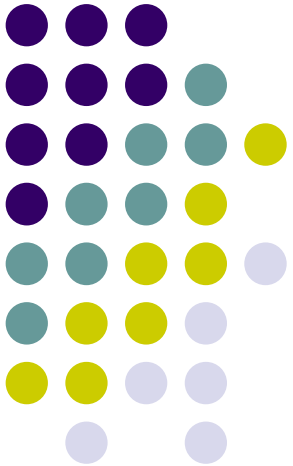
Chapter 2: System Structures

WHAT'S AHEAD:

- Operating System Services
- User and Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- System Boot

WE AIM:

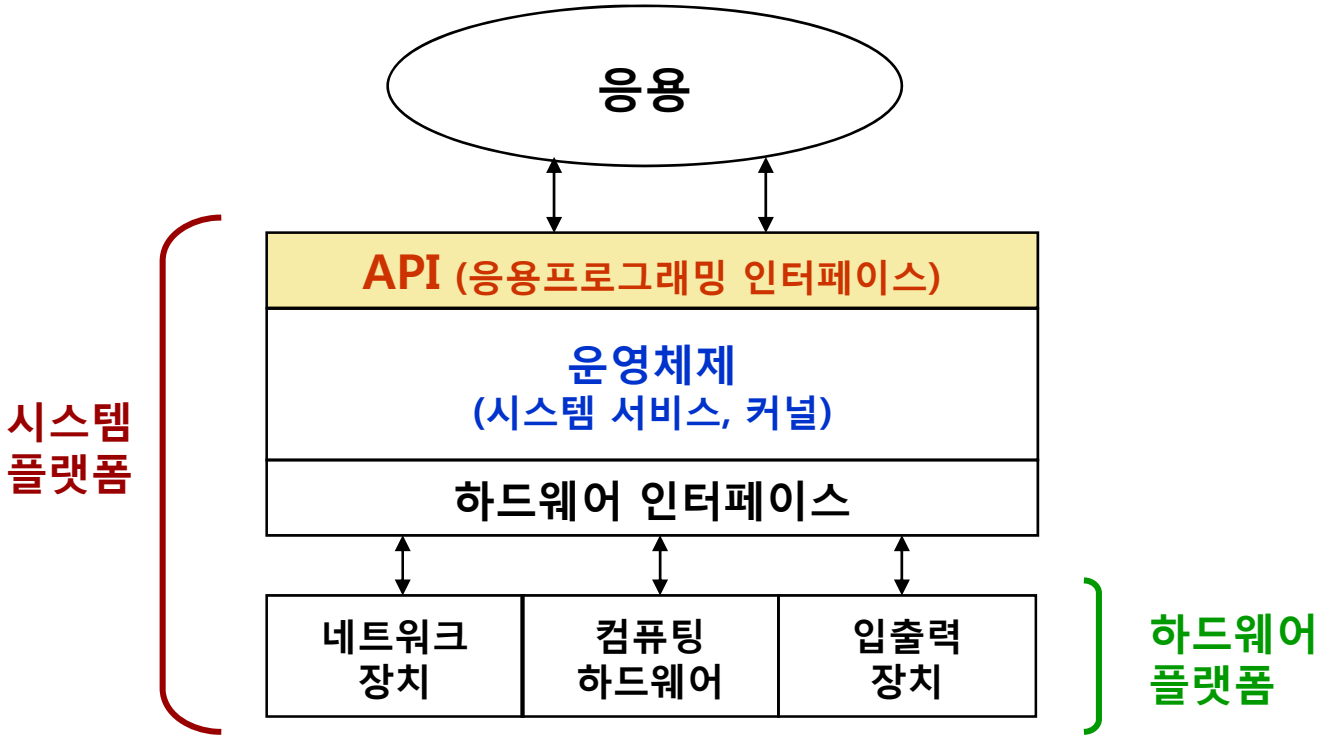
- To describe the services an OS provides to users, processes, and other systems
- To discuss the various ways of structuring an OS
- To explain how OSs are installed and customized and how they boot



Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심 요점

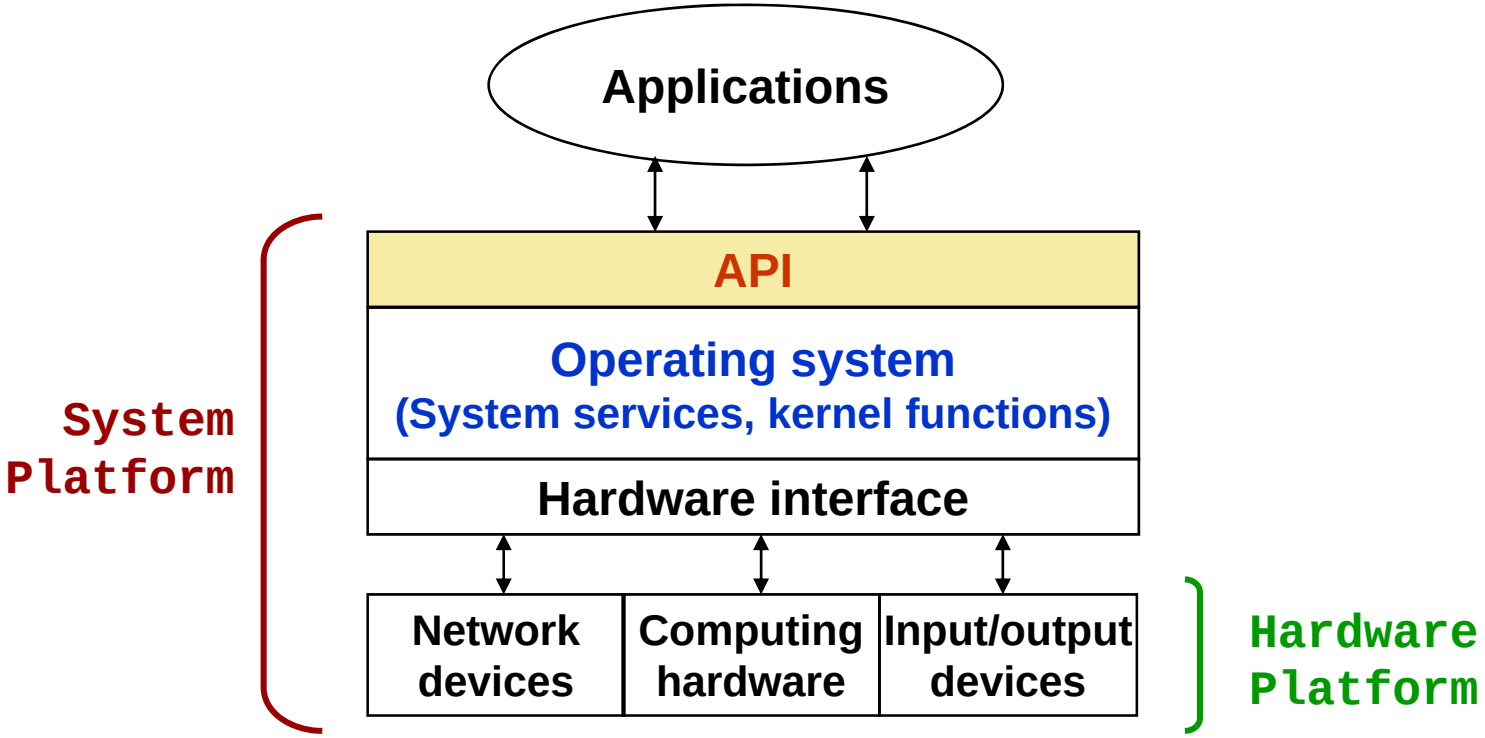
플랫폼으로서의 운영체제



- 운영체제는 응용 프로그램의 개발과 실행을 위한 플랫폼 제공
- 응용 소프트웨어는 응용 프로그래밍 인터페이스 (API)를 통하여 시스템 서비스 이용

Core Ideas

OS as a Platform

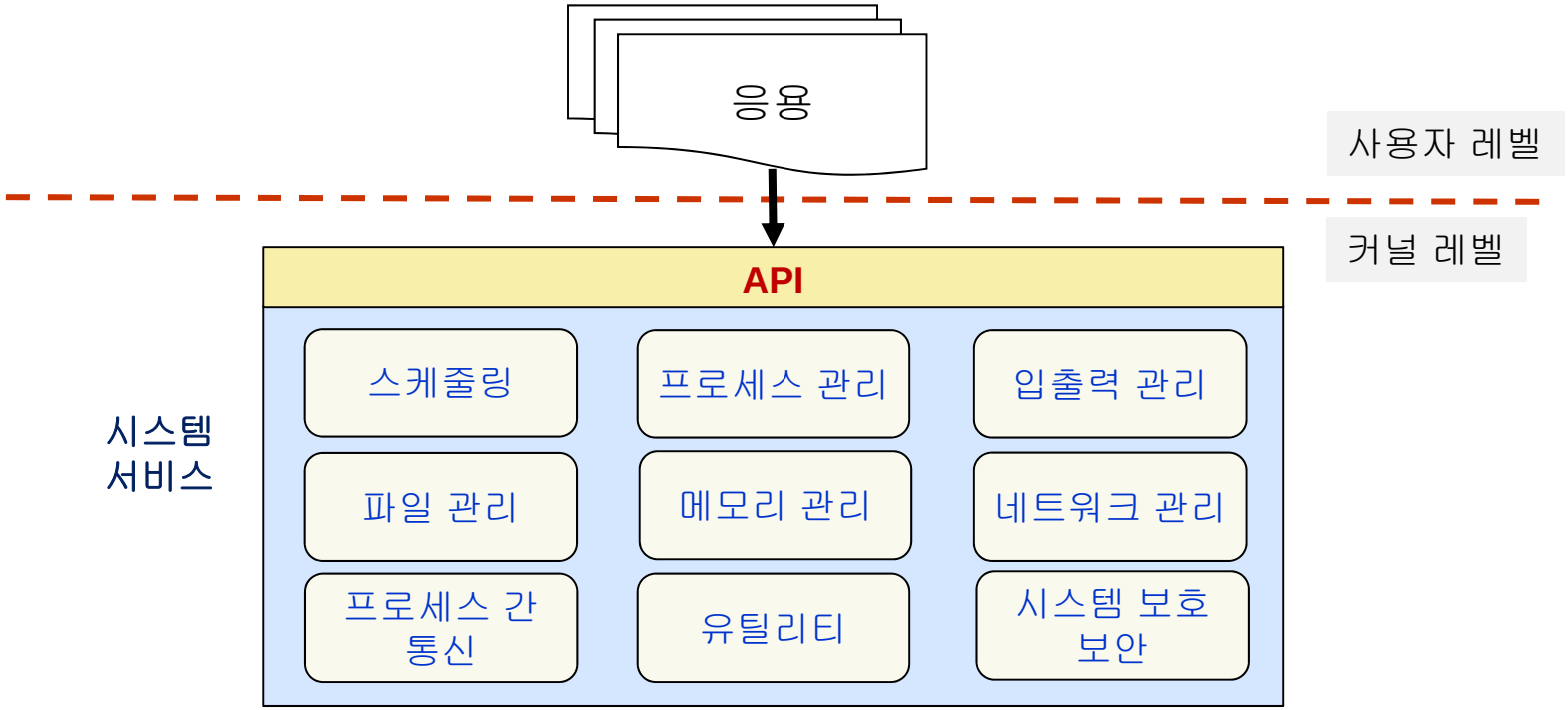


- OS provides a system platform for development and execution of application programs.
- Applications use system services through application programming interface (API).

핵심 요점

시스템 서비스

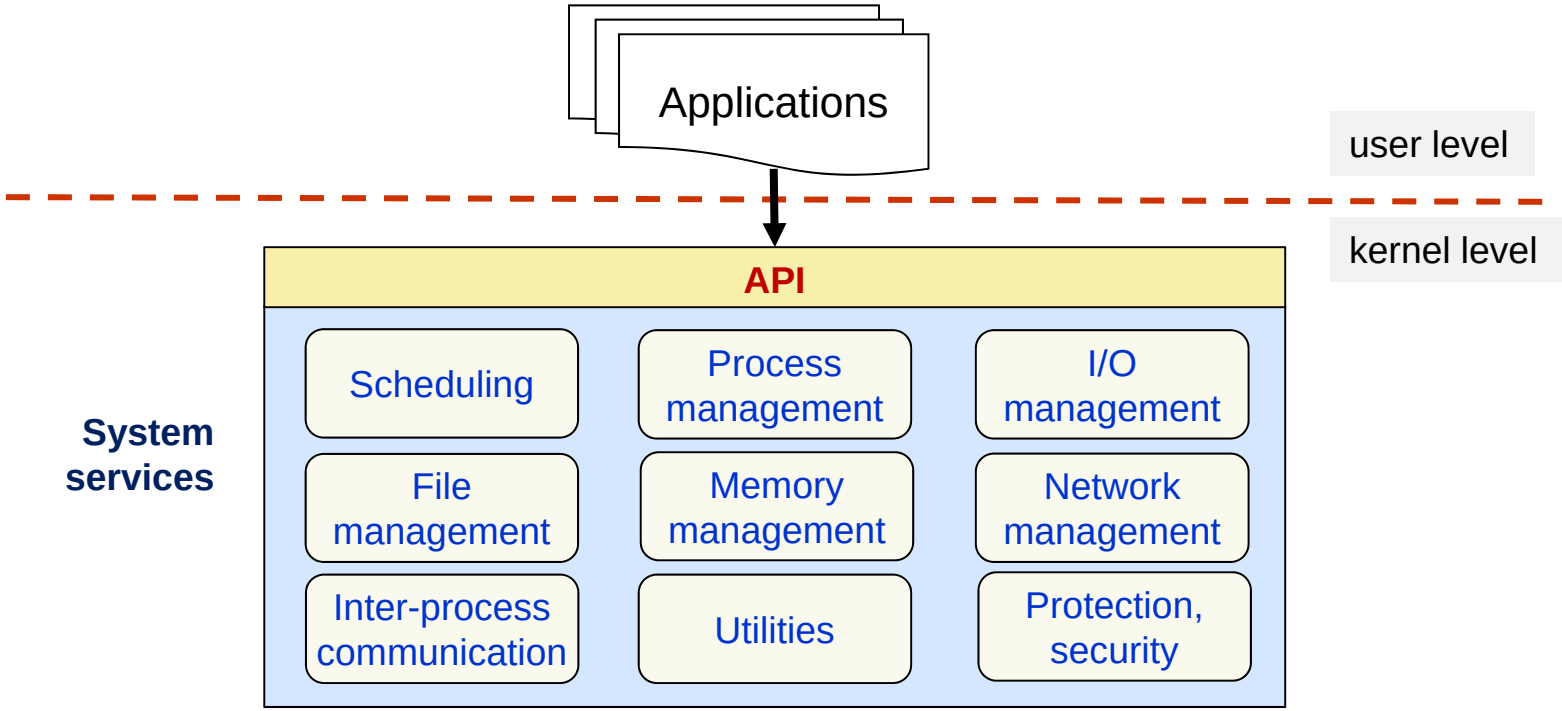
- 시스템 서비스 = 운영체제 서비스 = 커널 서비스
  - 운영체제나 커널이 사용자/응용에 제공하는 각종 기능
  - 일반적으로 커널 레벨에서 시스템 콜의 형식으로 제공됨
  - 현재 대부분의 운영체제에서 시스템 콜 방식에 기초한 API로서 상위 레벨에 제공됨
- 시스템 서비스 관점에서 본 시스템의 구성



Core Ideas System Services

- System services = OS services = Kernel services
  - functionalities that an OS or kernel provides for applications/users
  - usually provided as system calls at the kernel level
  - currently, most OSs export API based on system call convention

- System depicted from the viewpoint of system services



Operating System Services

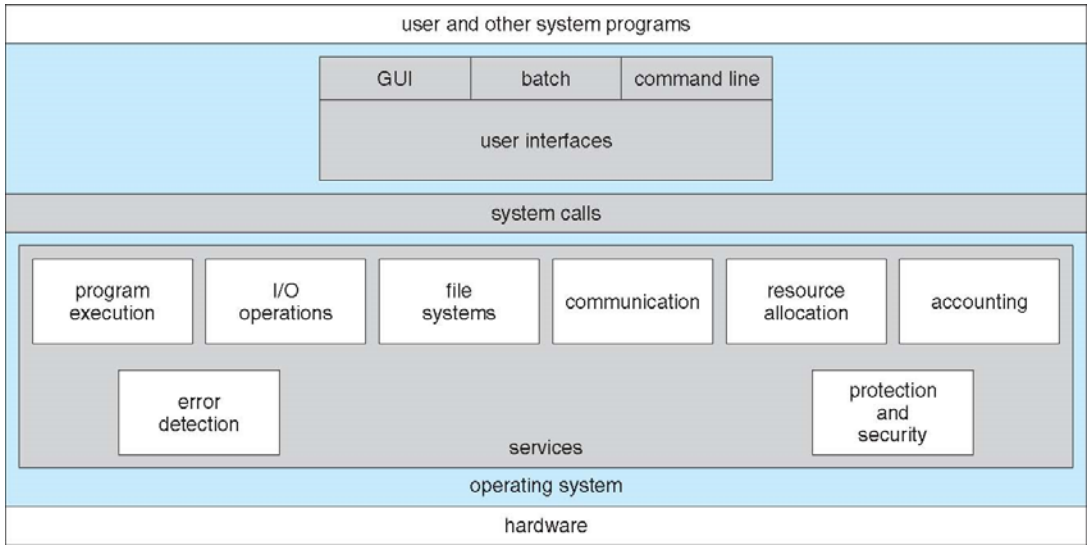
- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - **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.

Operating System Services (Cont.)

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU time, main memory, file storage, I/O devices, etc.
  - **Accounting** - To keep track of which users use how much and what kinds of resources
  - **Protection and security** - To control use of information and concurrent processing in a multiuser or networked computer system
    - Protection ensures that all access to system resources is controlled
    - Security of the system from outsiders requires authentication and access control



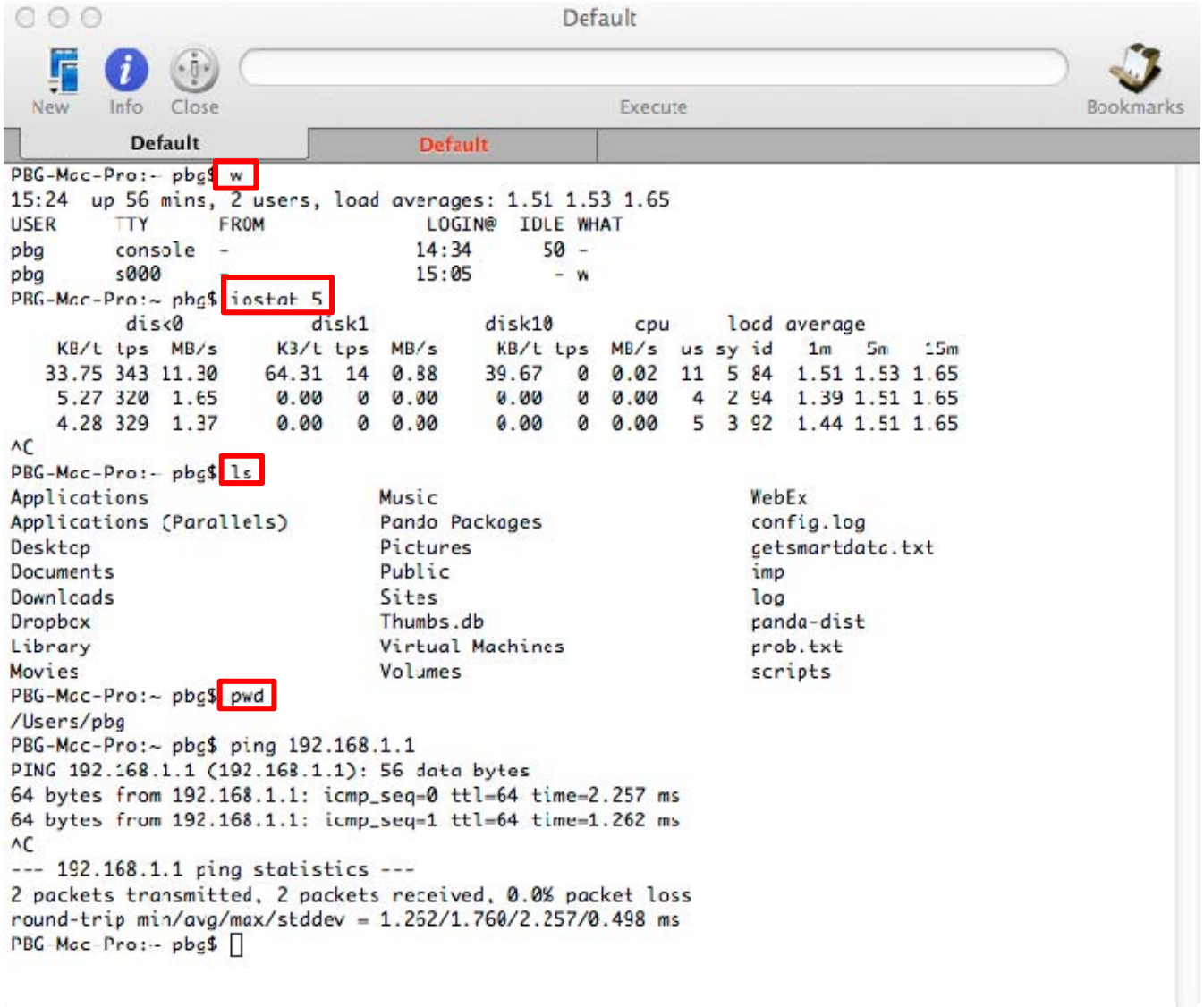
A view of OS services

User and Operating System Interface

CLI (Command-Line Interface [*or* Interpreter])

- 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 mod ification
- *You* type in commands on the screen

Bourne Shell Command Interpreter



GUI (Graphical User Interface)

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - Invented at Xerox PARC (Alto workstation)

- Many systems now include both CLI and GUI
  - 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)

- "Look & feel" (first used by Apple)
  - The look: colors, shapes, layout, typefaces, etc.
  - The feel: buttons, boxes, menus, etc.

Touchscreen Interfaces

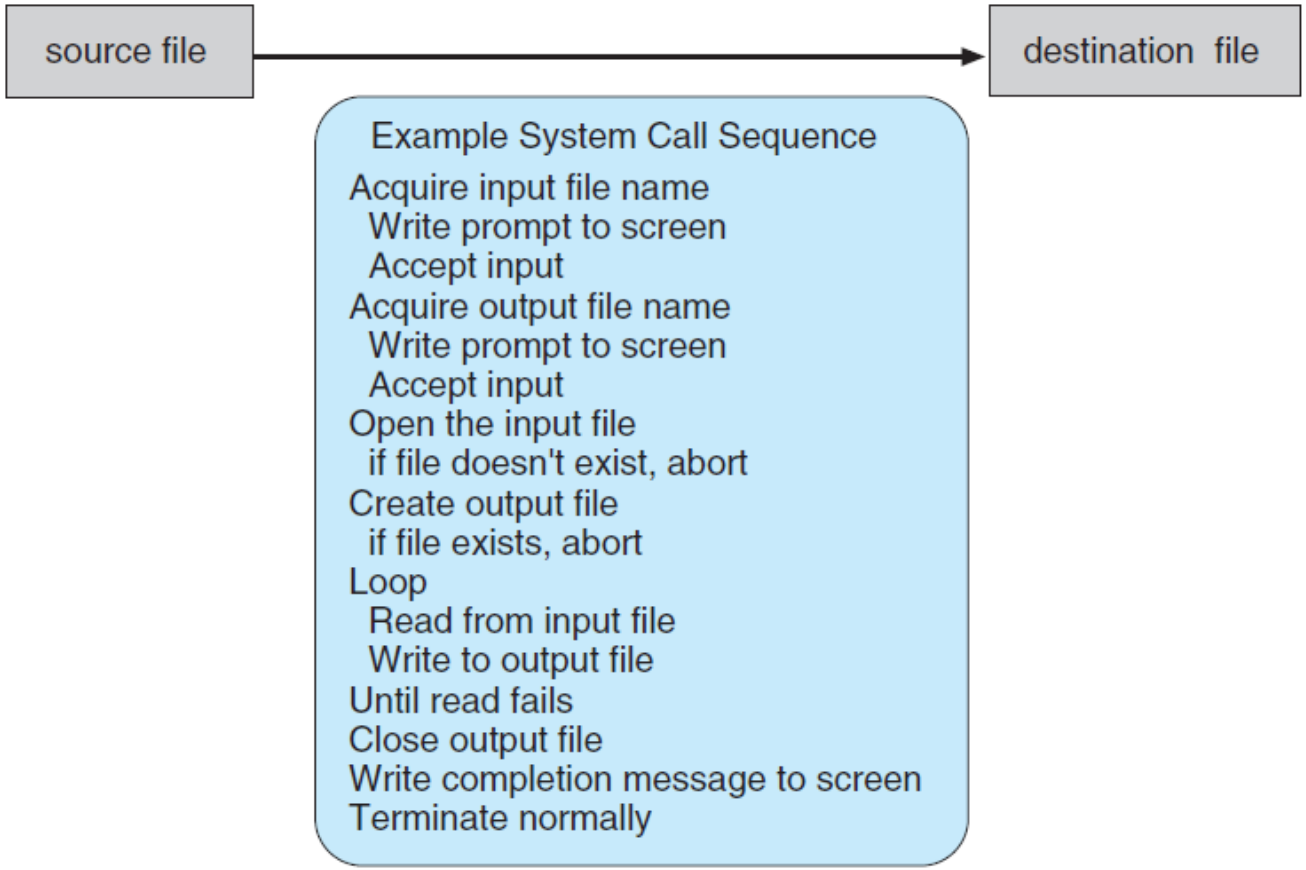
- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Merits
  - User friendly
  - Fast response
  - Error free input
  - Make computing easy , powerful and fun
- Demerits
  - Finger stress and fatigue
  - Large screen is required
  - Big screen leads to low battery life
  - Poorly readable in direct sunlight

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?
  - Program portability and simpler interface: easy to migrate to different platform or version of the same OS
  - More useful functionality: may provide pre-call and post-call code

Example of System Calls

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



Example of Standard API

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return value — function name — parameters

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

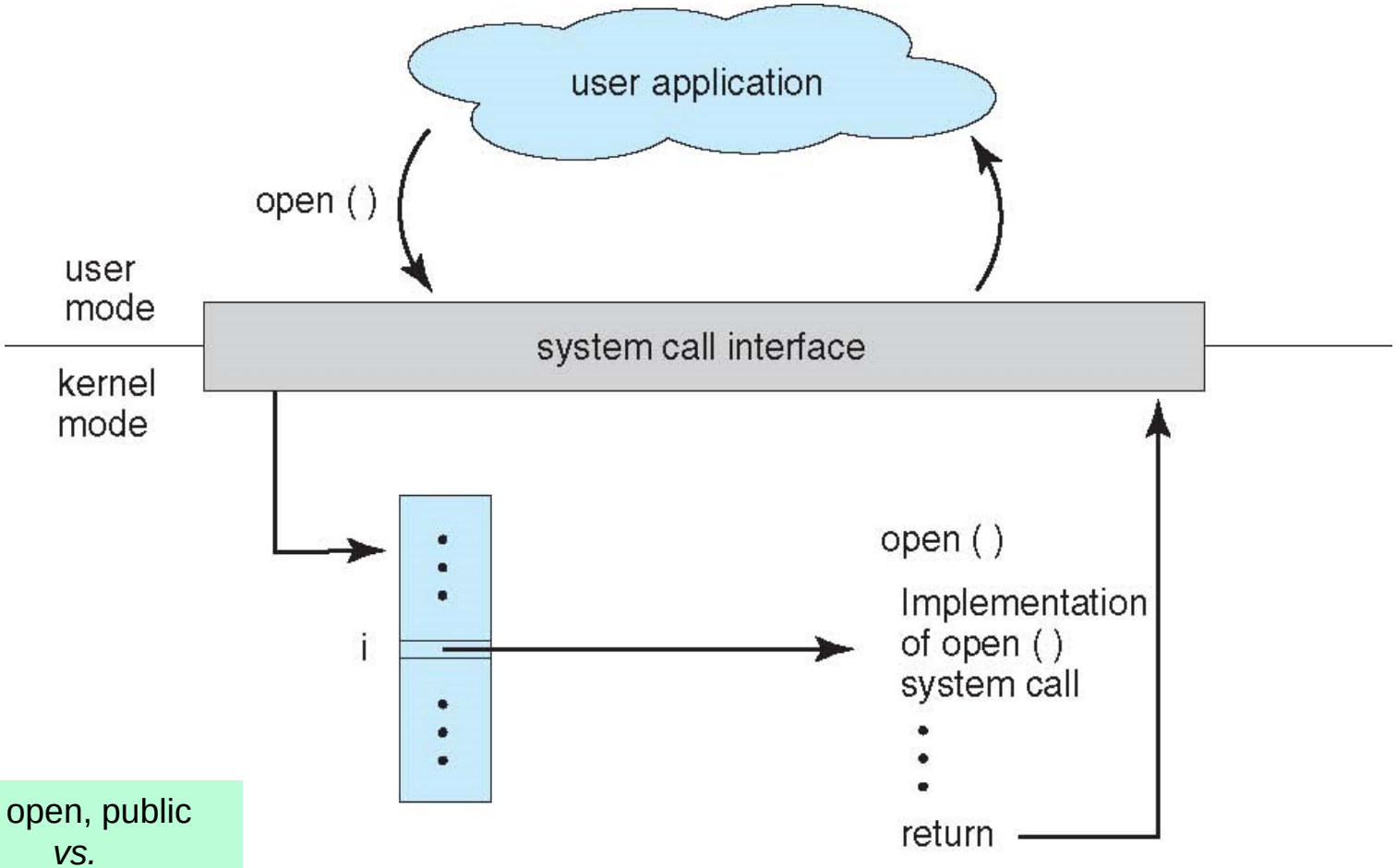
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

- **Note that the system-call names used throughout this text are generic.**
- **API and system-call interface are sometimes used interchangeably.**

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



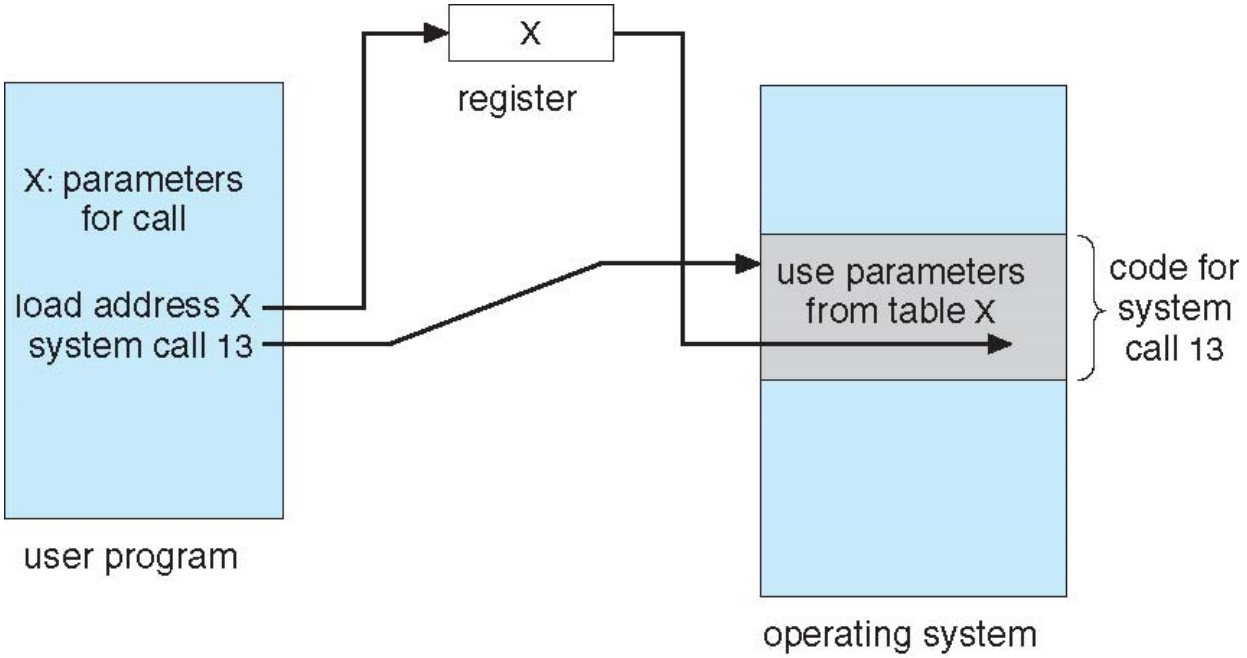
API: open, public
vs.
Implementations: usually protected, private, proprietary

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call

- Three general methods used to pass parameters to the OS
  - Direct and simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Indirect: parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Linux example

- C program

```
int rv;
rv = syscall(SYS_chmod, "/etc/passwd", 0664);
if (rc == -1)
fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

See where the parameters are stored.

- Assembly program
  - Store the system call number in **%eax**, and invoke software interrupt

```
movl  $0xf, %eax       ; system call number for chmod = 15
leal  filename, %ebx   ; load %ebx with the address of a given 'filename'
movl  $0664,%ecx       ; load %ecx with the permissions for a given 'filename'
int  0x80              ; interrupt. after the call, an integer is returned in %eax
cmpl  $-1, %eax        ; check if the return value is -1
je  .print_error       ; if (%eax)<0, jump to code that prints the error message
```

Types of System Calls

- 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

  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

Types of System Calls

- 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

Types of System Calls (Cont.)

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

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

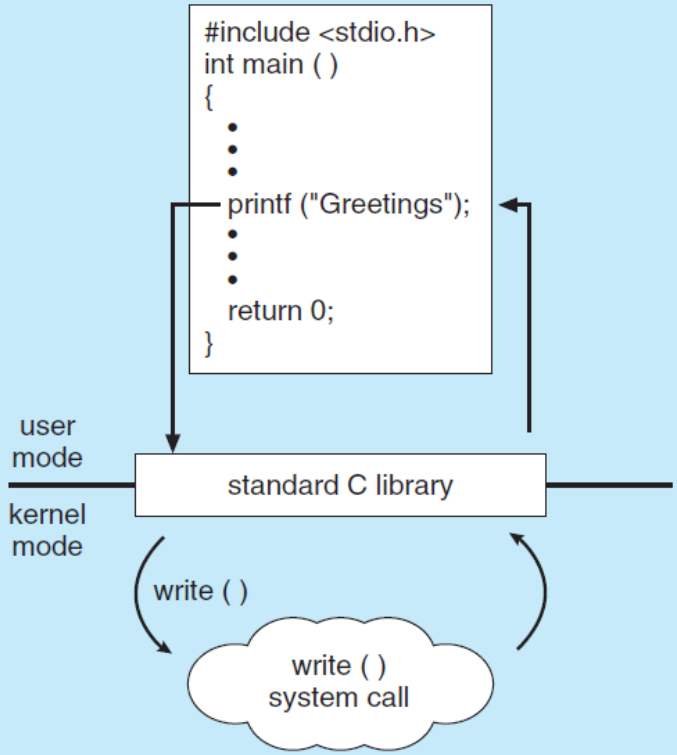
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

Standard C Library Example

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

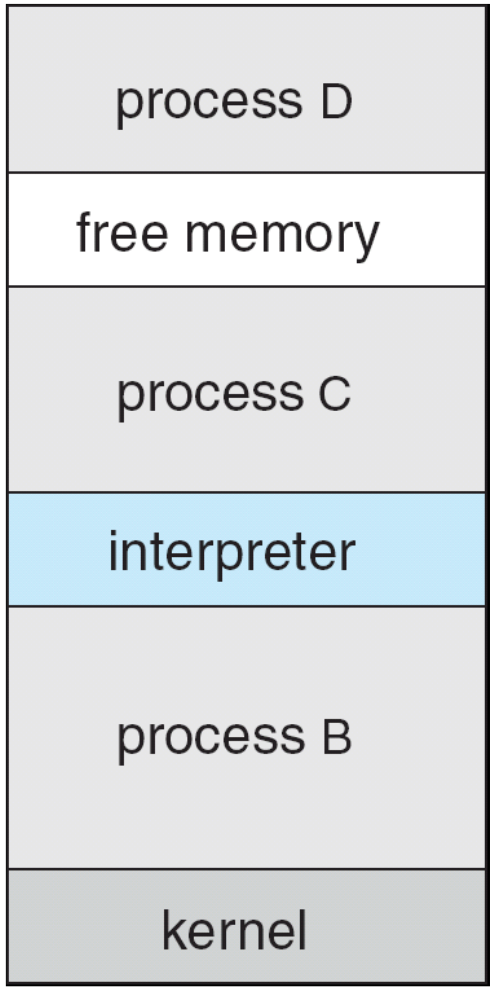
EXAMPLE OF STANDARD C LIBRARY

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. This is shown below:



Example: FreeBSD

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



System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a file modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

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

Operating System Design and Implementation

- Design and implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different operating systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and system goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate
  Policy: **What** will be done?
  Mechanism: **How** to do it?
- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- Specifying and designing OS is highly creative task of software engineering

➢ Critical problem
  - There are no ways to analyze the OS performance in the design process

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 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

emulation: imitation of behavior of a computer with the help of another type of computer

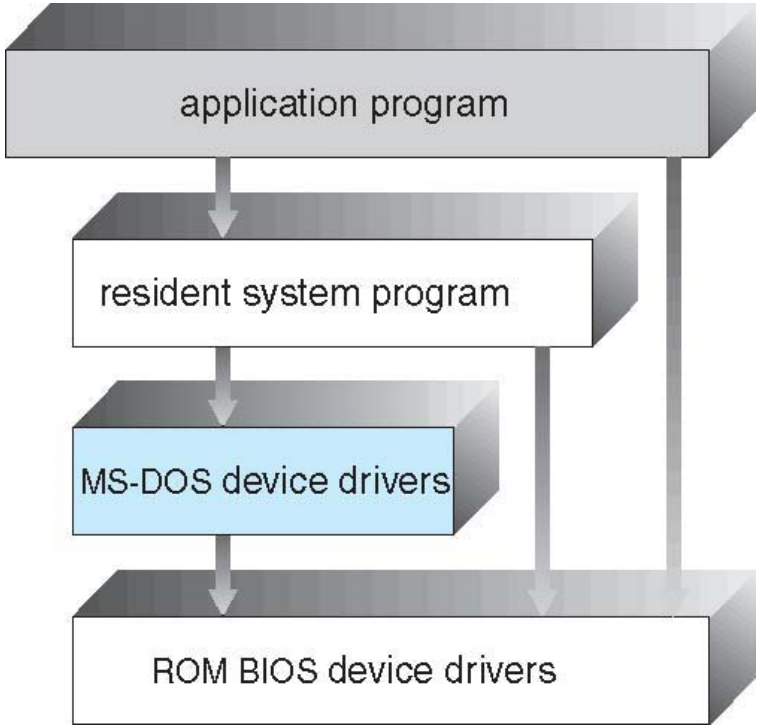
Operating System Structure

- General-purpose OS is a very large program

- Various ways to structure one as follows
  - Simple structure
  - Layered approach
  - Microkernels
  - Modules
  - Hybrid systems

Simple Structure

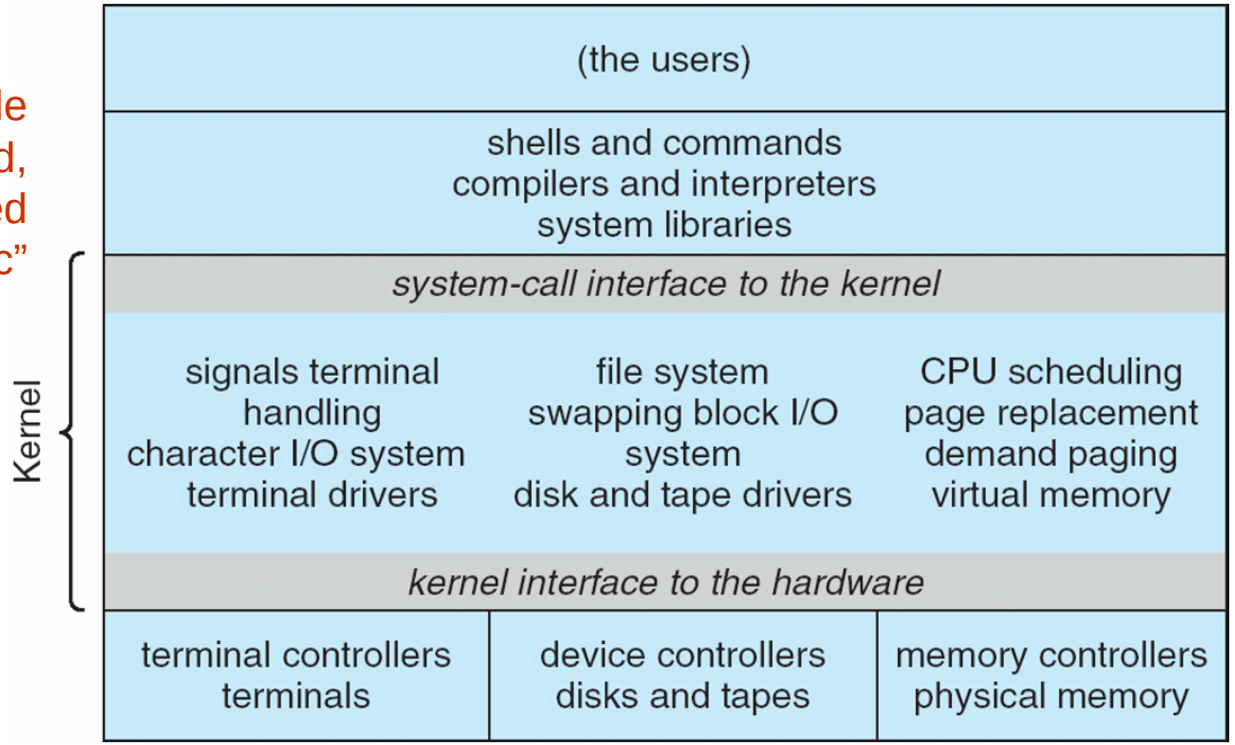
- I.e. MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Applications are allowed direct access to hardware components – No concept of protection



Traditional UNIX System Structure

- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware

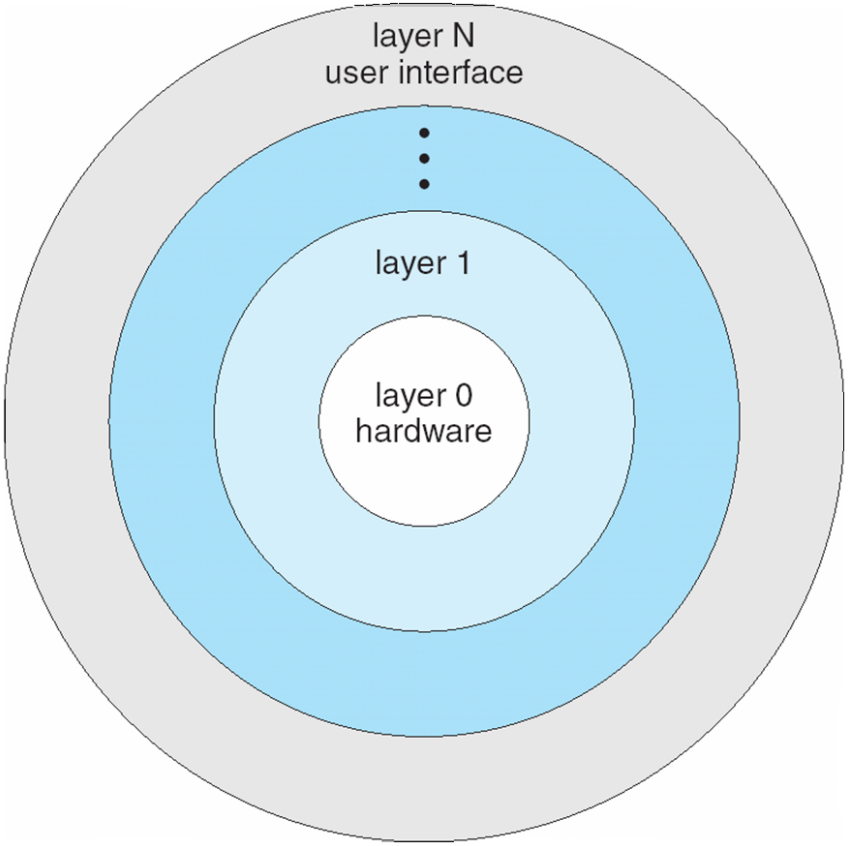
Beyond simple but not fully layered, often called "monolithic"



Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

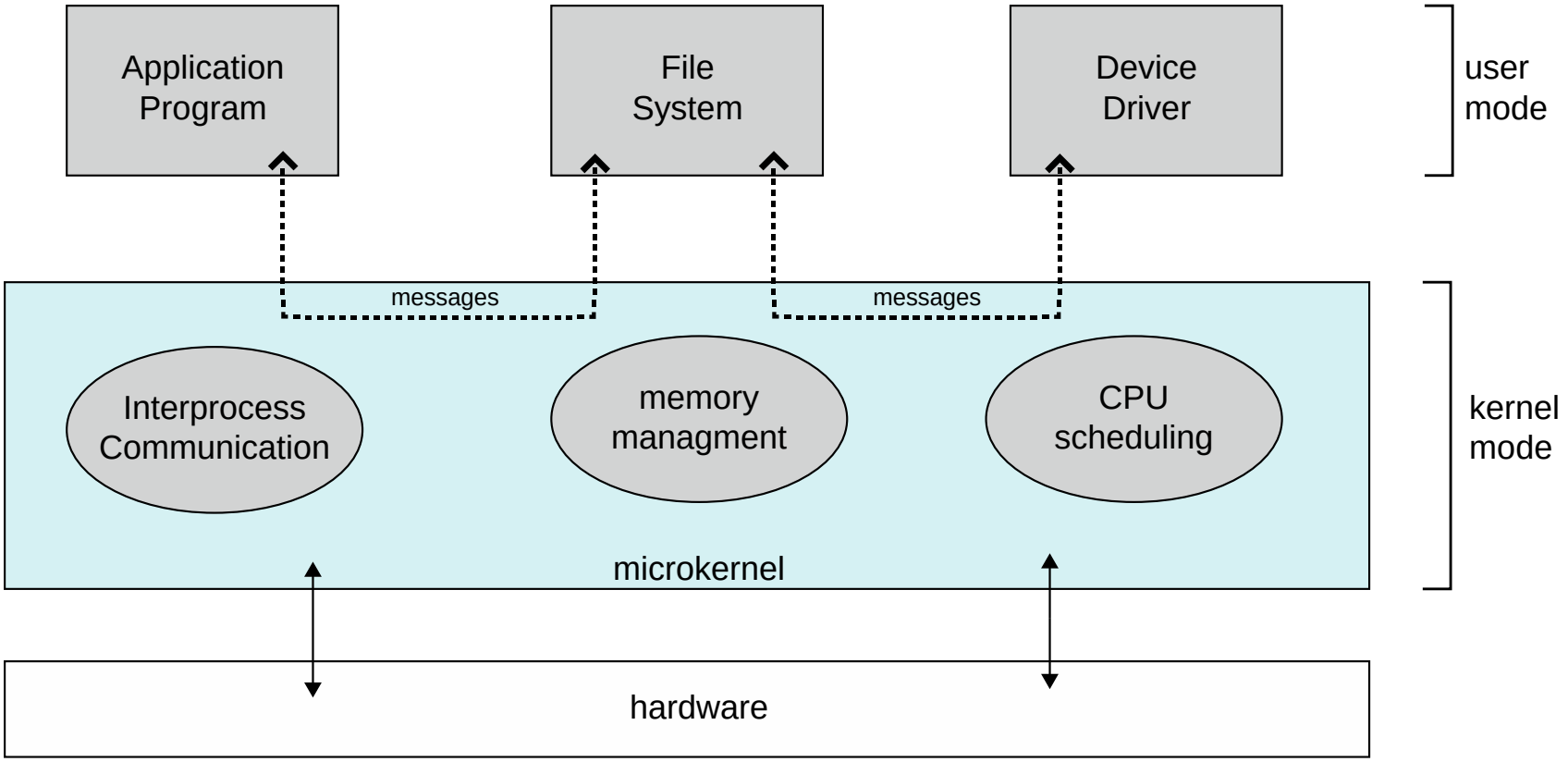


Microkernel System Structure

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

Microkernel architecture differs from *monolithic* architecture in that the monolithic kernel implements all system services and other core functions within a single layer. Unix and its variants are typical examples of monolithic architecture.

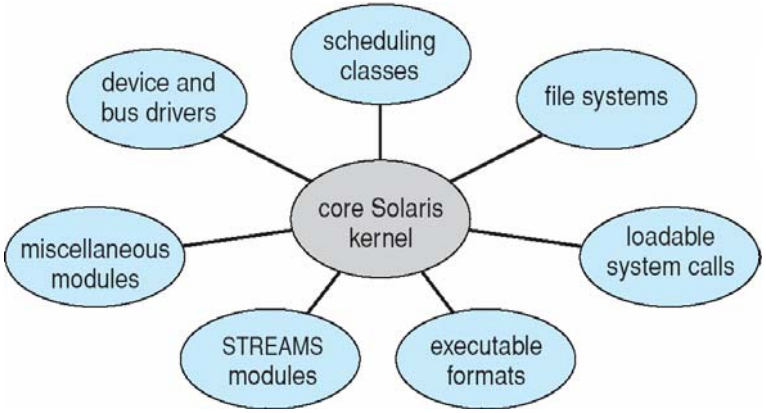
Microkernel System Structure



Modules

- Loadable (kernel) modules
  - can dynamically load (and unload) executable modules at runtime
  - OS becomes more flexible and easily extendable
- Most modern operating systems implement loadable kernel modules
  - Used by Unix-like OSs (such as Linux, FreeBSD, OS X) and MS Windows
  - 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

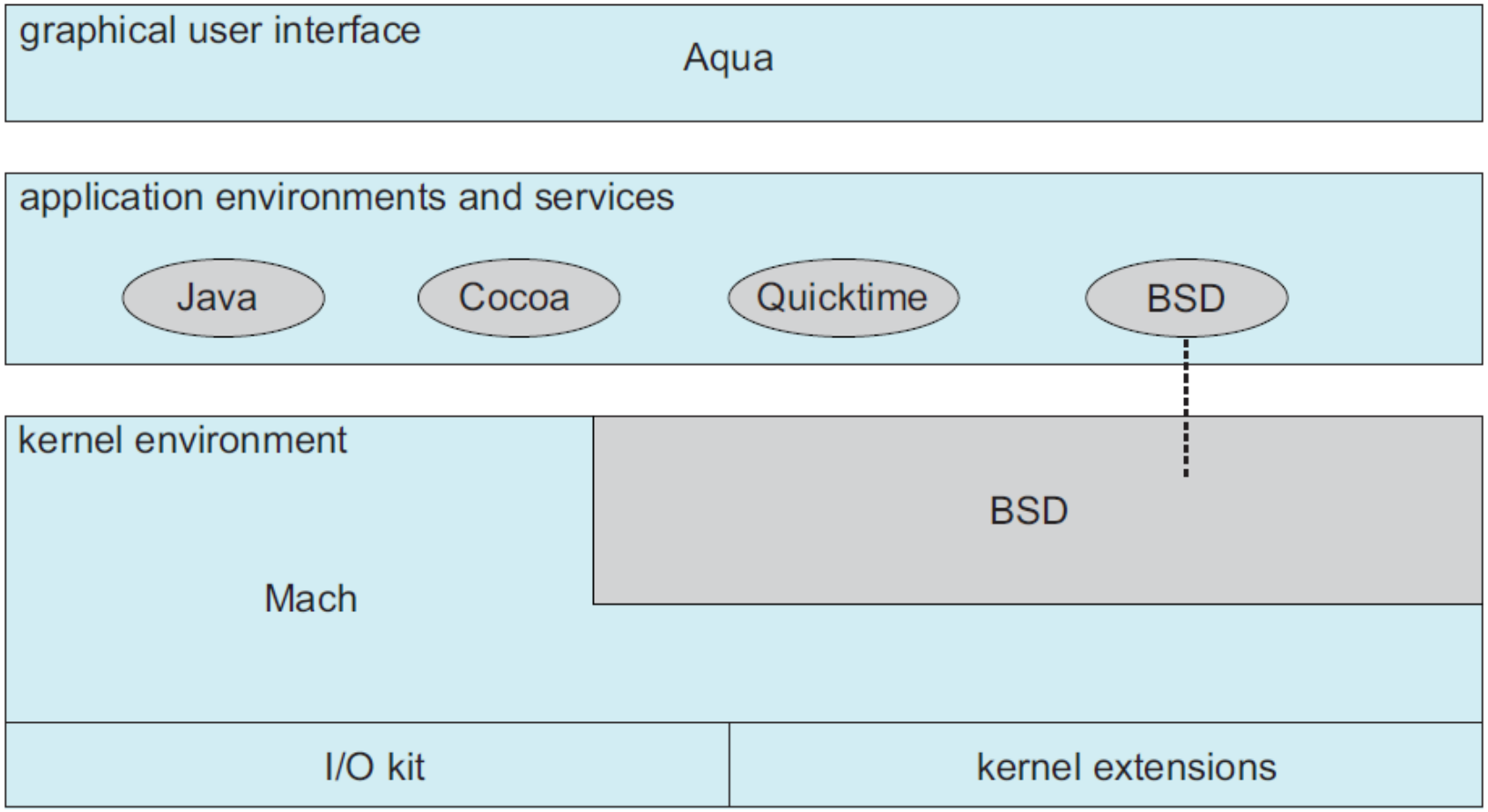
Solaris loadable modules: Solaris OS is organized around a core kernel with seven types of loadable kernel modules



Hybrid Systems

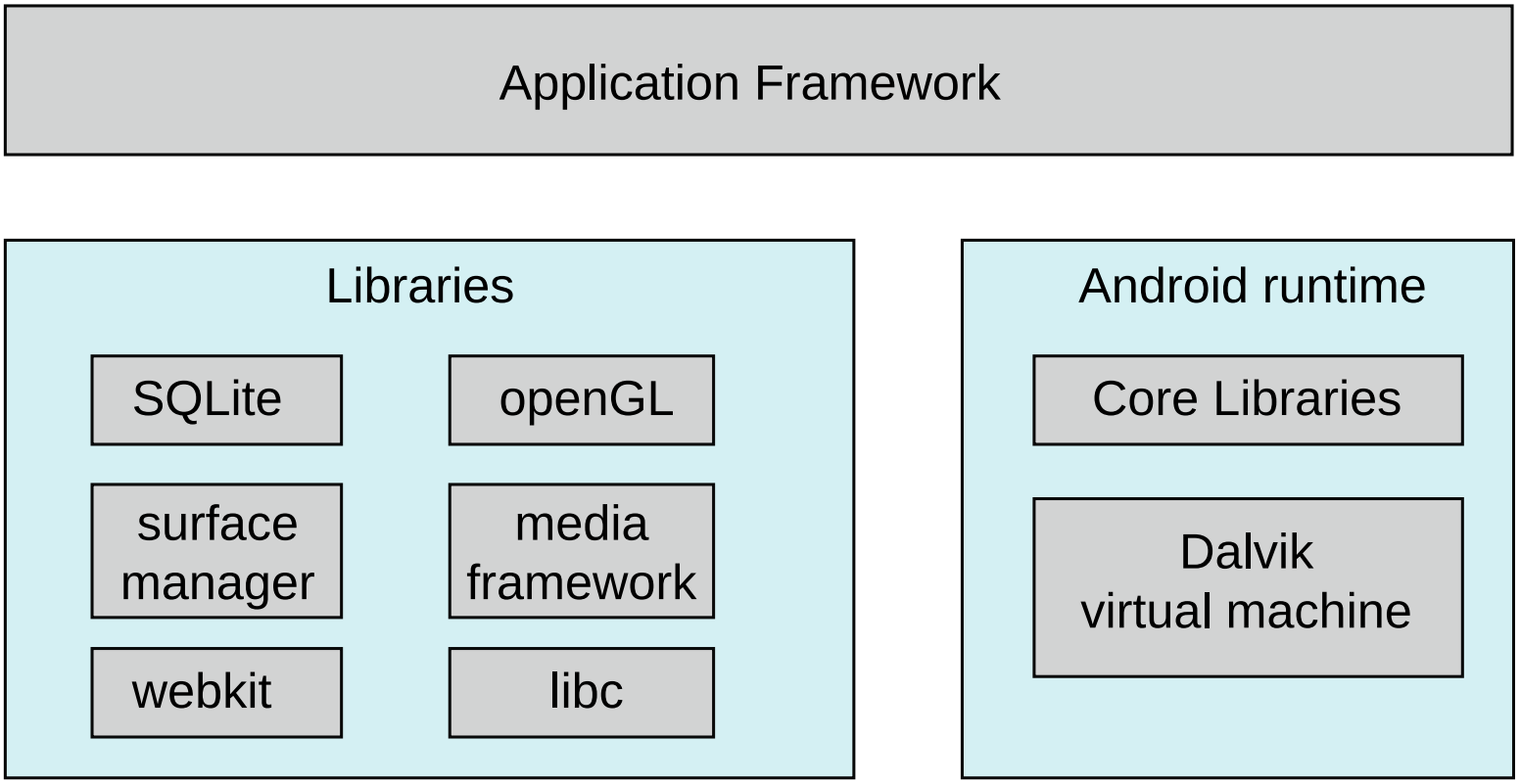
- Most modern operating systems actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic,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

Mac OS X Structure



- iOS: Apple mobile OS for iPhone and iPad
- Structured on Mac OS X, added functionality

Android Architecture



- Developed by Open Handset Alliance (mostly Google) - Open source
- Based on Linux kernel but modified: Adds power management

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where (1st) boot block at fixed location is loaded by (0th) ROM code, which (2nd) loads bootstrap loader from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running

[Wikipedia] GNU **GRUB** (short for GNU GRand Unified Bootloader) is a boot loader package from the GNU Project. GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user with the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular OS's partitions.

Summary

- 운영체제의 역할 중 필수적인 것
  - 응용프로그램이나 사용자가 다양한 기능들(프로그램들)을 사용할 수 있게 함
  - 이러한 기능들을 시스템 서비스 혹은 OS 서비스라 부름
- 시스템 서비스의 구현
  - 제공하는 기능 혹은 함수 (functions)는?
  - 시스템 콜, 이에 따른 인자 전송방법
  - 시스템 콜의 유형
  - API
- 사용자와 운영체제의 인터페이스
  - CLI, GUI, touchscreen, 등
- 기타 시스템 프로그램
  - 파일 관리, 프로그래밍 언어 지원, 프로그램 로딩/실행, 통신, 프로그램 개발 및 실행, 등

- One of the essential roles of OS
  - Enable applications or users to utilize its functions (programs)
  - Such functions are called system services or OS services
- Implementation of system services
  - What are the functions provided by OS?
  - System calls and argument passing methods
  - Types of system calls
  - API
- Interface between users and OS
  - CLI, GUI, touchscreens, etc.
- Other system programs
  - File management, programming language support, program loading/execution, communication, program development and execution, etc.

Summary (Cont.)

- OS는 API를 통해서 시스템 서비스를 응용 프로그램에게 제공
  - 시스템 서비스(혹은 system call)의 유형 : 프로세스, 메모리, 입출력, 파일시스템, 등의 관리
- OS의 설계/구현
  - 정책(policy)과 메커니즘을 구분
  - 다양한 언어 사용: 어셈블리어, C, C++, 스크립트 언어, 등
- OS의 구조
  - 단순하고 단일화된 구성, 계층적 구성, 마이크로커널, 모듈, 혼합방식, 등
- 로더에 의한 시스템 부팅
  - (전원 켜기) → ROM → disk

- OS uses API to provide system services to applications.
  - System services or system calls: managemental activities for processes, memory, I/O and file system.
- OS design and implementation
  - Separate policy and mechanism
  - Use various languages: assembly languages, C, C++, script languages, etc.
- OS structures
  - Simple and monolithic structure, hierarchical layers, microkernel, module and hybrid systems.
- System boot by loaders
  - (power-on) → ROM → disk