Session Plan

* Operating System Services
* User Operating System Interface
* System Calls
* Types of System Calls
* System Programs
* Operating System Structure

Chapter Objectives

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

**Operating System Services**

* Operating systems provide an environment for execution of programs and services to programs and users 🡨 most important feature o.s gives us is run our program

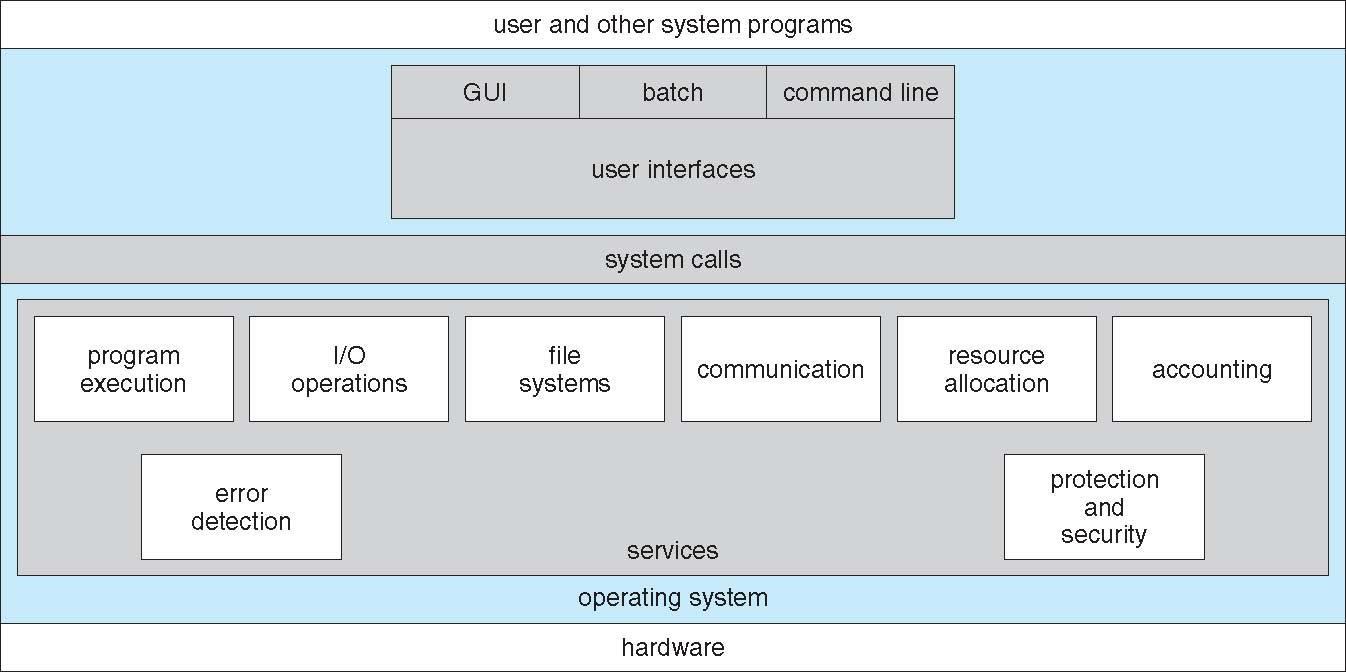
It is loaded to memory and then compiles,

program cant run forever

* 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**)**, Graphics User Interface **(**GUI**)**, Batch 🡨gives us a way to interact with the hardware, provide us universal interface, we don’t care how they work. Powerful use Command-Line
  + **Program execution** - The system must be able to load a program into memoryand to run that program, end execution, either normally or abnormally (indicating error) 🡨Programs are finite, terminate after the last instruction
  + **I/O operations** - A running program may require I/O, which may involve a fileor an I/O device 🡨handles in the o.s operation
* 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 toread and write files and directories, create and delete them, search them, list file Information, permission management. 🡨we can browse file, delete, update; useful when running multiple programs and we need them to communicate with each other
  + **Communications** –Processes may exchange information, on the same computer orbetween 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
* 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 runningconcurrently, resources must be allocated to each of them 🡨in charge of allocating hardware resources and do it effectively
    - 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 ofcomputer resources 🡨 keep track of the what happens in the o.s. Maybe learn the user behavior and learn what the user likes.
  + **Protection and security -** The owners of information stored in a multiuser ornetworked computer system may want to control use of that information, concurrent processes should not interfere with each other🡨just share data with trusted parties
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defendingexternal I/O devices from invalid access attempts

🡨 systems calls talks to program execution, I/O operations,…

**A View of Operating System Services**



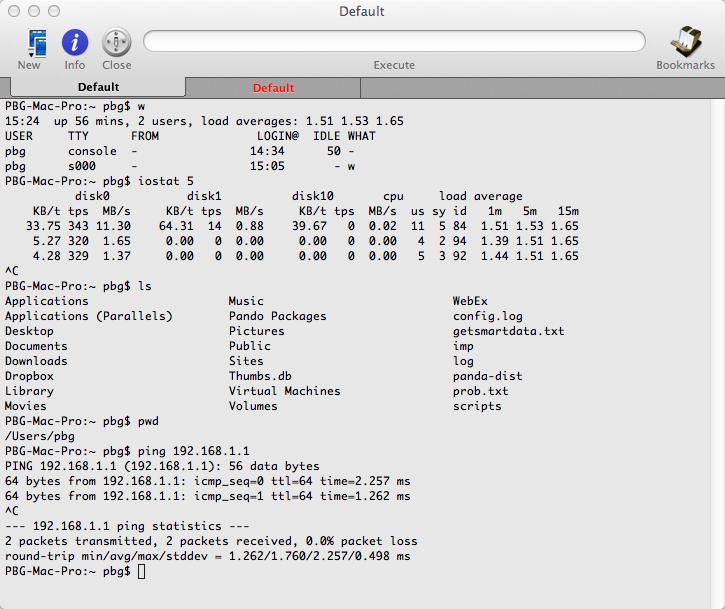
🡨we cannot invoke O.S services directly we use system call and execute commands on our behalf

**User Operating System Interface - CLI**

CLI or command interpreter 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
  + If the latter, adding new features doesn’t require shell modification

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 is “Aqua” GUI interface with UNIX kernel underneath and shells available
  + Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

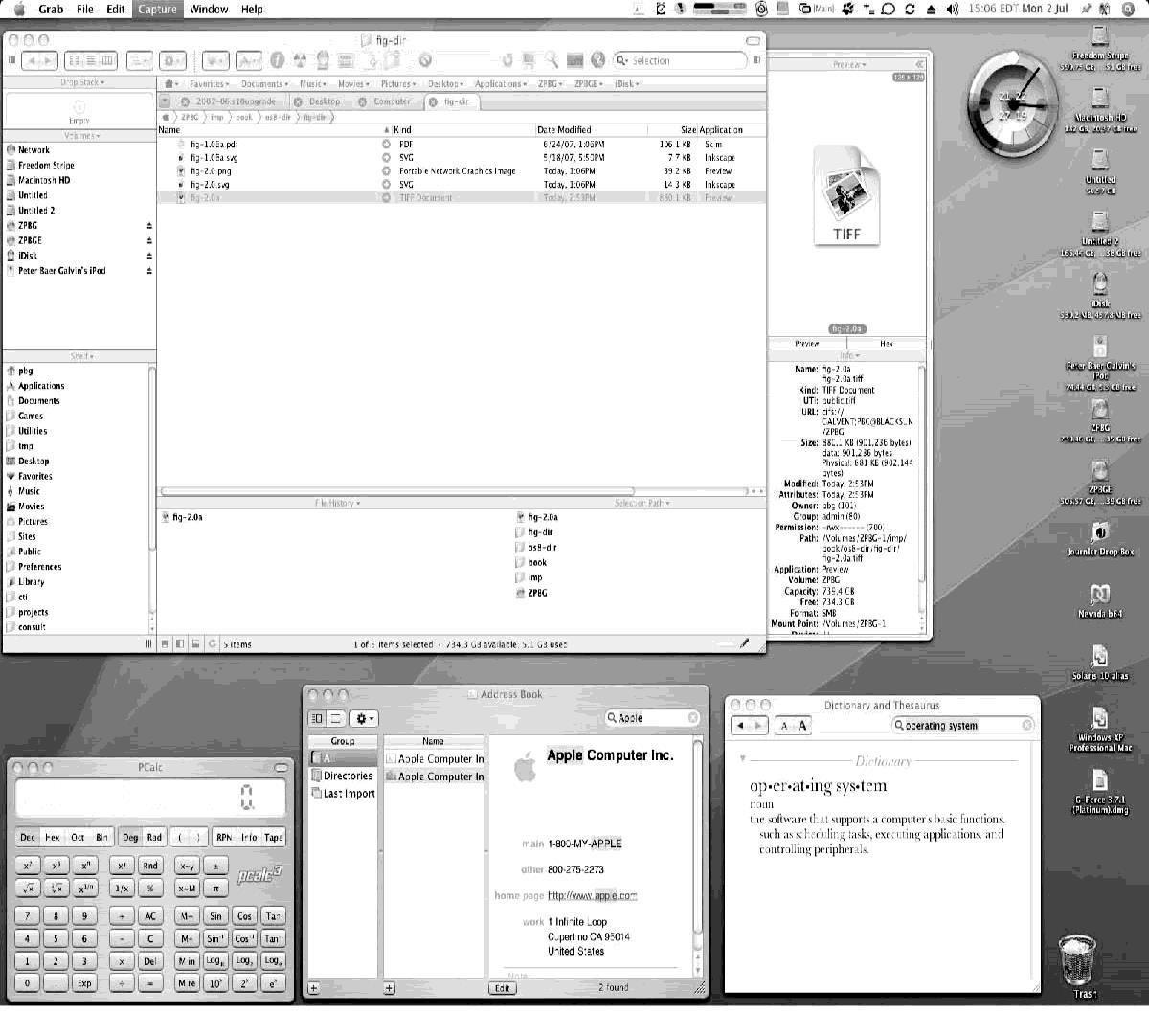


Touchscreen Interfaces



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

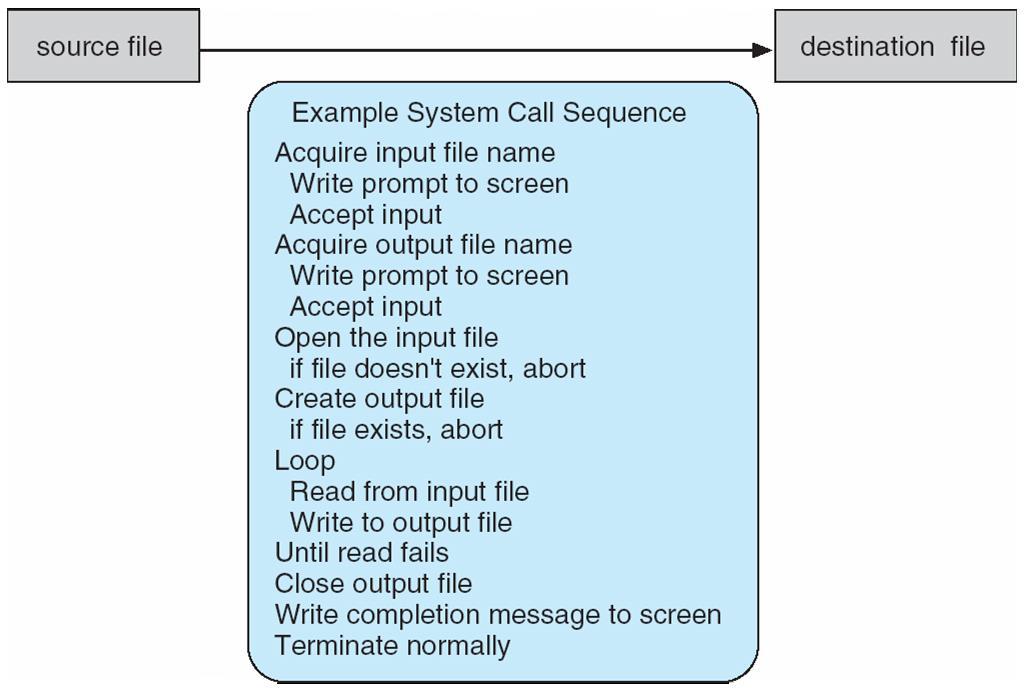
The Mac OS X GUI



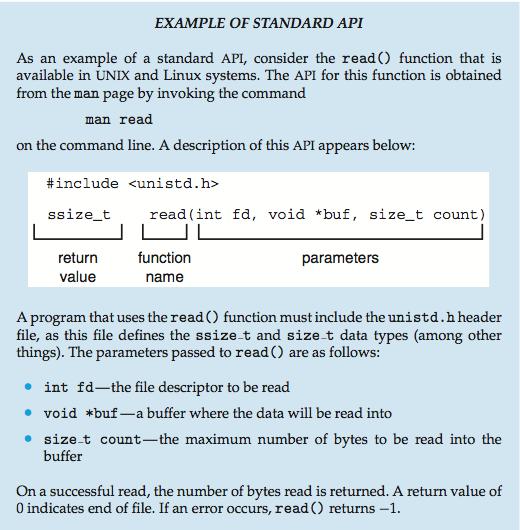
**System Calls**

* 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
* 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 API rather than system calls? 🡨portability, reusability of code, safer(vulnerabilities),can download java api and run code in any machine

Example of System Calls

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

Example of Standard API

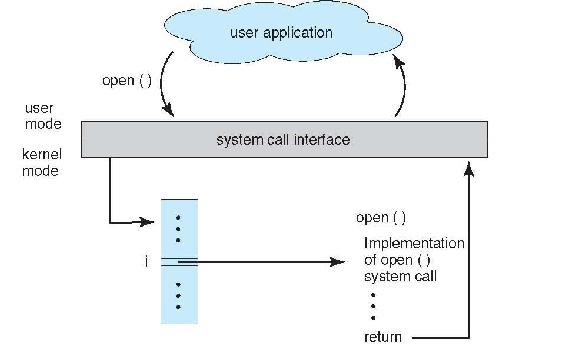


🡨type of paramenters,return value, and paramenters is something we care about in order to call system calls

System Call Implementation

* Typically, a number associated with each system call
  + System-call interface maintains a table indexed according to these numbers
* The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
* 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)

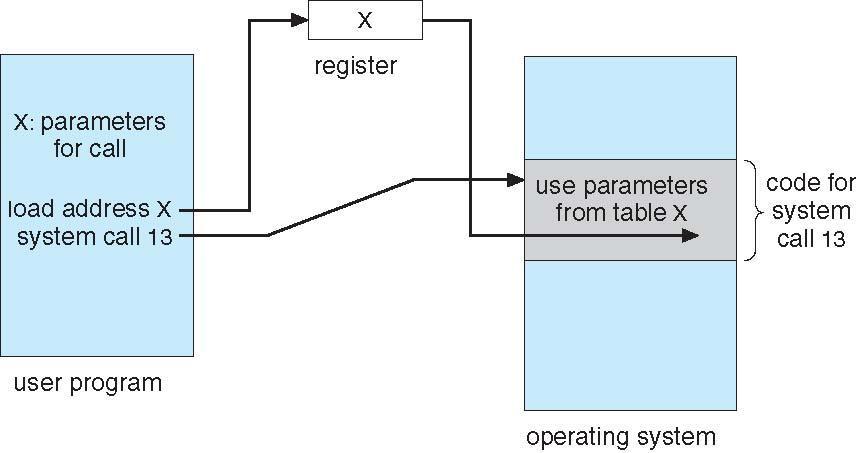
API – System Call – OS Relationship



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🡨limited register and limited capacity in register(capacity), sometimes we have more parameters going to register therefore we create a block or table with its own address. Now at memory we know the address of the register. Or we can push them to stack and pop as we need parameters
  + 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

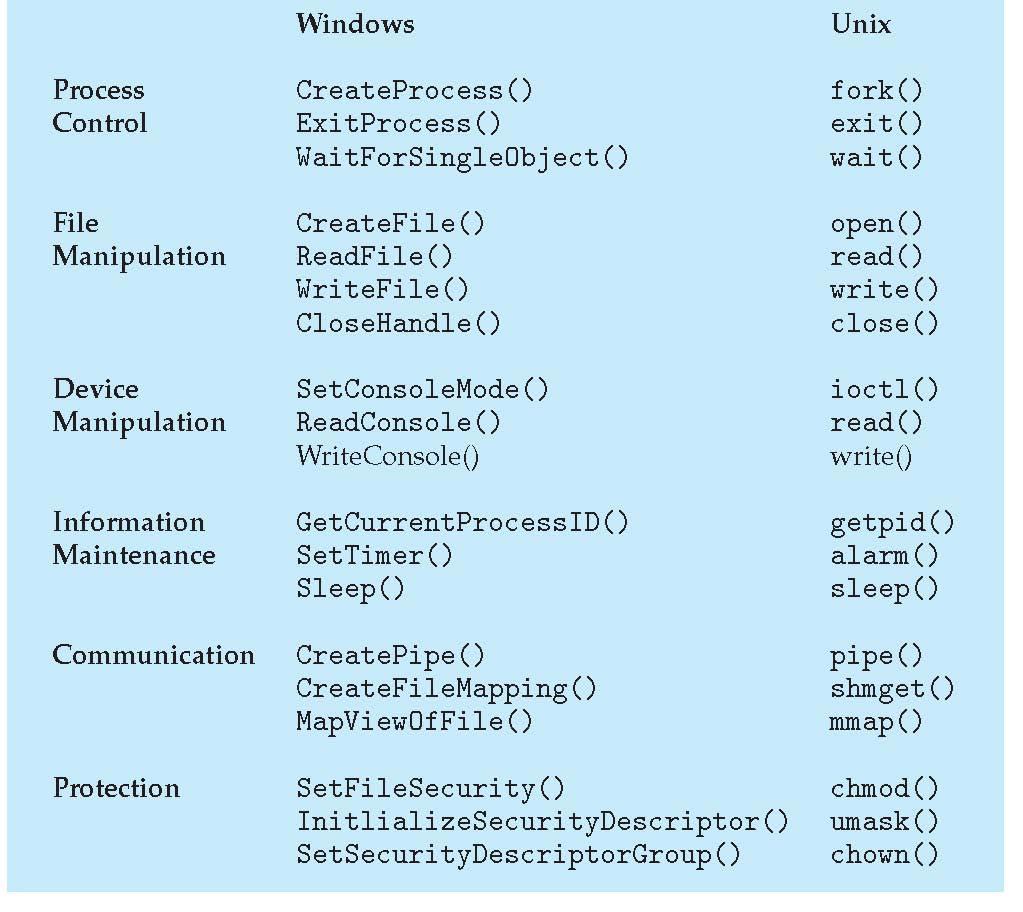
Parameter Passing via Table🡨pass it into register, We look in the table and O.S we execute the code



**Types of System Calls 🡨skipped it in class**

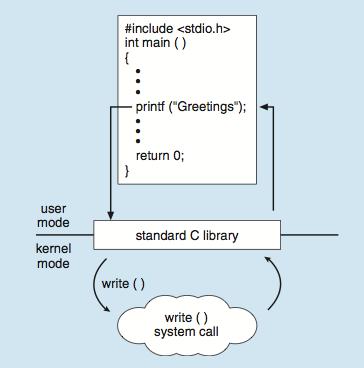
* Process control
  + create process, terminate process
  + end, abort
  + load, execute
  + get process attributes, set process attributes
  + wait for time
  + wait event, signal event
  + allocate and free memory
  + Dump memory if error
  + Debugger for determining bugs, single step execution
  +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

Examples of Windows and Unix System Calls

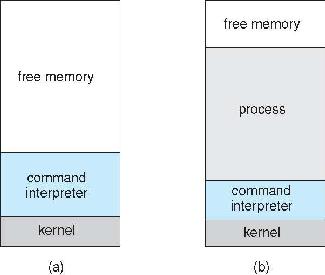


Standard C Library Example

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



Example: MS-DOS



At system startup running a program

• 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

Example: Arduino

• Single-tasking

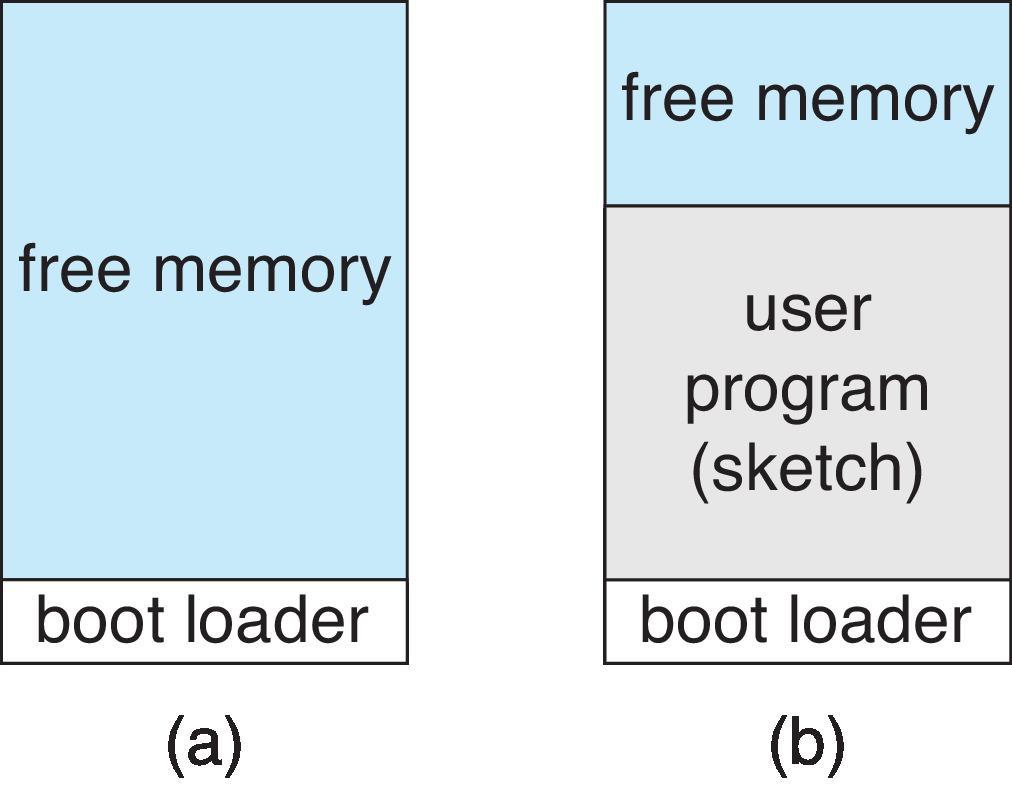
• No operating system

* Programs (sketch) loaded via USB into flash memory

• Single memory space

• Boot loader loads program

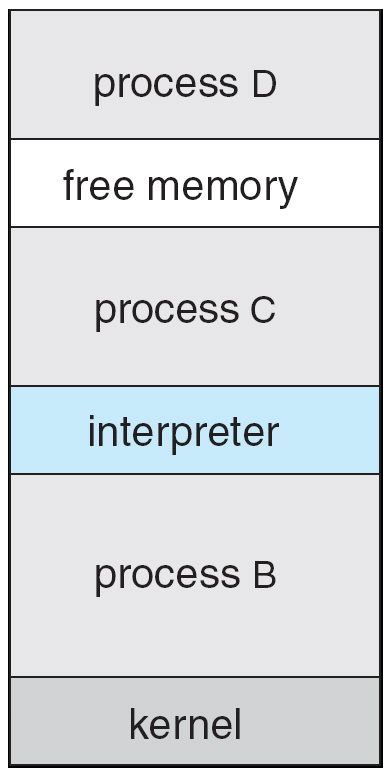
• Program exit -> shell reloded



At system startup running a program

Example: FreeBSD

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



System Services

* System 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
  + 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 anddirectories
* **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
* **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 sometimesprovided
* **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
* **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

**Operating System Structure**

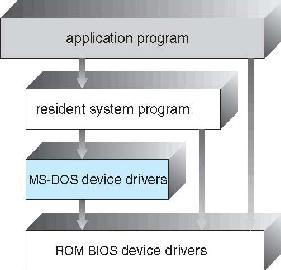
* General-purpose OS is very large program
* Various ways to structure ones
  + Simple structure – MS-DOS
  + More complex -- UNIX
  + Layered – an abstraction
  + Microkernel – Mach

Simple Structure – MS-DOS

•MS-DOS – written to provide the most functionality in the least space

• Not divided into modules(components)

* Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



|  |
| --- |
|  |

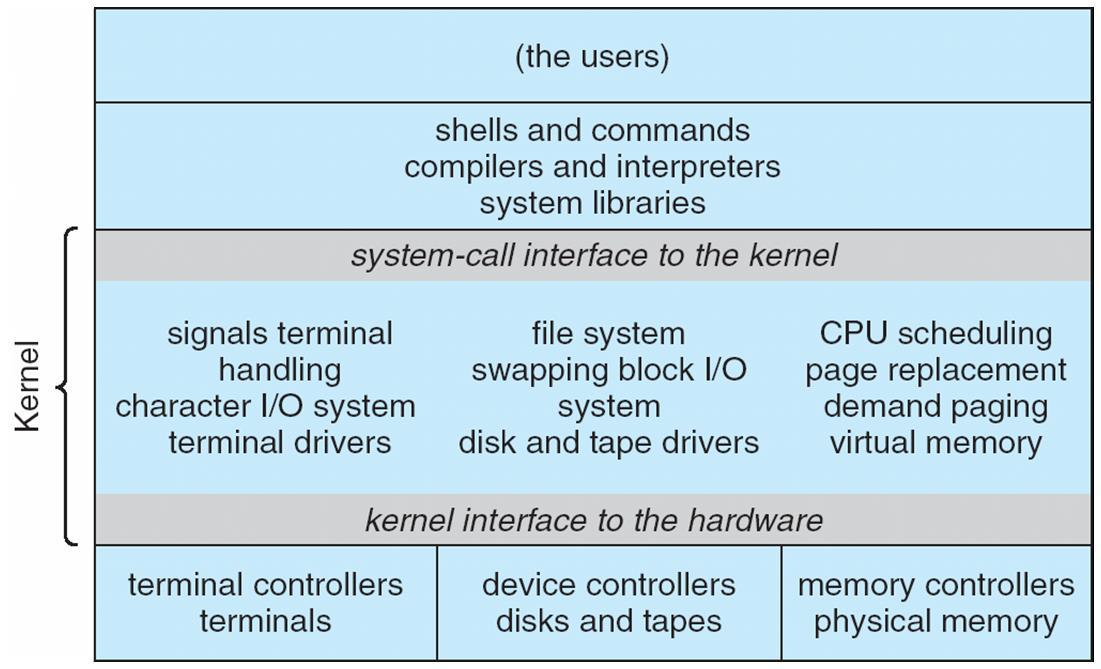
🡨see application program can access ROM BIOS device driver. This is faulty and cann allow access to memory and also security problems

Non Simple Structure -- 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



🡨Kernal:very hard to fix if bugs

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🡨Suppose we have 3 layer, this is very difficult to implement

LayerN+1: can invoke lower levels, invoke data(functions)

Layer N: data operations(functions), invokes lower layer

Layer N-1:invokes data functions

.

.

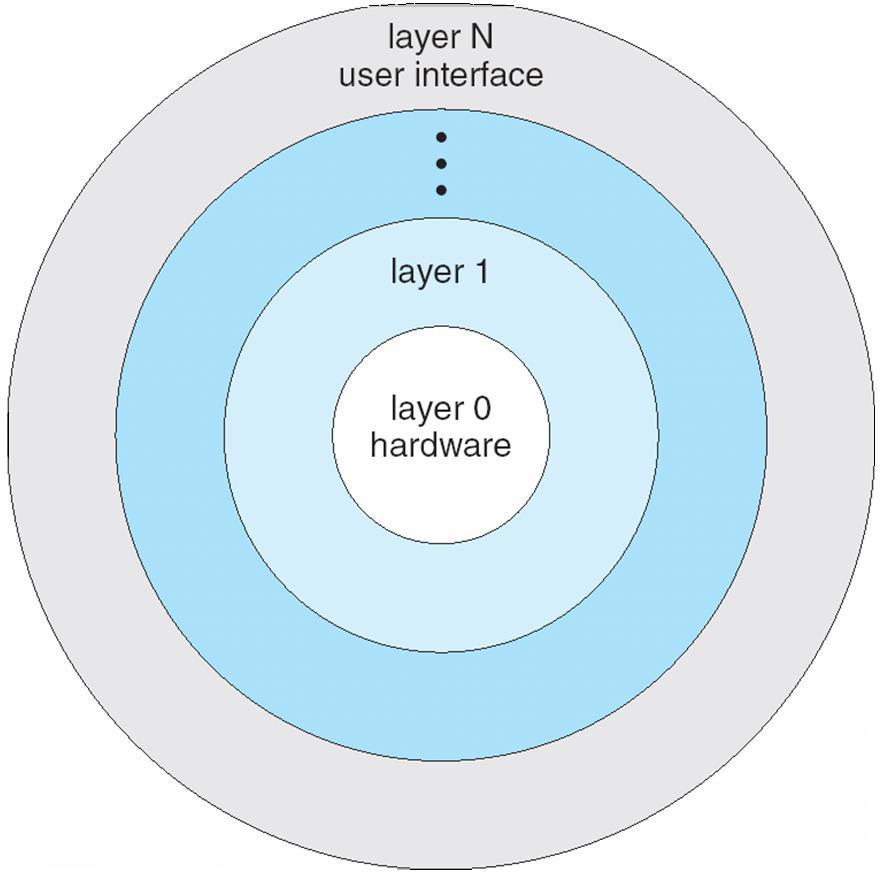
.

Layer 0:returns data back

🡨 slower because of communication you have to traverse through the different layers, overhead increases because of this. Remains as ideal idea conceptually.

better functionality, reliability is one-way. Lower layer doesn’t rely on top layers. Better security eliminates escalations issues.

can you jump from Layer N+1 to Layer N-1???



Microkernel System Structure

• Moves as much from the kernel into user space

•Mach example of microkernel

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

•Communication takes place between user modules using message passing

•Benefits:

• Easier to extend a microkernel

• Easier to port the operating system to new architectures

• More reliable (less code is running in kernel mode)

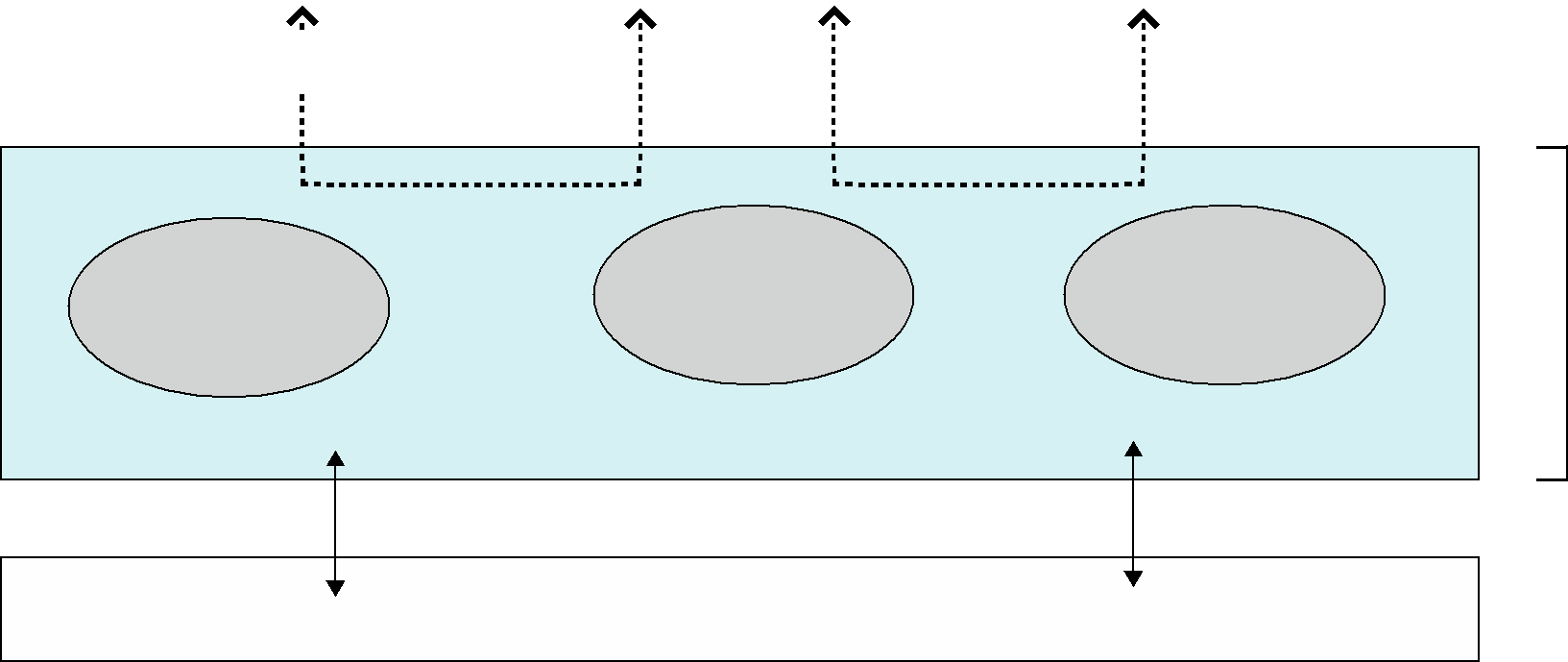
• More secure

• Detriments:

•Performance overhead of user space to kernel space communication

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Application |  | File |  | Device |  |  |
| Program |  | System |  | Driver |  |  |
|  |  |  |  |  |  |  |  |

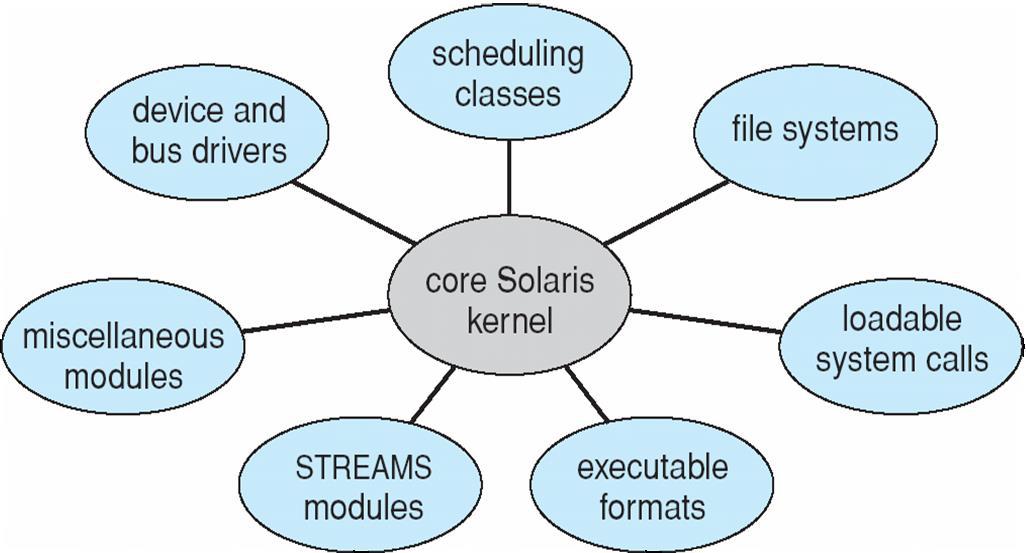
Microkernel System Structure



Modules

* Many modern operating systems implement loadable 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
* Overall, similar to layers but with more flexible
  + Linux, Solaris, etc.

Solaris Modular Approach



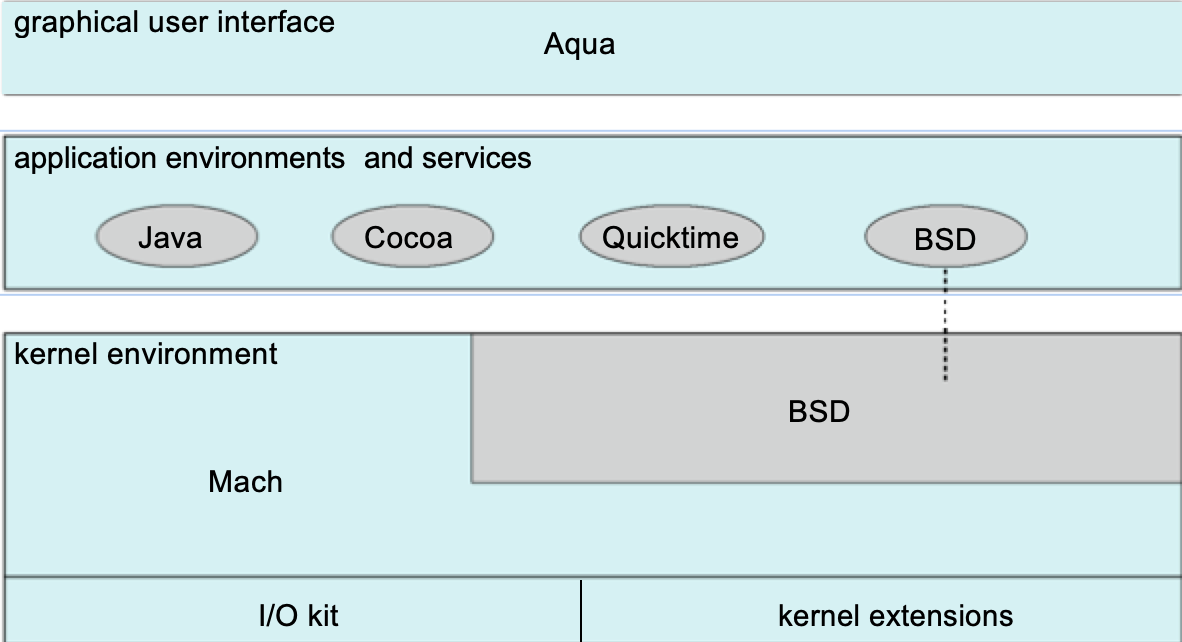
Hybrid Systems

🡨different modules can communicate with each other, extending the system is also ok. Security and reliability since we made Kernal smaller. Therefore, less likely to cause problems

function call will increase. Similar to layer approach. Outsource more functions to the core Solaris Kernal

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



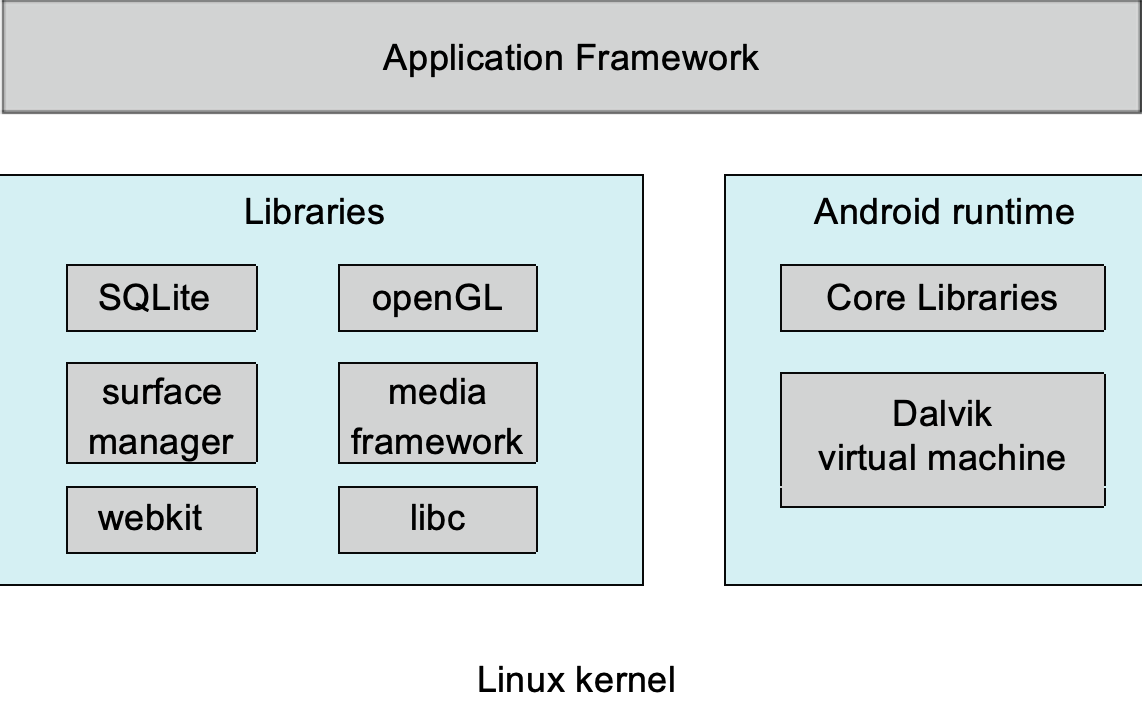
iOS

* 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 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 ArchitectureApplications



One thing you learned in this lecture: The functionality of each kernel and how they execute commands from the user

One thing I found unclear was with the layer approach and if its possible to jump between different layers.