

Operating Systems

Prof. Yongkun Li

中科大-计算机学院 特任教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 2 Operating System Structures

Objectives

- Operating System Services
 - User Operating System Interface
 - System Calls
- Operating System Structure
- Operating System Design and Implementation
- MISC: Debugging, Generation & System Boot

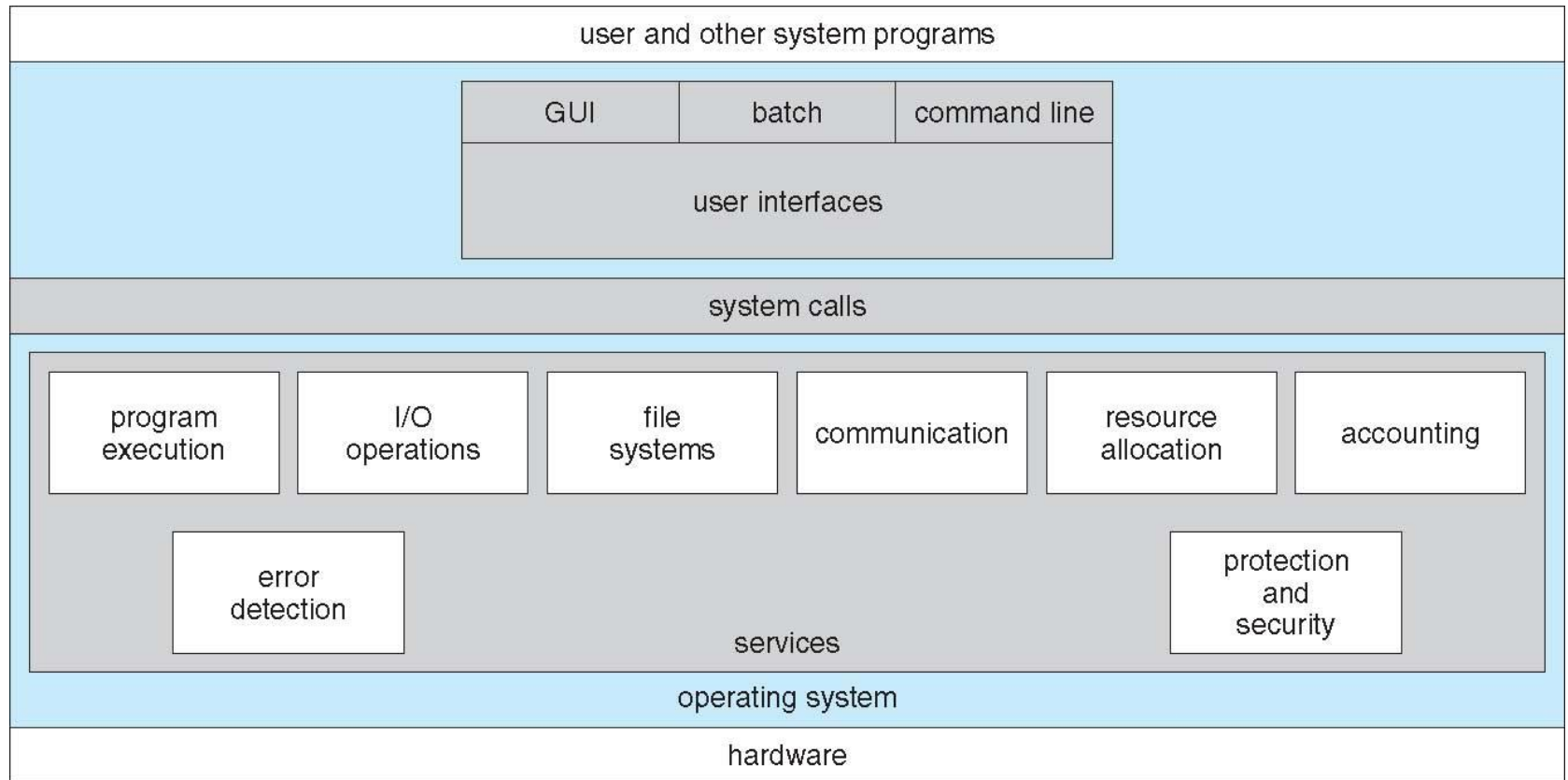
Operating System Services

Services Overview, User Interface

Operating System Services

- Operating systems provide
 - an environment for **execution** of programs and
 - **services** to programs and users
- Services may differ from one OS to another
- What are the common classes?
 - **Convenience** of the user
 - **Efficiency** of the system

Overview of Operating System Services



OS Services for Helping Users

- **Program execution**
 - Load a program into memory
 - Run the program
 - End execution
 - either normally or
 - abnormally (indicating error)

OS Services for Helping Users

- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
 - Common I/Os: read, write, etc.
 - Special functions: recording CD/DVD
- Notes: Users usually cannot control I/O devices directly, so OS provides a mean to do I/O
 - Mainly for efficiency and protection

OS Services for Helping Users

- **File-system manipulation** - The file system is of particular interest
 - OS provides a variety of file systems
- Major services
 - read and write files and directories
 - create and delete files and directories
 - search for a given file
 - list file information
 - permission management: allow/deny access

OS Services for Helping Users

- **Communications:** information exchange between processes
 - Processes on the same computer
 - Processes between computers over a network
- Implementations
 - **Shared memory**
 - Two or more processes read/write to a shared section of mem.
 - **Message passing**
 - Packets of information are moved between processes by OS

OS Services for Helping Users

- **Error detection** – OS needs to be constantly aware of possible errors
- **Error types**
 - CPU
 - memory hardware: memory error, power failure, etc.
 - I/O devices: parity error, connection failure, etc.
 - user program: arithmetic overflow, access illegal mem.
- **Error handling**
 - Ensure correct and consistent computing
 - Halt the system, terminate an error-causing process etc.

OS Services for Ensuring Efficiency

- Systems with multiple users can gain efficiency by sharing the computer resources
- **Resource allocation**
 - Resources must be allocated to each user/job
 - **Resource types** - CPU cycles, main memory, file storage, I/O devices
 - Special allocation code may be required, e.g., CPU scheduling routines depend on
 - Speed of the CPU, jobs, number of registers, etc.

OS Services for Ensuring Efficiency

- **Accounting** - To keep track of
 - which users use how much and what kinds of resources
- Usage
 - Accounting for **billing** users
 - Accumulating **usage statistics**, can be used for
 - Reconfiguration of the system
 - Improvement of the efficiency

OS Services for Ensuring Efficiency

- **Protection and security**
 - Concurrent processes should not interfere w/ each other
 - Control the use of computer
- **Protection**
 - Ensure that all access to system resources is **controlled**
- **Security**
 - **User authentication** by password to gain access
 - Extends to **defending external I/O devices** from invalid access attempts

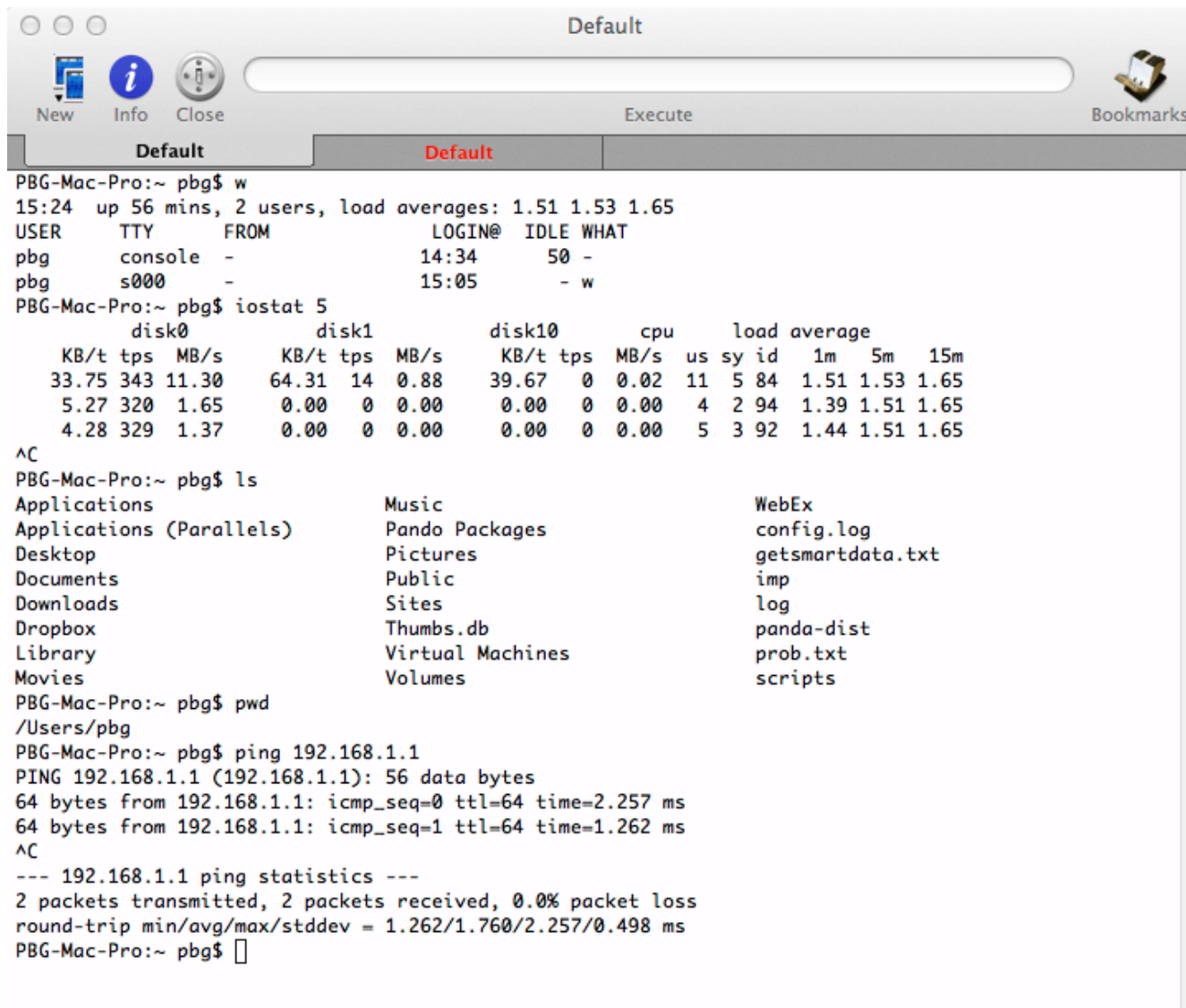
OS Services for Helping Users

- **User interface** - Almost all operating systems have a user interface (**UI**).
 - Three forms
 - **Command-Line (CLI)**
 - Shell command
 - **Batch**
 - Shell script
 - **Graphics User Interface (GUI)**
 - Windows system

User Operating System Interface - CLI

- Command line interface or command interpreter
 - Allows direct command entry
 - Included in the kernel or treated as a special program
- Sometimes multiple flavors implemented – **shells**
 - Linux: multiple shells (C shell, Korn Shell etc.)
 - Third-party shell or free user-written shell
 - Most shells provide similar functionality (personal preference)

Bourne Shell Command Interpreter



```
PBG-Mac-Pro:~ pbg$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER      TTY      FROM          LOGIN@  IDLE WHAT
pbg       console -             14:34   50  -
pbg       s000    -             15:05   -  w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0      disk1      disk10      cpu      load average
      KB/t tps MB/s    KB/t tps MB/s    KB/t tps MB/s  us sy id  1m  5m  15m
      33.75 343 11.30    64.31 14  0.88    39.67  0  0.02  11  5 84  1.51 1.53 1.65
      5.27 320  1.65     0.00  0  0.00     0.00  0  0.00   4  2 94  1.39 1.51 1.65
      4.28 329  1.37     0.00  0  0.00     0.00  0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications          Music                 WebEx
Applications (Parallels)  Pando Packages      config.log
Desktop               Pictures              getsmartdata.txt
Documents             Public                imp
Downloads             Sites                 log
Dropbox               Thumbs.db            panda-dist
Library               Virtual Machines     prob.txt
Movies                Volumes              scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$
```


User Operating System Interface - CLI

- **Main function** of CLI
 - Get and execute the next user-specified command
 - Many commands manipulate files
- Two ways of **implementing commands**
 - The command interpreter itself contains the code
 - Jump to a section of its code & make appropriate system call
 - Number of commands determines the size of CLI
 - Implements commands through system program (UNIX)
 - CLI does not understand the command
 - Use the command to identify a file to be loaded into memory and executed
 - Exp: `rm file.txt` (search for file `rm`, load into memory and execute w/ `file.txt`)
 - Add new commands easily

User Operating System Interface - GUI

- User-friendly graphical user interface
 - **Mouse-based window-and-menu system** (**desktop** metaphor)
 - **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 in early 1970s
- 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
 - 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



Choices of Interfaces

- Personal preference
- CLI: **more efficient, easier for repetitive tasks**
 - System administrator
 - Power users who have deep knowledge of a system
 - Shell scripts
- GUI: user-friendly
- The design and implementation of user interface is **not a direct function** of the OS

System Call

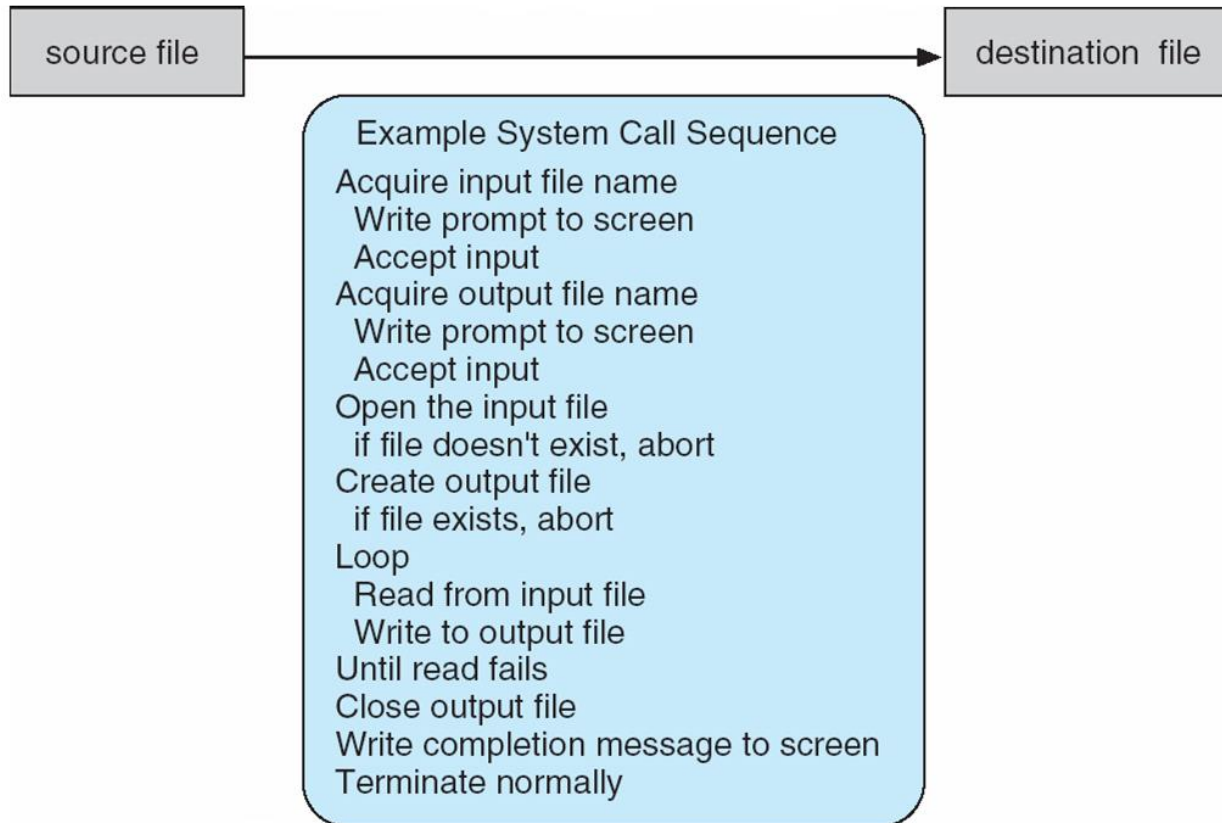
Usage, Implementation, Types

System Calls

- **Programming interface** to the services provided by the OS
- Implementation language
 - Typically written in a high-level language (C or C++)
 - Certain low-level tasks (direct hardware access) are written using assembly language
- **Example** of using system call
 - Read data from a file and copy to another file
 - `open()`+ `read()` + `write()`?

Example of System Calls

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



System Call

- Simple programs may make heavy use of the OS
 - A system executes thousands of system calls per second
 - Not user-friendly
- Each OS has its own name for each system call
 - This course/textbook uses generic examples

System Call

- How to use?
 - Mostly accessed by programs via a high-level **API** rather than direct system call use
- Why prefer API rather than invoking system call?
 - **Easy of use**
 - Simple programs may make heavy use of the OS
 - **Program portability**
 - Compile and run on any system that supports the same API

- **Application Programming Interface (API)**
 - A set of functions that are available to application programmers

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

| | | |
|--------------|---------------|------------|
| return value | function name | parameters |
|--------------|---------------|------------|

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

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

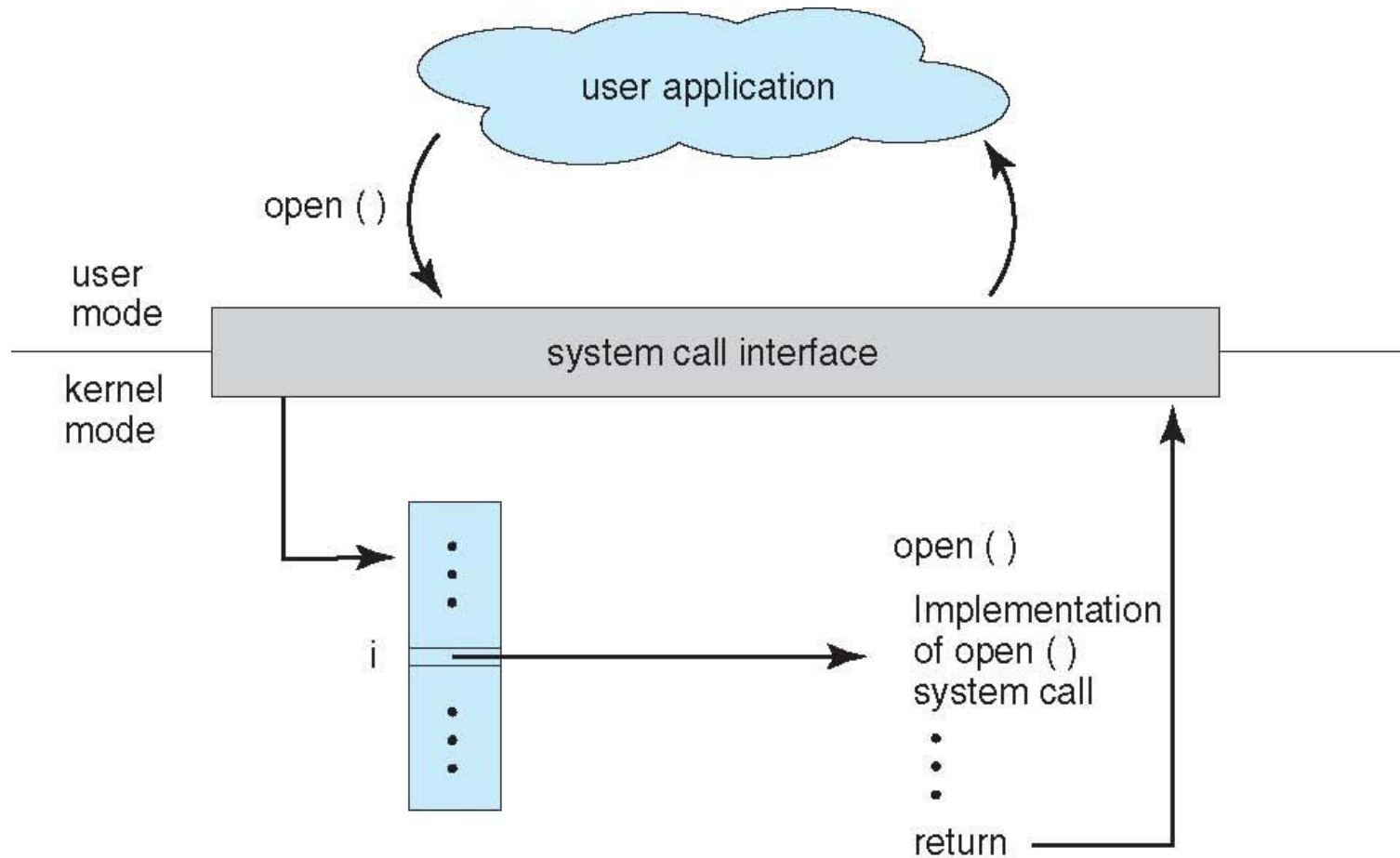
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

- **Application Programming Interface (API)**
 - A set of functions that are available to application programmers
- Three most common APIs
 - Win32 API for Windows
 - POSIX API for POSIX-based systems
 - including virtually all versions of UNIX, Linux, and Mac OS X
 - Java API for the Java virtual machine (JVM)
- How to use API?
 - Via a library of code provided by OS
 - Libc: UNIX/LINUX with C language

System Call Implementation

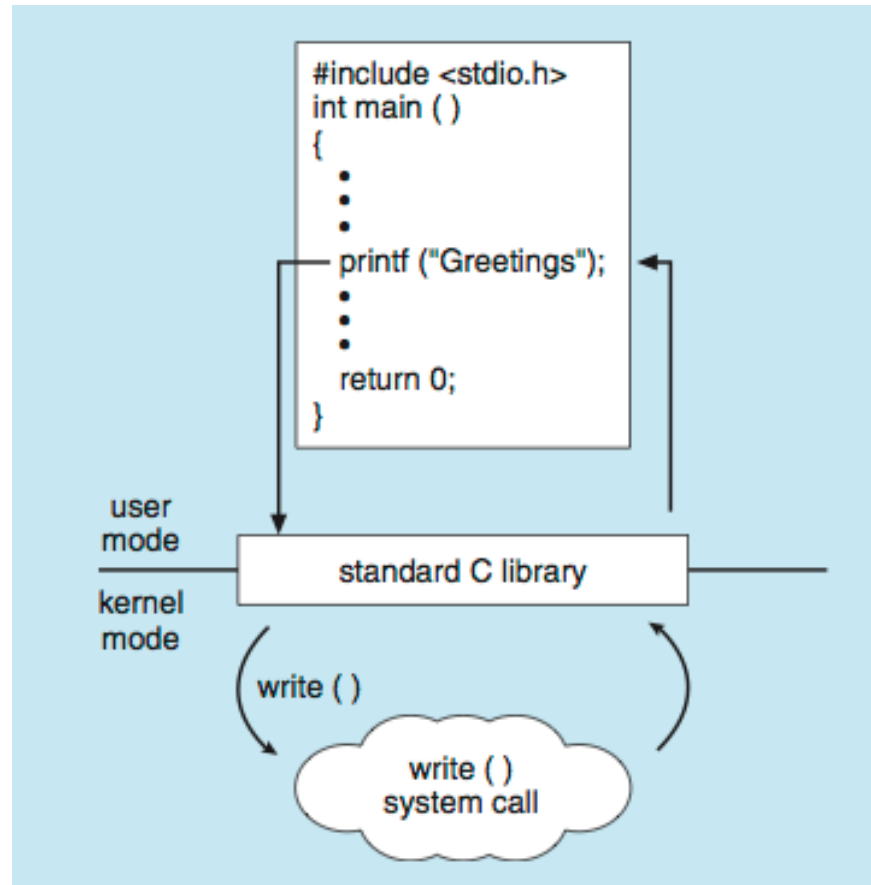
- Who invokes system call: **System call interface**
 - Provided by the **run-time support system**, which is
 - a set of functions built into libraries within a compiler
- How?
 - intercepts function calls in the API
 - invokes necessary system calls
- Implementation
 - Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to the numbers

API – System Call – OS Relationship



Standard C Library Example

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



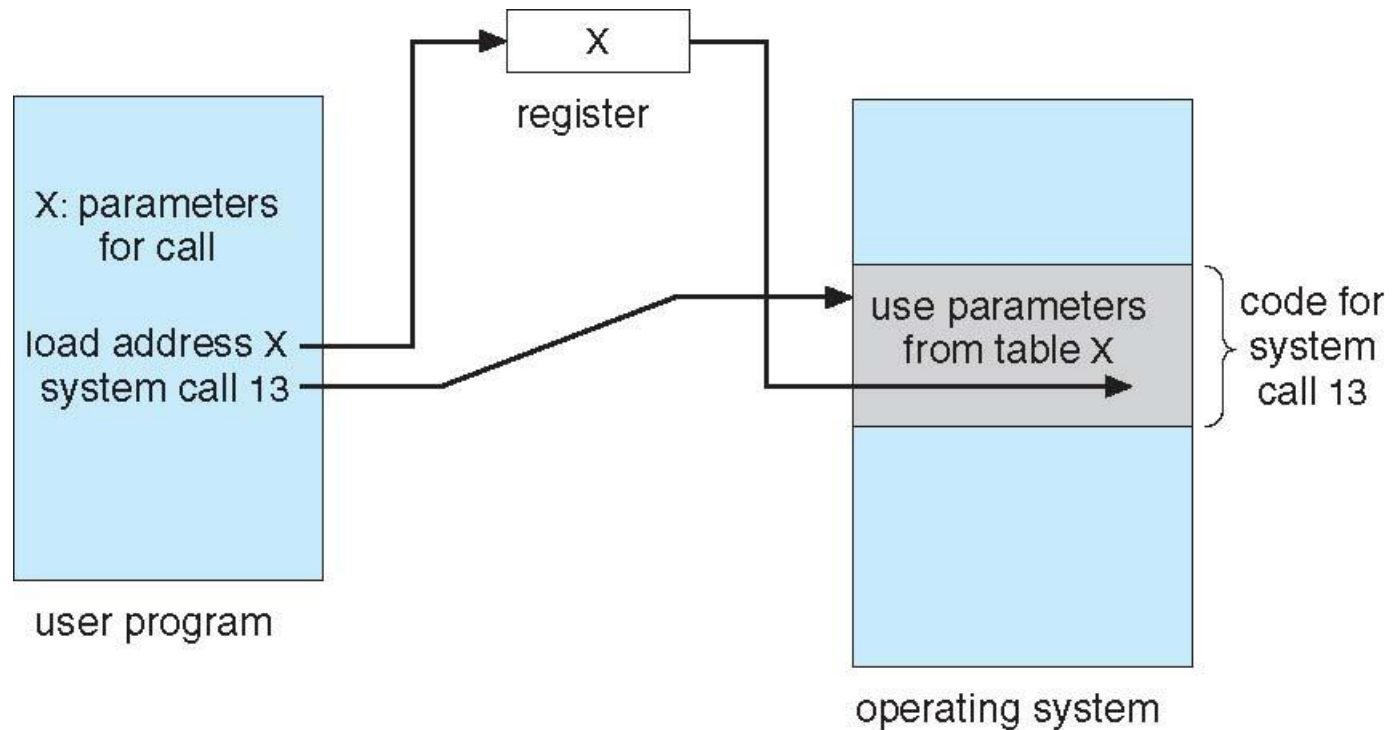
Implementation Benefits

- The caller needs to know nothing about
 - how the system call is implemented
 - what it does during execution
 - Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface are hidden from programmer by API
 - Managed by run-time support library

System Call Parameter Passing

- More information is required than simply the identity of desired system call
 - Parameters: file, address and length of buffer
- Three methods to **pass parameters** to the OS
 - Simplest: pass the parameters in **registers**
 - In some cases, may be more parameters than registers
 - Table-based
 - 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
 - Stack-based
 - Parameters are placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

Parameter Passing via Table



Types of System Calls

- Six major categories
 - Process control
 - File manipulation
 - Device manipulation
 - Information maintenance
 - Communications
 - Protection

Types of System Calls

- Process control
 - `end()`, `abort()`
 - Halt a running program normally or abnormally
 - Transfer control to invoking command interpreter
 - Memory dump & error message
 - Written to disk and examined by debugger
 - Respond to error: alert window (GUI system) or terminate the entire job (batch system)
 - Error level: normal termination (level 0)

Types of System Calls

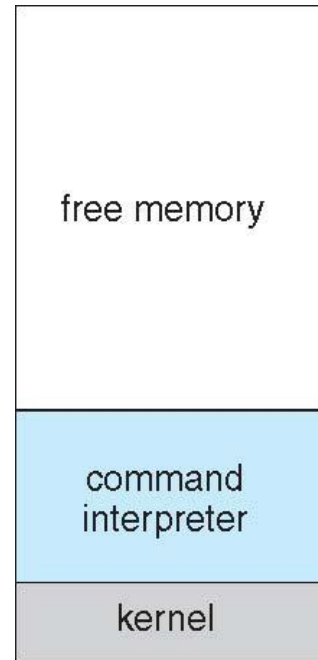
- Process control
 - `end()`, `abort()`
 - `load()`, `execute()`
 - Where to return?
 - Return to existing program: save mem. image
 - Both programs continue concurrently: multiprogram
 - `create_process()`, `terminate_process()`
 - `get_process_attributes()`, `set_process_attributes()`
 - Job's priority, maximum allowable execution time, etc

Types of System Calls

- Process control
 - `end()`, `abort()`
 - `load()`, `execute()`
 - `create_process()`, `terminate_process()`
 - `get_process_attributes()`, `set_process_attributes()`
 - `wait_time()`
 - `wait_event()`, `signal_event()`
 - `acquire_lock()`, `release_lock()`

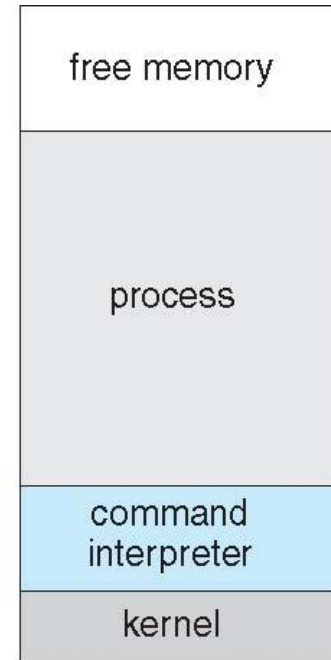
Example of Process Control: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
 - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup

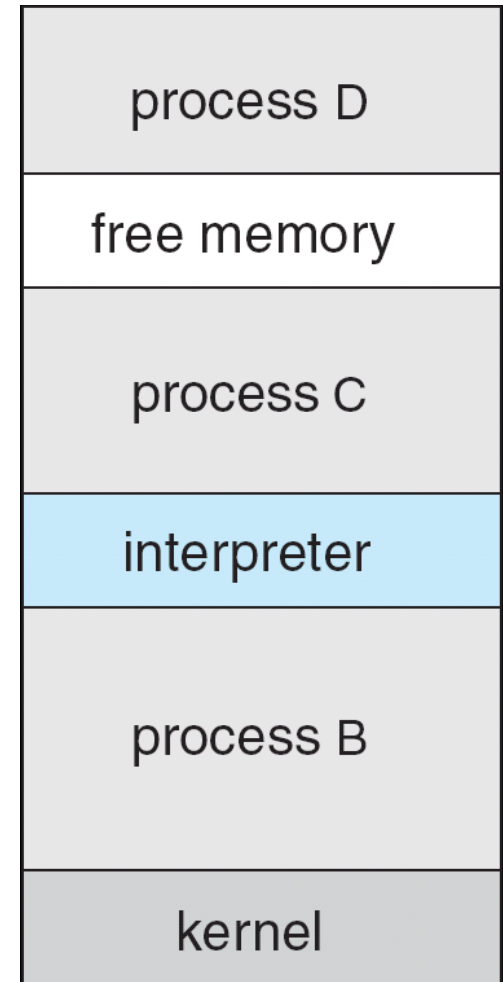


(b)

running a program

Example of Process Control: FreeBSD

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



Types of System Calls

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

Types of System Calls

- Information maintenance
 - Get time or date, set time or date
 - Get system data, set system data
 - Num. of current users, os version, amount of free mem. & disk
 - Debugging
 - Dump memory
 - Single-step execution
 - Time profile: timer interrupt
 - The amount of time that the program executes at a particular location

Types of System Calls

- Communications
 - Message-passing model
 - Host name, IP, process name
 - `Get_hostid()`, `get_processid()`, `open_connection()`, `close_connection()`, `accept_connection()`, `read_message()`, `write_message()`
 - Useful for exchanging smaller amounts of data
 - Shared-memory model
 - Remove the normal restriction of preventing one process from accessing another process's memory
 - Create and gain access to shared mem. region
 - `shared_memory_create()`, `shared_memory_attach()`
 - Threads: memory is shared by default
 - Efficient and convenient, having protection and synchronization issues

Types of System Calls

- Protection
 - Control access to resources
 - All computer systems must be concerned
 - Permission setting
 - `get_permission()`, `set_permission()`
 - Allow/deny access to certain resources
 - `allow_user()`, `deny_user()`

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() |

<https://www.kernel.org/doc/man-pages/>
<http://man7.org/linux/man-pages/>

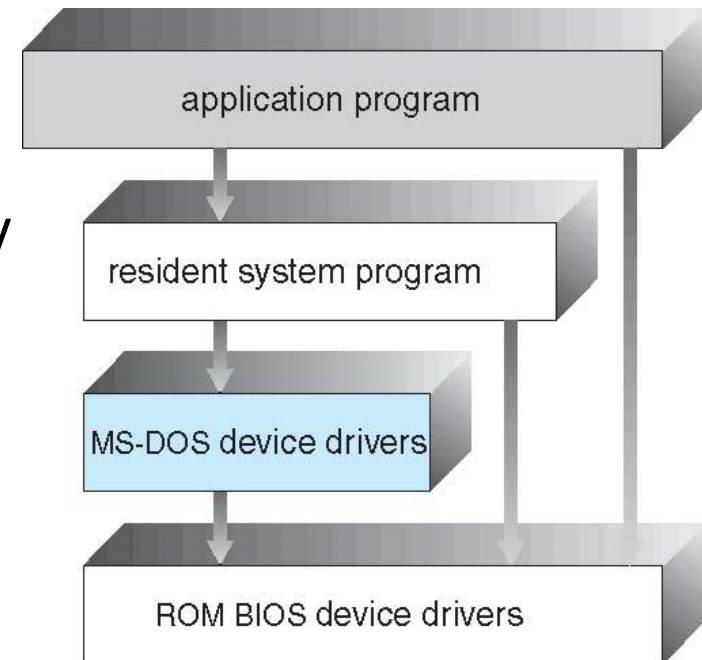
Operating System Structures

Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - Monolithic-- UNIX
 - Layered – an abstraction
 - Microkernel –Mach
 - Modules
 - Hybrid system – most OSes

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the **least space**
 - Do not have well-defined structures
 - Not divided into modules
 - Its interfaces and levels of functionality are not well separated
 - Application programs can access basic I/O routines
 - Vulnerable to errant programs
 - Limited by hardware



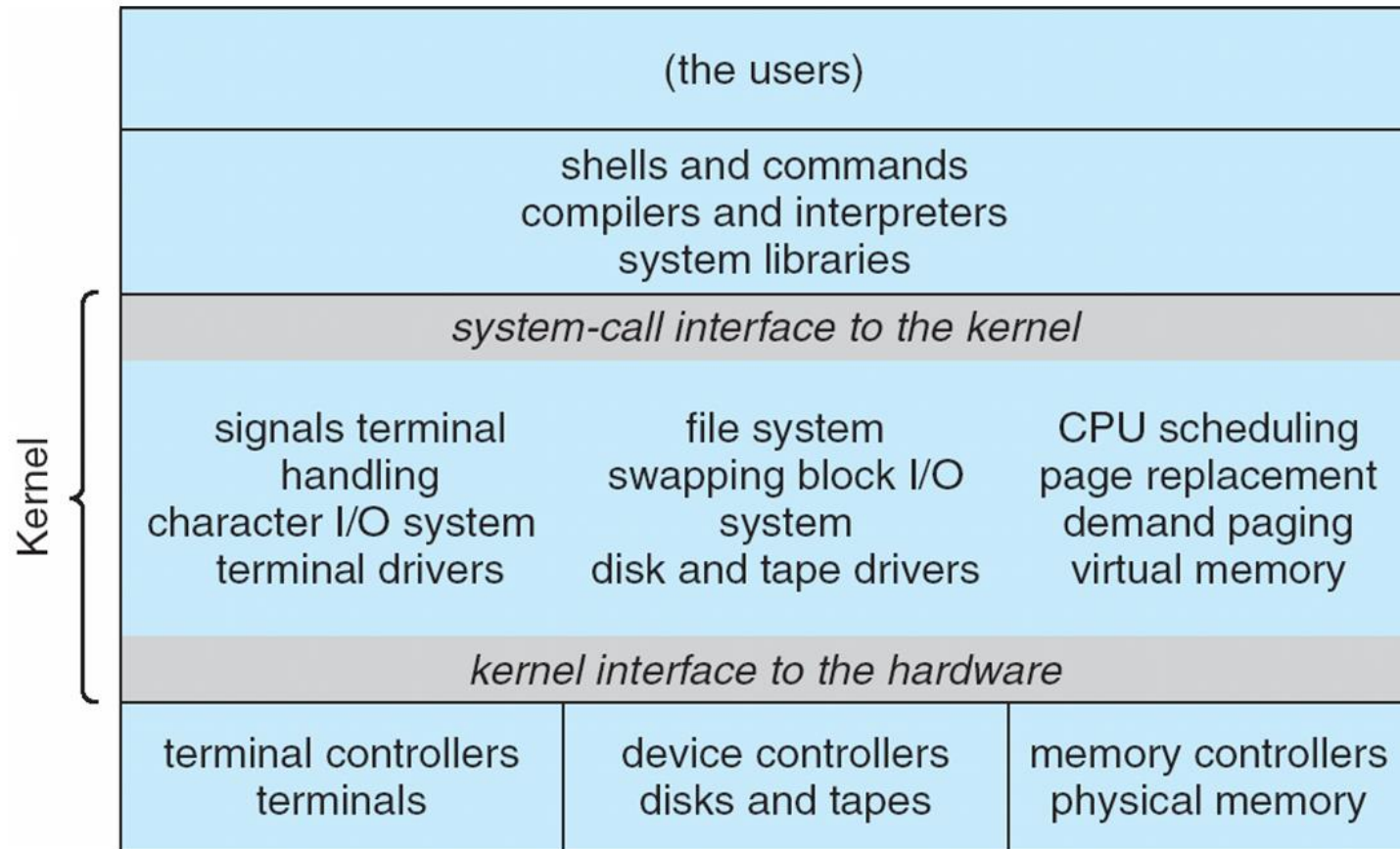
Monolithic Structure -- UNIX

- UNIX

- The original UNIX operating system had **limited structuring**, it consists of **two separable parts**
 - Systems programs
 - The kernel
 - Consists of **everything** below the system-call interface and above the physical hardware
 - A series of interfaces and device drivers
- **Monolithic structure**: combine all functionality in one level
 - File system, CPU scheduling, memory management, and other operating-system functions
 - Difficult to implement and maintain
 - Performance advantage

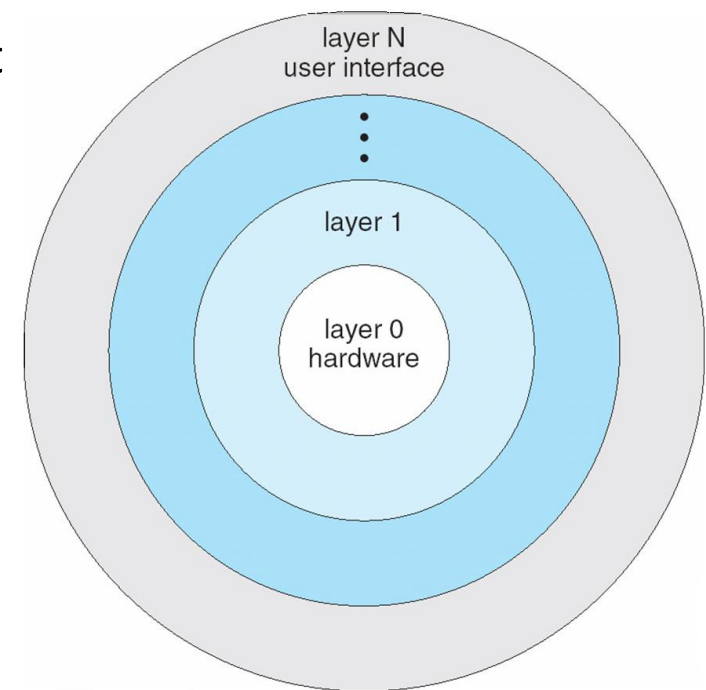
Traditional UNIX System Structure

- Beyond simple but not fully layered



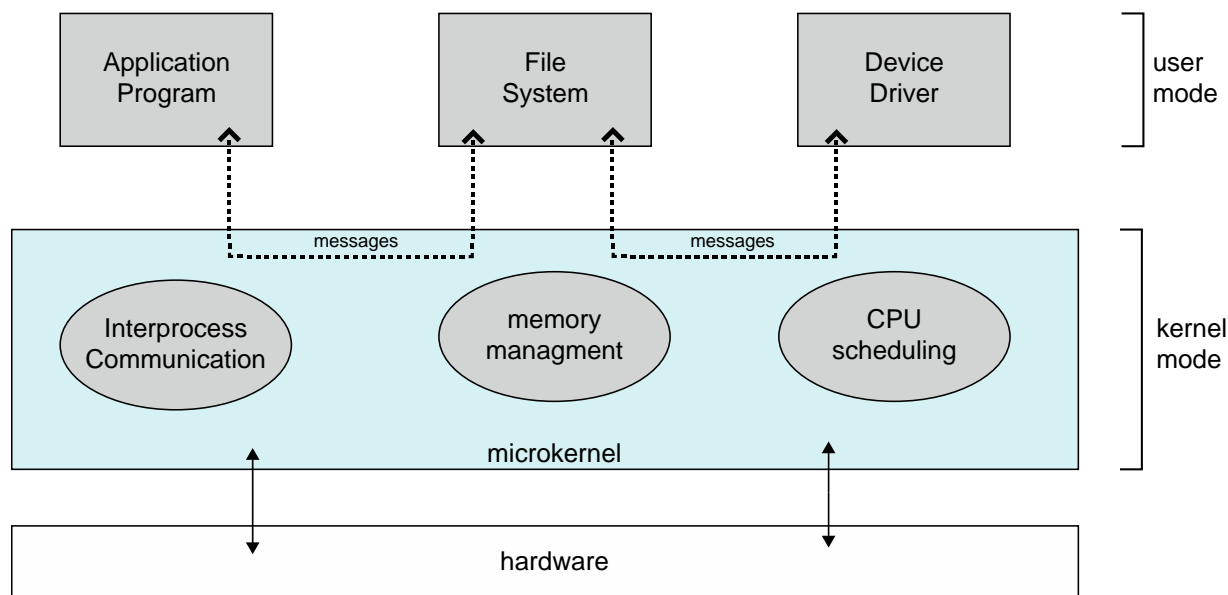
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers
 - The bottom layer (0), is the hardware; the highest layer (N) is the user interface
- Implementation
 - Each layer is an implementation of an abstract object made up of data and operations
- Advantages
 - Simple to construct and debug
 - Hides the existence of DS, Ops, hardware from upper layers
- Challenges
 - How to define various layers?
 - Efficiency problem
 - I/O->memory manage->CPU scheduling->hardware



Microkernel System Structure

- Moves as much from the kernel into user space as possible
 - Provides minimal process, memory management and communication
 - **Mach**: example of **microkernel** (developed by CMU in mid-1980s)
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Main function
 - Communication between client program and services (also in user space)
 - Provided through **message passing**

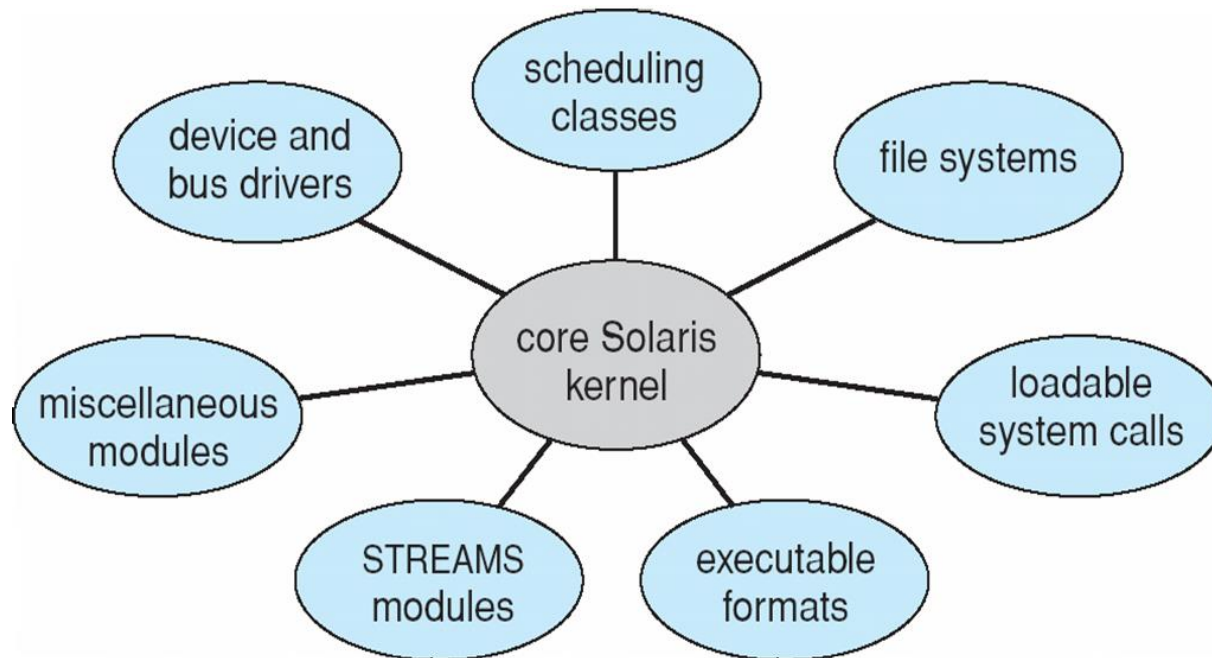


Microkernel System Structure

- Moves as much from the kernel into user space as possible
 - Provides minimal process, memory management and communication
 - **Mach**: example of **microkernel** (developed by CMU in mid-1980s)
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Main function
 - Communication between client program and services (also in user space)
 - Provided through **message passing**
- Benefits
 - Easier to extend a microkernel: add services to user space, no changes to kernel
 - Easier to port the operating system to new architectures
 - More reliable & more secure (less code is running in kernel mode)
- Detriments
 - Performance overhead of user space to kernel space communication

Modules

- Many modern operating systems implement **loadable kernel modules**
 - The Kernel has a set of core components
 - Links in additional services via modules (boot time or run time)
 - Common in most modern OSes



Modules

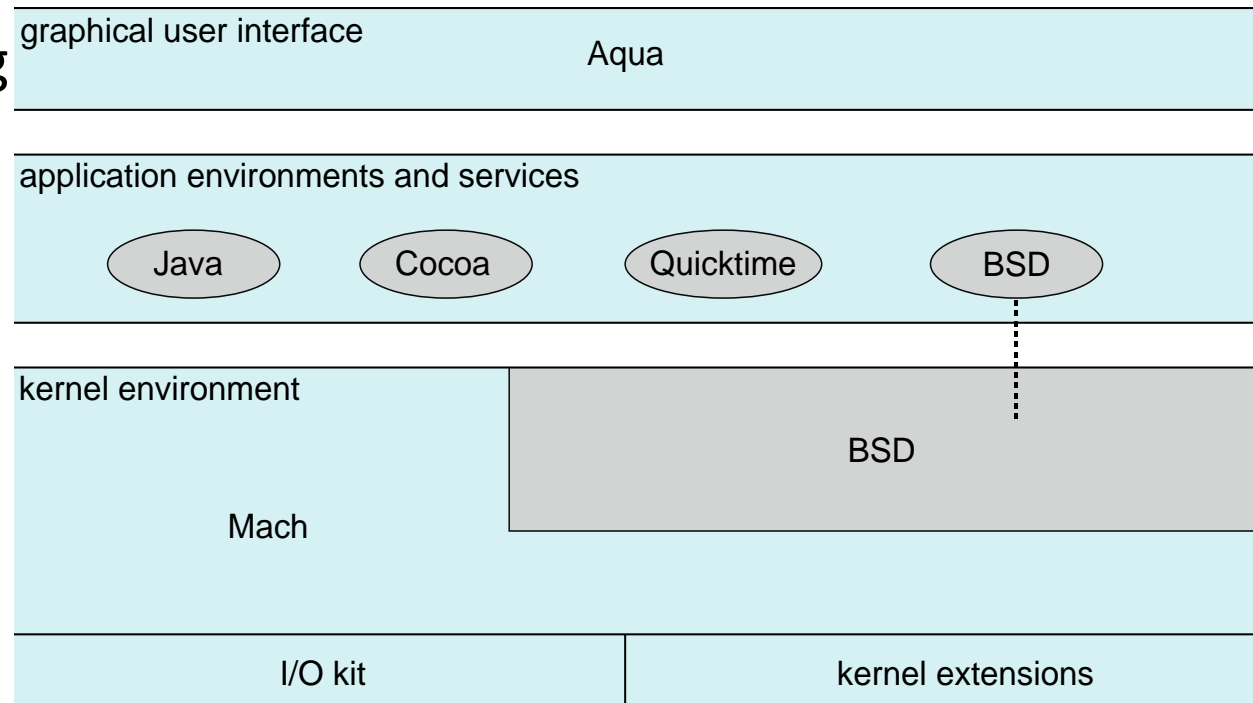
- Many modern operating systems implement **loadable kernel modules**
 - The Kernel has a set of core components
 - Links in additional services via modules (boot time or run time)
 - Common in most modern OSes
- Similar to layered system
 - Any module can call any other model
 - **More flexible**
- Similar to the microkernel
 - Primary module has only core functions
 - No need to invoke message passing
 - **More efficient**

Hybrid Systems

- Most modern operating systems combine different structures, resulting in **hybrid systems**
 - Why? Address performance, security, usability needs
- Examples
 - Linux kernel
 - Monolithic: single address space (**for efficient performance**)
 - Modular: dynamic loading of functionality
 - Windows
 - Mostly monolithic, plus microkernel for different subsystem personalities (running in user-mode), also support loadable kernel module
 - Apple Mac OS X
 - Mach microkernel, BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure

- Layered system: user interface + application environment & services + kernel (Mach+BSD UNIX)
- Mach Microkernel
 - Memory management
 - inter-process communication
 - Thread scheduling
- BSD UNIX
 - CLI
 - POSIX API
 - Networking
 - File system



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

Media Services

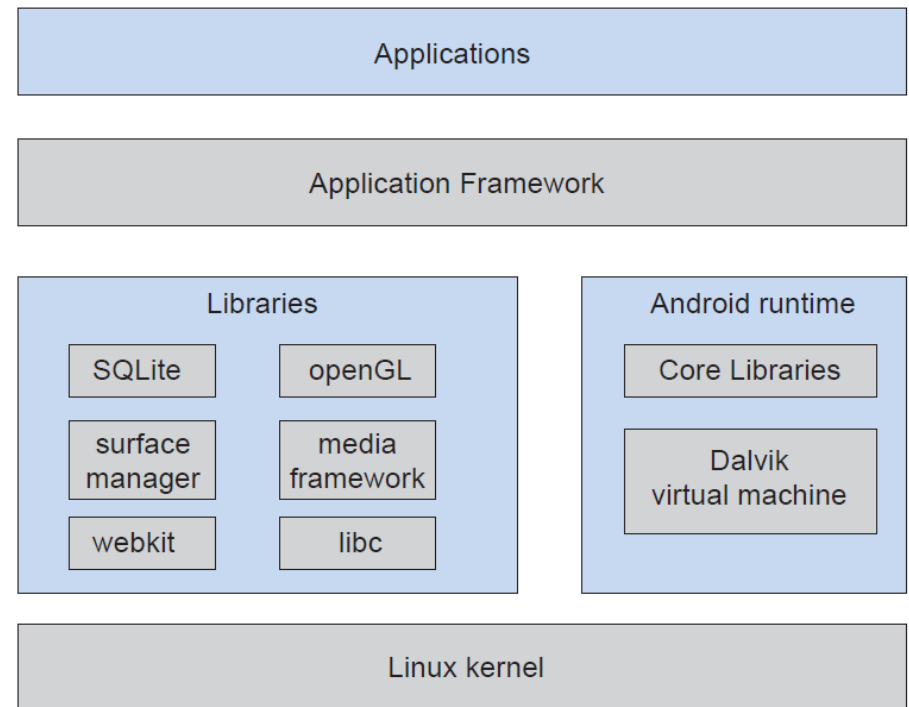
Core Services

Core OS

- Structure
 - **Cocoa Touch** is 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)
 - Similar stack to IOS
 - Open Source
- Based on Linux kernel
 - Provides process, memory, device-driver management
- Optimization
 - Adds power management



Operating System Design and Implementation

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- First problem: **Design goals and specifications**
 - Affected by choice of hardware, type of system (batch, time-sharing, single/multiple users, distributed, real-time, etc)
 - User goals
 - Convenient to use, easy to learn, reliable, safe, and fast
 - System goals
 - Easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
 - **No unique solution to the problem of defining the requirements**

Operating System Design and Implementation

- Important principle to separate
 - **Mechanism**: *How* to do it?
 - **Policy**: *What* will be done?
- Examples
 - Timer mechanism (for CPU protection)
 - Policy decision: How long the timer is to be set?
 - Priority mechanism (in job scheduling)
 - Policy: I/O-intensive programs have higher priority than CPU-intensive ones or vice versa
- Benefits: maximum flexibility
 - Change policy without changing mechanism

OS Implementation

- Much variation
 - Early OSes in assembly language
 - Now C, C++
- Actually usually a mix of languages
 - Main body in C
 - Lowest levels in assembly
 - Systems programs in C, C++, scripting languages
- Pros and cons
 - Code can be written faster, easier to understand/debug
 - More high-level language, easier to **port** to other hardware
 - Slower & increased storage requirement

Implementation

- Performance?
 - Major performance improvements: better data structures and algorithms
 - How about developing excellent assembly-language code in OS implementation?
 - Modern compiler is well optimized
 - A small amount of the code is critical to performance, easy to do specialized optimization
 - Interrupt handler
 - I/O manager
 - Memory manager
 - CPU scheduler

MISC

Debugging, Generation, Booting

Operating-System Debugging

- **Failure analysis**
 - **log files**: written with error information when process fails
 - **core dump**: a capture of the memory of the processes
 - **crash dump**: memory state when OS crashes
- **Performance tuning**
 - *Trace listings* of system behavior
 - **Interactive tools**: top displays resource usage of processes
- 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.”

Operating System Generation

- Operating systems are designed to run on any of **a class of machines**
 - The system must be **configured or generated** for each specific computer site
- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
 - Read from file, ask the operator or probe
 - Generation methods
 - Modify source code and completely recompile
 - Select modules from precompiled library and link together

System Boot

- System booting on most computer systems
 - Bootstrap program (residing in ROM) locates the kernel, loads it into memory, and starts it
 - ROM needs no initialization, cannot be easily infected by virus
 - Diagnostics to determine machine state
 - Initialization: CPU registers, device controllers, memory
 - Some use two-step process: a simple bootstrap loader fetches a more complex bootstrap program, which loads kernel (large OSes)
 - Some store the entire OS in ROM (Mobile OS)
- Common bootstrap loader allows selection of kernel from multiple disks, versions, kernel options (**GRUB**)

Summary

- Operating system services
- System calls
 - Relationship between system call and API
- Operating system structures
 - Modular is important
 - Generally adopt a hybrid approach
- Design principles
 - Separate policy from mechanism

Summary of Part I (Ch1 & Ch2)

- | | |
|---|--|
| <ul style="list-style-type: none">• OS Overview<ul style="list-style-type: none">– OS Functionality– Multiprogramming & Multitasking• OS Operations<ul style="list-style-type: none">– Dual Mode & System Call• OS Components<ul style="list-style-type: none">– Process Management– Memory Management– Storage Management• Computing Environment | <ul style="list-style-type: none">• Ch2 OS Structure<ul style="list-style-type: none">– Operating system services– System calls– Operating system structures– Design principles• Process management<ul style="list-style-type: none">– Concept, scheduling, operation, communication, synchronization• Memory management<ul style="list-style-type: none">– Main memory, virtual mem• Storage management<ul style="list-style-type: none">– Storage, FS, I/O |
|---|--|

End of Chapter 2