Operating System

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 Debugging

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

Very Brief History of OS

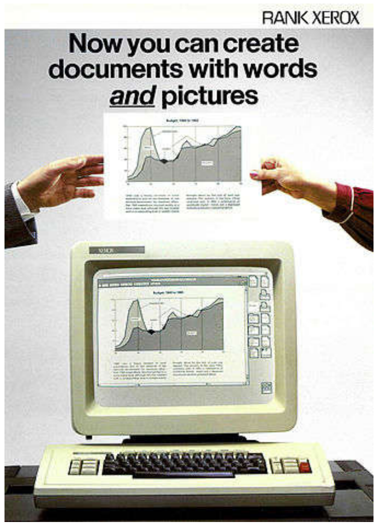
Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics

"I think there is a world market for maybe five computers." – *Thomas Watson, chairman of IBM, 1943*

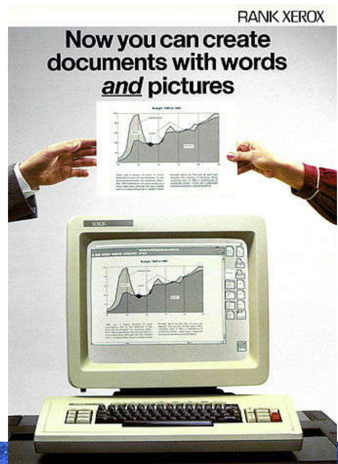
Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - PCs, Workstations, Rise of GUIs



Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    - Ubiquitous devices, widespread networking



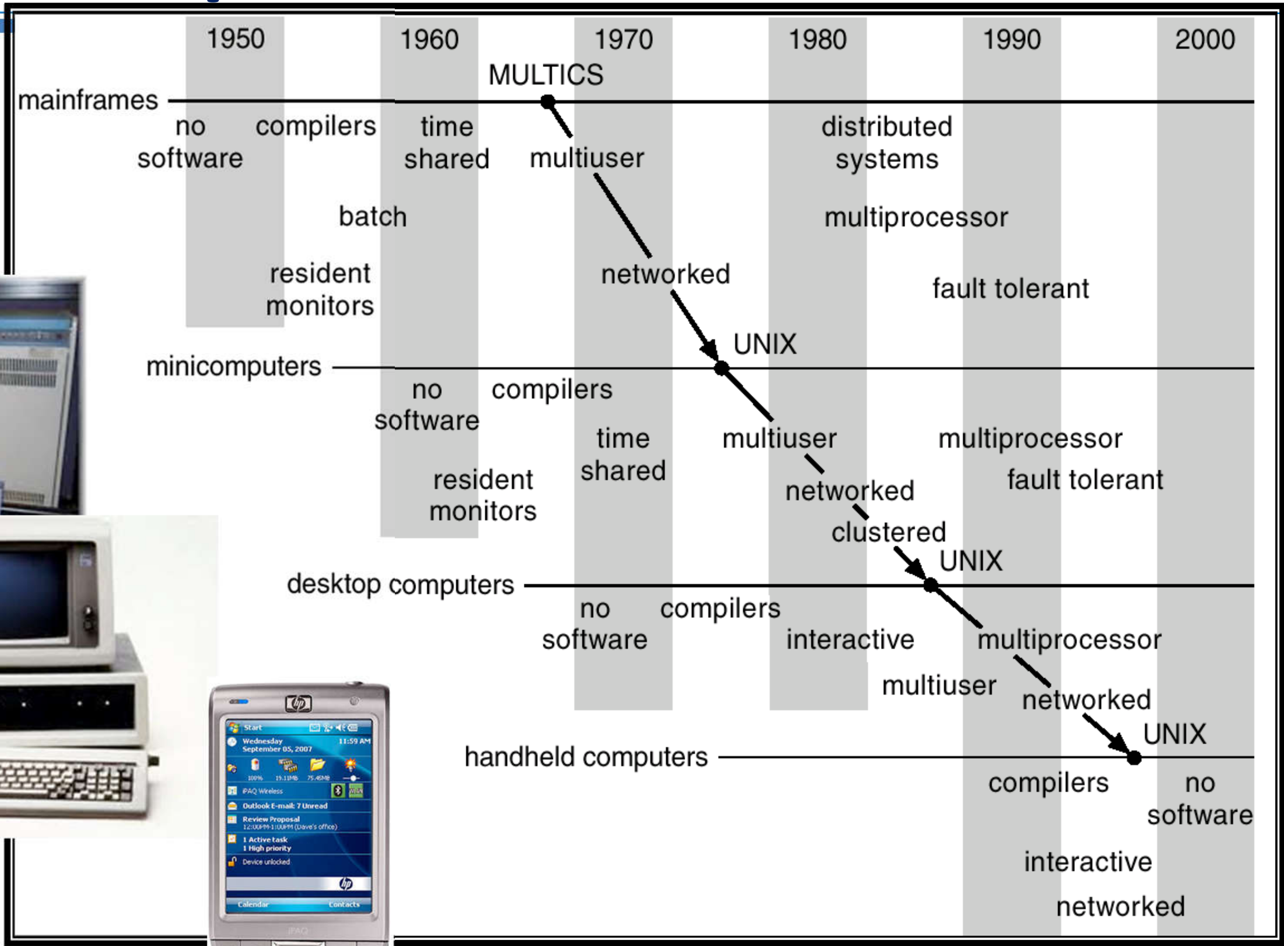
Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    - Ubiquitous devices, Widespread networking
- Rapid change in hardware leads to changing OS
  - Batch $\Rightarrow$ Multiprogramming $\Rightarrow$ Timesharing $\Rightarrow$ Graphical UI $\Rightarrow$ Ubiquitous Devices
  - Gradual migration of features into smaller machines
- Today
  - Small OS: 100K lines / Large: 10M lines (5M browser!)
  - 100-1000 people-years

OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD → NextStep → XNU →
  Apple OS X, iPhone iOS
- MINIX → Linux → Android OS, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,…
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → 10 → …

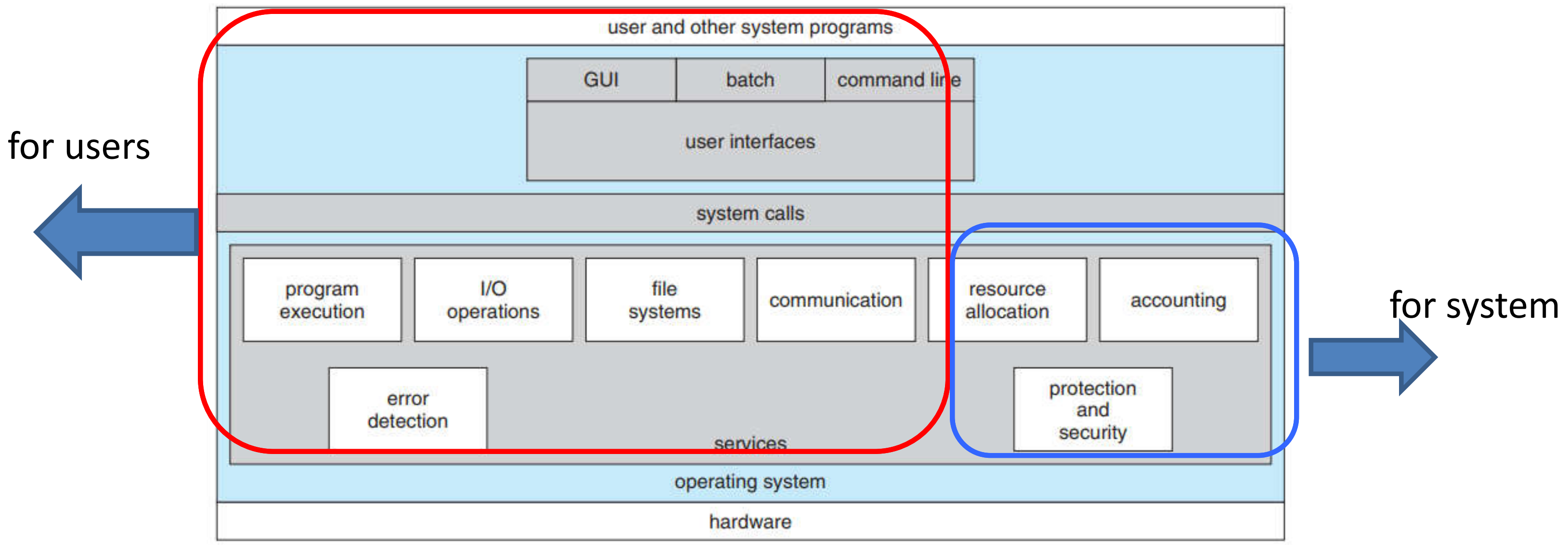
Migration of OS Concepts and Features



Operating-System Service

Advantages of OS

- Different views from OS
  - Interface to users and programs (interactions)
  - Services (predefined comfort routines)
  - Components & their interconnects (SW architecture)



A view of operating system services

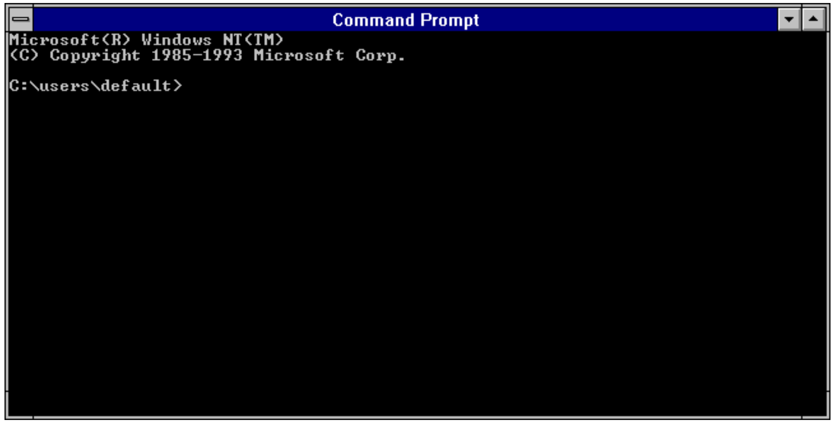
Operating-System Services : User Interface

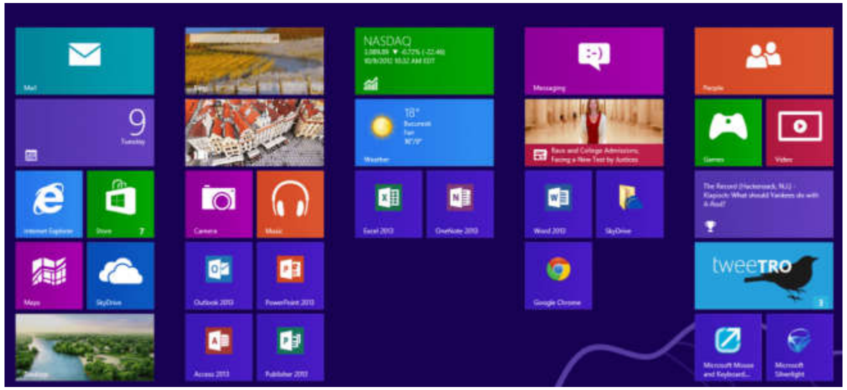
User Interface (UI)

- command-line interface (CLI):

  enter text commands on cmd prompt

  batch interface: commands and directives to control them are in files



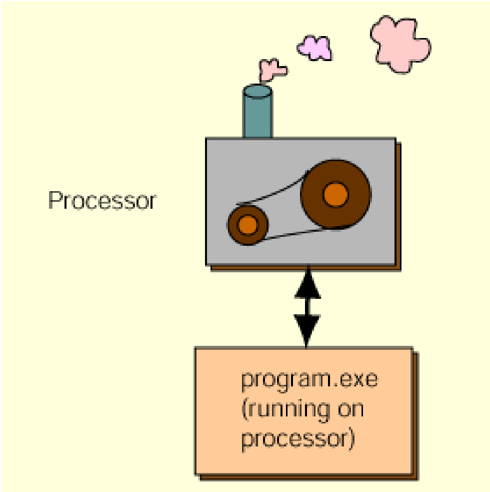


Operating-System Services :Program Execute

Program Execute

- Load a program into memory
- Run that program
- The program must be able to end exec,

  either normally or abnormally (error)



Operating-System Services:I/O Operations

for the user

I/O Operations

- Required by a running program
- Involve a file or an I/O device
- OS handles I/O for efficiency&protection

  users usually cannot directly control I/O devices

Operating-System Services: File-system Manipulation

File-system Manipulation

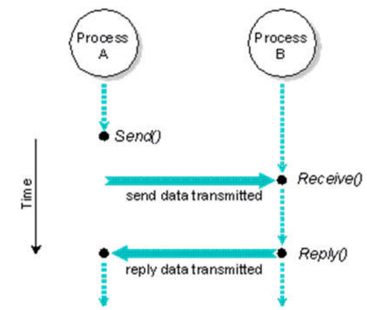
for the user

- Create, delete, read, write file/directory
- Search or list file information
- Allow or deny access to files/directories based on file ownership

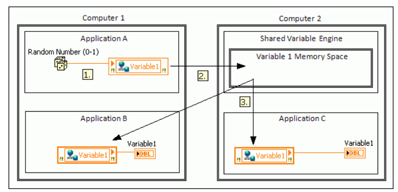
Operating-System Services : Communications

for the user

Communications



- Exchange info between processes via
  - shared memory
    - ✓two or more processes read and write to a shared section of memory
  - message passing
    - ✓packets of info in predefined formats are moved between processes by OS



Operating-System Services : Error Detection

Error Detection **for the user**

- Errors: memory error, power failure, parity error on disk, arithmetic overflow, access to illegal mem location, etc.
- OS needs to act to ensure correct and consistent computing
- Or halt the system
- Or terminate an error-causing process
- Or return error code to a process

Operating-System Services: resource allocation

Resource Allocation

for the system

- Upon multiple users or jobs running at the same time, resource allocation using
  - Special allocation code:

    CPU cycles, main mem, file storage
  - or General request and release code:

    I/O devices

Operating-System Services: accounting

Accounting **for the system**

- Keep track of which users use how much and what kind of resources
- Used for accounting or accumulating usage stats
- Usage stats are useful to reconfigure system to improve computing services

Operating-System Services: Protection and Security

Protection and Security

for the system

- Upon concurrent processes, it is not possible for one process to interfere with the others or with the OS
- Protection: ensure that all access to system resources is controlled
- Security: defend against outsiders

  user authentication,

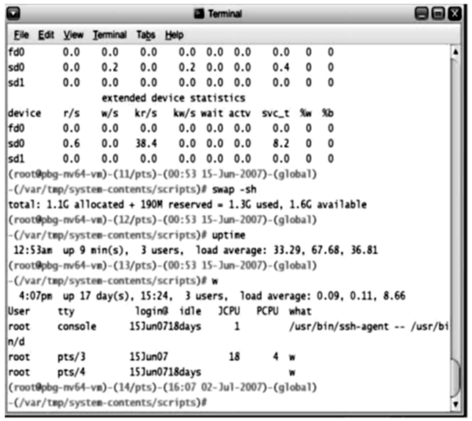
  access-validity check, etc.

User and Operating-System Interface

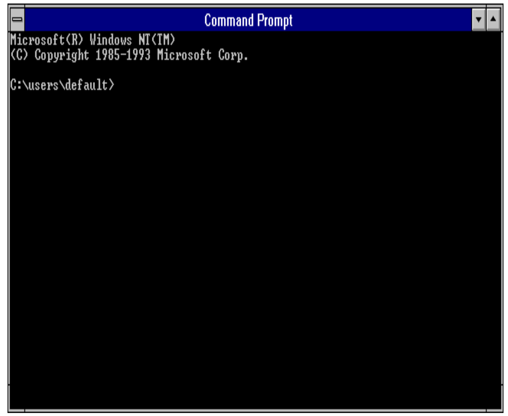
OS interface (command line)

- Command interpreters
  - Kernel-based vs. system program (Win, UNIX)
- Multiple interpreters, *shell*
  - UNIX, Linux shells: *Bourne shell*, *C shell*, *Bourne-Again shell*, *Korn shell*
  - Third party shell!
  - Similar functionality, user preference
- Shell implementations
  - Internal codes (make jump)
  - System programs (UNIX)
    - rm file.txt

pros and cons ?



Bourne shell in Solaris 10



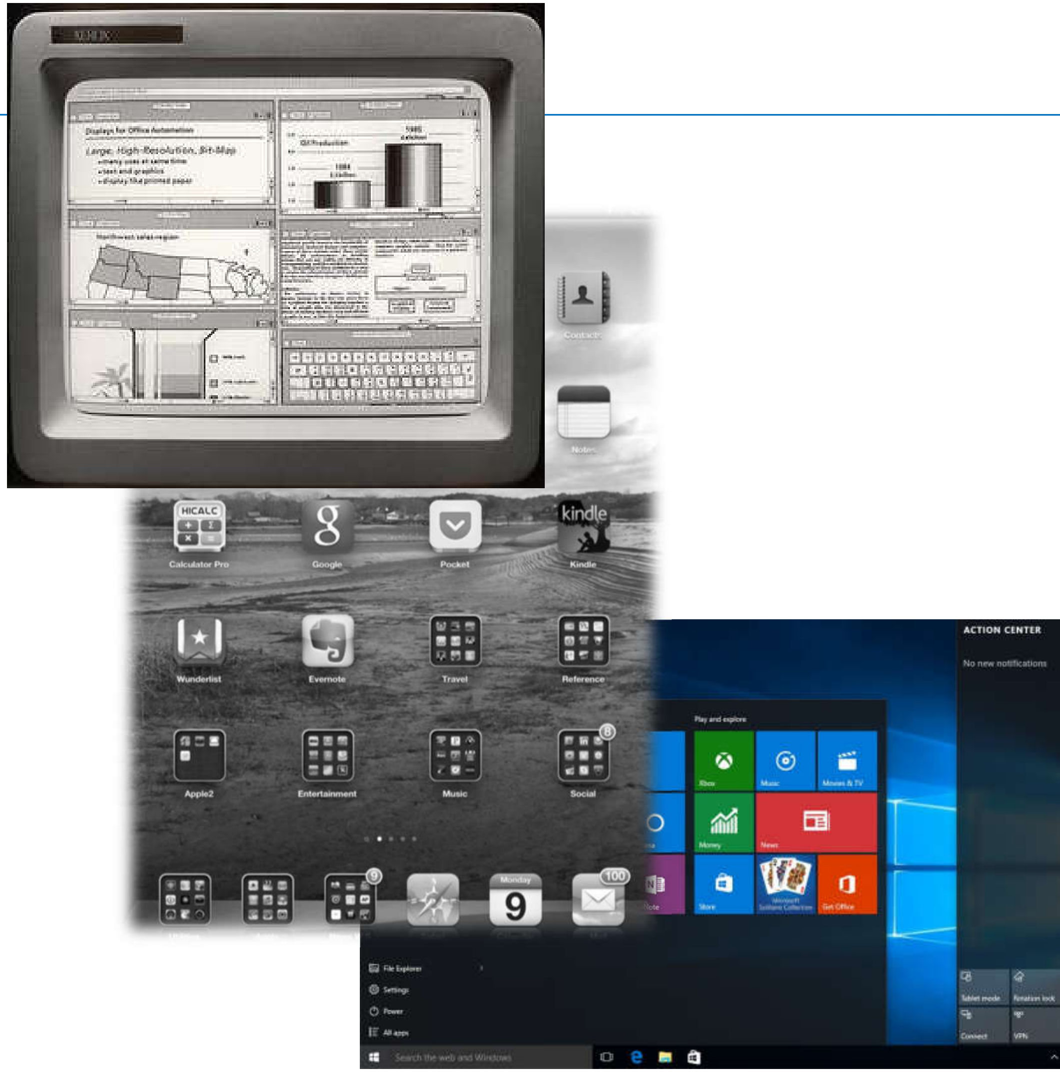
shell in windows

Bourne Shell Command Interpreter

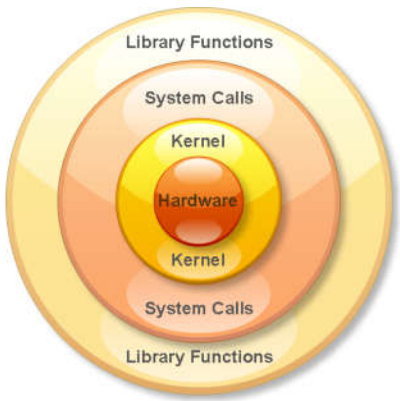
```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           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$
```

OS interface (GUI)

- Xerox PARC (1970)
- Xerox Alto (1973)
- Apple Macintosh (1980)
- Aqua in Mac OS X
- Microsoft Windows
- UNIX
  - CDE (Common desktop environment)
  - X-Windows
  - KDE (K Desktop environment)
  - GNOME (GNU project)



System Calls



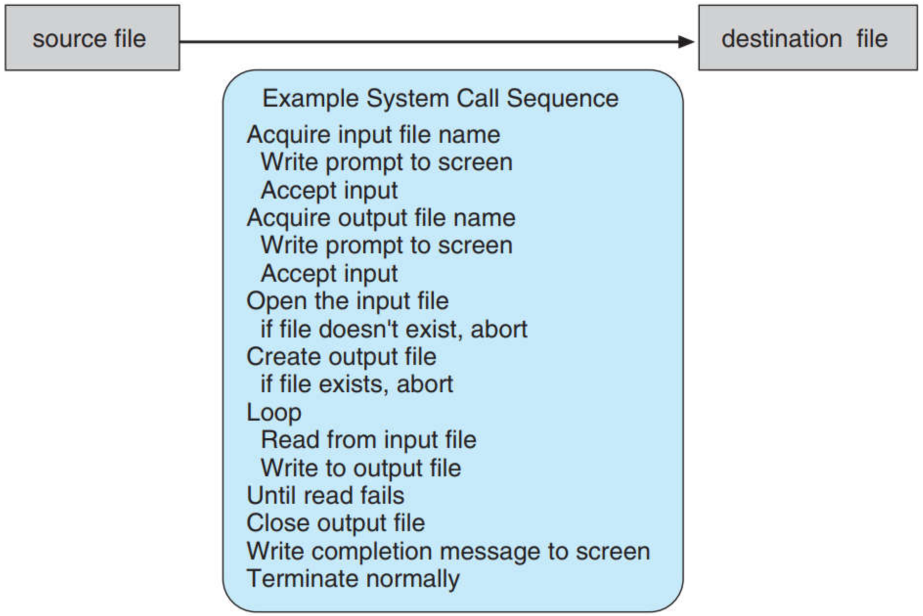
System calls

Programming interface to the services provided by the OS

- C, C++, Assembly
- API (Application program interface)
  - Win API
  - POSIX API (UNIX, Linux, Mac OS X): *libc*
  - JAVA API

Example of Standard API

- Example: Copy file



Why use APIs rather than 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)
```

$\underbrace{\texttt{ssize\_t}}_{\text{return value}}$ $\underbrace{\texttt{read}}_{\text{function name}}$ $\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$

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.

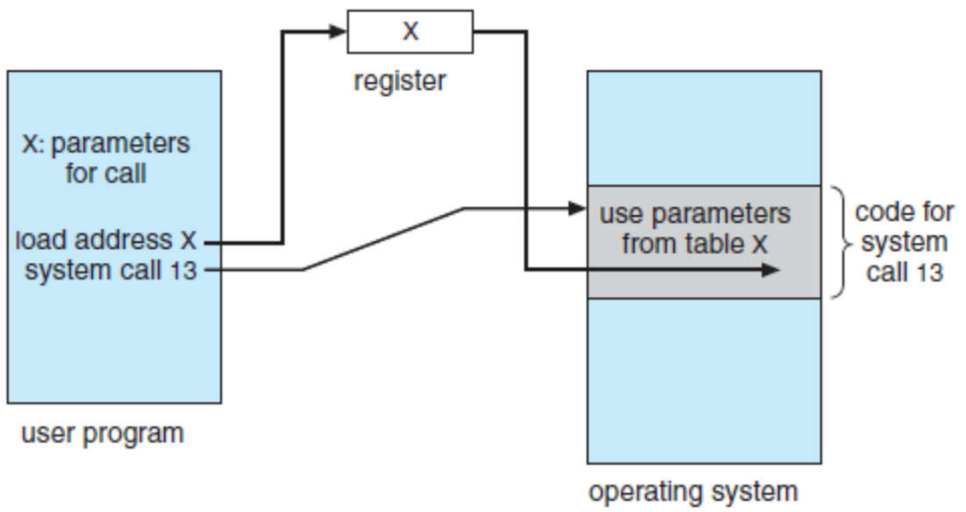
System Call Implementation

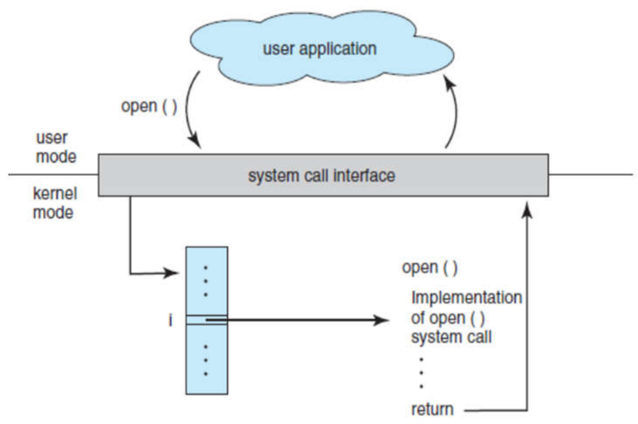
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
  - old way: int 0x80 and iret
  - new way: sysenter and sysexit

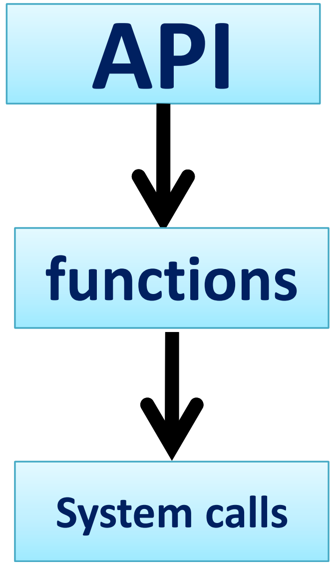
```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
```

System call interface

- Parameter passing
  - Register
  - Register pointer to mem. block
  - Stack (PUSH, POP)







where to pass parameters?

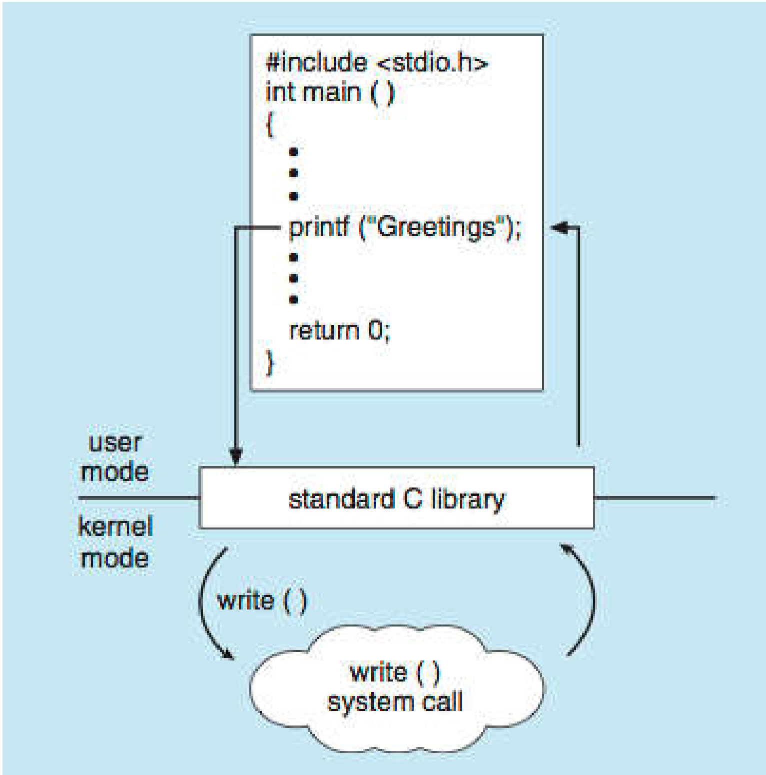
Types of system calls

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications
- Protections

	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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

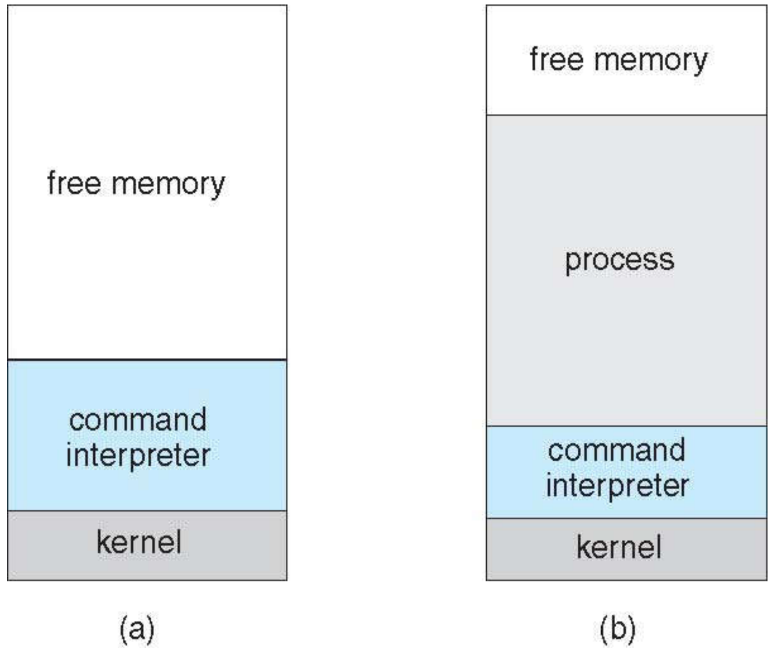
Standard C Library Example

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



Example: 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

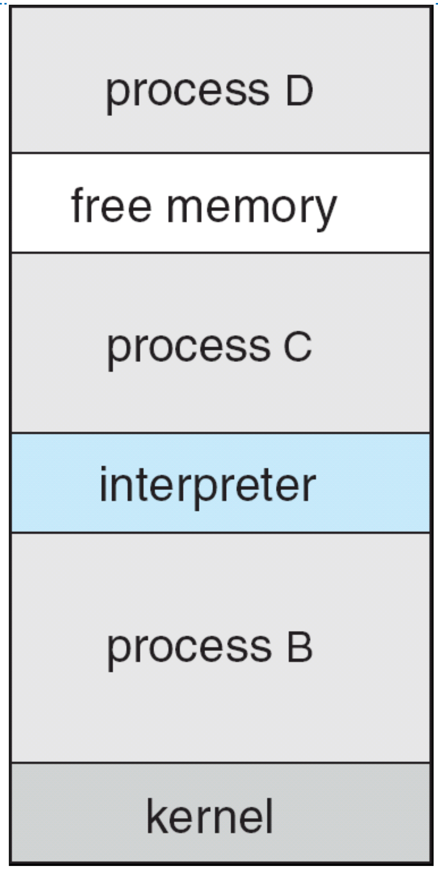


(a) At system startup (b) running a program

MS-DOS execution

Example: 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 of 0 – no error or > 0 – error code



FreeBSD running multiple programs

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

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

Operating-System Design and Implementation

OS design & implementation

- Goals
  - User vs. System
- Mechanisms & policy
  - how vs. what
- Implementation
  - Assembly, C or C++
    - MCP: ALGOL
    - MULTICS: PL/1
    - Linux, Win: C, Assembly
  - C is supported on diff. ISAs, CPUs

Implementation

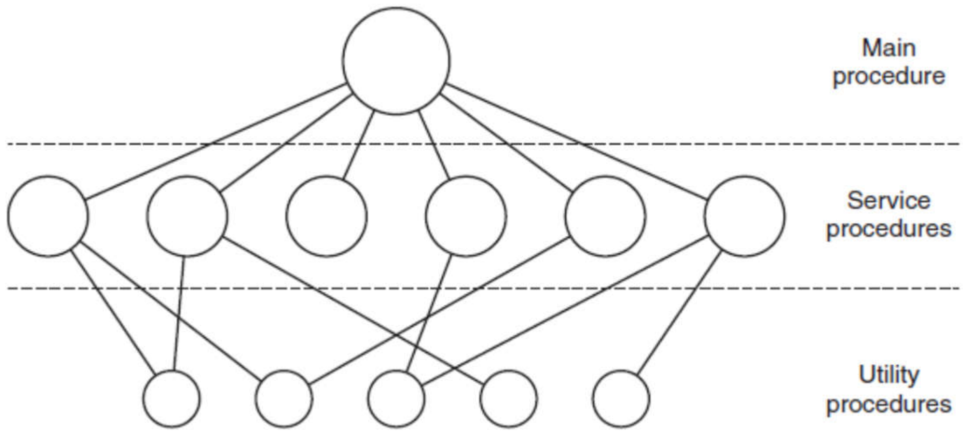
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More 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

OS structure: 1. Simple structure (Monolithic)

- The most common organization
- OS is a single large program in kernel mode

- Problems
  - Crash in called procedures?
  - Unwieldy & difficult to understand

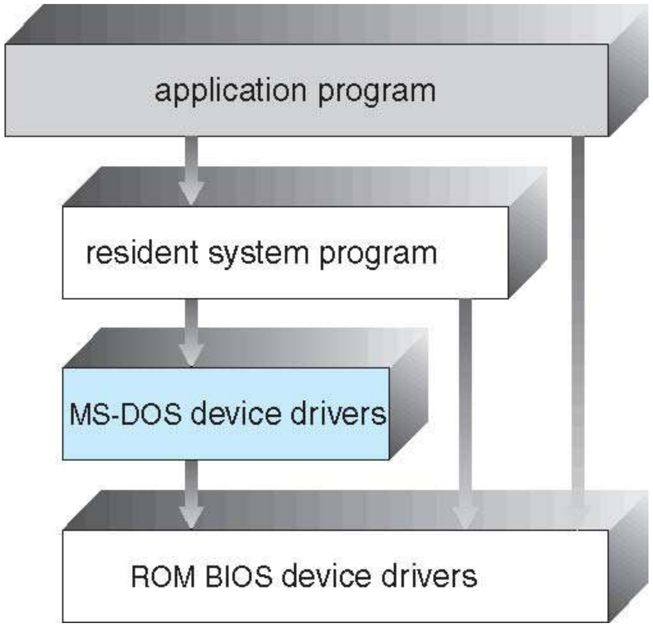


A simple structuring model for a monolithic system

Samples： MS-DOS

- Simple structure
  - MS-DOS, UNIX (Traditional)

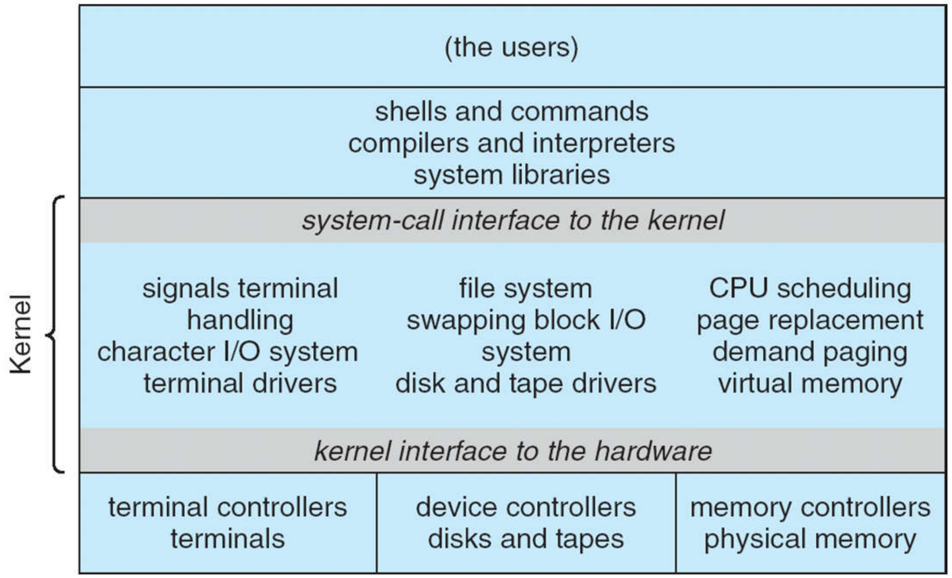
- Not divided into modules
- Interfaces and levels of functionality are not well separated
- App programs can directly write to display and disk drivers, leaving MS-DOS vulnerable to errant/malicious programs



MS-DOS layer structure

Samples： UNIX

- Not divided into modules
- Kernel + system programs
- Monolithic structure:
  - ✓enormous amount of functionality to be combined into one kernel;
  - ✓difficult to implement and maintain

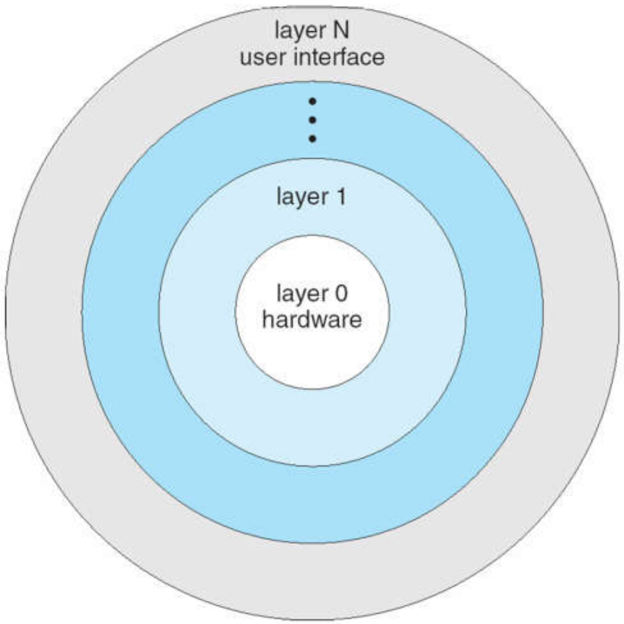


Traditional UNIX system structure

Beyond simple but not fully layered

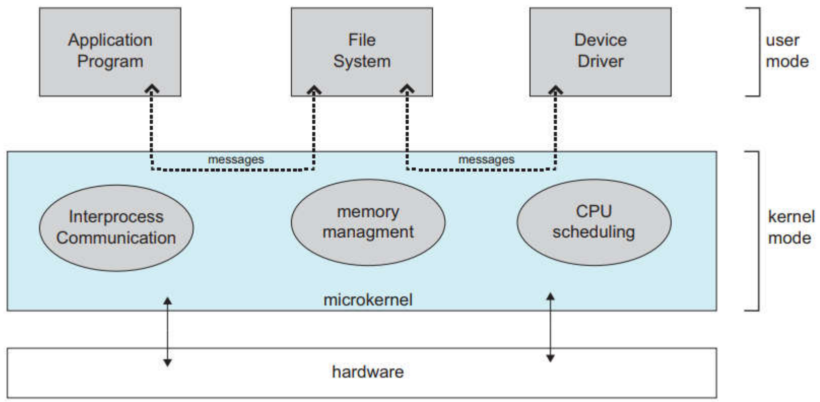
OS structure: 2. Layered approach

- Layered approach
  - abstractions
  - adv.
    - Simplicity
      - Construction
      - Debugging
    - Functions and operations of low layers
  - dis. adv.
    - Layer definition problem
      - MMU, backing store, scheduler (?)
    - Less efficient



OS structure: 3. Microkernel

- Microkernel
  – Moves as much from the kernel into "*user*" space
  – Communication takes place between user modules using message passing
- Examples
  – Mach (CMU)
  – Mac OS X kernel (Darwin)
- 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

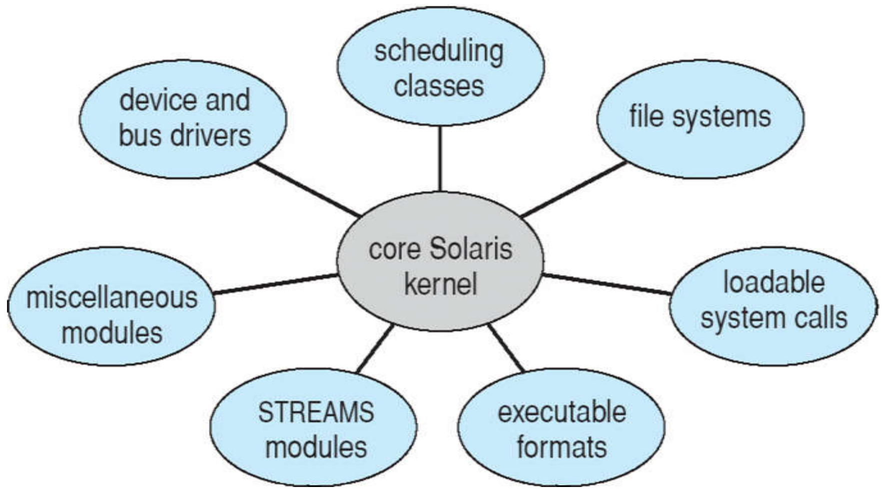


Architecture of a typical microkernel

- Detriments:
  – Performance overhead of user space to kernel space communication
  – Windows NT 4 (microkernel) slow!
    – vs. Windows XP (monolithic) fast!

OS structure: 4. Modules

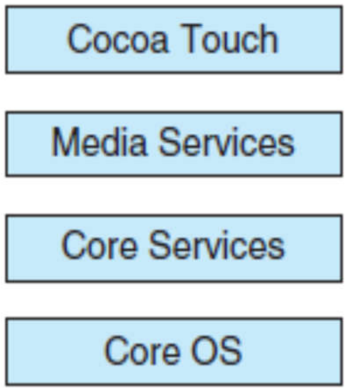
- Modules
  - Most modern operating systems implement kernel modules
    - Uses object-oriented approach
    - Each core component is separate
    - Each talks to the others over known interfaces
    - Each is loadable as needed within the kernel
    - Faster than microkernel
      - No need of message passing
    - Better than layered
      - Direct module communications
  - Overall, similar to layers but with more flexible
    - Linux, Solaris, etc



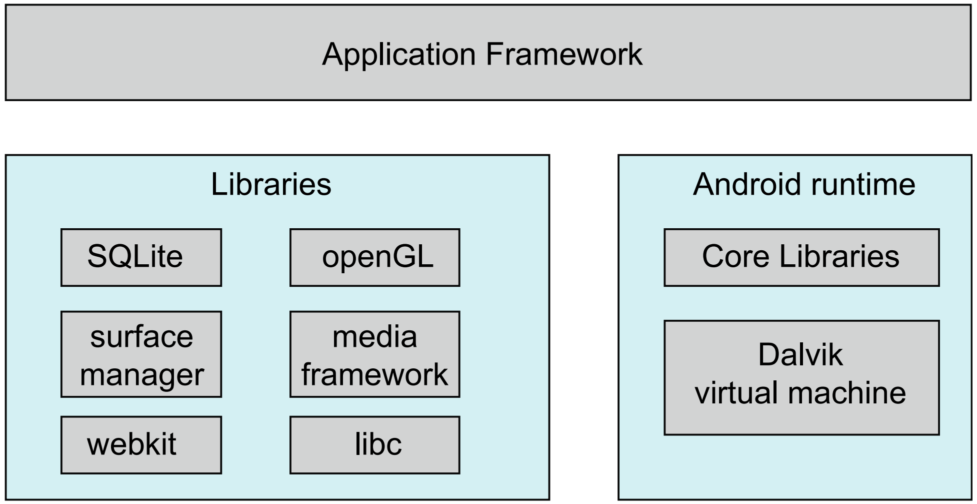
Solaris loadable modules

OS structure: 5. Hybrid

- Hybrid systems
  - Most modern operating systems
    - Mac OS
    - iOS
    - Android
  - Better to address
    - Reliability
    - Security
    - Usability
  - Examples
    - Linux & Solaris
      - kernel: monolithic
      - +feature: loadable
    - Windows
      - monolithic+microkernel



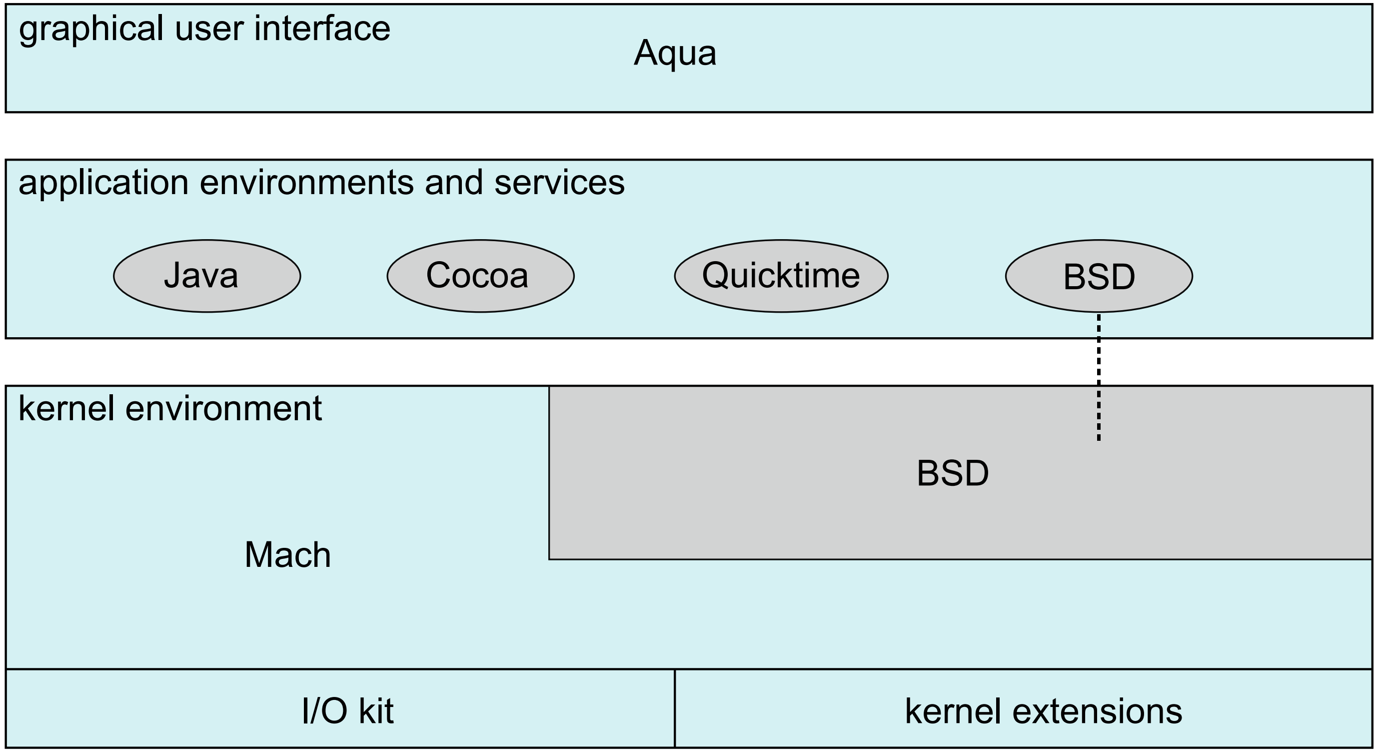
Architecture of Apple's iOS



Android

Mac OS X Structure

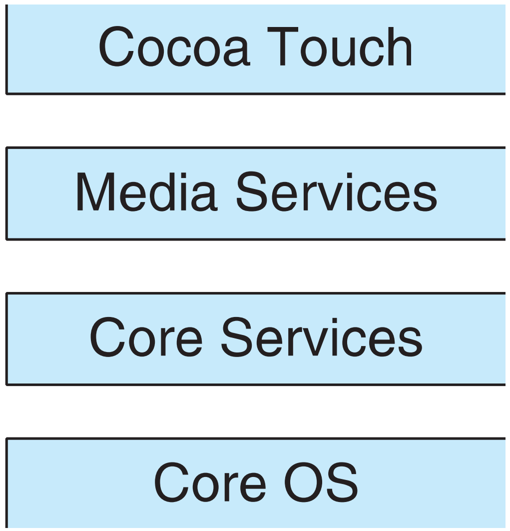
a hybrid structure



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

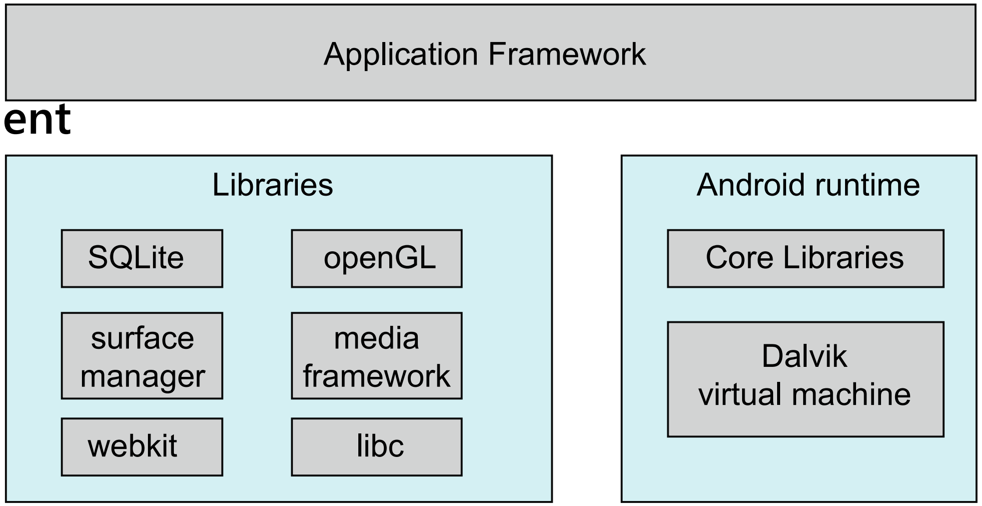
iOS

- 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

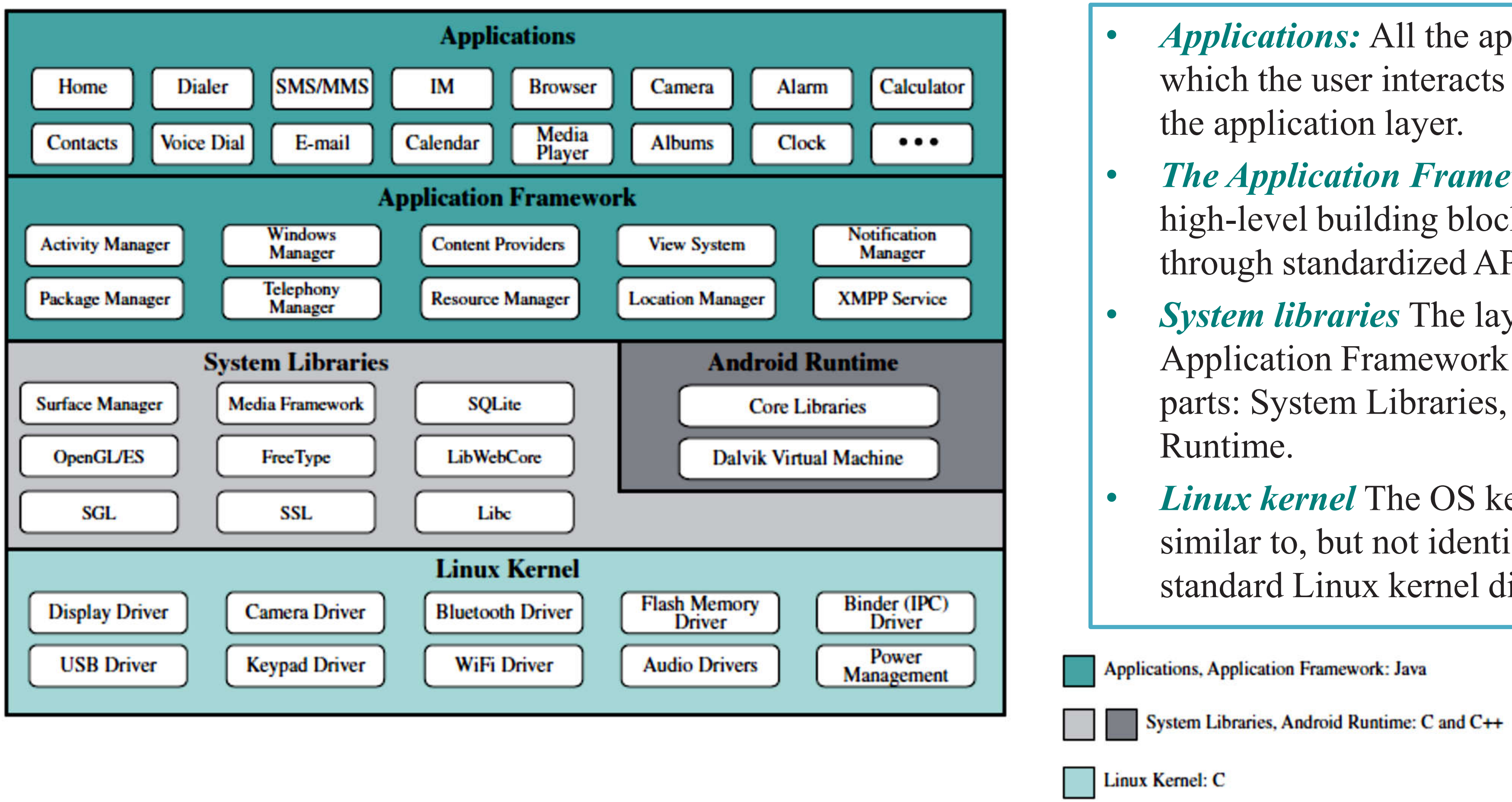


Android

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

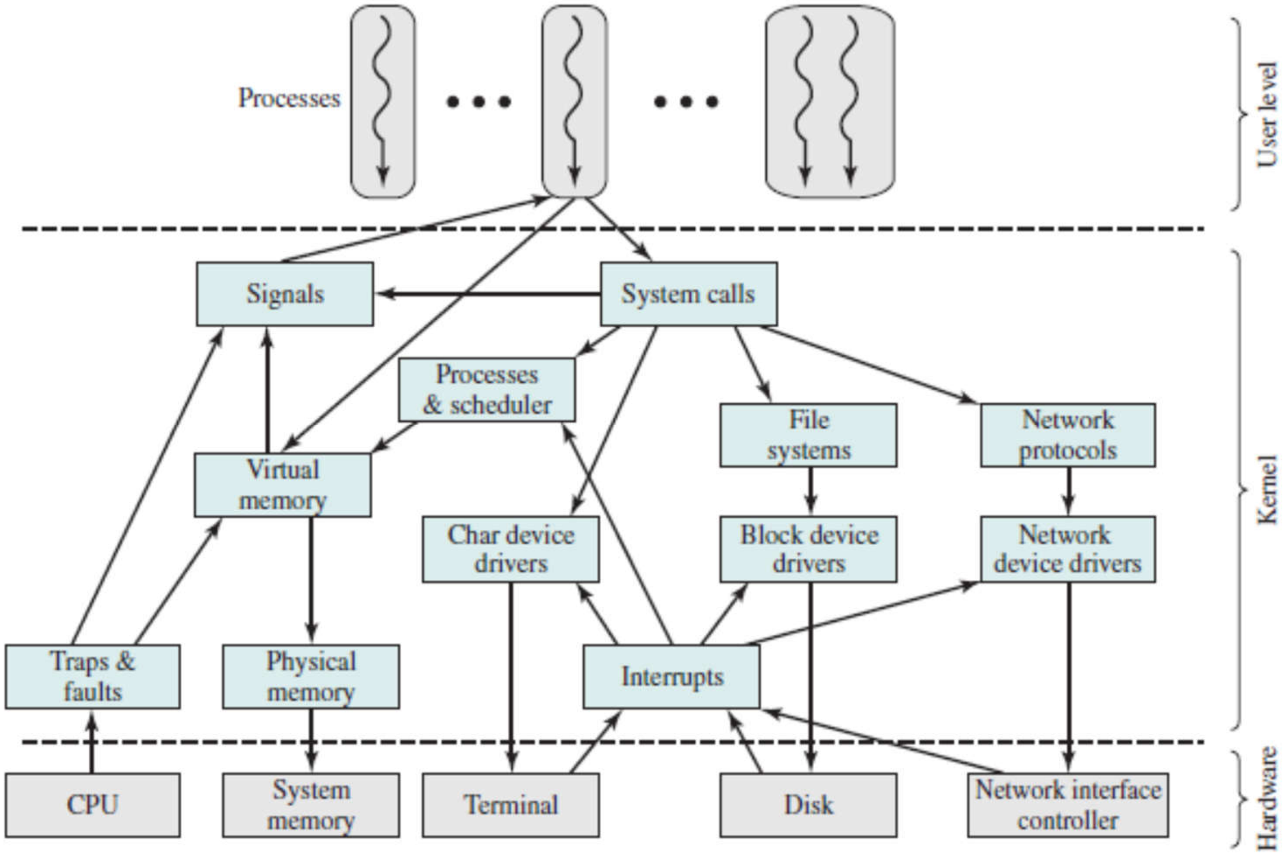


Android SW architecture (detail)



- *Applications:* All the applications with which the user interacts directly are part of the application layer.
- *The Application Framework* layer provides high-level building blocks, accessible through standardized APIs
- *System libraries* The layer below the Application Framework consists of two parts: System Libraries, and Android Runtime.
- *Linux kernel* The OS kernel for Android is similar to, but not identical with, the standard Linux kernel distribution.

Linux kernel components



- **Signals:** The kernel uses signals to call into a process. For example, signals are used to notify a process of certain faults, such as division by zero.
- **System calls:** The system call is the means by which a process requests a specific kernel service.
- **Processes and scheduler:** Creates, manages, and schedules processes.
- **Virtual memory**: Allocates and manages virtual memory for processes.

Operating System Debugging

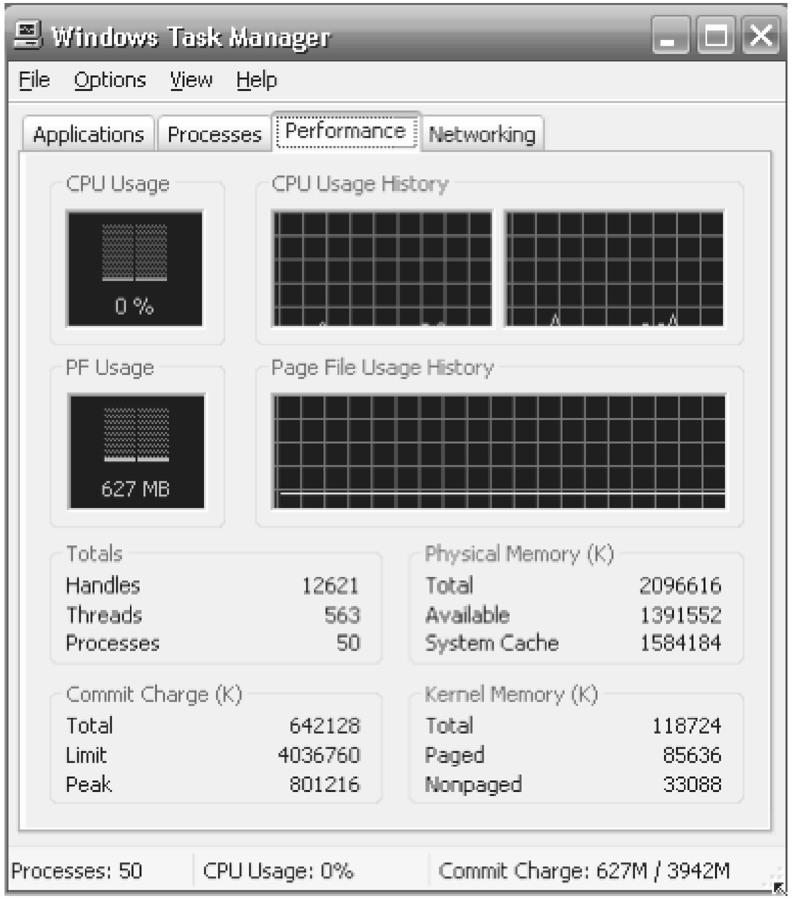
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OSes 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



DTrace

DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems

Probes fire when code is executed within a provider, capturing state data and sending it to consumers of those probes

Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                              U
  0       -> ioctl                            K
  0         -> getf                           K
  0           -> set_active_fd                K
  0           <- set_active_fd                K
  0         <- getf                           K
  0         -> get_udatamodel                 K
  0         <- get_udatamodel                 K
...
  0         -> releasef                       K
  0           -> clear_active_fd              K
  0           <- clear_active_fd              K
  0           -> cv_broadcast                 K
  0           <- cv_broadcast                 K
  0         <- releasef                       K
  0       <- ioctl                            K
  0     <- ioctl                              U
  0   <- _XEventsQueued                       U
  0 <- XEventsQueued                          U
```

End of Chapter 2