Open Source OS

Module 0: Overview

Module 0: Overview

- What is Operating System
- Computer-System Organization and Architecture
- OS Operations: Process/Memory/File/Storage Management
- Computing Environments
- Free and Open-Source Operating Systems
- Operating System Services
  - System Calls
- Design and Implementation
  - Operating System Structure

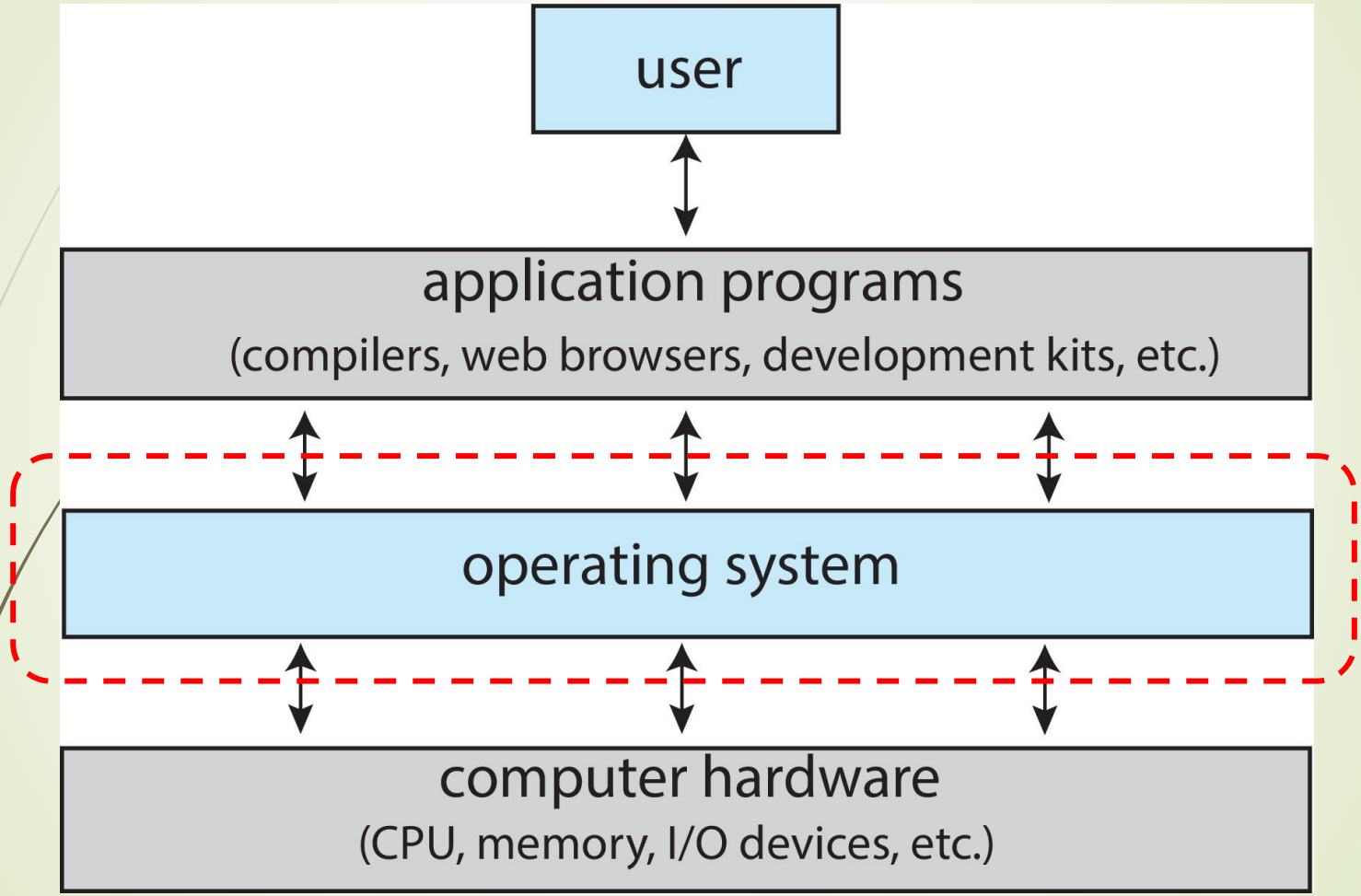
What is an Operating System?

- A program that acts as an intermediary between a user and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

Computer System Structure

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

Abstract View of Components of Computer



[Source: Operating Systems Concepts, 10th ed.]

What Operating Systems Do – Two Views

- Users want convenience, **ease of use** and **good performance**
  - OS is an environment for completing tasks
- But shared computer such as **mainframe** or **server** must keep all users happy
  - OS is a **resource allocator** and **control program** ensuring efficient use of HW and correct execution of user programs

What Operating Systems Do

- Users of **servers** usually use shared resources
- Mobile devices are limited in resource, optimized for battery life
  - User interfaces of smartphones and tablets such as touch screens, voice recognition
- Embedded systems in devices and automobiles have little or no user interface
  - Run without user intervention

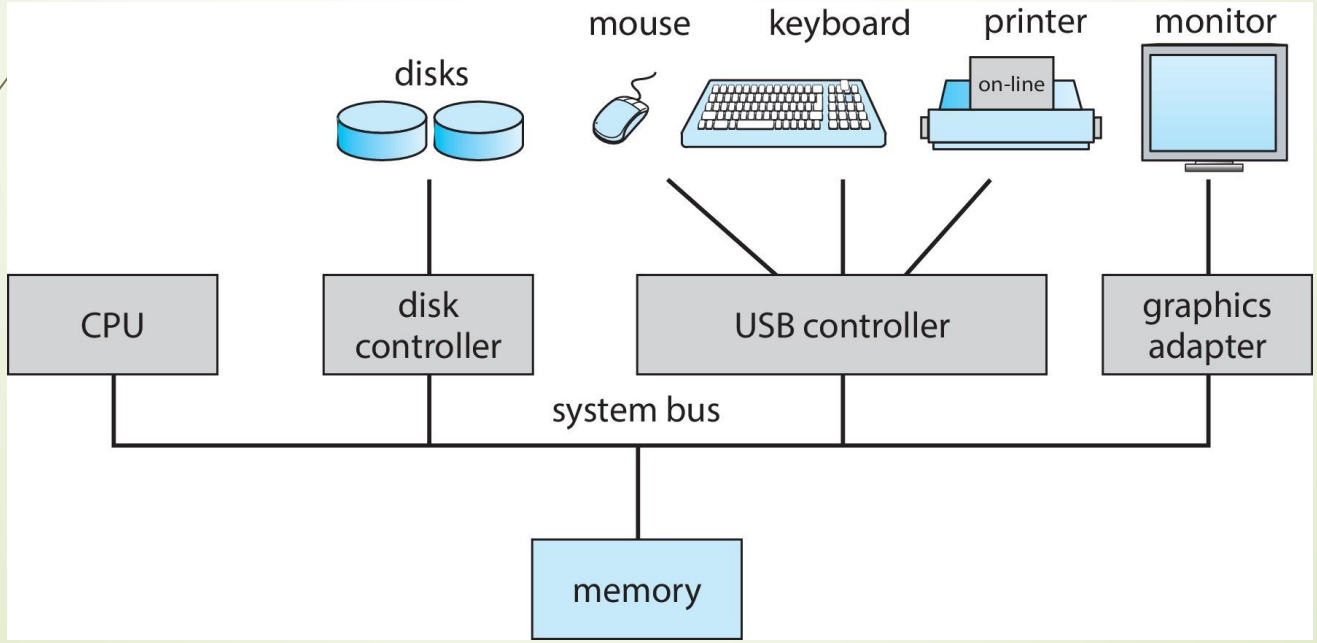
Operating System Definition – no universal

- A common definition: "The one program running at all times on the computer" is the **kernel**, part of the OS
  - ***System programs*** – also ships with the OS, but not part of the kernel
- Others:
  - ***Application programs*** - all programs not associated with the OS
  - ***Middleware*** – a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics

Overview of Computer System Structure

Computer System Organization

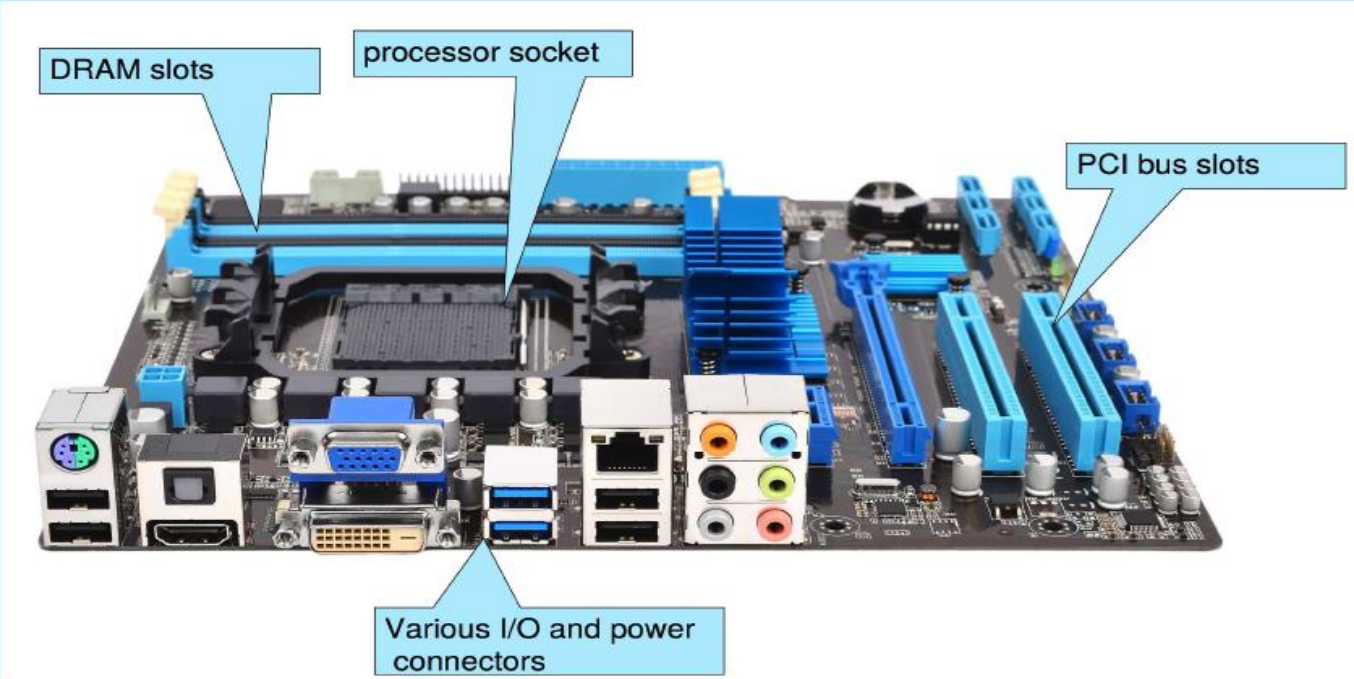
- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices



[Source: Operating Systems Concepts, 10th ed.]

PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

[Source: Operating Systems Concepts, 10th ed.]

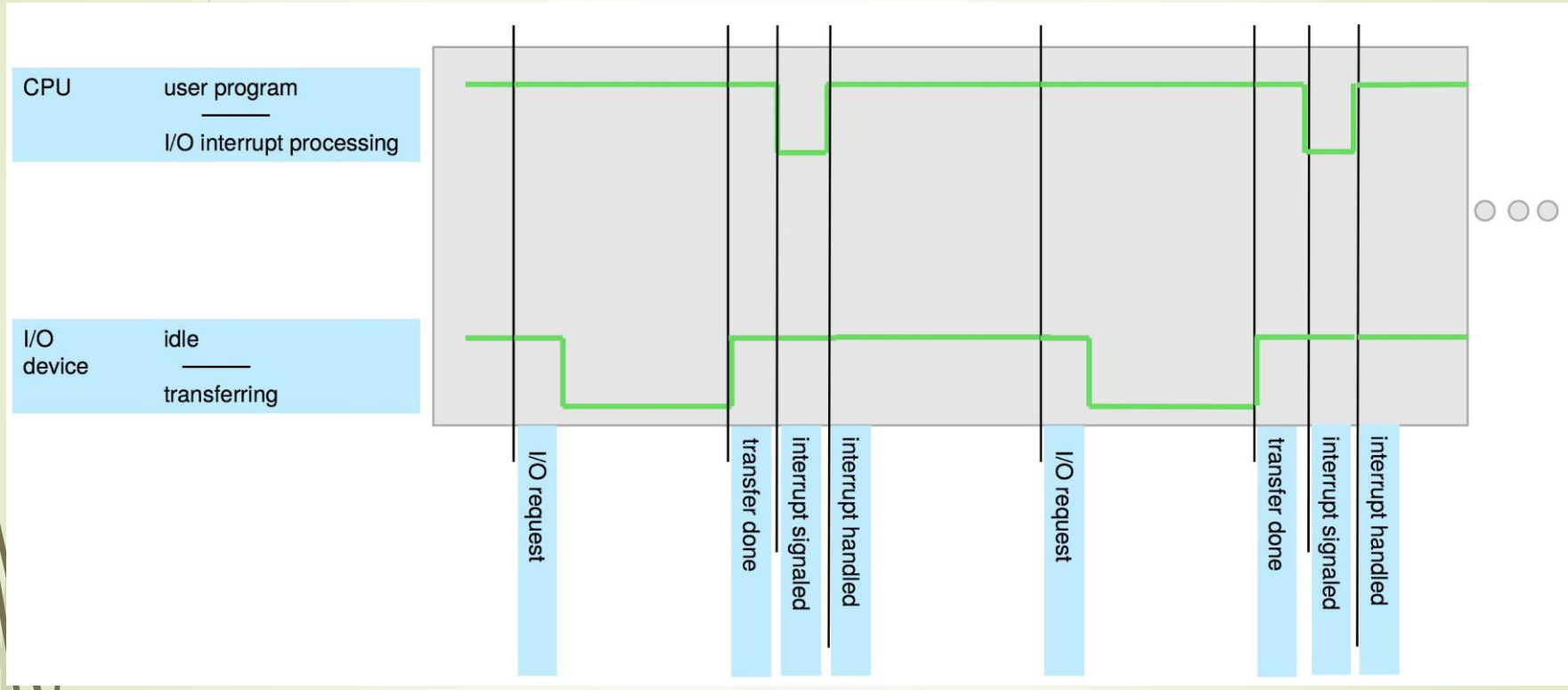
Computer-System Operation

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

Common Functions of Interrupts

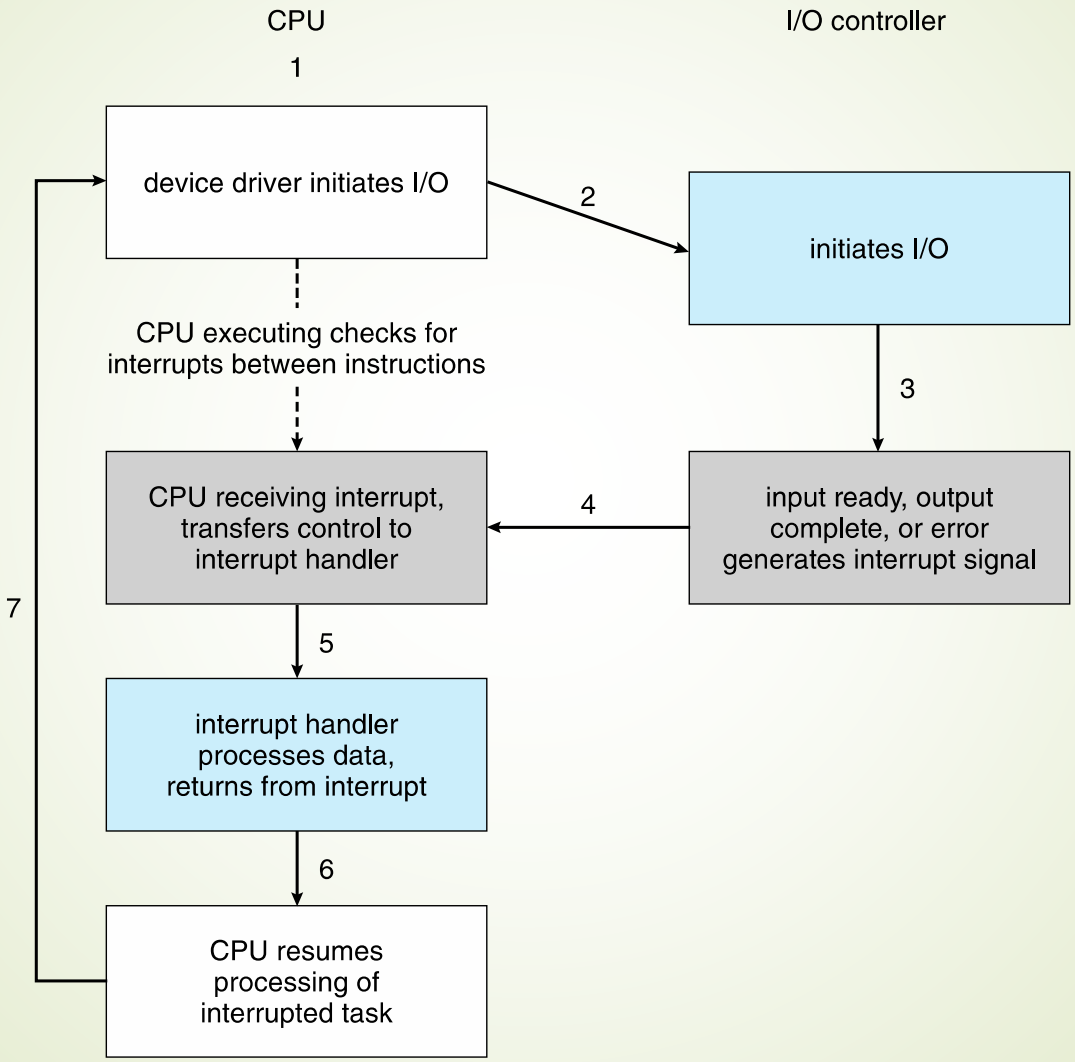
- Modern OS is **interrupt-driven**
- When receiving an interrupt, CPU control transfers to the interrupt service routine, through the **interrupt vector**
  - It contains the addresses of all service routines
- CPU state must be saved
  - CPU registers
  - Program counter: the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request

Interrupt Timeline



[Source: Operating Systems Concepts, 10th ed.]

Interrupt-driven I/O Cycle



[Source: Operating Systems Concepts, 10th ed.]

I/O Structure

- Two methods for handling I/O
  - Synchronous: After I/O starts, control returns to user program only upon I/O completion
  - Asynchronous: After I/O starts, control returns to user program without waiting for I/O completion

Storage Structure

Storage Structure

- Main memory – the only large storage media that the CPU can access directly
  - **Random access**
  - Typically **volatile**
  - Typically **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity

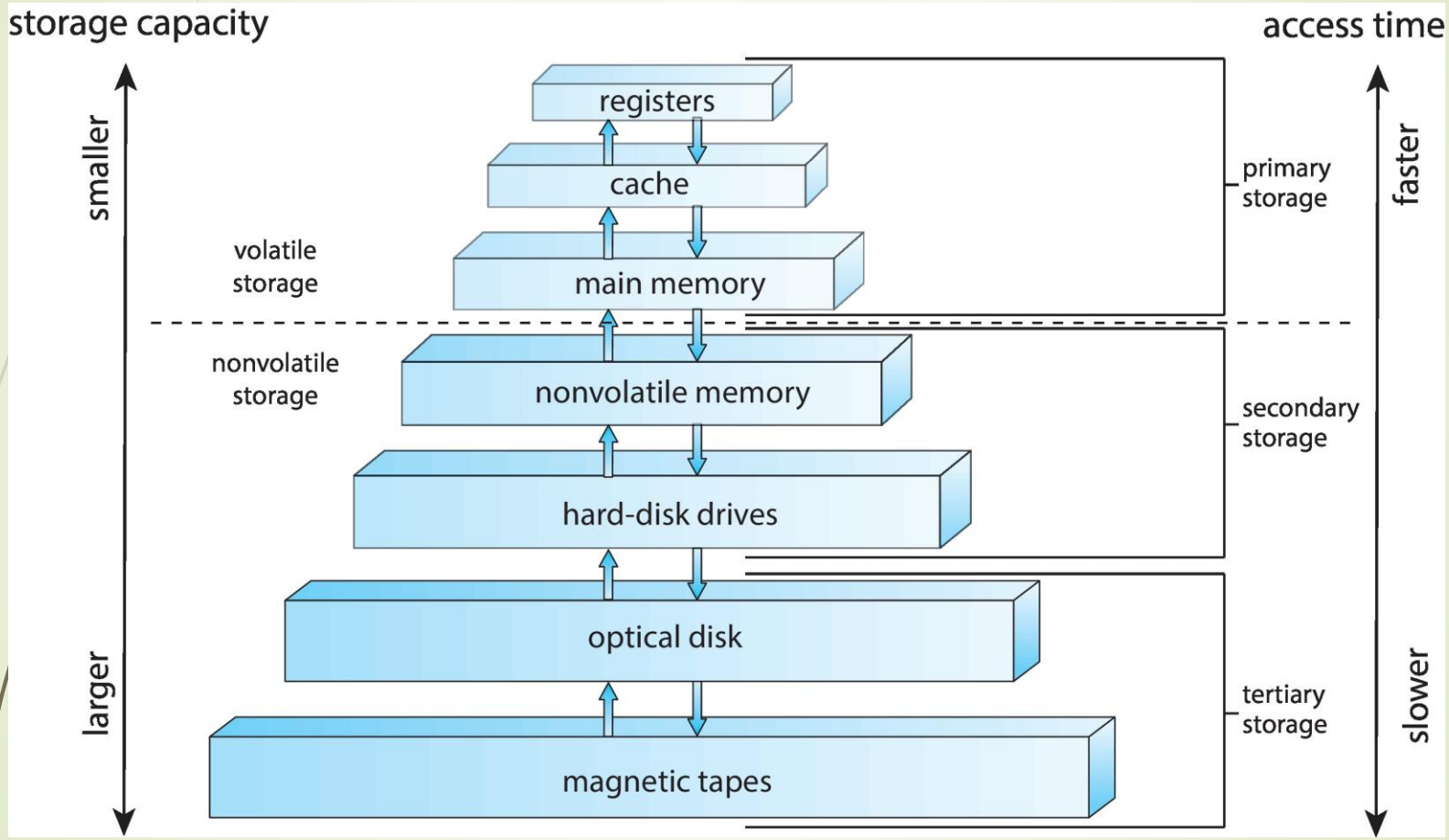
Storage Structure (Cont.)

- **Hard Disk Drives** (**HDD**) – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory** (**NVM**) devices – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops

Storage Hierarchy

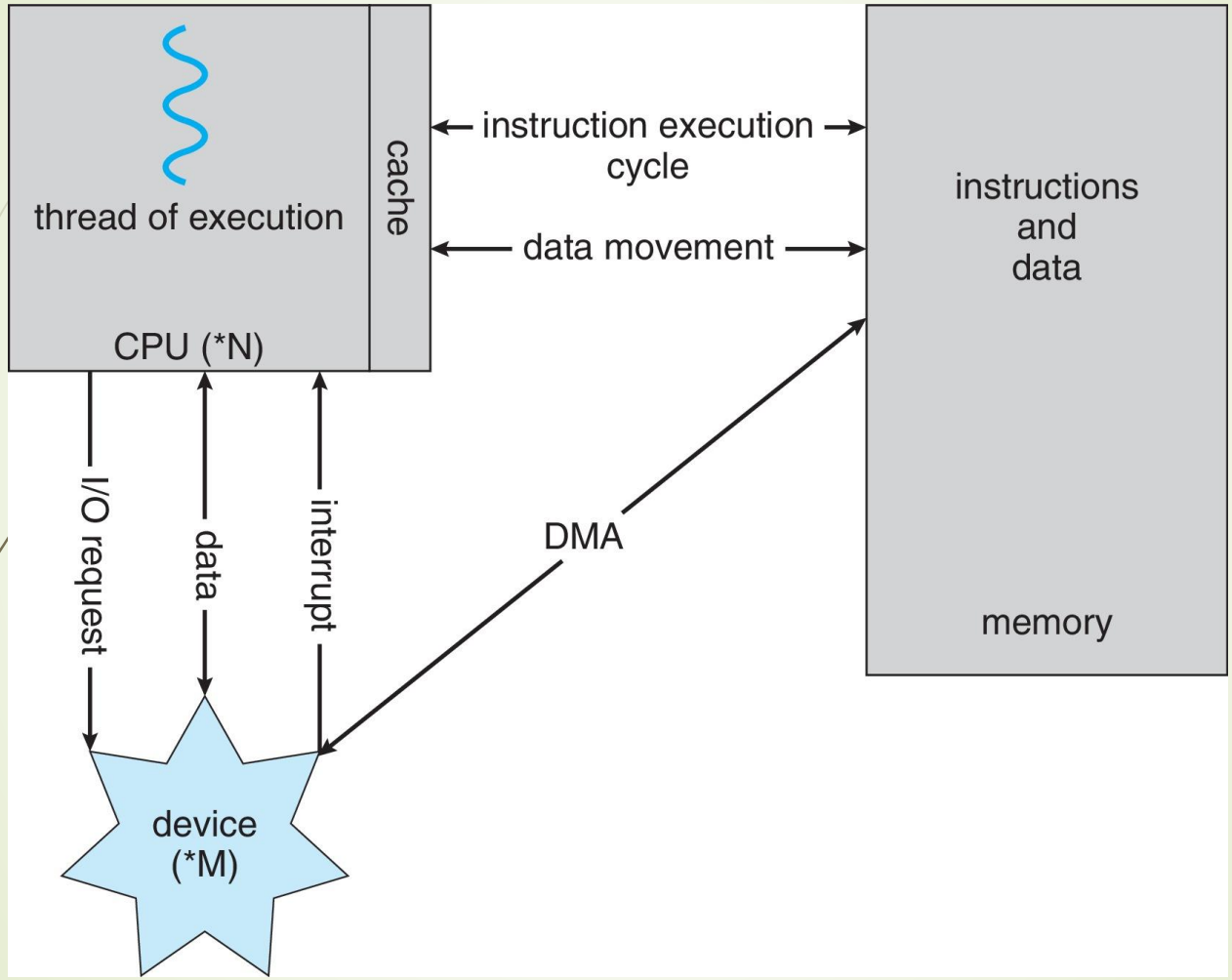
- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system
  - Main memory can be viewed as a cache for secondary storage

Storage-Device Hierarchy



[Source: Operating Systems Concepts, 10th ed.]

How a Modern Computer Works



A von Neumann architecture

[Source: Operating Systems Concepts, 10th ed.]

Direct Memory Access Structure

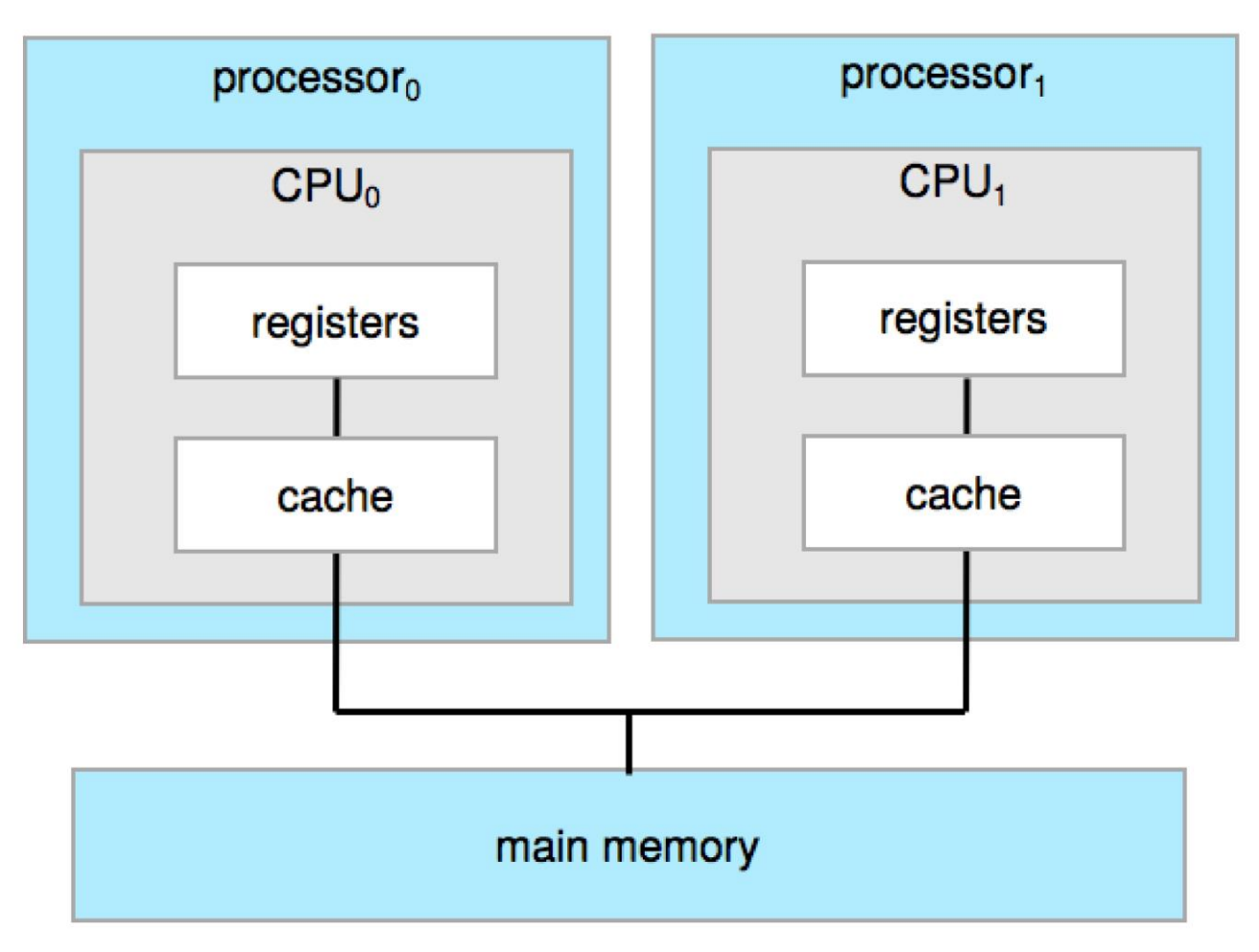
- Used for high-speed I/O devices
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than one interrupt per byte

Computer System Architecture

Computer-System Architecture

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance
  - Two types:
    1. **Asymmetric Multiprocessing** – each processor is assigned a specie task
    2. **Symmetric Multiprocessing** – each processor performs all tasks

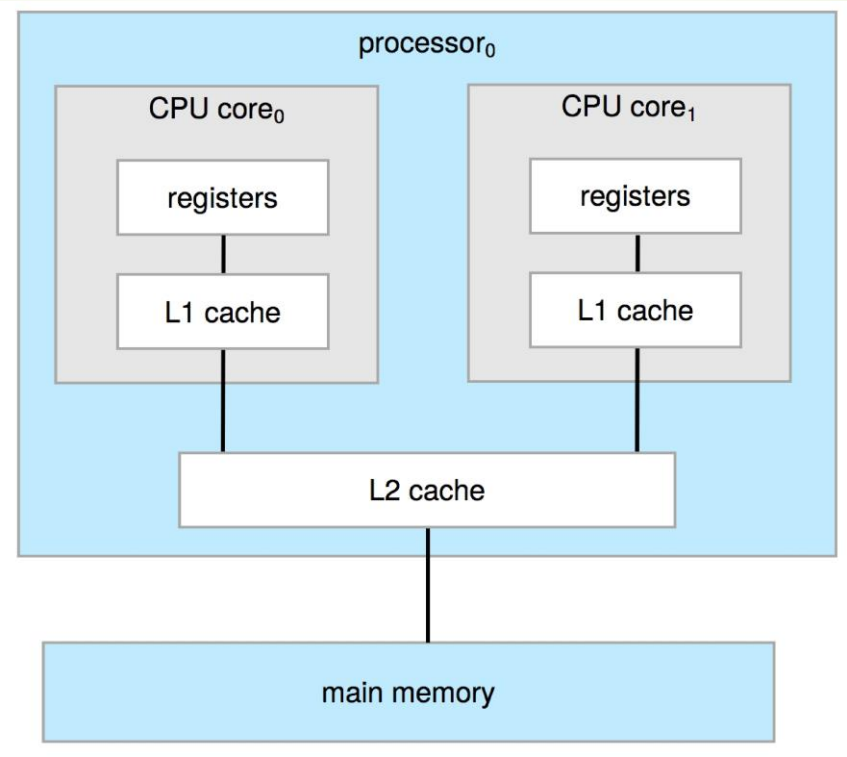
Symmetric Multiprocessing Architecture



[Source: Operating Systems Concepts, 10th ed.]

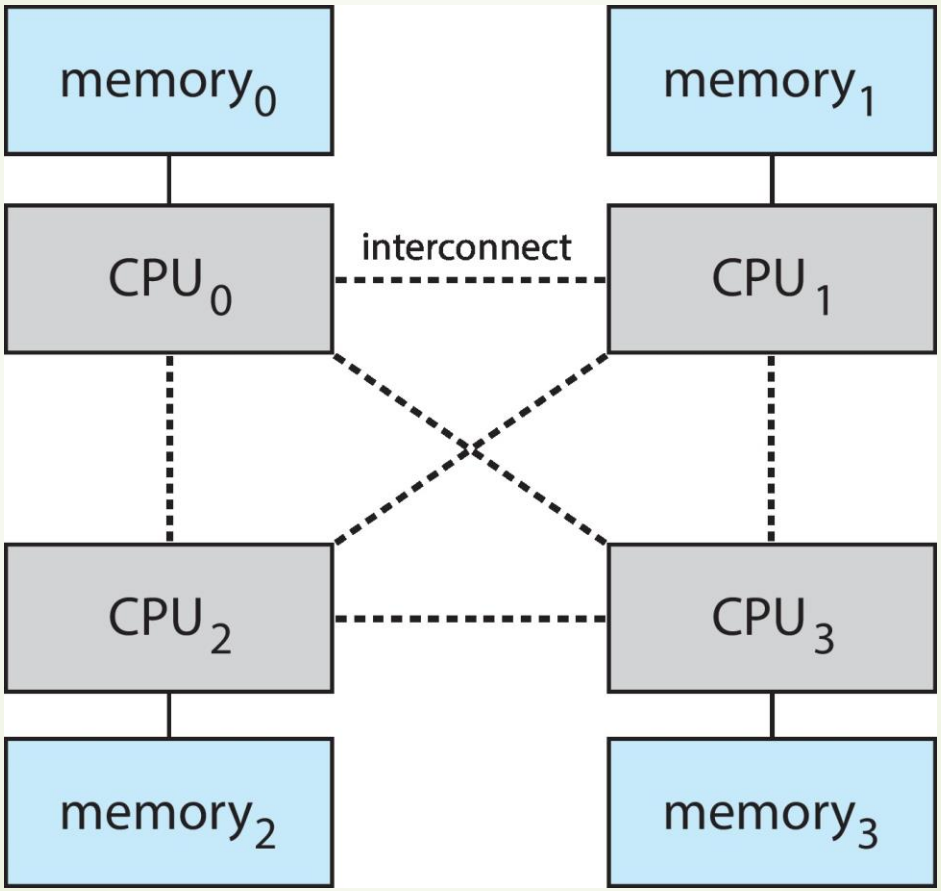
Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems



[Source: Operating Systems Concepts, 10th ed.]

Non-Uniform Memory Access System



[Source: Operating Systems Concepts, 10th ed.]

Operating System Operations

Operating-System Operations

- Bootstrap program – simple code to initialize the system and load OS kernel
- **System daemons** - services provided outside of the kernel
- **Interrupt-driven kernel**
  - Hardware interrupt from devices
  - Software interrupt (**exception** or **trap**):
    - Software error (e.g., division by zero)
    - Request for OS service – **system call**
    - Other process problems include infinite loop, processes modifying each other or the OS

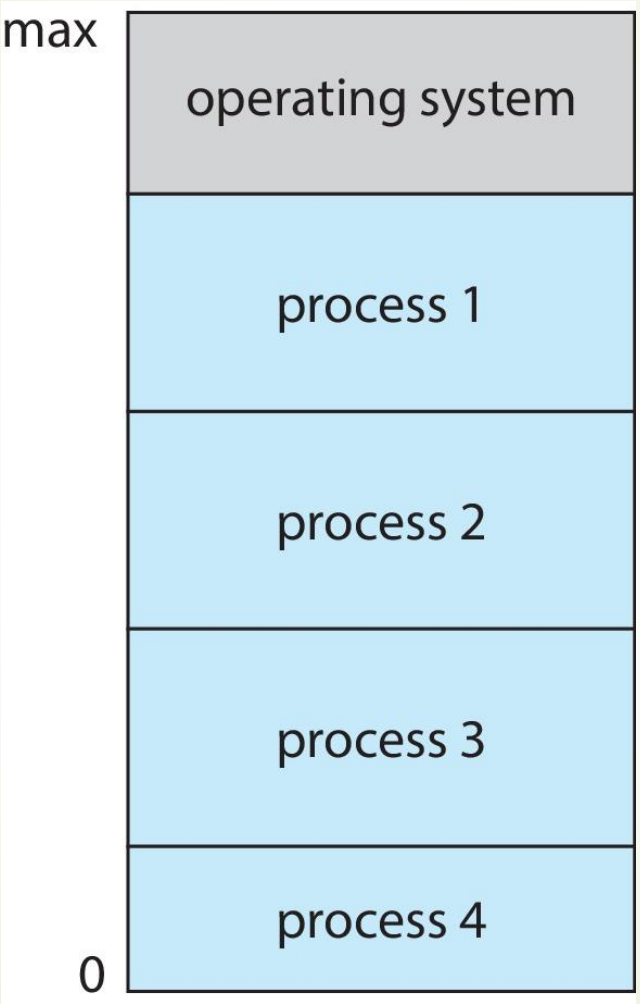
Multiprogramming (Batch system)

- Single user cannot always keep CPU and I/O devices busy
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - Load a subset of total jobs in memory
  - Select one job to run via **job scheduling**
  - When job has to wait (for I/O for example), OS switches to another job

Multitasking (Timesharing)

- A logical extension of Batch systems – CPU switches jobs so fast that users can interact with each job, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨ **process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Memory Layout for Multiprogrammed System

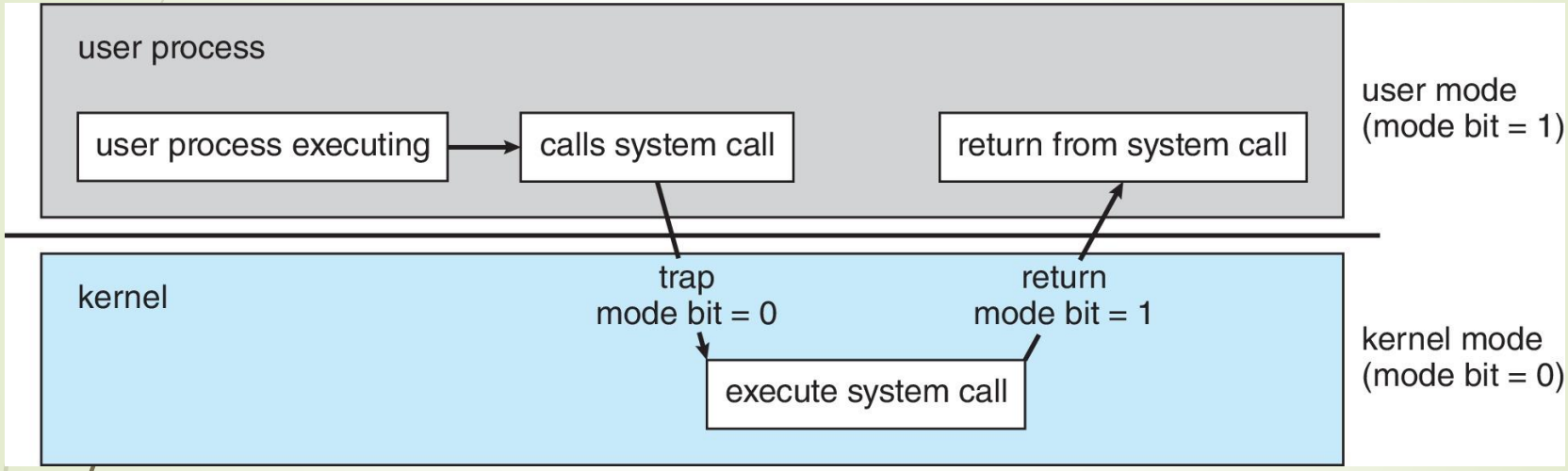


[Source: Operating Systems Concepts, 10th ed.]

Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - When user code is running ⇨ mode bit: "user"
  - When kernel code is executing ⇨ mode bit: "kernel"
- Users can not explicitly set the mode bit to "kernel"
  - Via **privileged** instructions, only executable in kernel mode
  - System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode



[Source: Operating Systems Concepts, 10th ed.]

Timer

- Timer to prevent infinite loop (or process hogging resources)
  - Timer is set to interrupt the computer at fixed time period
    - A counter decremented by the physical clock
    - OS set the counter (privileged instruction)
    - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Process Management

- A process is a program in execution
  - It is a unit of work in the system
  - Program is a ***passive*** *entity;* process is an ***active*** *entity*
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires release of any reusable resources

Process Management

- Single-threaded process has one **program counter** specifying address of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- System has many (system/user) processes running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

Process management activities:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- Von Neumann architecture
  - To execute a program, all (or part) of the instructions must be in memory
  - All (or part) of the data that is needed by the program must be in memory
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

File-system Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
- File system management activities:
  - Creating and deleting files and directories
  - Primitives to manipulate files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Disks usually used to store data that does not fit in main memory or data that must be kept for a "long"period of time
- Proper management is very important
  - Each physical storage medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- Entire speed of computer operation hinges on disk subsystem and its algorithms

Mass-Storage Management

- Storage management activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy
  - **Cache coherency**: consistency between multiple copies of the same data

Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

[Source: Operating Systems Concepts, 10th ed.]

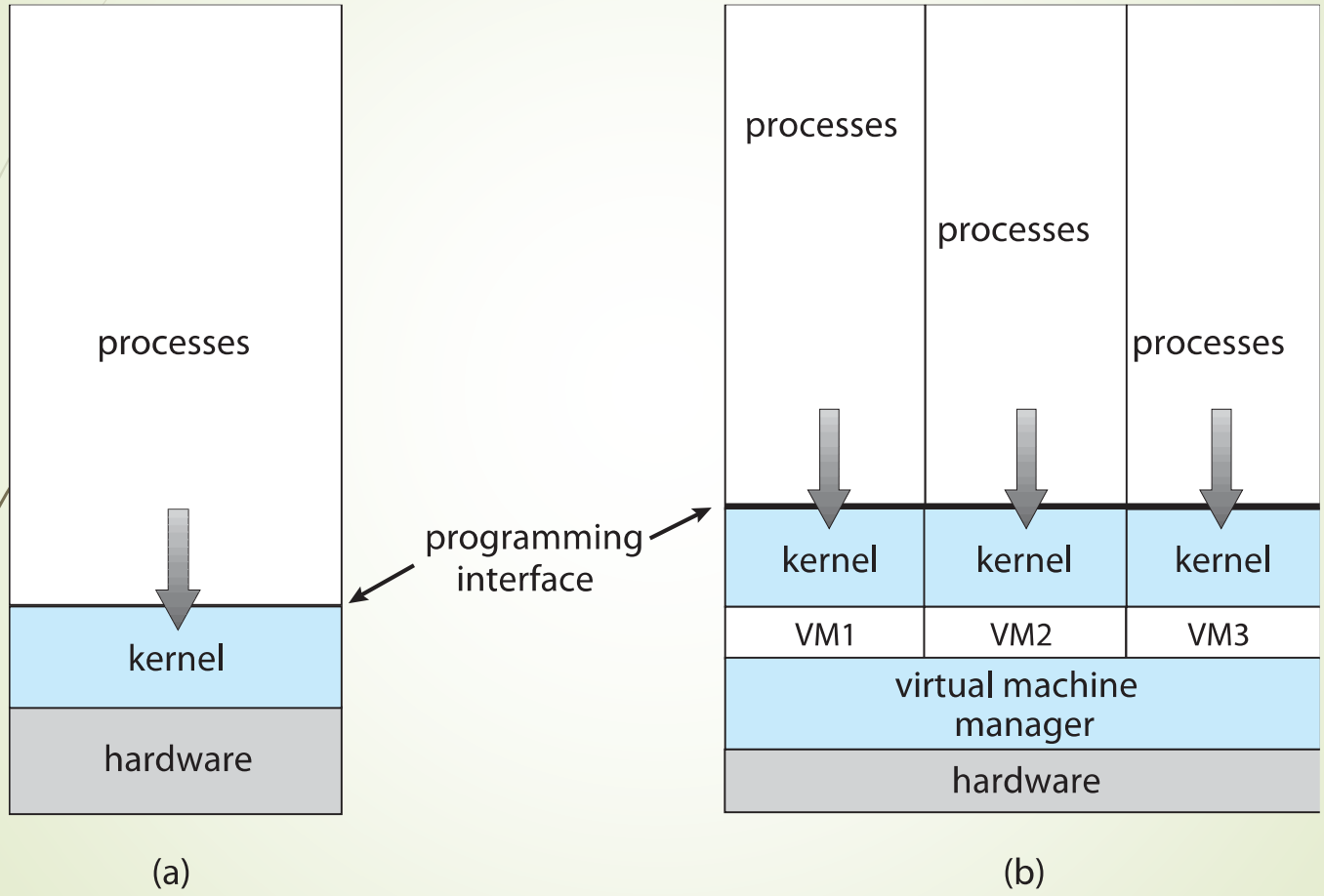
I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including
    - buffering (storing data temporarily while it is being transferred)
    - caching (storing part of data in faster storage for performance)
    - spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface via drivers for specific hardware devices

Virtualization

- Allows OS to run applications within other OSes
  - Vast and growing applications
- **Emulation** used when source CPU type different from target type (i.e. PowerPC to Intel x86)
  - Generally slowest method
  - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - Consider VMware running Linux and MacOS guests, each running applications, all on native Win10 **host** OS
  - **VMM** (virtual machine manager) provides virtualization services

Computing Environments - Virtualization



[Source: Operating Systems Concepts, 10th ed.]

Computing Environments

Computing Environments

- Traditional
- Mobile
- Client-Server
- Pear-to-Pear
- Cloud computing
- Real-time Embedded

Traditional

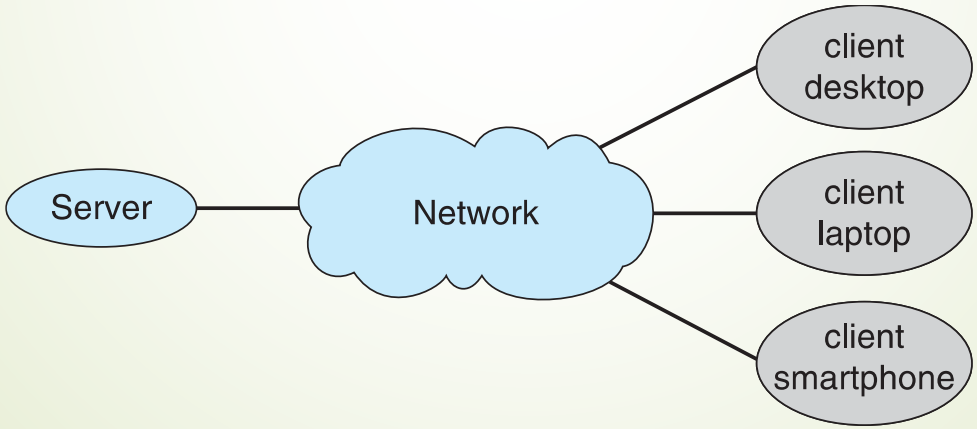
- Stand-alone general-purpose machines
  - But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers** (**thin clients**) are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks

Mobile

- Handheld smartphones, tablets, etc.
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Examples: **Apple iOS** and **Google Android**

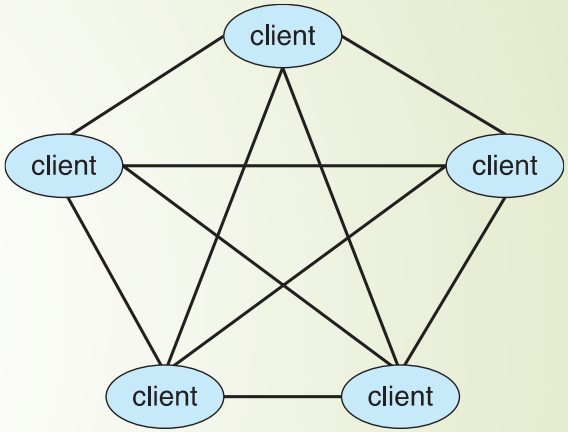
Client Server

- Client-Server Computing
  - Dumb terminals supplanted by smart PCs
  - Many systems now **servers**, responding to requests generated by **clients**
    - **Compute-server system** provides an interface to client to request services (i.e., database)
    - **File-server system** provides interface for clients to store and retrieve files



Peer-to-Peer

- Another model of distributed system
- P2P does not distinguish clients and servers
  - Each node acts as client, server, or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast requests and respond to requests for service via ***discovery protocol***
  - Examples: Napster and Gnutella, **Voice over IP** (**VoIP**) such as Skype



Cloud Computing

- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage

Cloud Computing (Cont.)

- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
  - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
  - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Free and Open-Source Operating Systems

- Operating systems made available in source-code format, rather than just binary **closed-source** and **proprietary**
  - Counter to the **copy protection** and **Digital Rights Management** (**DRM**) movement
  - Started by **Free Software Foundation** (**FSF**), which has "copyleft" **GNU Public License** (**GPL**)
    - Free software and open-source software are two different ideas championed by different groups of people
    - http://gnu.org/philosophy/open-source-misses-the-point.html/

Free and Open-Source Operating Systems

- Examples: **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Can use VMM to run guest OS for exploration
  - VMware Player (Free on Windows)
  - Virtualbox (open source and free on many platforms - http://www.virtualbox.com)

Operating Systems Services

Operating System Services

- OSes provide services to programs and users
- For users:
  - **User interface** (**UI**)
    - E.g. **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **touch-screen**, **Batch**
  - **Program execution** - to load a program into memory and to run that program, end execution, either normally or abnormally

Operating System Services

- For users (cont.):
  - **I/O operations** - needed by a running program, for a file or an I/O device
  - **File-system manipulation** - is of special interest
    - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management

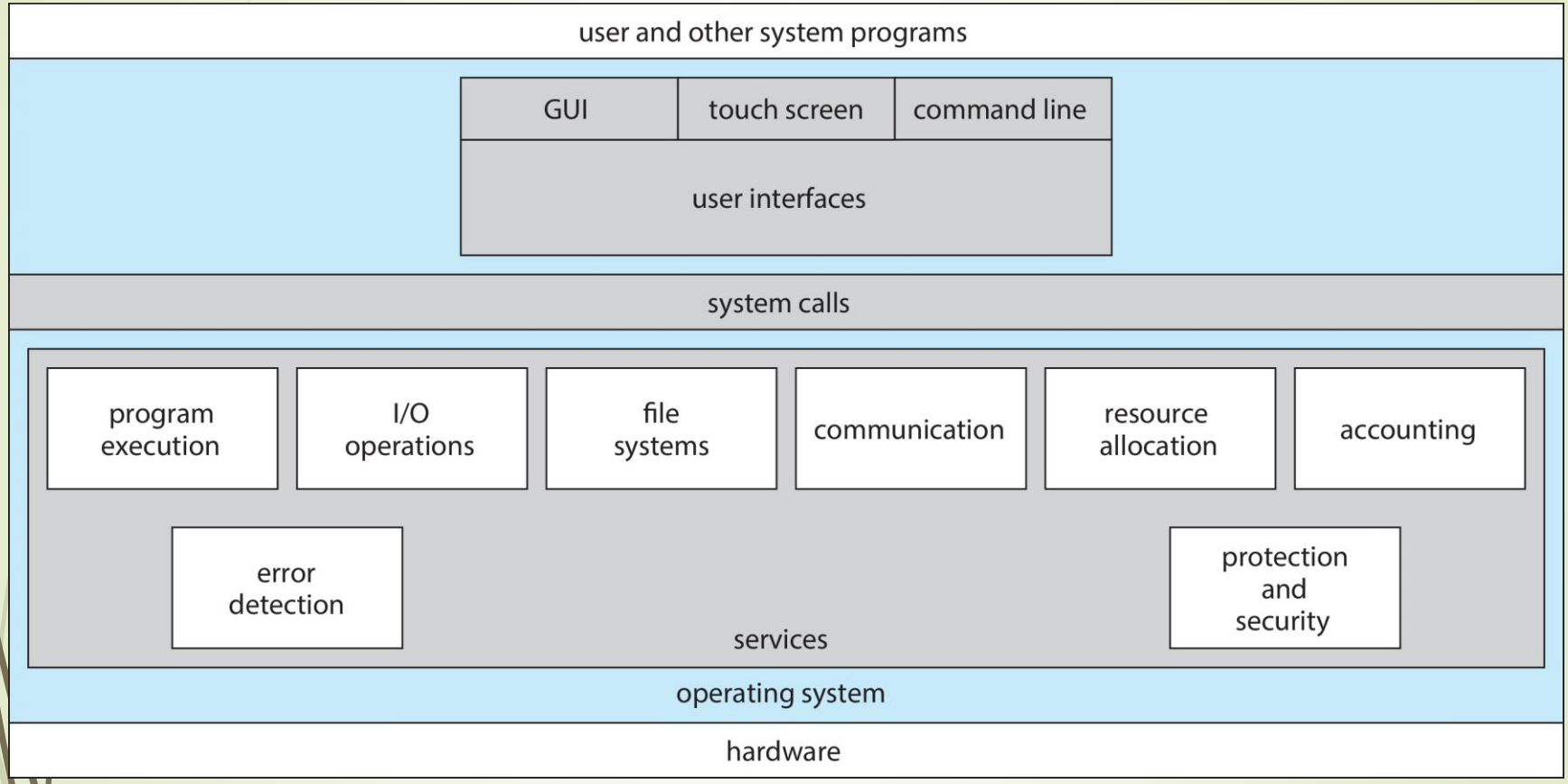
Operating System Services (Cont.)

- For users (Cont.):
  - **Communications** – to exchange information, on the same computer or over a network
    - via shared memory or through message passing
  - **Error detection** – OS needs to be constantly aware of possible errors
    - in CPU and memory, in I/O devices, in user programs
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the efficient use of the system for users and programmers

Operating System Services (Cont.)

- OS functions for ensuring efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple jobs running concurrently, resources must be allocated
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices
  - **Logging -** To keep track of which users use how much and what kinds of resources
  - **Protection and security -** to control use of information, concurrent processes should not interfere with each other
    - **Protection**: ensuring controlled access to resources
    - **Security**: requires user authentication, and defending external I/O devices from invalid access

Overview of OS Services

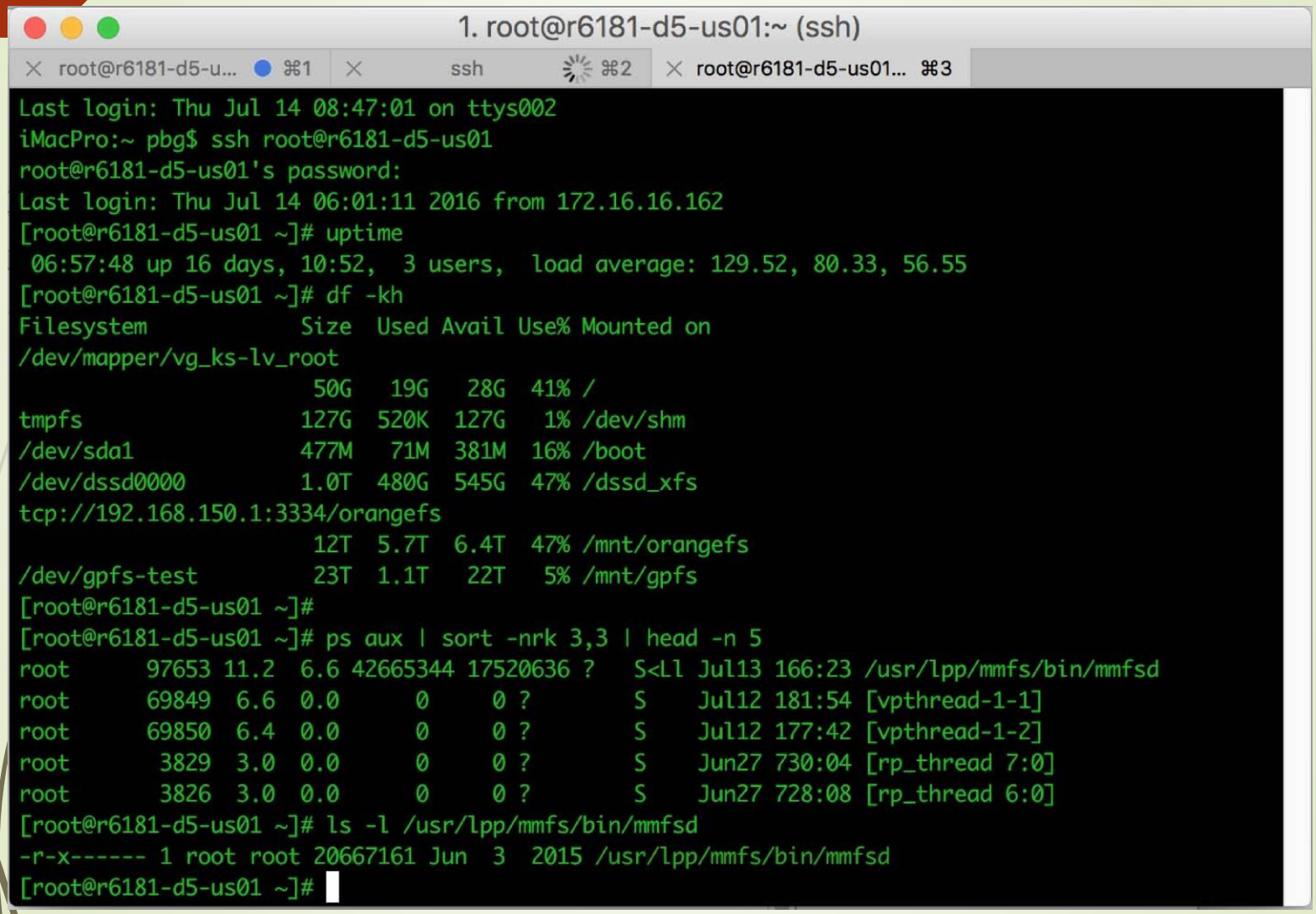


[Source: Operating Systems Concepts, 10th ed.]

Command Line interpreter

- CLI allows direct command input
  - either implemented in kernel, or by system programs
  - Sometimes multiple flavors implemented – **shells**
- Primarily fetches a user command and executes it
  - Either built-in commands, or just names of programs

Bourne Shell Command Interpreter

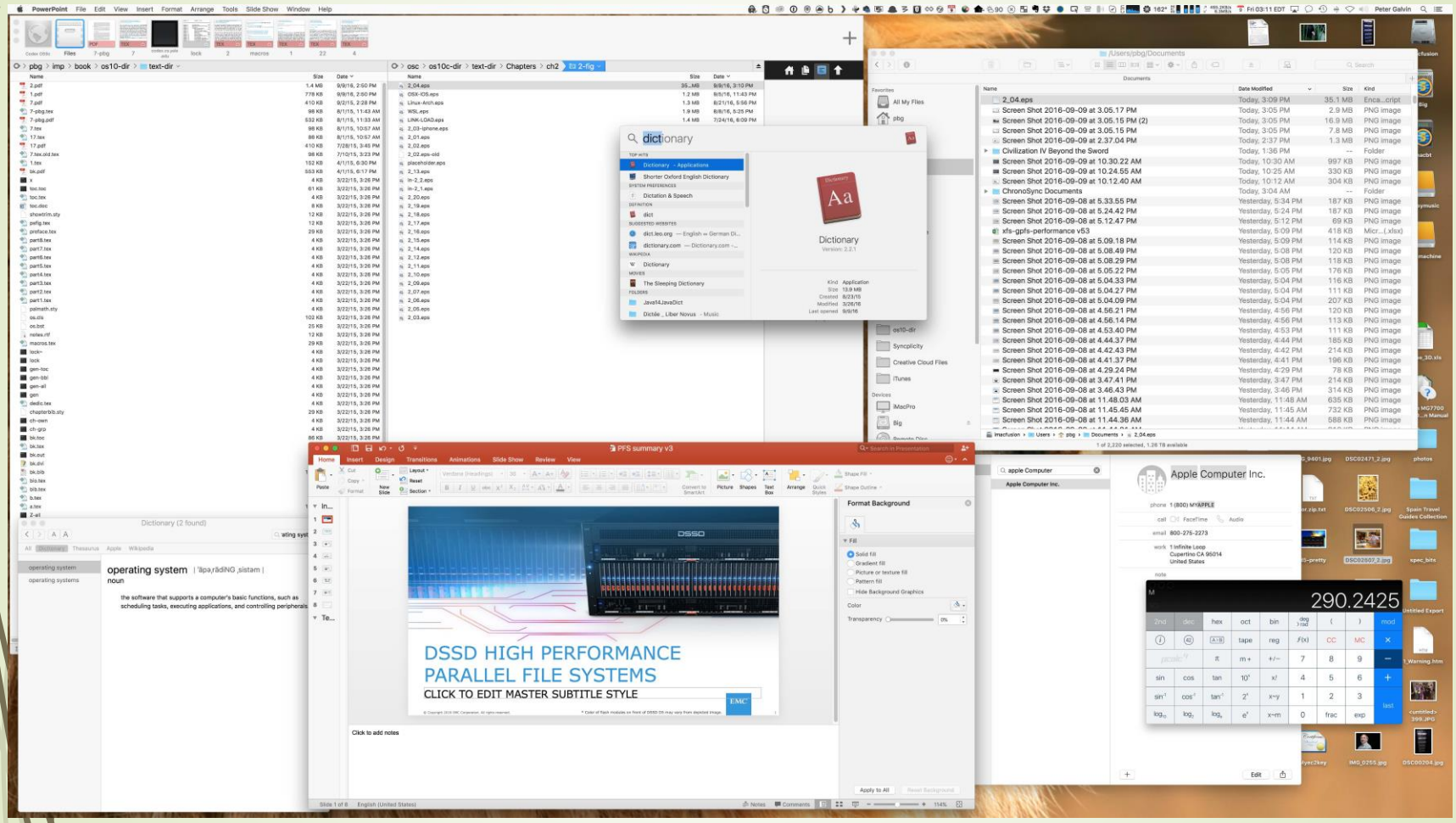


[Source: Operating Systems Concepts, 10th ed.]

User Operating System Interface - GUI

- User-friendly **desktop** metaphor via mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Mouse buttons over objects can cause various actions
    - provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel and shells
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

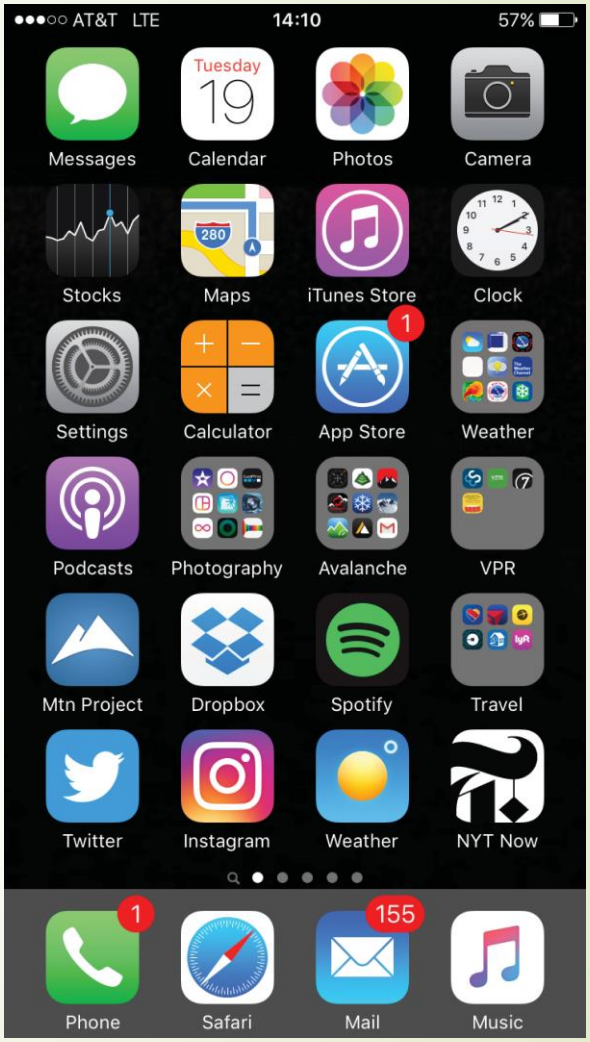
The Mac OS X GUI



[Source: Operating Systems Concepts, 10th ed.]

Touchscreen Interfaces

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

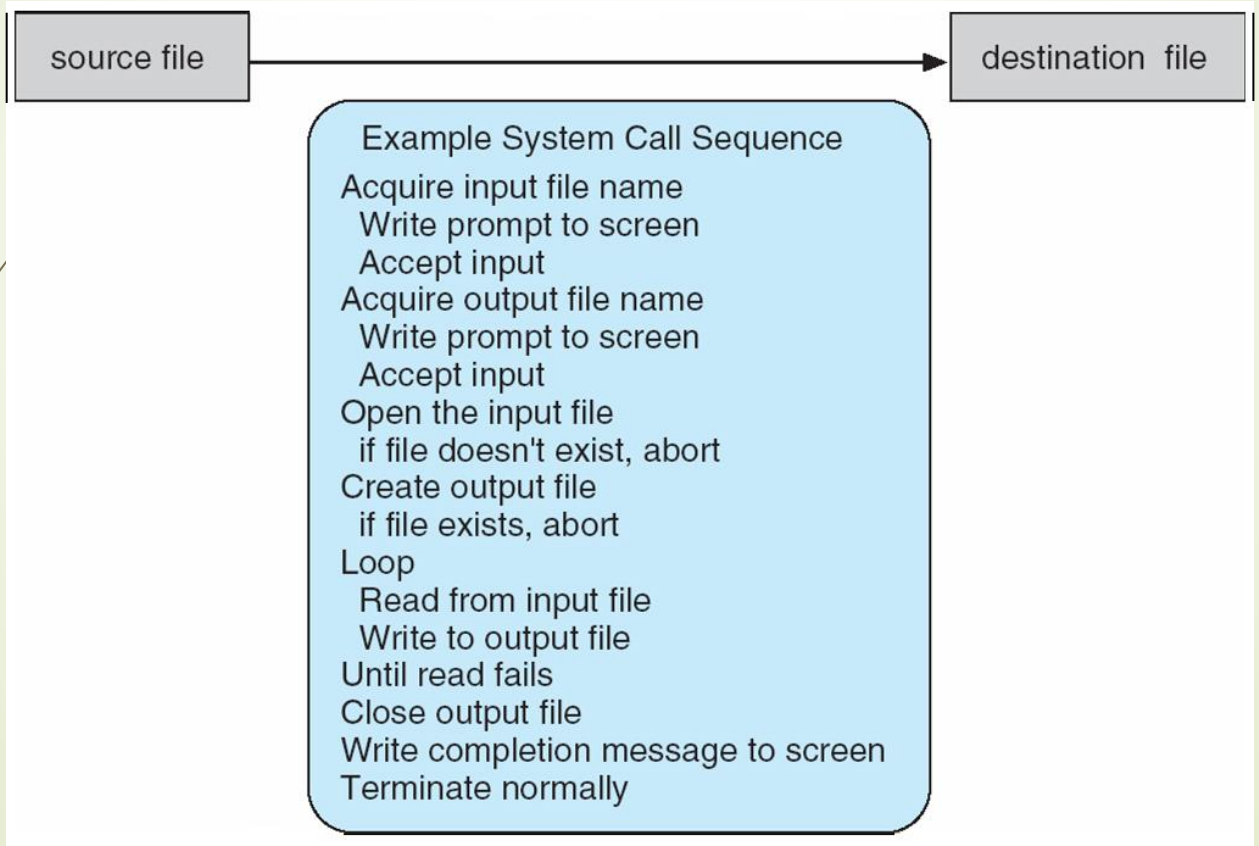


System Calls

- Programming interface to OS services
  - Typically written in a high-level language (C or C++)
  - Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct call
- 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)

Example of System Calls

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



[Source: Operating Systems Concepts, 10th ed.]

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

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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

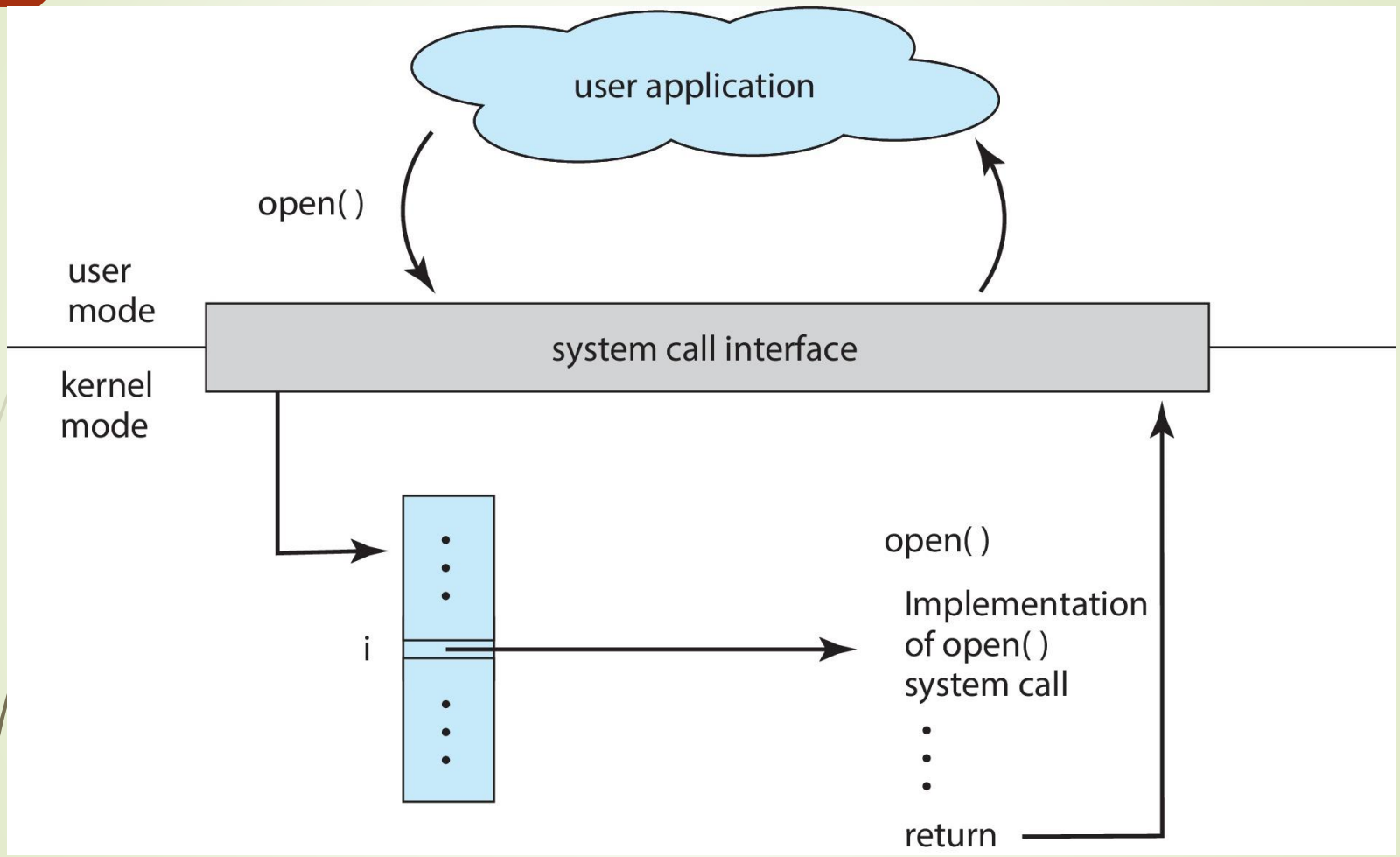
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns –1.

[Source: Operating Systems Concepts, 10th ed.]

System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
  - It invokes the intended system call in OS kernel and returns status and any return values
- The caller needs to know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result
  - 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



[Source: Operating Systems Concepts, 10th ed.]

System Call Parameter Passing

- Parameters needed according to OS and the system call
- Three general methods used to pass parameters to the OS
  - Simplest: parameters in registers
    - But the number or length of parameters are limited
  - 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 OS

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls (Cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

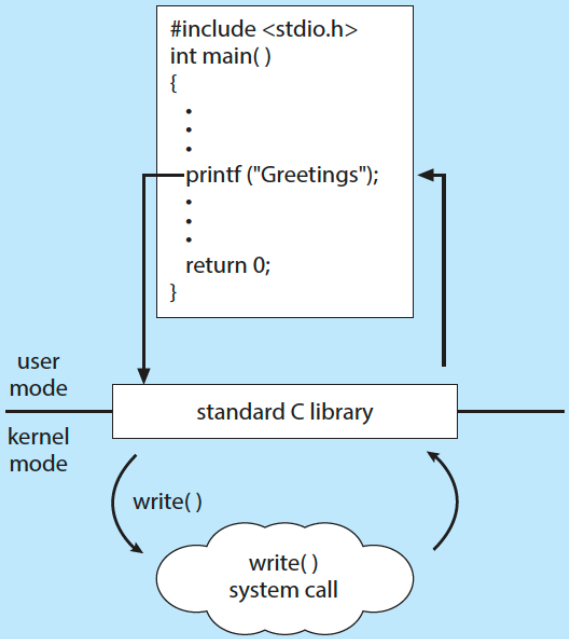
[Source: Operating Systems Concepts, 10th ed.]

Standard C Library Example

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

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



[Source: Operating Systems Concepts, 10th ed.]

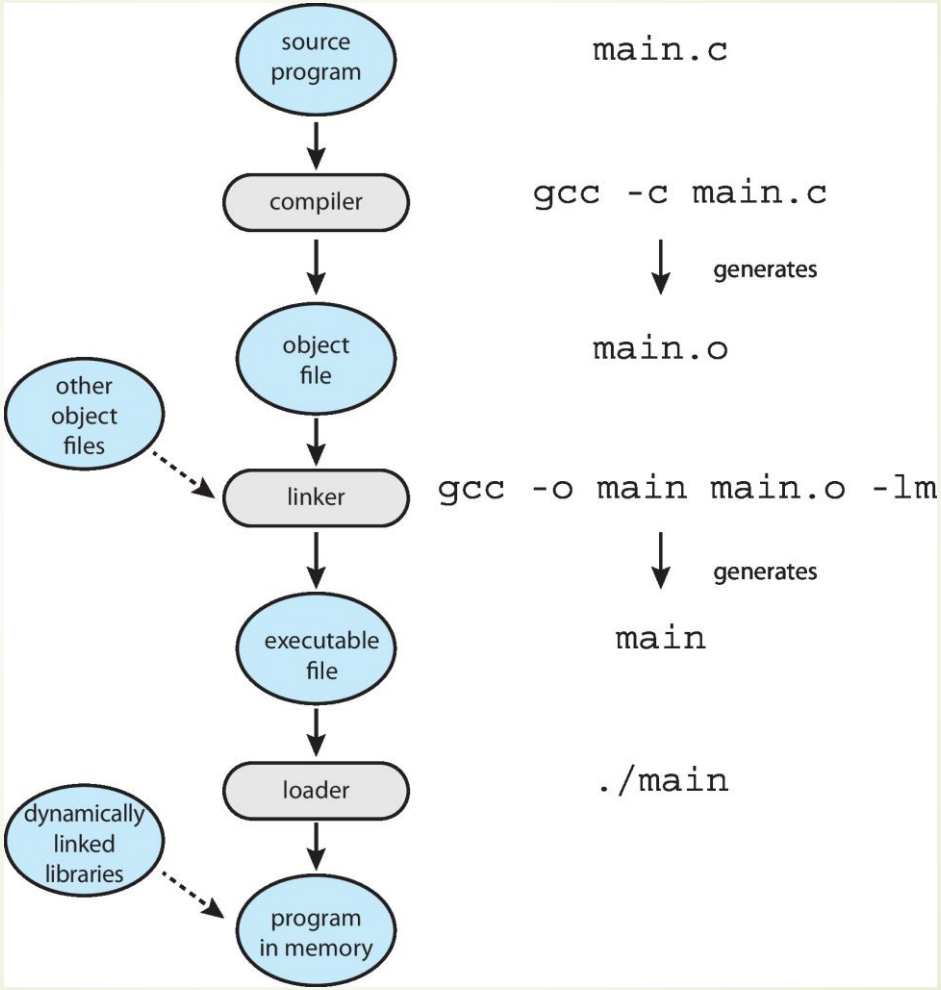
System Services

- Most users' view of the OS is defined by system programs, not the actual system calls
- System programs provide a convenient environment for program development and execution
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Some are simply user interfaces to system calls; others are considerably more complex

Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
    - But modern general purpose systems don't link libraries into executables
    - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded at runtime as needed, shared by all that use the same version of that same library (loaded once)
  - Program resides on secondary storage as binary executable
- Executables must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Object, executable files have standard formats, so OS knows how to load and start them

The Role of the Linker and Loader



[Source: Operating Systems Concepts, 10th ed.]

Why Applications are OS Specific

- Apps compiled on one system usually not executable on other OS
  - Each OS provides its own unique system calls
  - Unique file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple OS
  - App written in languages that include a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each OS to run on each
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given OS on a given architecture, CPU, etc.

OS Design and Implementation

Design and Implementation

- Design and Implementation of OS is not "solvable", but some approaches have proven successful
  - Internal structure of different OS can vary widely
  - Start the design by defining goals and specifications
  - Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**

Policy and Mechanism

- **Policy**: **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism**: **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
  - It allows maximum flexibility if policy decisions are to be changed later
  - Example: change 100 to 200

Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest level in assembly
  - Main body in C
  - System programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating System Structure

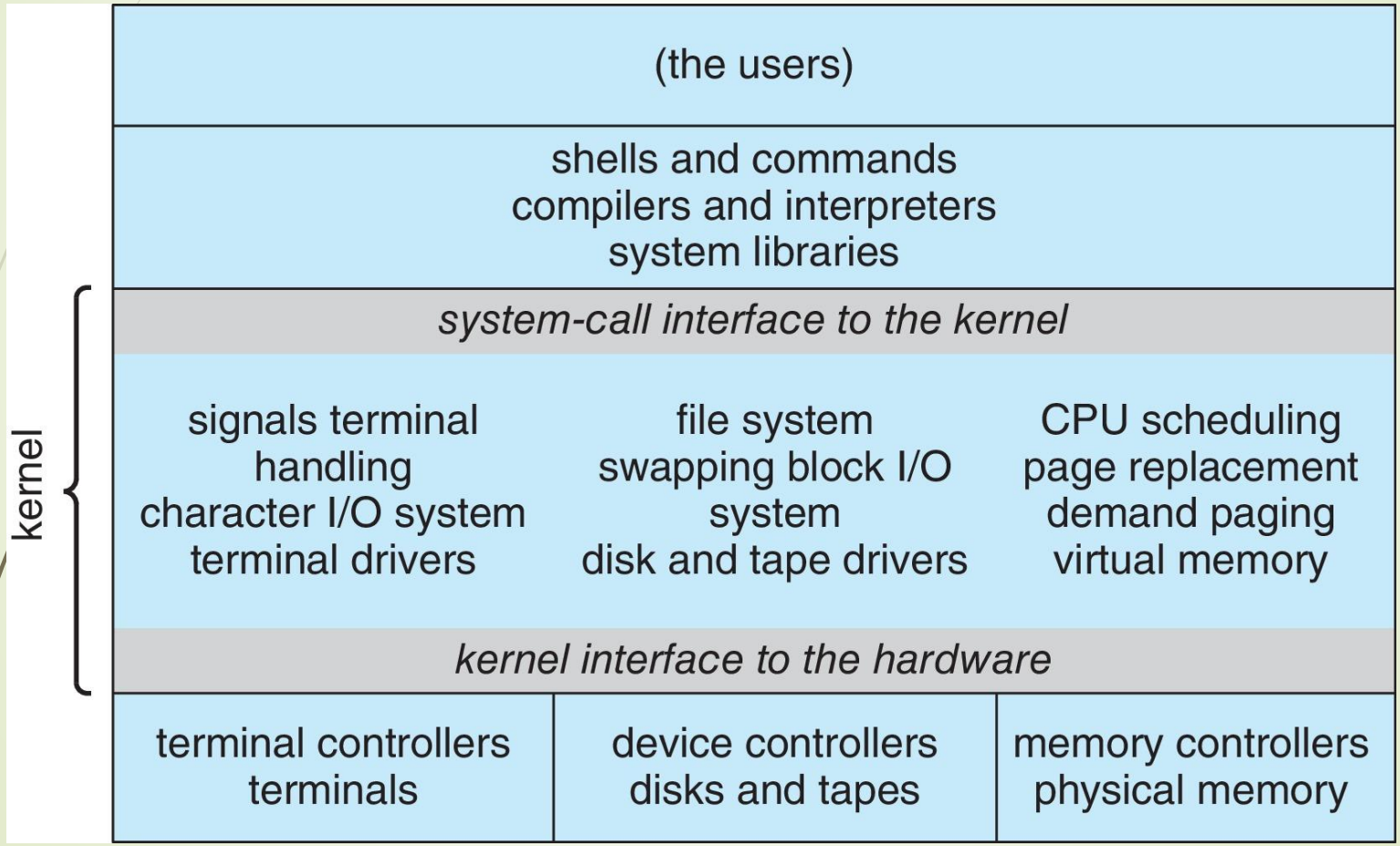
- General-purpose OS is a very large program
- Various ways to structure OS
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

Monolithic Structure – Original UNIX

- UNIX had limited structuring given limited hardware functionality
- The UNIX OS consists of two separable parts
  - System 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 OS functions in one level

Traditional UNIX System Structure

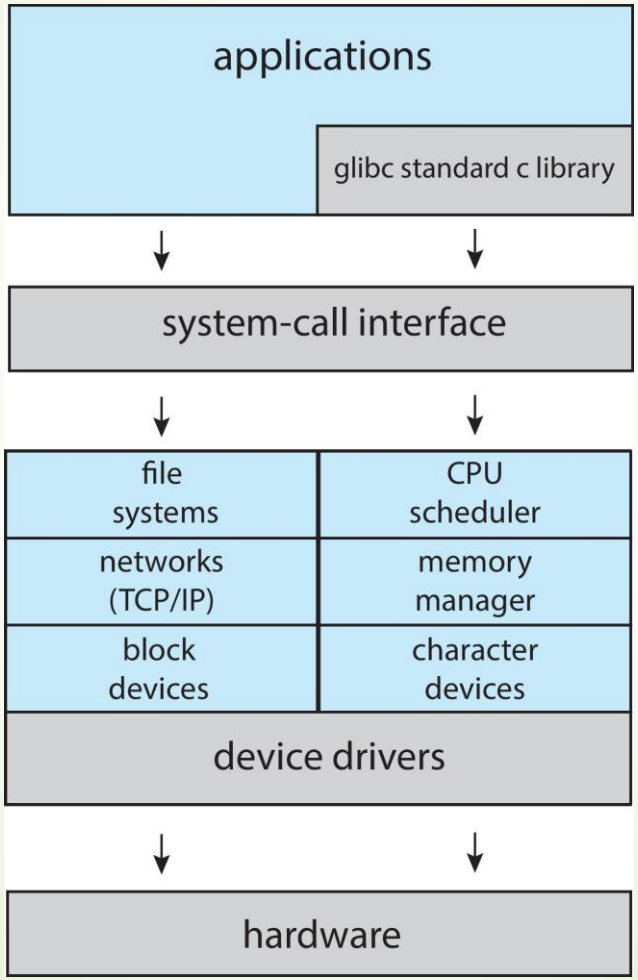
Beyond simple but not fully layered



[Source: Operating Systems Concepts, 10th ed.]

Linux System Structure

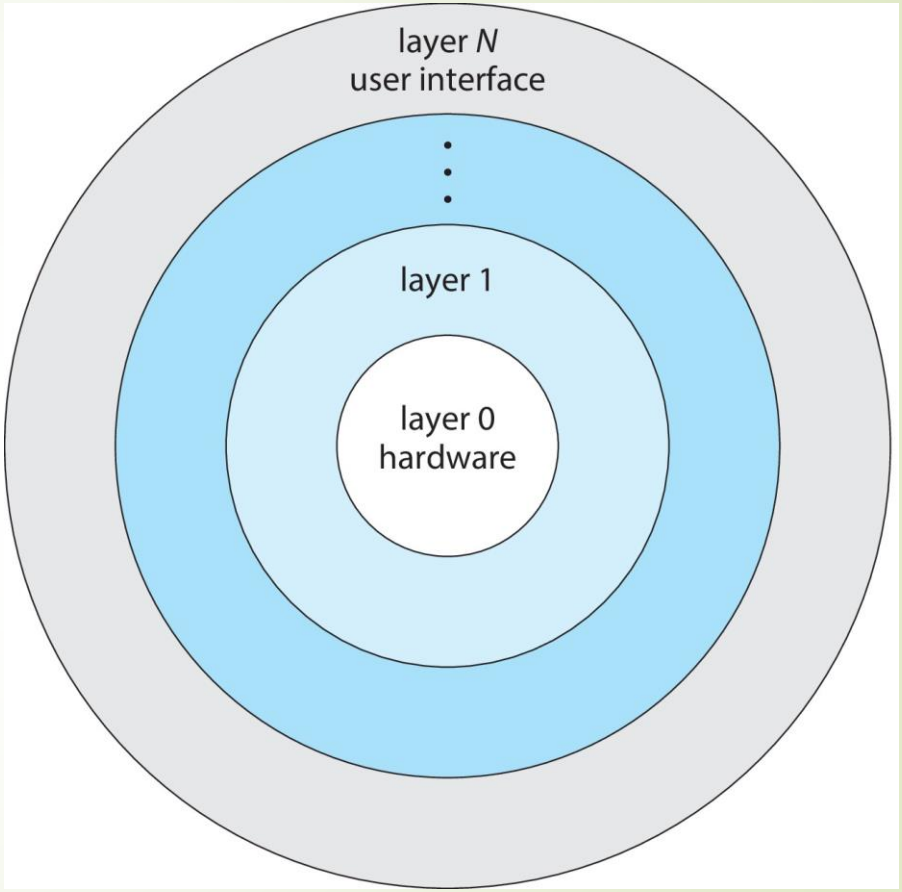
Monolithic plus modular design



[Source: Operating Systems Concepts, 10th ed.]

Layered Approach

- OS is divided into layers (levels), each built on top of lower layers
  - Hardware: the bottom layer (layer 0)
  - User interface: the highest (layer N)
- With modularity, layers are selected such that each only uses functions (operations) and services of lower-level layers

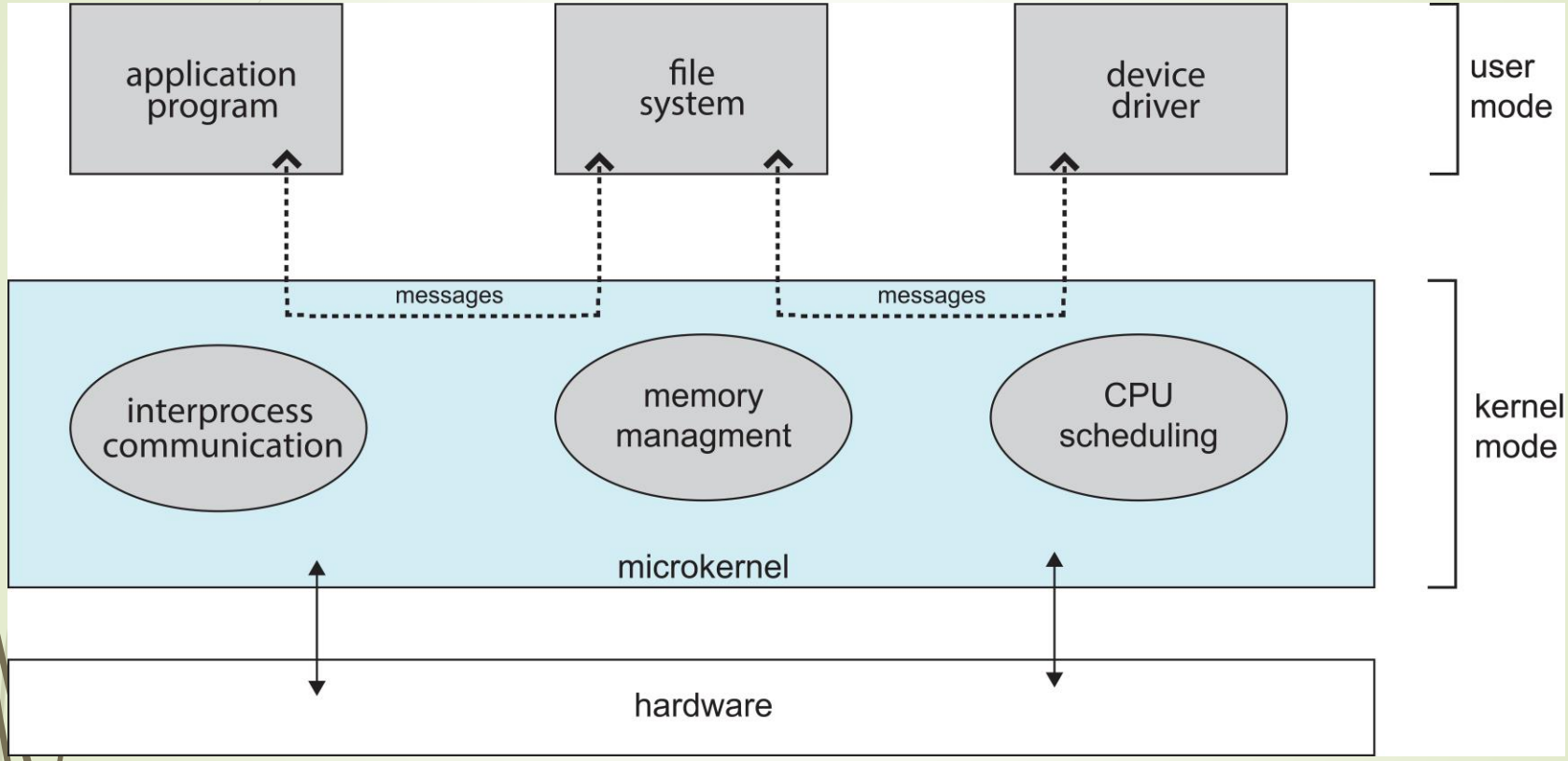


[Source: Operating Systems Concepts, 10th ed.]

Microkernels

- Moves as much from the kernel into user space
  - **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



[Source: Operating Systems Concepts, 10th ed.]

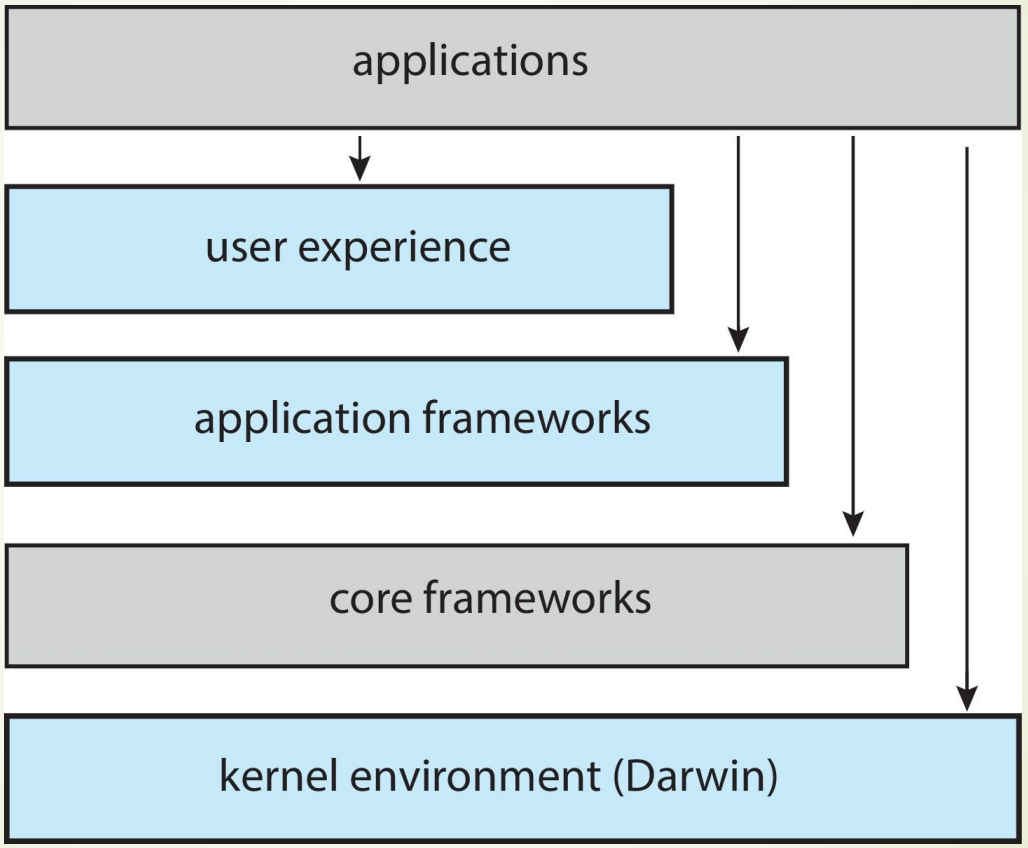
Modules

- Many modern OS implement **loadable kernel modules** (**LKMs**)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but more flexible
  - Linux, Solaris, etc.

Hybrid Systems

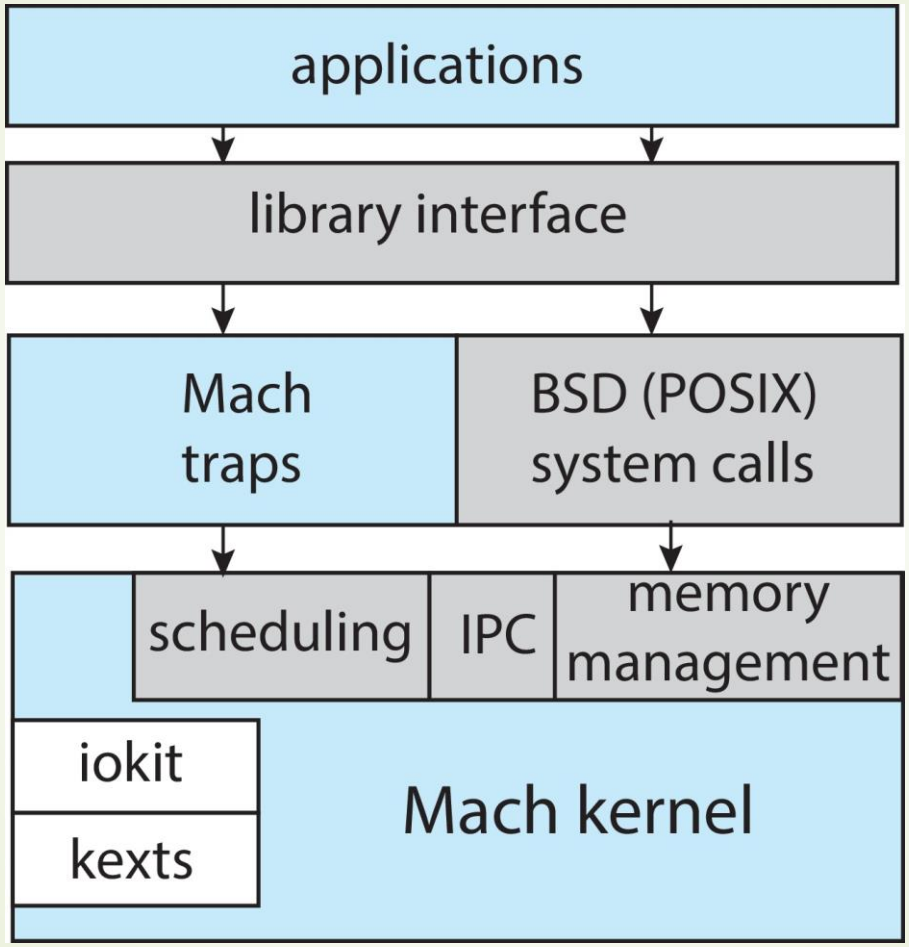
- Most modern OS are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for 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**)

macOS and iOS Structure



[Source: Operating Systems Concepts, 10th ed.]

Darwin

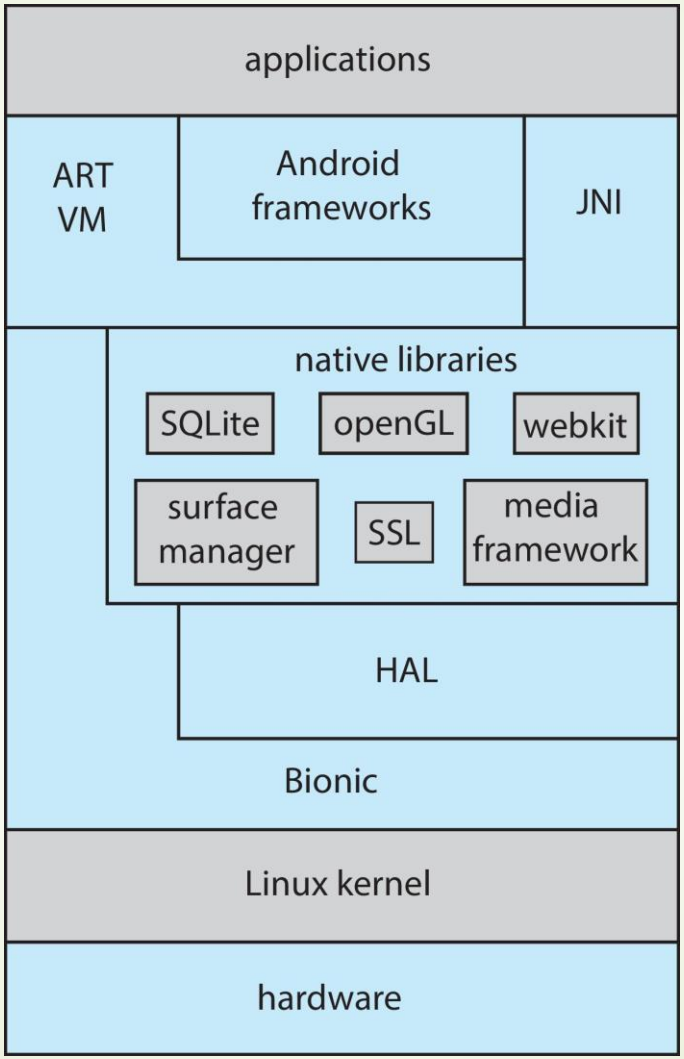


[Source: Operating Systems Concepts, 10th ed.]

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



[Source: Operating Systems Concepts, 10th ed.]

Building and Booting an Operating System

- OS generally designed to run on systems with variety of peripherals
- Commonly, OS already installed on purchased computer
  - If building an OS from scratch
    - Write/modify the OS source code
    - Configure the OS for the system on which it will run
    - Compile the OS
    - Install the OS
    - Boot the computer with its new OS

Building and Booting Linux

- Download Linux source code (http://www.kernel.org)
- Configure kernel via "`make menuconfig`"
- Compile the kernel using "`make`"
  - Produces `vmlinuz`, the kernel image
  - Compile kernel modules via "`make modules`"
  - Install kernel modules into `vmlinuz` via "`make modules_install`"
  - Install new kernel on the system via "`make install`"

System Boot

- When system powers on, execution starts at a fixed memory location
  - OS must be made available to hardware so it can be started
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
  - Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**

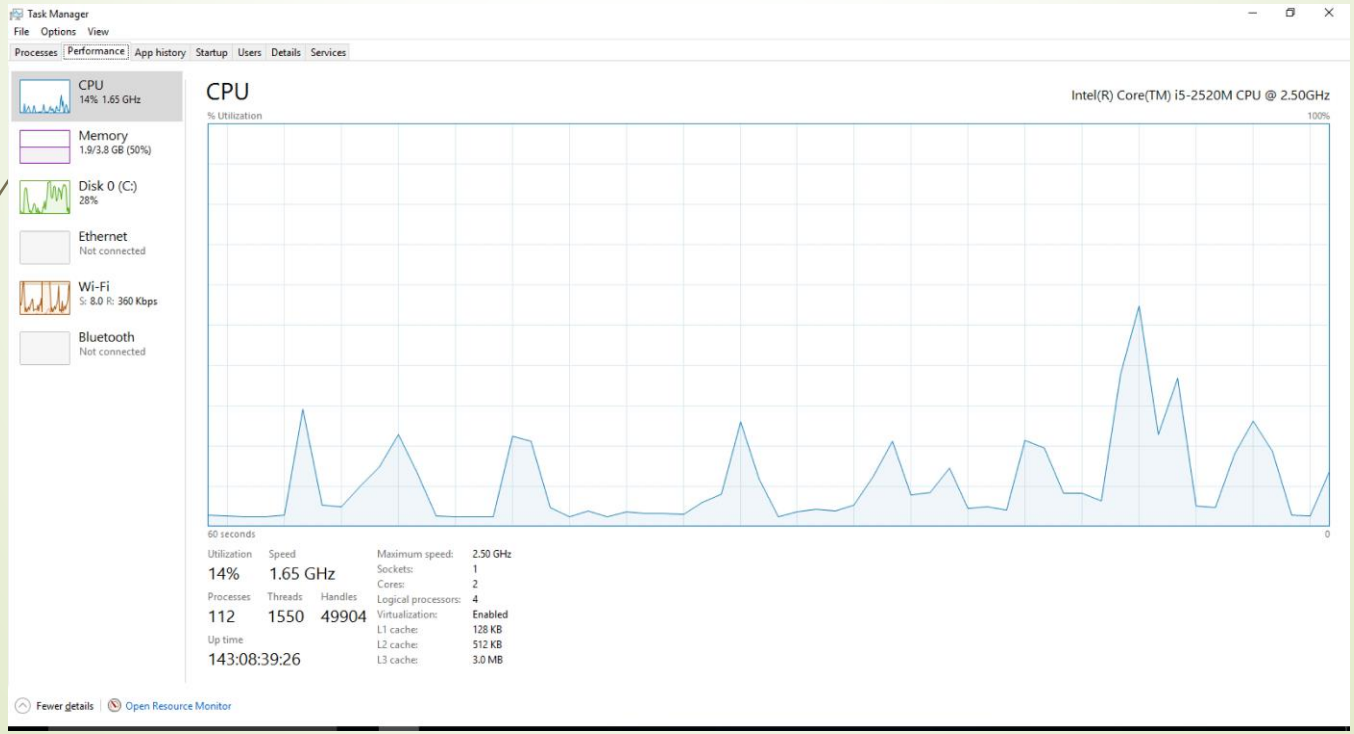
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
  - OS generates **log files** containing error information
  - Failure of an application can generate **core dump** file capturing memory of the process
  - OS can generate **crash dump** file containing kernel memory
- **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



[Source: Operating Systems Concepts, 10th ed.]

Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

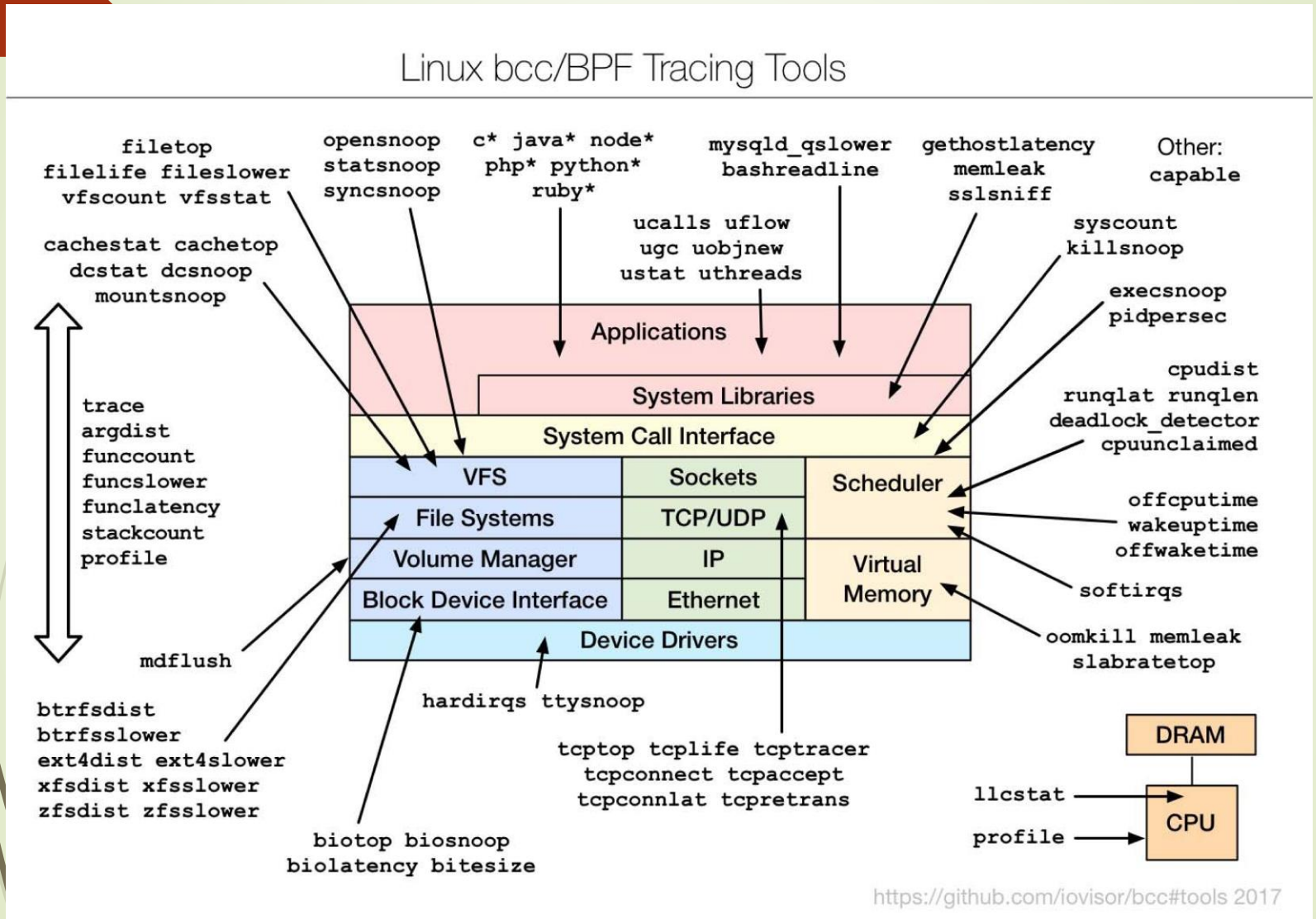
BCC

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, disksnoop.py traces disk I/O activity

```
TIME(s)            T      BYTES      LAT(ms)
1946.29186700      R      8              0.27
1946.33965000      R      8              0.26
1948.34585000      W      8192           0.96
1950.43251000      R      4096           0.56
1951.74121000      R      4096           0.35
```

- Many other tools (next slide)

Linux bcc/BPF Tracing Tools



[Source: Operating Systems Concepts, 10th ed.]

End of Module 0