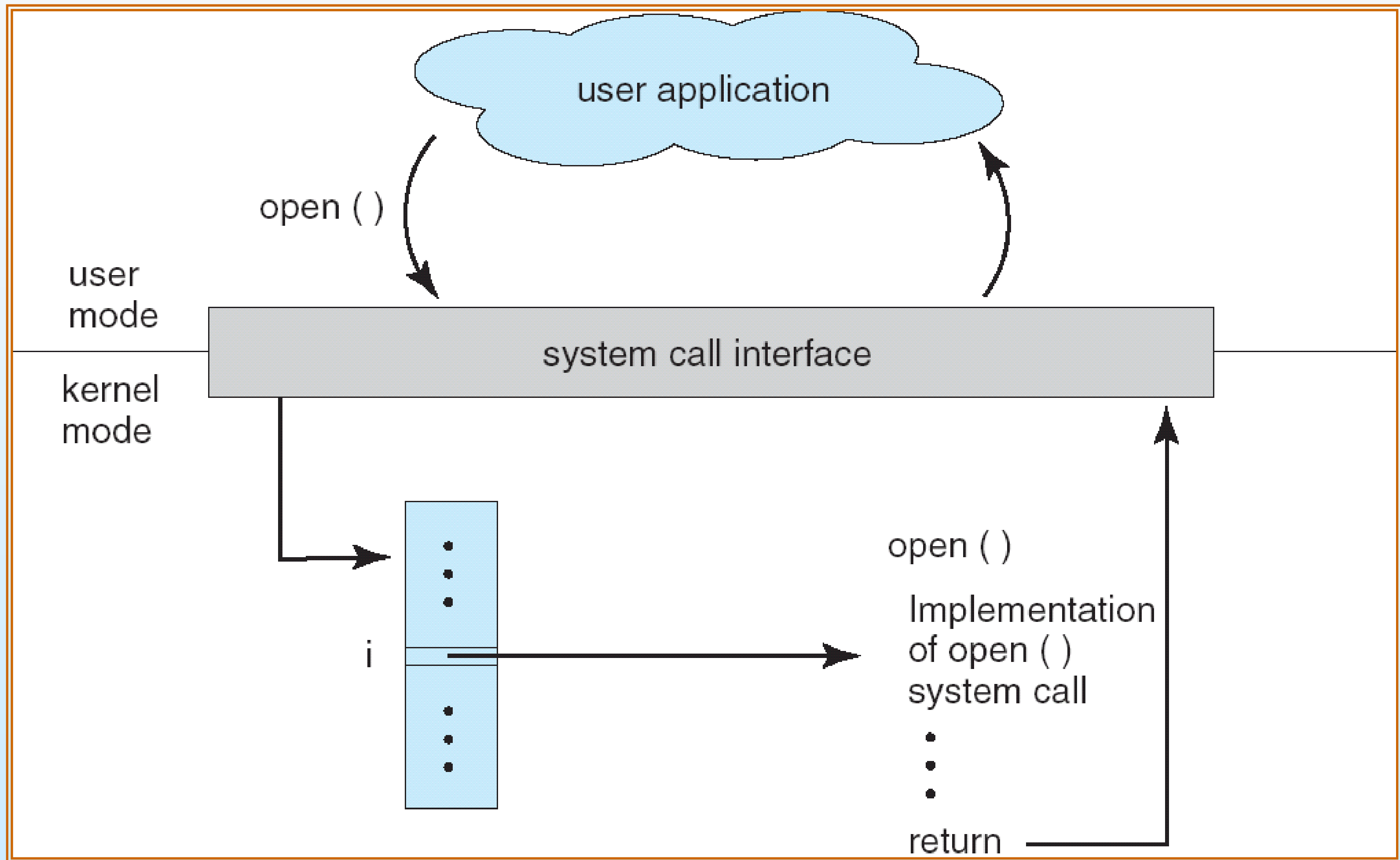
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation

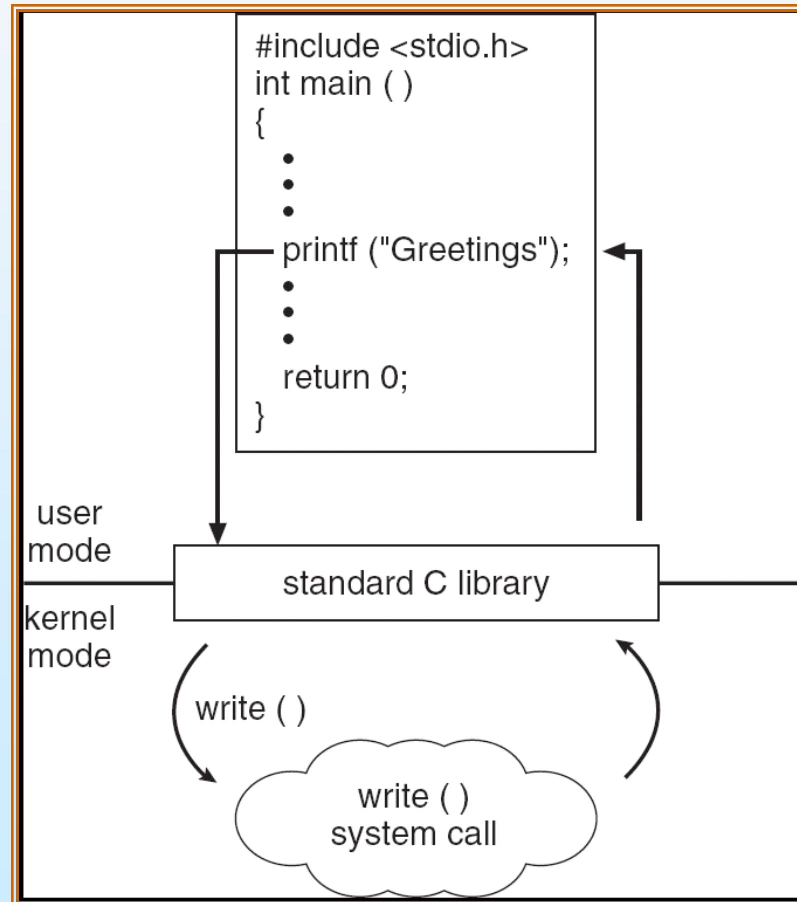
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- An API function is typically **implemented** by invoking a system call.
  - Win32: CreateProcess() invokes NTCreateProcess in the Windows Kernel.
- The system call 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 of 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)
- If any other system **supports the same API** then an application can run normally among different architectures.

API – System Call – OS Relationship
System Call Interface



Standard C Library Example

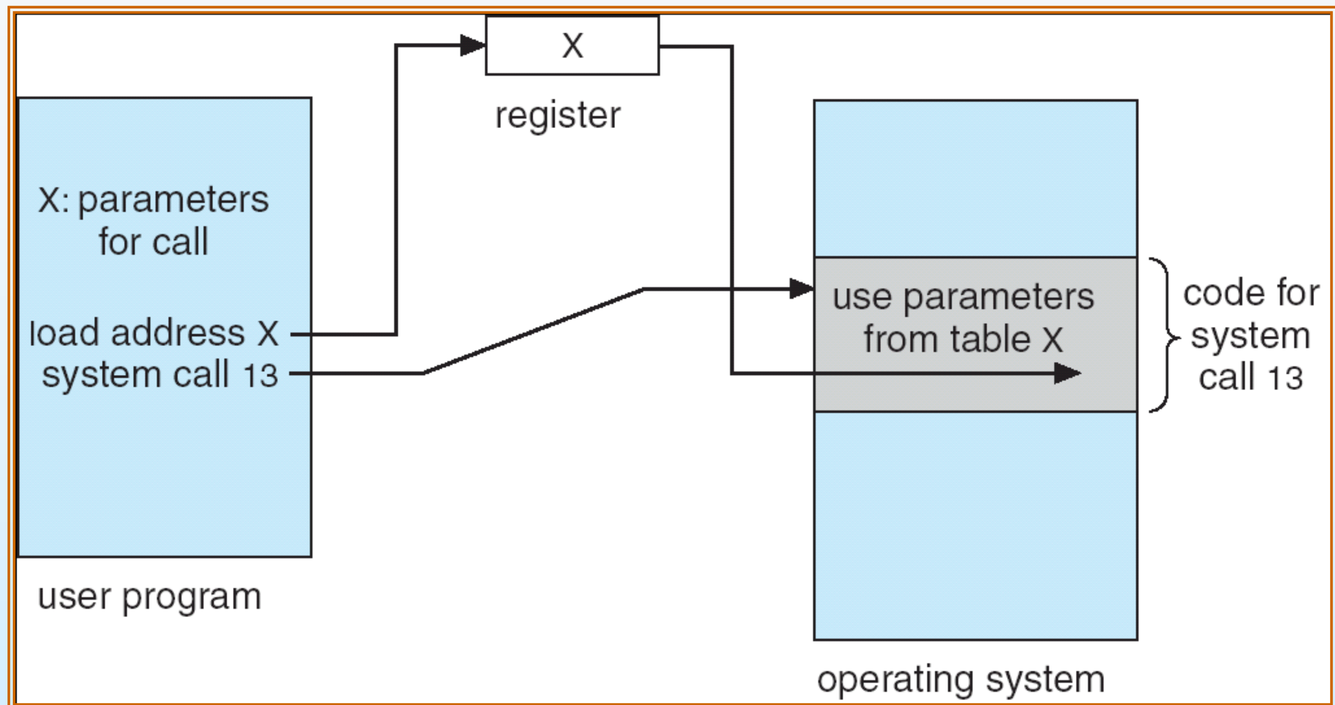
- C program invoking printf() library call, which calls write() system call



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, there 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



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
- System Boot

Types of System Calls

- **Process control**
  - End, abort, execute, create, wait for time, allocate and free memory
- **File management**
  - create file, delete file, open close, read, write, get file attributes
- **Device management**
  - Request device, release device, read, write, logically attach or detach devices
- **Information maintenance**
  - Get time or date, set time and date
- **Communications**
  - Create or delete communication connection
  - Transfer status information
  - Attach or detach remote devices

Process Control

- A running program normally terminates with **"end()".**
- If the program causes a problem an error message is generated and
  - The execution is halted with **"abort()"**
  - A **dump** in memory is taken
  - The debugger may be started to correct bugs
- The OS transfers control to the invoking command interpreter
- Execution then depends on the type of the interaction:
  - Interactive systems: user is given a pop-window for feedback
  - Batch: the command interpreter terminates the entire job and continues with the next job.

Process Control

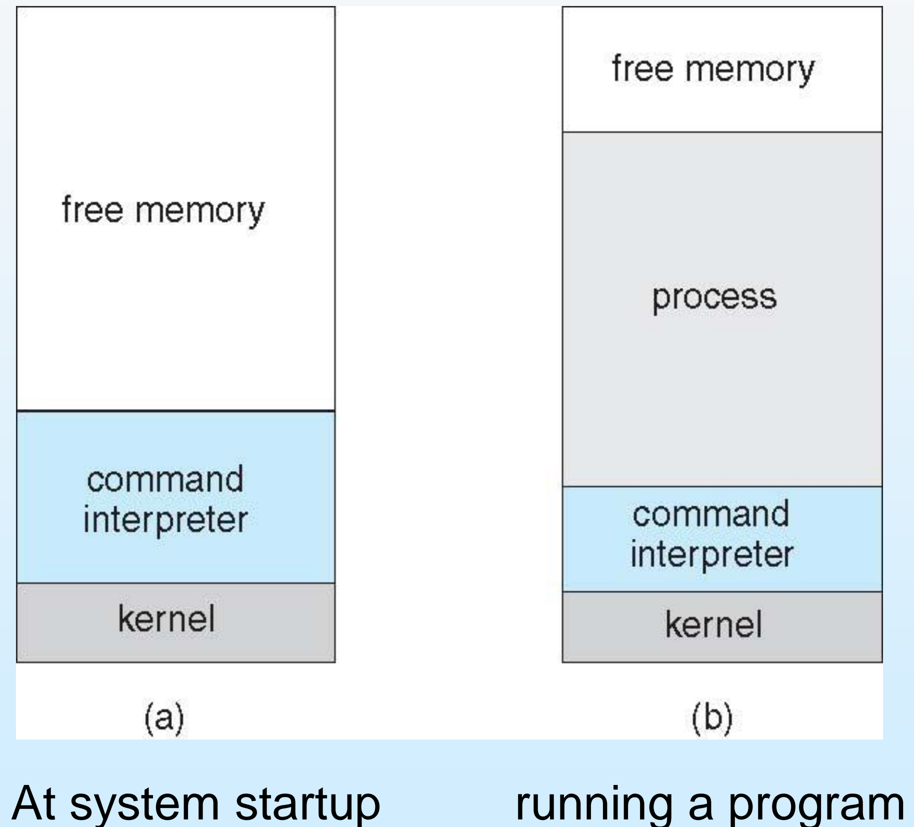
- After creating a process we need to **control it's execution**
  - get process attributes
  - set process attributes
  - terminate process
- **Waiting** before terminating
  - wait time
  - wait event
  - signal event
- **Time profile**
  - Indicates the amount of time a process spends executing at particular locations.

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()

Example: MS-DOS

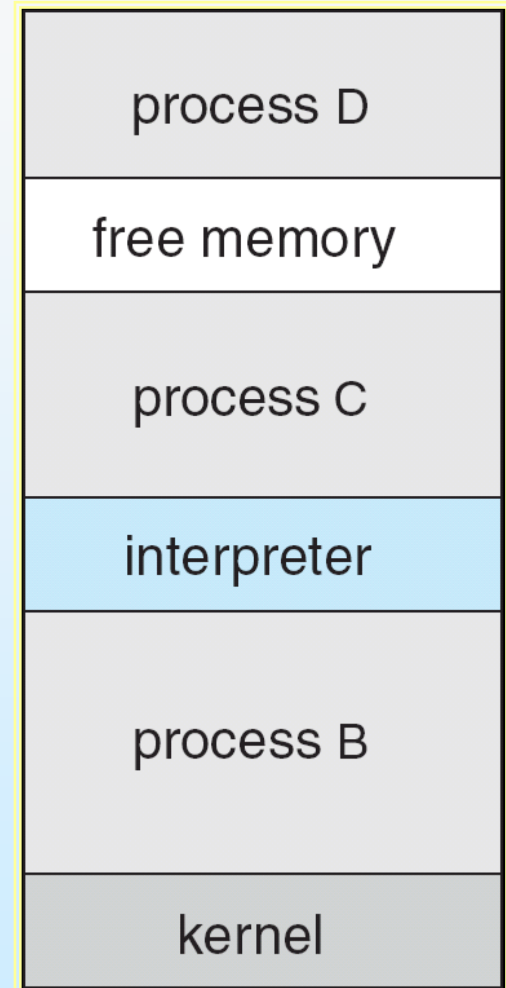
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



At system startup running a program

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



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
- System Boot

System Programs

- System programs provide a convenient environment for program development and execution. These 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

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
- System Boot

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
- Implementing an OS is **highly creative** but no book will tell you how to do it ☹
- However, **principles** of software engineering still hold ☺

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

OS implementation

- Once an OS is designed it must be implemented
  - Once written in assembly language, now most often in C and C++
  - The first system not written in assembly was MCP (Master Control Program) written in ALGOL.
  - MULTICS was written in Pl/1
  - Linux and Windows are mostly written in C, some parts in **assembly** regarding device drivers and register state restore.
- An OS is **more portable** if written in high-level language
  - MS-DOS was written in **Intel 8088 assembly language**
  - Linux written in C: available on Intel80x86, Motorola 680X0. SPARC and MIPS RX000.
    - More portable, but slower and increased storage requirements
- **Emulation** can allow an OS to run on non-native hardware

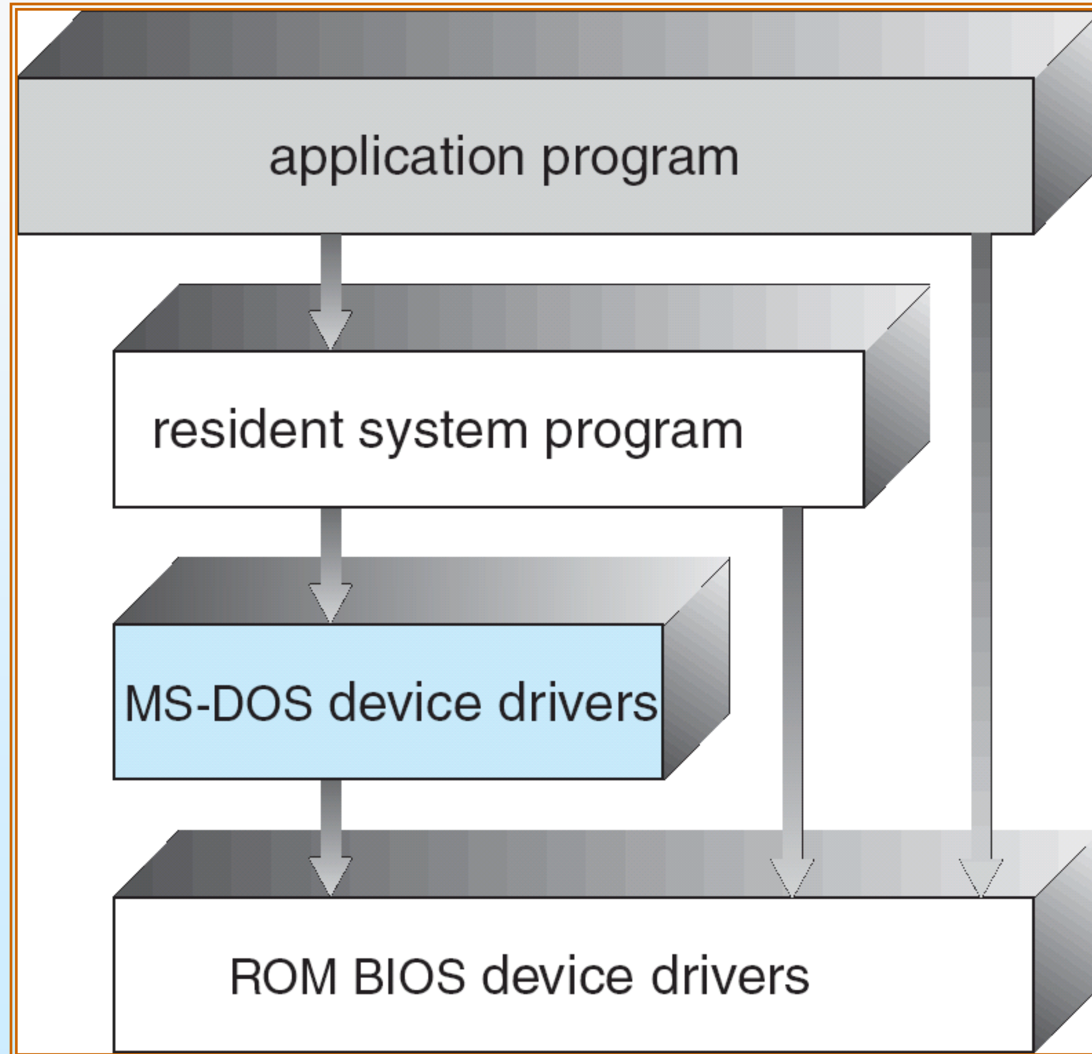
Chapter 2: Operating-System Structures

- 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
- System Boot

Simple Structure

- MS-DOS – written to provide the **most functionality in the least space**
  - Not divided into modules (**monolithic**)
  - Although MS-DOS has some structure, its interfaces and levels of functionality are **not well separated**
    - For example, application program can access basic I/O routines to write directly to the display and disk drives.
    - This makes MS-DOS vulnerable since a program can cause a crash of the system
  - **Basic hardware is accessible from high levels!**

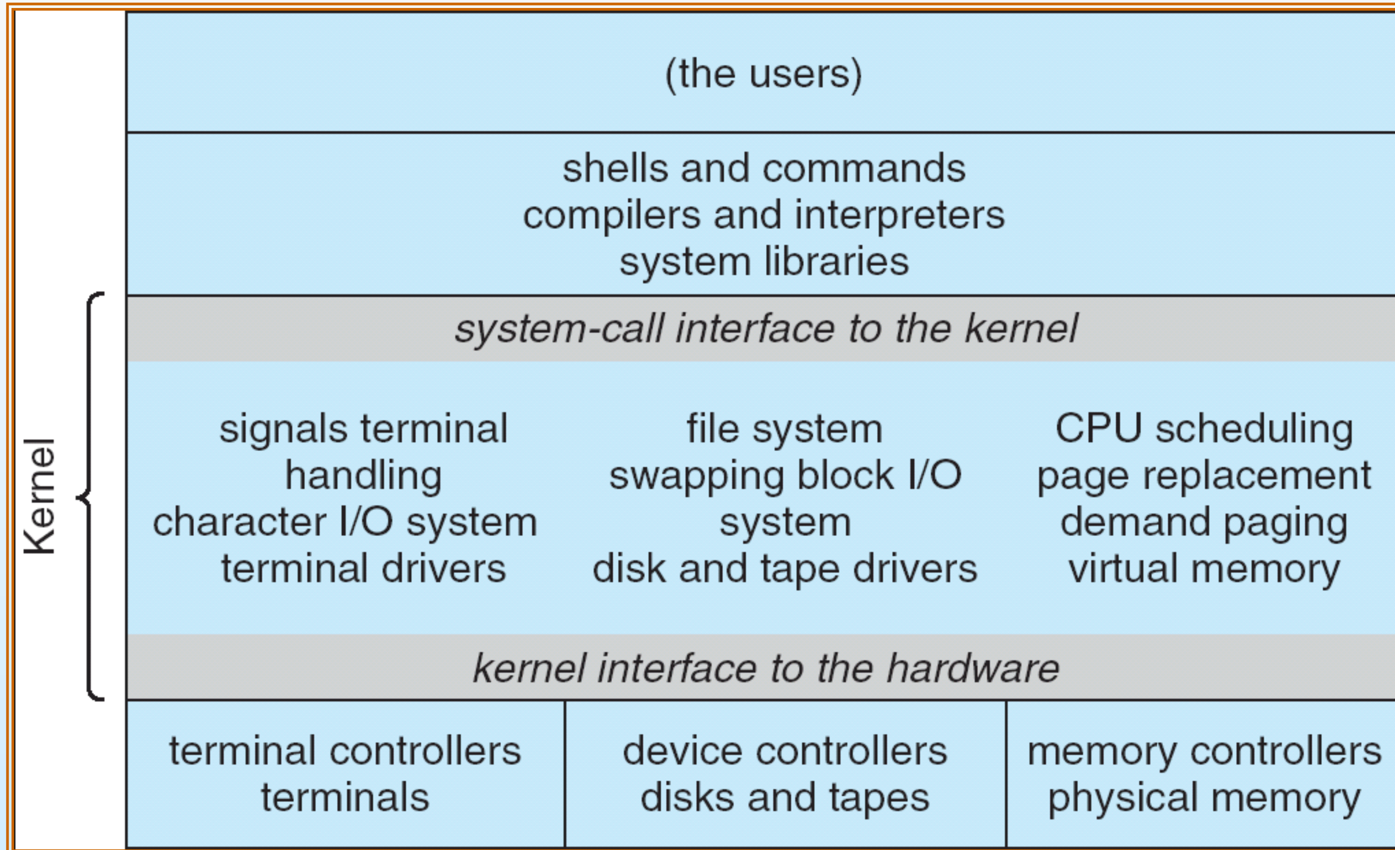
MS-DOS Layer Structure



UNIX structure

- 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



Two parts: **system programs** and the **kernel**

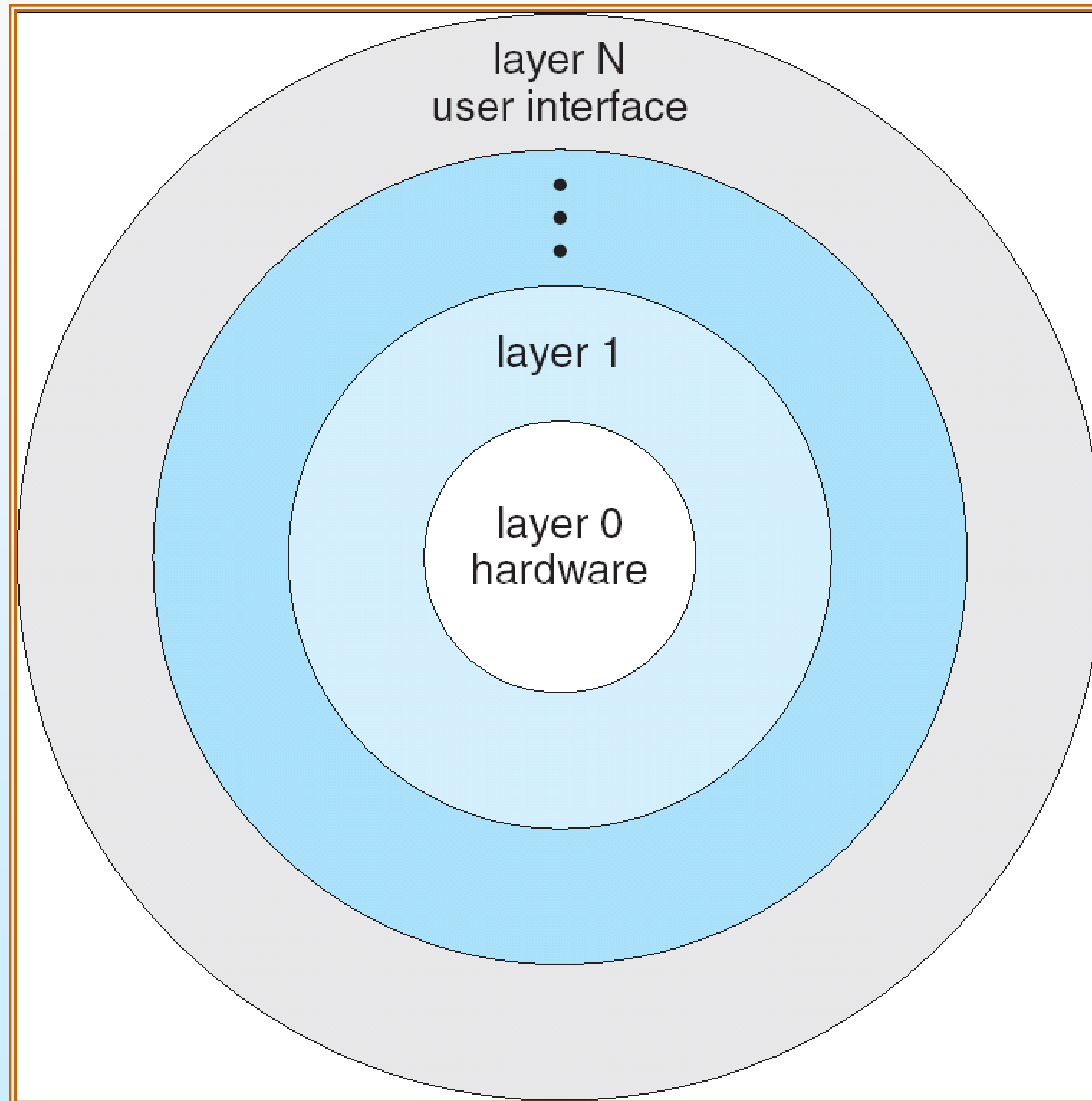
Kernel is further separated in a set of interfaces and device drivers

- Everything below the system call interface and above the physical hardware is the **Kernel**

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
- Advantages
  - **Simplicity** of construction and debugging
  - Each layer is implemented only with the operations provided by the lower level
    - Information hiding
- Disadvantages
  - Appropriately defining of various layers
  - Planning of layers
  - Tend to be **less efficient:** each layer adds overhead to the system call

Layered Operating System



Venus Layer Structure

layer 6:	user programs
layer 5:	device drivers and schedulers
layer 4:	virtual memory
layer 3:	1/0 channel
layer 2:	CPU scheduling
layer 1:	instruction interpreter
layer 0:	hardware

Figure 3.10 Venus layer structure.

OS/2 layer structure

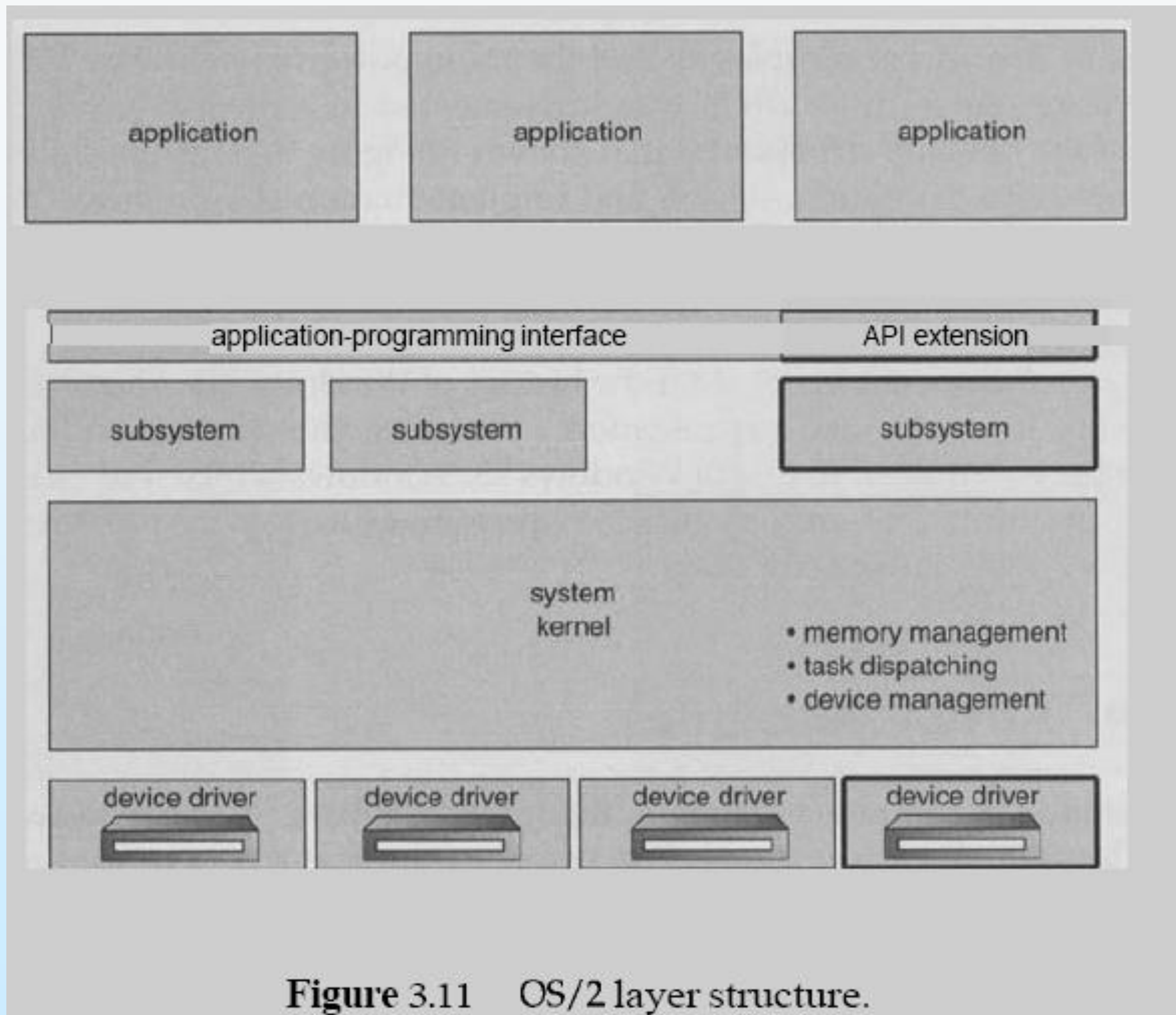
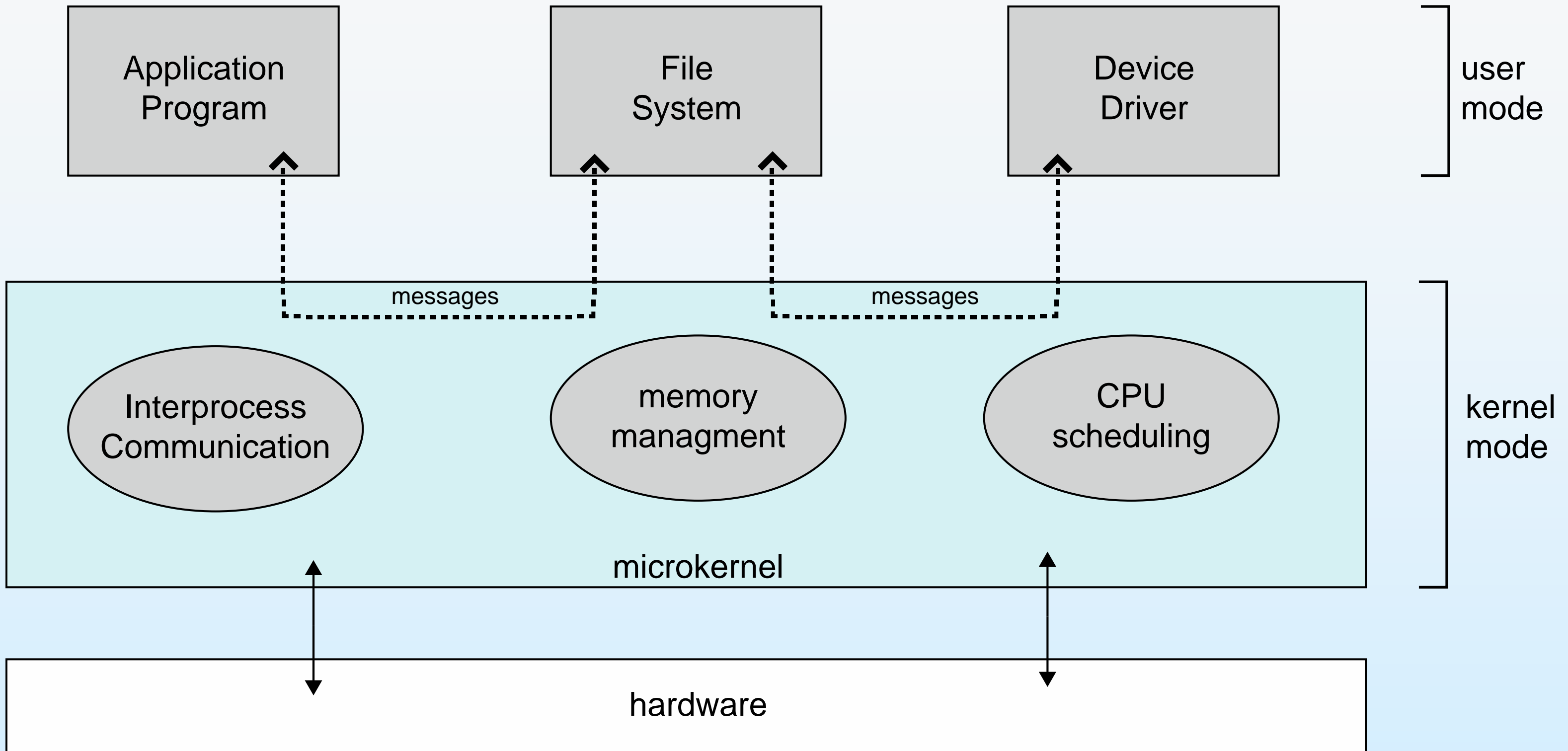


Figure 3.11 OS/2 layer structure.

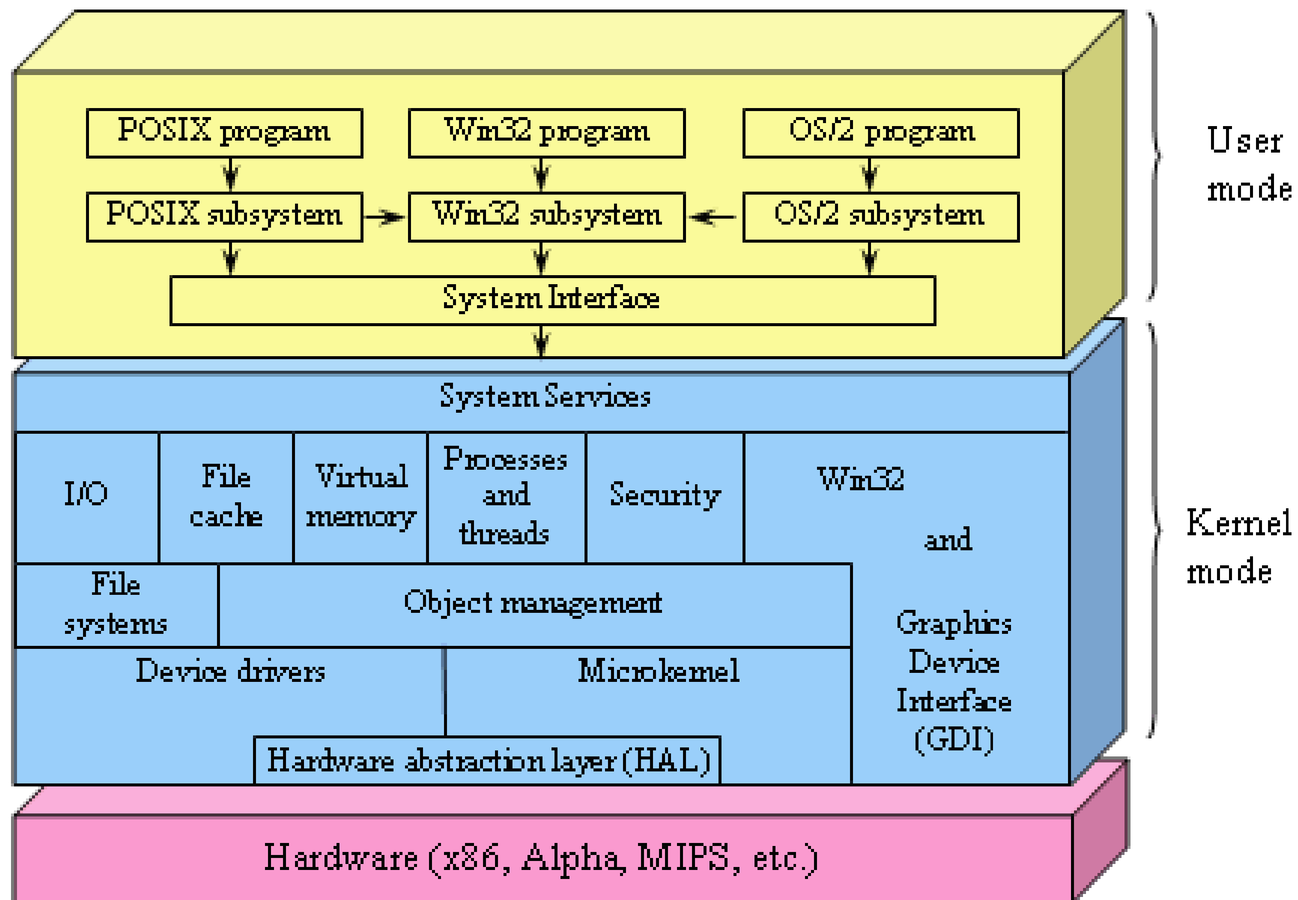
Microkernel System Structure

- Carnegie Mellon University, mid-1980s, OS called **Mach**
  - Modularize the kernel
- **Moves as much from the kernel into "*user*" space**
  - Take out of the kernel the **nonessential parts** and implement them as system and user-level programs
  - Little consensus on what remains in the kernel and what in the user space
- The main function of the **microkernel** is to provide communication facility between the client program and the various services that are running in the user space
  - Communication takes place between user modules using **message passing**
  - **A client and a server communicate with MP through the microkernel**
- 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
  - Windows NT was initially layered microkernel but then was changed in Windows 4.0
- Tru64 UNIX

Microkernel System Structure



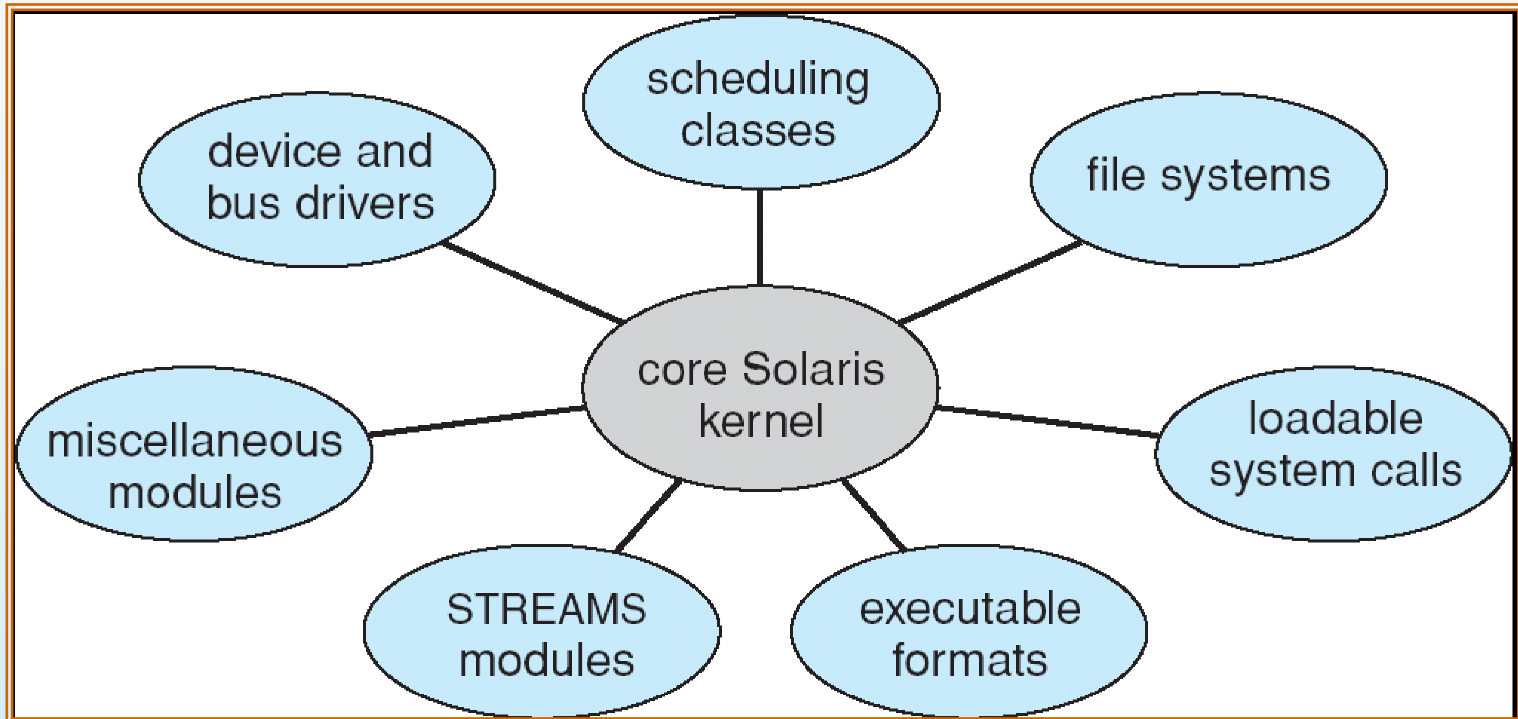
Windows NT Structure



Modular Kernels: 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 in a **more flexible** schema

Solaris Modular Approach



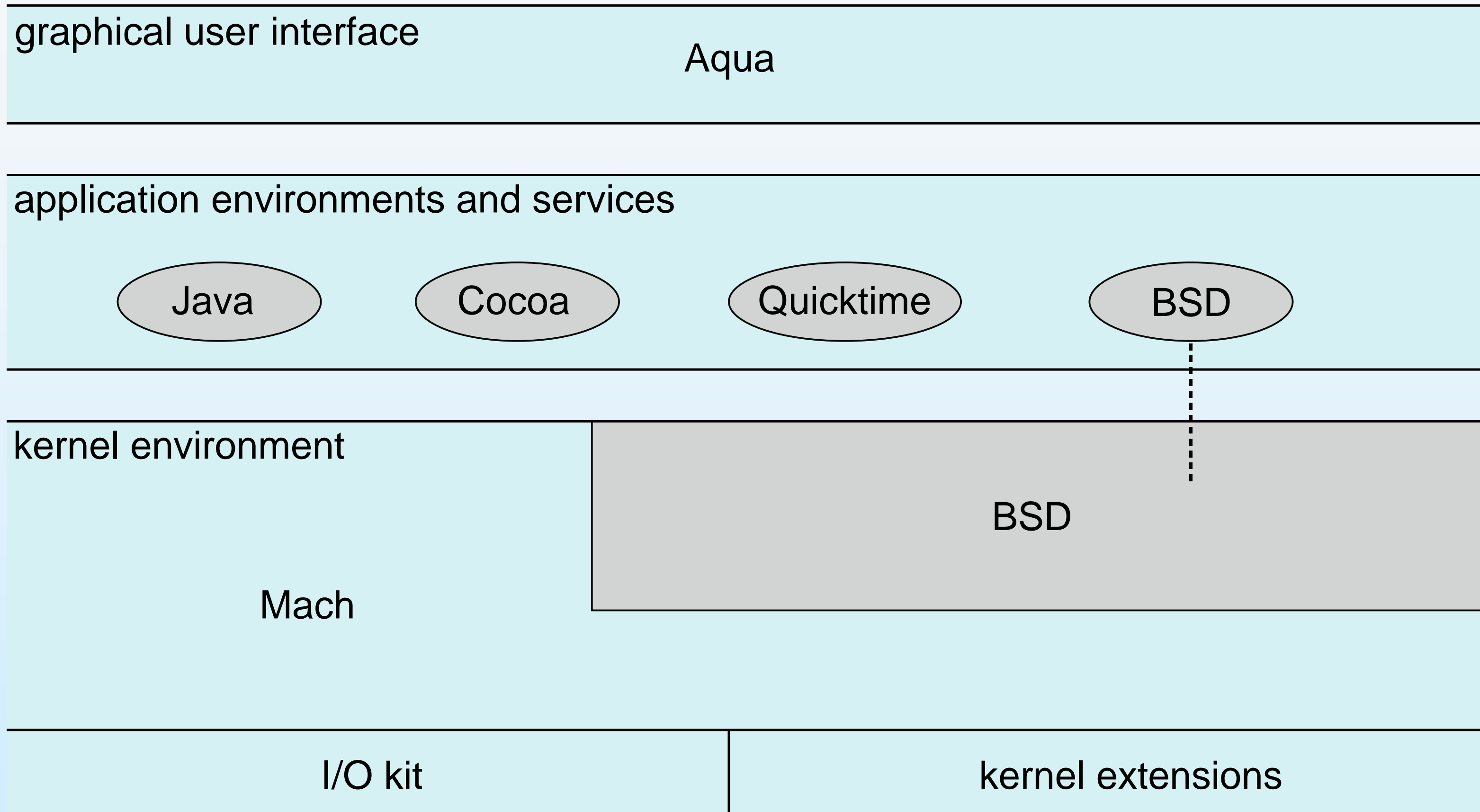
Modular Kernels

- Advantages
  - Certain features can be **implemented dynamically**,
    - Device and bus drivers can be added to the kernel.
    - **Support for different files systems** can be added as loadable modules.
    - Primary module has **only core functions** and knowledge of how to load and communicate with other modules
    - Modules do not need to invoke message passing in order to communicate.

Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure



iOS

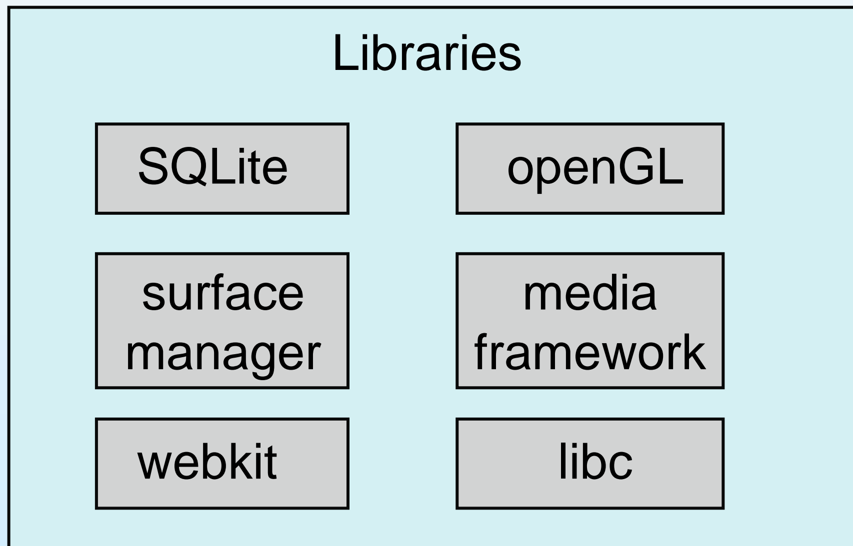
- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel

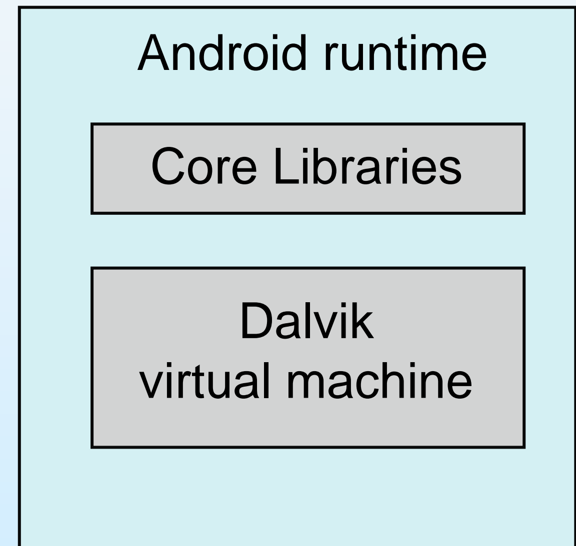
Cocoa Touch
Media Services
Core Services
Core OS

Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture





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
- **Operating Systems Debugging**
- Virtual Machines
- System Boot

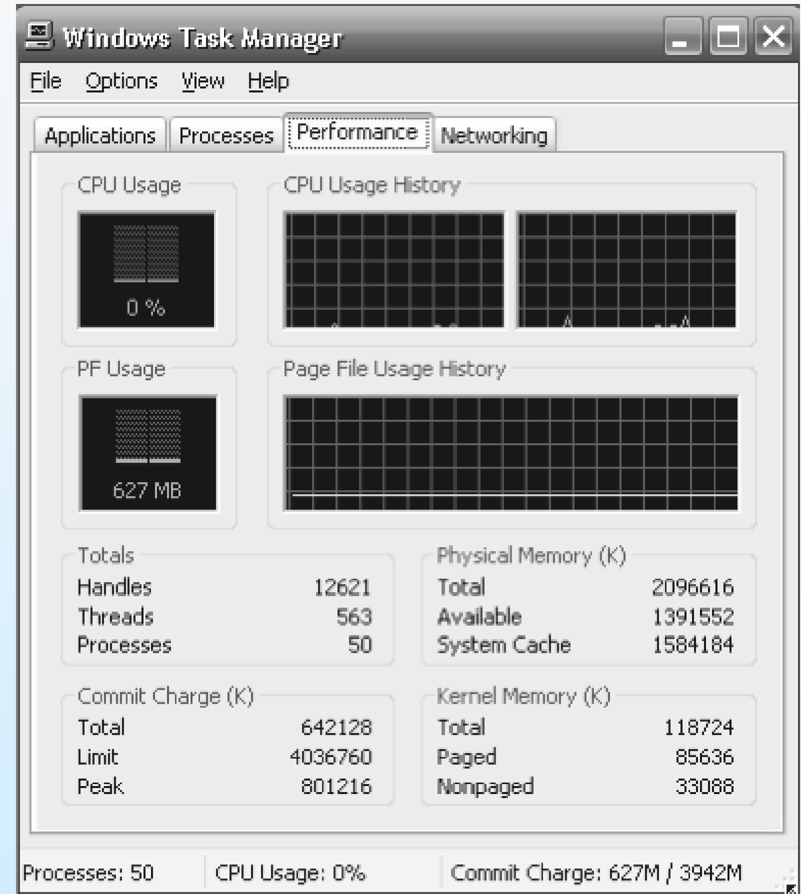
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS 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
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

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."

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



Dtrace (Cont.)

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
  self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
  @time[execname] = sum(timestamp - self->ts);
  self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
   gnome-settings-d        142354
   gnome-vfs-daemon        158243
   dsdm                    189804
   wnck-applet             200030
   gnome-panel             277864
   clock-applet            374916
   mapping-daemon          385475
   xscreensaver            514177
   metacity                539281
   Xorg                   2579646
   gnome-terminal         5007269
   mixer_applet2          7388447
   java                  10769137
```

Figure 2.21 Output of the D code.

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
- Operating Systems Debugging
- **Virtual Machines**
- System Boot

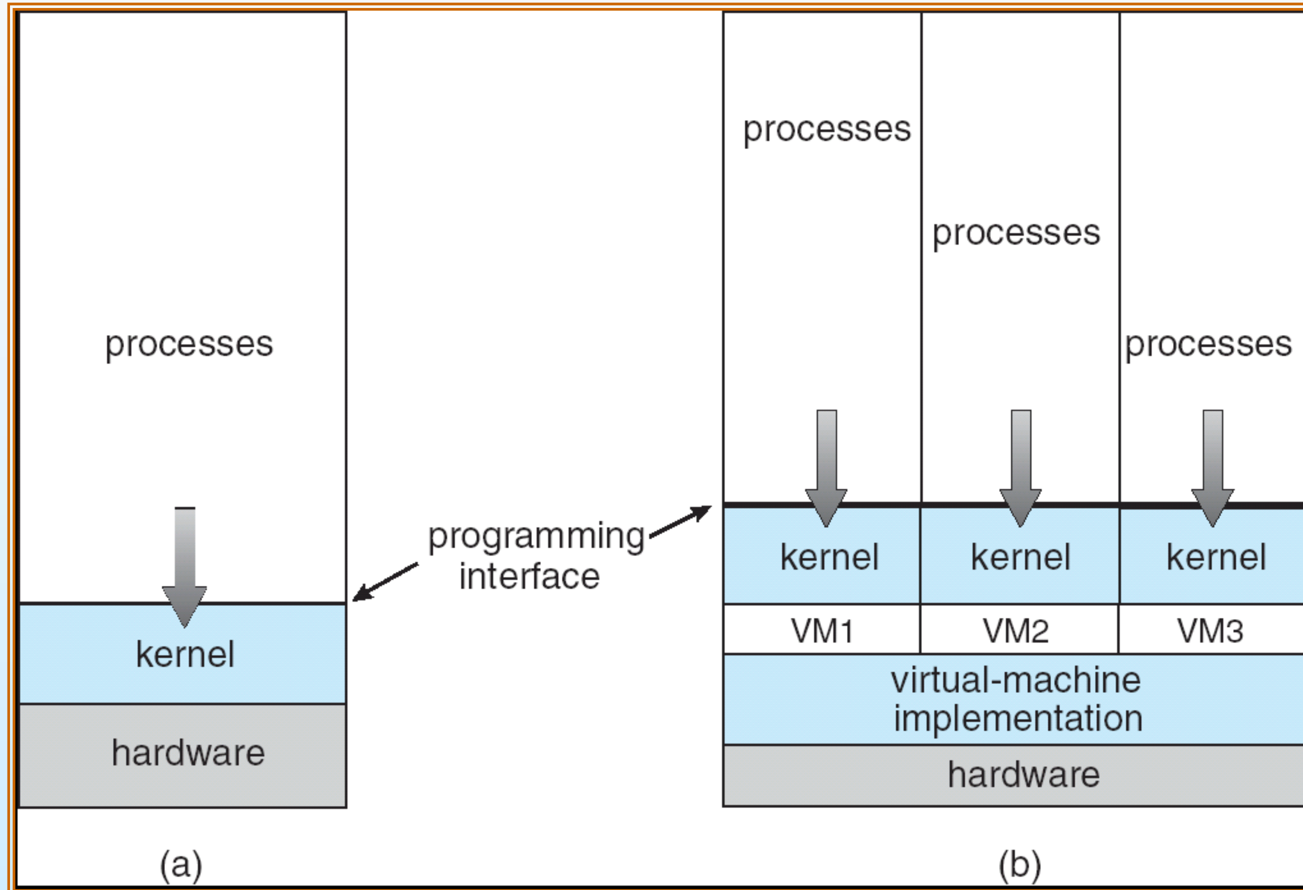
Virtual Machines 1

- 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
- A **virtual machine** (**VM**) is a **software implementation of a machine** (i.e. a computer) that executes programs like a physical machine.

Virtual Machines 2

- 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
- An essential characteristic of a virtual machine is that the software running inside **is limited to the resources and abstractions provided by the virtual machine** — it cannot break out of its virtual world.

Virtual Machines 3



(a) Nonvirtual machine (b) virtual machine

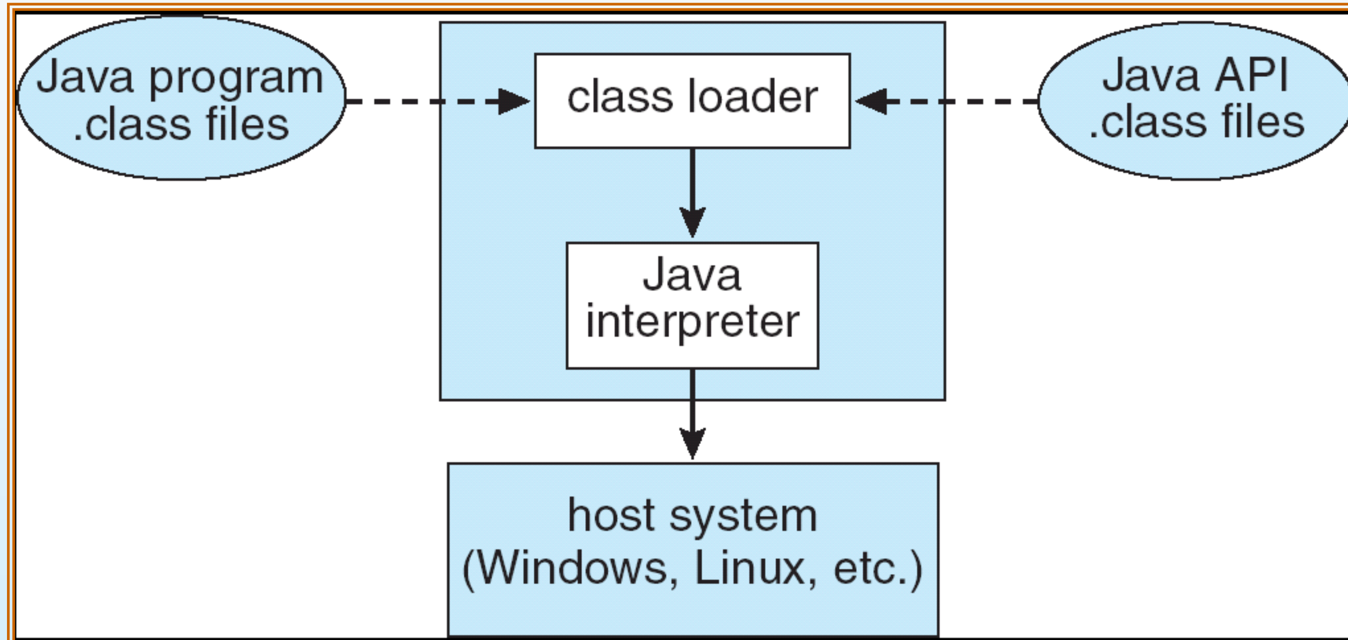
Virtual Machines 4

- 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**

Virtual Machines 5

- **A System virtual machine** provides a complete system platform which supports the execution of a complete operating system (OS).
  - VMware
  - Virtual PC (Microsoft)
- A **process virtual machine** is designed to run a single program, which means that it supports a single process.
  - This type of VM has become popular with the Java programming language, which is implemented using the Java virtual machine.
  - Another example is the **.NET Framework**, which runs on a VM called the Common Language Runtime.

The Java Virtual Machine



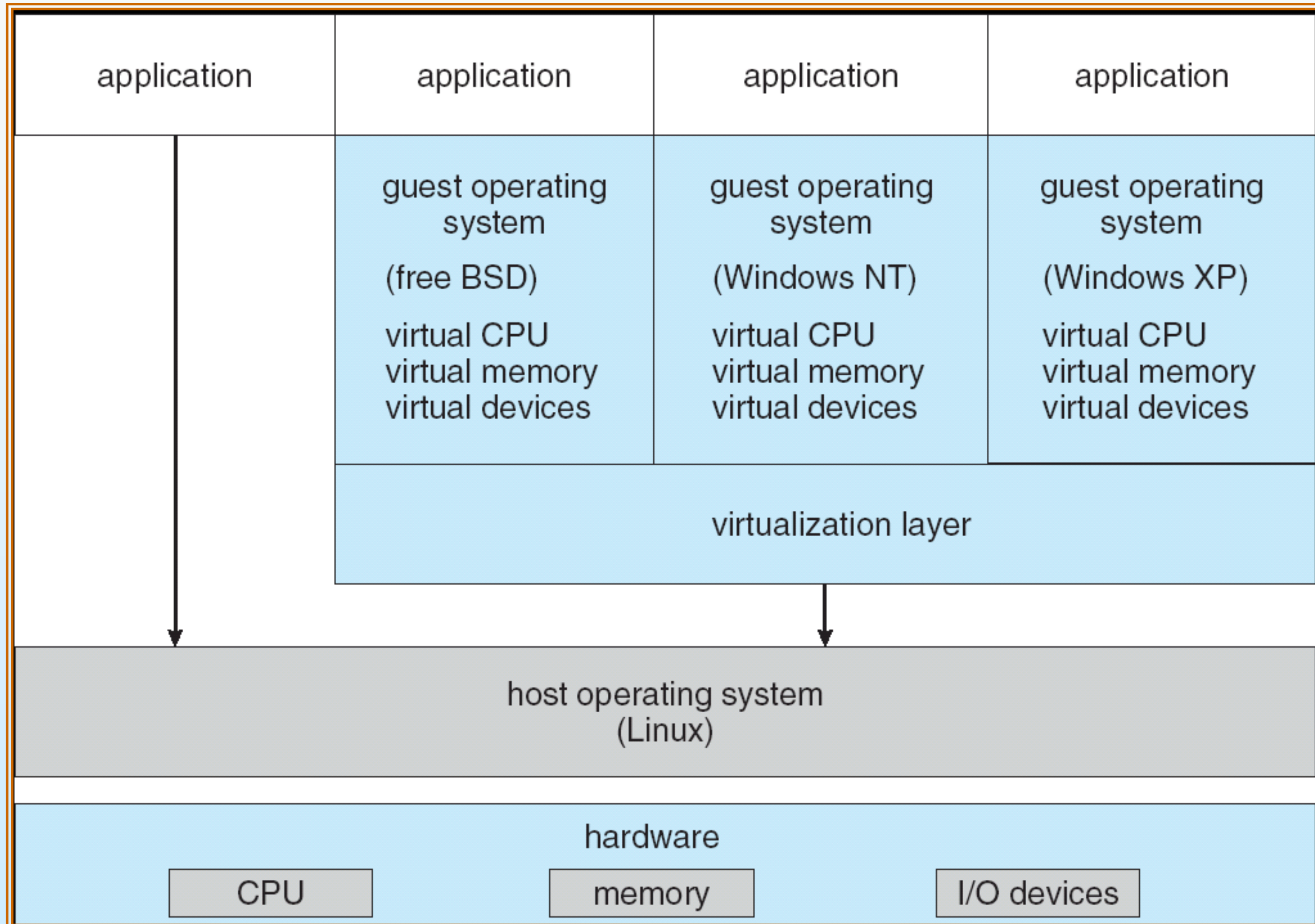
Virtualization 1

- The desire to run multiple operating systems was the original motivation for virtual machines, as it allowed time-sharing a single computer between several single-tasking OSes.
- Multiple VMs each running their own operating system (called **guest operating system**) are frequently used in **server consolidation**, where different services that used to run on individual machines in order to avoid interference are instead run in separate VMs on the same physical machine (called **host operating system**).
- This technique requires a process to share the CPU resources between guest operating systems and memory virtualization to share the memory on the host.

Virtualization 2

- The main advantages of system VMs are:
  - multiple OS environments can **co-exist on the same computer**, in strong isolation from each other
  - the virtual machine can provide an **instruction set architecture** (ISA) that is somewhat **different** from that of the real machine
  - application provisioning, maintenance, **high availability and disaster recovery.**
- The main disadvantage of system VMs is:
  - a virtual machine **is less efficient than a real machine** because it accesses the hardware indirectly
- The guest OSes do not have to be all the same, making it possible to run different OSes on the same computer (e.g., Microsoft Windows and Linux, or older versions of an OS in order to support software that has not yet been ported to the latest version).

VMware Architecture



System Virtual Machine Software

- ATL (A MTL Virtual Machine)
- Bochs, portable open source x86 and AMD64 PCs emulator
- CHARON-AXP, provides virtualization of AlphaServer to migrate OpenVMS or Tru64 applications to x86 hardware
- CHARON-VAX, provides virtualization of PDP-11 or VAX hardware to migrate OpenVMS or Tru64 applications to x86 or HP integrity hardware
- CoLinux Open Source Linux inside Windows
- Denali, uses paravirtualization of x86 for running para-virtualized PC operating systems.
- eVM Virtualization Platform for Windows by TenAsys
- Hercules emulator, free System/370, ESA/390, z/Mainframe
- Microsoft Virtual PC and Microsoft Virtual Server
- OKL4 from Open Kernel Labs
- Oracle VM
- SLKVM - scripts to handle kvm and vz virtual machines in a cluster environment
- Sun xVM
- VM from IBM
- VMware (ESX Server, Fusion, Virtual Server, Workstation, Player and ACE)
- vSMP Foundation (From ScaleMP)
- Xen (Opensource)
- IBM POWER SYSTEMS

Process Virtual Machine Software

- Common Language Infrastructure - C#, Visual Basic .NET, J#, C++/CLI (formerly Managed C++)
- Dalvik virtual machine - part of the Android mobile phone platform
- Java Virtual Machine - Java, Nice, NetREXX
- Juke Virtual Machine - A public domain ECMA-335 compatible virtual machine hosted at Google code.
- Low Level Virtual Machine (LLVM) - currently C, C++, Stacker
- Macromedia Flash Player - SWF
- Perl virtual machine - Perl
- CPython - Python
- Rubinius - Ruby
- SECD machine - ISWIM, Lispkit Lisp
- Sed the stream-editor can also be seen as a VM with 2 storage spaces.
- Smalltalk virtual machine - Smalltalk
- SQLite virtual machine - SQLite opcodes
- Tamarin (JavaScript engine) - ActionScript VM in Flash 9
- TrueType virtual machine - TrueType
- Valgrind - checking of memory accesses and leaks in x86/x86-64 code under Linux
- Virtual Processor (VP) from Tao Group (UK).
- Waba - Virtual machine for small devices, similar to Java
- Warren Abstract Machine - Prolog, CSC GraphTalk

System Boot 1

- *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
- 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 bootstrap loader fetches a more complex boot program from disk which in turn loads the kernel.
  - When power initialized on system, execution starts at **a fixed memory location**
    - Firmware used to hold initial boot code

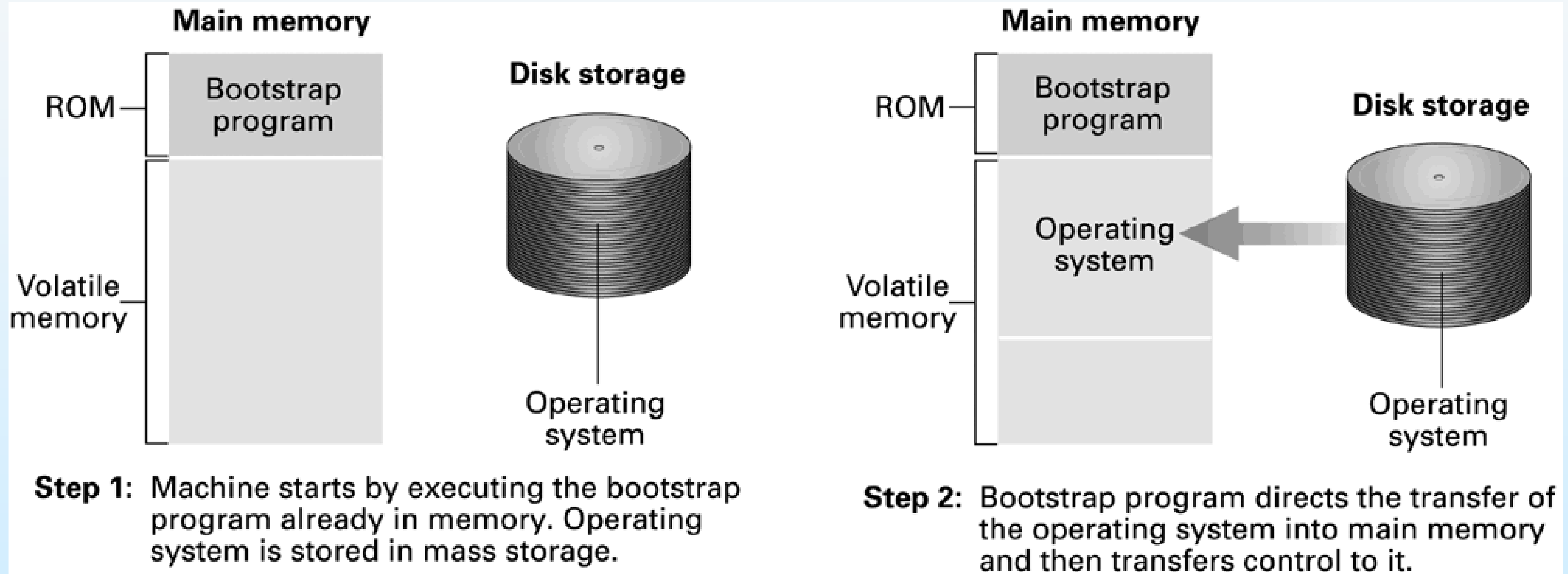
System Boot 2

- When CPU receives a reset event – power up or reboot – the **instruction register is loaded** with a predefined memory location and execution starts there.
  - The location is where the **initial bootstrap program resides**
  - Written in ROM
  - RAM status is not known at the boot moment
  - ROM needs **no initialization** and cannot be affected by viruses
- Bootstrap program can:
  - Run diagnostics to determine the state of the machine
  - Initializes CPU registers, device controllers and the contents of the main memory
  - Starts the Operating System

System Boot 3

- Some systems such as cellular phones, PDAs and game consoles **store the entire OS in the ROM.**
  - This is convenient for small OSs and simple supporting hardware
  - The problem is that changing the bootstrap code requires changing the ROM hardware chips.
  - For this reason **EPROM** (*Erasable Programmable Read-Only Memory*) is used.
  - EPROM is read-only but becomes writable if given a certain command.
- All forms of ROM are known also as *firmware*
  - A problem with *firmware* is that **executing programs is slower** than RAM.
  - Some systems **store the OS in ROM but load it in RAM** for faster execution.
  - ROM is expensive thus small amounts are available.

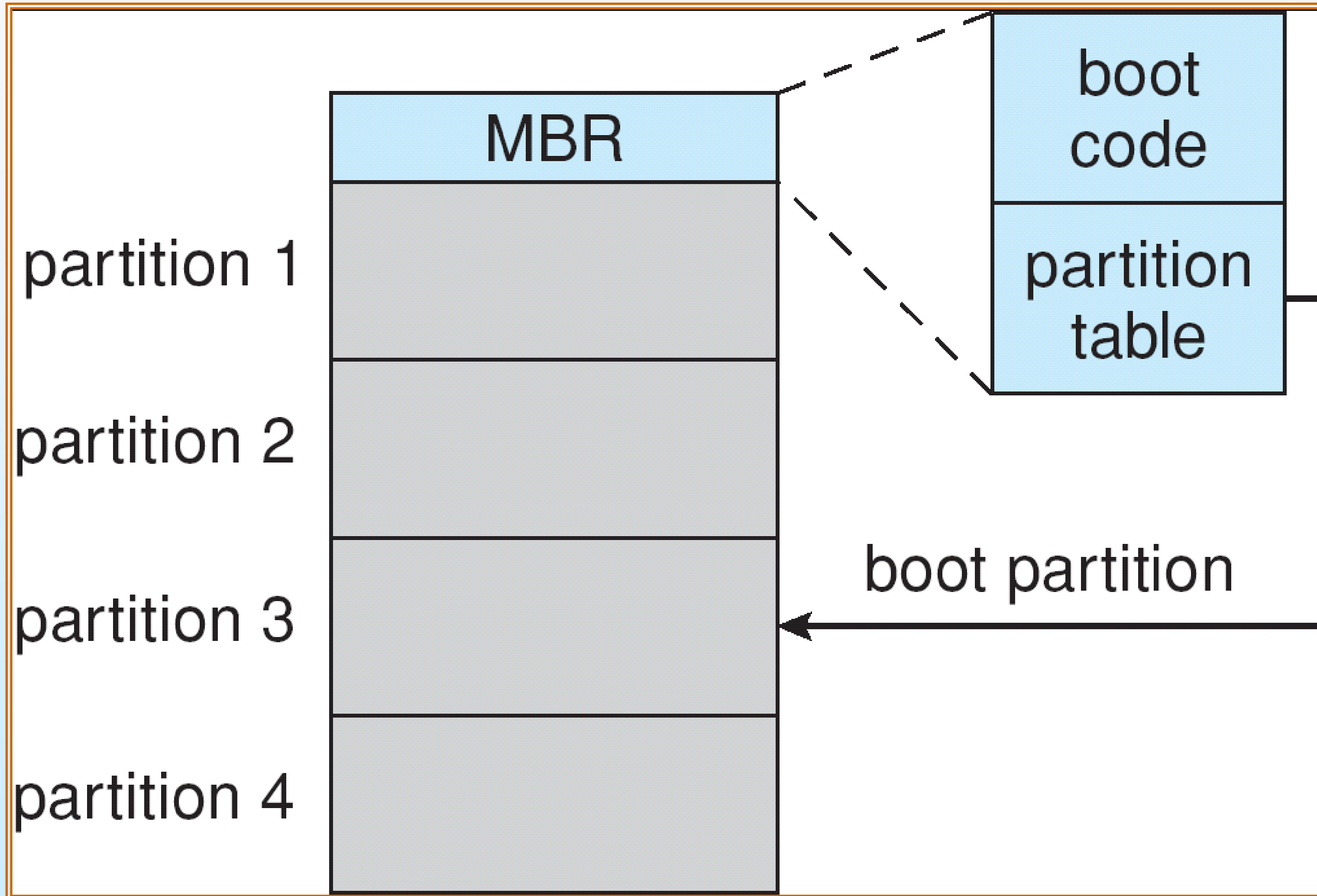
Figure 3.5 The booting process



System Boot 4

- Large OSs like Linux or Windows
  - Store the bootstrap loader in *firmware,* but store the OS in disk.
- The bootstrap program runs diagnostics, reads a single block at a fixed location form disk into memory and execute the code from that **boot block**.
  - The **program** stored in the **boot black** may be enough sophisticated to load the entire operating system in to RAM and begin its execution.
  - This **program** is usually very simple code since it fits in a single disk block.
    - **It only knows the address** on disk and length of the remainder of the bootstrap program
    - All of the disk-bound bootstrap and the OS itself can be easily changed by writing
- A disk which has a boot partition is called a **boot disk** or **system disk**
- Only the kernel is loaded into memory the system is said to be **running**

Booting from a Disk in Windows 2000



System Boot 5

- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options

- Once the OS initializes it performs
  - Loading the device drivers in order to control peripheral devices, such as a printer, scanner, optical drive, mouse and keyboard.
  - This is the final stage in the boot process, after which the user can access the system’s applications to perform tasks.

Readings

- Silberschatz - Chapter 2

End of Chapter 2