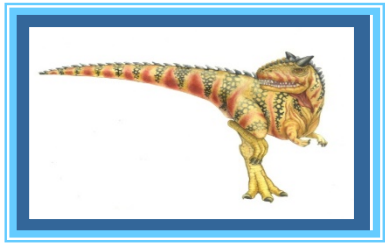
Chapter 2: Operating-System Structures



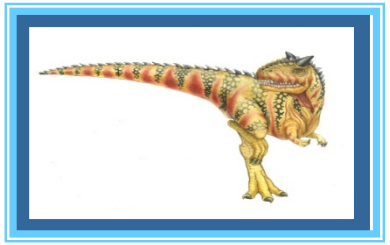
Chapter 2: Operating-System Structures

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

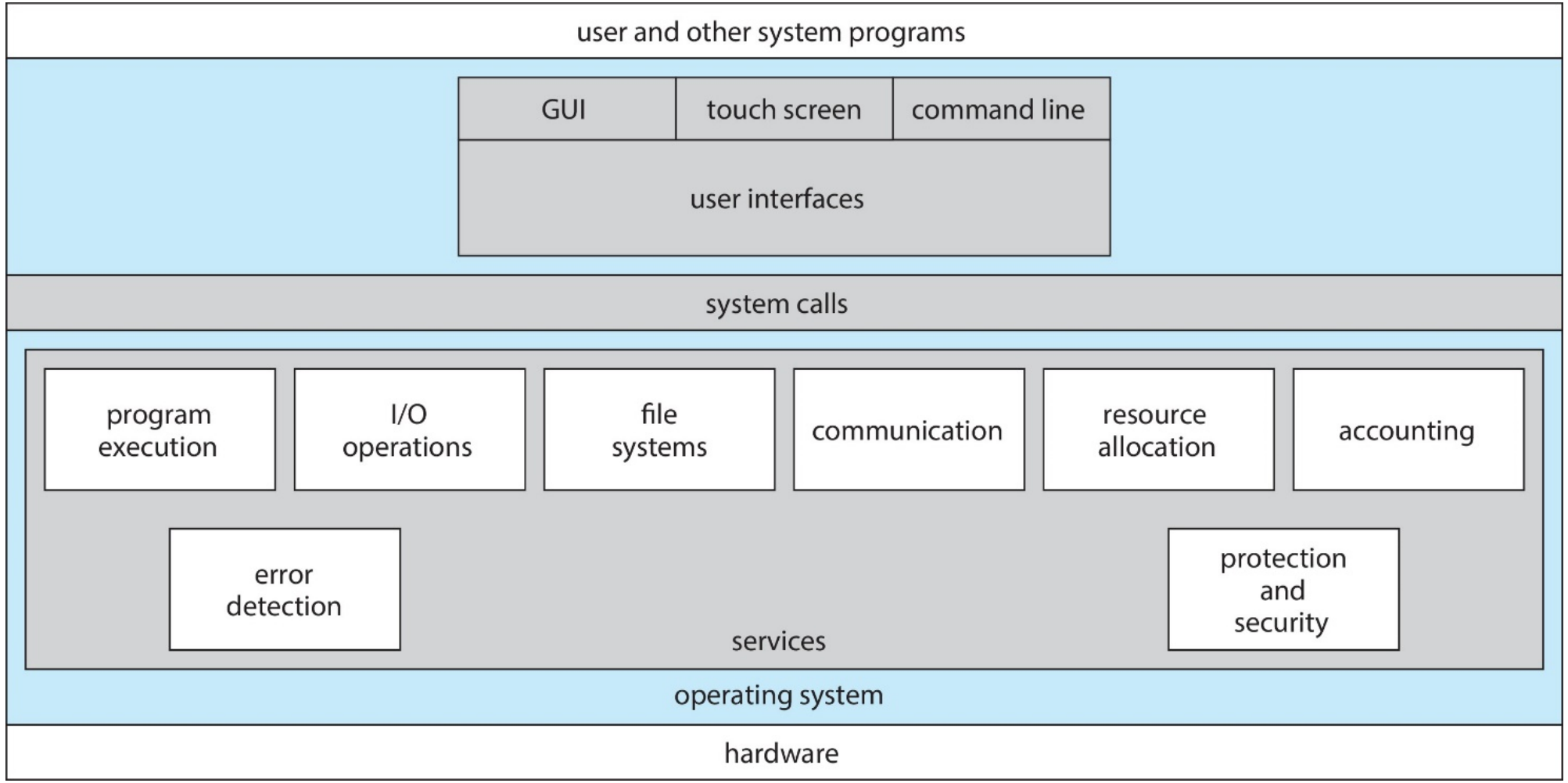
Objectives

- 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 designi ng 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.

2.1 Operating System Services



Operating System Services



A view of operating system services.

Operating System Services

- 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 Interface (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. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

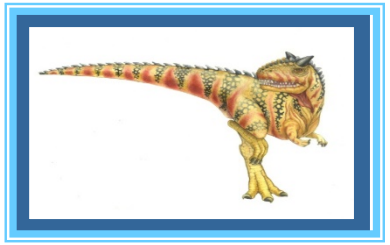
Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (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 multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles,mainmemory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - Accounting - 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
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

2.2 User Operating System Interface



User Operating System Interface

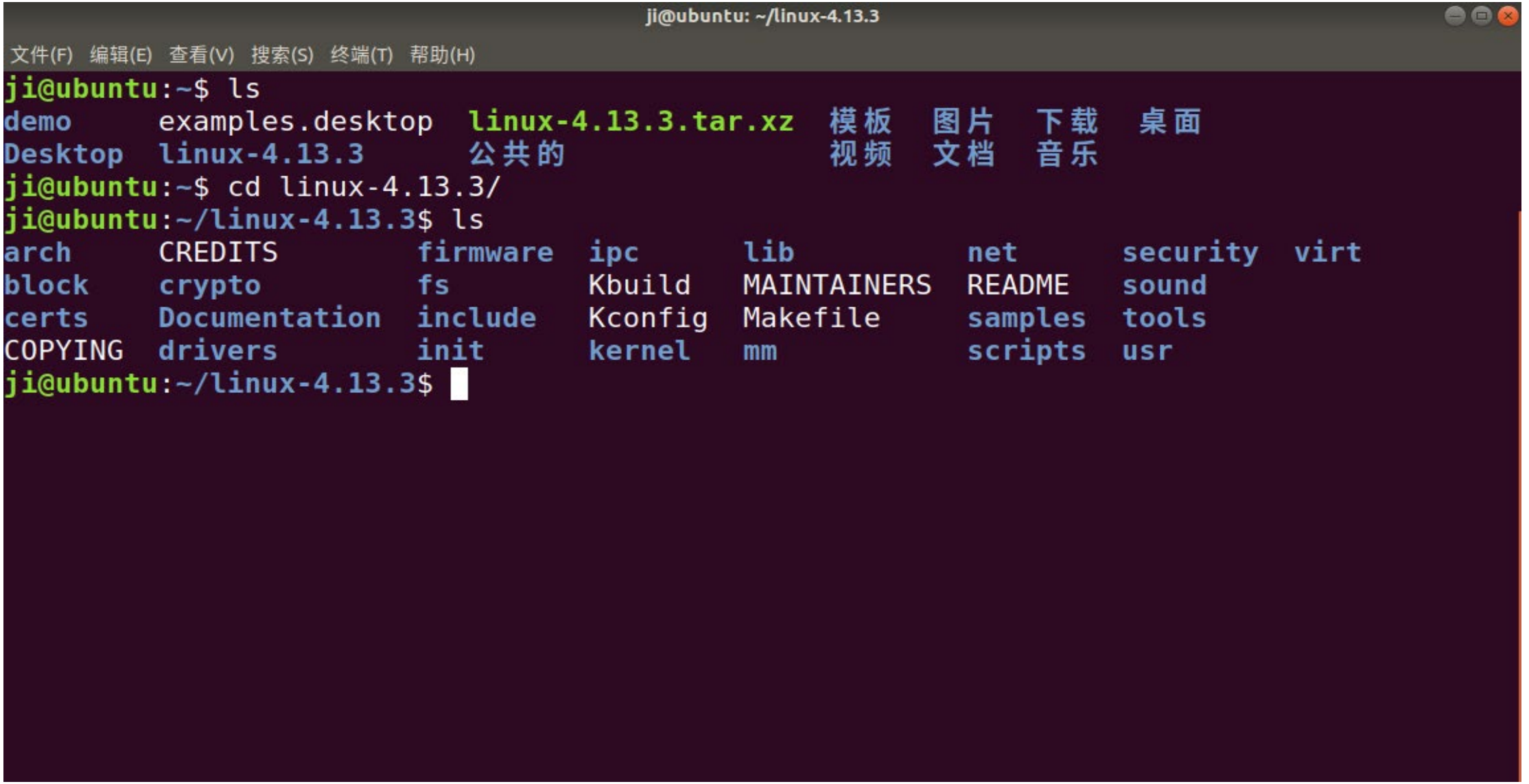
- Operating System Interface
  - Command Interface （命令接口）
  - Program Interface （程序接口，system call）

- User interface，命令接口- Almost all operating systems have a user interface (UI)
  - Command-Line Interface (CLI)，命令行用户接口，文本界面
  - Graphics User Interface (GUI), 图形用户接口
  - Touch-Screen Interface，触摸屏接口
  - Choice of Interface，语音接口
  - ……

User Operating System Interface - CLI

- CLI 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
  - Linux、UNIX：bash、sh、tcsh

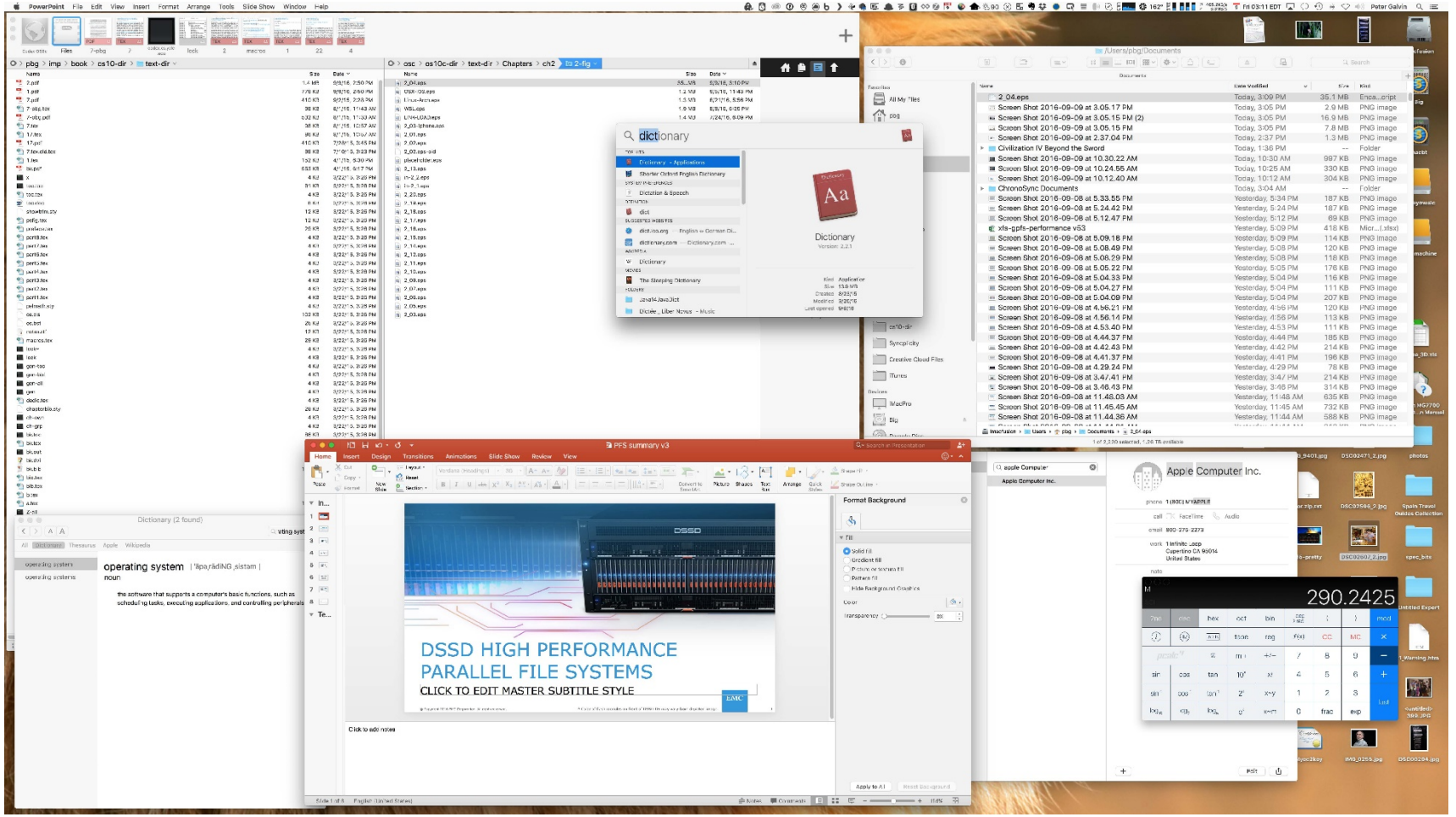
Bourne Shell Command Interpreter



User Operating System Interface - GUI

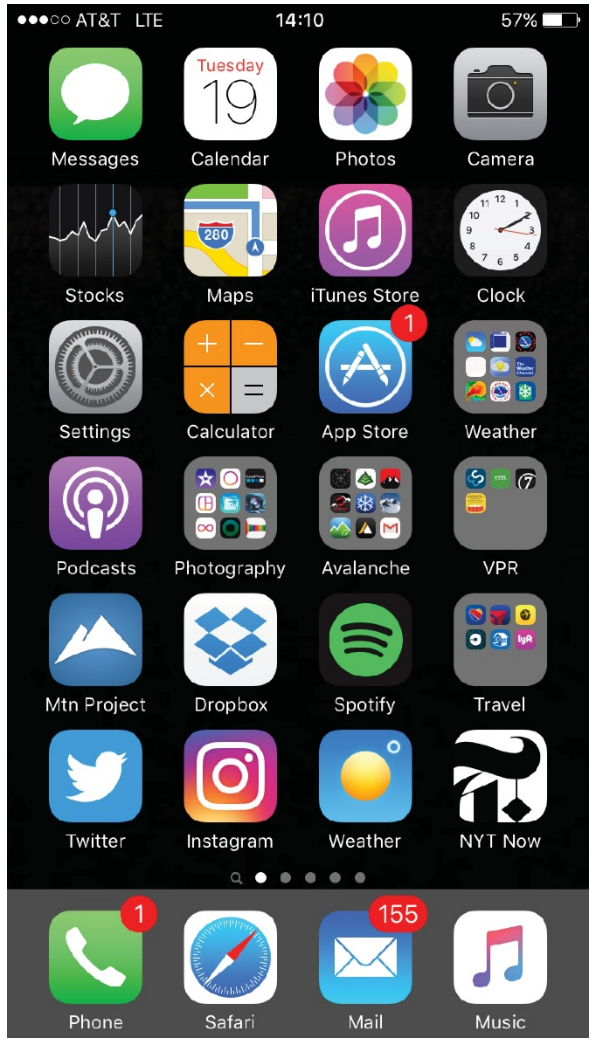
- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)
  - Linux

The Mac OS X GUI

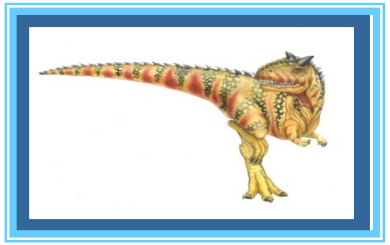


Touchscreen Interfaces

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



2.3 System Calls



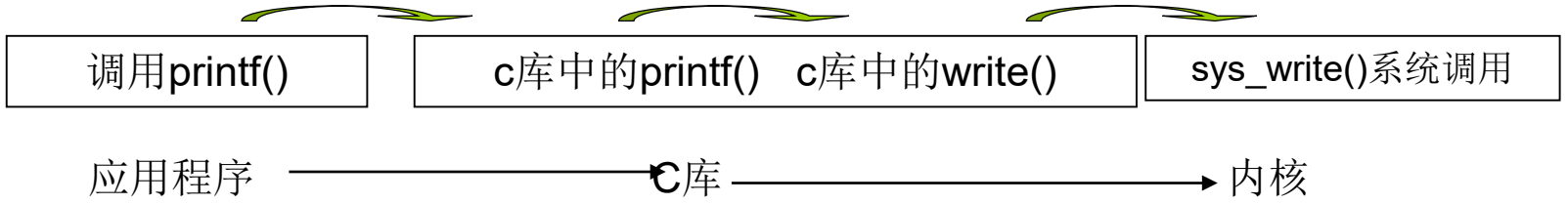
System Calls（系统调用、程序接口）

- System Calls ： Programming interface to the services provided by the OS
- System calls are the programming interface between processes and the OS kernel.
- 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?

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

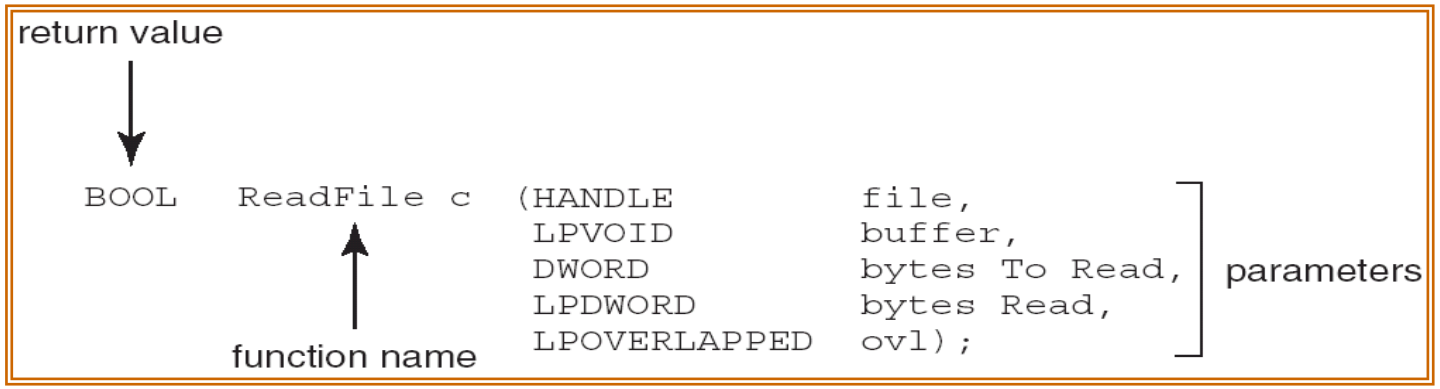
系统调用、API和C库

- 应用编程接口(API)其实是一组函数定义，这些函数说明了如何获得一个给定的服务；而系统调用是通过软中断向内核发出一个明确的请求，每个系统调用对应一个封装例程（wrapper routine，唯一目的就是发布系统调用）。一些API应用了封装例程。
  - API还包含各种编程接口，如：C库函数、OpenGL编程接口等
- 系统调用的实现是在内核完成的，而用户态的函数是在函数库中实现的



Example of Standard API

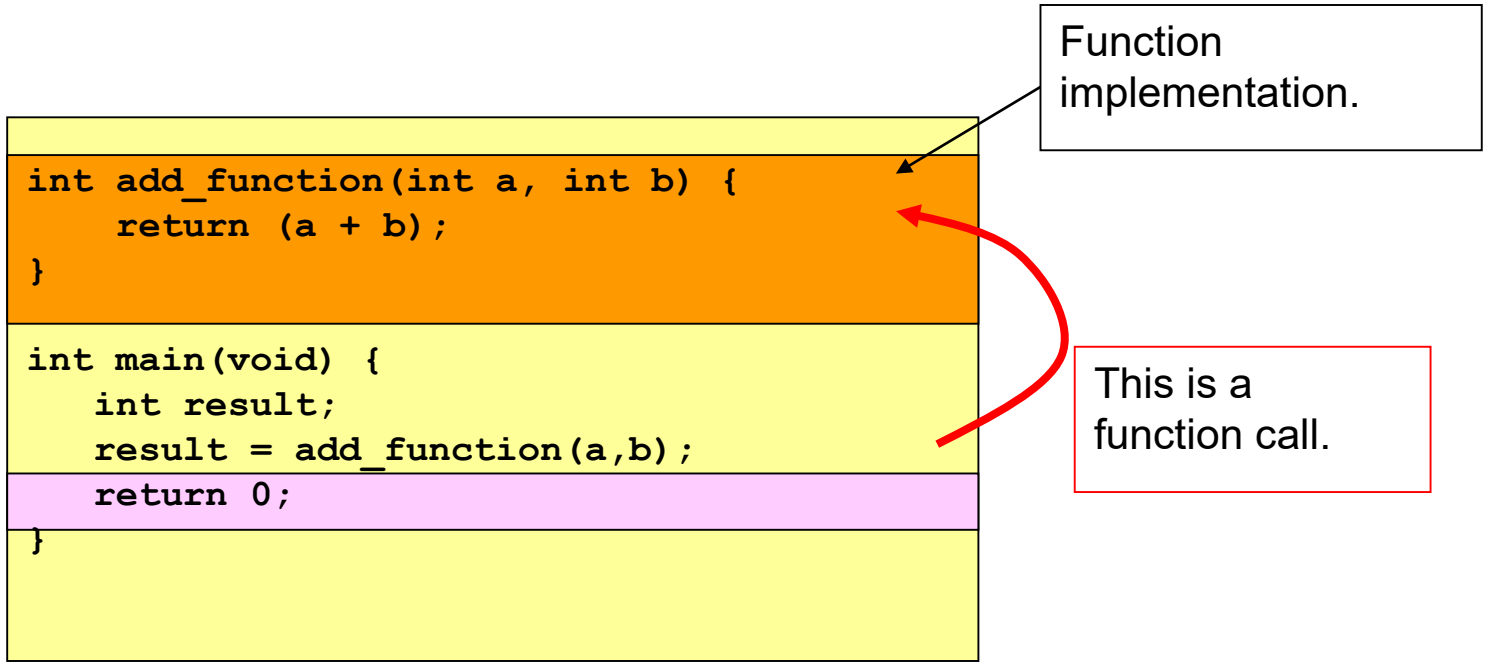
- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

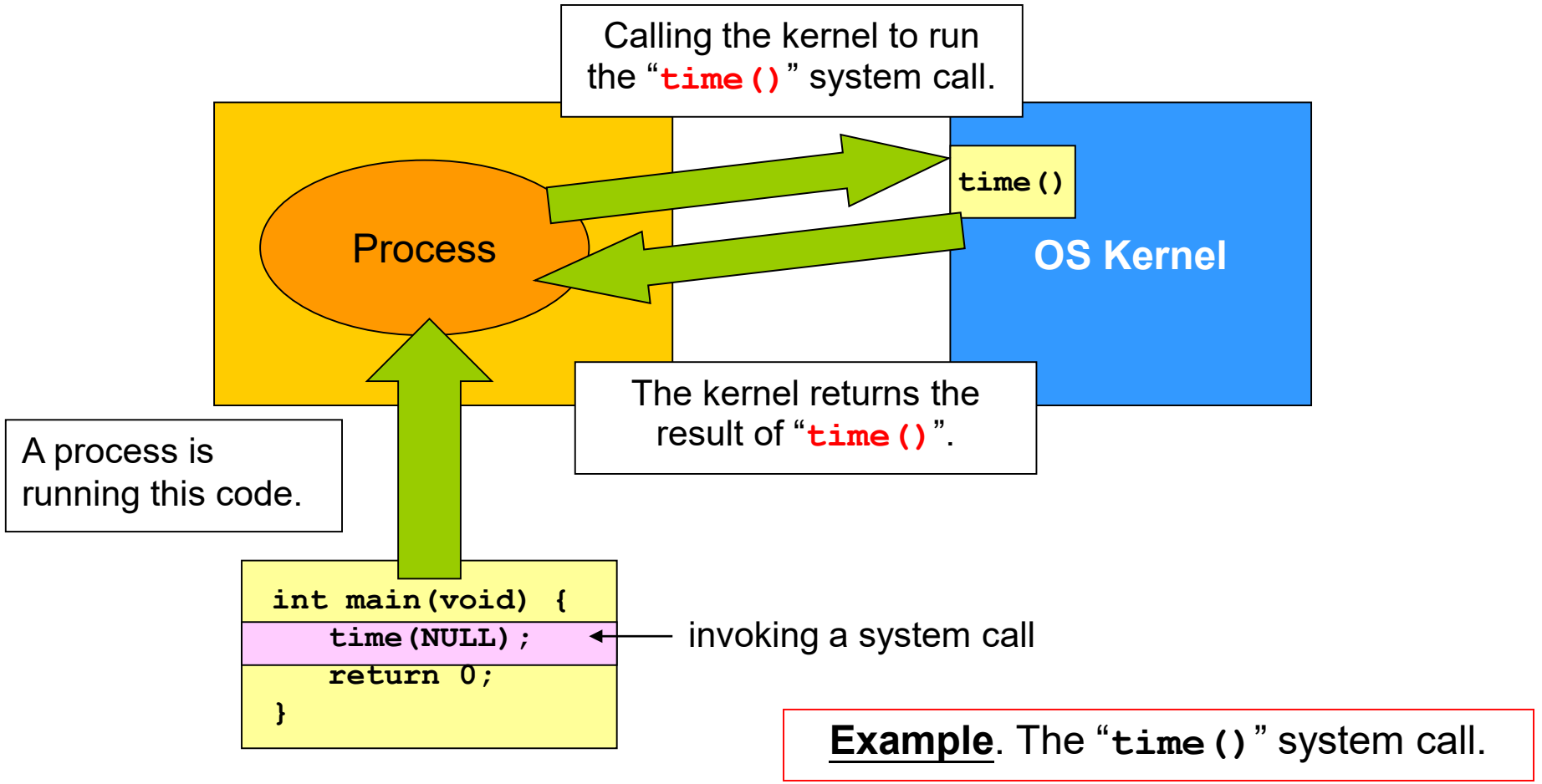
System Calls

- What is a system call?
  - Informally, a system call is similar to a function call, but…
  - The function implementation is inside the OS, or we name it the OS kernel.



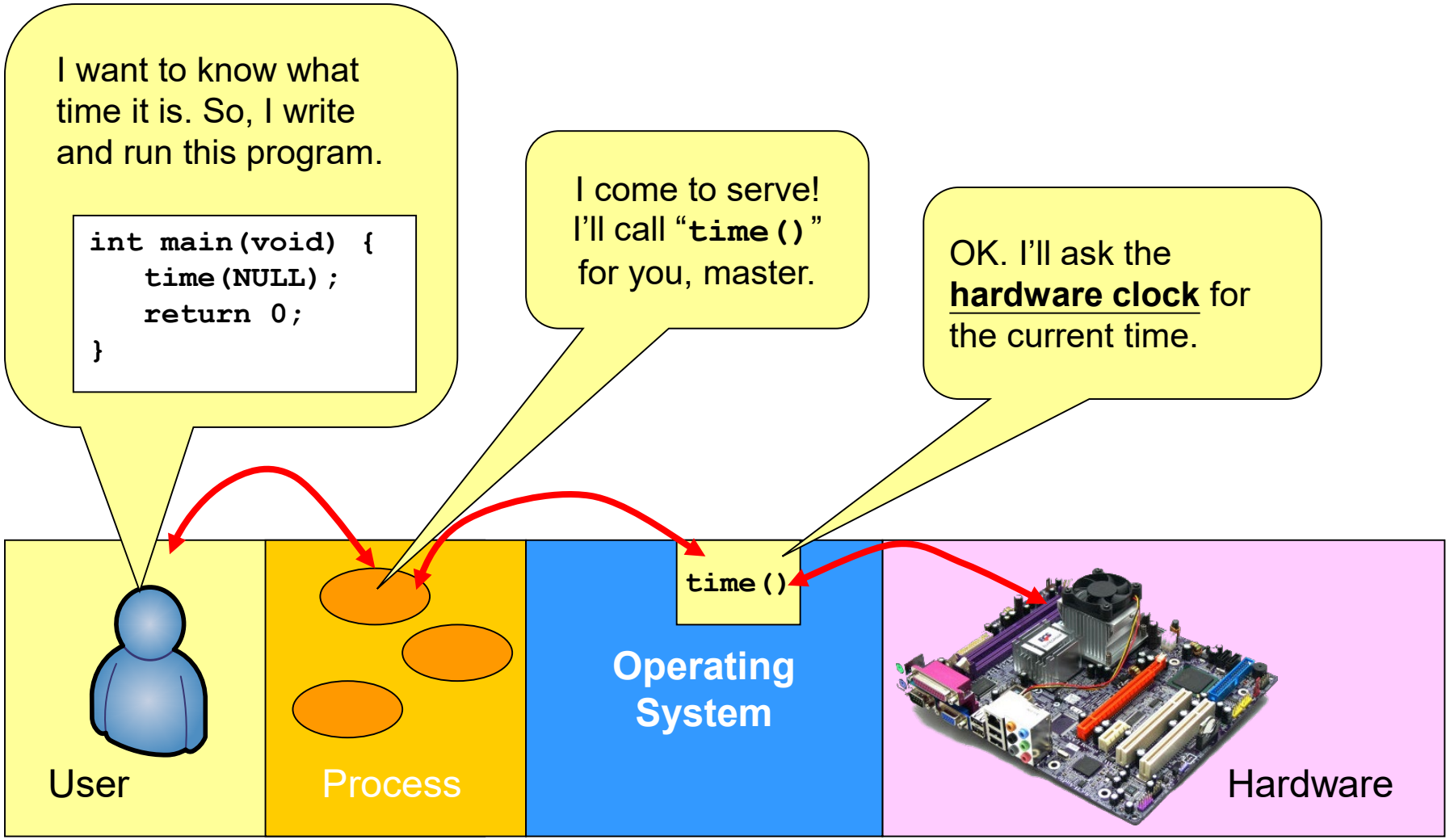
System Calls

- What is a system call?



Example. The "`time()`" system call.

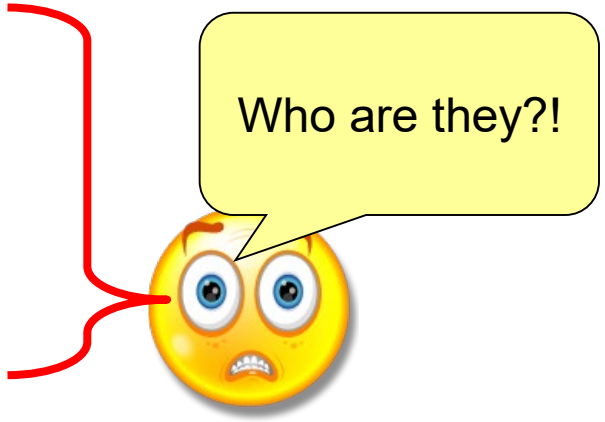
System Calls



System Calls

- When do we know if a “function” is a system call?
  - Read the man page “`syscalls`” under Linux.
- Let’s guess: who are system calls?

Name	System Call?
`printf()`	No
`malloc()`	No
`fopen()`	No
`fclose()`	No



System Call vs Library Function Call

- If they are not system calls, then they are function calls!
- Take `fopen()` as an example.
  - `fopen()` invokes the system call `open()`.
  - So, why people invented `fopen()`?
  - Because `open()` is too primitive and is not programmer-friendly!

The following two calls have the same effect, but…

Function call	`fopen("hello.txt", "w");`
System call	`open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);`

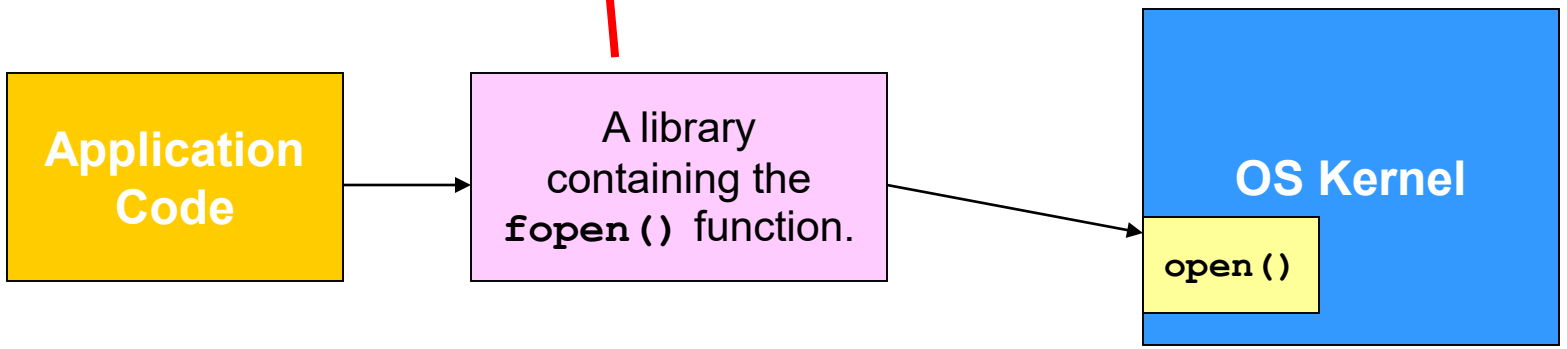
As a programmer, which one do you prefer?

System Call vs Library Function Call

- Those functions are usually packed inside an object called the library file.

```
$ ls /lib/libc*
/lib/libc-2.9.so  ......
$ _
```

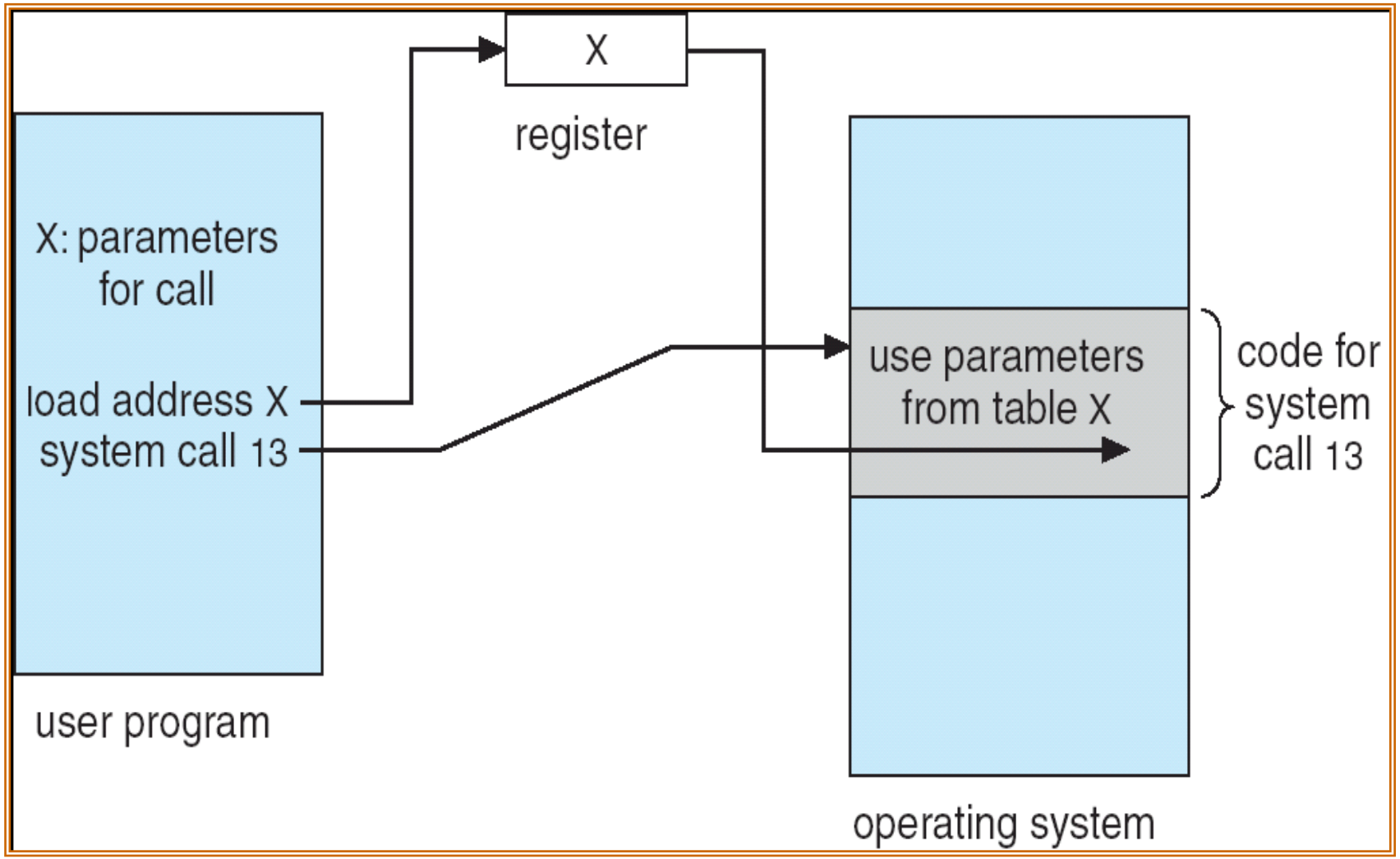
The C standard library file.



System Call Parameter Passing

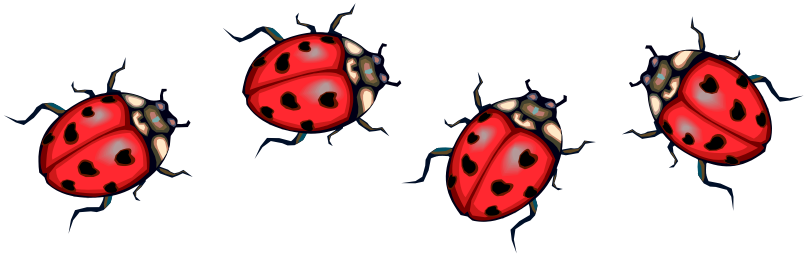
- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - 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



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection



✓ UNIX环境高级编程（第2版），W.Richard Stevens，尤晋元译，人民邮电出版社
✓ Linux/UNIX系统编程手册（上、下册），Michael Kerrisk著，孙剑等译，人民邮电出版社，

Types of System Calls

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

Types of System Calls (cont.)

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

Types of System Calls (Cont.)

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

Types of System Calls (Cont.)

- 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()`

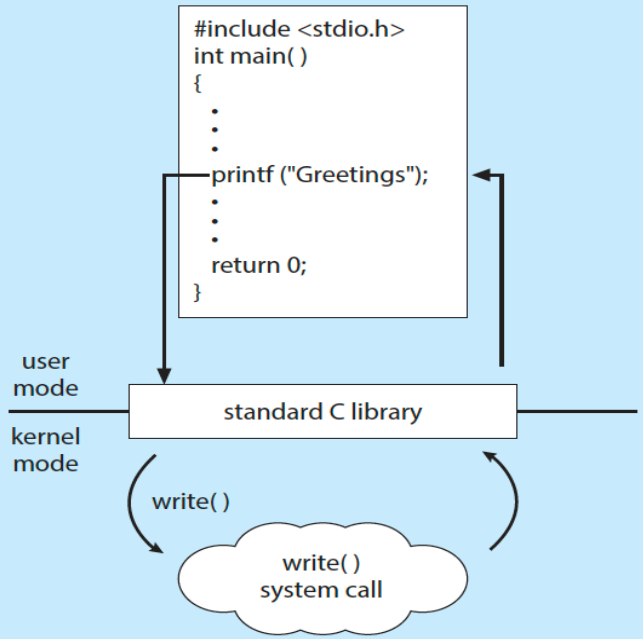
Win32 CreateProcess :example\newproc-win32.c

Standard C Library Example

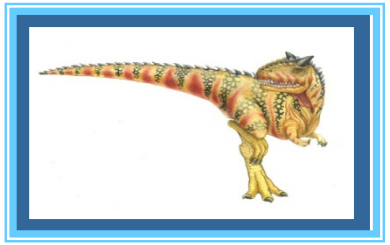
- C program invoking printf() library call, which calls write() system call

THE STANDARD C LIBRARY

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



2.4 System Services



System Services

- System programs provide a convenient environment for program development and execution. The can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls

如：Linux/Unix各种命令（程序）

System Services (cont.)

- 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

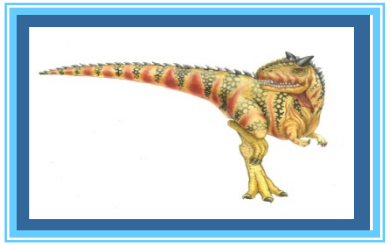
System Services (cont.)

- 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

2.5 Linkers and Loaders



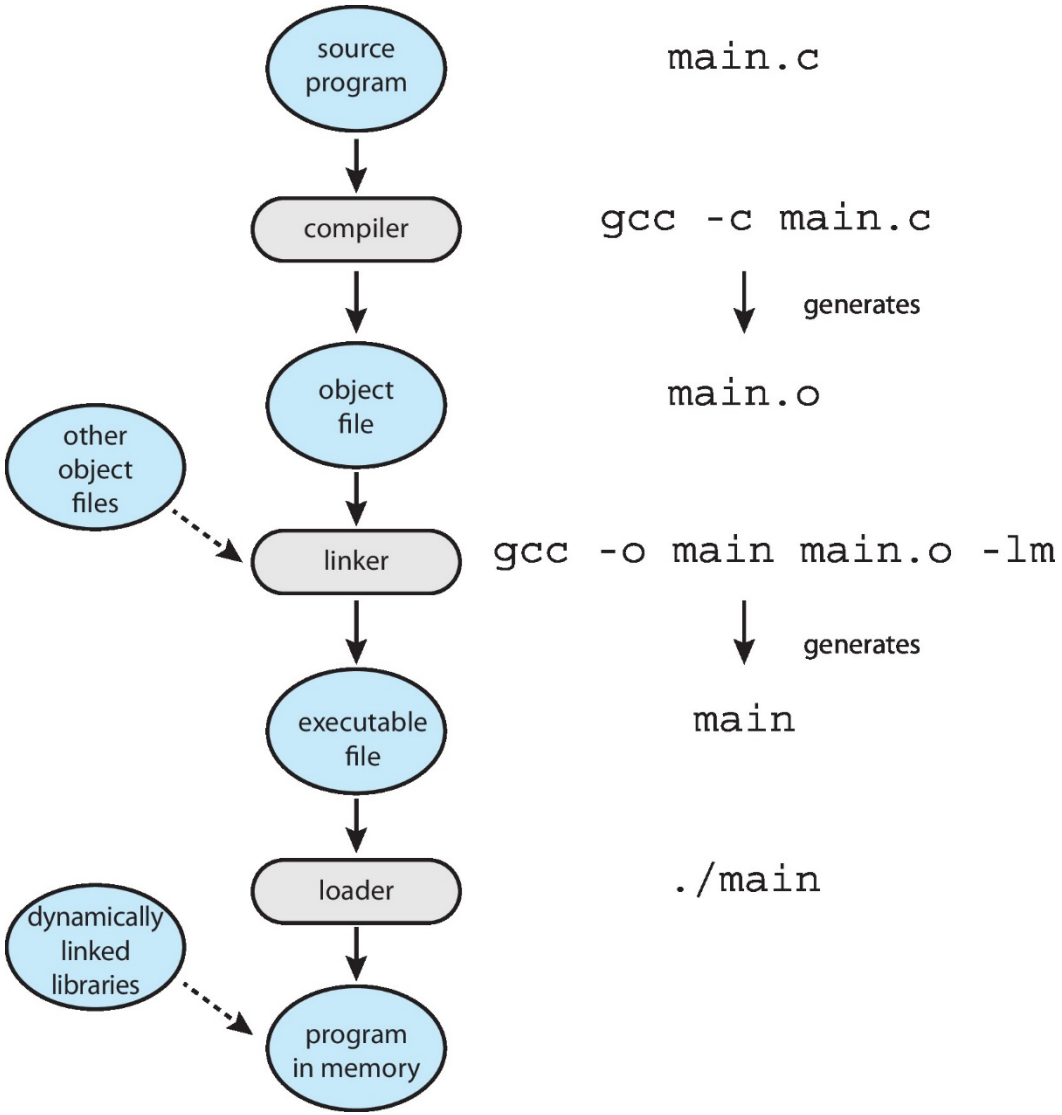
Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- 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

Linkers and Loaders

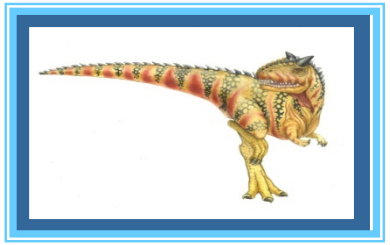
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them

The Role of the Linker and Loader



2.6 Why Applications Are Operating-System Specific

(为什么应用程序只运行在特定的操作系统上)



Why Applications are Operating System Specific

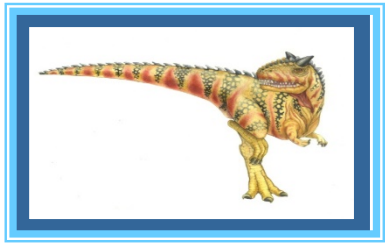
- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc
- **Apps can be multi-operating system**
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple 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

同一个应用程序如何在不同的操作系统上运行？

Why Applications are Operating System Specific

- **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

2.7 Operating-System Design and Implementation



2.6 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

操作系统的设计问题

- 操作系统设计有着不同于一般应用系统设计的特征：
  - 复杂程度高
  - 研制周期长
  - 正确性难以保证

  Windows 2000开发的艰辛与规模
    - 最早Uinx是1400行代码；
    - Windows xp有4000万行代码；
    - fedroa core有2亿多行代码，Linux kernel 4.10有2千多万行代码。
- 解决途径：
  - 良好的操作系统结构
  - 先进的开发方法和工程化的管理方法（软件工程）
  - 高效的开发工具

Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy策略: What will be done?
  Mechanism机制: 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

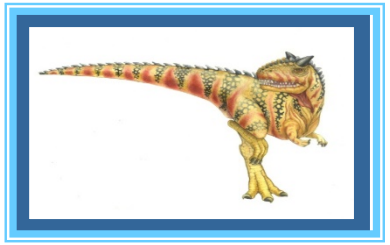
操作系统的设计考虑

- 功能设计：操作系统应具备哪些功能
- 算法设计：选择和设计满足系统功能的算法和策略，并分析和估算其效能
- 结构设计：选择合适的操作系统结构
- 按照系统的功能和特性要求，选择合适的结构，使用相应的结构设计方法将系统逐步地分解、抽象和综合，使操作系统结构清晰、简单、可靠、易读、易修改，而且使用方便，适应性强

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

2.8 Operating-System Structure



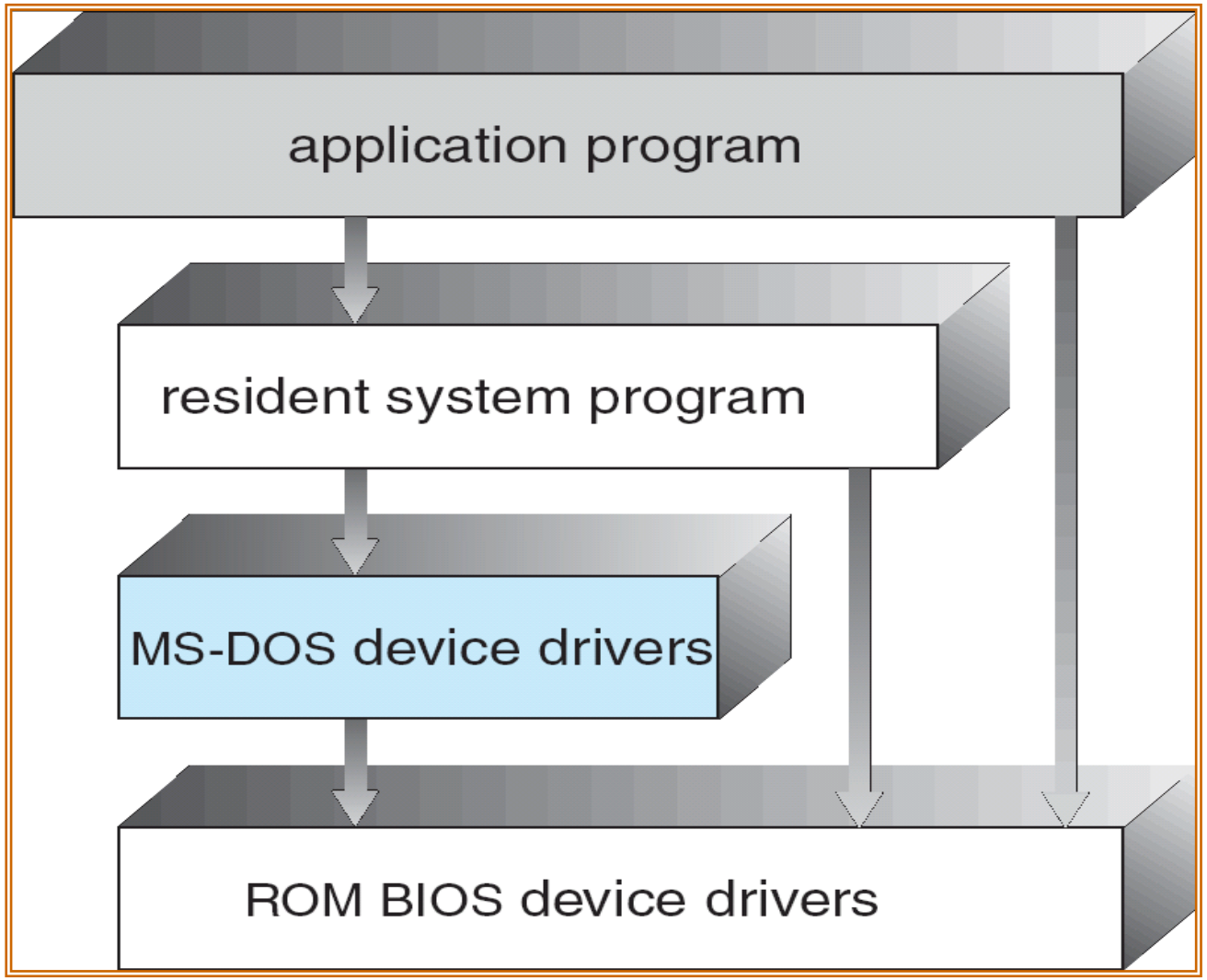
2.7 Operating-System Structure

- Simple Structure简单结构
- Layered Approach层次化结构
- Monolithic Kernels Structure单/宏内核结构
- Microkernel 微内核
- Modules 模块
- Hybrid Systems 混合结构

一、Simple Structure（简单结构）

- 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

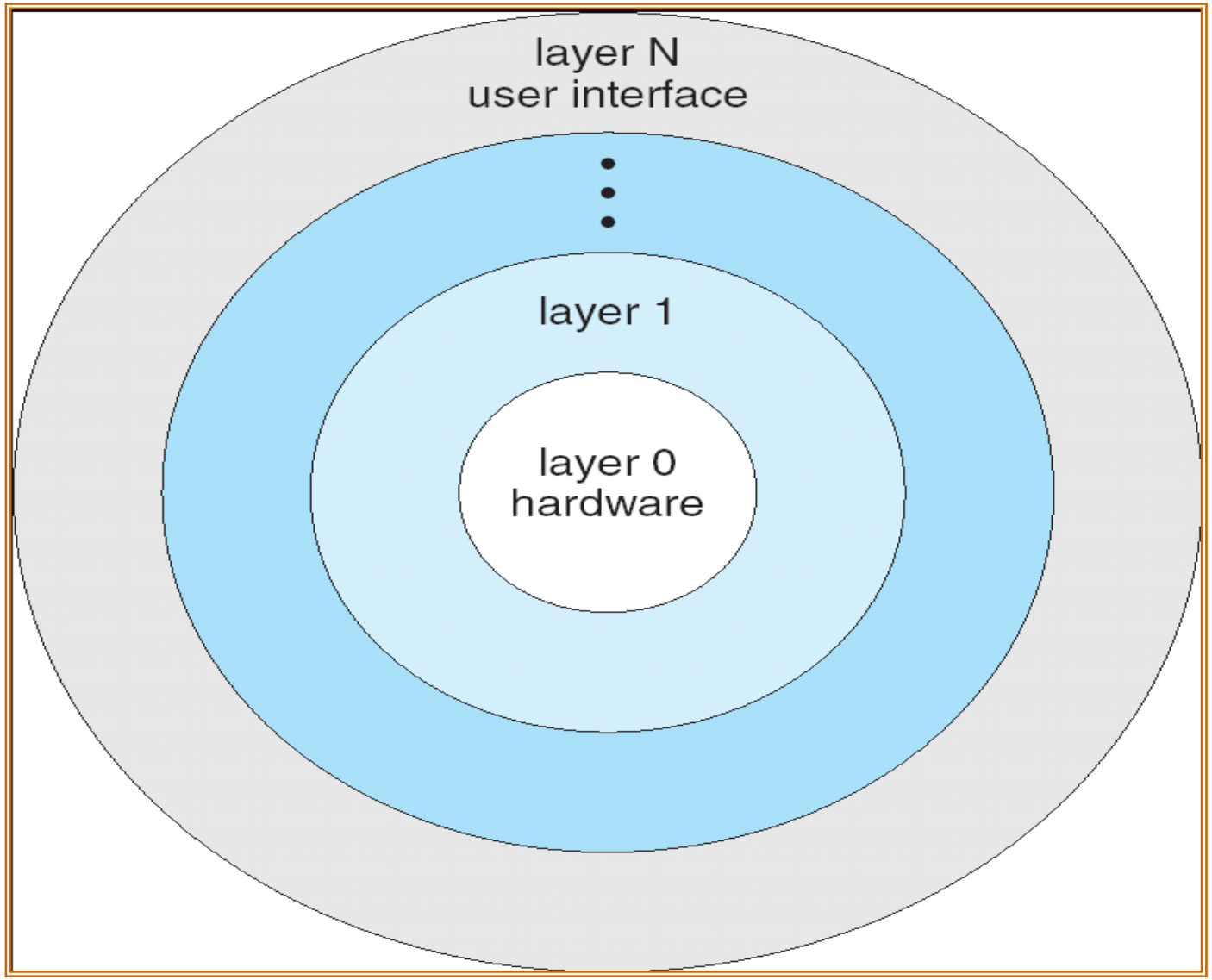
MS-DOS Structure



二、Layered Approach（层次结构）

- 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

Layered Operating System



三、Monolithic Structure（单/宏内核结构）

(2) 宏内核：　　宏内核是构建系统内核的传统方法。在这种方法中，内核的全部代码，包括所有子系统（如内存管理、文件系统、设备驱动程序）都打包到一个文件中。内核中的每个函数都可以访问内核中所有其他部分。如果编程时不小心，很可能会导致源代码中出现复杂的嵌套。

- Monolithic Kernels： the entire code of the kernel — including all its subsystems such as memory management, filesystems, or device drivers — is packed into a single file. Each function has access to all other parts of the kernel;　　continuous address
- Earliest and most common OS architecture (UNIX)
- Every component of the OS is contained in the Kernel
- Examples: OS/360, VMS and Linux

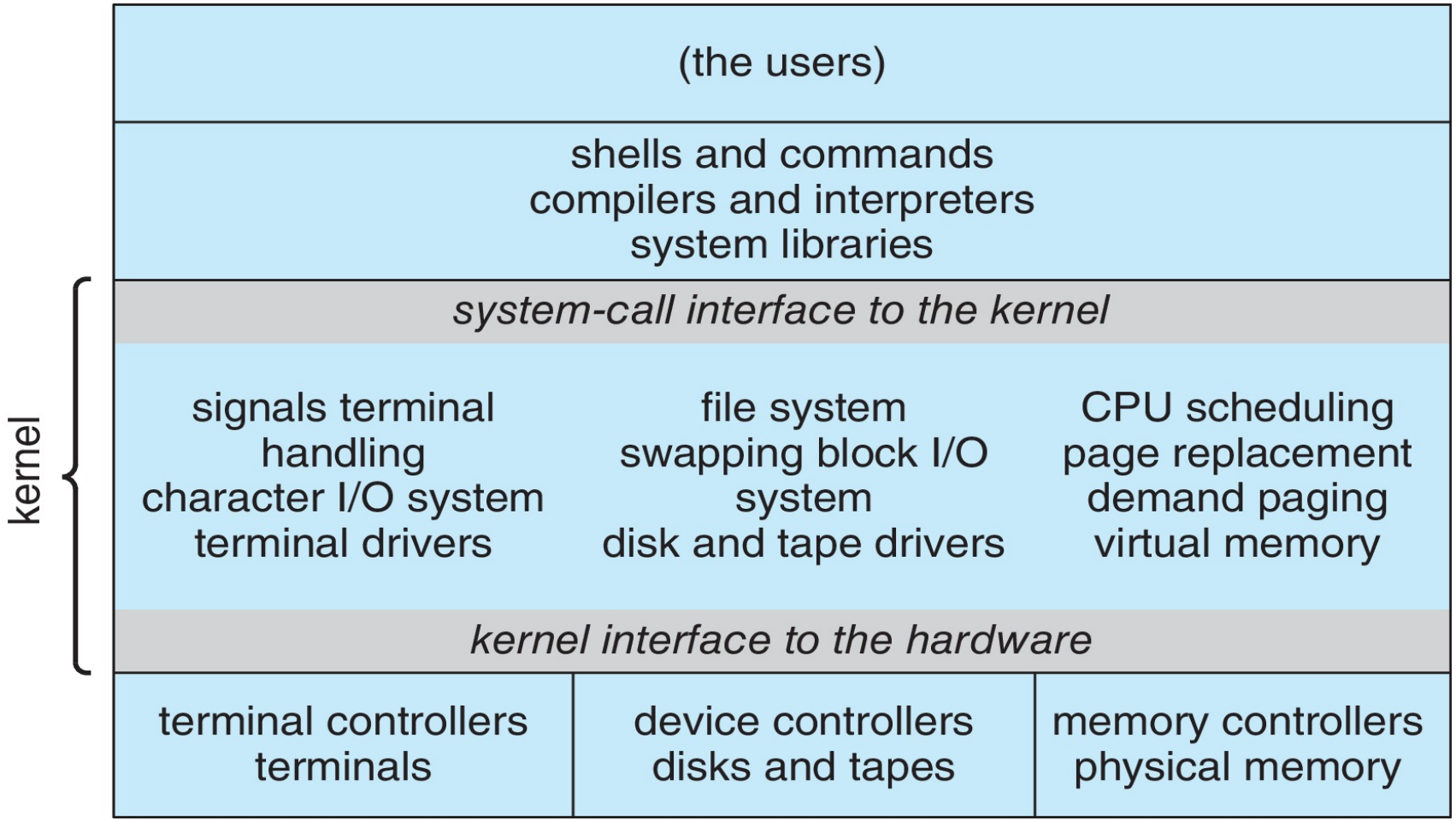
Monolithic Structure

- Advantages:
  - highly efficient because of direct communication between components
- Disadvantages:
  - Difficult to isolate source of bugs and other errors
  - Hard to modify and maintain
  - Kernel gets bigger as the OS develops.

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
  - 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

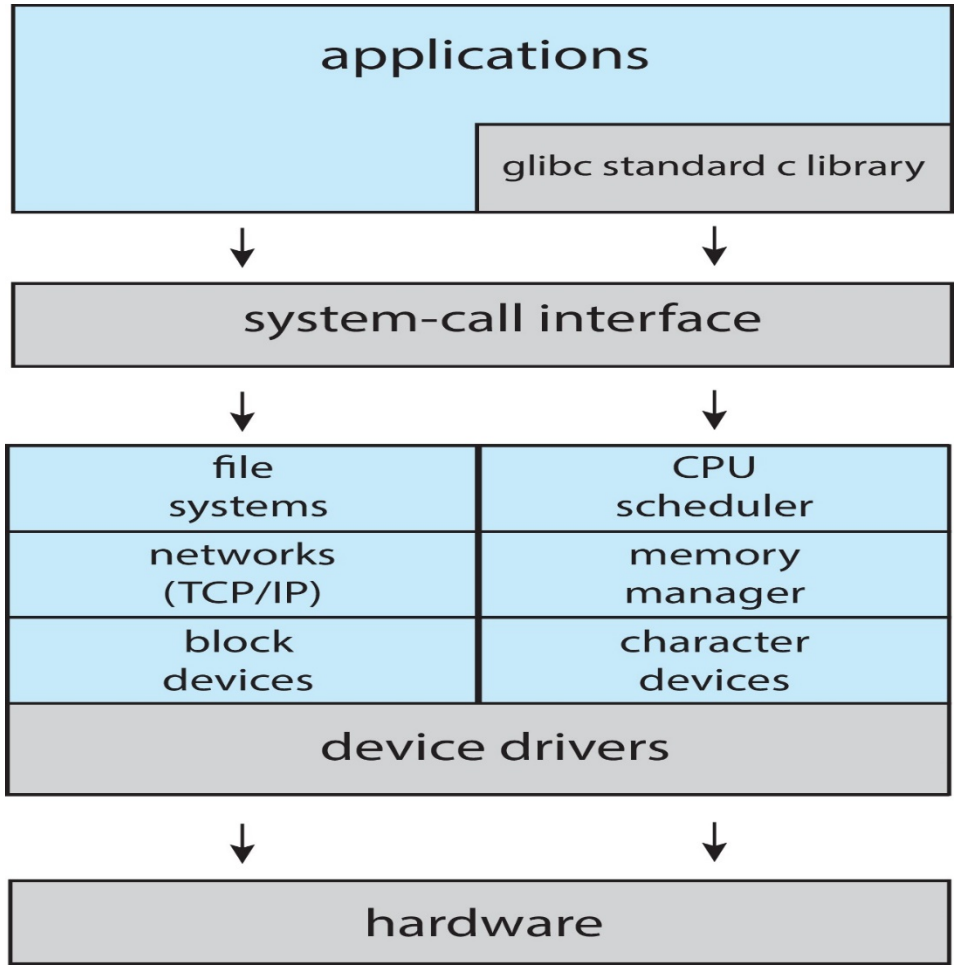
Traditional UNIX System Structure



Beyond simple but not fully layered

Linux System Structure

Monolithic plus modular design



四、Microkernel System Structure （微内核结构）

- Microkernels — only the most elementary functions are implemented directly in a central kernel— the microkernel. All other functions are delegated to autonomous processes that communicate with the central kernel via clearly defined communication interfaces
- 由下面两大部分组成 ：
  - “微”内核
  - 若干服务
- Moves as much from the kernel into “*user*” space
- Communication takes place between user modules using message passing

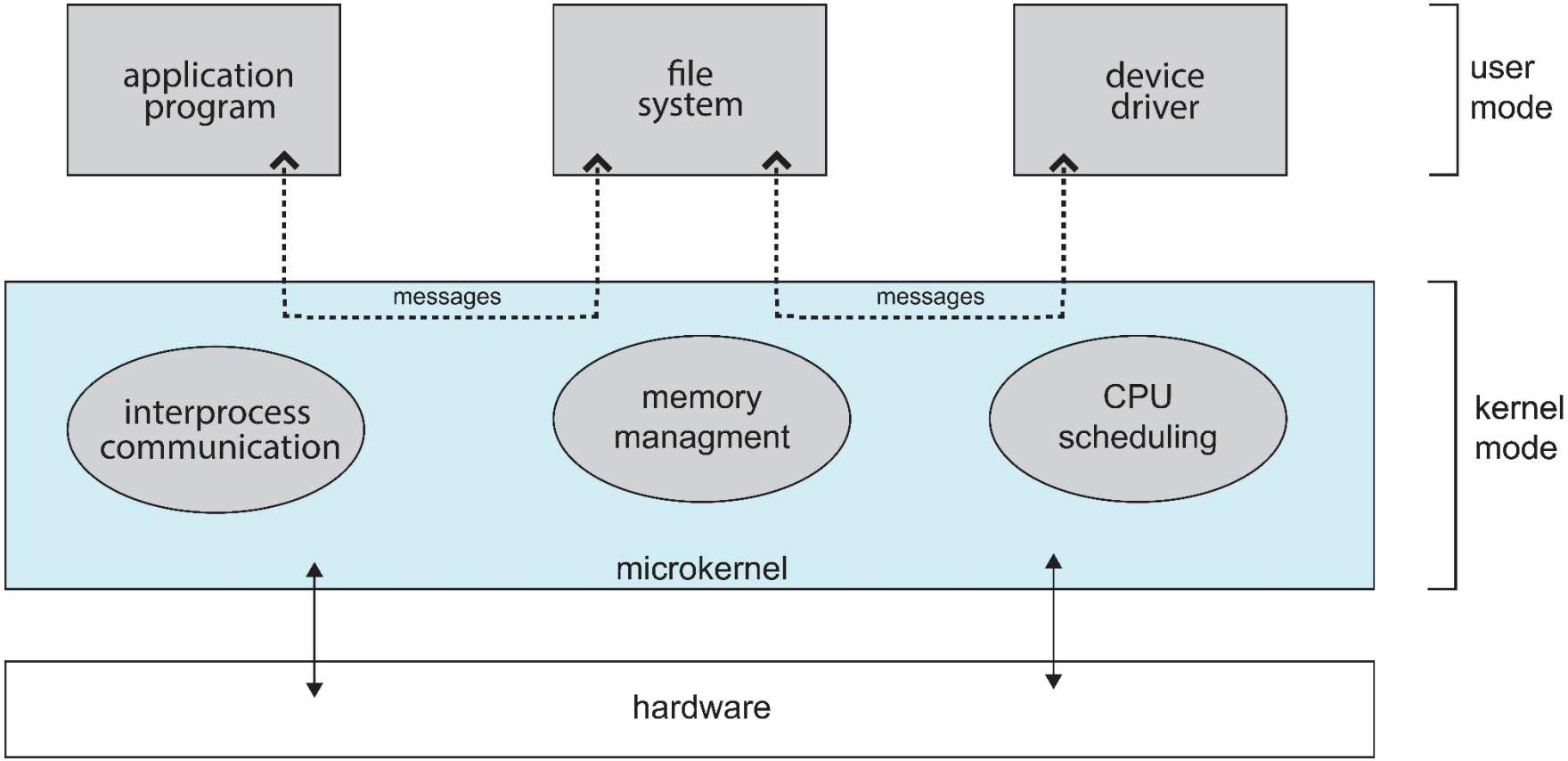
(1) **微内核**：这种范型中，只有最基本的功能直接由中央内核（即微内核）实现。所有其他的功能都委托给一些独立进程，这些进程通过明确定义的通信接口与中心内核通信。例如，独立进程可能负责实现各种文件系统、内存管理等。（当然，与系统本身的通信需要用到最基本的内存管理功能，这是由微内核实现的。但系统调用层次上的处理则由外部的服务器进程实现。）理论上，这是一种很完美的方法，因为系统的各个部分彼此都很清楚地划分开来，同时也迫使程序员使用“清洁的”程序设计技术。这种方法的其他好处包括：动态可扩展性和在运行时切换重要组件。但由于在各个组件之间支持复杂通信需要额外的CPU时间，所以尽管微内核在各种研究领域早已经成为活跃主题，但在实用性方面进展甚微。

Microkernel System Structure （微内核结构）

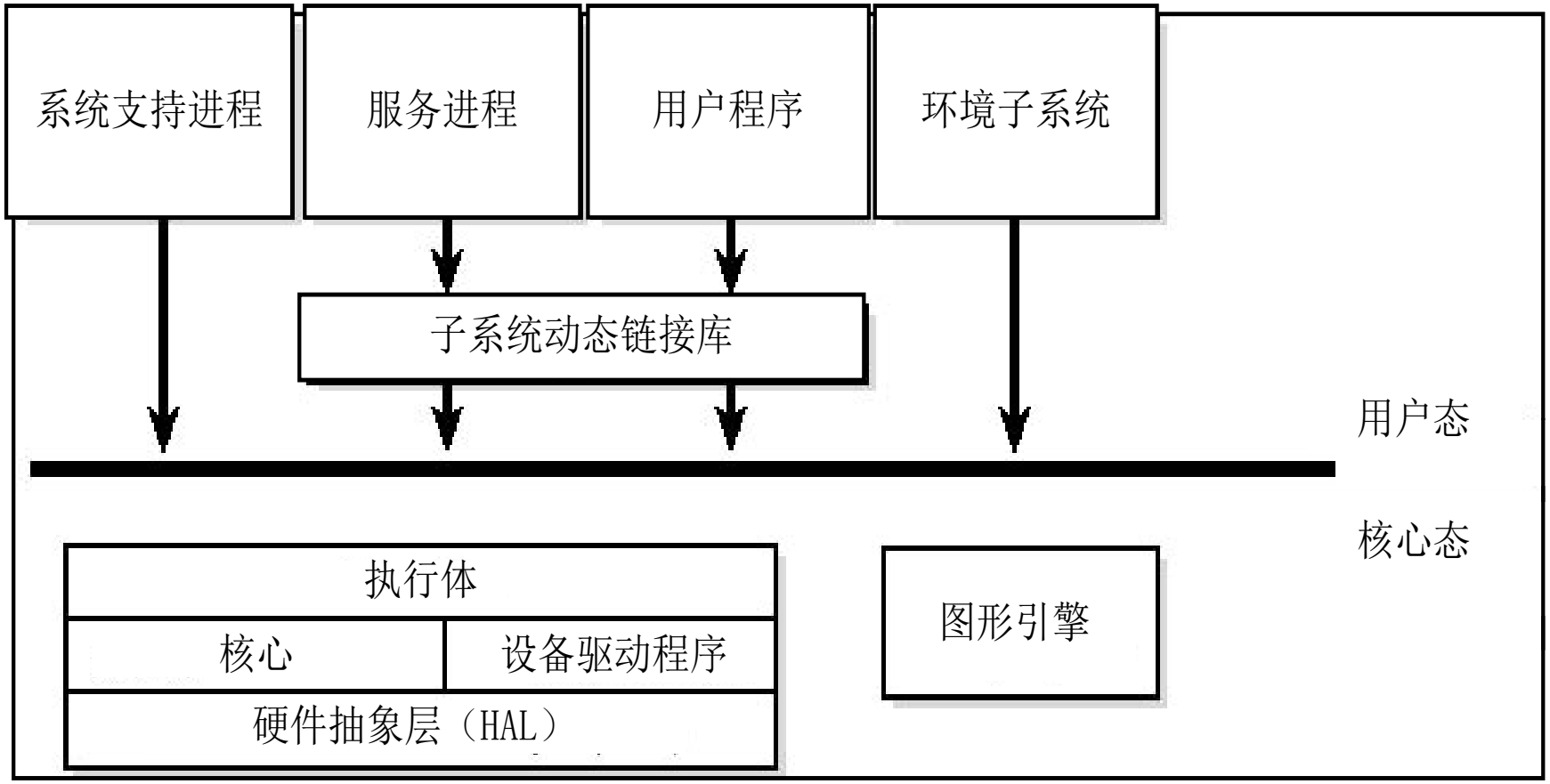
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

- Windows NT … Windows 8、Windows 10、Mac OS
- L4 ： http://www.l4hq.org/
- 华为：鸿蒙

Microkernel System Structure



Windows Kernel



Windows NT 4.0 起，采用 microkernel 的架构

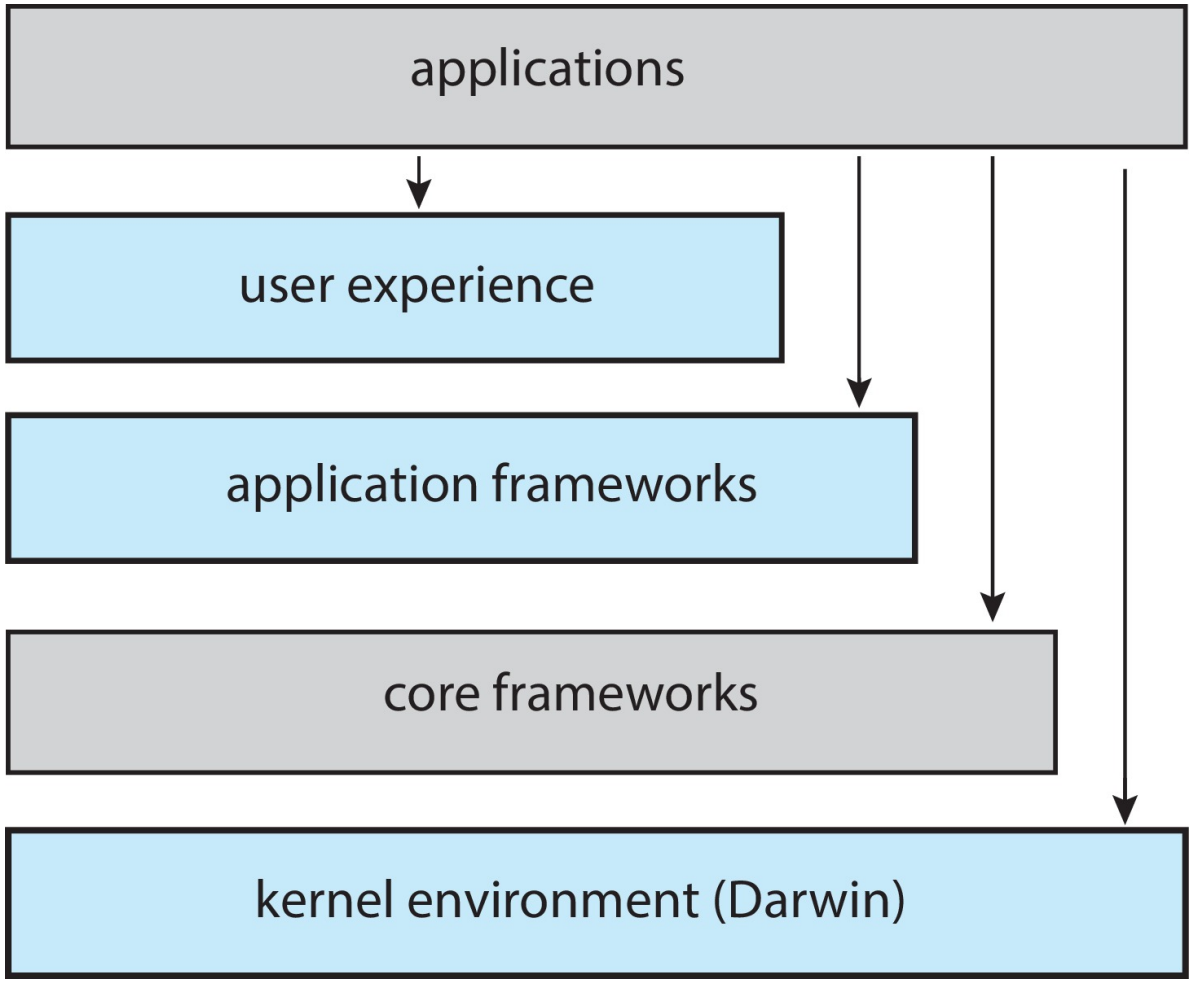
五、Modules（模块）

- Most modern operating systems implement kernel modules （内核模块）
  - 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
- Overall, similar to layers but with more flexible
- Linux , Solaris, etc

六、Hybrid Systems（混合系统）

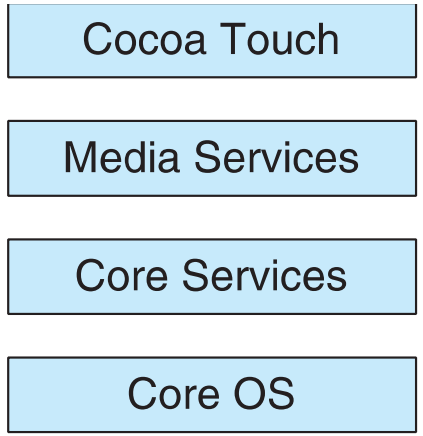
- Most modern operating systems actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic,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

macOS and iOS Structure



iOS

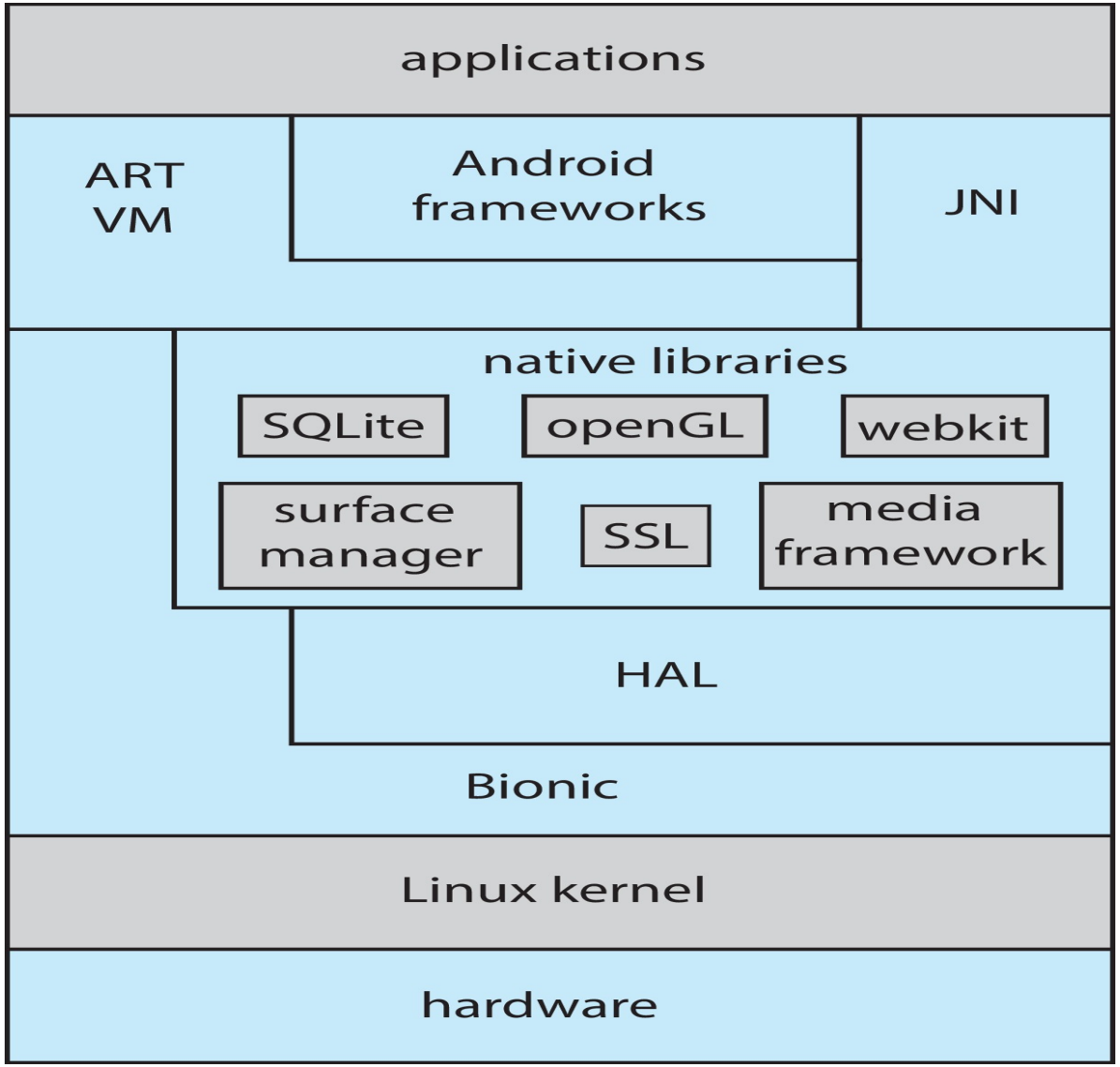
- Apple mobile OS for *iPhone*, *iPad*
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - Cocoa Touch Objective-C API for developing apps
  - Media services layer for graphics, audio, video
  - Core services provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



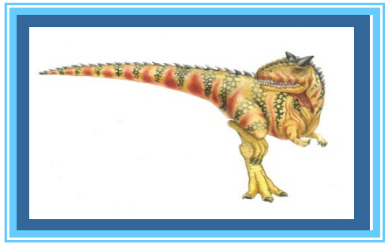
Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



2.9 Building and Booting an Operating System



Building and Booting an Operating System

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

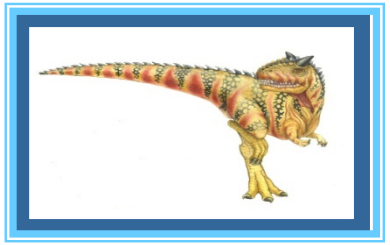
Building and Booting Linux

- 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`"

WINDOWS 启动

- ROM中POST（Power On Self-Test）代码
- BIOS/EFI（Extended Firmware Interfacte）
- MBR(Main Boot Record)
- 引导扇区(Boot sector)
- NTLDR/WinLoad
- NTOSKRNL/HAL/BOOTVID/KDCOM
- SMSS.EXE
- WinLogon.EXE

2.10 Operating-System Debugging



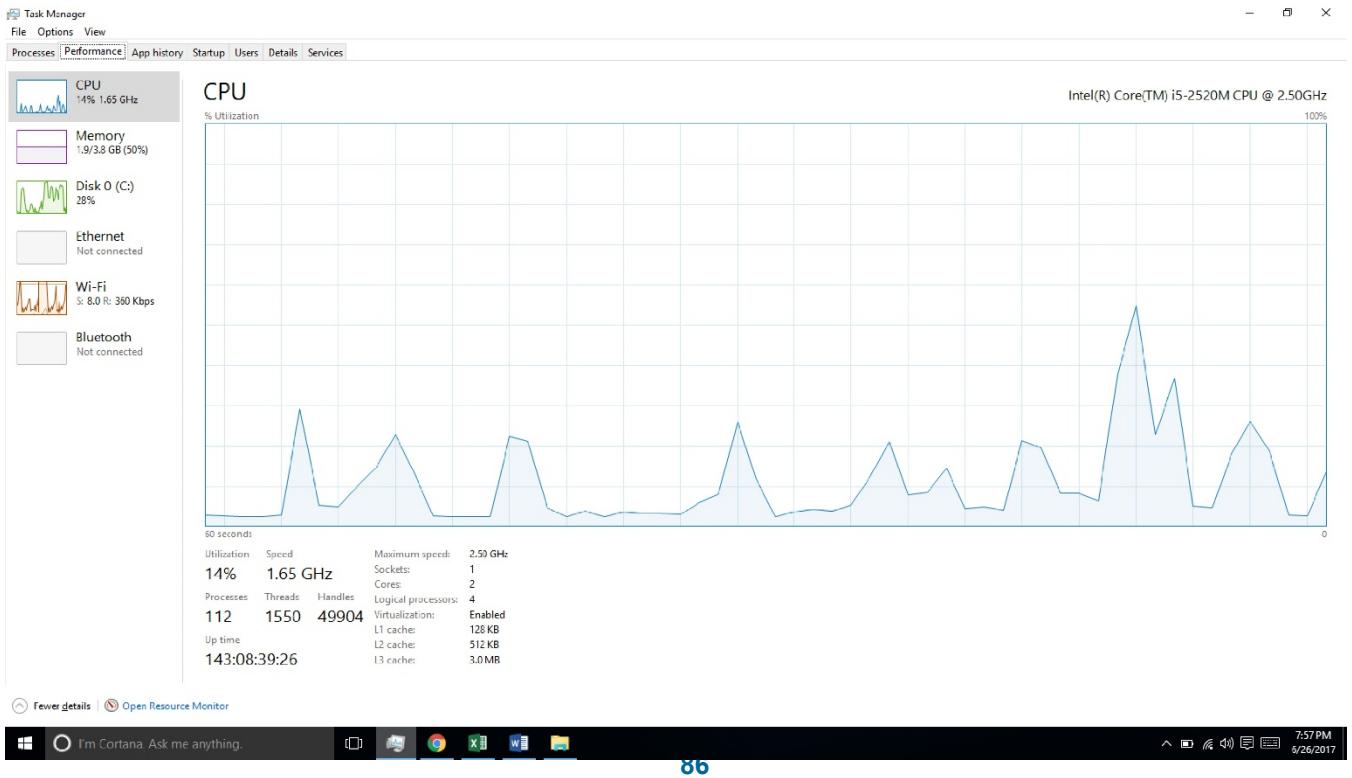
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Also **performance tuning**
- OS 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
- 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.”

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



Tracing

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

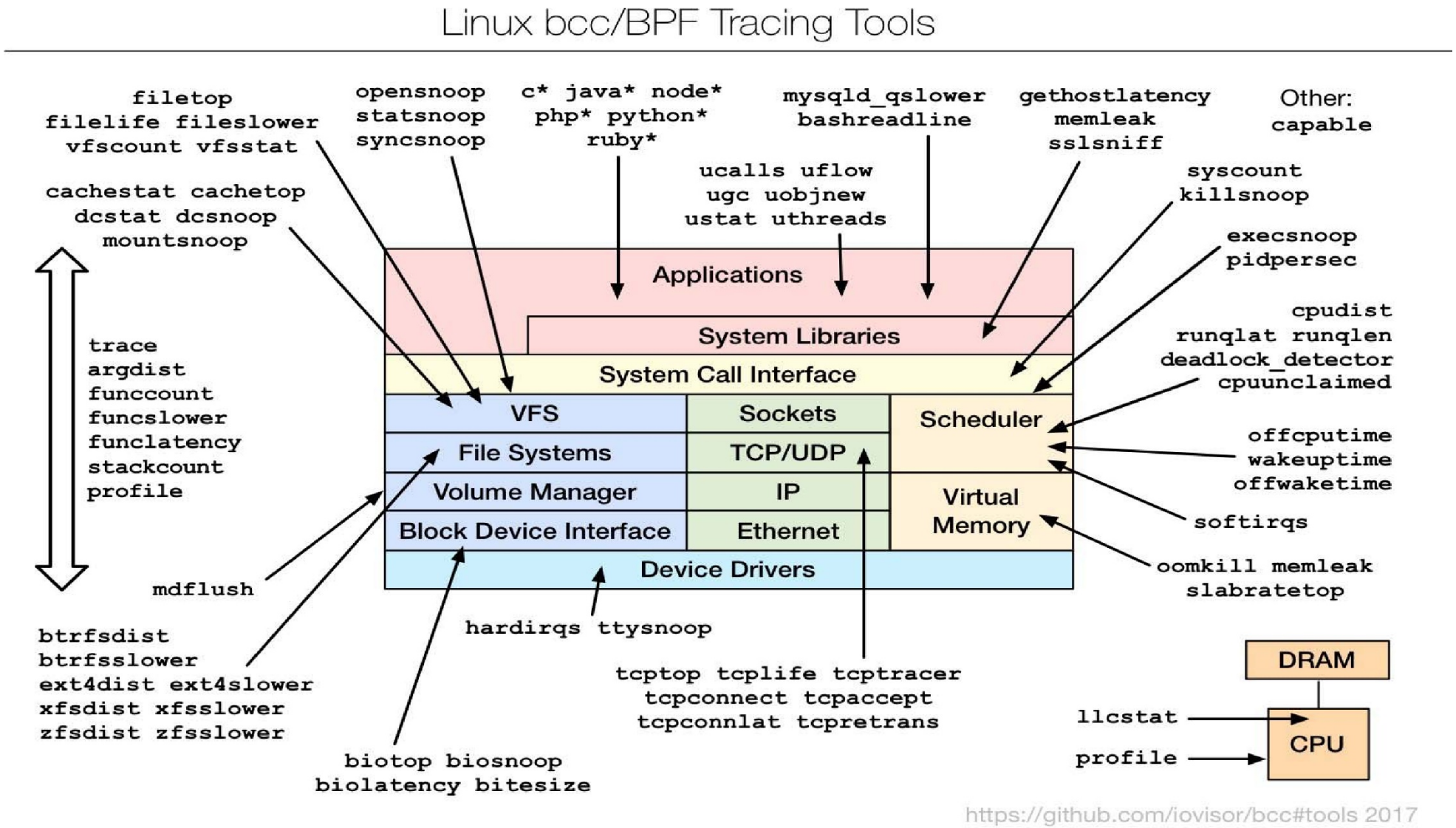
BCC

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

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

- Many other tools (next slide)

Linux bcc/BPF Tracing Tools



习题分析

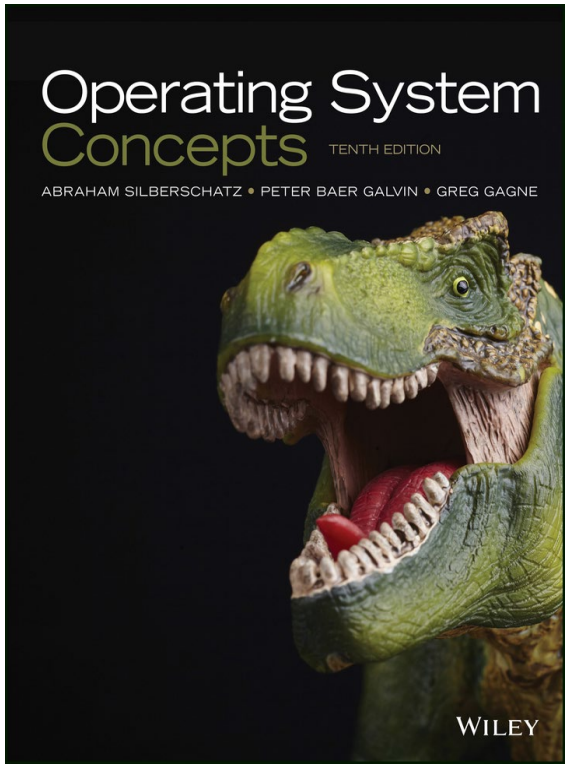
- 习题分析及练习\ex_ch1-2.doc

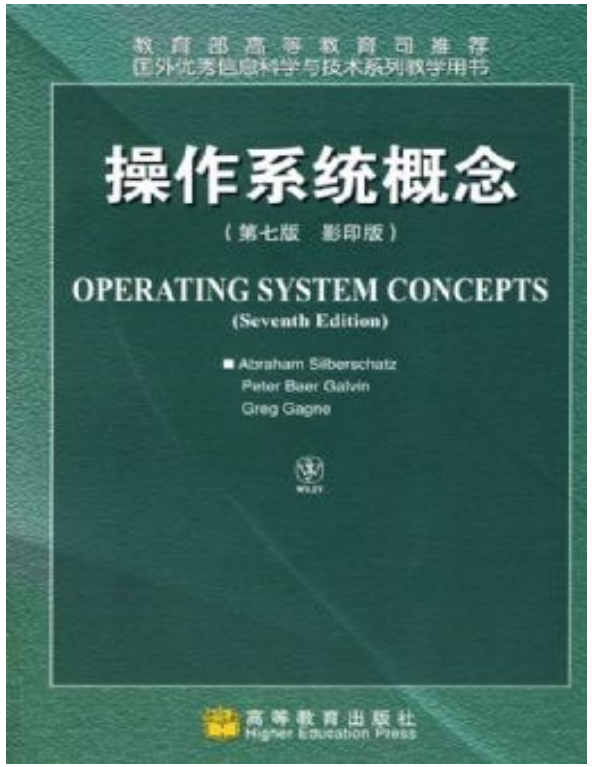
Homework

- 在“学在浙大”中，请按时完成

Reading Assignments

- Read for next week:
  - Chapters 3 of the text book:





End of Chapter 2