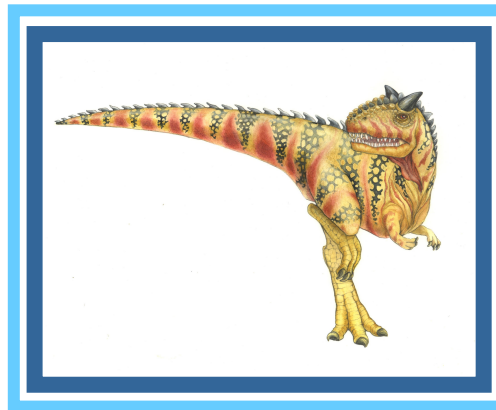


Chapter 2: Operating-System Structures





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
- Operating System Generation
- System Boot





Objectives

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





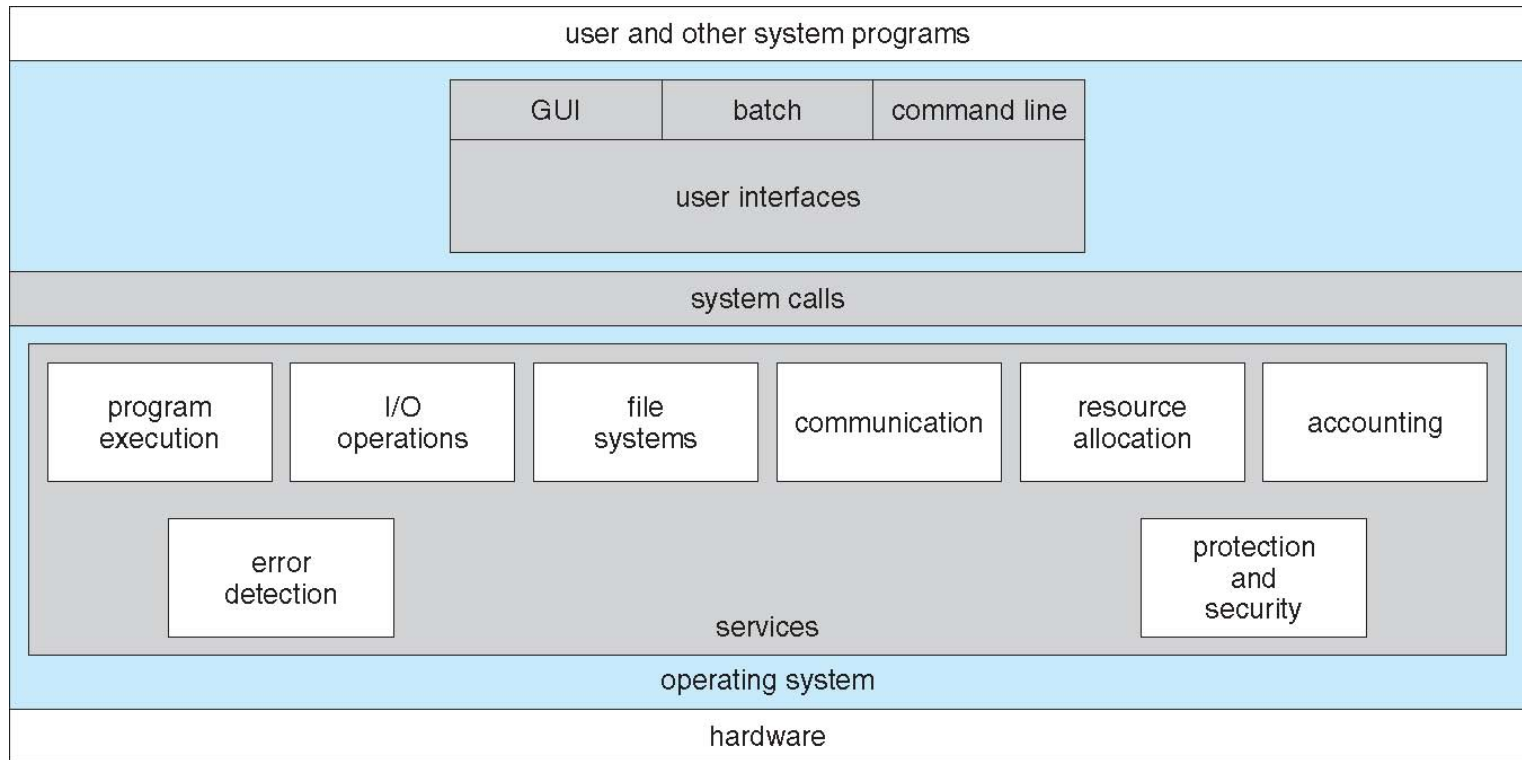
Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users





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 (CLI)**, **Graphical 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





Operating System Services (Cont.)

- 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.)

- 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.
 - **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





User Operating System Interface - CLI

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

- Sometimes implemented in kernel, sometimes as special programs (Windows and Linux)
- Sometimes multiple flavors implemented – **shells**, e.g., C-Shell, Korn Shell, Bourne Shell
- Primarily fetches a command from user and executes it
- Sometimes commands are built-in, sometimes they are just names of programs (used in Unix)
 - ▶ If the latter, adding new features doesn't require shell modification





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 (1973)
- 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)





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. Why?
 - Portability
 - More convenient, as they hide system-call details
- 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)

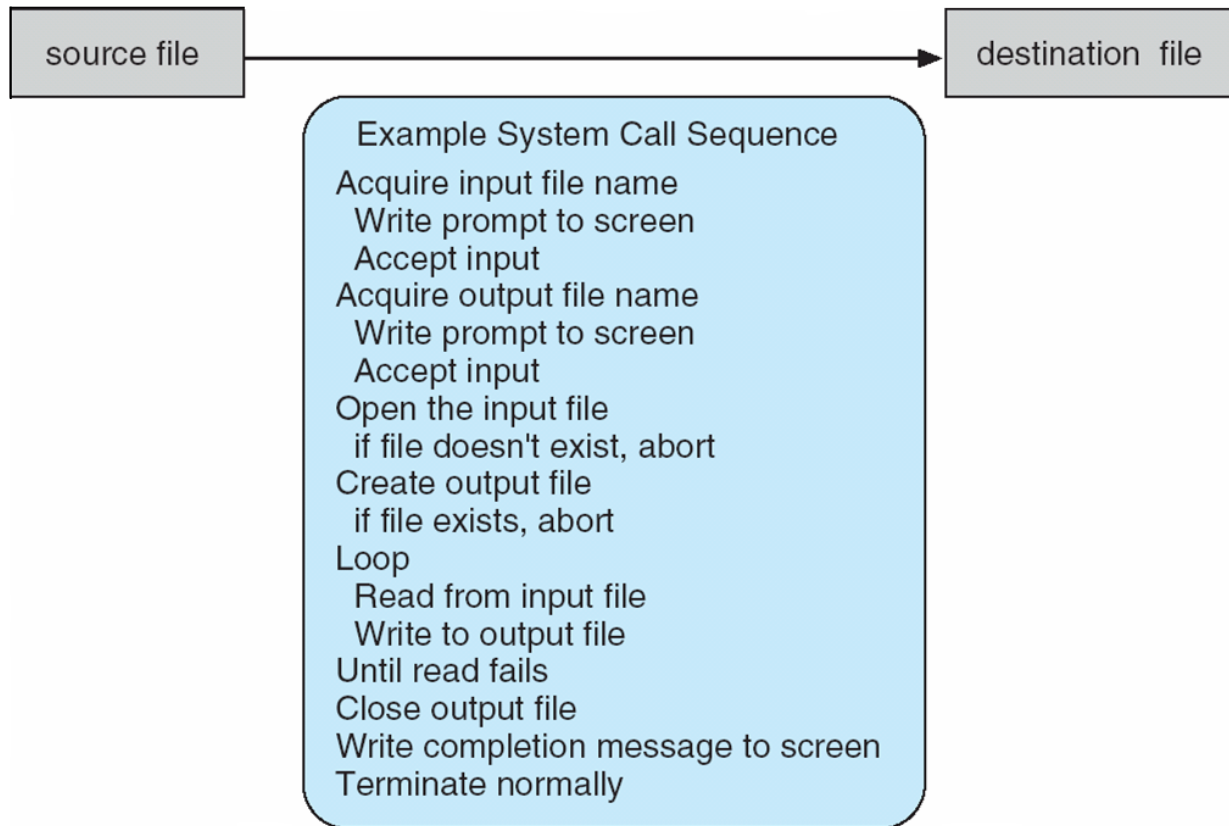
Note that the system-call names used throughout this text are generic





Example of System Calls

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





Example of Standard API

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

ssize_t	read	(int fd, void *buf, size_t count)
return value	function name	parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

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

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





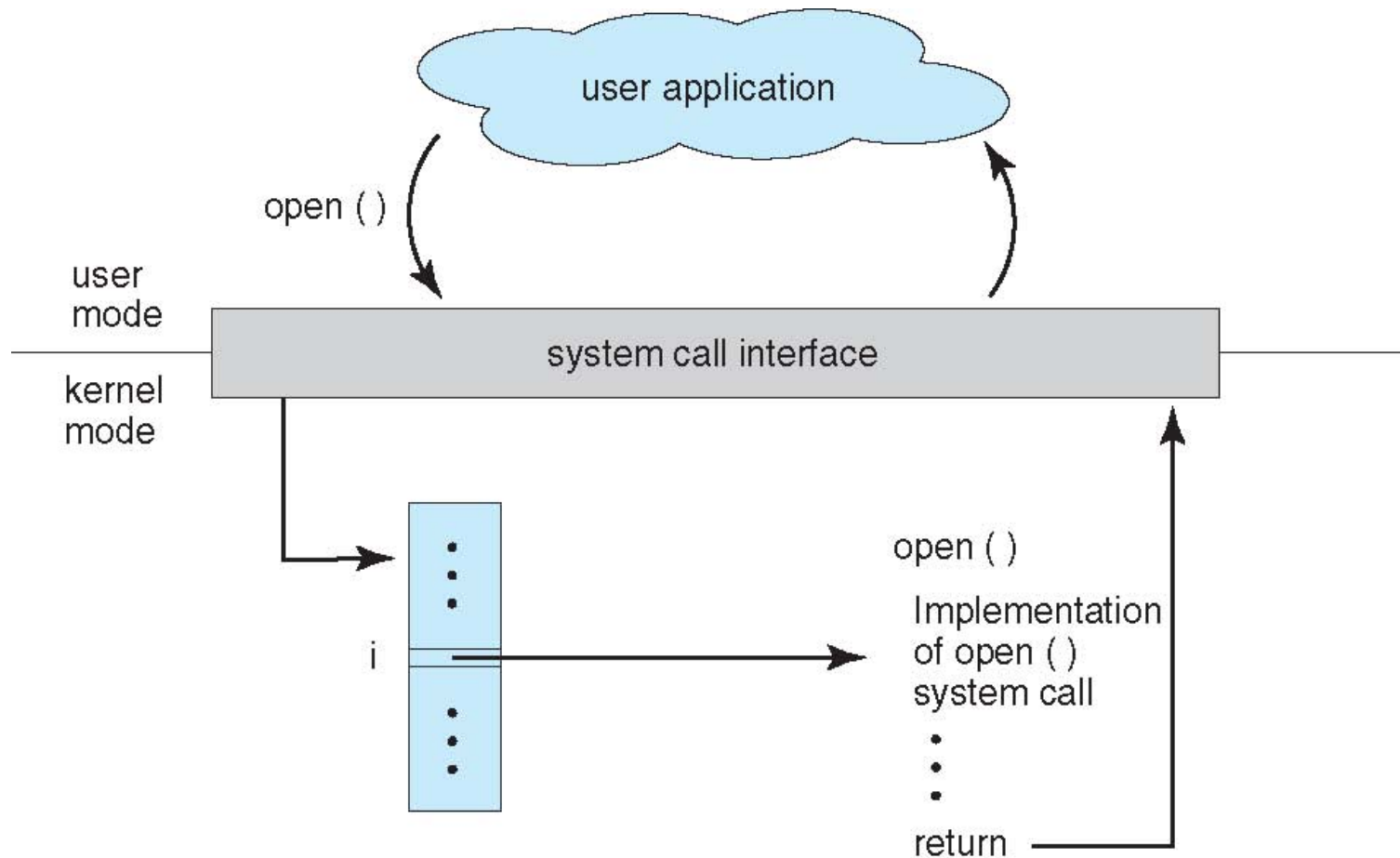
System Call Implementation

- Typically, a number associated with each system call
 - **System-call interface** maintains a table indexed by 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
- 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
 - 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





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

- 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

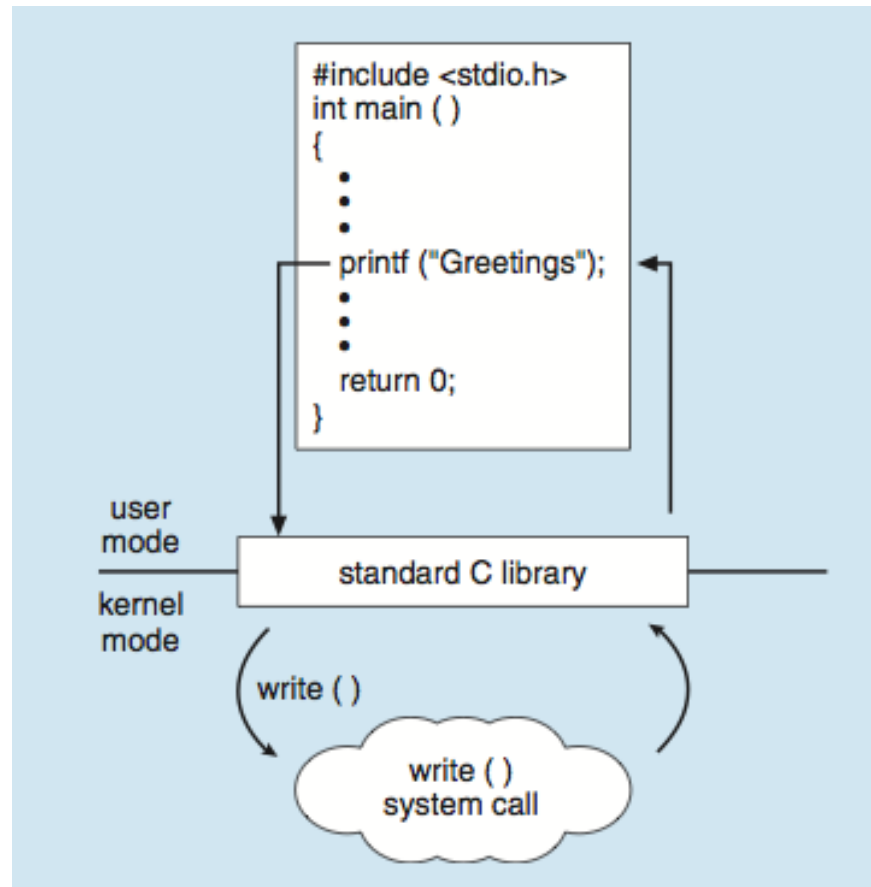
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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





Standard C Library Example

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





System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information such as date, time, memory available, etc
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
- Most users' view of the operation system is defined by system programs, not the actual system calls





System Programs

- 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 Programs (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 Programs (Cont.)

■ Background Services

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





Operating System Design and Implementation

- Design and Implementation of OS does not have a “perfect solution”, 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?
- 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. Need flexible parametrized mechanisms.
- Examples:
 - Timer construct is a mechanism for ensuring protection, but duration setting is a policy.
 - Priority scheduling is a mechanism, but which kinds of processes get higher priority is a policy
- 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
- Linux and Windows kernels are mostly written in C
- High-level language is easier to **port** to other hardware
 - Linux is written in C and is available on Intel x86, Oracle SPARC and IBM PowerPC
 - MS-DOS and early versions of Windows were written in Intel x86 Assembly and ran only on Intel x86 HW





Implementation (Cont.)

- Development in high-level language is more productive
 - Code can be written faster
 - Code is more compact
 - Code is easier to understand and debug
- High-level code can benefit from compiler technology (optimizations)
 - compiler can automatically analyze the code and do better optimization than the human mind
 - Improvements in compiler technology can be applied by recompiling





Implementation (Cont.)

- Summary of advantages of writing an OS in a high-level language:
 - Portability
 - Productivity and convenience
 - Automated code optimization by a compiler
- Only possible disadvantages are reduced speed and increased storage, but
 - Compilers often do a better job at analyzing and optimizing large-scale code on a complex modern architecture
 - Major performance improvements are likely to be the result of better algorithms and data structures as opposed to better coding
 - Only small sections of the OS are performance critical, e.g. interrupt handler, CPU scheduler, I/O manager, memory manager
 - ▶ Bottleneck routines may be identified and written in assembly





Operating System Structure

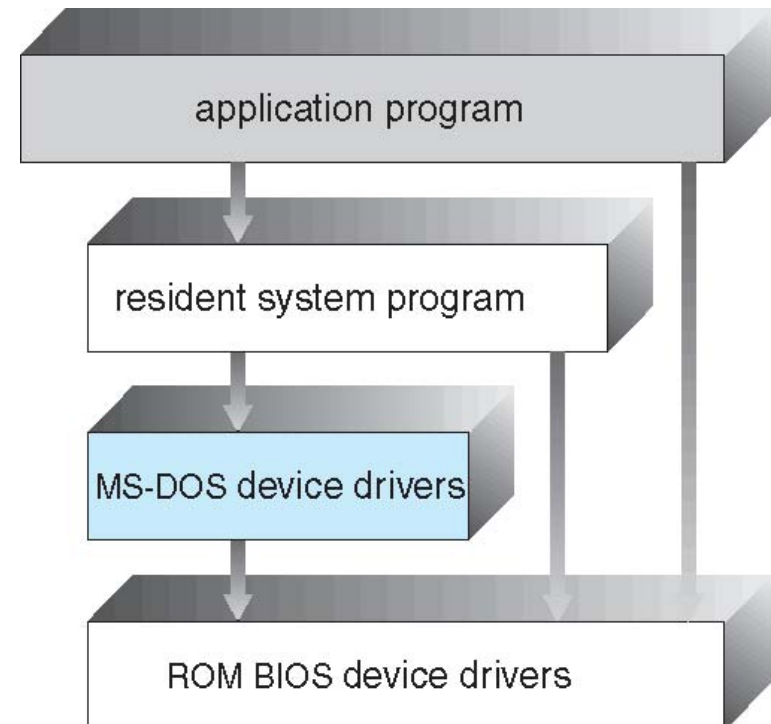
- General-purpose OS is very large program
- Various ways to structure it:
 - Simple structure (Monolithic), e.g., MS-DOS, original Unix
 - Layered – an abstraction
 - Microkernel, e.g., Mach
 - Modules
 - Hybrid Systems





Simple Structure (Monolithic) -- MS-DOS

- 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
 - Application programs can access BIOS to write directly to the display and disk drives





Limited Structuring -- UNIX

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

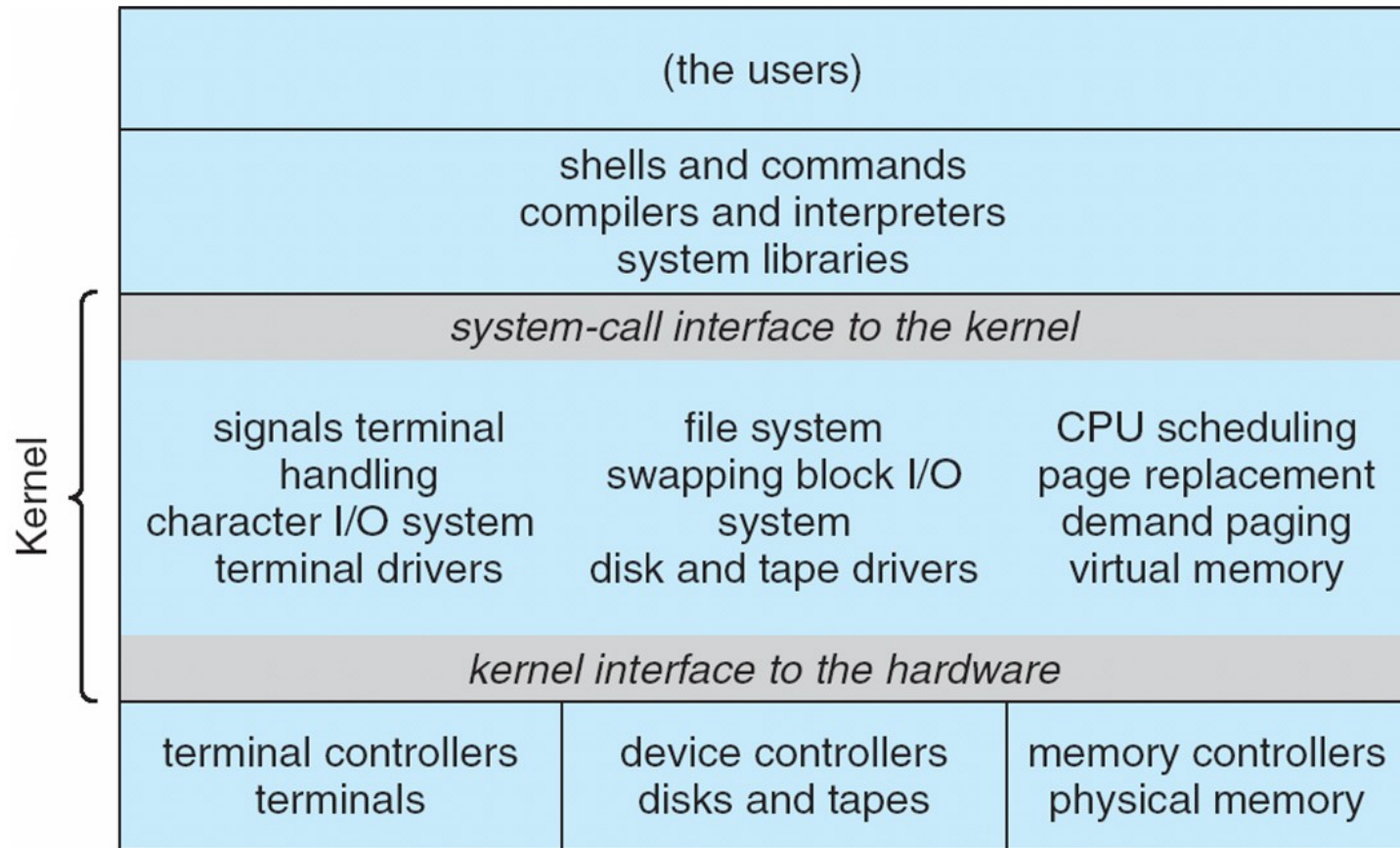
- System programs
- The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
- Difficult to implement and maintain
- Performance advantage due to little overhead in system-call interface and communication within the kernel





Traditional UNIX System Structure

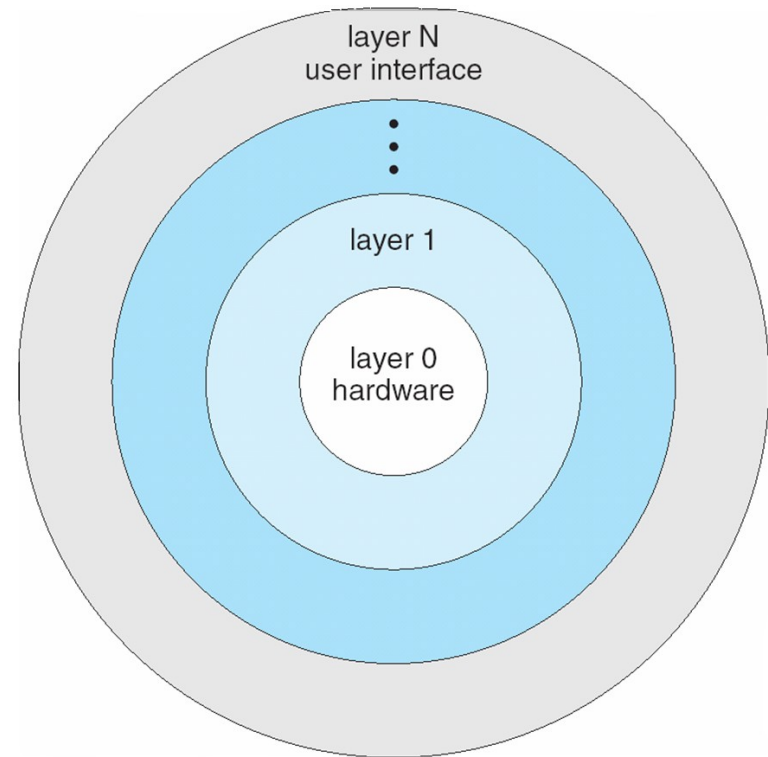
Beyond simple but not fully layered





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
- Simplifies construction and debugging
- Hard to define layers, e.g., backing-store driver and CPU scheduler
- Less efficient due to the overhead of passing and modifying parameters through multiple layers





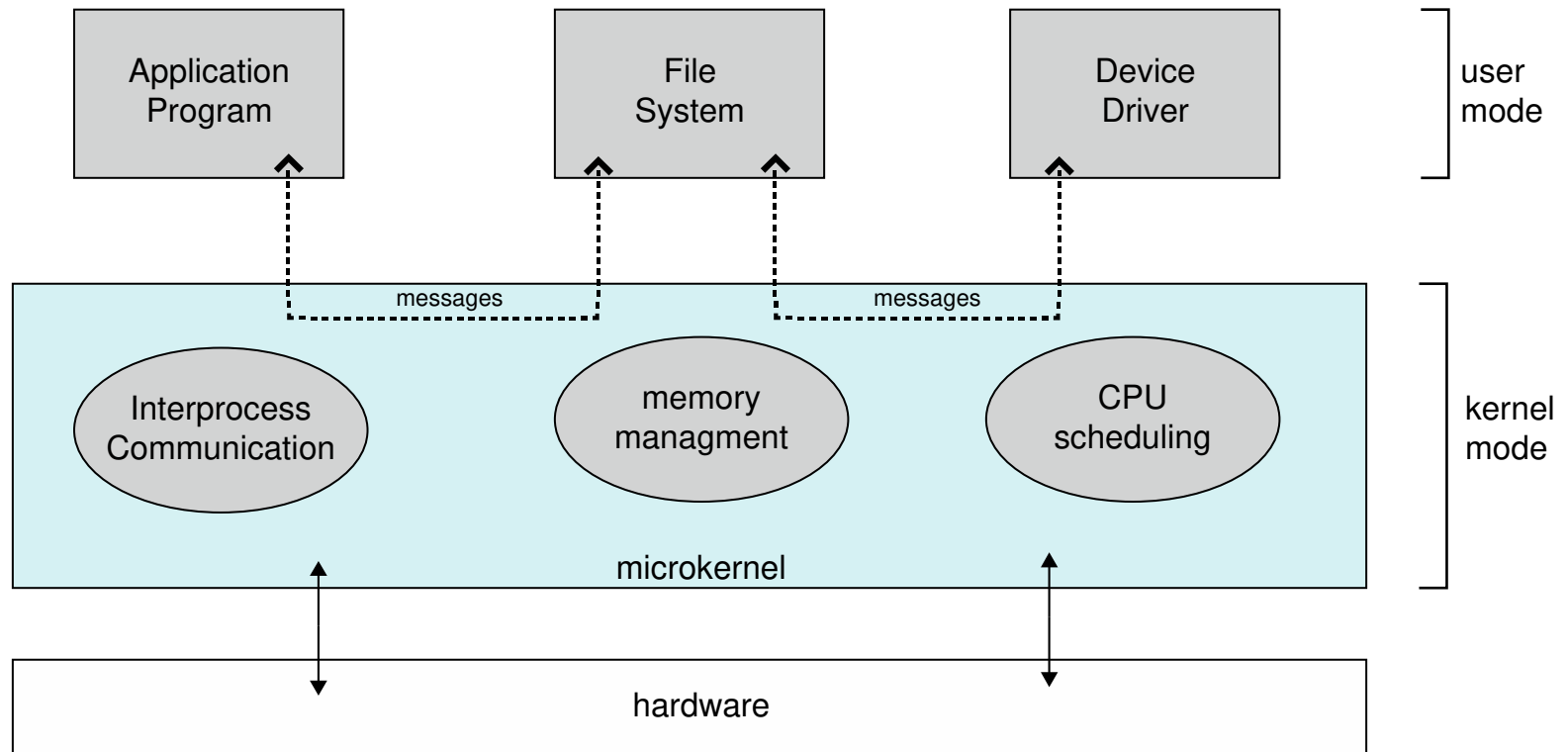
Microkernel System Structure

- Moves nonessential components from the kernel into user space
- **Mach** example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach (CMU, 80s)
- Communication takes place between modules using **message passing** through the kernel
- Benefits:
 - Easier to extend OS without recompiling the kernel
 - Easier to port the operating system to new architectures
 - More reliable and secure (less code is running in kernel mode); if a service fails, the rest of the OS remain intact
- Disadvantage:
 - Performance overhead of user space to kernel space communication
 - ▶ Win NT was microkernel but grew into monolithic Win XP





Microkernel System Structure





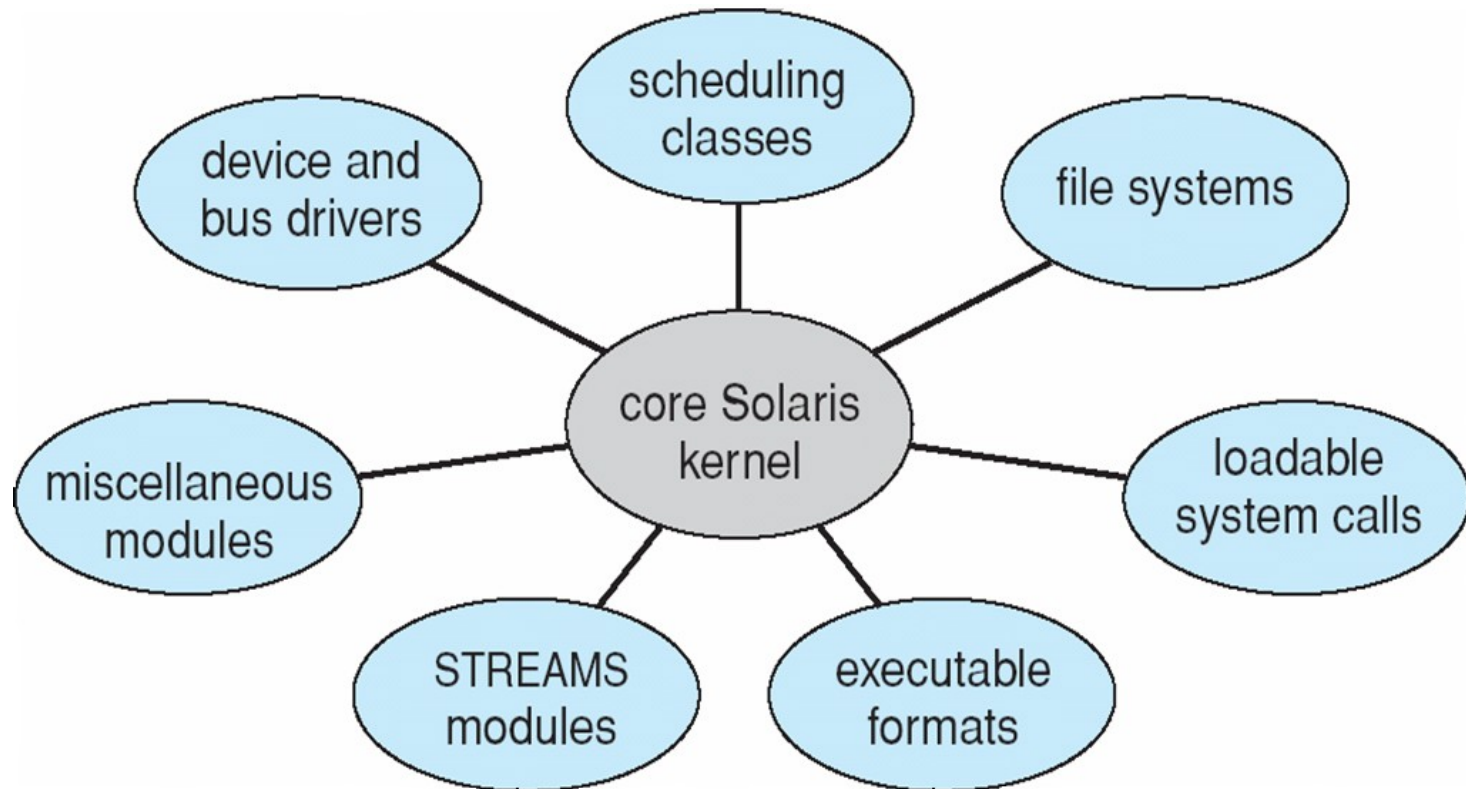
Modules

- Kernel provides core services; other services are provided as **loadable kernel modules** (boot time or run time)
 - Each core component is separate
 - Each talks to others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, similar to layered but more flexible, because any module can call any other module (no hierarchy)
- Also similar to microkernel but more efficient, because modules do not need to invoke message passing
- **Perhaps** the best current methodology for OS design
- Common design in modern Unix systems (Solaris, Linux, Mac OS) and Windows





Solaris Modular Approach





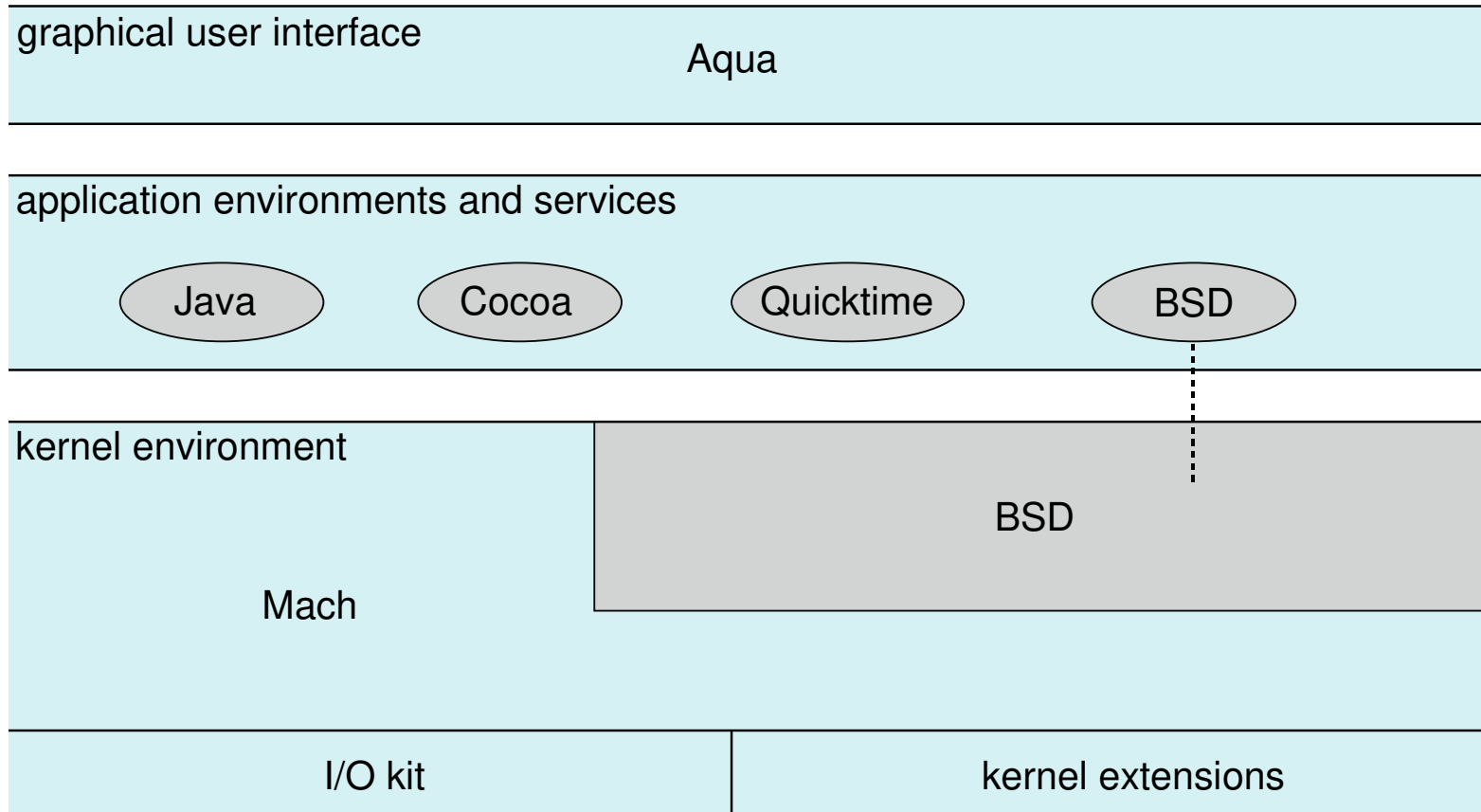
Hybrid Systems

- Most modern operating systems do not adopt one pure model
 - Hybrid combines multiple approaches to address performance, security, usability needs
 - Linux and Solaris are monolithic, because the OS runs in a single address space (for performance) but modular for dynamic loading of some functionalities
 - Windows mostly monolithic (for performance), but with microkernel features such as providing support for different subsystems (**personalities**) in user mode; it also supports loadable kernel modules
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
 - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)





Mac OS X Structure





iOS

- Apple mobile OS for ***iPhone***, ***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 mobile apps
 - **Media services** layer for graphics, audio, video
 - **Core services** provides cloud computing, databases
 - Core operating system, based on Mac OS X kernel (shown in previous slide)

Cocoa Touch

Media Services

Core Services

Core OS





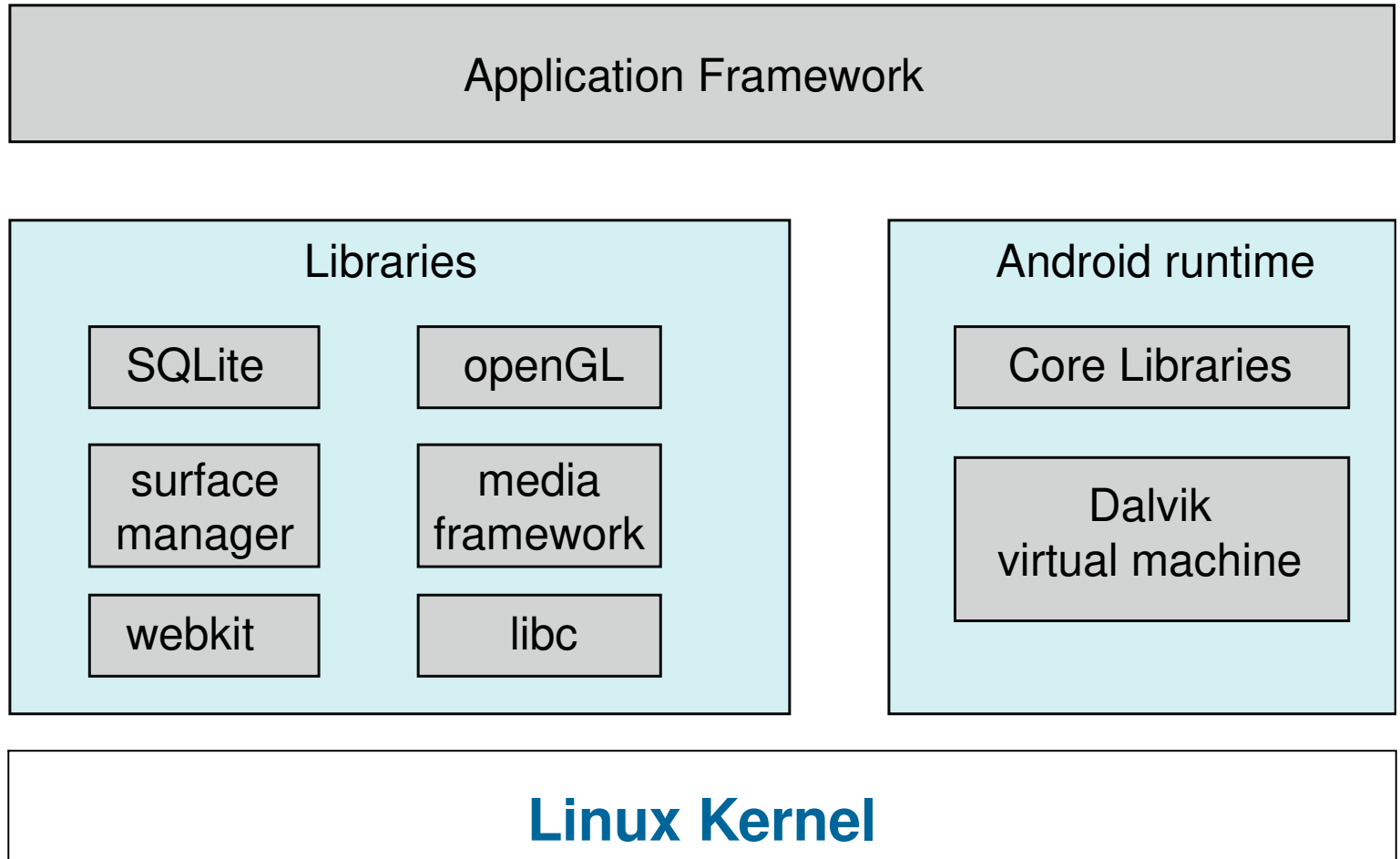
Android

- Developed by Open Handset Alliance (mostly Google)
 - Open Source
 - Runs on a variety of mobile platforms
- 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 that runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc





Android Architecture



End of Chapter 2

