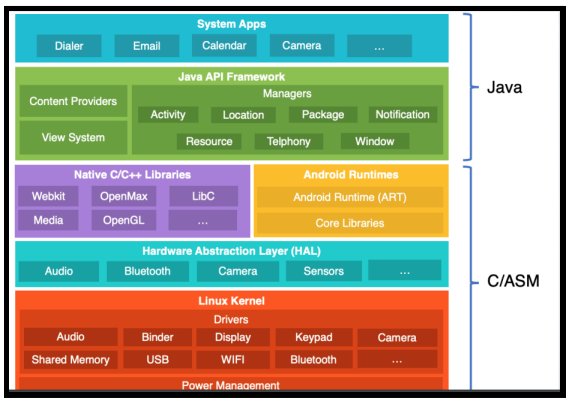
**Lecture Notes:**

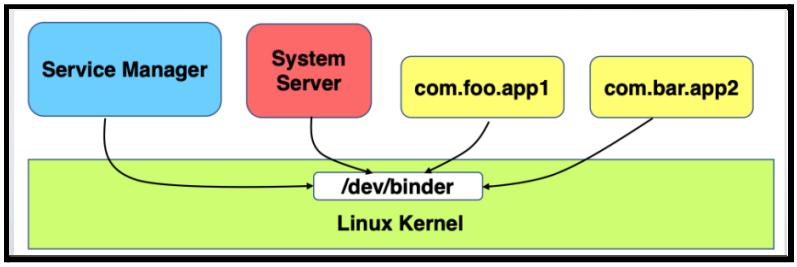
- **Mobile OS:**
- History of mobile OSes:
  - Early smart devices are PDAs (touchscreen, Internet).
  - Symbian is the first modern mobile OS. It was released in 2000 and ran in Ericsson R380, the first “smartphone" (mobile phone + PDA). It only supported proprietary programs.
  - Many smartphone and mobile OSes followed after:
    - Palm OS (2001)
    - Windows CE (2002)
    - Blackberry (2002)
    - Introduction of iPhone (2007) ← This was a game changer
      It had 4GB flash memory, 128 MB DRAM, and multi-touch interface.
      Originally, it only ran iOS proprietary apps but the App Store opened in 2008 and allowed third party apps.
- Design considerations for mobile OS:
  - Resources are very constrained:
    - Limited memory
    - Limited storage
    - Limited battery life
    - Limited processing power
    - Limited network bandwidth
    - Limited size
  - User perception are important: Latency $\gg$ throughput.
    Users will be frustrated if an app takes several seconds to launch.
  - The environment is frequently changing. Cellular signals change from strong to weak and then back to strong.
- Process management in mobile OS:
  - On a desktop/server, an application = a process. This is not true on mobile OSes.
  - On mobile OSes:
    - When you see an app present to you it does not mean an actual process is running.
    - Multiple apps might share processes.
    - An app might make use of multiple processes.
    - When you close an app, the process might be still running.
  - Multitasking is a luxury in mobile OS.
  - Early versions of iOS did not allow multi-tasking mainly because of battery life and limited memory.
  - Only one app runs in the foreground. All the other user apps are suspended.
  - The OS's tasks are multi-tasked because they are assumed to be well-behaving. Starting with iOS 4, the OS APIs allow multitasking in apps but are only available for a limited number of app types.
- Memory management in mobile OS:
  - Most desktop and server OSes today support swap space.
  - Mobile OSes typically do not support swapping.
  - iOS asks applications to voluntarily relinquish allocated memory.
  - Android will terminate an app when free memory is running low.
  - App developers must be very careful about memory usage.

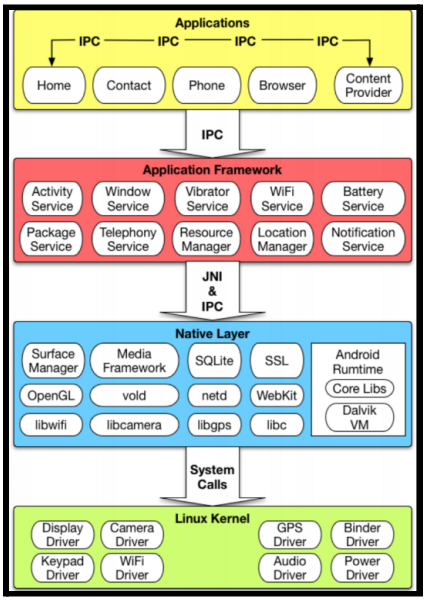
- Storage in mobile OS:
  - App privacy and security is hugely important in mobile devices.
  - Each app has its own private directory that other apps cannot access.
  - The only shared storage is external storage.
- Android:
  - History of Android:
    - Android Inc was founded by Andy Rubin et al. in 2003.
    - The original goal is to develop an OS for a digital camera.
    - Later, the focus shifted on Android as a mobile OS.
    - Android was later bought by Google.
    - Originally, no carrier wanted to support it except for T-Mobile. While preparing for the public launch of Android, the iPhone was released.
    - Android 1.0 was released in 2008 (HTC G1).
    - In 2019, Android has ~87% of the mobile OS market while iOS has ~13%.
  - Android OS stack:



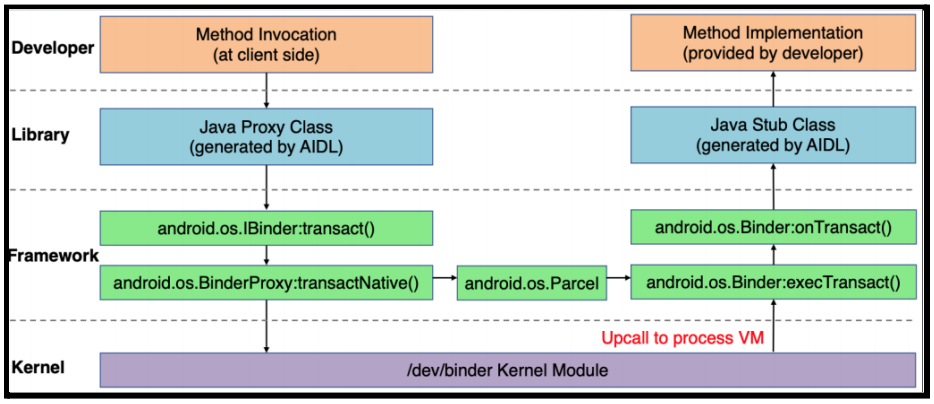

  - Linux kernel vs. Android kernel:
    - The Linux kernel is the foundation of the Android platform.
    - However, there are a few tweaks:
      - binder - interprocess communication mechanism
      - shmem - shared memory mechanism
      - logger
  - Android runtime:
    - **Runtime:** A component provides functionality necessary for the execution of a program.
      E.g. scheduling, resource management, stack behavior, etc
    - Prior to Android 5.0 (Dalvik):
      - Each Android app has its own process, runs its own instance of the Dalvik virtual machine (process virtual machine).
      - The VM executes the Dalvik executable (.dex).
      - The Dalvik virtual machine is register-based compared to stack-based of a JVM.
    - After Android 5.0 (ART):
      - Backward compatible for running Dex bytecode.
      - New feature - Ahead-Of-Time (AOT) compilation.
      - Improved garbage collection.

- Android process creation:
  - All Android apps derive from a process called Zygote.
  - Zygote is started as part of the init process.
  - It preloads Java classes, resources, and starts the Dalvik VM.
  - It registers a Unix domain socket.
  - It waits for commands on the socket.
  - It forks off child processes that inherit the initial state of VMs. It uses Copy-on-Write only when a process writes to a page will a page be allocated.
- Java API framework:
  - The main Android OS from app point of view.
  - It provides high-level services and environment to apps.
  - It interacts with low-level libraries and Linux kernels.
  - Some components:
    - Activity Manager - manages the lifecycle of apps.
    - Package Manager - keeps track of apps installed.
    - Power Manager - wakelock APIs to apps.
- Native C/C++ libraries:
  - Many core Android services are built from native code.
  - They require native libraries written in C/C++.
  - Some of them are exposed through the Java API framework as native APIs such as the Java OpenGL API
- Android Binder IPC:
  - Android Binder IPC allows communication among apps, between system services, and between app and system service.



- Binder is implemented as an RPC:
    1. Developer defines methods and object interface in an .aidl file.
    2. Android SDK generates a stub Java file for the .aidl file and exposes the stub in a Service.
    3. Developer implements the stub methods.
    4. Client copies the .aidl file to its source.
    5. Android SDK generates a stub (a.k.a proxy).
    6. Client invokes the RPC through the stub.
- Binder information flow:




- **OS Security:**
- Protection:
    - File systems implement a protection system:
        - Who can access a file?
        - How can they access it?
    - A protection system dictates whether a given action performed by a given subject on a given object should be allowed.
      E.g. You can read and/or write your files, but others cannot.
      E.g. You can read "/etc/motd", but you cannot write it.
- DAC vs MAC:
    - **DAC (Discretionary Access Control):** Users define their own policy on their own data.
    - **MAC (Mandatory Access Control):** The administrator defines a system level policy to control the propagation of data between users.
    - DAC and MAC are not exclusive and can be used together.
- Discretionary Access Control:
    - Unix protection on files:
        - Each process has a User ID and one or more group IDs.
        - The system stores the following with each file:
            - The user who owns the file and the group the file is in.
            - Permissions for users, any one in the file group, and other.
        - This is shown by the output of the "ls -l" command.
    - Unix protection on directories:
        - Directories have permission bits, too.
        - The write permission on a directory allows users to create or delete a file.
        - The execute permission allows users to use pathnames in the directory.
        - The read permission allows users to list the contents of the directory.
        - The special user root (UID 0) has all privileges. It is required for administration.

- Unix permissions on non-files:
  - Many devices show up in the file system.
    E.g. /dev/tty1
  - They have permissions just like for files. However, other access controls are not represented in the file system.
    E.g. You must usually be root to do the following:
    - Bind any TCP or UDP port number less than 1024.
    - Change the current process's user or group ID.
    - Mount or unmount most file systems.
    - Create device nodes (such as /dev/tty1) in the file system.
    - Change the owner of a file.
    - Set the time-of-day clock; halt or reboot machine.
- Setuid:
  - Some legitimate actions require more privileges than UID.
    E.g. how users change their passwords stored in root-owned /etc/passwd and /etc/shadow files?
  - The solution is the setuid and setgid programs.
    Run with privileges of the file's owner or group.
    Each process has a real and effective UID/GID.
    Real is a user who launched the setuid program.
    Effective is the owner/group of the file, used in access checks.
  - Have to be very careful when writing setuid code.
    Attackers can run setuid programs any time (no need to wait for root to run a vulnerable job).
    Attacker controls many aspects of the program's environment.
- Unix security hole: Even without root or setuid attackers can trick root owned processes into doing things.

- Mandatory Access Control:
  - Mandatory access control (MAC) can restrict propagation.
    E.g. A security administrator may allow you to read but not disclose the file.
  - MAC prevents users from disclosing sensitive information whether accidentally or maliciously.
    E.g. Classified information requires such protection.
  - MAC prevents software from surreptitiously leaking data. Seemingly innocuous software may steal secrets in the background (Trojan Horse).