Distributed Computing

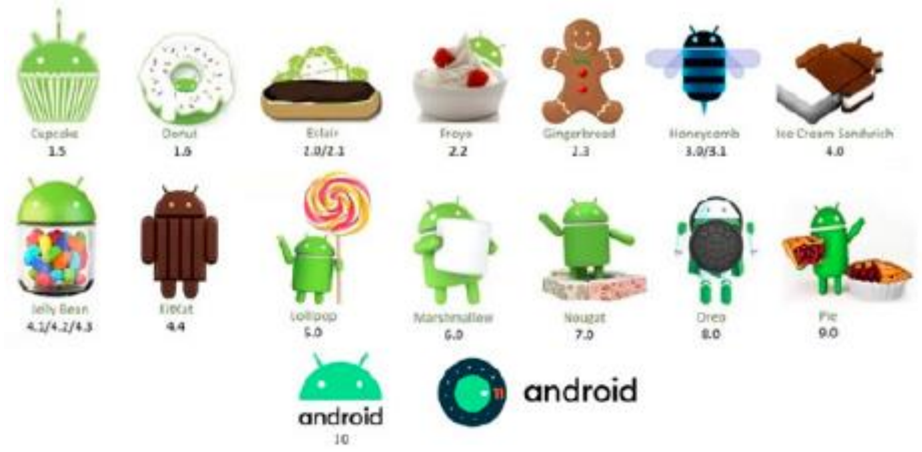
Experiment – 10

Case Study: Android Stack	
Learning Objective:	Describe the concepts of distributed File Systems with some case studies
Learning Outcome:	Ability to perform case study on distributed file system.
Course Outcome:	**CSL801.6**
Program Outcome:	(**PO**1) **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. (**PO**2) **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. **(PO11) Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.** **(PO12) Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.
Bloom's Taxonomy Level:	Understanding
Theory:	**Case Study: Android Stack** **1. Introduction** Android, originating from a customized Linux kernel and a diverse array of open-source software elements, has emerged as a pivotal player in the landscape of mobile operating systems. Tailored primarily for touchscreen-centric devices like smartphones and tablets, Android embodies a dynamic ecosystem characterized by innovation and accessibility. At its essence, Android is designed to meet the unique requirements of mobile devices, particularly those reliant on battery power. These devices, equipped with a plethora of hardware features such as GPS receivers, cameras, light sensors, and orientation sensors, alongside connectivity options like Wi-Fi and LTE, depend on Android to facilitate seamless interactions between software

and hardware.
Fundamental to Android's functionality is its role as an orchestrator, abstracting the complexities of hardware access and management for applications. By furnishing a standardized environment, Android empowers developers to harness the complete potential of device hardware sans the intricacies. This abstraction layer not only streamlines application development but also fosters uniformity and interoperability across a broad spectrum of devices.
Android's architecture is structured to enable seamless integration of applications with underlying hardware features. Whether it entails leveraging GPS for location-based services, accessing the camera for multimedia capture, or utilizing sensors for intuitive user interactions, Android offers a robust framework for applications to leverage the capabilities of modern mobile devices.
Acting as a conduit between hardware and software, Android provides a cohesive platform that enables developers to craft innovative and immersive user experiences. By furnishing a stable and standardized environment, Android propels the ongoing evolution of mobile technology, empowering users and developers to explore the frontiers of what's achievable on handheld devices.

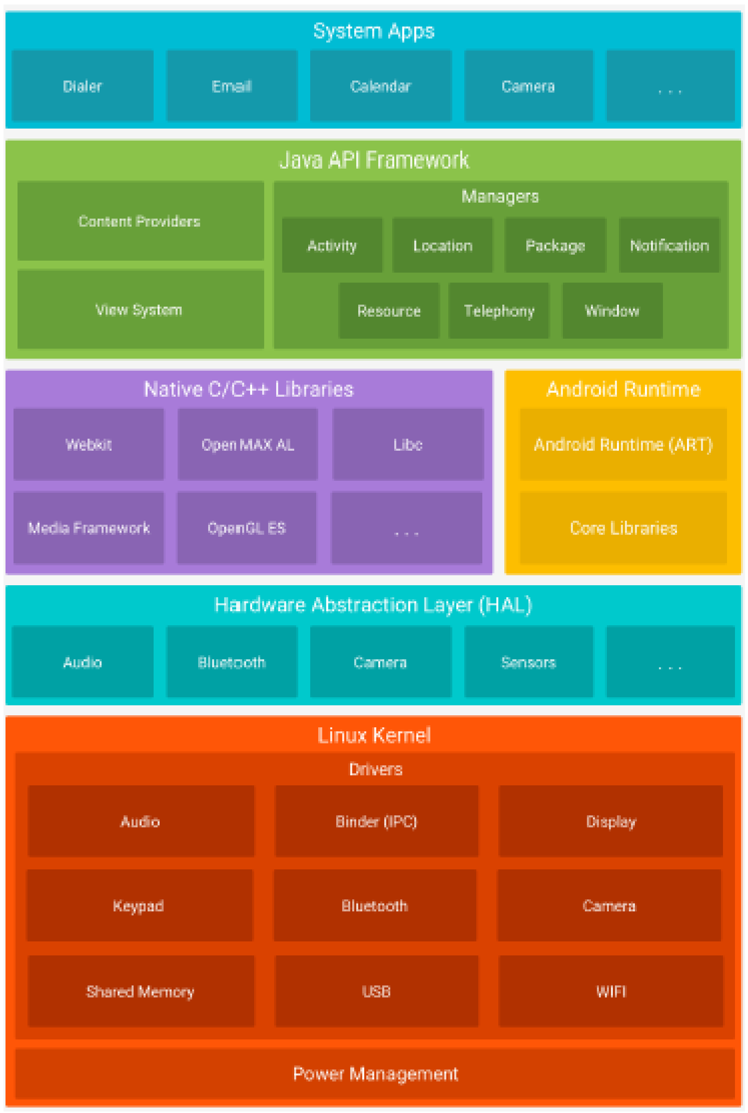
2. History

The origins of the Android platform can be traced back to Android Inc., a startup established in 2003 with a vision to pioneer an advanced operating system tailored for mobile devices. In a significant move in 2005, Google acquired Android Inc., marking a pivotal moment that laid the groundwork for what would eventually evolve into one of the most influential mobile platforms worldwide.
Subsequent to the acquisition, Google introduced the Android Open Source Project (AOSP) in 2007, heralding the commencement of Android's journey as an open-source operating system. This strategic decision democratized mobile development, empowering a global community of developers to actively contribute to the platform's innovation and expansion.
Since its inception, Android has undergone a series of updates and iterations, each aimed at enriching the platform's capabilities and refining the user experience. These updates, released in alphabetical order and christened after desserts or sugary treats, have become emblematic of the Android ecosystem.
The early iterations of Android, including "Cupcake," "Donut," "Eclair," and "Froyo," introduced fundamental features and laid the groundwork for future advancements. With each major release, Google introduced a plethora of new functionalities, performance enhancements, and bug fixes, propelling the platform's evolution forward.



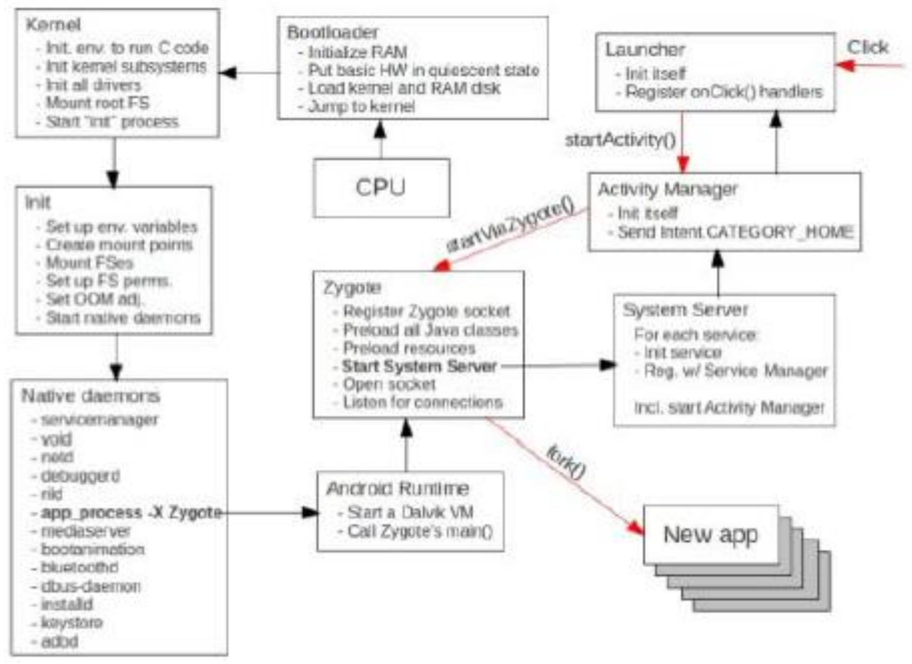
In 2013, Google unveiled Android KitKat, continuing the tradition of naming releases after delectable desserts. The rationale behind this naming convention, as explained by Google, underscored the company's commitment to celebrating the sweetness of technology and its profound impact on people's lives.

3. Android Architecture

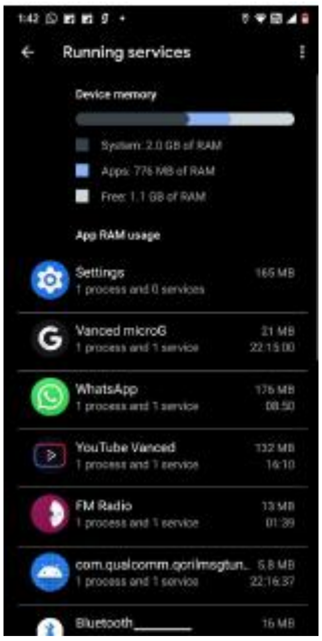


1. **Linux Kernel:** At the heart of the Android platform lies the robust Linux kernel, which provides essential functionalities like threading and low-level memory management. Leveraging this widely recognized kernel not only ensures smooth operation but also grants access to robust security features. Furthermore, it enables device manufacturers to develop tailored hardware drivers, leveraging a well-supported kernel.
2. **Hardware Abstraction Layer (HAL):** Serving as a bridge between the Android framework and underlying hardware components, the HAL offers standardized interfaces for seamless communication. This abstraction enables applications to access device capabilities such as the camera or Bluetooth module without delving into hardware-specific intricacies, fostering interoperability and ease of development.
3. **Android Runtime (ART):** Each Android application operates within its own managed process under ART, executing optimized bytecode known as DEX files. This bytecode minimizes memory usage, particularly crucial for resource-constrained devices. The efficient runtime environment ensures smooth performance and optimal resource utilization, enhancing the user experience.
4. **Native C/C++ Libraries:** Core system components like ART and the HAL are implemented using native code written in C and C++. While interacting with the hardware, these components expose their functionalities through Java framework APIs. This approach enables developers to leverage powerful features like OpenGL ES for graphics rendering through Java APIs, enriching the capabilities of Android applications.
5. **Java API Framework:** The comprehensive Java API Framework encapsulates the entire feature set of the Android operating system, empowering developers with a rich set of APIs for application development. These APIs abstract system-level complexities, offering access to features such as UI building, resource management, notifications, and activity lifecycle management, simplifying the development process.
6. **System Apps:** Android comes pre-loaded with core applications for essential tasks such as email, messaging, browsing, and contacts management. Despite being pre-installed, these system apps hold no special status compared to third-party applications. Users retain the freedom to replace default system apps with alternatives, underscoring the platform's customizable and user-centric nature.

4. Kernel and Startup



Android employs a sophisticated process management system to efficiently allocate system resources and ensure a smooth user experience. Understanding the lifecycle hierarchy of processes and the mechanisms for inter-process communication (IPC) is essential for optimizing app performance and resource utilization. Here's a comprehensive overview:
Process Lifecycle Hierarchy: Android categorizes processes into five distinct states based on their importance and visibility:


1:42

Running services

Device memory

▪ System: 2.0 GB of RAM
▪ Apps: 776 MB of RAM
▪ Free: 1.1 GB of RAM

App RAM usage

Settings 1 process and 0 services	165 MB
Vanced microG 1 process and 1 service	21 MB 22:15:00
WhatsApp 1 process and 1 service	176 MB 08:50
YouTube Vanced 1 process and 1 service	132 MB 16:10
FM Radio 1 process and 1 service	13 MB 01:39
com.qualcomm.qcrilmsgtun... 1 process and 1 service	5.8 MB 22:16:37
Bluetooth	16 MB

1. **Foreground Process:** The currently active app or any app visible on the screen is categorized as a foreground process. Additionally, apps with ongoing activities or services that interact with the user may also fall into this classification, signifying their immediate relevance to the user experience.
2. **Visible Process:** While not in the foreground, a visible process still influences the user interface. This category encompasses background services or activities that have an indirect impact on the UI, contributing to the overall user experience without being the primary focus of attention.
3. **Service Process:** Processes housing background services, which lack direct ties to visible UI components, are classified as service processes. These services execute tasks that don't necessitate user interaction but continue to run in the background, enhancing the functionality of the application or system.
4. **Background Process:** Invisible to the user, background processes operate without direct user interaction and have minimal impact on the user experience. Although maintained in memory for quick access if required, they do not significantly consume CPU or non-memory resources, ensuring efficient resource management.
5. **Empty Process:** An empty process contains no active application data and may be retained for caching purposes or terminated by the system to reclaim resources as necessary. These processes serve auxiliary roles, aiding system performance and resource utilization without directly contributing to user-facing functionalities.

Threads: Threads are units of execution within a program that allow concurrent execution of tasks. In Android, the Java Virtual Machine (JVM) supports multiple threads, each with its own priority level. Higher-priority threads are given preference in execution. Additionally, threads can be marked as daemon threads, which are automatically terminated when all non-daemon threads have completed.

Inter-Process Communication (IPC): Android provides several mechanisms for IPC, facilitating communication between different processes running on the system. These mechanisms include Intents, Binder or Messenger with a Service, and BroadcastReceiver. Each mechanism offers unique advantages and can be tailored to specific use cases. Additionally, Android's IPC mechanisms incorporate security features to verify the identity of connecting applications and enforce security policies.

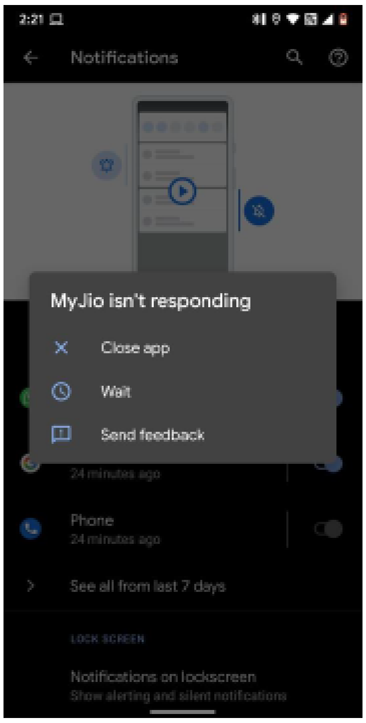
There are two primary methods through which processes communicate with each other in the Android ecosystem:

1. **Using Intents:** Intents are a mechanism for performing actions within an application or between different components of an application. They are commonly utilized to initiate activities, broadcast receivers, and services, as well as to facilitate communication between two activities. Intents enable seamless interaction and data exchange between various components of an Android application, enhancing its functionality and user experience.

2. **Using Android Interface Definition Language (AIDL):** AIDL allows developers to define a programming interface that both the client and service agree upon for communication via Inter-Process Communication (IPC). In Android, processes are typically isolated, and direct memory access between them is restricted. AIDL enables processes to communicate by decomposing objects into primitives understandable by the operating system, and marshalling them across process boundaries. This facilitates efficient and secure communication between different components of an Android application, enabling them to exchange data and perform tasks collaboratively.

5. How is DeadLock Handled?

In the context of multiprocessing systems, deadlocks can pose significant challenges to system stability and performance. Android, being a complex operating system, employs strategies to handle deadlocks effectively. One such strategy is detecting and recovering from deadlocks, acknowledging the difficulty in preventing them entirely. Here's an overview of how Android deals with deadlocks and the techniques it utilizes:



MyJio isn't responding
Close app
Wait
Send feedback

Detect and Recover Approach: Android adopts a detect and recover approach to manage deadlocks. Rather than striving to prevent deadlocks entirely, which can be challenging in a dynamic and resource-intensive environment like Android, the system allows deadlocks to occur. Upon detection, the system initiates recovery mechanisms to resolve the deadlock

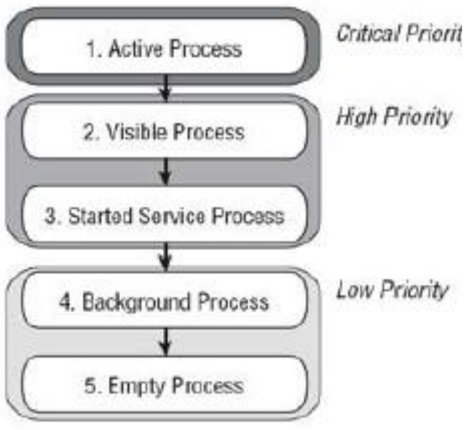
and restore normal functionality.
Detection Mechanisms: Deadlocks in Android are identifiable due to the inherent characteristics of deadlock situations. In a deadlock scenario, each process retains resources while awaiting others, resulting in a cyclic dependency. Android's system algorithms continuously monitor resource allocation and process states, accurately identifying deadlock situations.
Recovery Process: Upon detection of a deadlock, Android employs specialized algorithms to track resource allocation and process states. These algorithms analyze the deadlock situation and execute recovery strategies to break the deadlock. Typically, the system may intervene by forcibly releasing resources held by certain processes or restarting affected processes to restore system functionality.
Preventing Deadlocks: Although Android primarily focuses on detecting and recovering from deadlocks, preventing them entirely is preferable. Deadlock prevention entails ensuring that the conditions leading to deadlocks do not materialize. These conditions include:

1. Mutual Exclusion: Preventing simultaneous resource access by processes.
2. Hold and Wait: Prohibiting processes from holding resources while awaiting additional resources.
3. No Preemption: Restricting the forced removal of resources from processes.
4. Circular Wait: Ensuring the absence of a circular chain of processes, each waiting for a resource held by the next.

Banker's Algorithm: One effective method for deadlock avoidance is the Banker's Algorithm. It is a resource allocation and deadlock avoidance algorithm that evaluates the safety of resource requests before granting them. By analyzing the system's current state and potential resource allocations, the Banker's Algorithm determines if granting a resource request would lead to a safe state or risk deadlock.

Preventing Recursive Locks: Another measure to prevent deadlocks involves avoiding recursive locks, where a process attempts to acquire a lock it already holds. Preventing recursive locks reduces the likelihood of deadlock situations arising from improper resource management within processes.

6. CPU Scheduling



The Android operating system incorporates the O(1) scheduling algorithm, derived from the Linux Kernel 2.6, known as the Completely Fair Scheduler (CFS). This algorithm ensures consistent scheduling of processes within a constant time frame, regardless of the number of processes running concurrently.
However, the conventional "fair" scheduling policy of Linux can be influenced by the "nice" level of a process. Threads with a higher niceness experience less frequent execution compared to those with a lower niceness. Android mitigates this issue by organizing threads into distinct cgroups (control groups) based on their priority levels.
Here's how Android utilizes cgroups to enforce stringent foreground and background scheduling:

1. **Foreground vs. Background Scheduling:** Android segregates threads into two primary cgroups: foreground/default and background. Threads within the foreground/default cgroup undergo regular scheduling, while those in the background cgroup are allocated a restricted portion of CPU time.
2. **Thread Priority and Cgroup Assignment:** Threads are assigned to the foreground/default or background cgroups depending on their priority levels. Threads with elevated priorities, such as foreground/UI threads, are placed in the foreground/default cgroup to ensure they are minimally affected by background activities. Conversely, threads with lower priorities, such as background worker threads, are allocated to the background cgroup, where their CPU time is limited.
3. **Implicit Cgroup Assignment for Processes:** Android automatically relocates entire processes to the background cgroup if they are deemed non-essential to the user. This encompasses background processes or service processes, irrespective of whether individual threads within these processes have requested foreground scheduling priority.

7. **Management Studies -**

- **Memory Management**

Memory management is a critical aspect of the Android Runtime (ART) and the Dalvik virtual machine, ensuring efficient utilization of system resources and optimal app performance. Here's an overview of how memory is managed in Android, including garbage collection, shared memory, allocation, and restriction:

Paging and Memory-Mapping:

- Android Runtime and Dalvik utilize paging and memory-mapping techniques to manage memory efficiently.
- Any memory modifications made by an app, such as allocating new objects or touching memory-mapped pages, remain resident in RAM and cannot be paged out.
- However, files memory-mapped without modification, such as code, can be paged out of RAM if needed by the system.

Garbage Collection:

- Managed memory environments like ART and Dalvik implement garbage collection to reclaim unused memory.
- Garbage collection identifies data objects that cannot be accessed in the future and releases the resources used by those objects back to the heap.

Shared Memory:

- Android optimizes RAM usage by sharing memory across processes through various techniques:
    1. Apps are forked from an existing process called Zygote, allowing common framework code and resources to be shared.
    2. Static data is mapped into processes, facilitating sharing between processes and enabling paging out when necessary.
    3. Android utilizes explicitly allocated shared memory regions (e.g., ashmem or gralloc) to share dynamic RAM across processes.

Allocation and Reclaiming App Memory:

- The Dalvik heap is constrained to a single virtual memory range for each app process, defining the logical heap size.
- The heap size can grow as needed but is limited by the system-defined limit for each app.
- When an app exceeds its heap capacity and attempts to allocate more memory, it may receive an OutOfMemoryError.

Restriction on App Memory:

- Android sets a hard limit on the heap size for each app to maintain a functional multitasking environment.
- The exact heap size limit varies based on the available RAM on the device.

Switching Between Apps:

- Android caches apps that are not in the foreground or running a foreground service in a cache when users switch between apps.
- Cached processes allow for faster app switching, as the system reuses the process when the user returns to the app.

- **Storage Management**

  Managing internal storage in the Android operating system involves two main tasks: monitoring storage space usage and providing apps with read/write access to internal storage. Here's an overview of how these tasks are handled in Android 10 and above:

  Monitoring Storage Space:
  - The operating system continuously monitors the internal storage space to prevent it from becoming full. When storage space reaches a certain threshold, the system may trigger alerts or take actions to free up space, such as clearing cache files or prompting the user to delete unused data.

  Access to Internal Storage:
  - In Android 10 and above, apps are given limited access to internal storage, particularly restricting access to media elements of the device.
  - Apps are granted permissions to read/write data in specific directories within internal storage, ensuring data security and privacy.
  - Most Android devices utilize e-MMC-based storage, offering read and write speeds of up to 300MBps, ensuring efficient data access and manipulation.

  File System Structure:
  - Android uses a Linux-based file system structure, with a root directory and various partitions, each serving different purposes.
  - The root directory contains essential system files and directories, while other partitions store user data, system files, and recovery information.
  - Only internal storage or memory cards are accessible to the user, while other partitions are managed by the operating system itself for system stability and security.

  StorageManager Interface:
  - The StorageManager interface serves as the gateway to the system's storage service, handling storage-related tasks such as managing Opaque Binary Blobs (OBBs).
  - OBBs contain a filesystem that can be encrypted on disk and mounted on-demand by applications.
  - OBBs are used to provide large binary assets without packaging them into APKs, but their security cannot be guaranteed if modified by any program.

- **I/0 Management**

  The management of various Input/Output (I/O) devices is a crucial function of an operating system, ensuring seamless communication between applications and physical devices. Here's an overview of how operating systems handle I/O devices:

  Types of I/O Devices:
  1. **Block Devices:** Devices where data is transferred in entire blocks, such as hard disks, USB cameras, and Disk-On-Key (USB flash drives).
  2. **Character Devices:** Devices where data is transferred one character at a time, like serial ports, parallel ports, and sound cards.

Device Controllers and Drivers:

- Device drivers are software modules that allow the operating system to interact with specific hardware devices.
- Device controllers act as intermediaries between devices and device drivers, managing the communication between the electronic and mechanical components of I/O units.

Synchronous vs Asynchronous I/O:

- **Synchronous I/O:** CPU execution waits while the I/O operation proceeds.
- **Asynchronous I/O:** I/O operation proceeds concurrently with CPU execution, allowing for improved system responsiveness and efficiency.

Communication with I/O Devices:

1. **Special Instruction I/O:** CPU instructions specifically designed for controlling I/O devices, enabling data transfer to and from devices.
2. **Memory-Mapped I/O:** Sharing the same address space for memory and I/O devices, allowing devices to transfer data directly to/from memory without CPU involvement.
3. **Direct Memory Access (DMA):**
    - DMA grants I/O modules authority to read from or write to memory without CPU intervention.
    - DMA hardware controls data exchange between main memory and I/O devices, reducing CPU overhead.
    - The CPU is only involved at the beginning and end of the transfer, minimizing interruptions.

DMA is particularly beneficial for fast devices like disks, as it reduces the burden on the CPU by transferring entire blocks of data without frequent interrupts. This enhances system efficiency and performance, especially in scenarios with high I/O demand.

The Kernel I/O Subsystem plays a critical role in managing Input/Output operations within an operating system. It provides various services to efficiently handle I/O requests and optimize data transfer between devices and applications. Here's an overview of the services provided by the Kernel I/O Subsystem:

1. **Scheduling:**
    - The Kernel orchestrates I/O requests through scheduling mechanisms to optimize system efficiency and response time.
    - Blocking I/O system calls initiated by applications are queued on device queues and reorganized by the I/O scheduler for efficient execution.
2. **Buffering:**
    - The Kernel manages buffers, temporary memory areas utilized to temporarily store data during its transfer between devices or between devices and applications.
    - Buffering facilitates the synchronization of data flow between data producers and consumers, as well as accommodating varying data transfer rates between devices.
3. **Caching:**
    - The Kernel maintains cache memory, a segment of fast-access

memory storing duplicates of frequently accessed data.
    - Accessing cached data offers superior efficiency compared to accessing the original data directly, thereby enhancing overall system performance.
4. **Spooling and Device Reservation:**
    - Spooling involves buffering output intended for devices such as printers that cannot accept interleaved data streams.
    - The spooling system sequentially copies queued spool files to devices, managed either by a system daemon process or an in-kernel thread.
5. **Error Handling:**
    - The Kernel I/O Subsystem oversees error detection and recovery procedures, safeguarding against hardware malfunctions and application errors to uphold system reliability.
6. **Main Memory and I/O Device Interaction:**
    - During data transfer, CPU involvement is minimized, limited to initiating and concluding the transfer.
    - Direct Memory Access (DMA) enables I/O modules to read from or write to memory without requiring CPU intervention, thus reducing CPU overhead during data transfer operations.

8. Battery Optimization

Optimizing battery consumption is crucial for ensuring a positive mobile user experience. Here are three important strategies to make your app power-efficient:

1. **Lazy First Approach:**
    - Prioritize minimizing battery usage by reducing, deferring, and coalescing operations.
    - **Reduce:** Identify and eliminate redundant or unnecessary operations within your app to reduce energy consumption.
    - **Defer:** Delay non-urgent actions, reducing the frequency of active states and conserving battery power.
    - **Coalesce:** Group related tasks together to minimize device activations, optimizing energy usage.
2. **Take Advantage of Platform Features:**
    - Android offers a plethora of APIs and features tailored for battery optimization.
    - **Implement Relevant APIs:** Incorporate relevant Android APIs into your app to leverage system-level optimizations for better power efficiency.
    - **Utilize Internal Platform Mechanisms:** Make use of internal platform mechanisms designed to conserve battery life, even if they aren't directly programmable APIs.
    - **Stay Informed:** Remain updated about system-level processes and features that could impact battery usage, ensuring your app adapts to optimize power consumption effectively.

	3. **Use Battery Profiling Tools:** • Employ battery profiling tools to identify components and operations within your app that drain battery excessively. • **Analyze App Behavior:** Utilize battery profiling tools to analyze your app's behavior and pinpoint areas for optimization. • **Monitor Usage Patterns:** Regularly monitor battery usage patterns to detect any anomalies and take necessary corrective actions to enhance power efficiency.
Conclusion:	In conclusion, this case study has provided a holistic insight into the Android operating system stack, offering a thorough understanding of its architecture and components. From its reliance on the Linux kernel to the higher-level frameworks like the Android Runtime and Java API, we've dissected the intricate layers that underpin Android's functionality and versatility in mobile development. By exploring concepts such as memory management, I/O subsystems, and battery optimization strategies, we've not only delved into the technical intricacies of Android but also gleaned valuable insights into crafting efficient and user-centric applications. This comprehensive understanding equips developers with the knowledge and tools necessary to leverage Android's capabilities effectively and deliver exceptional user experiences in the ever-evolving mobile landscape.
References:	1. https://developer.android.com/guide/platform 2. https://en.wikipedia.org/wiki/Android_(operating_system) 3. https://android-developers.googleblog.com/2021/08/android-app-excellence-duolingo.html 4. https://cs.paperswithcode.com/paper/android-os-case-study

Rubrics for Assessment

Timely Submission	Late	Submitted just after deadline	On time Submission
	0 points	*1 points*	*2 points*
Understanding	Student is confused about the concept	Students has justifiably understood the concept	Students is very clear about the concepts
	0 points	*2 points*	*3 points*
Performance	Students has not performed the Experiment	Student has performed with help	Student has independently performed the experiment
	0 points	*2 points*	*3 points*
Development	Student struggles to write code	Student can write code with the requirement stated	Student can write exceptional code with his own ideas
	0 points	*2 points*	*3 points*
Overall Discipline in Lab	No Discipline maintained	Follows Rules	Sincerely performs the practical is always on time
	0 points	*1 points*	*4 points*