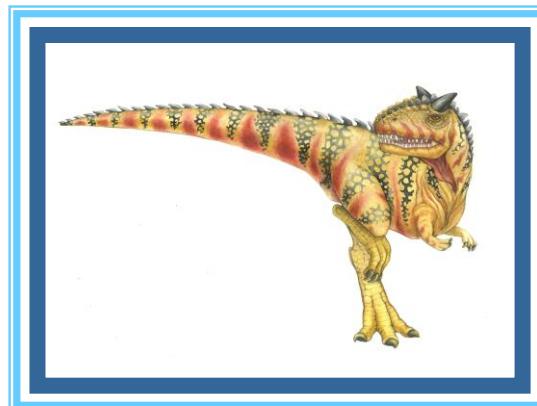
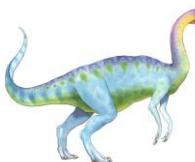


Chapter 2: Operating-System Structures





Chapter 2: Operating-System Structures

-
- Operating System Services
 - User and Operating System-Interface
 - System Calls
 - System Services
 - Linkers and Loaders
 - Why Applications are Operating System Specific
 - Operating-System Design and Implementation
 - Operating System Structure
 - Building and Booting an Operating System
 - Operating System Debugging





Objectives

- Identify services provided by an operating system
 - ★ Illustrate how **system calls** are used to provide operating system services
 - ★ Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
 - ★ Illustrate the process for booting an operating system
- Apply tools for monitoring operating system performance
- Design and implement kernel modules for interacting with a Linux kernel





Operating System Services

2 types!

#1

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**). → make user use OS easily
 - ▶ Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen**, **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) main function of OS
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device





Operating System Services (Cont.)

1

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

- **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





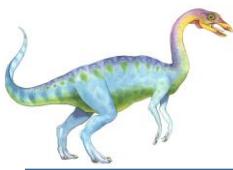
Operating System Services (Cont.)

#2

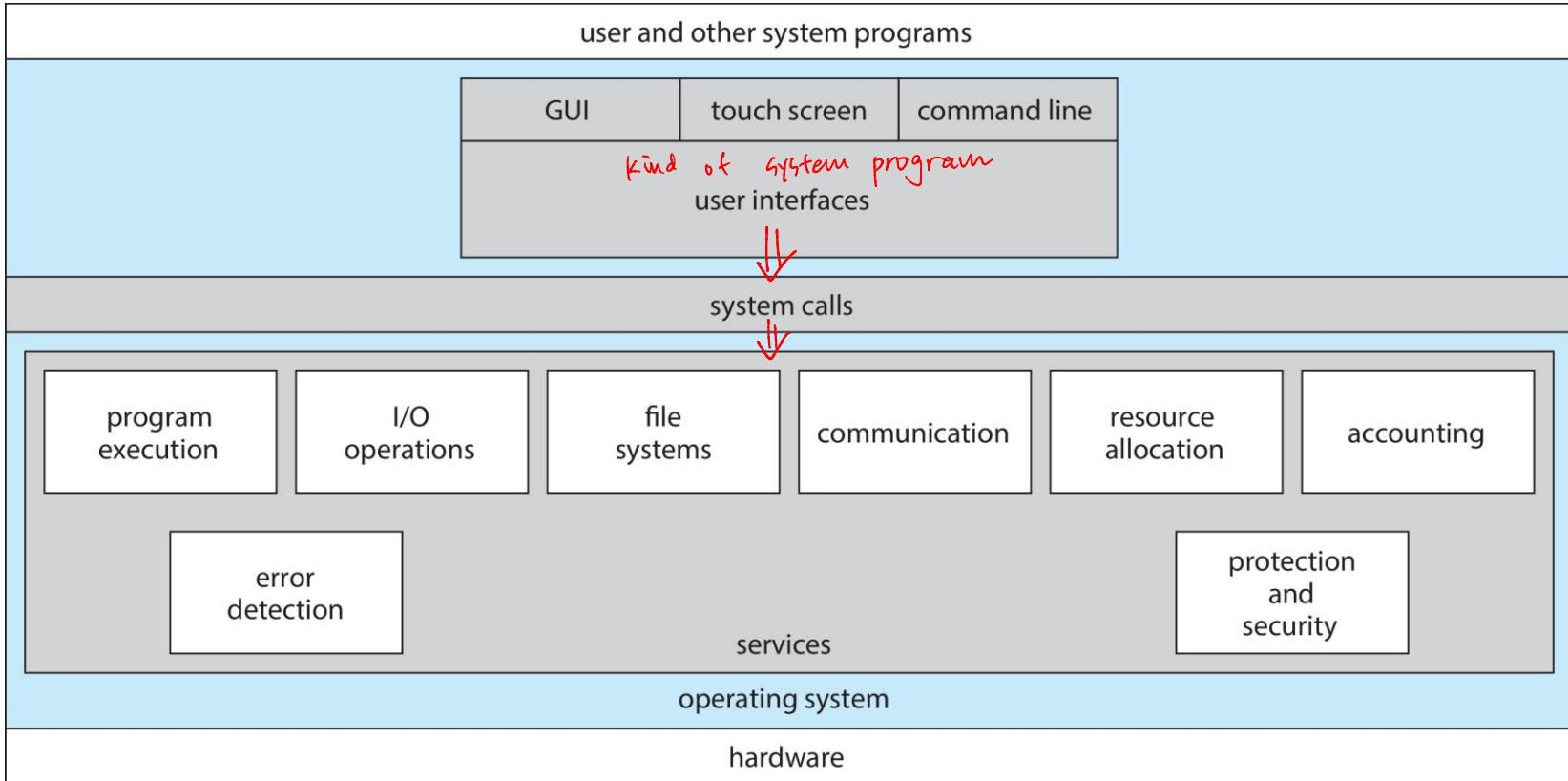
Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ▶ Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Logging** - To keep track of which users use how much and what kinds of computer resources
- **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





A View of Operating System Services





User Operating System Interface - CLI

CLI or **command interpreter** allows direct command entry

Sometimes implemented in kernel, sometimes by system program

↳ on application layer

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





Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u... ● ⌘1 X ssh ⚡ ⌘2 X root@r6181-d5-us01... ⌘3

Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbgs$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
06:57:48 up 16 days, 10:52, 3 users, load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs           127G  520K  127G   1% /dev/shm
/dev/sda1        477M   71M   381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                  12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test   23T  1.1T  22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0     0     0 ?      S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0     0     0 ?      S    Jul12 177:42 [vpthread-1-2]
root      3829  3.0  0.0     0     0 ?      S    Jun27 730:04 [rp_thread 7:0]
root      3826  3.0  0.0     0     0 ?      S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```





User Operating System Interface - GUI

User-friendly **desktop** metaphor interface

Usually mouse, keyboard, and monitor

Icons represent files, programs, actions, etc

Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)

Invented at Xerox PARC

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

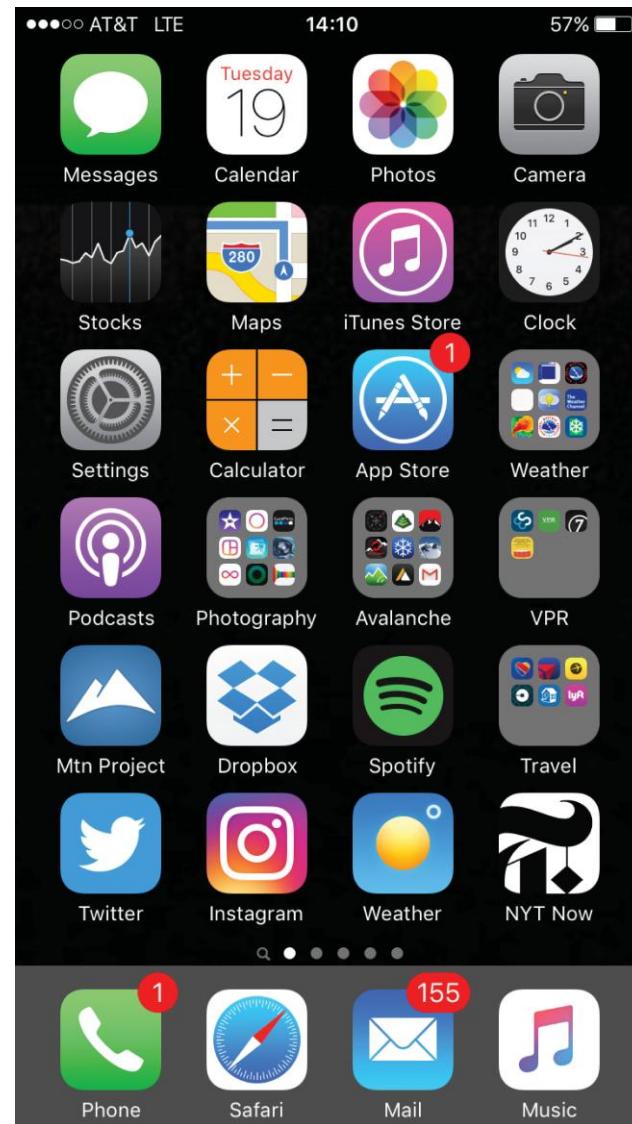
Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)





Touchscreen Interfaces

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





The Mac OS X GUI

The screenshot displays a Mac OS X desktop environment with several windows open:

- Finder:** Shows a list of files and folders in a directory tree. The current path is `pbg > imp > book > os10c-dir > text-dir > ch2 > 2-fig`. The list includes various files like `2.pdf`, `2.fig`, `2-04.eps`, and `2-04.pdf`.
- Dictionary:** A window titled "Dictionary Version: 2.2.1" showing search results for "operating system". It lists definitions from "Shorter Oxford English Dictionary", "Merriam-Webster", and "The Sleeping Dictionary".
- TextEdit:** A window titled "PFS summary v3" containing a presentation slide about DSSD High Performance Parallel File Systems.
- TextEdit:** Another window showing a table of mathematical functions.
- File Manager:** A window titled "apple Computer Inc." displaying a file structure and details for a file named "apple Computer Inc."





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**

Programming Interface (API) rather than direct system call

use *↳ including system call*

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)

Note that the system-call names used throughout this text are generic

```
import java.util.Scanner;  
....  
Scanner scanner = new Scanner(System.in);  
String userinput = scanner.nextLine();  
System.out.println(userinput);  
....
```

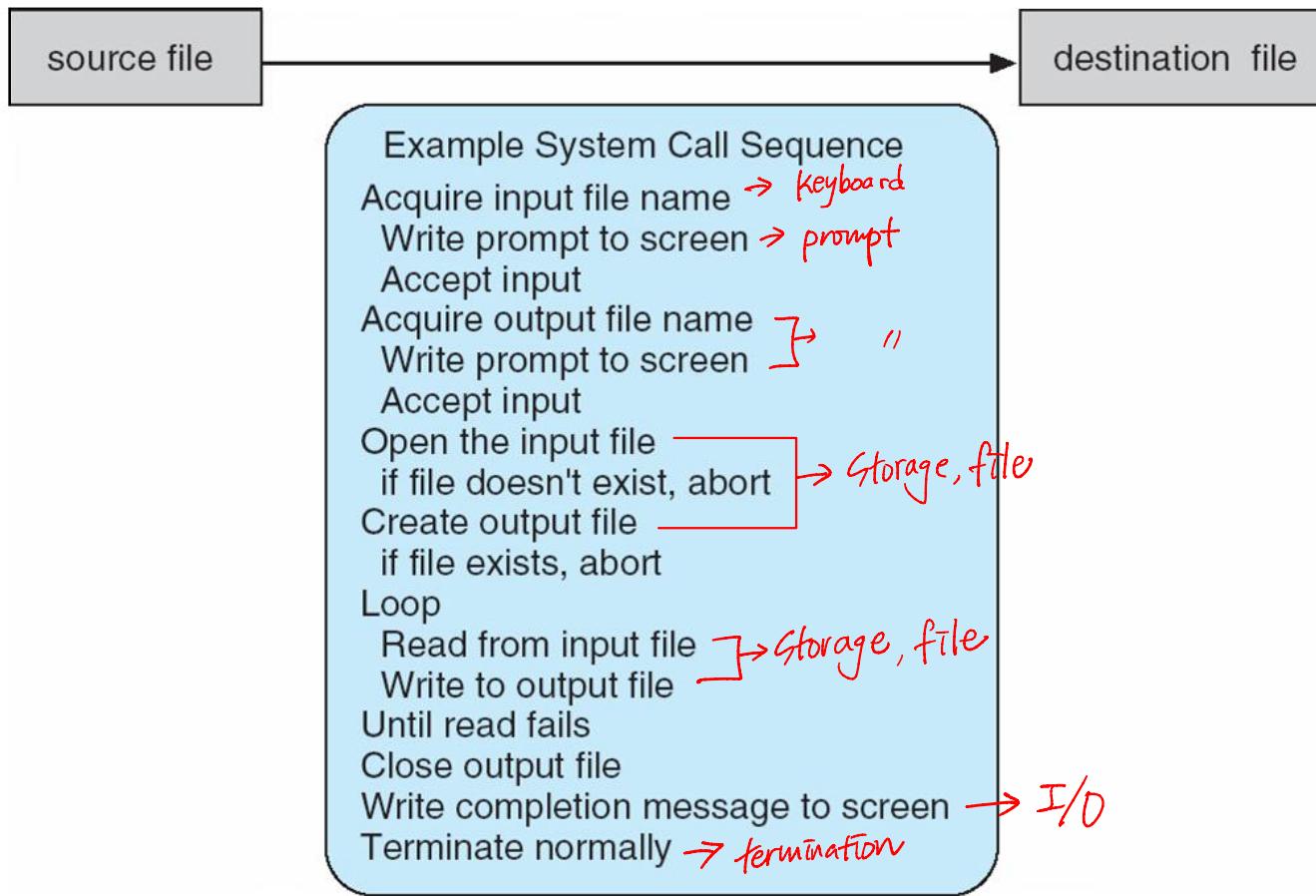




Example of System Calls



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





Java code

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;
public class FileCopyExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // Name of Source file
        String sourceFileName = scanner.nextLine();
        Path sourcePath = Paths.get(sourceFileName);
        String destinationFileName = scanner.nextLine(); // Name of Destination file
        Path destinationPath = Paths.get(destinationFileName);
        try { // Execution of file copying
            Files.copy(sourcePath, destinationPath);
            System.out.println("File copied successfully.");
        } catch (Exception e) {
            System.out.println("Error occurred while copying the file.");
            e.printStackTrace();
        }
        finally {
            scanner.close();
        }
    }
}
```





Example of Standard API

encapsulation of system call

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)
```

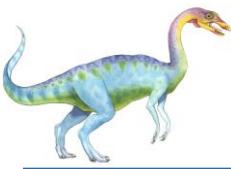
return value function name parameters

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:

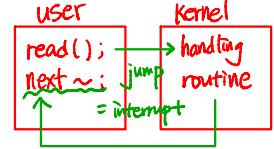
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

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.





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

:: encapsulation -

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

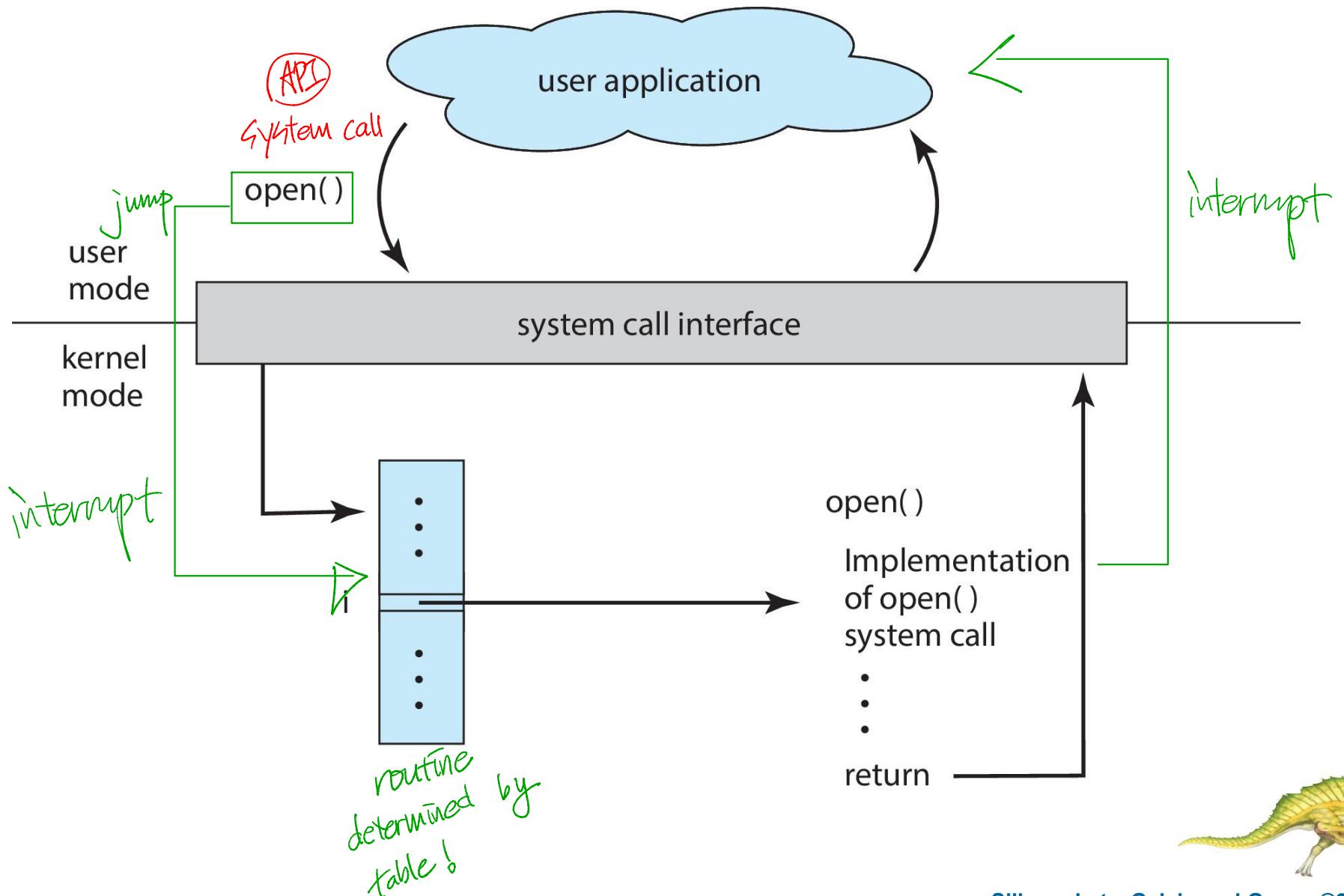
Most details of OS interface hidden from programmer by API

- ▶ Managed by run-time support library (set of functions built into libraries included with compiler)





API – System Call – OS Relationship





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

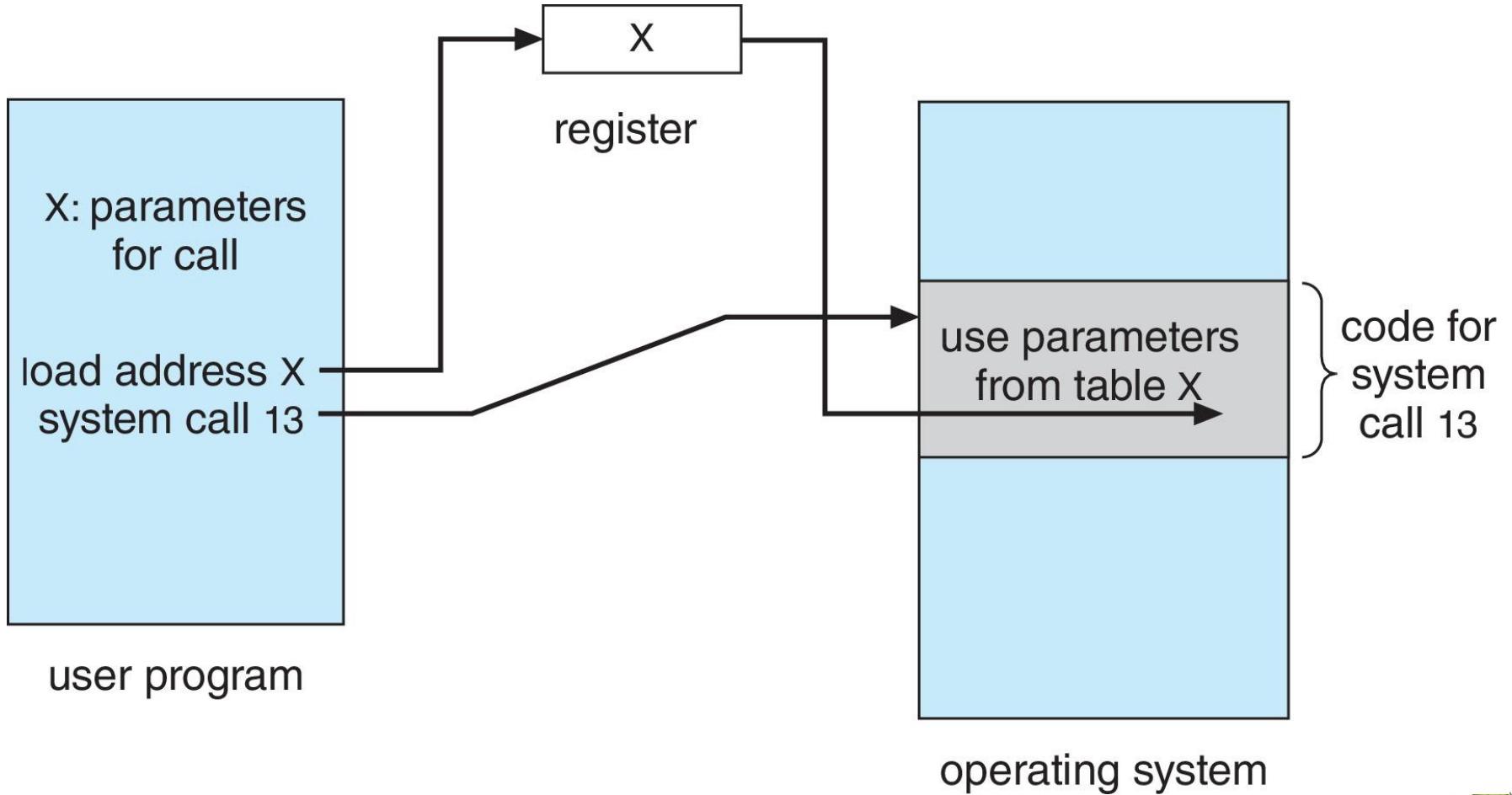
- o Simplest: pass the parameters in registers ... small size, quite precious resource.
 - ▶ In some cases, may be more parameters than registers
- o Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register ... conventional
 - ▶ This approach taken by Linux and Solaris
- o Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system ↗ MEM acts like Stack!

Block and stack methods do not limit the number or length of parameters being passed





Parameter Passing via Table (block)





Types of System Calls

unit of resource allocation.

#1

Process control *make user program working*

create process, terminate process

end, abort

load, execute

get process attributes, set process attributes

wait for time *stop for a while*

wait event, signal event = *make other process wait / notify occurrence of some events to other process*

allocate and free memory

Dump memory if error

Debugger for determining **bugs**, **single step execution** *figure out bugs*

Locks for managing access to shared data between processes





Types of System Calls (cont.)

#2

File management

- create file, delete file
- open, close file *available Random-access supported files.*
file pointer : current location of read/write of file.
- read, write, reposition
- get and set file attributes

#3

Device management

- request device, release device
- read, write, reposition *only for Random Access Device* ... *data from device*
- get device attributes, set device attributes
- logically attach or detach devices





Types of System Calls (Cont.)

#4

Information maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

#5

Communications

create, delete communication connection

send, receive messages if **message passing model** to host name
or **process name**

► From **client** to **server**



available to send msg to remote host
with receiver's ID.
via kernel. slow transfer data
no sync issue reliable by system call

Shared-memory model create and gain access to memory regions

transfer status information

different processes in same system,
they share region of memory to communicate

attach and detach remote devices

fast.
Sync issue
massive data





Types of System Calls (Cont.)

#6

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.

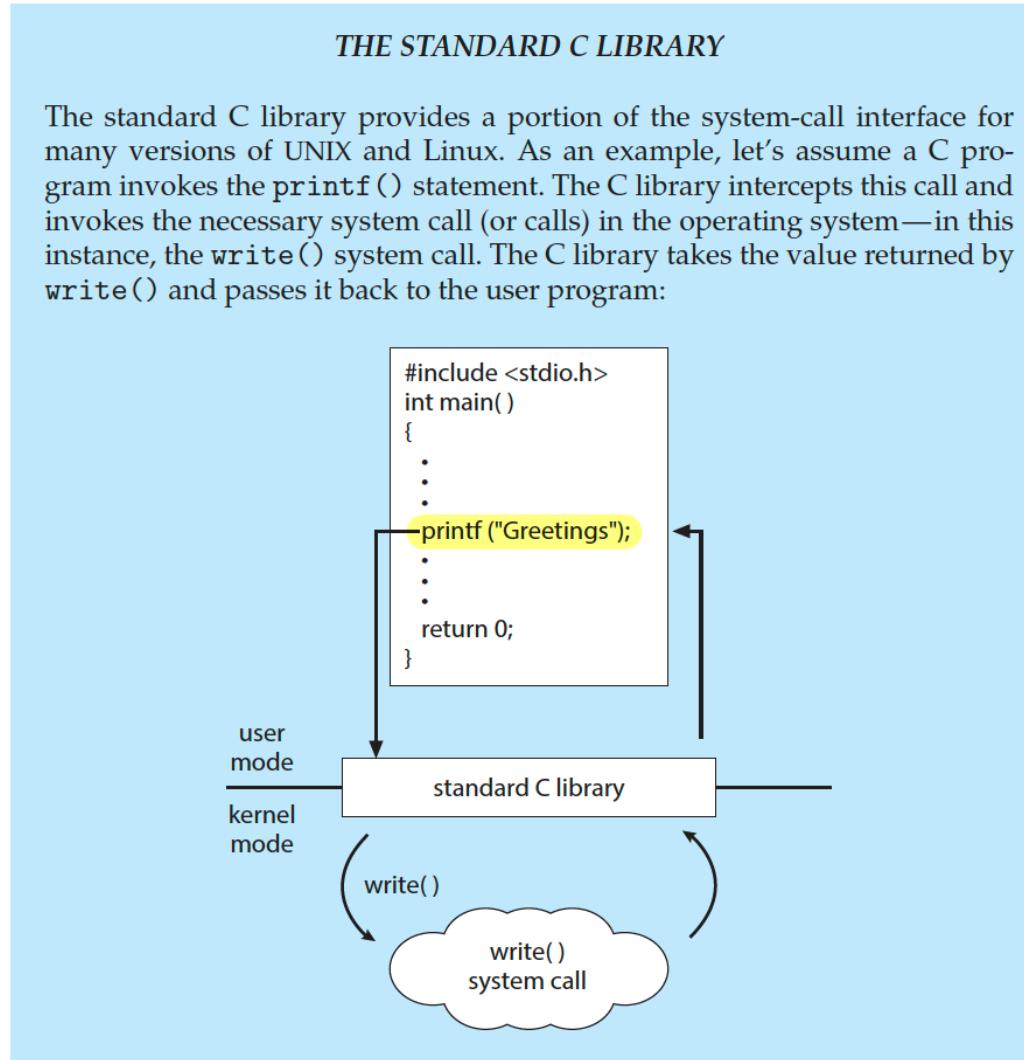
	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance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





Standard C Library Example

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





Example: Arduino \doteq MS DOS

Single-tasking

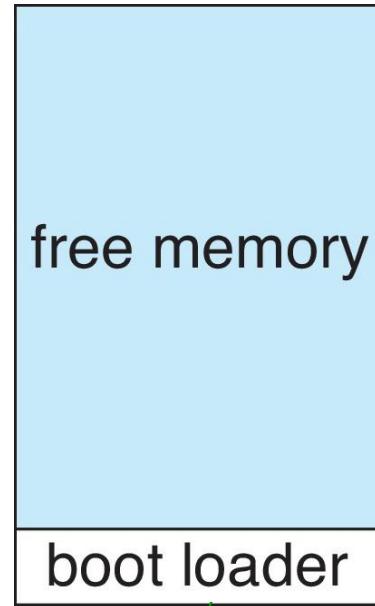
instead
of

No operating system \Rightarrow user program directly access MEM. < efficient HW modifying
Programs (sketch) loaded via
USB into flash memory

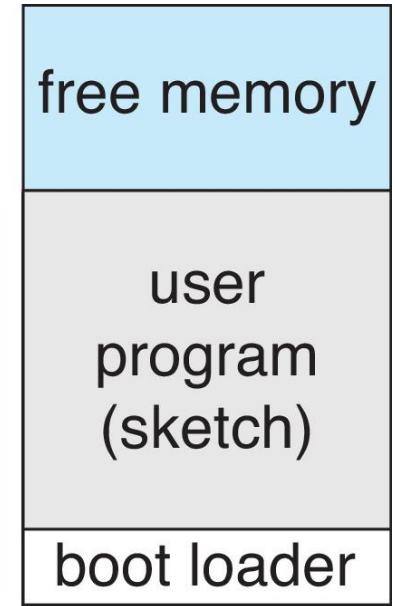
Single memory space

Boot loader loads program

Program exit -> shell reloaded



\doteq command line interpreter
(a)



(b)

At system startup

running a program





Example: FreeBSD

Unix variant

Multitasking *multi processing*

User login -> invoke user's choice of shell

Shell executes `fork()` system call to create process

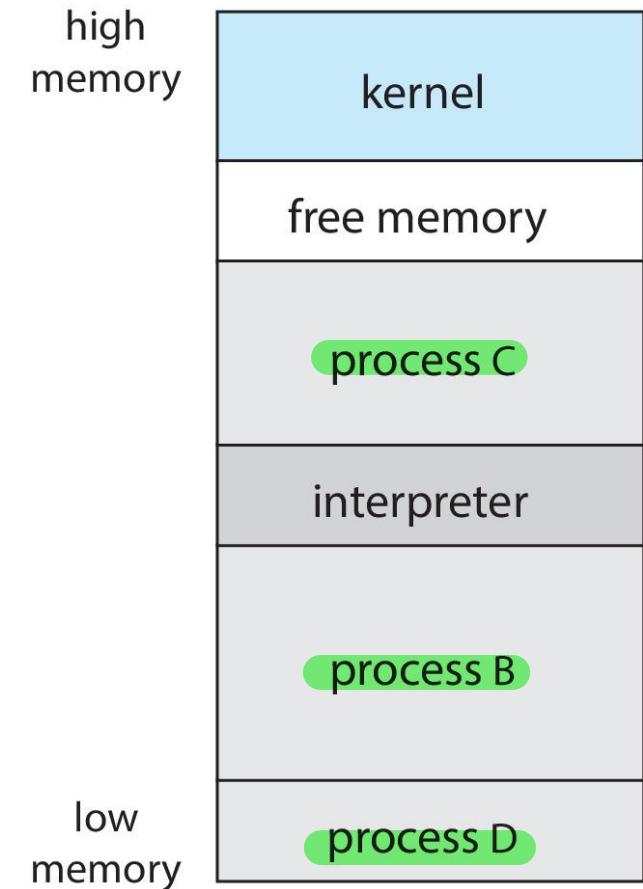
Executes `exec()` to load program into process

Shell waits for process to terminate or continues with user commands

Process exits with:

`code = 0` – no error

`code > 0` – error code





System Services

System Program includes System calls routines for using resources

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 when, who, ...

Programming language support

Program loading and execution

Communications

Background services like vaccine program... rarely interact with user

Application programs

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





System Services (cont.)

IDE

Provide a convenient environment for program development and execution

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

File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

Status information

Some ask the system for info - date, time, amount of available memory, disk space, number of users

Others provide detailed performance, logging, and debugging information

Typically, these programs format and print the output to the terminal or other output devices

Some systems implement a **registry** - used to store and retrieve configuration information
Config info of installed programs.





System Services (Cont.)

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





System Services (Cont.)

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

compiler → compile src into object file.
machine-level

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 executable = comb of object files. (main + lib)

Program resides on secondary storage as binary executable

Must be brought into memory by **loader** to be executed

Relocation assigns final addresses to program parts and adjusts code and data in program to match those addresses

Modern general purpose systems don't link libraries into executables

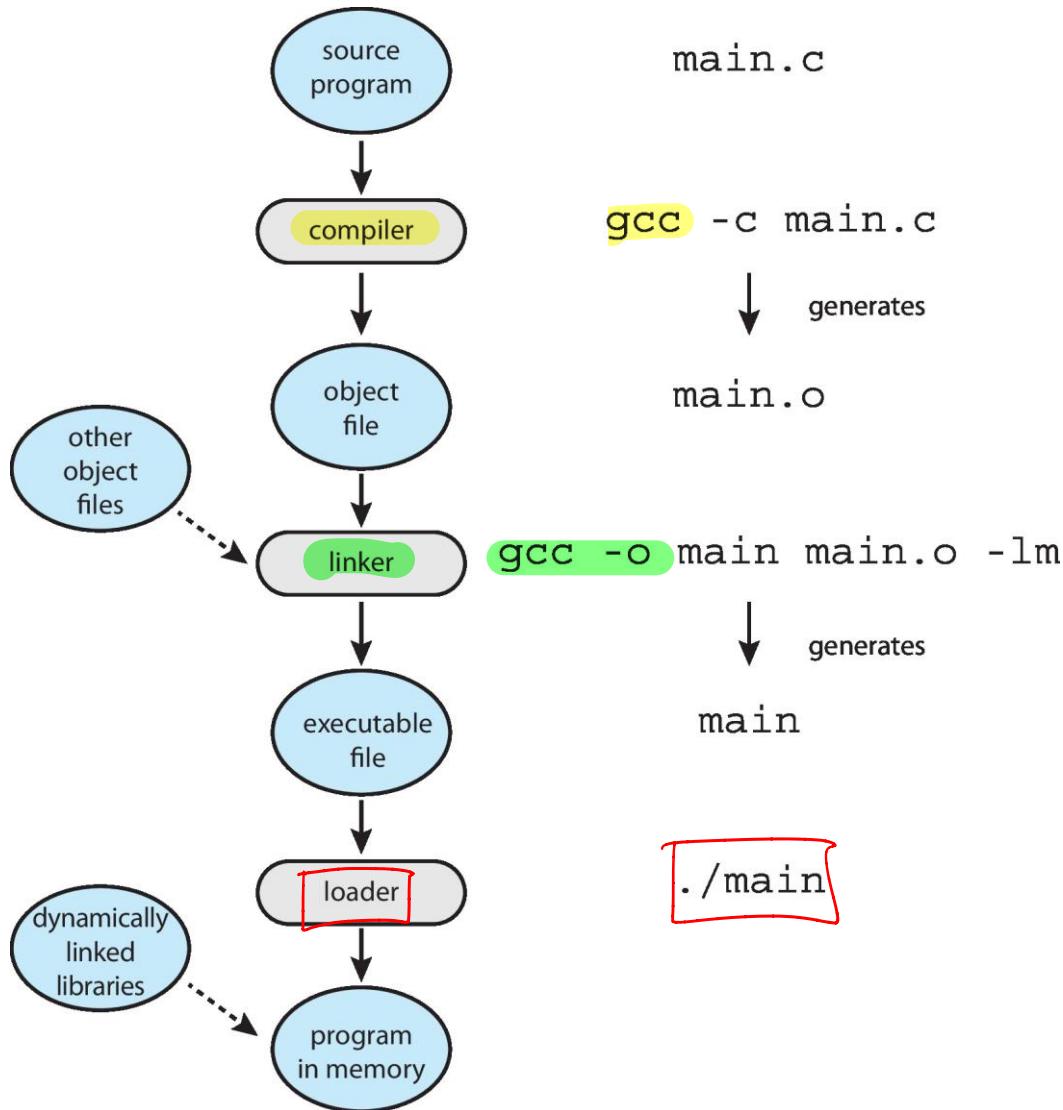
Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once) loaded if they need, shared to anyone who need them also.

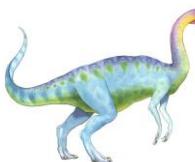
Object, executable files have standard formats, so operating system knows how to load and start them





The Role of the Linker and Loader





Why Applications are 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

+ HW architecture

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

★ **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc





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 the design 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? *How to utilize policy*

Mechanisms determine how to do something, policies decide what will be done

form of argument

independent from policy as much as possible. ⇒ more flexibility for both.

- ★ The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

Specifying and designing an OS is highly creative task of **software engineering**





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





Operating System Structure

General-purpose OS is very large program

Various ways to structure ones

- {
 - Simple structure – MS-DOS like Arduino
 - More complex -- UNIX
 - Layered – an abstraction
 - Microkernel -Mach





Monolithic Structure – Original UNIX

- ◆ UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

Systems programs *tools help user using OS, System Call → get service*
The kernel

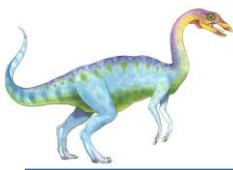
- ▶ Consists of everything below the system-call interface and above the physical hardware
- ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

*functions communicate each other by "function call"
in a single kernel module.*

*pros ... easy to communicate & efficient & fast
by just jump to other function - with parameters.*

*cons ... unreliable, ⇒ single error can make whole kernel error
hard to upgrade, ⇒ upgrade whole parts of kernel*

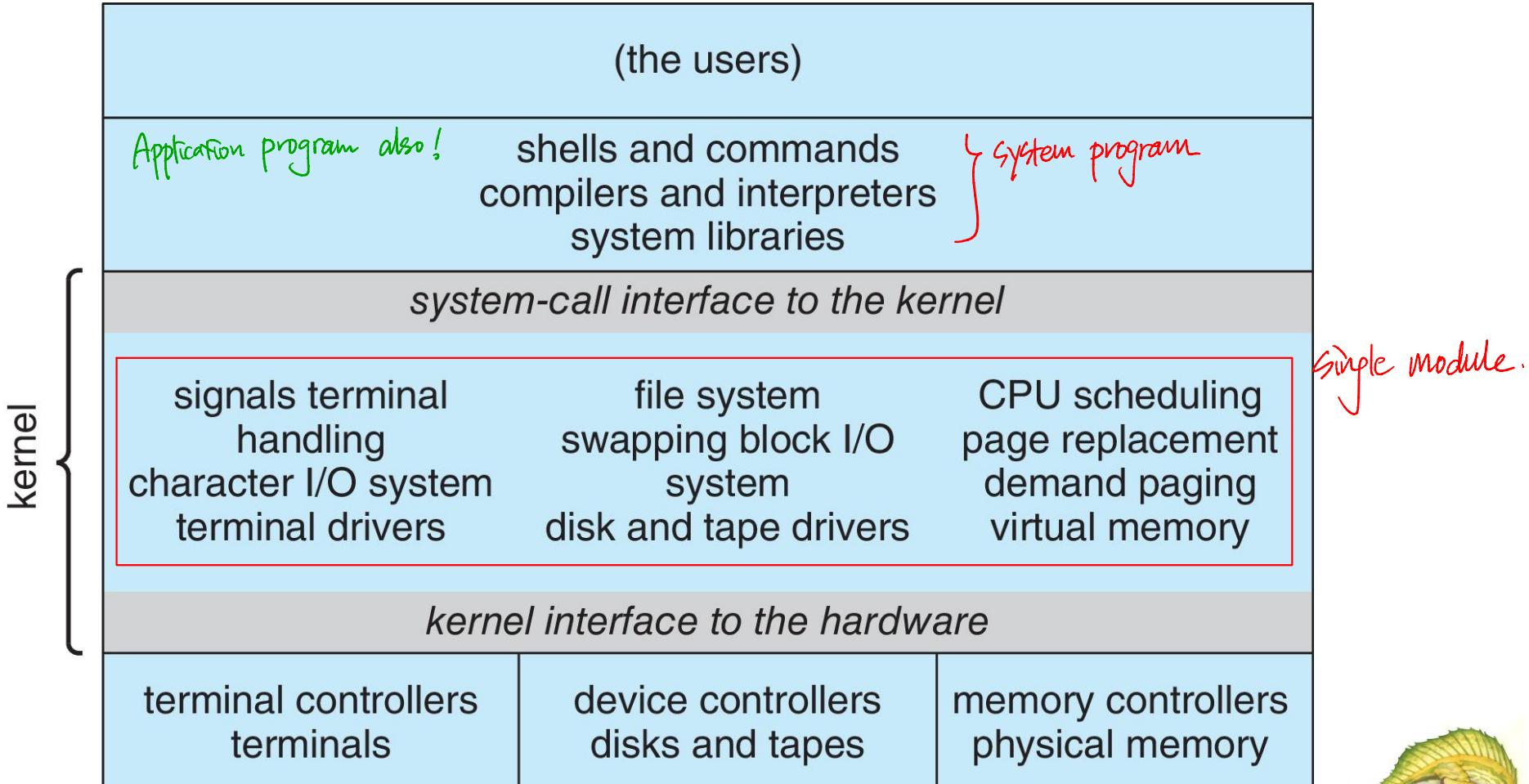




Traditional UNIX System Structure

= monolithic

Beyond simple but not fully layered



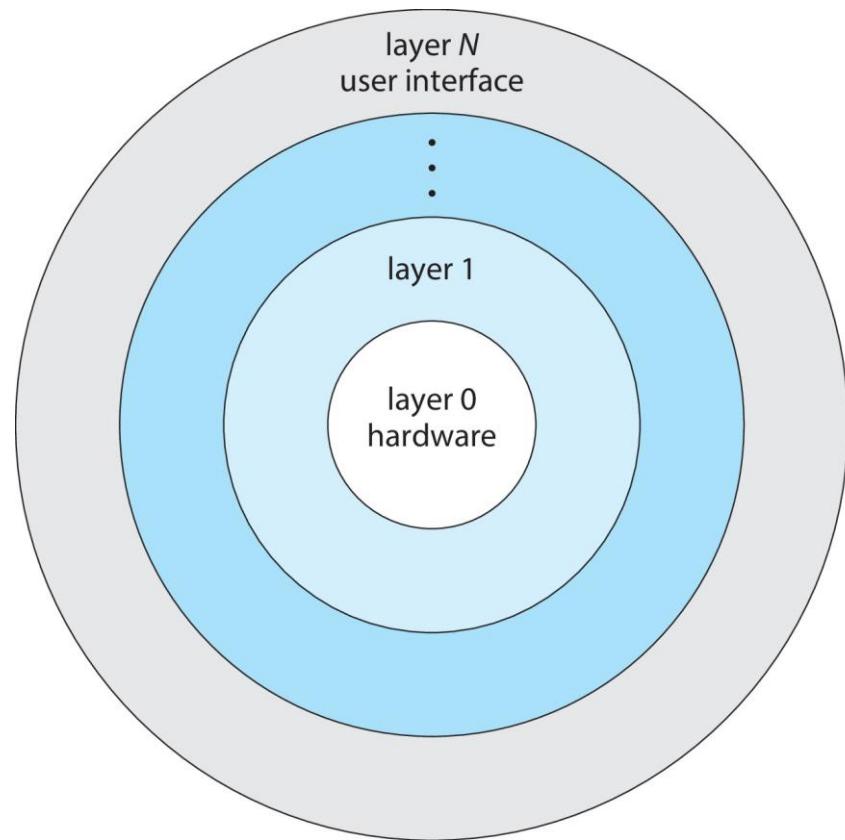


Layered Approach

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

ambiguous case to which
layer a particular function should belong to
e.g., memory management



layer i cannot affect layer $i-1$ or layer $i+1$
 \Rightarrow easy to upgrade, modify





Microkernels

Moves as much from the kernel into user space ⇒ except essential function, others are handled in user space in MEM
Mach example of **microkernel**

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

Communication takes place between user modules using **message passing**

Benefits:

Easier to extend a microkernel

Easier to port the operating system to new architectures

More reliable (less code is running in kernel mode) ⇒ small size to manage

More secure

Detrimental factors:

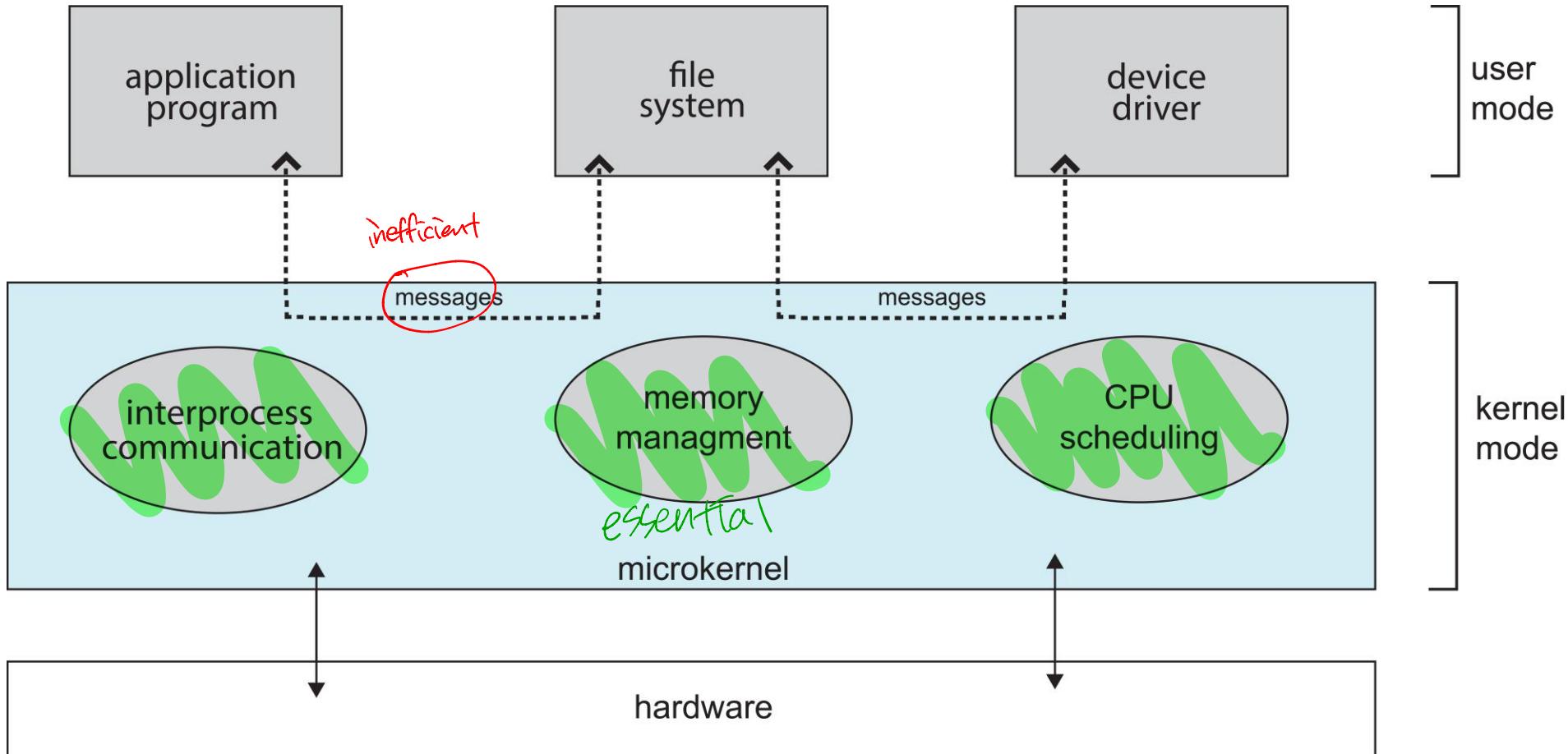
Performance overhead of user space to kernel space communication

additional, unnecessary computation
MEM usage
time spending
⋮





Microkernel System Structure





Modules

Many modern operating systems implement **loadable kernel modules (LKMs)**

Uses object-oriented approach \Rightarrow completely separated, minimal affect~~ions~~

Each core component is **separate**

Each talks to the others over known interfaces

Each is **loadable as needed within the kernel**

Overall, similar to layers but with more **flexible**

Linux, Solaris, etc

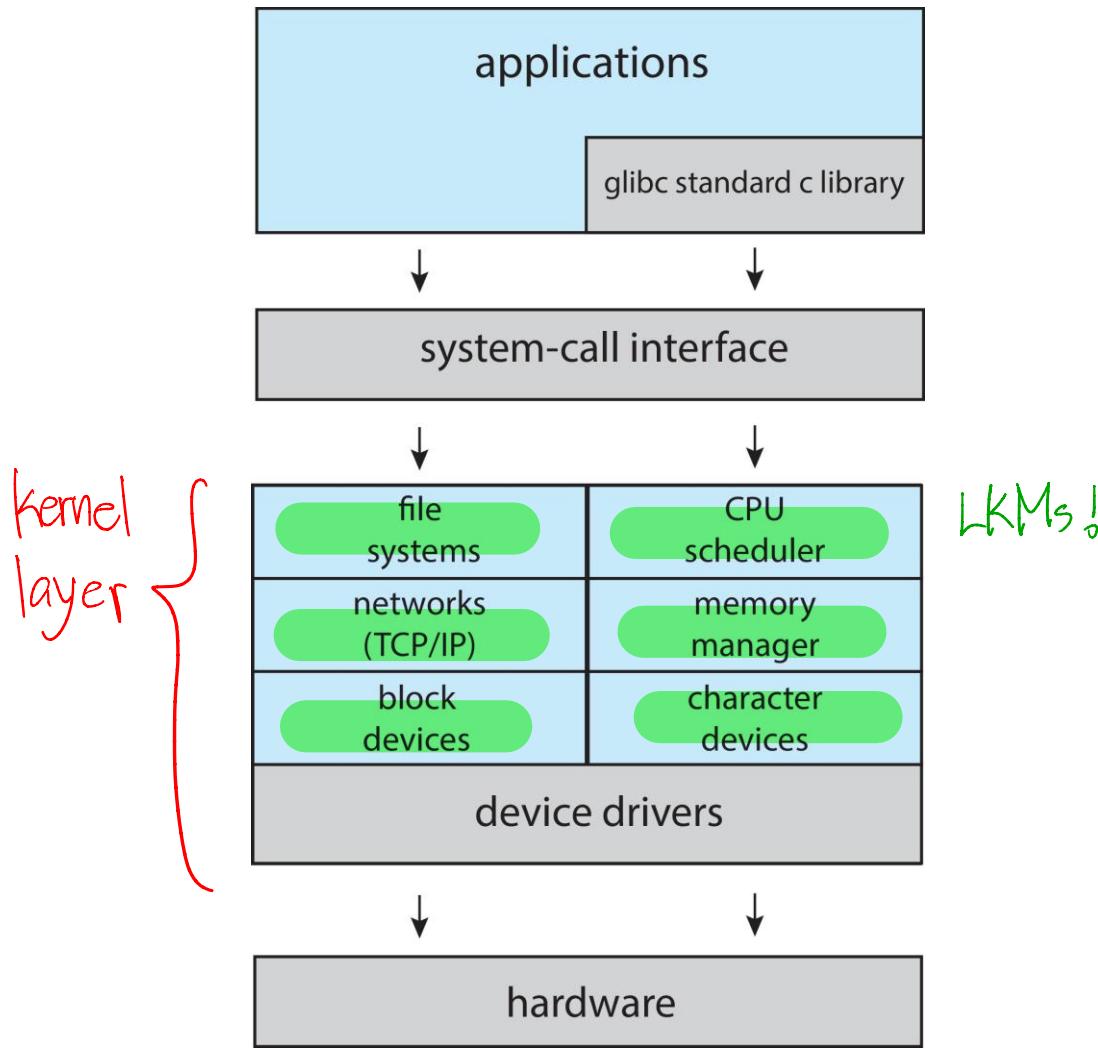
no need, no load
 \Rightarrow low cost
easy to update
easy to replace
reliability





Linux System Structure

Monolithic plus modular design





Hybrid Systems

Most modern operating systems are actually not one pure model

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

Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality

Windows mostly monolithic, plus microkernel for different subsystem *personalities* like WSL

Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment

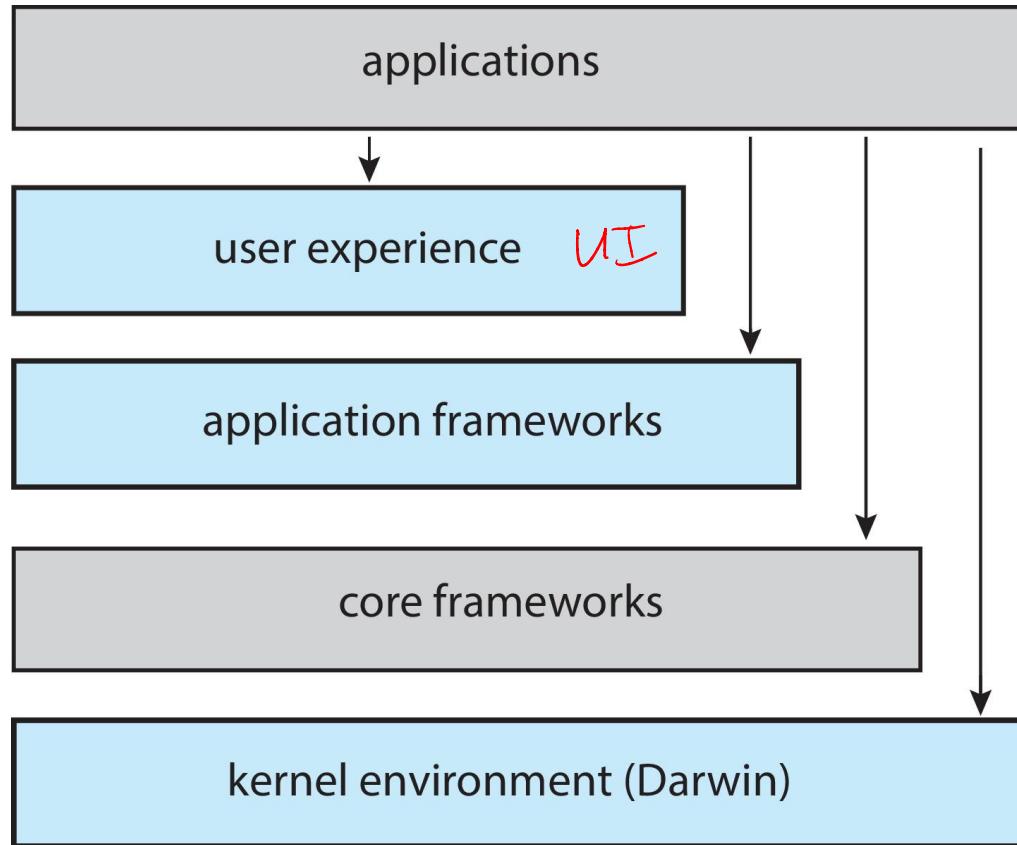
Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)





macOS and iOS Structure

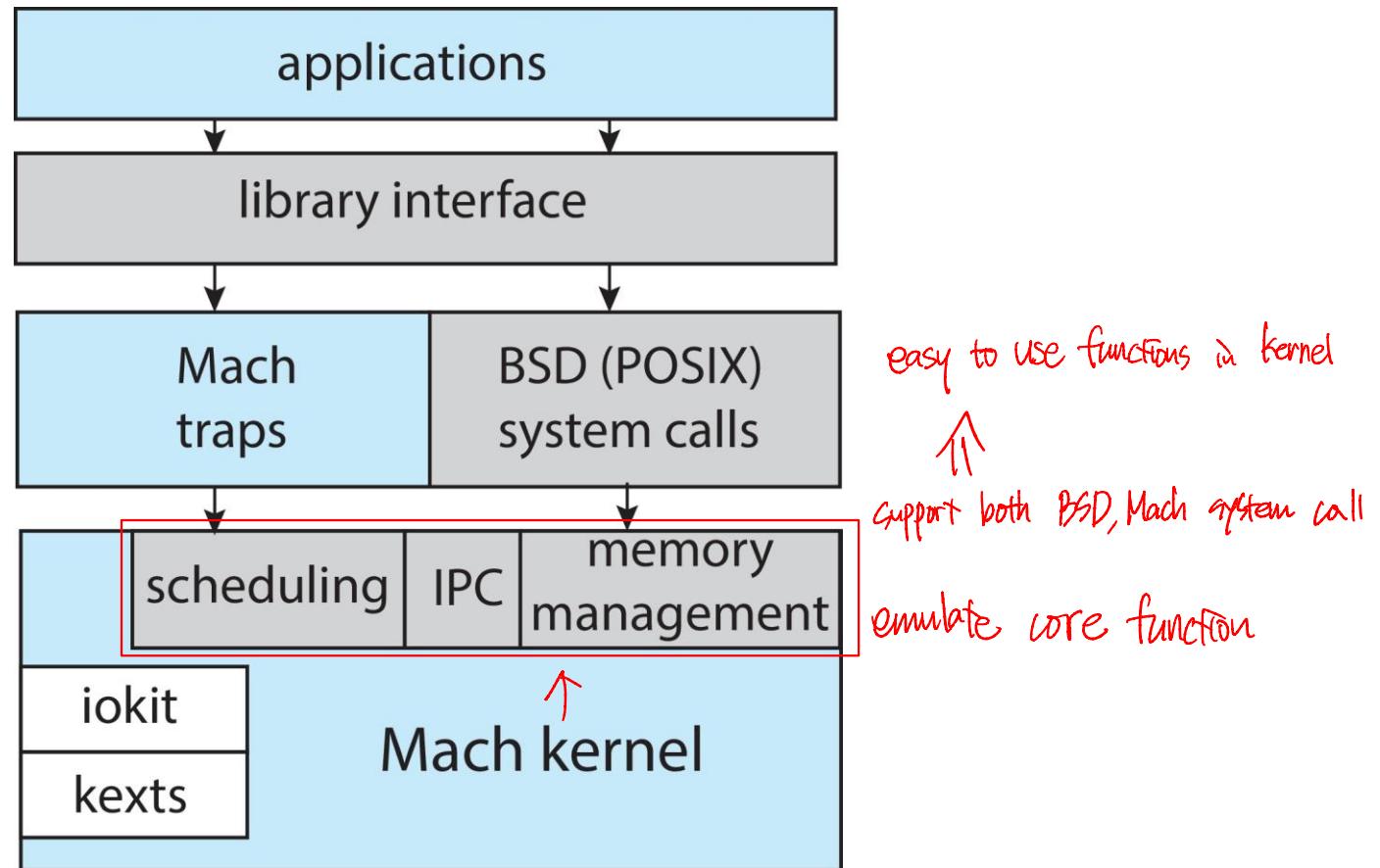
look-like layered system.





Darwin

layered





iOS

Apple mobile OS for *iPhone, iPad*

Structured on Mac OS X, added functionality

Does not run OS X applications natively

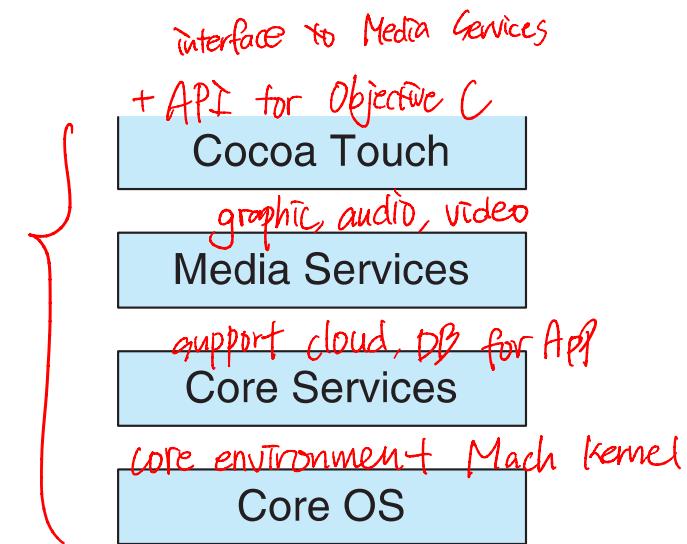
- ▶ Also runs on different CPU architecture (ARM vs. Intel)

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





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

modified JVM for power efficiency

Runtime environment includes core set of libraries and Dalvik virtual machine

Apps developed in Java plus Android API

- ▶ Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM

Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc





Android Architecture

Runtime Environment

⇒ provide all GW, resources
for executing program.

Java, Kotlin

Dalvik VM

APT compile
⇒ native code:
more efficiency

applications

ART
VM

Android
frameworks

Android Runtime VM

C/C++ native

efficiency
access kernel
native lib

JNI
Java Native
Interface

native libraries

SQLite

openGL

webkit

surface
manager

SSL

media
framework

HAL

HW Abstract Layer - flexibility

Bionic

extension of C lib

Linux kernel

hardware





Building and Booting an Operating System

Operating systems generally designed to run on a class of systems with variety of peripherals

Commonly, operating system already installed on purchased computer

But can build and install some other operating systems

If generating an operating system from scratch

- ▶ Write the operating system source code
- ▶ Configure the operating system for the system on which it will run
- ▶ Compile the operating system
- ▶ Install the operating system
- ▶ Boot the computer and its new operating system





Building and Booting Linux

Download Linux source code (<http://www.kernel.org>)

Configure kernel via “make menuconfig”

Compile the kernel using “make”

 Produces vmlinuz, the kernel image

 Compile kernel modules via “make modules”

 Install kernel modules into vmlinuz via “make modules_install”

 Install new kernel on the system via “make install”





System Boot

- 1 When power initialized on system, execution starts at a fixed memory location
- 2 Operating system must be made available to hardware so hardware can start it
 - Small piece of code – **bootstrap loader**, BIOS, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
 - Modern systems replace BIOS with **Unified Extensible Firmware Interface (UEFI)** for 64-bit
- 3 Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- 4 Kernel loads and system is then running
- 5 Boot loaders frequently allow various boot states, such as single user mode



End of Chapter 2

