Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

Operating System Services

Operating System Interface

System Calls, API, C Libraries

Linkers and Loaders

Operating System Structures

Building and Booting an Operating System

Operating System Services

Operating systems provide an environment that offer services to programs and users

Two categories of services

- Services to the user, e.g., UI (user interface), file tree
- Services for the efficient operations, e.g.,
  - management of memory
  - CPU scheduling

Services to the User

- User interface (UI)
  - Command-Line Interface (CLI), Graphics User Interface (GUI), touch-screen, batch processing (批处理)
- Program execution
  - Load a program into memory
  - Run a program, and then end execution.
- I/O operations
- File-system manipulation
  - Read-write
- Communications
  - Process exchange information
- Error detection
  - Handle possible errors (from hardware or software)

Services for the Efficient Operations

Resource allocation

When multiple users or multiple jobs are using computer concurrently, resources must be allocated to each of them

- Many types of resources
  - CPU, main memory, file storage, I/O devices.

Logging (日志)

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

Protection and security

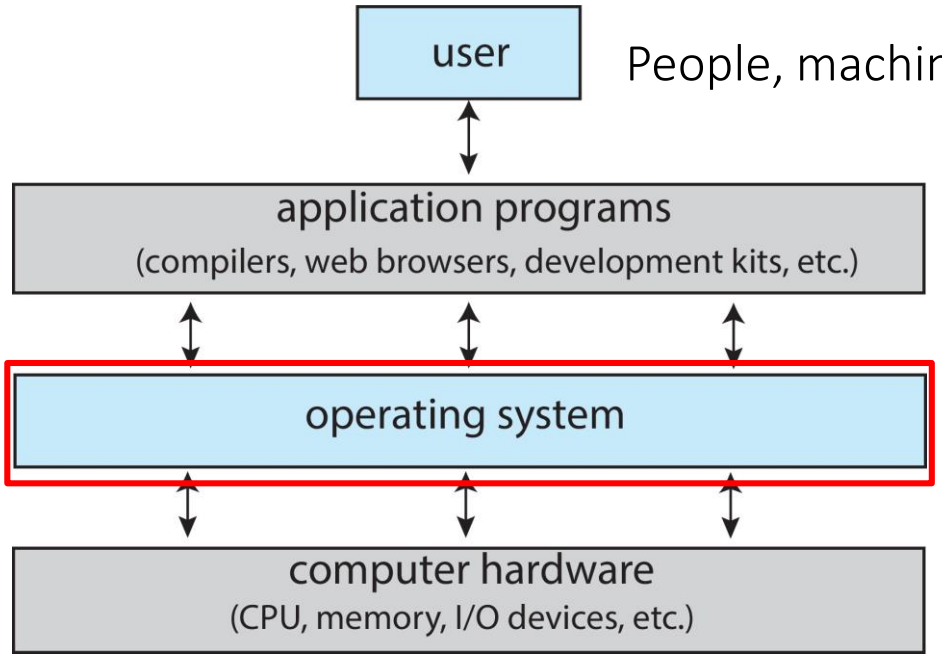
Protection

- Ensure that all access to system resources is controlled

Security

- Avoid attack from outside the system

Abstract View of Components of Computer



People, machines, other computers

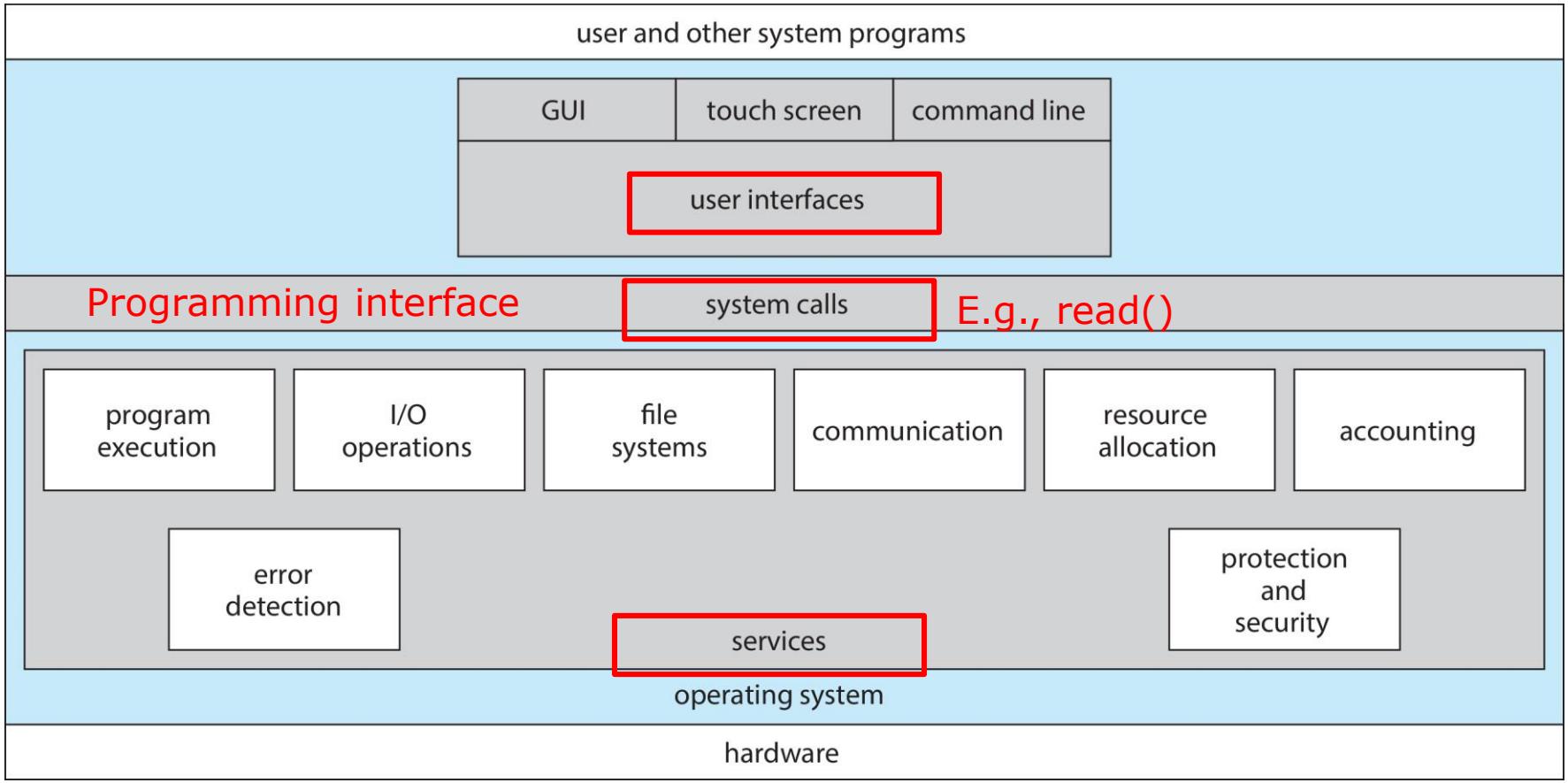
E.g., Word processors, compilers, web browsers, database systems, video games

Controls and coordinates use of hardware among various applications and users

Provides basic computing resources

An abstract view

A View of Operating System Services



A detailed view

User Operating System Interface : CLI

CLI (Command Line Interface) or command interpreter allows direct command entry

Commands are

- sometimes implemented in kernel
  - commands built-in
- sometimes by systems program
  - names of programs
  - Advantage: adding new features doesn't require modification of interpreter

In Kernel and By System Program

In kernel: Commands as functions

```
int main() {
   ....
   while (1){
       read command (str) from UI
       if (command(str) is C1)
            C1(…);  //function for C1
       ……
       if (command(str) is Cn)
            Cn(…);
   }
   ……
}

void C1()
{
   ……
}
```

Commands as system programs

```
int main() {

   ....
   while (1){
       read command (str) from UI
       load str.exe and run
   }
   …
}
```

C1.exe
C2.exe
…
Cn.exe

On hard disk

Questions:

1. How to add a new command Cm in above two implementations?
2. Which one is easier to add a new command

In Kernel and By System Program

In kernel: Commands as functions

```
int main() {
   ....
   while (1){
        read command (str) from UI
        if (command(str) is C1)
             C1(…);  //function for C1
        ……
        if (command(str) is Cn)
             Cn(…);
      if (command is Cm)
       Cm(….);

  }
   ……
}

void C1()
{
   ……
}
…….
```

Need to change Kernel

Commands as system programs

```
int main() {

   ....
   while (1){
        read command (str) from UI
        load str.exe and run
  }
   …
}
```

No change to Kernel

c1.exe
c2.exe
…
cn.exe
Cm.exe

On hard disk

Questions:

1. How to add a new command Cm in above two implementations?
2. Which one is easier to add a new command

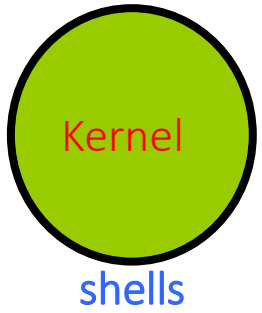
User Operating System Interface - CLI

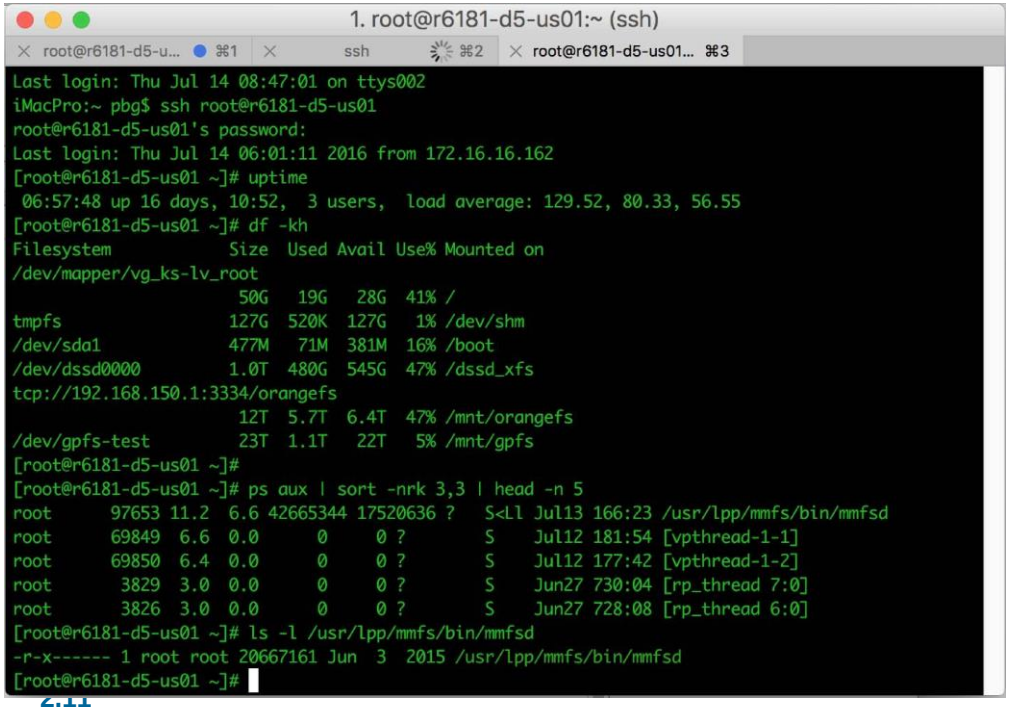
Shell (vs. Kernel)

Multiple flavors of interpreters implemented

Different types of shells in Linux

- The Bourne Shell (sh)
- The GNU Bourne-Again Shell (bash)
- The C Shell (csh)
- The Korn Shell (ksh)
- The Z Shell (zsh)





Bourne Shell Command Interpreter

User Operating System Interface - GUI

GUI (Graphical User Interface)

User-friendly

- desktop metaphor (象征) interface
  - Icons represent files, programs, actions, etc.

Invented at Xerox PARC (1970s)

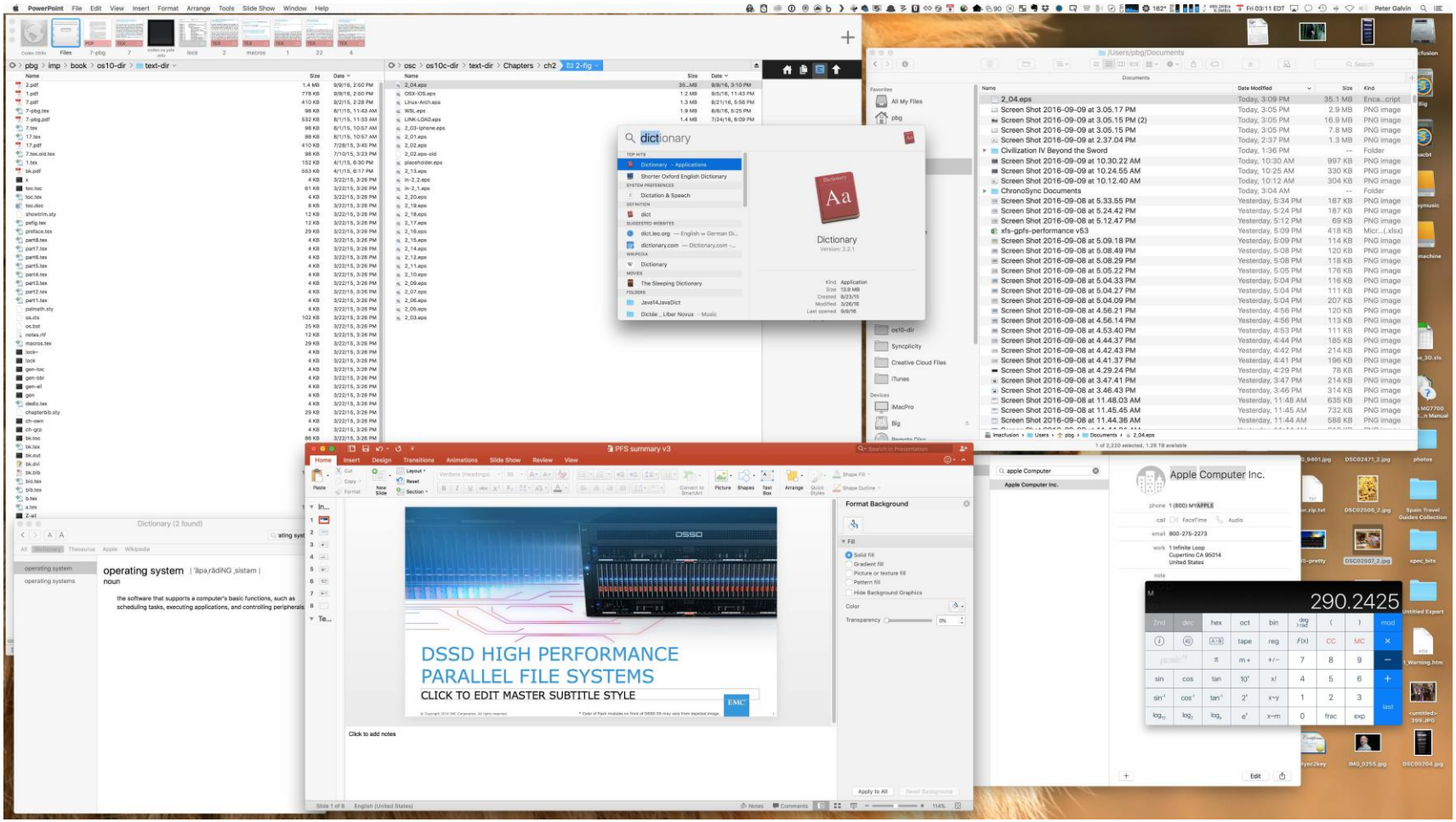
Many systems now include both CLI and GUI interfaces

Microsoft Windows is GUI with CLI “cmd” 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

The Mac OS X GUI

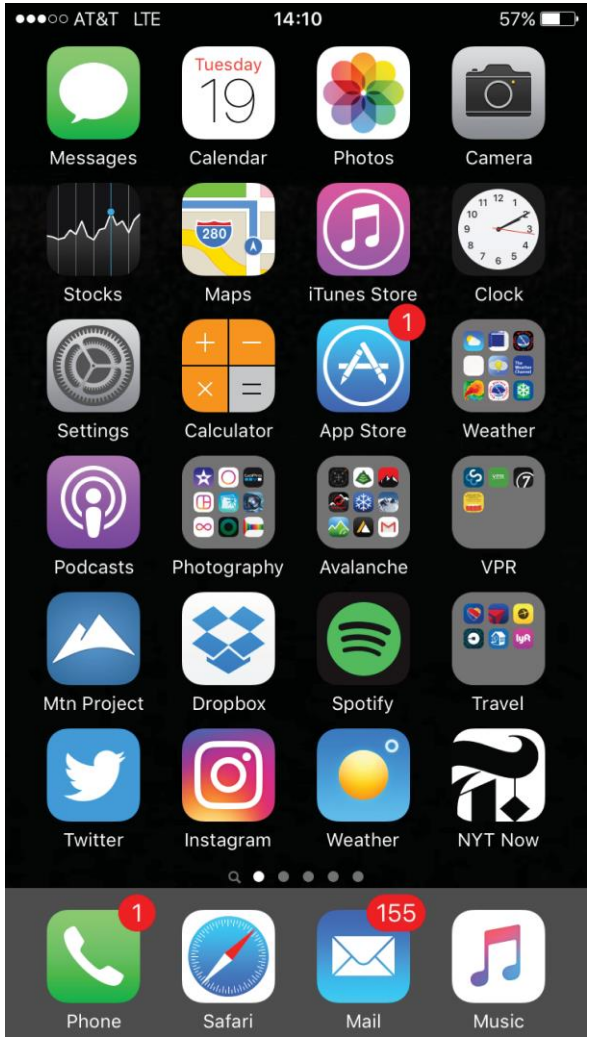


GUI: Touchscreen Interfaces

Touchscreen devices require new interfaces

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

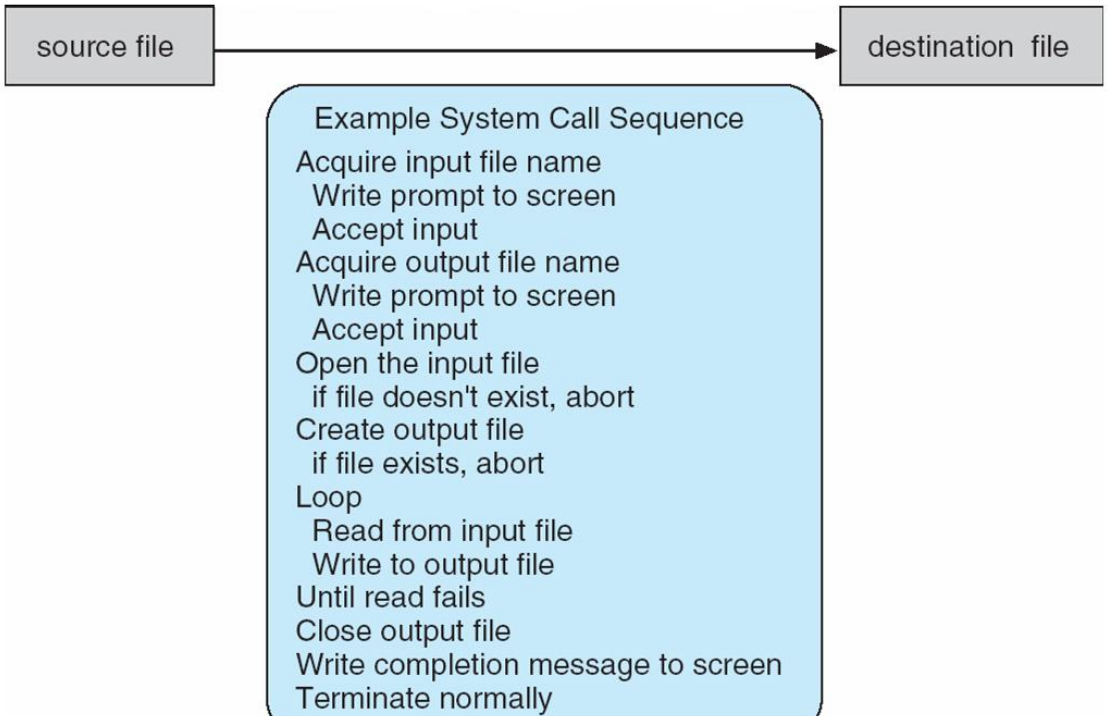
Voice commands



System Calls

System calls provide an programming interface for programs to use services of operating systems.

- Examples: write(…), read(…) on Unix/Linux
- Typically written in a high-level language (C or C++)
- An example: system call sequence to copy the contents of one file to another file



API

Typically, application developers design programs according to an API rather than directly system calls.

The API (Application Programming Interface) specifies a set of functions that are available to an application programmer.

E.g.,

- When call Windows function CreateProcess, actually, it is making a system call NTCreateProcess to Kernel

API

Three most common APIs (libraries):

Win32 API for Windows

POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)

- POSIX (Portable (可移植的) Operating System Interface) is
  - a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Java API for the Java virtual machine (JVM)

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.

size_t: unsigned int /long

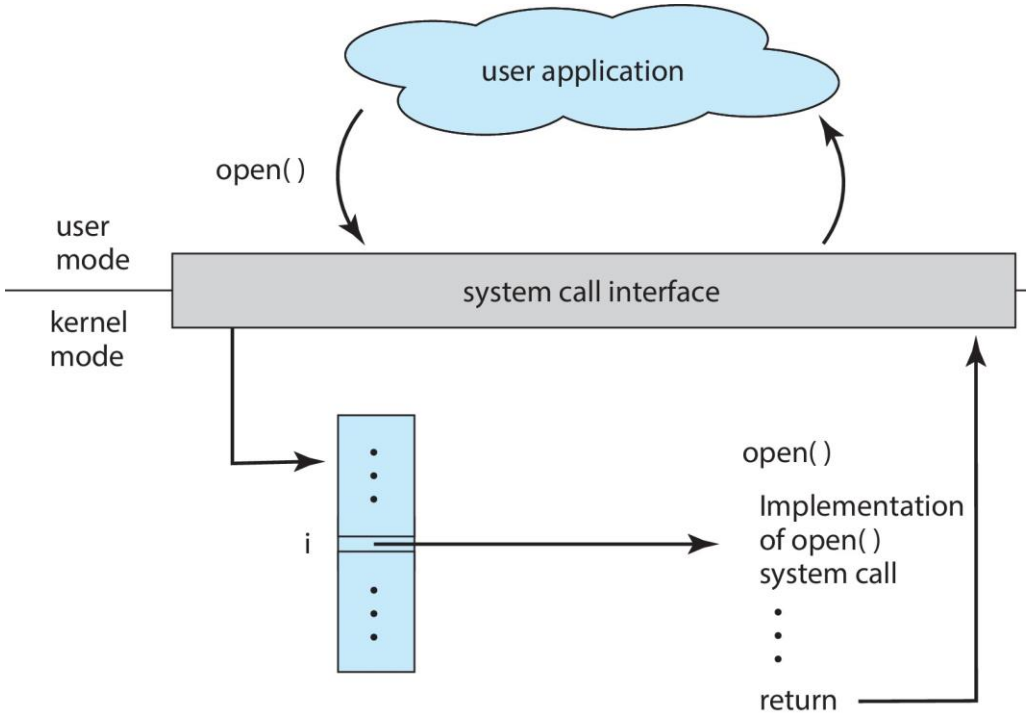
ssize_t: signed size_t, e.g. int/long

API and System Call

The caller just needs to obey API

know nothing about how the system call is implemented

Most details of OS interface hidden from programmer by API



The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values

Each system call has a number (as index)

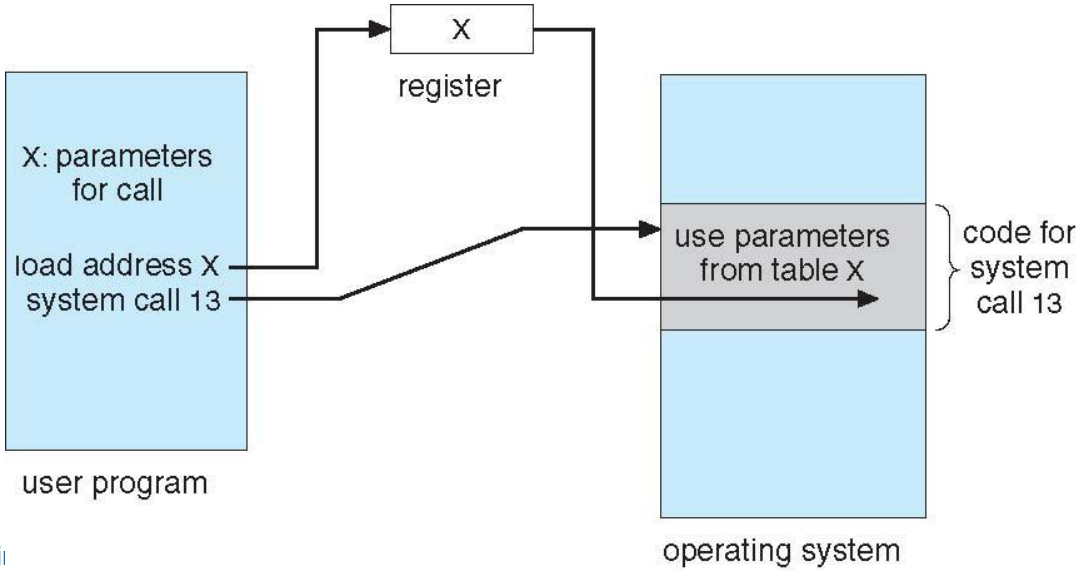
System-call interface maintains a table: indexed according to these numbers

System Call Parameter Passing

Three general methods used to pass parameters to the OS

1. Simplest/Fastest: pass the parameters in registers
   - Disadvantage: more parameters than registers?
2. Parameters stored in a block/table, in memory, and address of block passed as a parameter in a register.
3. Parameters are pushed onto the stack (栈) by the program and off the stack by the operating system

No limit of the number of parameters



Method 2:
Parameter through table

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
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
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

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()`

For Linux, type the Command: man 2 syscalls, you can find out system calls info

http://manpages.ubuntu.com/manpages/cosmic/man2/syscalls.2.html

https://filippo.io/linux-syscall-table/

Standard C Library vs System Call

Standard C library

contains functions which make system calls or do not make system calls

makes programmers' work much easier

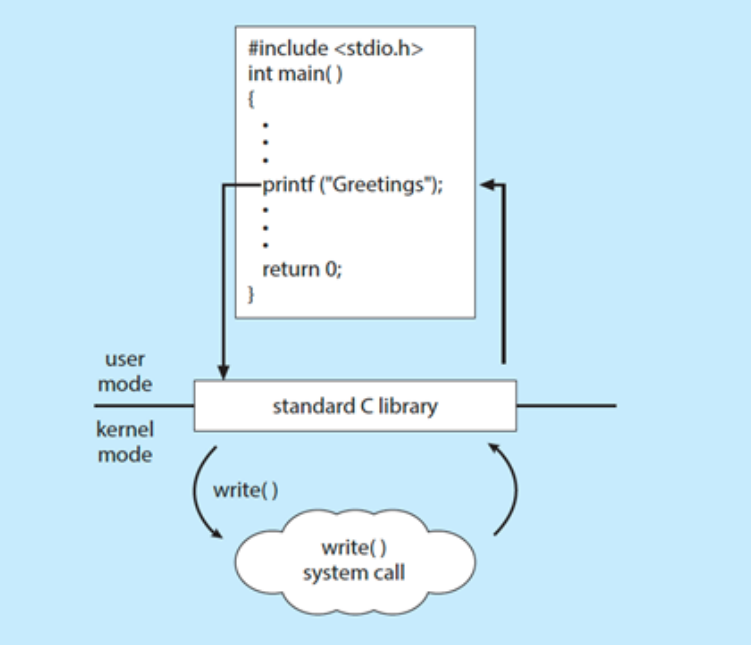
Advantages of using Standard C Library

It is simpler to call a function in a standard C library rather than to make a system call

Portability

- Source code executed in one OS can be run in another OS
  - E.g., *printf*
    - » All OSes supports *printf*
    - » Source code that use *printf* can be reused in another OS after re-compiled.

Standard C Library vs System Call: Portability



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

Library function call

System call

- The code of printf itself is different on different operating systems.
- If direct system call is used, program is not portable.

Standard C Library vs System Call

Advantages of using system calls

System calls are usually more powerful than functions from the Standard C Library

- E.g., you can create processes, share memory between processes, etc. These advanced features are not available in the Standard C Library.

It's a little bit faster than using a library function (since internally the library function uses a system call anyway).

In practice, for normal programmers, code portability is the most important issue, so use functions from the C Standard Library as much as possible in your own code!

Standard C Library vs C POSIX Library

The C POSIX library

was developed at the same time as the ANSI C standard.

includes additional functions to those introduced in standard C

Standard C library vs C POSIX library

subset $\sqsubset$ superset

Standard C Library Example

```c
#include <stdio.h>
#define SIZE 1024

int main(void) {
 char prompt[] = "Type a command: ";
 char buf[SIZE];

 // Ask the user to type a command:
 printf(prompt);

 // Read a string from kb using fgets.
 // Note that fgets also puts into the array buf
 // the '\n' character typed by the user
 // when the user presses the Enter key):
 fgets(buf, SIZE, stdin);
 // Replace the Enter key with '\0':
 for(int i = 0; i < SIZE; i++) {
  if(buf[i] == '\n' || buf[i] == '\r') {
   buf[i] = '\0';
   break;
  }
 }

 // print the command:
 printf("command: %s\n", buf);
 return 0;
}
```

Prototype for fgets function:
char *fgets(char *str, int n, FILE *stream)

'\r': carriage return, ASCII: 13
'\n': new line, ASCII: 10

"ENTER KEY":
- Linux: '\n'
- Windows: '\r\n'
- Mac: '\r'

System Calls C Example

```c
#include <unistd.h>
#include <string.h>
#define SIZE 1024
int main(void) {
      char prompt[] = "Type a command: ";
      char buf[SIZE];

      // Ask the user to type a command:
      write(1, prompt, strlen(prompt));

      // Read from the standard input the command typed by the user (note
      // that read puts into the array buf the '\n' character typed
      // by the user when the user presses the Enter key on the keyboard):
      read(0, buf, SIZE);

      // Replace the Enter key typed by the user with '\0':
      for(int i = 0; i < SIZE; i++) {
            if(buf[i] == '\n' || buf[i] == '\r') {
                  buf[i] = '\0';
                  break;
            }
      }

      // output the user typed command:
      write(1, buf, strlen(buf));
      write(1, "\n", 1);
      return 0;
}
```

Operating System Definition

The definition of OS (Lecture 1)

"The one program in memory at all times on the computer" is the kernel, part of the operating system

Everything else is either

a system program (ships with the operating system, but not part of the kernel) , or

an application program, all programs not associated with the operating system

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
- Programming language support
- Program loading and execution
- Communications
- Background services (e.g., launch OS, disk checking, daemons(守护进程))

Most users' view of the operation system is

- defined by system programs, not the actual system calls (for system or application programmers)

Why Applications are Operating System Specific

- Apps compiled on one system usually are not executable on other operating systems
  - Each operating system provides its own unique system calls
- How can apps be used in multi-operating systems
  - Written in an interpreted language like **Python, Ruby**, and interpreter available on multiple operating systems
  - Written in a language that includes a **VM** containing the running app (like **Java**)
  - Written in a standard language (like **C**), compiled separately on each operating system to run on each OS
- Application Binary Interface (ABI) is
  - about how different components of binary code can interface for a given operating system on a given architecture in low-level details

Operating System Implementation History

Much variation

- Early OSes in assembly language
- Then system programming languages like Algol, PL/1
- Now high level language C, C++
  - High-level language is easier to port to other hardware
  - But slower

Actually usually a mix of languages

- Lowest levels in assembly language
- Main body in C
- Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

Emulation can allow an OS to run on non-native hardware

Operating System Structure

General-purpose OS is a very large program

- Linux (2015): 20.2 MLOC (million lines of code)
- Windows 7: 40 MLOC
- Windows 8: 60 (?) MLOC
- Mac OS X (2005): 85 MLOC
- Windows 10: ?

Various ways to structure ones

- Simple structure – MS-DOS
- More complex -- UNIX
- Layered
- Microkernel -Mach

Monolithic（庞大而单一）– Traditional UNIX

The original UNIX operating system: limited structuring (due to early hardware).

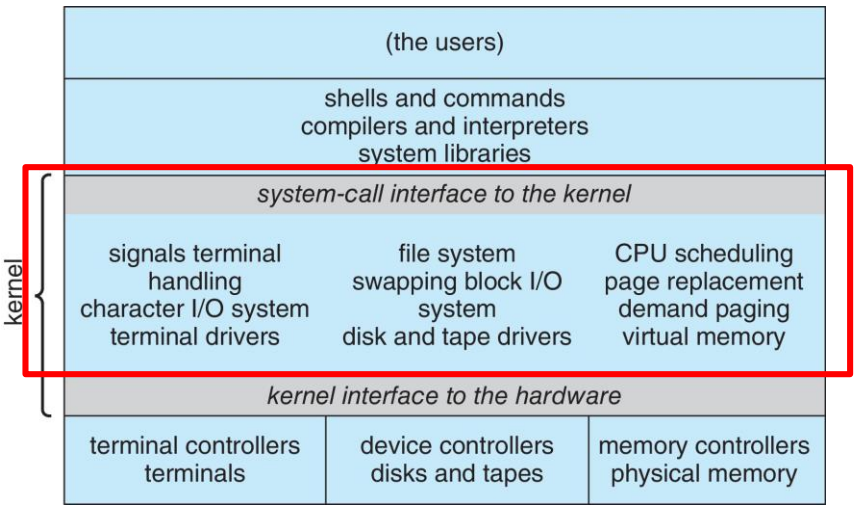
The UNIX OS consists of two separable parts

Systems programs

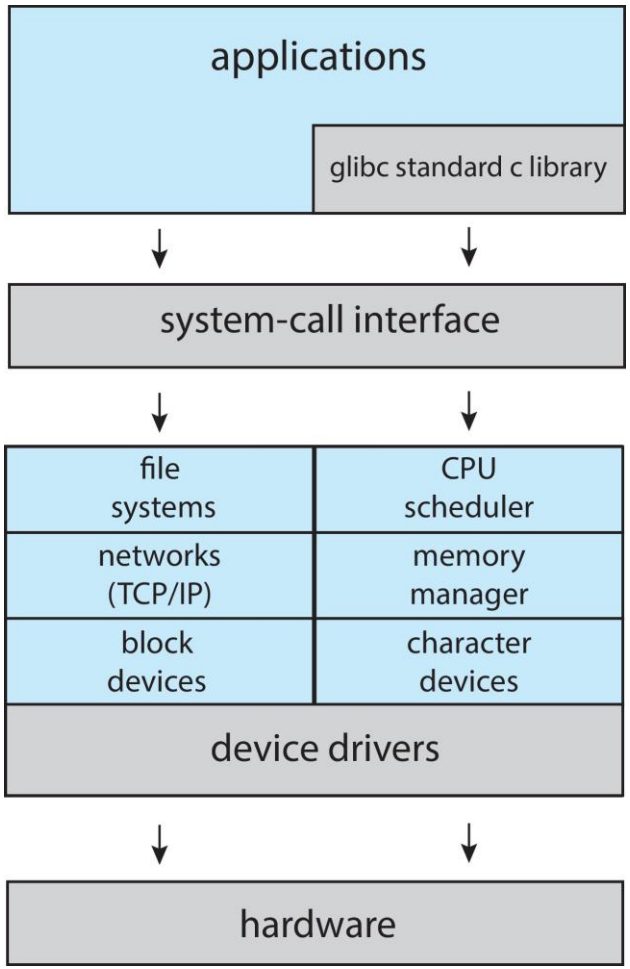
The kernel

- Consists of everything below the system-call interface and above the physical hardware
- Provides the file system, CPU scheduling, memory management, and other operating-system functions;
- Consists of a large number of functions in one level

Beyond simple but not fully layered



Monolithic Plus Modular - Linux System Structure

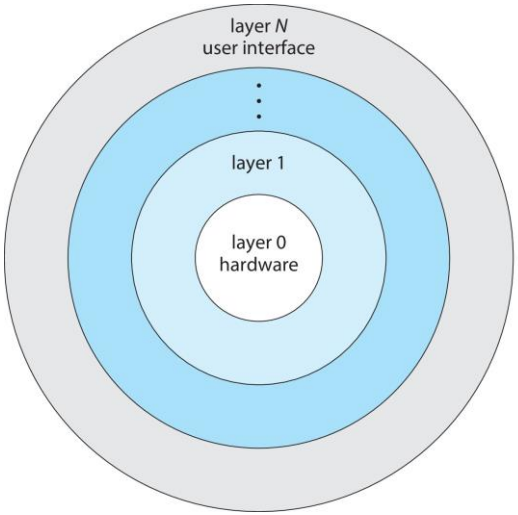


- Advantages for monolithic design
  - High speed
  - High efficiency
- Advantages for modular design
  - changes in one component affect only that component, and no others
  - Modules can be modified easily.

Layered Approach

The operating system is divided into a number of layers (levels)

- The bottom layer (layer 0), is the hardware.
- The highest (layer N) is the user interface.
- Each layer is built on top of lower layers
  - With modularity, each layer uses functions (operations) and services of only lower-level layers

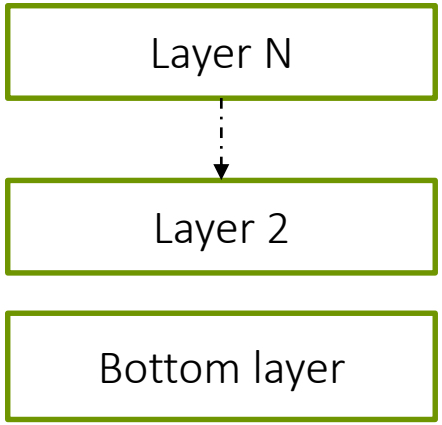


Advantage

- Simplicity of construction and debugging.

Disadvantages

- Hard to define each layer.
- Poor performance.



Microkernels (微内核)

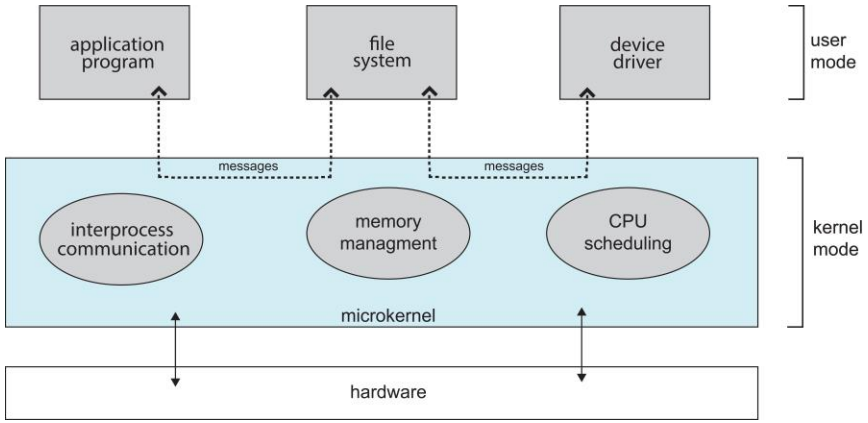
Moves as much components from the kernel into user space

E.g., Mach

Mac OS X kernel (Darwin) partly based on Mach

Communication takes place between user modules using message passing

- Microkernels provide minimal process and memory management, in addition to a communication facility.
- Communication between user modules through message passing.



[1]Mach is a kernel developed at Carnegie Mellon University to support operating system research, primarily distributed and parallel computing. Mach is often mentioned as one of the earliest examples of a microkernel.

Microkernels (微内核)

Benefits

Easier to

- extend to a microkernel
- port the operating system to new architectures

More reliable (less code is running in kernel mode)

More secure

Detriments/drawbacks

overhead of communication between user space and kernel space

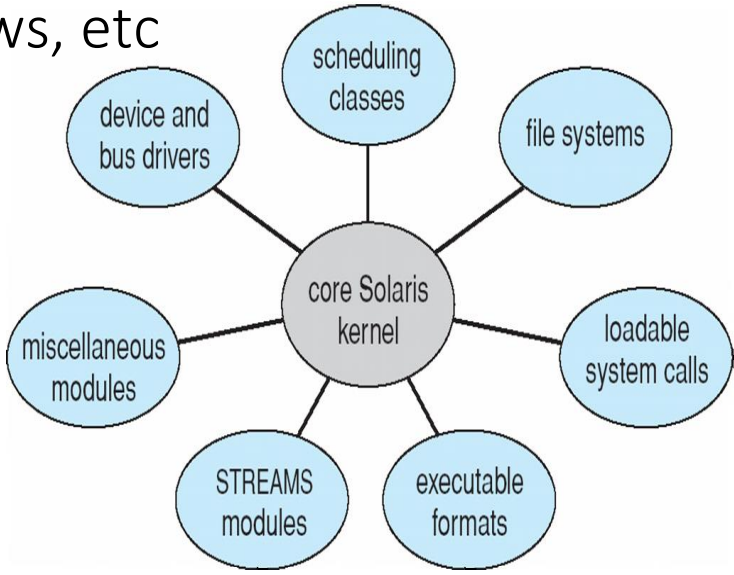
Modules

Many modern operating systems implement loadable kernel modules (best practice)

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

Overall, similar to layers but with more flexible

- E.g., Linux, Solaris[1], macOS, Windows, etc



[1]Solaris is a Unix operating system originally developed by Sun Microsystems. It superseded their earlier SunOS in 1993. **Oracle Solaris**, so named as of 2010, has been owned by Oracle Corporation since the Sun acquisition by Oracle in January 2010.

Hybrid Systems

Most modern operating systems: not one pure model (structure)

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

For example

Linux and Solaris kernels: monolithic (in kernel memory), plus modular (for dynamic loading of functionality)

Windows mostly monolithic, plus microkernel for different subsystem *personalities,* also provide support for dynamically loadable kernel modules.

Apple Mac OS X, Microkernel plus layered, Aqua[1] (GUI) plus Cocoa[2] (API) programming environment

[1]Aqua is the graphical user interface (GUI) and visual theme of Apple's macOS operating system. It was originally based around the theme of water, with droplet-like components and a liberal use of reflection effects and translucency. Its goal is to "incorporate color, depth, translucence, and complex textures into a visually appealing interface" in macOS applications. At its introduction, Steve Jobs noted that "one of the design goals was when you saw it you wanted to lick it".

[2] **Cocoa** is Apple's native object-oriented application programming interface (API) for their operating system macOS.

Examples: Apple macOS and iOS

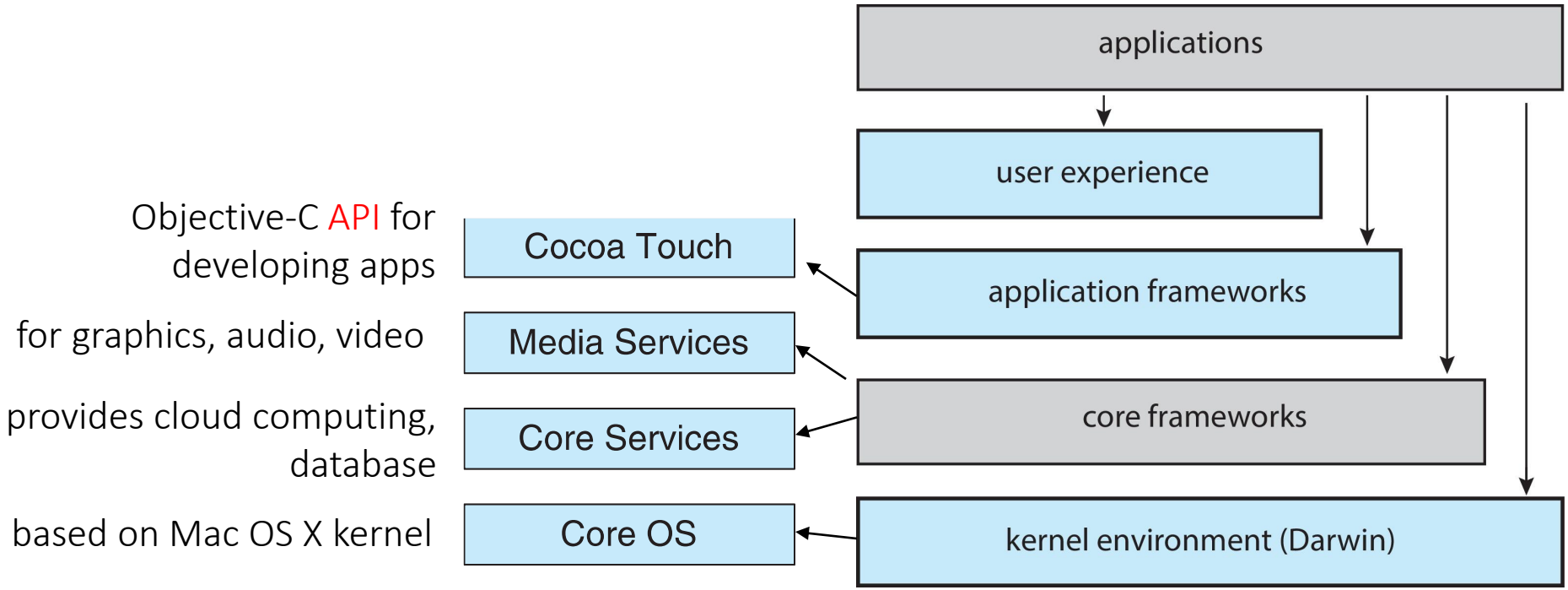
macOS: for desktop computers (CPU: Intel)

iOS : for mobile phone (CPU: ARM)

i.e., iPhone, iPad

Structured on Mac OS X, added functionality

Does not run OS X applications natively



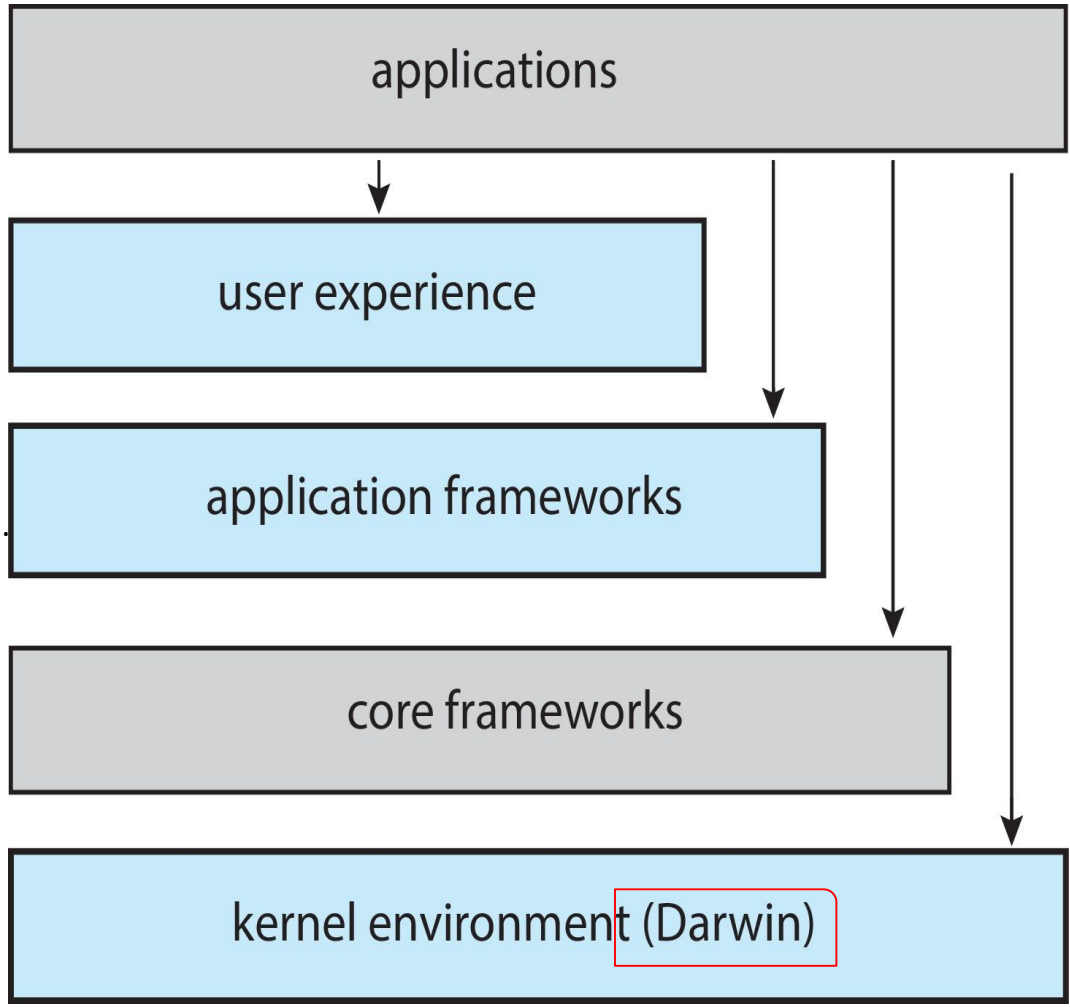
Examples: Apple macOS and iOS Structure

defines the software interface that allows users to interact with the computing devices.

includes the *Cocoa* and *Cocoa Touch* frameworks, which provide an API for the Objective-C and Swift programming languages.

defines frameworks that support graphics and media including, Quicktime and OpenGL.

Includes the Mach microkernel and the BSD UNIX kernel.



See next page

Examples: Apple macOS and iOS Structure - Darwin

Released as open source already

Darwin provides two system-call interfaces

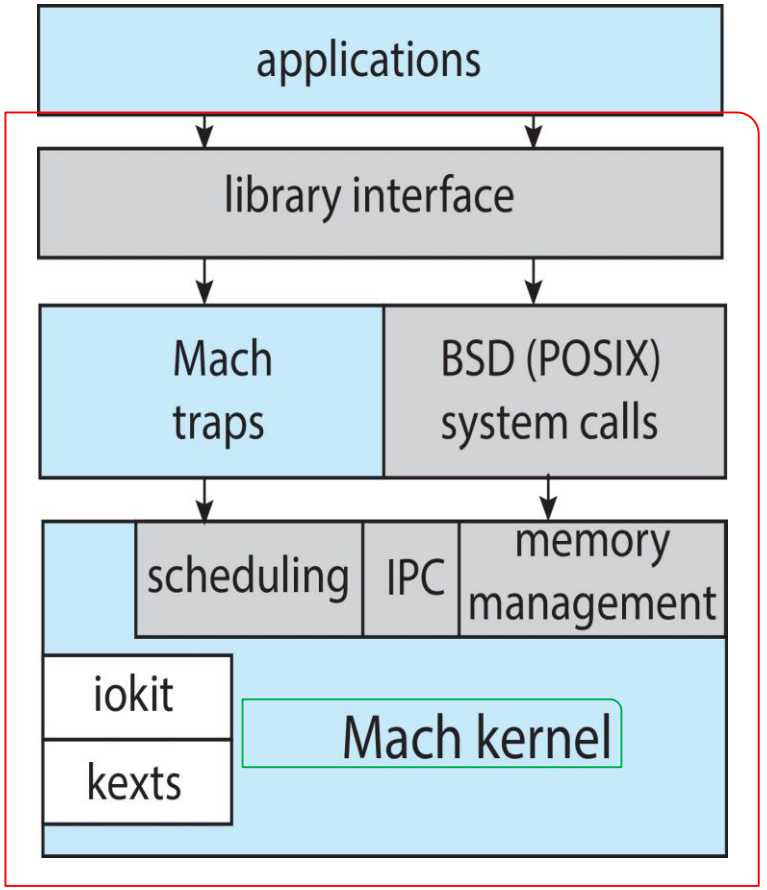
- Mach system calls and BSD system calls

Iokit

- development of device drivers

Kexts

- dynamically loadable modules



Examples: Android

Android

- Developed by Open Handset Alliance (mostly Google)
- Developed for Android smartphones and tablet computers
- Open-sourced (iOS is close-sourced)
- Similar stack to iOS
  - Based on Linux kernel but modified
- Provides process, memory, device-driver management
- Adds power management

RunTime Environment (RTE) includes core set of libraries and Dalvik virtual machine

- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc(c library in linux)
- Dalvik virtual machine

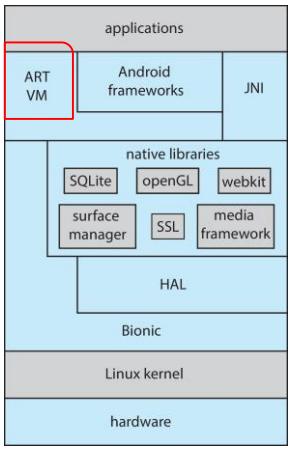
Examples: Android - Dalvik Virtual Machine

Dalvik virtual machine (cont.)

- Apps are developed in Java plus Android API
- Java code is compiled to Java bytecode (as usual) then translated to Dalvik bytecode (for historical reasons), later translated to binary ARM microprocessor instructions when you install the app on the mobile phone.
  - .java -> .class -> Dalvik executable (.dex file)-> binary instructions

ART VM[1] is replacing Dalvik VM

- Dalvik: slow



[1] ART (Android RunTime): a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities.

Operating System Boot

When power is initialized on system, execution starts at a fixed memory location

Operating system must be made available to hardware so hardware can start it

One step process

- Small piece of code – bootstrap loader（引导程序）, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it

two-step process

- ROM code（BIOS）loads boot block (with bootstrap loader) in hard disk
- Bootstrap loader loads kernel

Load An Operating System

Bootstrap loader (引导程序) – simple code to initialize the system

- Load Kernel
- Starts system daemons (守护进程, services provided outside of the kernel)

Comparison of Boot Process in Mac, Windows, and Linux

OS → / Features ↓	Windows	Linux	Mac
BIOS	Yes	Yes	No
POST	Yes	Yes	Yes
Boot Loader	NLTDR	GRUB or LiLo	BootX or boot.efi
Kernel	NTOSKERNEL	INIT , initrd	mach_init , launchd
Supporting Files	win.sys, HAL.DLL, system.ini,sysedit.exe,config.exe, autoexec.BAT, MSCONFIG.exe	/sbin//init, /etc/inittb /etc/rc.local , /runlevel	mkextcache, launchd, loginwindow , /System/Library/CoreServices

More Interest: Want to Write an OS by Yourself?

If generating an operating system from scratch

1. Write the operating system 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

More Interest: Want to Write an OS by Yourself?

If generating an operating system from an existing system

1. Download the Linux source code from http://www.kernel.org.
2. Configure the kernel using the "**make menuconfig**" command. This step generates the .config configuration file.
3. Compile the kernel using the "**make**" command. The make command compiles the kernel based on the configuration parameters identified in the .config file, producing the file **vmlinuz**, the kernel image.
4. Compile the kernel modules using the "**make modules**" command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the .config file.
5. Use the command "**sudo make modules_install**" to install the kernel modules into **vmlinuz**.
6. Install the new kernel on the system by entering the "**sudo make install**" command

End of Chapter 2