INTRO AND OPERATING SYSTEM STRUCTURES

Chapter 1 and 2

GOAL

Chapter 1

- Grand tour of major operating systems components
- Coverage of basic computer system organization

Chapter 2

- Services an operating system provides to users, processes, and other systems
- Various ways of structuring an operating system
- How operating systems are installed and customized and how they boot

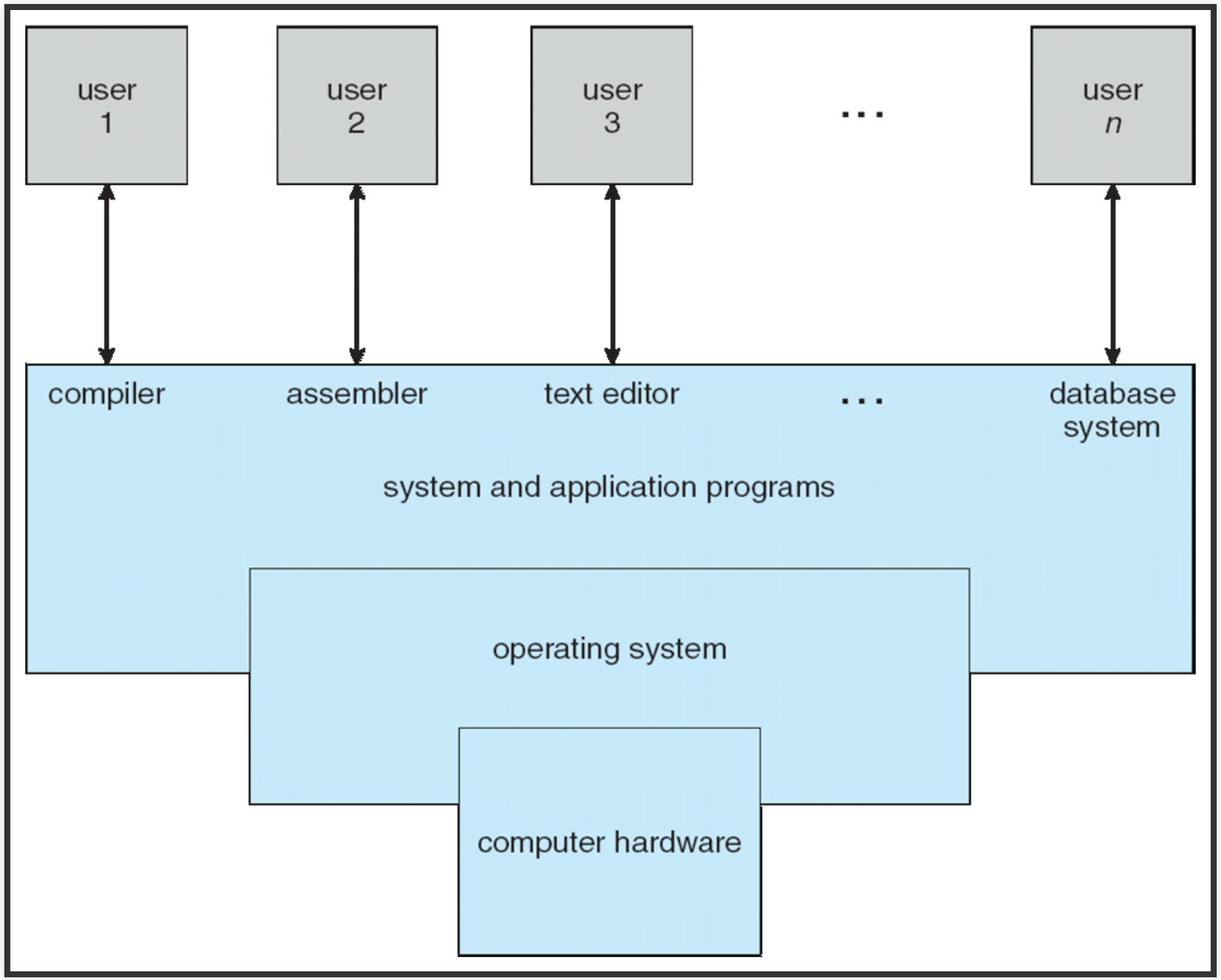
CHAPTER 1 - INTRODUCTION

WHAT OPERATING SYSTEMS DO

COMPUTER SYSTEM STRUCTURE

1. Hardware
2. Operating system
3. Application programs
4. Users

ABSTRACT VIEW



WHAT IS AN OPERATING SYSTEM?

- A program
  - intermediary between a user and the hardware
- Goals:
  - Execute user programs
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

USER VIEW

Role of OS

- PC/Laptop → Monitor, keyboard, mouse, system unit
  - Role: Ease of use and performance
- Mainframe
  - Role: Maximize resource utilization
- Workstations on network
  - Role: Compromise between individual usability and resource utilization

USER VIEW

Role of OS

- Mobile devices → Touch screen, voice input
  - Role: Power management, ease of use
- Embedded computers
  - Role: Designed to run without user intervention

SYSTEM VIEW

OS is a **resource allocator**

- CPU Time
- Memory space
- File storage space
- I/O Devices

SYSTEM VIEW

OS is a **control program**

- Prevents
  - errors
  - improper use

WHAT IS AN OPERATING SYSTEM?

⚠ No universally accepted definition

"Everything a vendor ships when you order an operating system" is good approximation

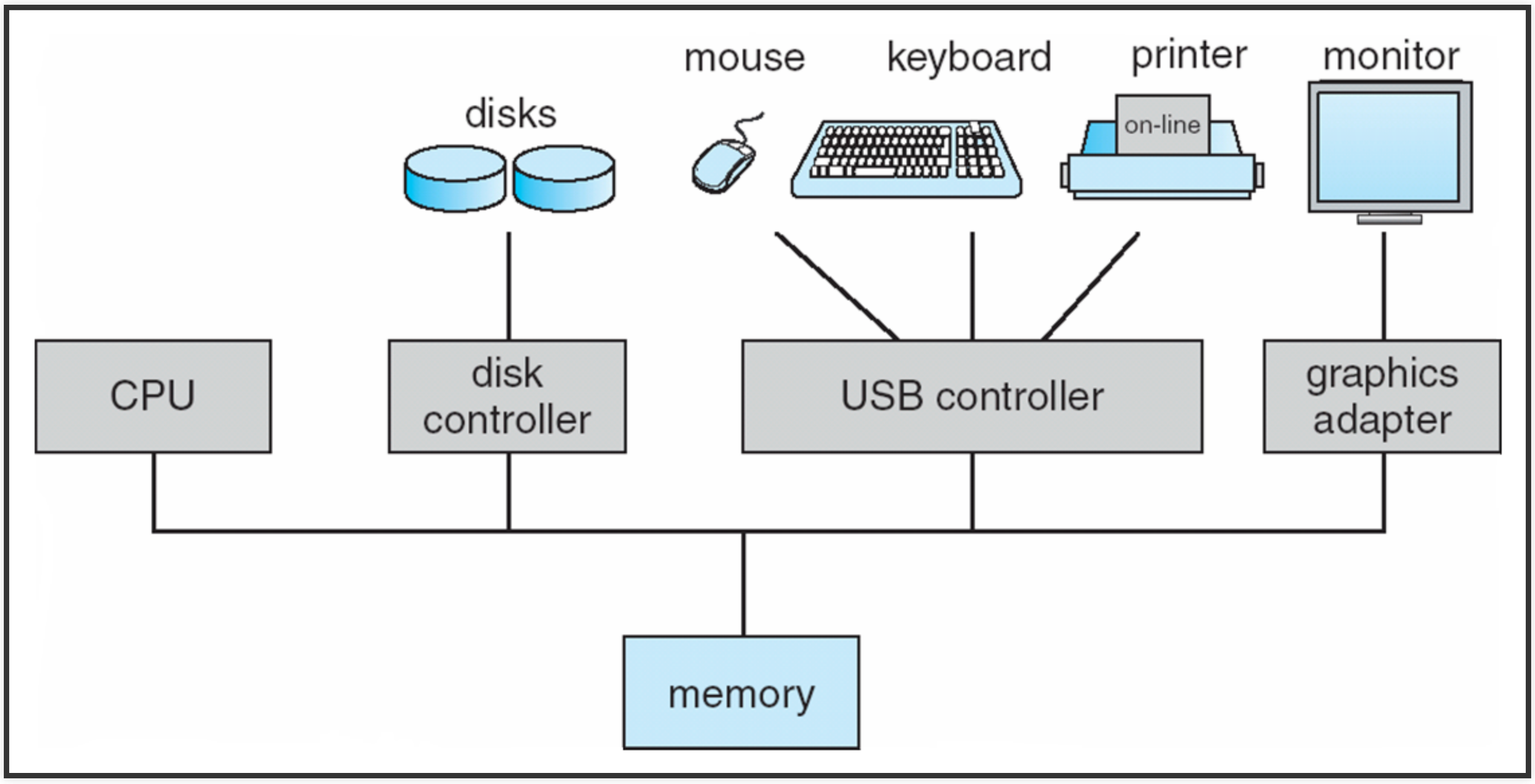
"The one program running at all times on the computer" is the kernel.

Everything else is either

- A system program (ships with the operating system)
- An application program

COMPUTER-SYSTEM ORGANIZATION

MODERN SYSTEM



INTERRUPTS

Event → signalled by interrupt

- Hardware: Trigger interrupt by sending a signal to CPU
- Software: By triggering a system call.

> An operating system is interrupt driven

INTERRUPT VECTOR

When system is interruptet, it immediately transfers execution to a fixed location

The operating system preserves the state of the CPU by storing registers and the program counter

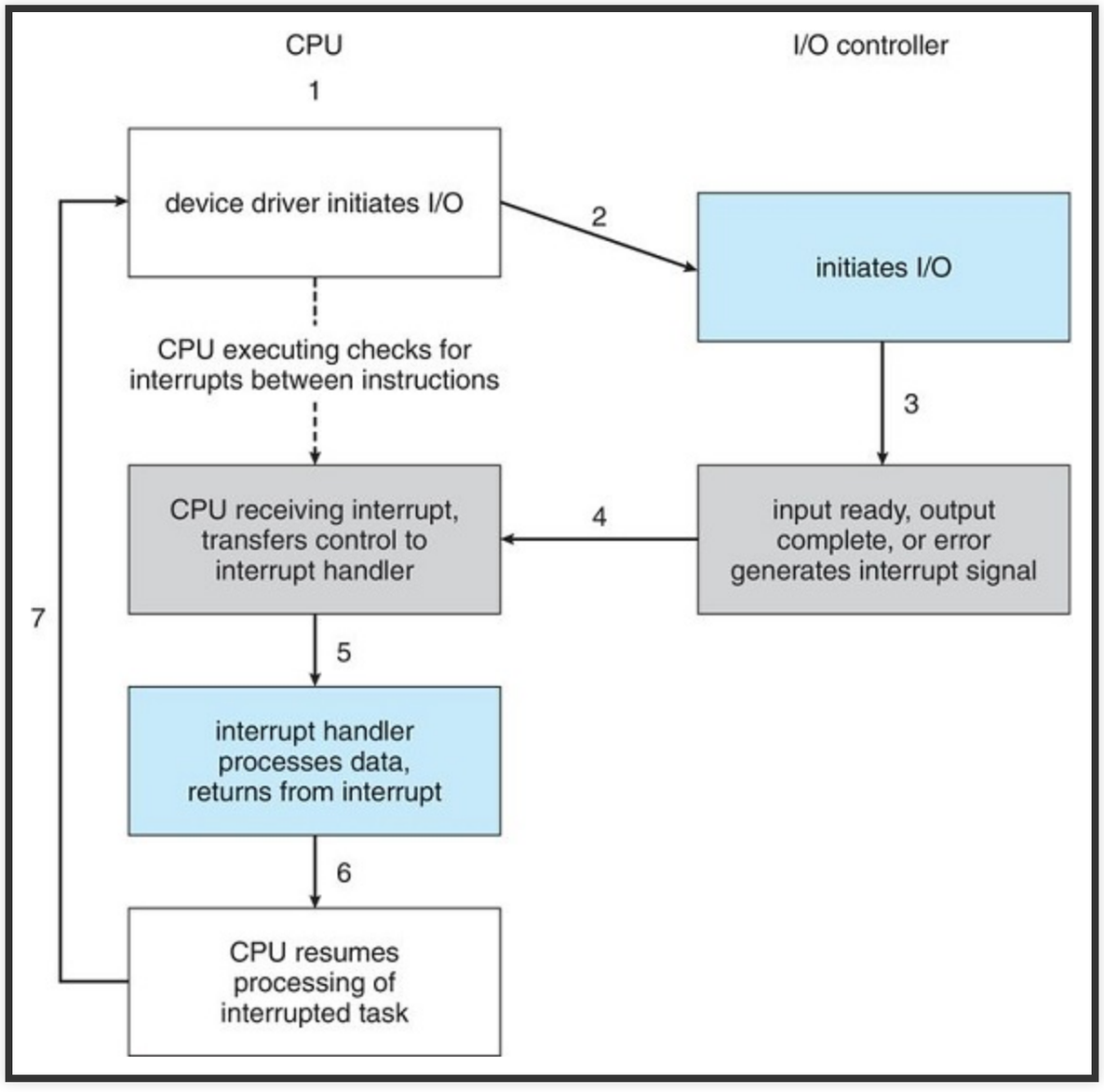
Interrupt Vector is an array with addresses of the interrupt service routines

A **trap** is a software-generated interrupt caused either by an error or a user request

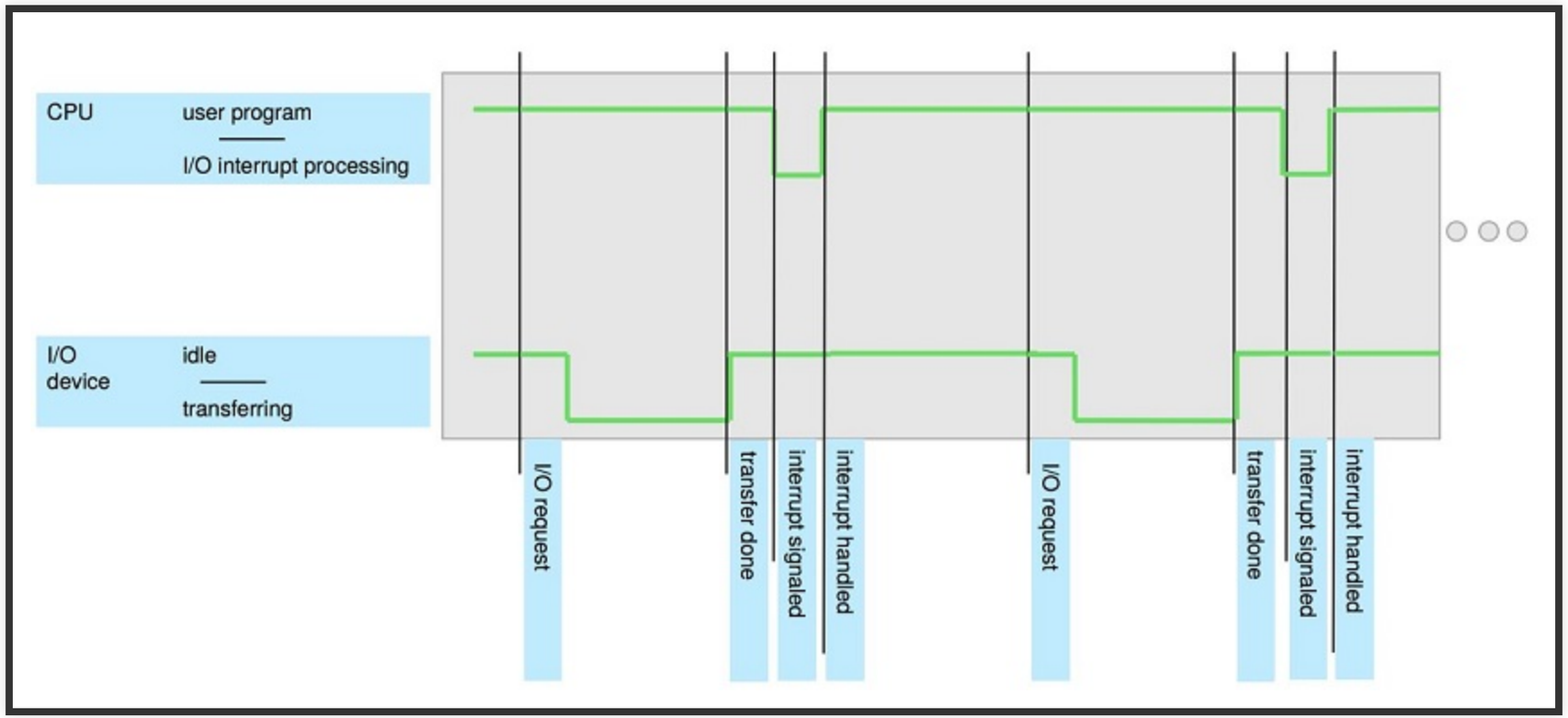
INTERRUPTS BY HARDWARE

- CPU hardware has a wire called the **interrupt-request line**
  - CPU senses after executing every instruction
- If detecting a controller has sent interrupt
  1. Reads interrupt number
  2. Jumps to **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector.

INTERRUPT IMPLEMENTATION



INTERRUPT TIMELINE



INTERRUPT REQUEST LINES

Most CPUs have 2

1. **Nonmaskable interrupt:** Reserved for events such as unrecoverable memory errors.
2. **Maskable:** Can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.

INTERRUPT PRIORITY LEVELS

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

LOADING INSTRUCTIONS

- CPU can load instructions only from memory
  - Programs must be loaded into memory first
- Main Memory → random-access memory (or RAM)
  - Implemented in semiconductor tech: dynamic random-access memory (DRAM)

STORAGE DEFINITIONS

- **Bit**: Basic unit, 0 or 1 / **Byte**: 8 Bits
- **Word**: Given computer architecture's native unit of data
  - 64-bit registers/64-bit memory addressing → 64-bit (8-byte) words
- **Kilobyte**: KB - 1.024 bytes
- **Megabyte**: MB - 1.024^2 bytes
- **Gigabyte**: GB - 1.024^3 bytes
- **Terabyte**: TB - 1.024^4 bytes
- **Petabyte**: PB - 1.024^5 bytes

VON NEUMANN ARCHITECTURE

Instruction–execution cycle

1. Fetch instruction from memory → instruction register
2. Instruction decoded → may cause operands to be fetched from memory → internal register
3. Instruction on the operands executed
4. Result may be stored back in memory

> Memory unit sees only a stream of memory addresses

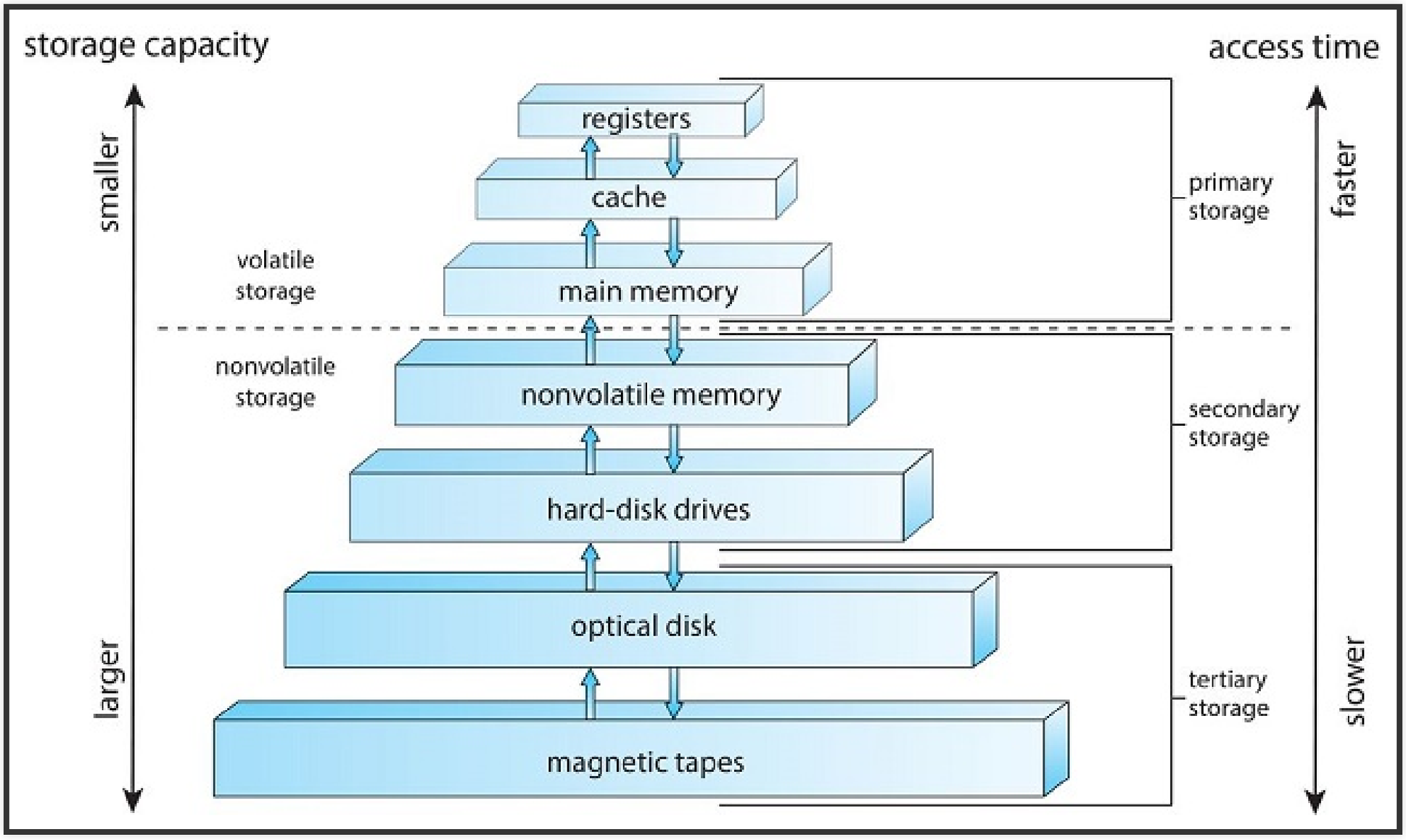
VON NEUMANN ARCHITECTURE

Ideally, we want the programs and data to reside in main memory permanently.

Not possible:

1. Main memory is usually too small to hold program and data
2. Main memory is volatile: it loses its contents when power is turned off.

STORAGE STRUCTURE



I/O STRUCTURE

General-purpose computer system → CPUs and device controllers connected through a common bus.

Device controller maintains some local buffer storage and a set of special-purpose registers.

Device driver for each device controller:

- Understands the device controller
- Provides the rest of the OS with uniform interface to device.

OPERATION

To start an I/O operation

- Driver loads the appropriate registers
- Controller, examines the registers to determine action to take
  - Example: "read a character from the keyboard"
- Controller starts the transfer of data from device to its local buffer.
- Once transfer complete → controller informs the driver via an interrupt
- Driver returns control to the operating system

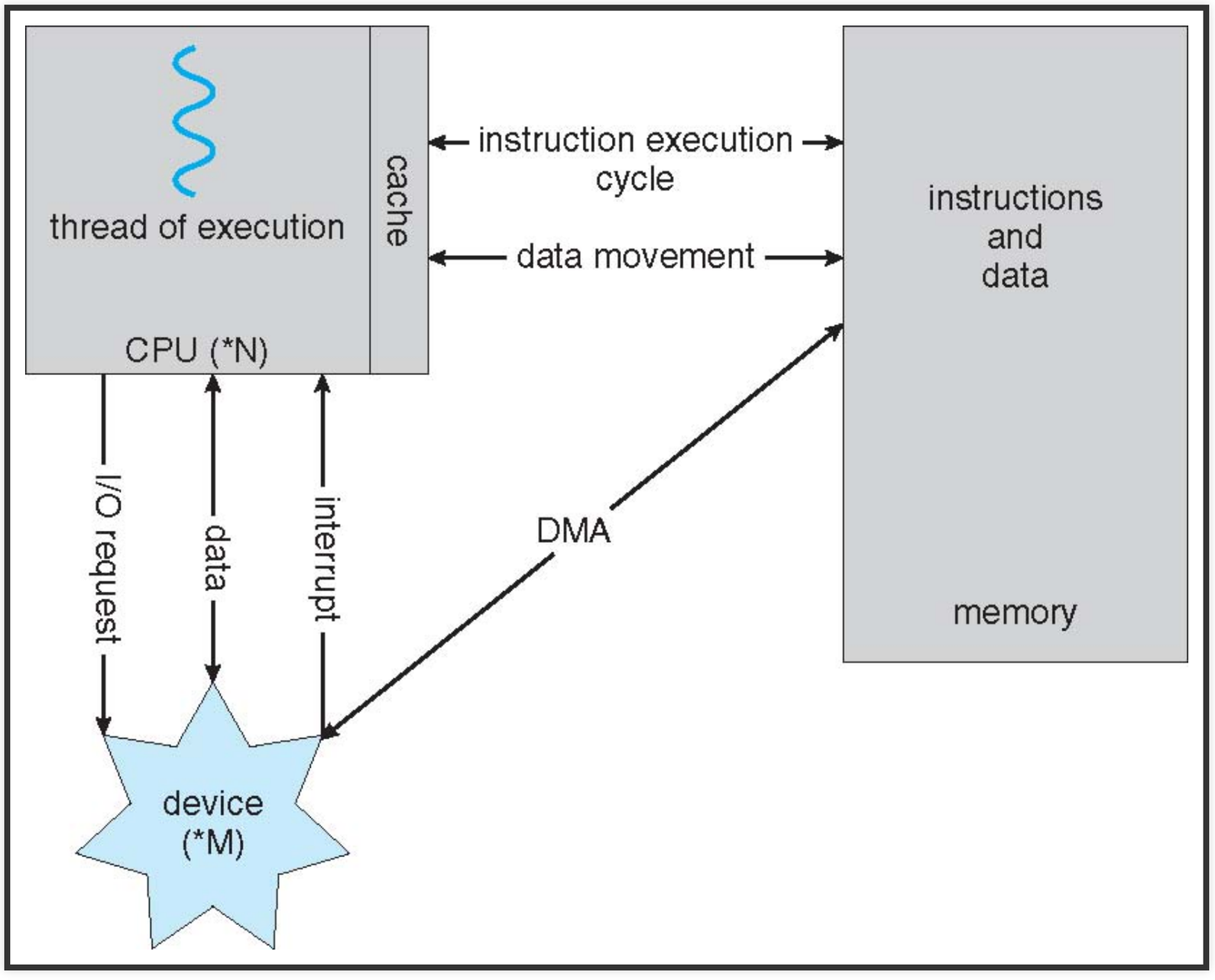
OPERATION

⚠ Interrupt-driven I/O is fine for moving small amounts of data → can produce high overhead when used for bulk data movement such as disk I/O .

DIRECT MEMORY ACCESS (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- One interrupt per block, not one per byte

HOW IT WORKS


thread of execution
cache
CPU (*N)
instruction execution cycle
data movement
instructions and data
I/O request
data
interrupt
DMA
memory
device (*M)

COMPUTER-SYSTEM ARCHITECTURE

SINGLE-PROCESSOR SYSTEMS

Previously, systems use a single general-purpose processor (PDAs through mainframes)

Most systems have special-purpose processors as well

- Graphics processor
- Disk processor
- Keybord processor

MULTIPROCESSOR SYSTEMS

Multiprocessors/Multicore systems growing in use and importance

- Also known as parallel systems, tightly-coupled systems

MULTIPROCESSOR ADVANTAGES

Advantages include

- Increased throughput
- Economy of scale
- Increased reliability – graceful degradation or fault tolerance

ASYMMETRIC MULTIPROCESSING

- Each processor is assigned a specific task.
- Boss processor controls the system
  - Other processors either look to the boss for instruction or have predefined tasks.

> This scheme defines a boss–worker relationship.

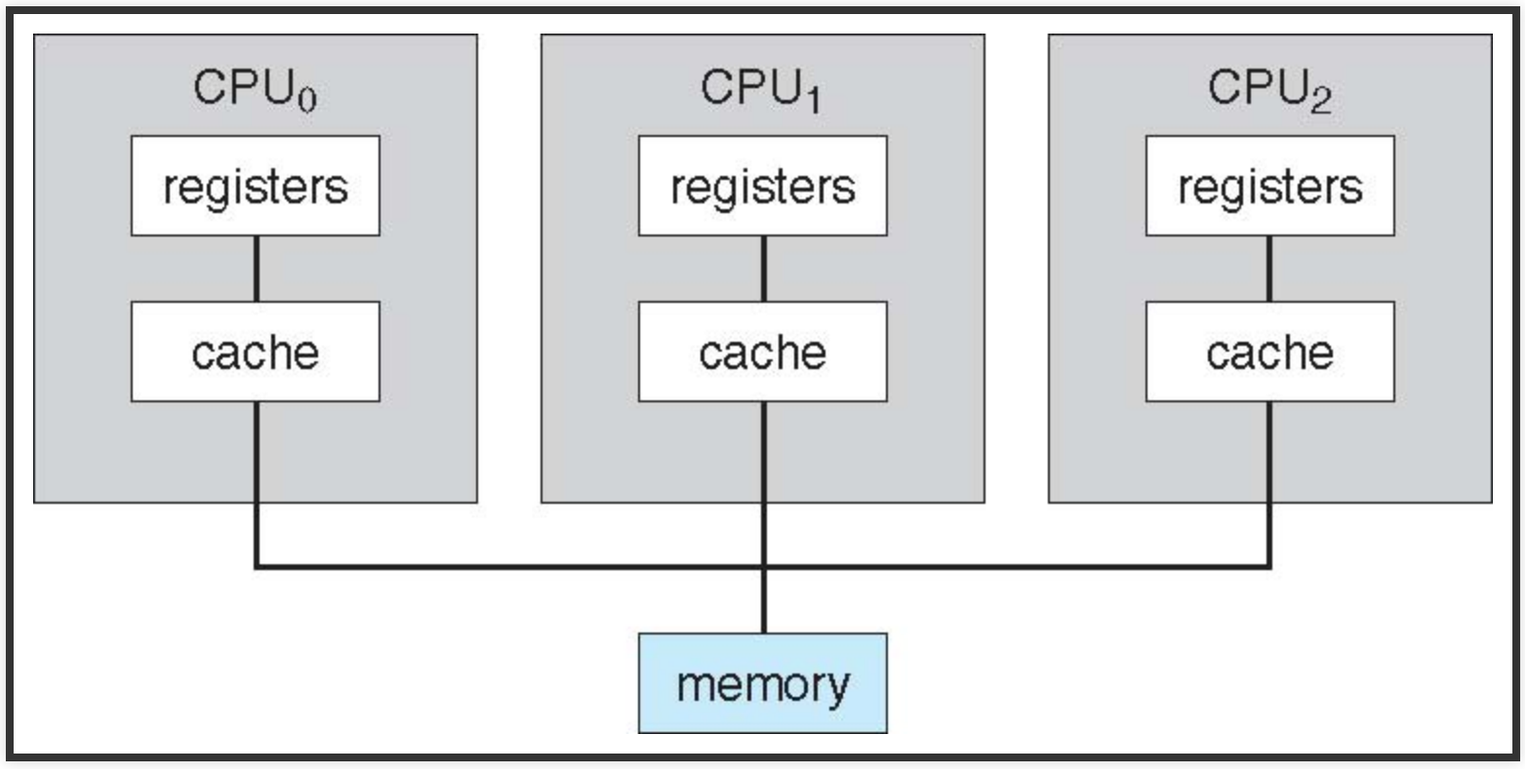
SYMMETRIC MULTIPROCESSING (SMP)

- Most common
- Each processor performs all tasks within the operating system.

Each processor

- **Own** set of registers, and local cache.
- **All** processors share physical memory.

SYMMETRIC MULTIPROCESSING ARCH.

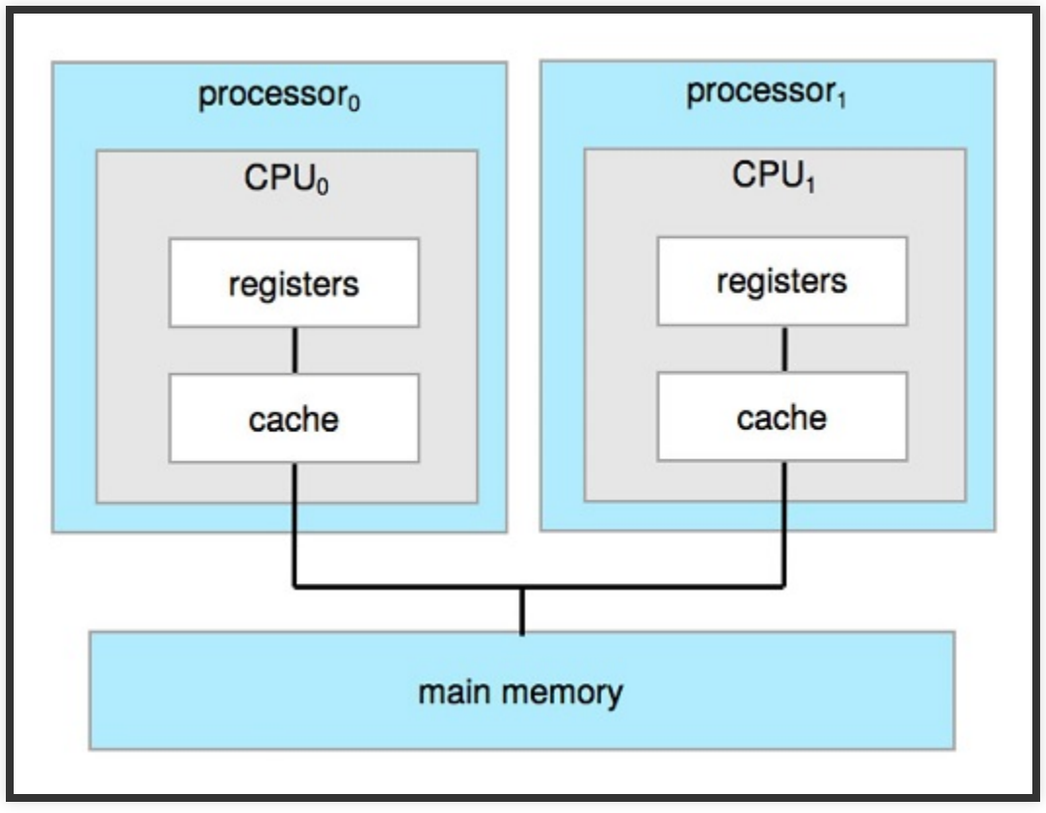


MULTICORE DESIGN

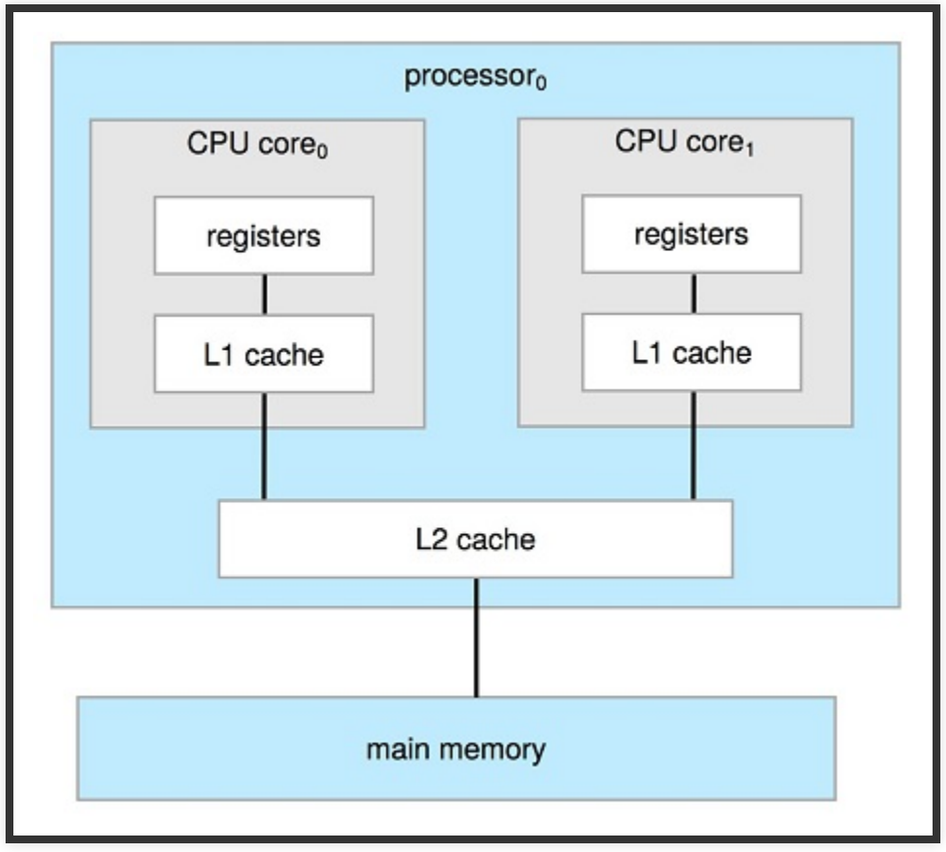
Multiple computing cores on a single chip.

Efficient: on-chip communication is faster than between-chip communication

DUAL-CORE DESIGN



DUAL-CORE DESIGN



CLUSTERED SYSTEMS

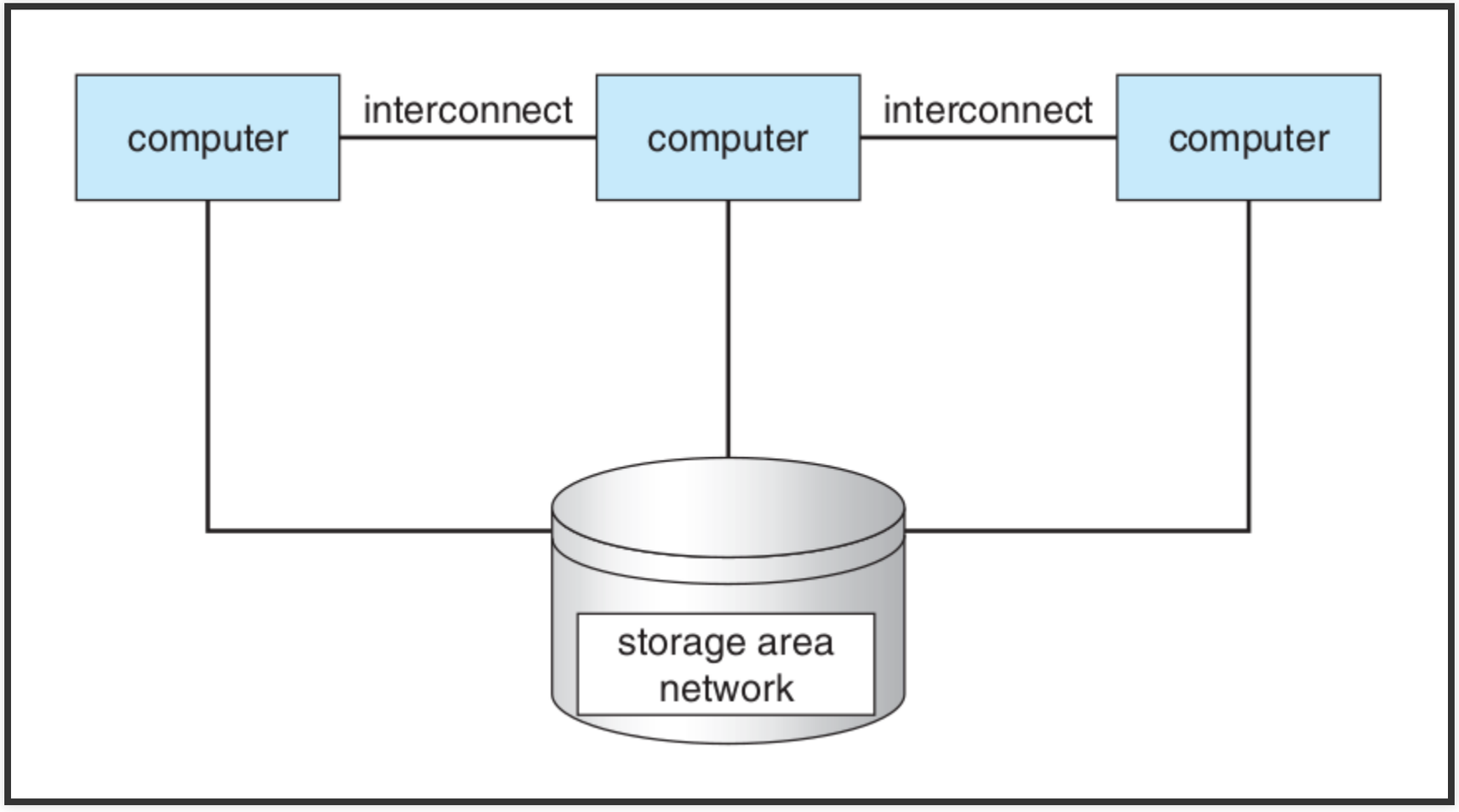
Clustered system → gathers together multiple CPUs.

Differ from the multiprocessor systems: Composed of two or more individual systems — joined together.

Usually used to provide high-availability service

- **Asymmetric clustering** has one machine in hot-standby mode
- **Symmetric clustering** has multiple nodes running applications, monitoring each other

CLUSTERED


computer
interconnect
computer
interconnect
computer
storage area
network

DEFINITIONS OF COMPONENTS

- **CPU**: Hardware that executes instructions.
- **Processor**: Physical chip that contains one or more CPUs.
- **Core**: Basic computation unit of the CPU.
- **Multicore**: Including multiple computing cores on the same CPU.
- **Multiprocessor**: Including multiple processors.

OPERATING-SYSTEM OPERATIONS

Internally, operating systems vary greatly, now we will study similarities

MULTIPROGRAMMING

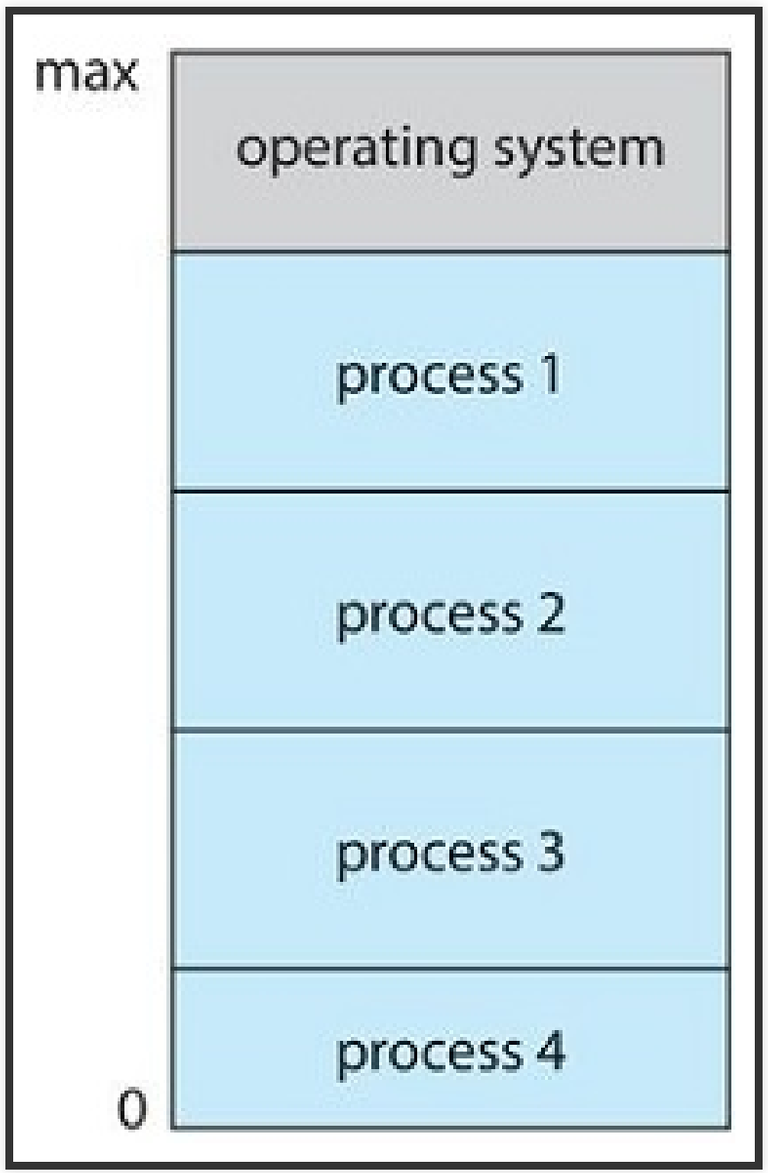
Multiprogramming needed for efficiency

- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- One job selected and run via job scheduling
- When it has to wait (for I/O for example), OS switches to another job

TIMESHARING

Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing

MEMORY LAYOUT



OPERATING-SYSTEM OPERATIONS

A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

DUAL-MODE AND MULTIMODE OPERATION

Should distinguish between the execution of operating-system code and user-defined code

Need two separate modes of operation:

- **User mode**
- **Kernel mode** (supervisor mode, system mode, or privileged mode)

MODE BIT

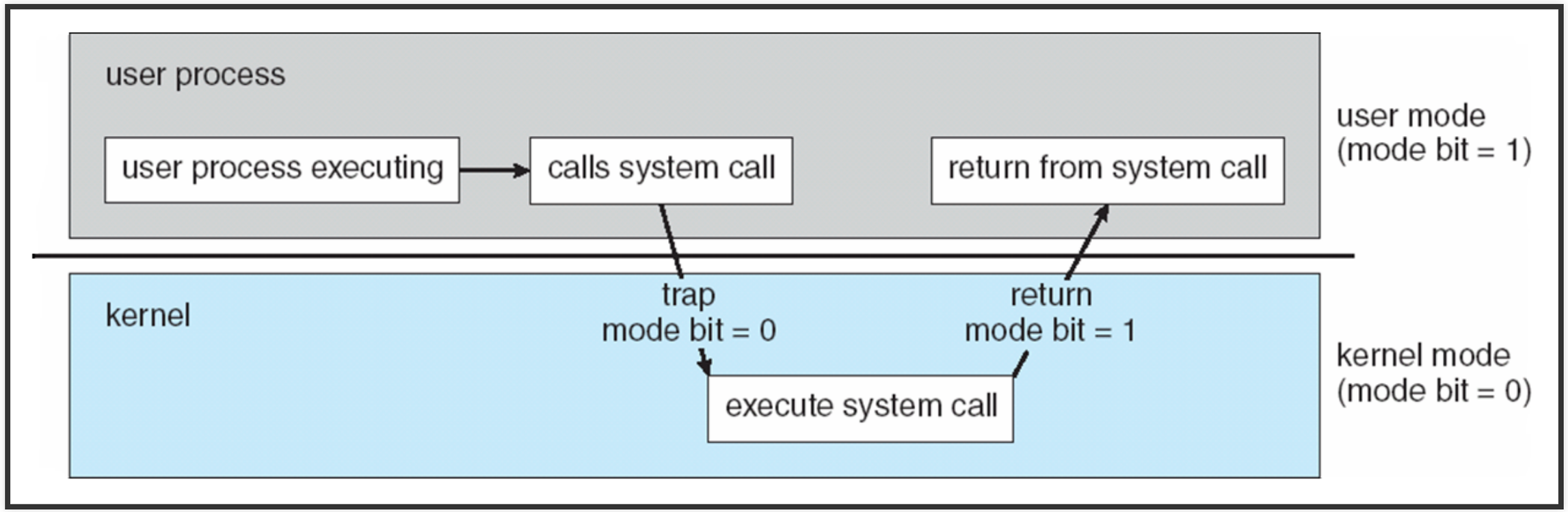
The **mode bit** → hardware supported bit - indicate current mode

- kernel (0)
- user (1).

Some instructions designated as **privileged**, only executable in kernel mode

System call changes mode to kernel, return from call resets it to user

USER AND KERNEL MODE



EXTEND THE PRINCIPLE

Easily extended beyond 2 modes

- Intel processors have four separate protection rings
  - Ring 0 is kernel mode
  - Ring 1+2 in practice rarely used
  - Ring 3 User mode

Virtualization has separete support to tell when VMM is in control of the system

USER AND KERNEL MODE

MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode.

Microsoft Windows 7/10, as well as Unix and Linux—take advantage of this dual-mode feature and provide greater protection for the operating system.

RESOURCE MANAGEMENT

OS Must manage the resources

- CPU
- Memory Space
- File-storage space
- I/O Devices

PROCESS VS PROGRAM

- A **process** is a **program** in execution.
  - Unit of work within 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 reclaim of any reusable resources

PROCESSES

Single-threaded process has one **program counter** specifying location of next instruction to execute Process executes instructions sequentially, one at a time, until completion

Multi-threaded process has one program counter per thread

Typically system has many processes, some user, some operating system running concurrently on one or more CPUs

PROCESS MANAGEMENT

OS responsibilities

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

MEMORY MANAGEMENT

Necessary

All data in memory before and after processing

All instructions in memory in order to execute

MEMORY MANAGEMENT

OS responsibilities

- Keep track of which parts of memory are currently being used
  - Who is using them
- Deciding which processes (or parts of processes) and data to move into and out of memory
- Allocating and deallocating memory space as needed

STORAGE MANAGEMENT

OS provides uniform, logical view of information storage

Abstracts physical properties to logical storage unit - file

Each 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)

FILE-SYSTEM MANAGEMENT

Files usually organized into directories Access control on most systems to determine who can access what

FILE-SYSTEM MANAGEMENT

OS activities:

- Creating and deleting files and directories
- Primitives to manipulate files and dirs
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

MASS-STORAGE MANAGEMENT

OS activities

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

CACHING

Because caches have limited size, **cache management** is an important design problem.

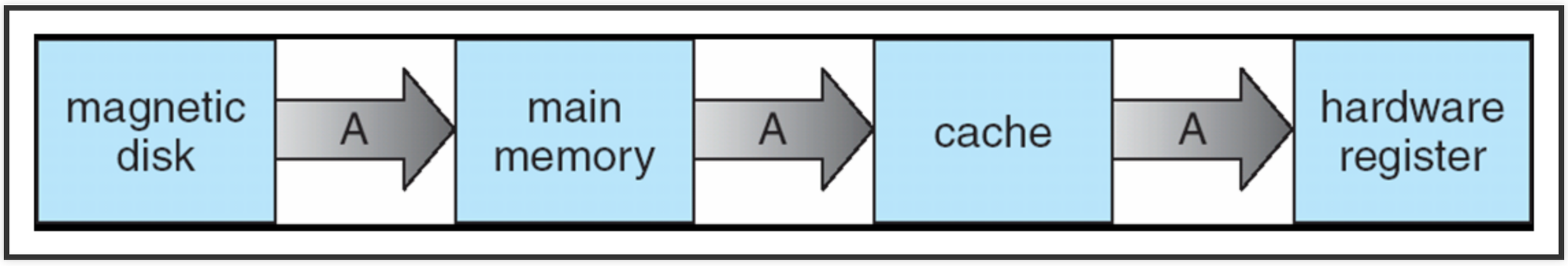
Main memory can be viewed as a fast cache for secondary storage

PERFORMANCE

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

DATA MIGRATION

Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



DATA MIGRATION

Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache

Distributed environment situation even more complex → Several copies of a datum can exist

I/O SYSTEM MANAGEMENT

One purpose of OS is to hide peculiarities of hardware devices from the user

I/O subsystem responsible for

PROTECTION AND SECURITY

PROTECTION VS SECURITY

Protection → any mechanism for controlling access of processes or users to resources defined by the OS

Security –> defense of the system against internal and external attacks

PROTECTION

Systems generally first distinguish among users, to determine who can do what

- User identities (user IDs, security IDs) include name and associated number, one per user
- User ID then associated with all files, processes of that user to determine access control
- Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

SECURITY

Huge range of **external attacks**

- denial-of-service
- worms
- viruses
- identity theft
- theft of service

VIRTUALIZATION

EMULATION

Emulation: Simulating computer hardware in software.

Allows software compiled for one architechturre to be run on another.

VIRTUALIZATION

Virtualization is a technology that allows operating systems to run as applications within other operating systems.

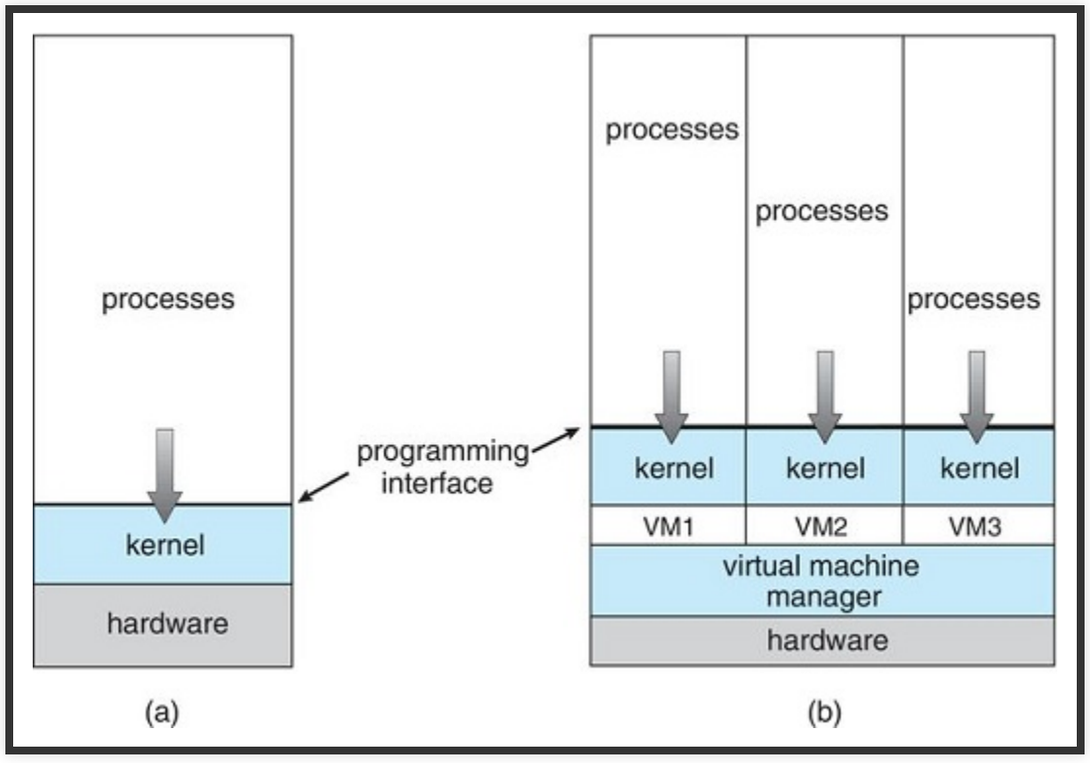
Emulation is used when the source CPU type is different from the target CPU type.

For example, an Apple laptop running Mac OS X on the x86 CPU can run a Windows guest to allow execution of Windows applications.

VIRTUALIZATION

Virtualization is a technology that allows us to abstract the hardware of a single computer (the CPU , memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer.

VIRTUALIZATION



VIRTUAL MACHINE MANAGER

The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

DISTRIBUTED SYSTEMS

DISTRIBUTED SYSTEMS

A **distributed system** is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains.

NETWORKS

- A **network**, in the simplest terms, is a communication path between two or more systems.
  - TCP/IP is the most common network protocol → Internet
- Wide-area network
- Local-area network (LAN) connects computers within a room, a building, or a campus.
- personal-area network (PAN) between a phone and a headset or a smartphone and a desktop computer.

NETWORK OPERATING SYSTEM

A **network operating system** is an operating system that provides features like

- File sharing across the network
- Communication scheme that allows different processes on different computers to exchange messages

> Acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers

DISTRIBUTED OPERATING SYSTEM

A **distributed operating system** provides a less autonomous environment.

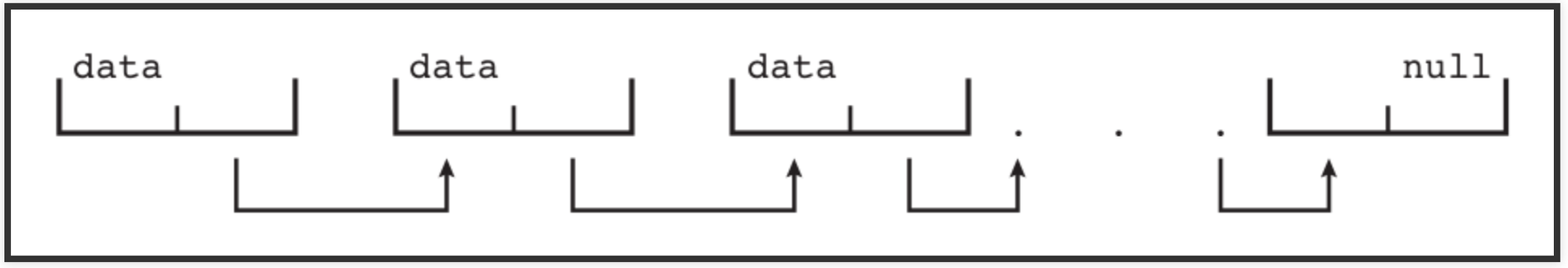
The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

KERNEL DATA STRUCTURES

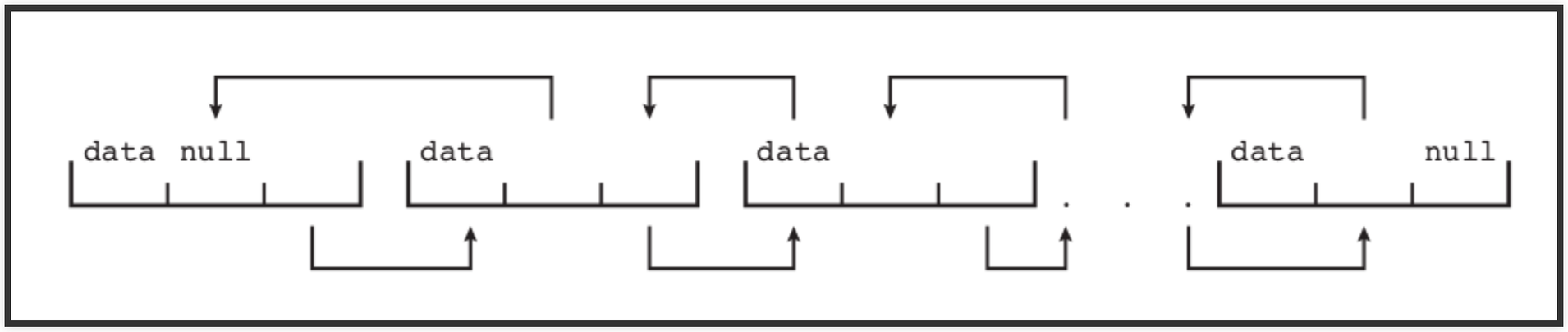
The way data are structured in the system.

Revisit Algorithm and Datastructure course for details

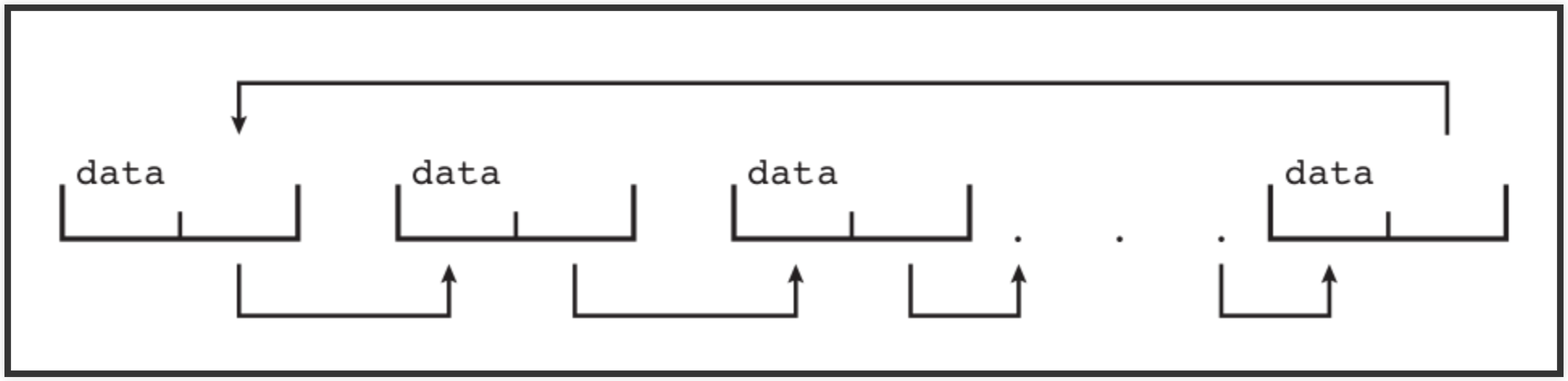
SINGLE LINKED LIST



DOUBLY LINKED LIST



CIRCULAR LINKED LIST



STACK

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**)

push and **pop** operations

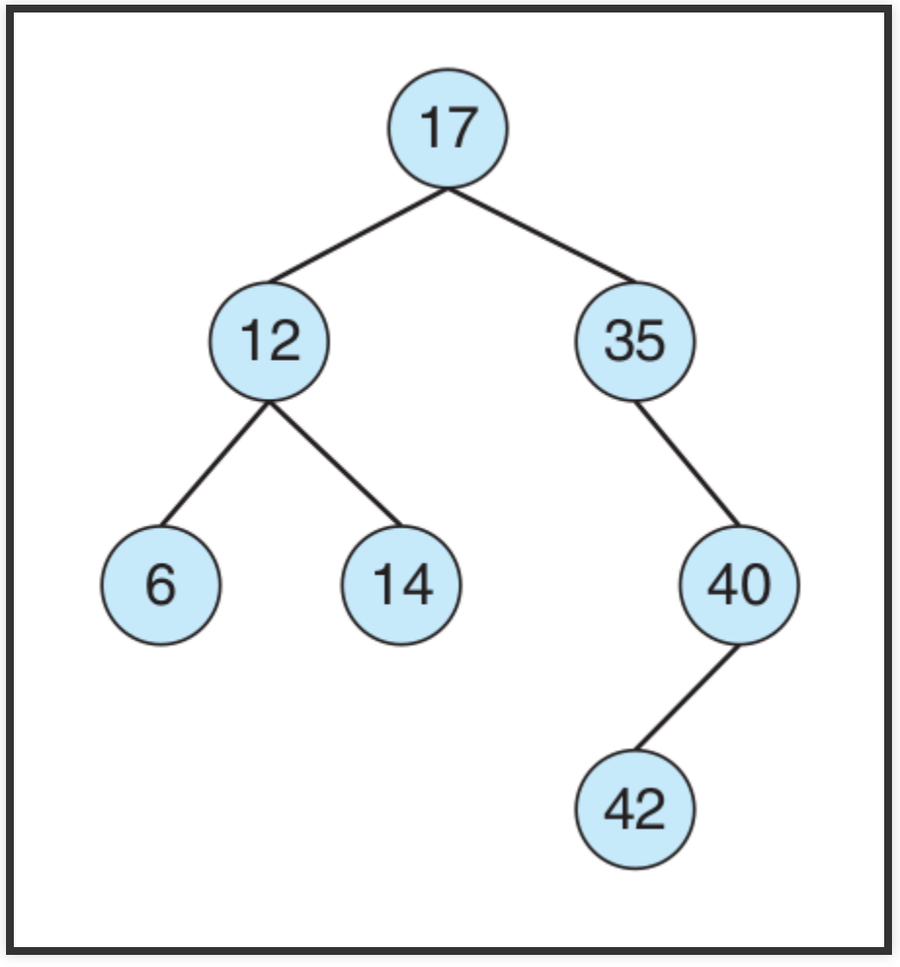
Used when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

QUEUES

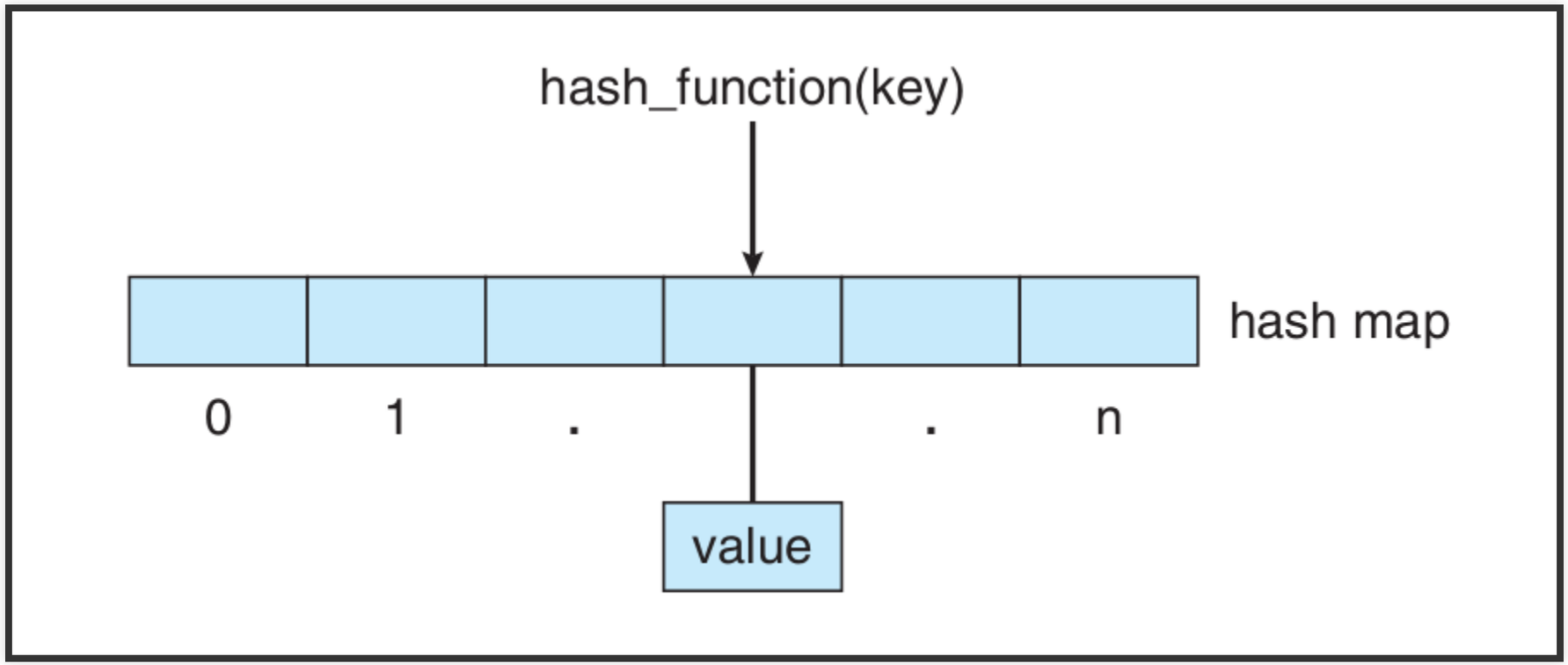
A **queue**, is a sequentially ordered data structure that uses the first in, first out (FIFO) principle

- Jobs that are sent to a printer are typically printed in the order in which they were submitted
- Queue with tasks that are waiting to be run on an available CPU

BINARY SEARCH TREES



HASH MAP



BITMAPS

A **bitmap** is a string of *n* binary digits that can be used to represent the status of *n* items.

Suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit

- 0 means that the resource is available,
- 1 indicates that it is unavailable

COMPUTING ENVIRONMENTS

TRADITIONAL COMPUTER

- Blurring over time
- Office environment
  - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
  - Now portals allowing networked and remote systems access to same resources
- Home networks
  - Used to be single system, then modems
  - Now firewalled, networked

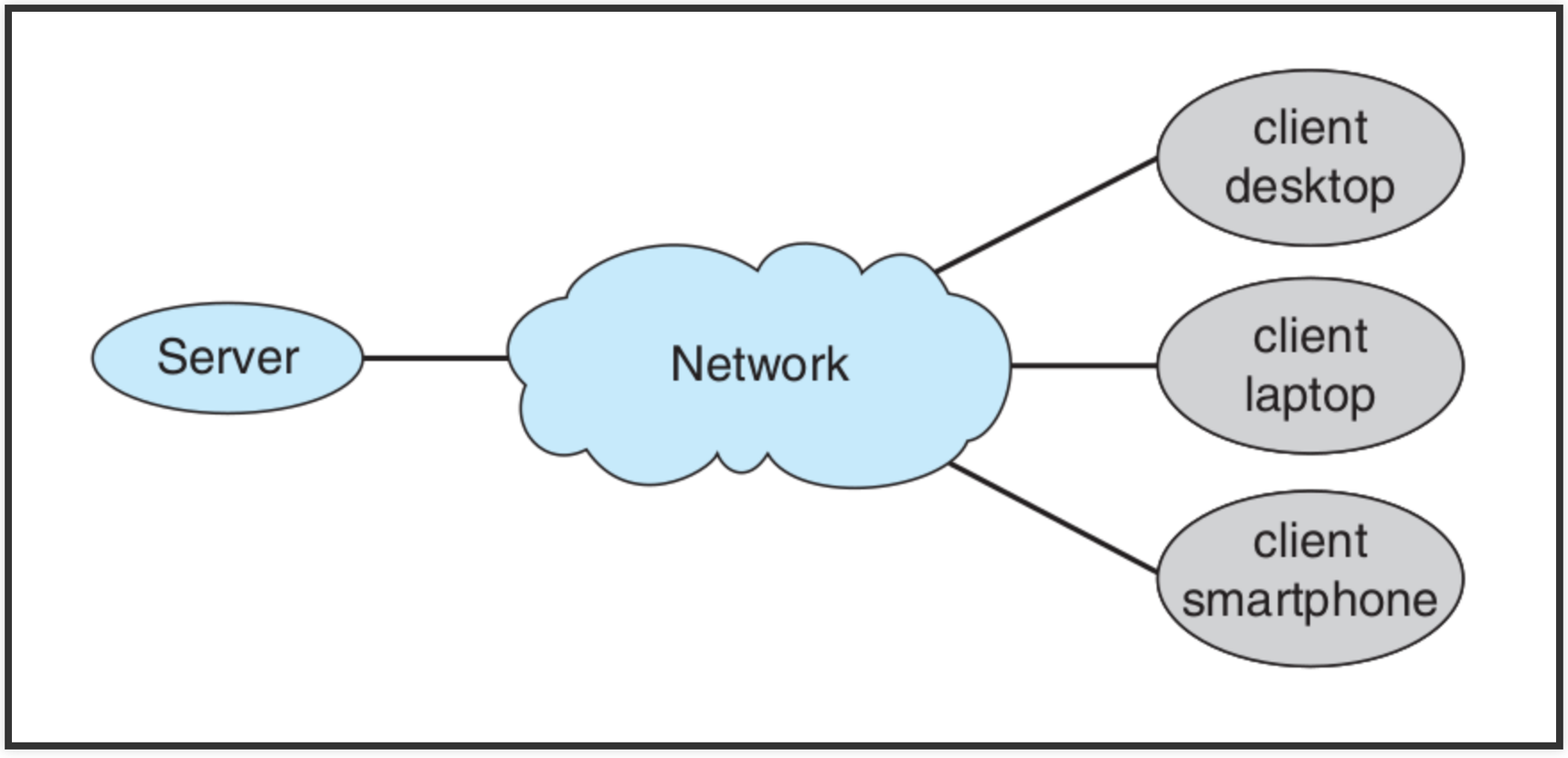
CLIENT-SERVER COMPUTING

Many systems now servers, responding to requests generated by clients

Compute-server provides an interface to client to request services (i.e. database)

File-server provides interface for clients to store and retrieve files

CLIENT SERVER



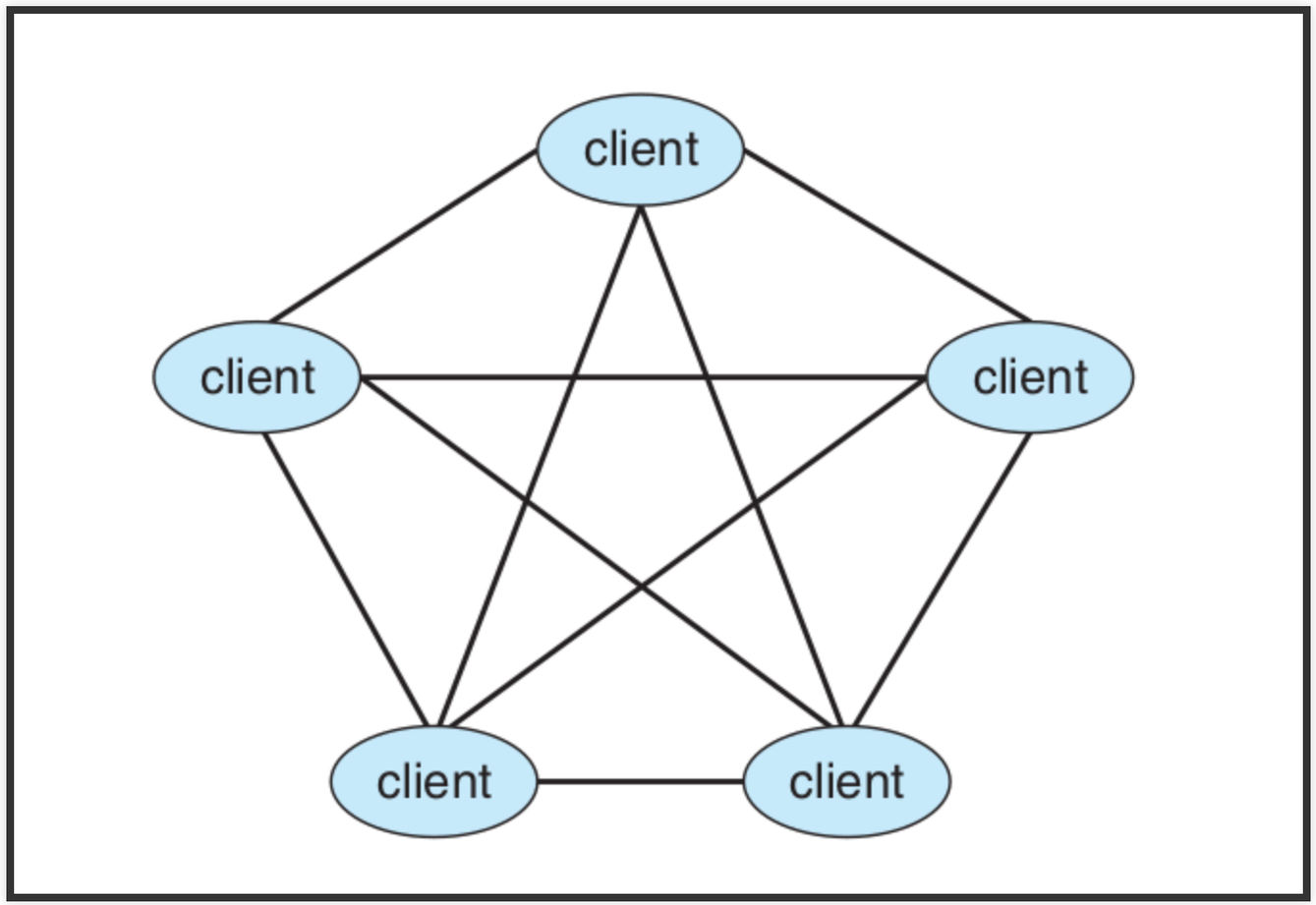
PEER-TO-PEER COMPUTING

Another model of distributed system

P2P does not distinguish clients and servers

- Instead all nodes are considered peers
- May each act as client, server or both
- Node must join P2P network
  - Registers its service with central lookup service on network, or
  - Broadcast request for service and respond to requests for service via **discovery protocol**

PEER-TO-PEER



MOBILE COMPUTING

Mobile computing refers to computing on handheld smartphones and tablet computers.

- Lightweight
- Portable

Tremendous growth the last years, completely dominated by IOS and Android

CLOUD COMPUTING

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network.

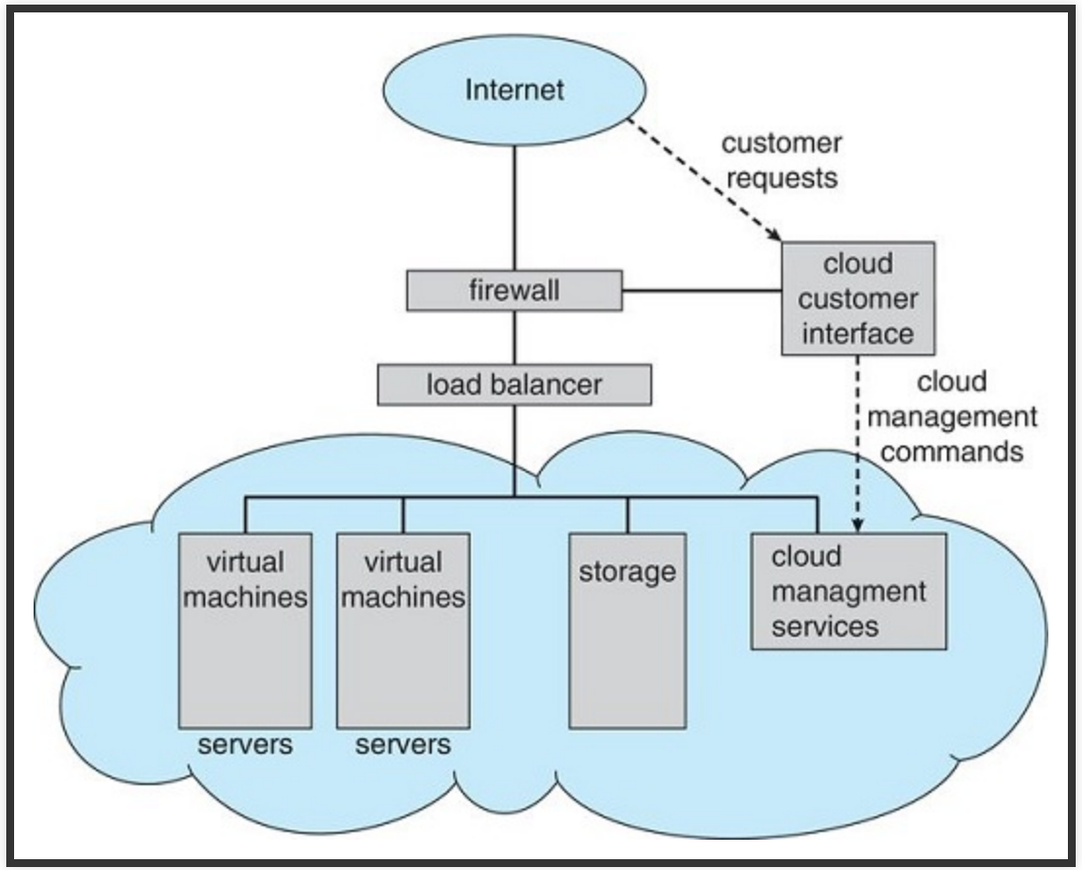
CLOUD COMPUTING TYPES

- Public cloud: available via the Internet to anyone willing to pay for the services
- Private cloud: a cloud run by a company for that company's own use
- Hybrid cloud — includes both public and private cloud components

CLOUD COMPUTING TYPES

- Software as a service (SaaS): one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (PaaS): a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (IaaS): servers or storage available over the Internet (for example, storage available for making backup copies of production data)

CLOUD COMPUTING



EMBEDDED SYSTEMS

- Most prevalent form of computers in existence
- Found everywhere! Car engines, manufacturing robots, optical drives, microwave ovens.
- Primitive, so OS provides limited features

REAL-TIME OPERATING SYSTEMS

- Embedded systems almost always run real-time operating systems
- used when rigid time requirements have been placed on the operation of a processor or the flow of data
- Processing must be done within the defined constraints, or the system will fail.
  - Brake system of a robotic arm - halts after craching into wall

OPEN-SOURCE OPERATING SYSTEMS

OPEN-SOURCE OPERATING SYSTEMS

Open-source operating systems are those available in source-code format rather than as compiled binary code.

- **Linux** is the most famous open-source operating system
- **Microsoft Windows** is a well-known example of the opposite closed-source approach
- **Apple's Mac OS X and iOS** comprise a hybrid approach.

OPEN SOURCE HISTORY

- 1950s: a great deal of software was available in open-source format.
- Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers.
- 1978: BSD UNIX started as a derivative of AT&T's UNIX .
- 1994: Due to lawsuit by AT&T, eventually a fully functional, open-source version, 4.4BSD-lite, was released.

OPEN SOURCE HISTORY

Copyrighted material

- copy protection/digital rights management (DRM) wouldn't be effective if source code were published.
- 1983: Richard Stallman started the GNU project to create free, open-source, UNIX - compatible OS.
- 1985: GNU Manifesto: All software should be free and open-sourced. Also formed the Free Software Foundation (FSF)

OPEN SOURCE HISTORY

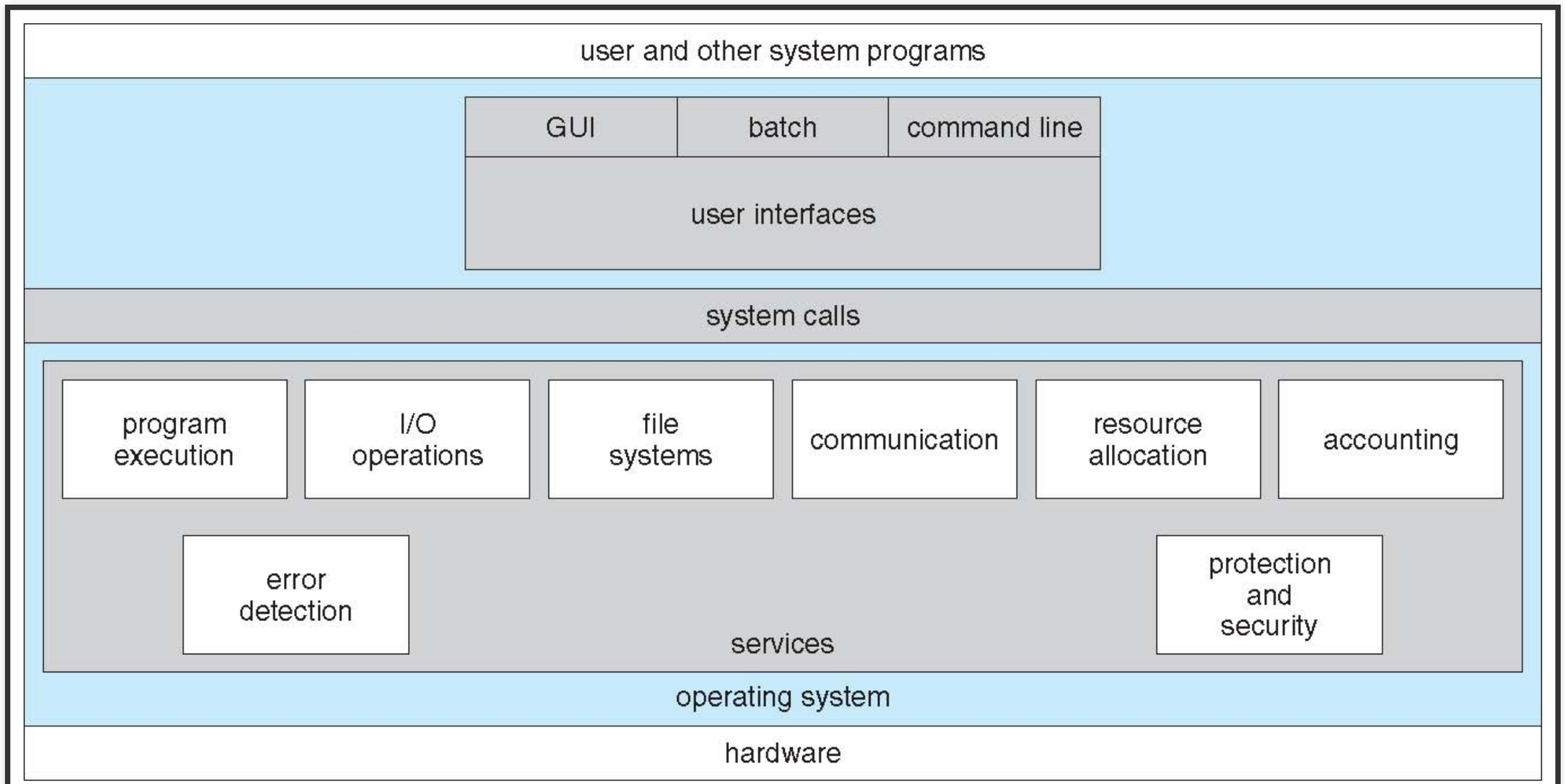
- 1991: Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide.
- 2005: Sun open-sourced most of the Solaris code as the OpenSolaris project. (Closed since 1991)

OPEN SOURCE HISTORY

CHAPTER 2 - OPERATING SYSTEM STRUCTURES

OPERATING-SYSTEM SERVICES

OPERATING-SYSTEM SERVICES



USER INTERFACE

Almost all operating systems have a user interface (UI). Varies between

- Command-Line (CLI)
- Graphics User Interface (GUI)
- Batch

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.

Programs need to

- read and write files and directories
- create and delete them
- search them
- list file Information
- permission management.

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

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

LOGGING

Accounting - To keep track of which users use how much and what kinds of computer resources

Just for usage statistics

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

USER AND OPERATING-SYSTEM INTERFACE

CLI

CLI or command interpreter allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- 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

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
- Invented at Xerox PARC

CLI & GUI

Many systems now include both CLI and GUI interfaces

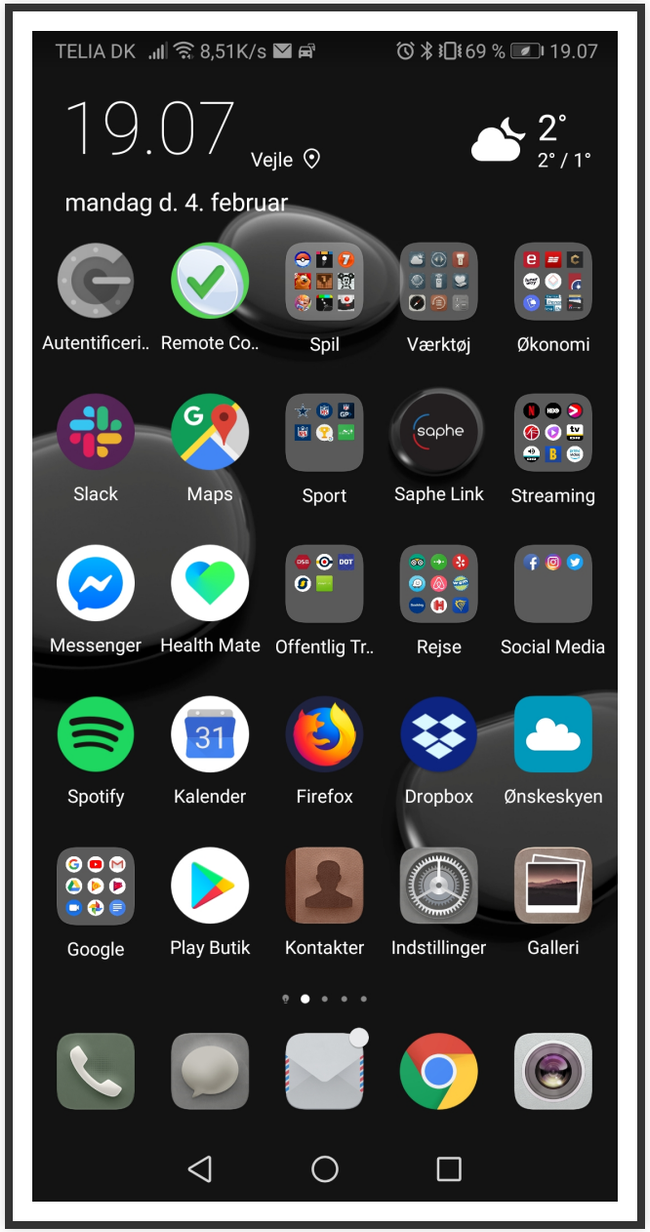
- Microsoft Windows is GUI with CLI “command” shell
- Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
- Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

TOUCHSCREEN INTERFACES

Touchscreen devices require new interfaces

- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry

TOUCHSCREEN INTERFACES



CHOICE OF INTERFACE

System administrators who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface.

On some systems, only a subset of system functions is available via the GUI

Shell scripts are very common on systems that are command-line oriented

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

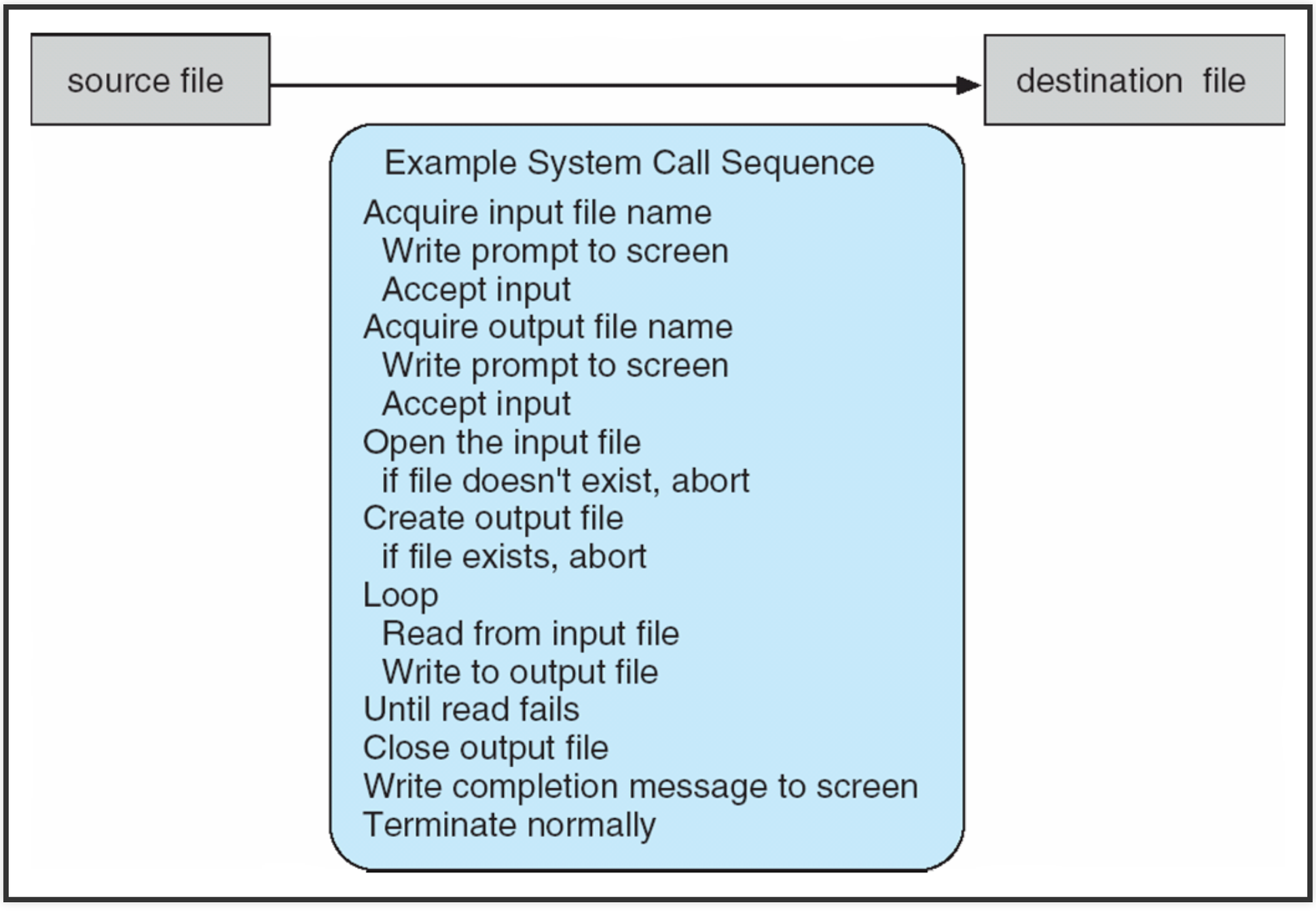
SYSTEM CALLS APIS

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)
- Java API for the Java virtual machine (JVM)

Why use APIs rather than system calls?

EXAMPLE OF SYSTEM CALLS



EXAMPLE OF STD 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)
```

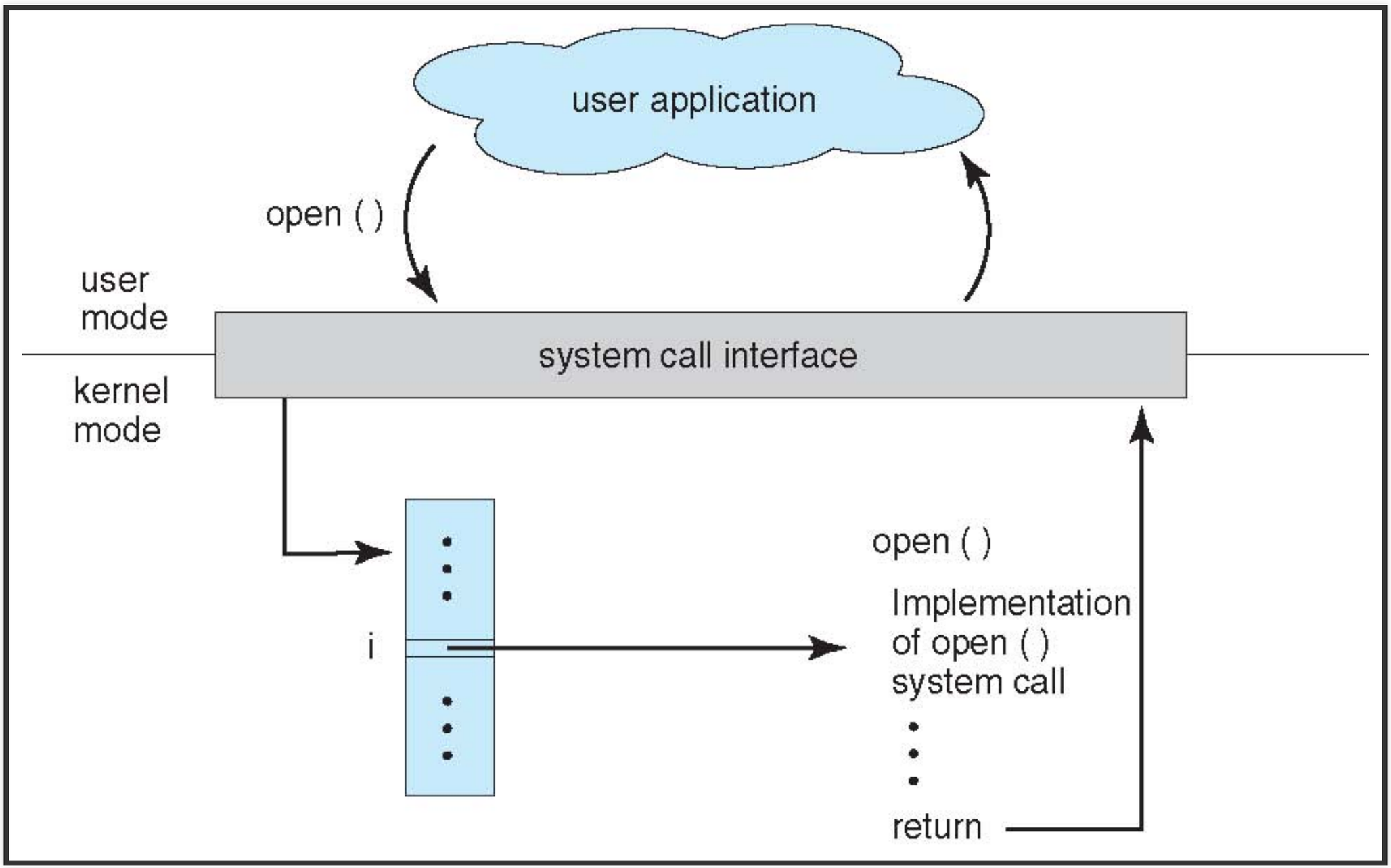
return value — function name — parameters

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 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 CALLS



SYSTEM CALLS

Most programming languages have run-time support

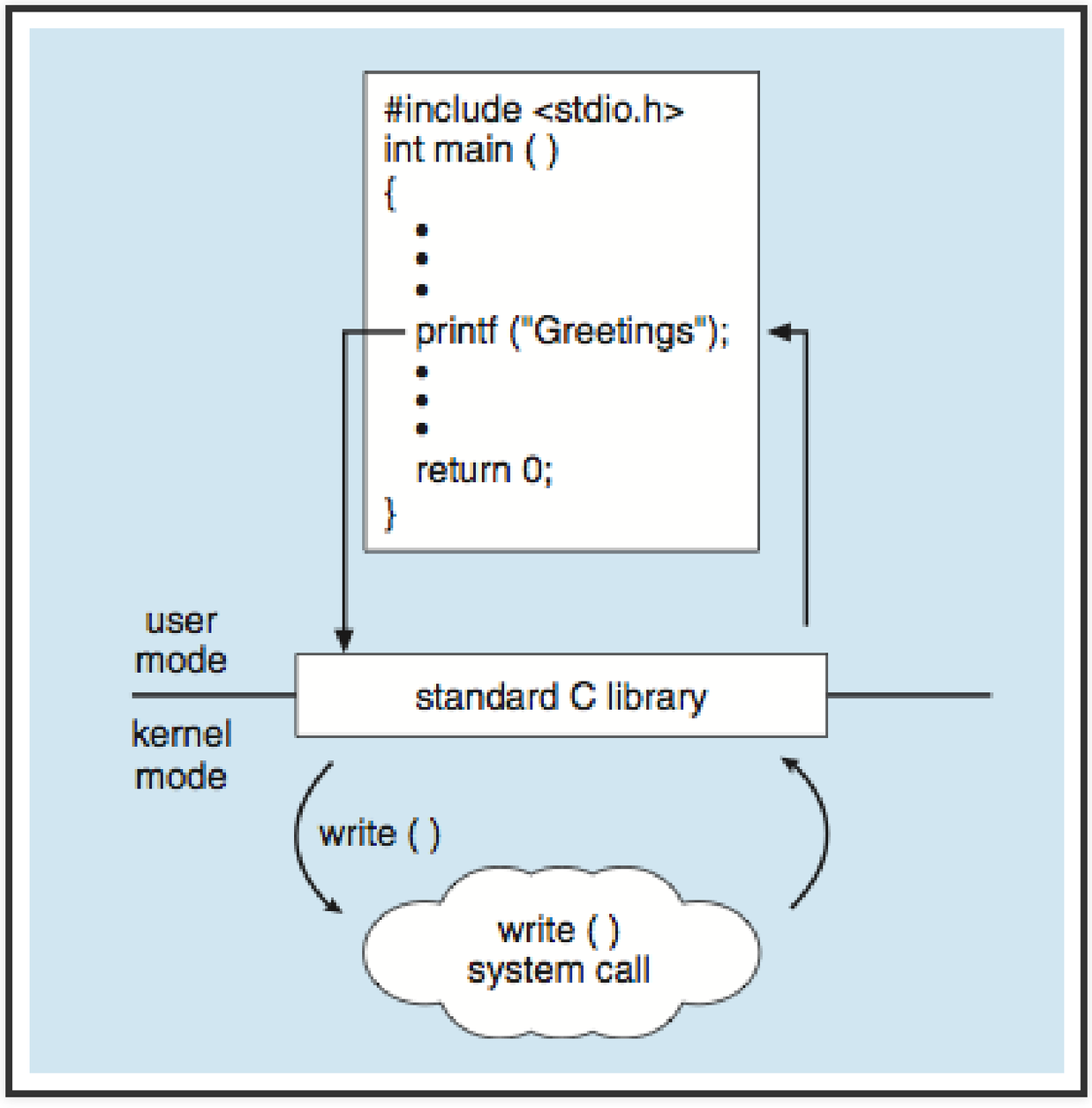
- Set of functions built into libraries included with a compiler
  - provides a **system-call interface** → link to system calls made available by the operating system.
- Typically, a number is associated with each system call
  - System-call interface maintains a table indexed according to these numbers.

SYSTEM CALL APIS

The caller need know nothing about how the system call is implemented or what it does during execution.

Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.

STANDARD LIBRARY

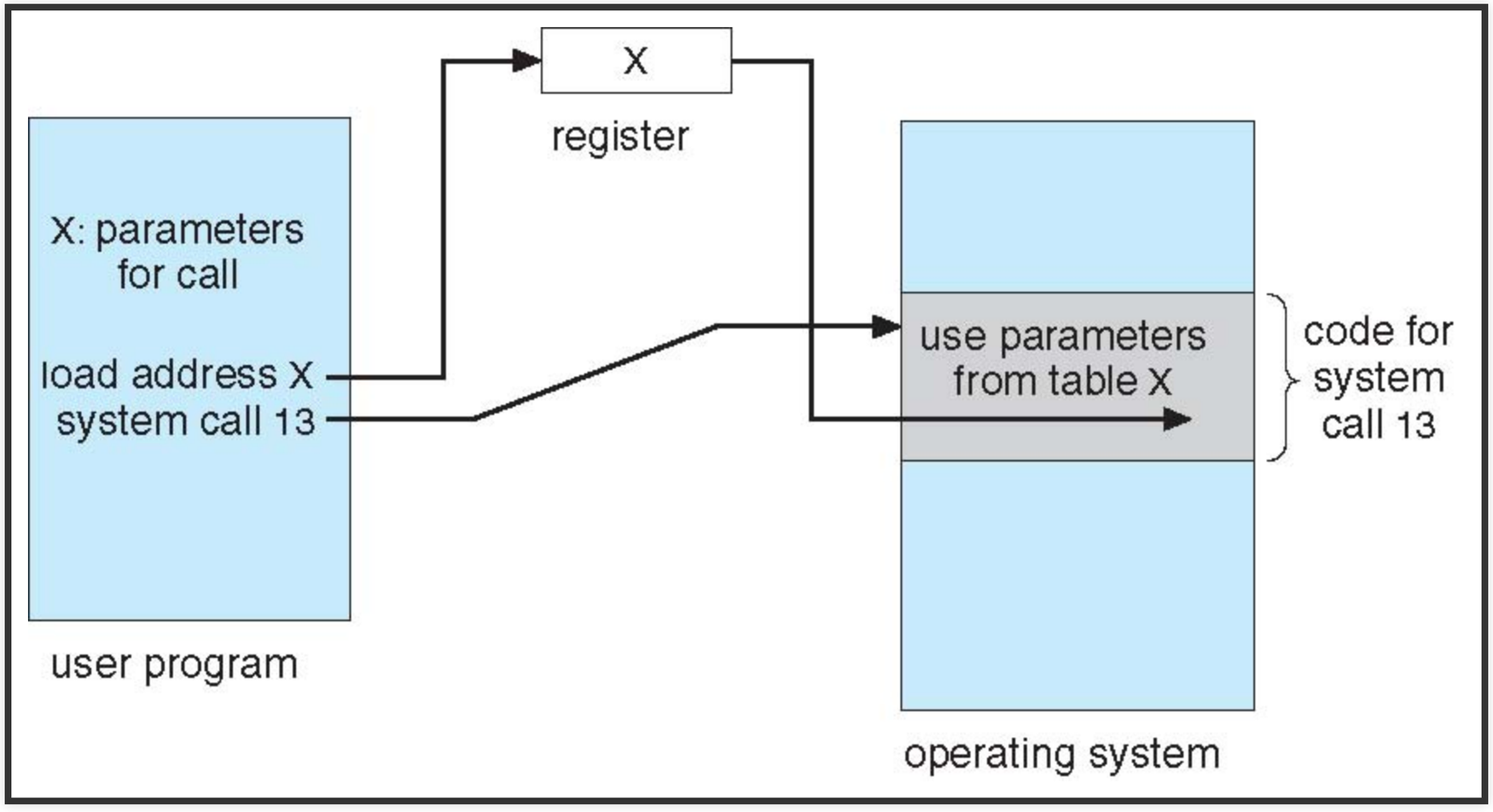


PASSING PARAMETERS

3 general methods to pass parameters to the OS.

- In registers → Simplest
- stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register → Linux & Solaris (if >5 ow registers)
- placed, or pushed, onto the stack by the program and popped off the stack by the operating system → do not limit the number or length of parameters

SYSTEM CALLS



TYPES OF SYSTEM CALLS

GROUPS OF SYSTEM CALLS

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications
- Protection.

SYSTEM CALLS EXAMPLES

	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()

PROCESS CONTROL

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

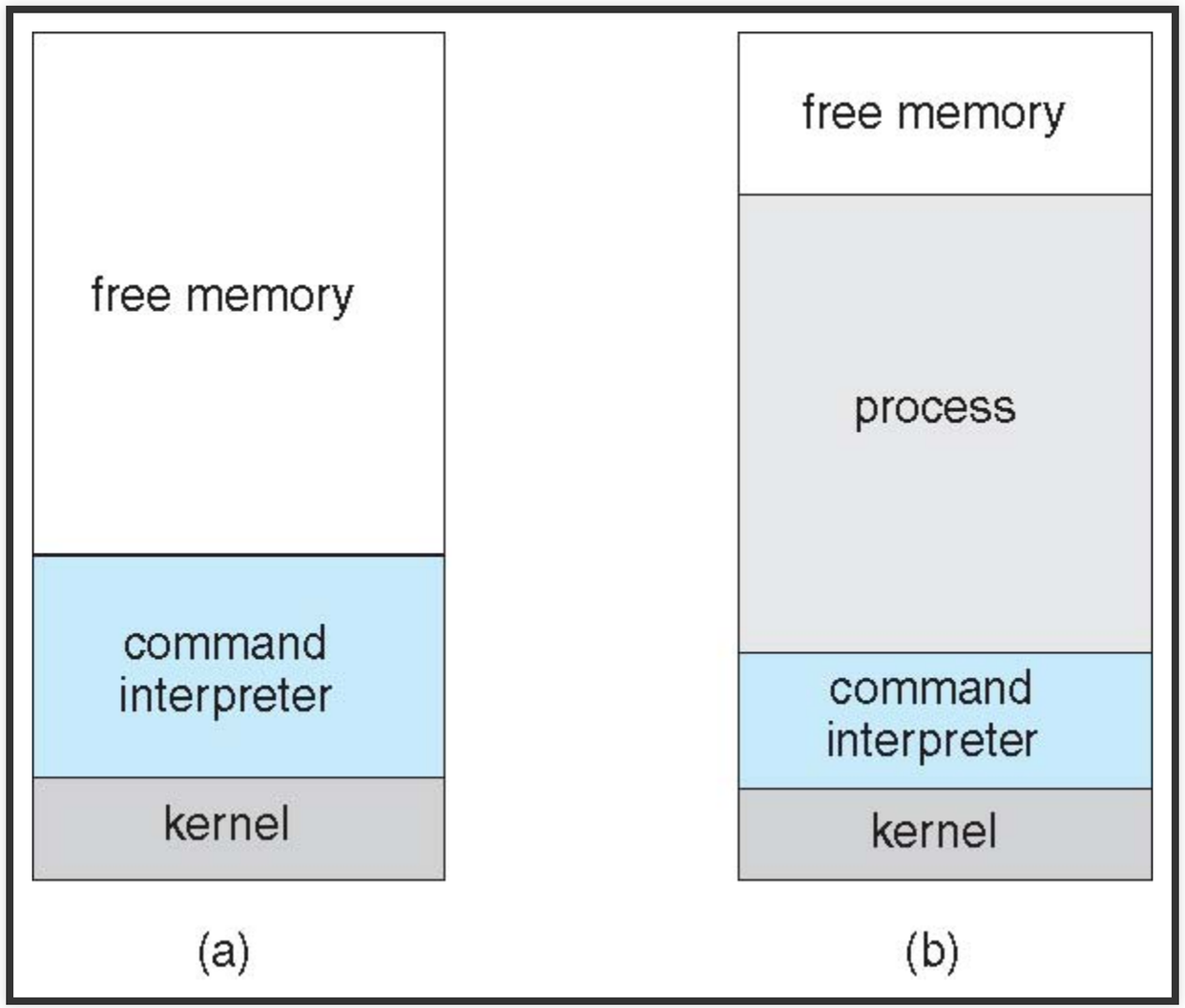
PROCESS CONTROL

- Dump memory if error
- **Debugger** for determining **bugs**, **single step** execution

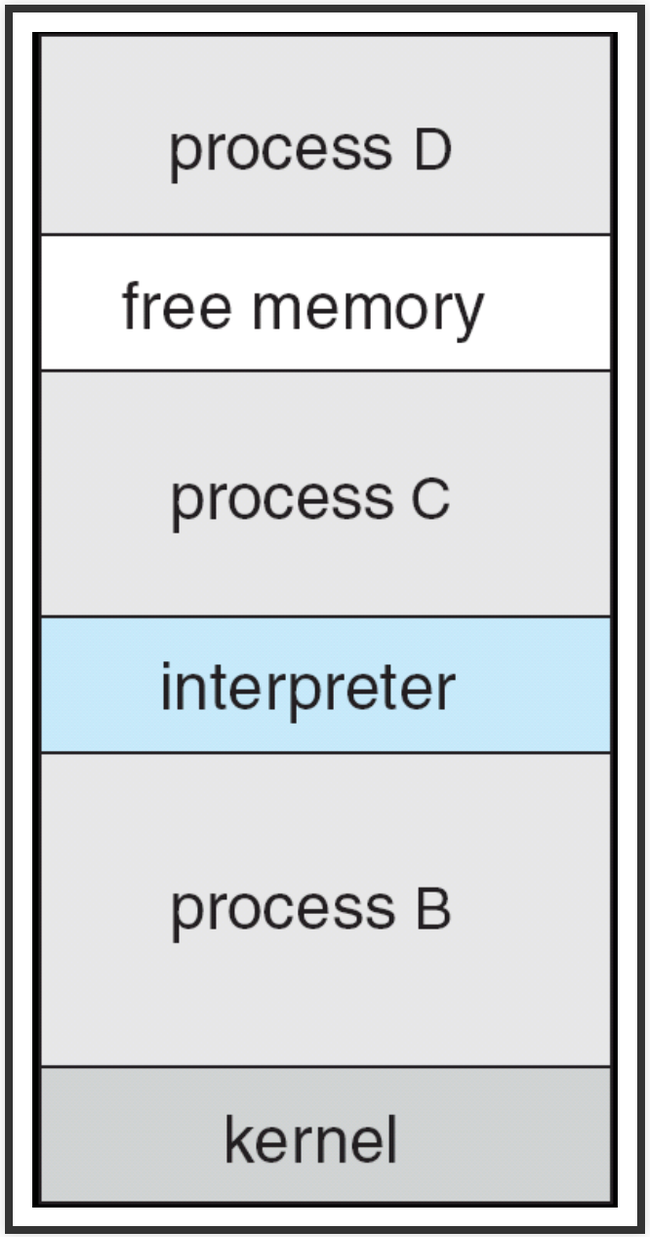
To ensure integrity between shared data

- Locks for managing access

MS-DOS


free memory
command interpreter
kernel
(a)
free memory
process
command interpreter
kernel
(b)

FREEBSD


process D
free memory
process C
interpreter
process B
kernel

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

INFORMATION MAINTENANCE

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes
- single step → trap executed after each command → caught by debugger

COMMUNICATION

- 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

PROTECTION

- Control access to resources
- Get and set permissions
- Allow and deny user access

SYSTEM SERVICES

SYSTEM SERVICES

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

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

Provide a convenient environment for program development and execution

Some of them are simply user interfaces to system calls; others are considerably more complex

SYSTEM PROGRAMS

- File management
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

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

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

BACKGROUND SERVICES

- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as services, subsystems, daemons

APPLICATION PROGRAMS

- Don’t pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke

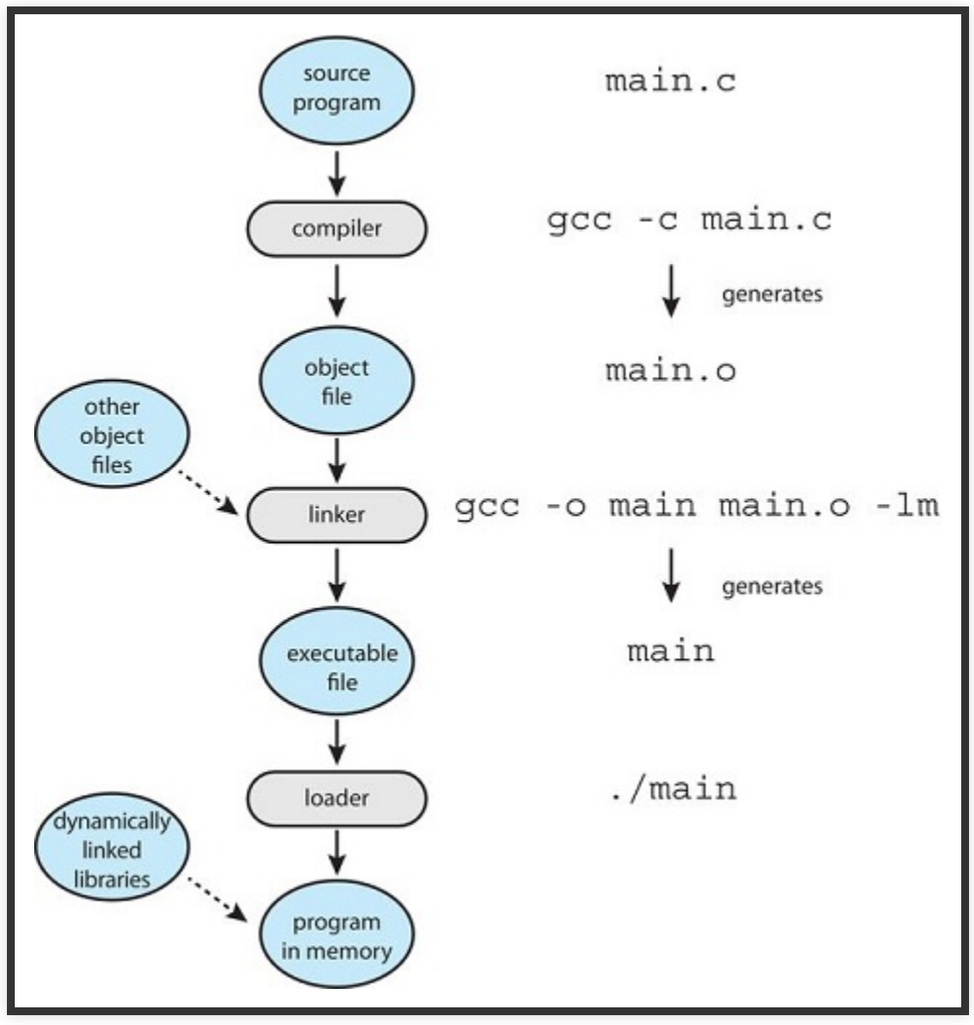
LINKERS AND LOADERS

LINKERS

- 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

LOADERS

THE ROLE OF THE LINKER AND LOADER



WHY APPLICATIONS ARE OPERATING SYSTEM SPECIFIC

OPERATING SYSTEM SPECIFIC

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc

LESS OPERATING SYSTEM SPECIFIC

- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each

OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

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

Affected by choice of hardware, type of system

DESIGN GOALS

Start by defining goals and specifications

- **User goals**
  - should be convenient to use, easy to learn, reliable, safe, and fast
- **System goals**
  - should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

MECHANISMS AND POLICIES

Important to separate

- **Policy**: *What* will be done?
- **Mechanism**: *How* to do it?

The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

IMPLEMENTATION

Much variation

- Early OSes in assembly language
- Then system programming languages like Algol, PL/1
- Now C, C++

IMPLEMENTATION

Actually usually a mix of languages

- Lowest levels in assembly
- Main body in C
- Systems 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

OPERATING-SYSTEM STRUCTURE

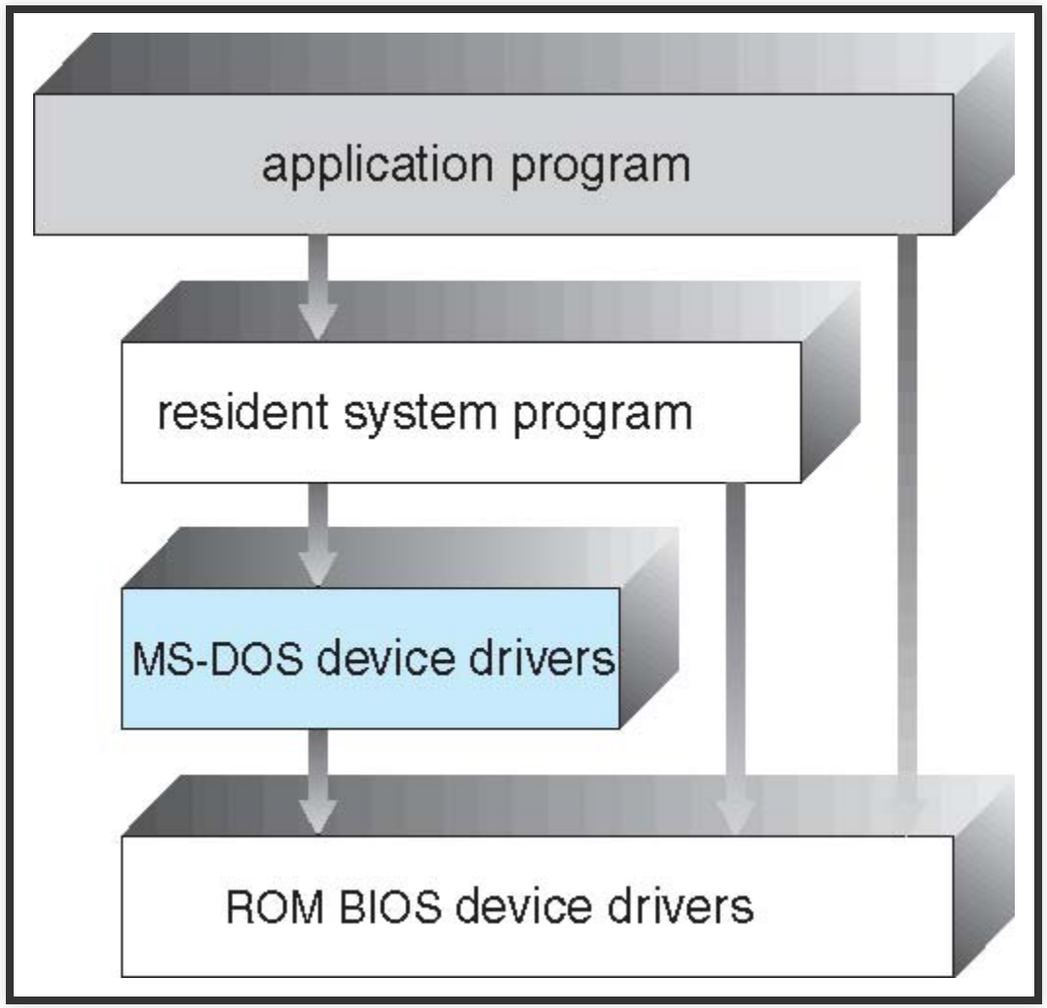
General-purpose OS is very large program

Various ways to structure one as follows

MONOLITHIC STRUCTURE

Put all of the functionality of the kernel into a single, static binary file that runs in a single address space.

MS-DOS LAYER STRUCTURE

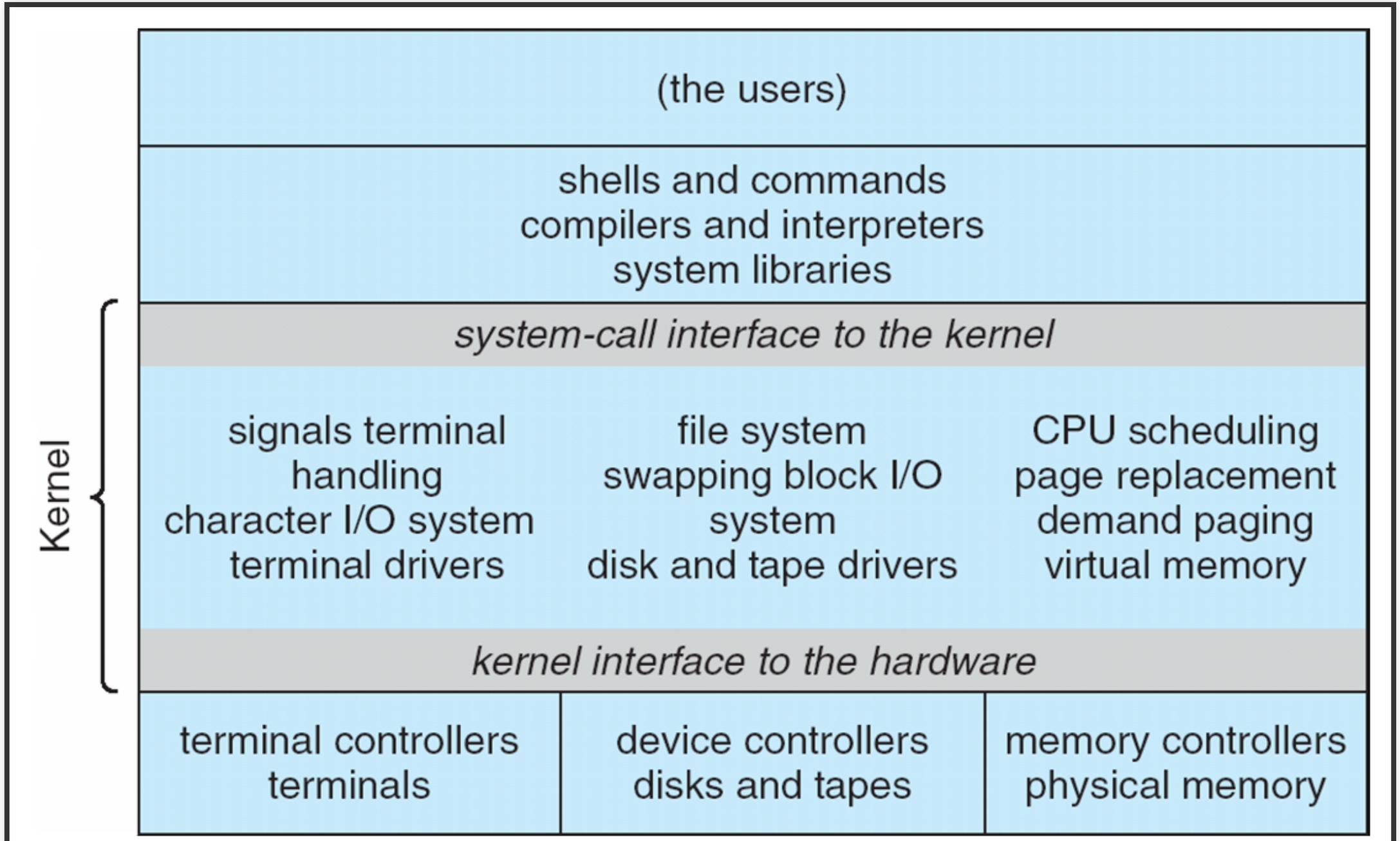


UNIX SYSTEM STRUCTURE

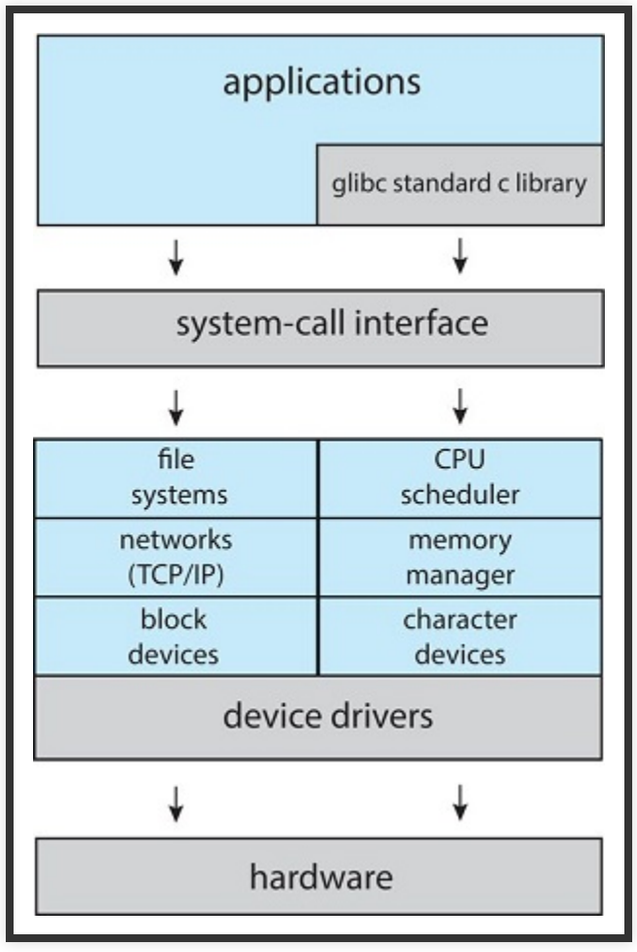
UNIX – limited by hardware functionality, the original UNIX OS had limited structuring. Consists of two separable parts

1. Systems programs
2. 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


(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory
Kernel

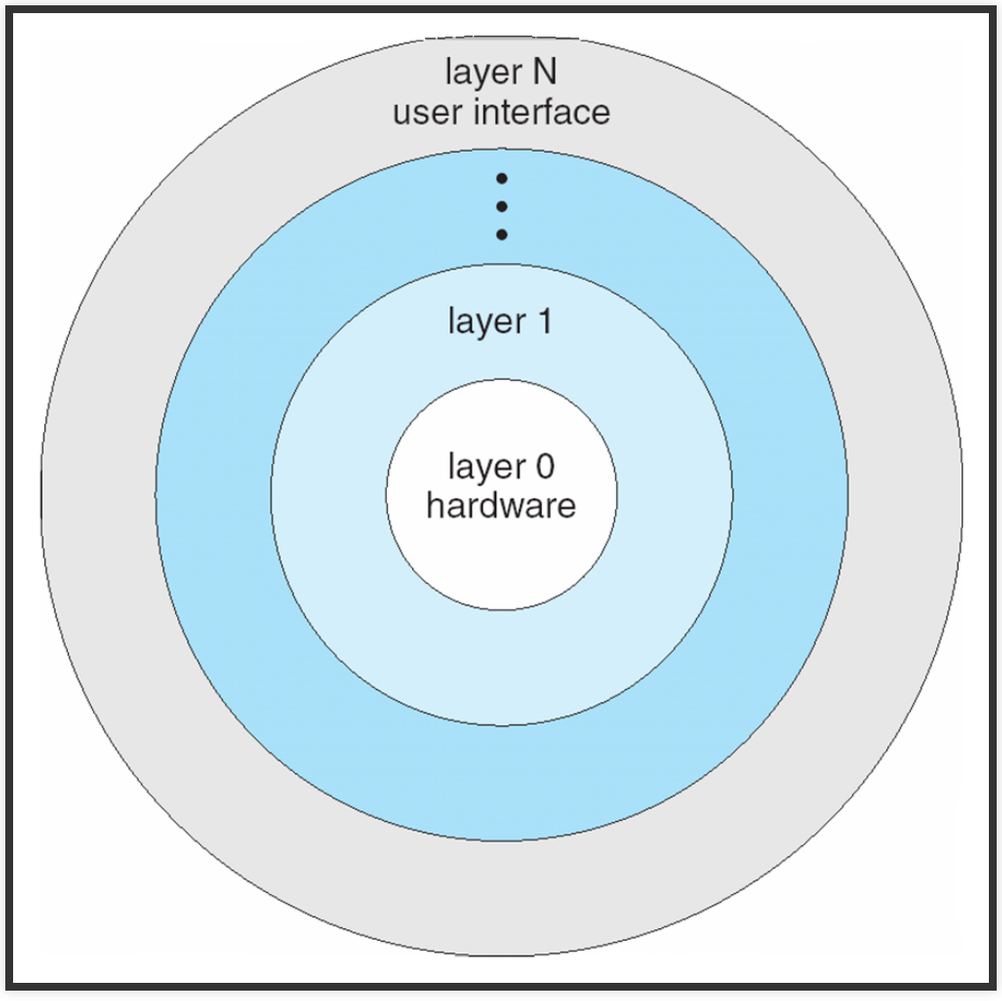
LINUX SYSTEM STRUCTURE



MONOLITHIC APPROACH

- Monolithic ⇒ Tightly Coupled system
- Would like a loosely coupled system

LAYERED APPROACH



MICROKERNELS

Moves as much from the kernel into user space

Mach example of **microkernel**

Mac OS X kernel (*Darwin*) partly based on Mach

Communication takes place between user modules using message passing

MICROKERNELS

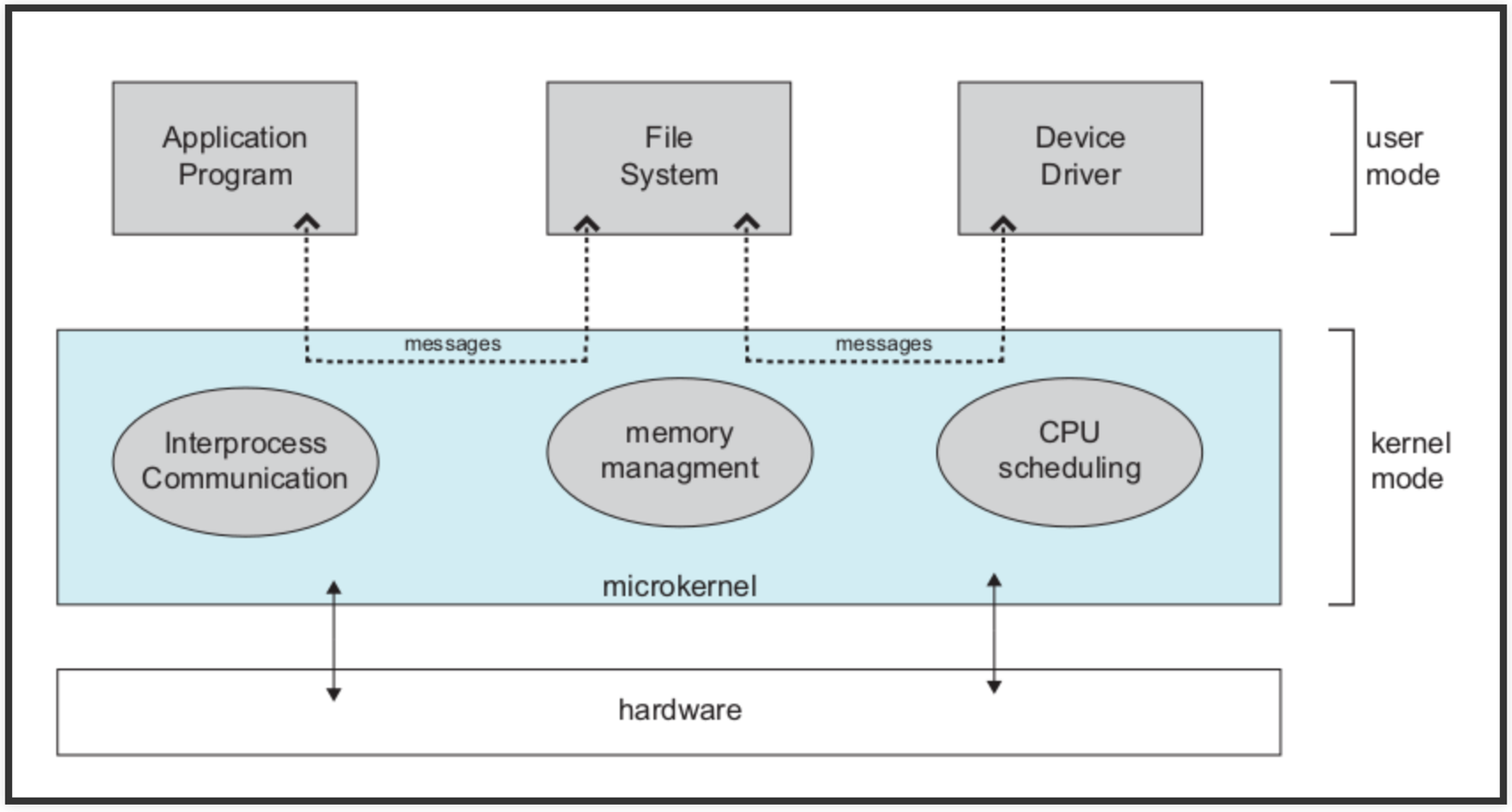
Benefits

- 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

MICROKERNELS



MODULES

Most modern operating systems implement **loadable kernel modules**

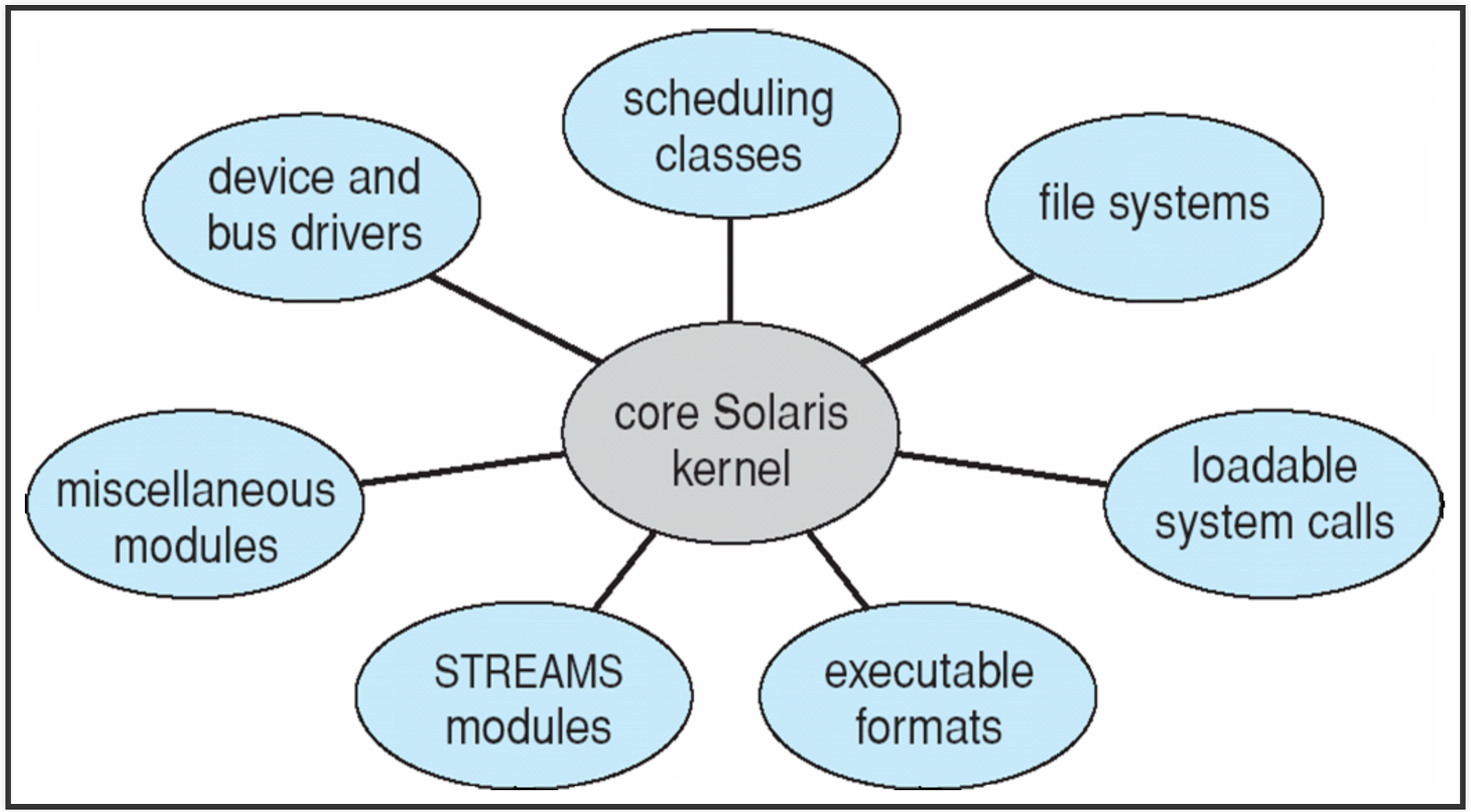
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

MODULES

Overall, similar to layers but with more flexibility, as all modules can call other modules

Used in modern UNIX (Solaris), Linux, Mac OS and Windows

SOLARIS LOADABLE MODULES



HYBRID SYSTEMS

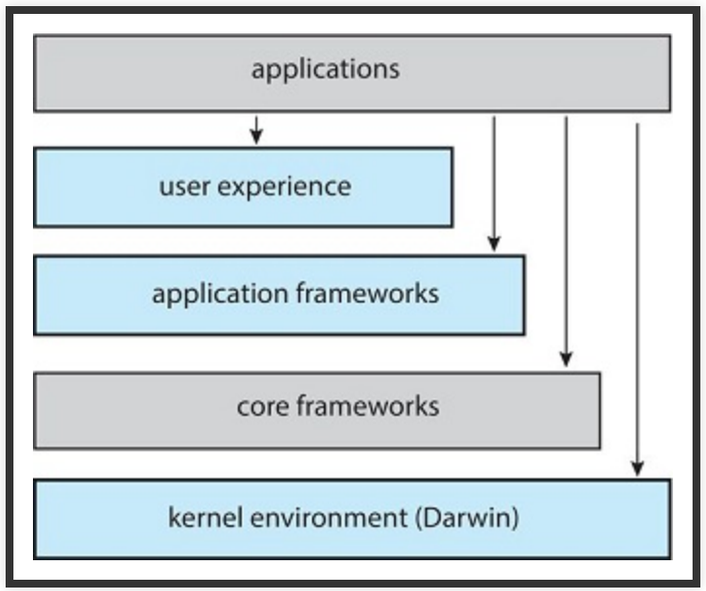
Most modern operating systems actually not one pure model

Hybrid combines multiple approaches to address performance, security, usability needs

HYBRID SYSTEMS

- 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

MAC OS X STRUCTURE

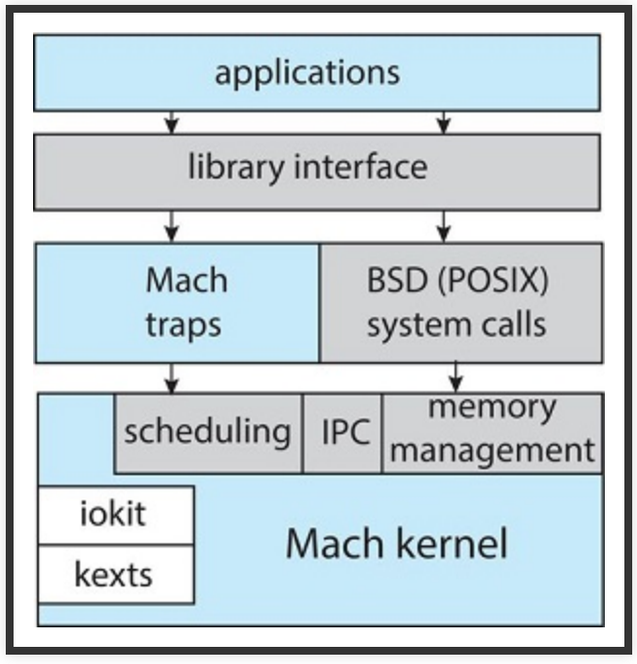


iOS

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

IOS


applications
library interface
Mach traps
BSD (POSIX) system calls
scheduling
IPC
memory management
iokit
kexts
Mach kernel

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

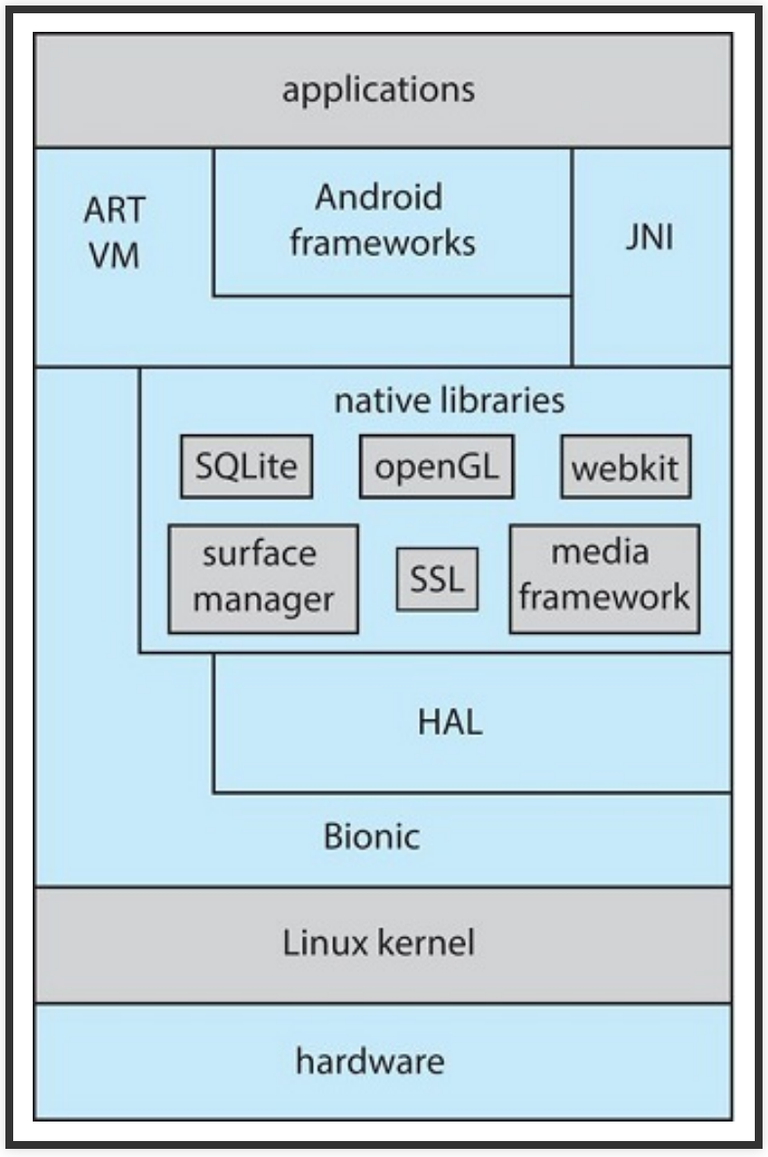
ANDROID

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 ARCHITECHTURE



BUILDING AND BOOTING AN OPERATING SYSTEM

OPERATING-SYSTEM GENERATION

Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site

OPERATING-SYSTEM GENERATION

System Generation (**SYSGEN**) program obtains information concerning the specific configuration of the hardware system

- Used to build system-specific compiled kernel or system-tuned
- Can general more efficient code than one general kernel

OPERATING-SYSTEM GENERATION

The kinds of information that must be determined

- CPU is used?
  - What options (extended instruction sets, floating-point arithmetic, etc.) are installed?
- Partitions on disk
- Boot partition formatting
- How much memory is available
- What devices are available

OPERATING-SYSTEM GENERATION

Building an OS from scratch

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

BUILDING YOUR OWN LINUX KERNEL

Assignment 2 will show you

SYSTEM BOOT

Bootstrap program or **bootstrap loader** locates the kernel

I.e. the instruction register is loaded with a predefined memory location, and execution starts there.

This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup.

BOOTSTRAP CODE

Changing the bootstrap code requires changing the ROM hardware chips.

Some systems resolve this problem by using erasable programmable read-only memory (**EPROM**), which is read-only except when explicitly given a command to become writable.

FIRMWARE

All forms of ROM are known as **firmware**, since their characteristics fall between hardware and software.

A disk that has a **boot partition** is called a **boot disk** or **system disk**.

EPROM

http://thenextweb.com/insider/2016/02/01/running-a-single-delete-command-can-permanently-brick-laptops-from-inside-linux/

github issue and discussion

OPERATING-SYSTEM DEBUGGING

DEBUGGING

Debugging: The activity of finding and fixing errors in a system, both in hardware and in software

FAILURE ANALYSIS

OSes 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

FAILURE ANALYSIS

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*

— Kernighan's Law

PERFORMANCE TUNING

Improve performance by removing bottlenecks

OS must provide means of computing and displaying measures of system behavior

PERFORMANCE TUNING - LINUX TOOLS

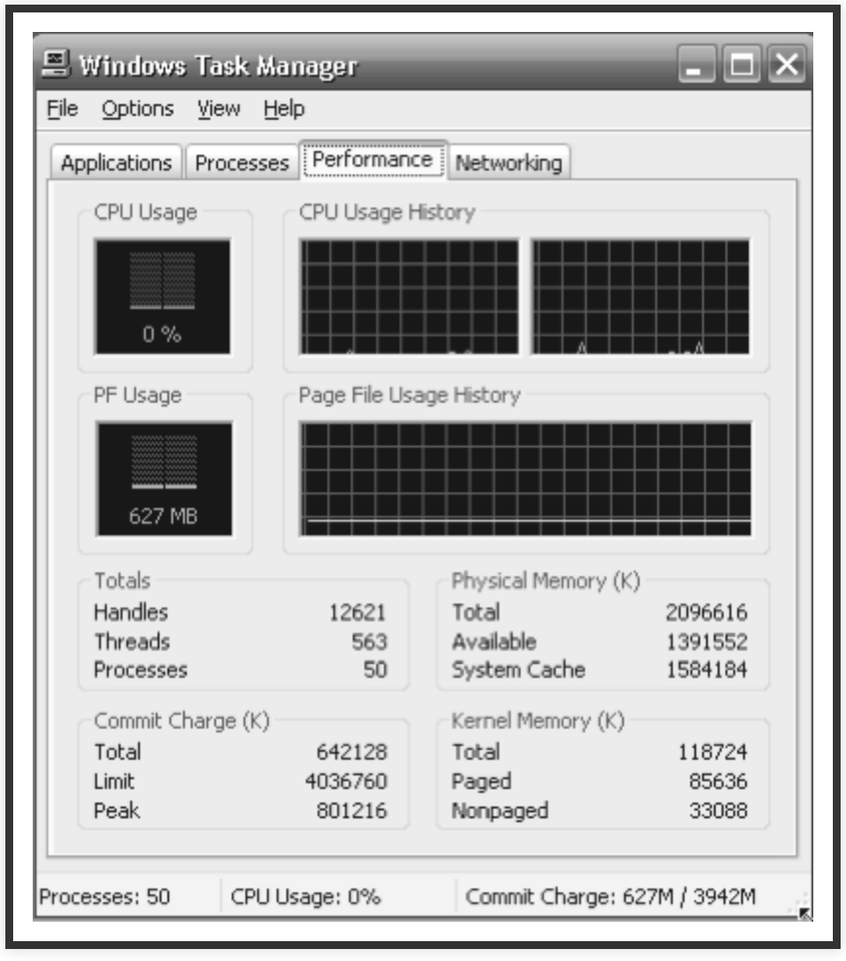
Per-Process

- ps —reports information for a single process or selection of processes
- top —reports real-time statistics for current processes

System-Wide

- vmstat —reports memory-usage statistics
- netstat —reports statistics for network interfaces
- iostat —reports I/O usage for disks

WINDOWS TASK MANAGER



QUESTIONS

BONUS

Exam question number 1: **Operating-System Structures**