CHAPTER FIVE

LINUX SYSTEMS AND MOBILE OPERATING SYSTEMS

Linux System design Principles, Kernel Modules, Process Management, Scheduling, Memory Management, Input-Output Management, File System, Inter-process Communication are described in this chapter. Also describes Mobile OS, iOS and Android Architecture and its SDK Framework, Media Layer, Services Layer, Core OS Layer and File System are described.

Linux System

Linux is a modern, free operating system based on UNIX standards. First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility. Linux uses many tools developed as part of Berkeley’s BSD operating system, MIT’s X Window System, and the Free Software Foundation’s GNU project.

The main system libraries were started by the GNU project, with improvements provided by the Linux community. Linux networking-administration tools were derived from 4.3BSD code. Recent BSD derivatives such as FreeBSD have borrowed code from Linux in return. The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

Design principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools.
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Main design goals are speed, efficiency, and standardization.

- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

The Linux design has the following components.

Components of a Linux System:

The Linux system is composed of three main bodies of code, the most important distinction is between the kernel and all other components.

Kernel. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.

Kernel code executes in kernel mode with full access to all the physical resources of the computer. All kernel code and data structures are kept in the same single address space.

System libraries. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the **C library**, known as libc.

System utilities. The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others known as **daemons** in UNIX terminology run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

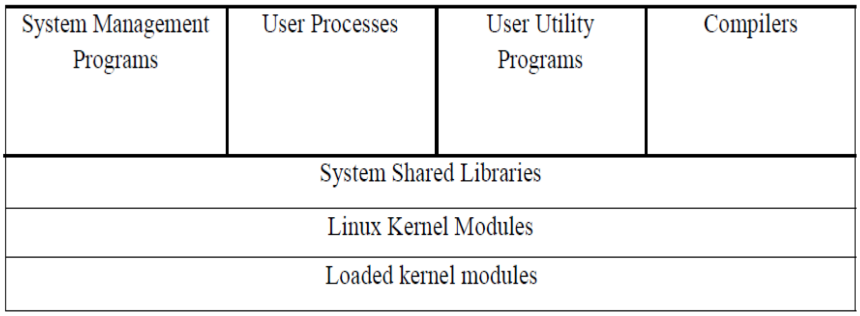


Fig 5.1 Components of a Linux System

All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer.

Linux refers to this privileged mode as **kernel mode**. Under Linux, no user code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries and runs in **user mode**. Unlike kernel mode, user mode has access only to a controlled subset of the system's resources.

One of the most important user utilities is the **shell**, the standard command-line interface on UNIX systems. Linux supports many shells; the most common is the **bourne-again shell** (bash).

Kernel Modules

Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel. A kernel module may typically implement a device driver, a file system, or a networking protocol. The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

Three components to Linux module support:

Module management, Driver registration, Conflict resolution.

Module Management:

Supports loading modules into memory and letting them talk to the rest of the kernel. Module loading is split into two separate sections:

- Managing sections of module code in kernel memory
- Handling symbols that modules are allowed to reference

The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use and will unload it when it is no longer actively needed.

Driver registration:

Allows modules to tell the rest of the kernel that a new driver has become available. The kernel maintains dynamic tables of all known drivers, provides a set of routines to allow drivers to be added to or removed from these tables at any time.

Registration tables include the following items:

- Device drivers
- File systems
- Network protocols
- Binary format

Conflict resolution:

A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

The conflict resolution module aims to:

- Prevent modules from clashing over access to hardware resources

- Prevent auto probes from interfering with existing device drivers
- Resolve conflicts with multiple drivers trying to access the same hardware

Process Management

A **process** is the basic context in which all user-requested activity is serviced within the operating system.

UNIX **process management separates the creation of processes and the running of a new program** into two distinct operations.

- The fork() system call creates a new process.
- A new program is run after a call to exec().

Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program.

Under Linux, process properties fall into three groups:

- Process Identity
- Environment
- Context.

Process Identity

A process identity consists mainly of the following items:

Process ID (PID): Each process has a unique identifier. The PID is used to specify the process to the operating system when an application makes a system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.

Credentials: Each process must have an associated user ID and one or more group IDs that determine the rights of a process to access system resources and files.

Personality: Process personalities are special feature in Linux - each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.

Process Environment

A **process's environment is inherited from its parent**. It is composed of two null-terminated vectors:

- The argument vector
- The environment vector.

The argument vector simply lists the command-line arguments used to invoke the running program;

The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.

Process Context

Process context is the state of the running program at any one time, like (ready, running , execution, terminated , waiting). It changes constantly.

Process context includes the following parts

Scheduling context: Scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process.

Accounting: The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.

File table: The file table is an array of pointers to kernel file structures representing open files. When making file-I/O system calls, processes refer to files by an integer, known as a file descriptor (fd), that the kernel uses to index into this table.

File-system context: Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.

Signal-handler table: UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the action to take in response to a specific signal. Valid actions include ignoring the signal, terminating the process, and invoking a routine in the process's address space.

Virtual memory context: The virtual memory context describes the full contents of a process's private address space. Linux provides the fork() system call, which duplicates a process without loading a new executable image.

Linux also provides the ability to create threads via the clone() system call. Linux does not distinguish between processes and threads. In fact, Linux generally uses the term task, rather than process or thread when referring to a flow of control within a program.

The clone() system call behaves identically to fork(), except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child (whereas a process created with fork() shares no resources with its parent).

Process Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports preemptive multitasking.

In such a system, the process scheduler decides which process runs and when. Linux has two separate process-scheduling algorithms.

- Time-sharing algorithm
- Completely Fair Scheduler (CFS)

Time-sharing algorithm: is designed for fair, preemptive scheduling among multiple processes (like round robin).

Completely Fair Scheduler (CFS): is designed for real-time tasks, where absolute priorities are more important than fairness (like priority). This Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a real-time range from 0 to 99 and a nice value ranging from 20 to 19.

Smaller nice values indicate higher priorities. Thus, by increasing the nice value, you are decreasing your priority and being “nice” to the rest of the system. CFS introduced a new scheduling algorithm called fair scheduling that eliminates time slices in the traditional sense.

Instead of time slices, all processes are allotted a proportion of the processor’s time. CFS calculates how long a process should run as a function of the total number of runnable processes.

To start, CFS says that if there are N runnable processes, then each should be afforded 1/N of the processor’s time. CFS then adjusts this allotment by weighting each process’s allotment by its nice value.

Processes with the default nice value have a weight of 1 their priority is unchanged. Processes with a smaller nice value (higher priority) receive a higher weight, while processes with a larger nice value (lower priority) receive a lower weight. CFS then runs each process for a “time slice” proportional to the process’s weight divided by the total weight of all runnable processes.

Memory Management

Linux’s physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory. It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.

Memory management under Linux has two components.

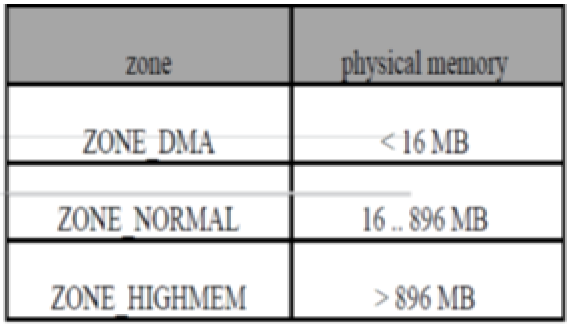
- **Management of Physical Memory:** Deals with allocating and freeing physical memory, pages, groups of pages, and small blocks of RAM.
- **Management of Virtual Memory:** Virtual memory, which is memory-mapped into the address space of running processes.

Management of Physical Memory

Due to specific hardware constraints, Linux separates physical memory into four different zones, or regions:

- **ZONE DMA** (the first 16 MB of physical memory comprise ZONE DMA)
- **ZONE DMA32**(certain devices can only access the first 4 GB of physical memory, despite supporting 64-bit addresses)
- **ZONE NORMAL** (comprises everything else, the normal, regularly mapped pages.)
- **ZONE HIGHMEM** (for “high memory”) refers to physical memory that is not mapped into the kernel address space)

The region is broken up recursively until a piece of the desired size is available.



zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Fig 5.2 Relationship of zones and physical addresses in Intel x86-32

All memory allocations in the Linux kernel are made either statically by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator.

The most important are the virtual memory system, the kmalloc() variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Analogous to the C language's malloc() function, this kmalloc() service allocates entire physical pages on demand but then splits them into smaller pieces. The kernel maintains lists of pages in use by the kmalloc() service.

Memory regions claimed by the kmalloc() system are allocated permanently until they are freed explicitly with a corresponding call to kfree(). Another strategy adopted by Linux for allocating kernel memory is known as slab allocation.

A slab is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A cache consists of one or more slabs.

There is a single cache for each unique kernel data structure — for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for inodes, and so forth.

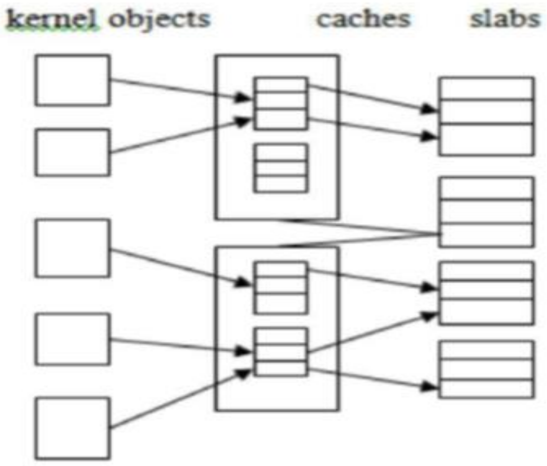


Fig 5.3 Slab Allocation

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. In Linux, a slab may be in one of three possible states:

Full: All objects in the slab are marked as used.

Empty: All objects in the slab are marked as free.

Partial: The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exist, a free object is assigned from an empty slab.

If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Buddy heap allocation:

The allocator uses a buddy system to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner (or buddy).

Whenever two allocated partner regions are freed up, they are combined to form a larger region a buddy heap. That larger region also has a partner, with which it can combine to form a still larger free region.

Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size.

Under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 5.3 shows an example of buddy-heap allocation.

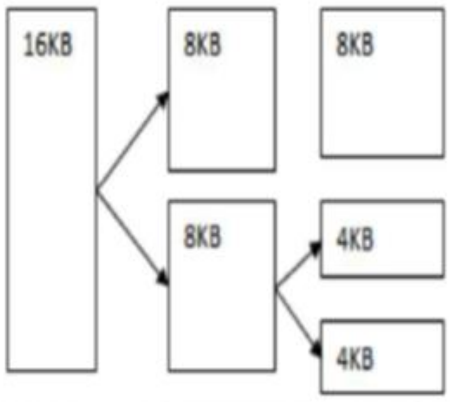


Fig 5.4 Splitting of Memory in Buddy System

Management of Virtual Memory

The Virtual Memory system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required. The Virtual Memory manager maintains two separate views of a process's address space:

Logical View: A logical view describing instructions concerning the layout of the address space. The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.

Physical View: A physical view of each address space which is stored in the hardware page tables for the process.

Virtual memory regions are characterized by:

- The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (demand-zero memory)
- The region's reaction to writes (page sharing or copy-on-write).

The kernel creates a new virtual address space

1. When a process runs a new program with the exec system call

2. Upon creation of a new process by the fork system call.

On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions.

Creating a new process with fork involves creating a complete copy of the existing process's virtual address space.

- The kernel copies the parent process's Virtual Memory Address descriptors, then creates a new set of page tables for the child.
- The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented.
- After the fork, the parent and child share the same physical pages of memory in their address spaces.

Input–Output Management

The Linux device-oriented file system accesses disk storage through two caches:

- Data is cached in the page cache, which is unified with the virtual memory system.
- Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.

Linux splits all devices into three classes:

- block devices allow random access to completely independent, fixed size blocks of data
- character devices include most other devices; they don't need to support the functionality of regular files.
- network devices are interfaced via the kernel's networking subsystem.

Block Devices: Provide the main interface to all disk devices in a system. The block buffer cache serves two main purposes:

- It acts as a pool of buffers for active I/O
- It serves as a cache for completed I/O

The request manager manages the reading and writing of buffer contents to and from a block device driver.

Character Devices: include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access block devices are accessed randomly, while character devices are accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

Network devices: are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem

File System

Linux's file system appears as a hierarchical directory tree obeying UNIX semantics. Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS).

The Linux VFS is designed around object-oriented principles and is composed of two components:

- A set of definitions that define what a file object is allowed to look like
    a. the inode-object and the file-object structures represent individual files
    b. the file system object represents an entire file system
- A layer of software to manipulate those objects.

The Linux ext3 File System

Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file. The main differences between ext2fs and ffs concern their disk allocation policies.

- In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.
- Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.

Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation

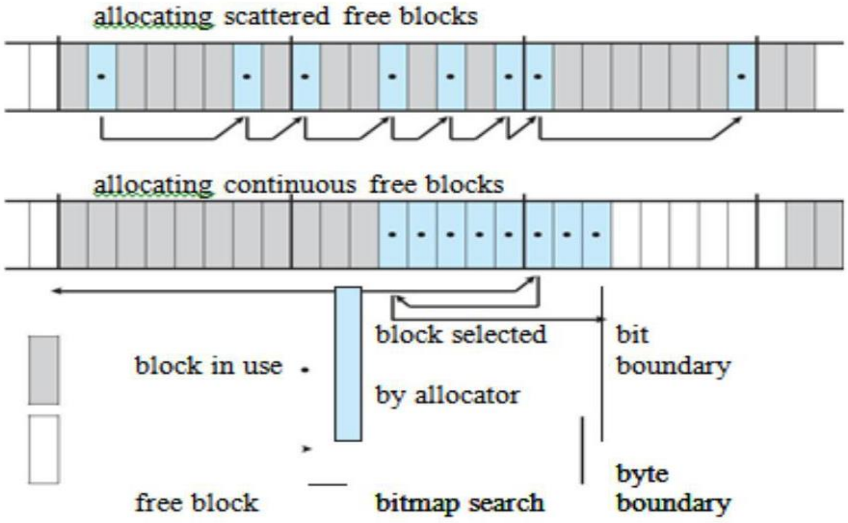


Fig 5.5 ext3 block allocation policies

Inter Process Communication

Linux provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

Like UNIX, Linux informs processes that an event has occurred via signals. There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process.

The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait queue structures.

Passing Data between Processes:

The pipe mechanism allows a child process to inherit a communication channel to its parent. Data written to one end of the pipe can be read at the other. Shared memory offers an extremely fast way of communicating.

Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. To obtain synchronization, however, shared memory must be used in conjunction with another interprocess-communication mechanism.

Mobile OS

Some **important features of Mobile Operating System** are

- Alternate Keyboards.
- Infrared Transmission.
- Touch Control.
- Automation.
- Wireless App Downloads.
- Storage and Battery Swap.
- Custom Home Screens.
- Messaging: SMS, MMS, C2DM (could to device messaging), GCM (Google could messaging)
- Multilanguage support.
- Multi touch.
- Video calling.
- Screen capture.

iOS Architecture

The iOS is the operating system created by Apple Inc. for mobile devices. The iOS is used in many of the mobile devices for apple such as iPhone, iPod, iPad etc. The iOS is used a lot and only lags behind Android in terms of popularity.

The iOS architecture is layered. It contains an intermediate layer between the applications and the hardware, so they do not communicate directly. The lower layers in iOS provide the basic services and the higher layers provide the user interface and sophisticated graphics.

The layered architecture of iOS is given as follows:

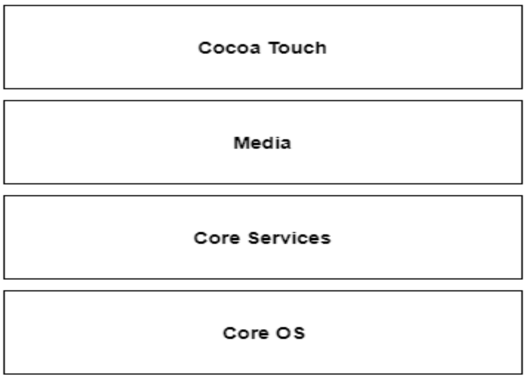


Fig 5.6 iOS Architecture

Layers in iOS Architecture

The different layers as shown in the above diagram are given as follows:

Core OS: All the iOS technologies are build on the low level features provided by the Core OS layer. These technologies include Core Bluetooth Framework, External Accessory Framework, Accelerate Framework, Security Services Framework, Local Authorization Framework etc.

Core Services: There are many frameworks available in the cure services layer. Details about some of these are given as follows:

- **Cloudkit Framework:** The data can be moved between the app the iCloud using the Cloudkit Framework.
- **Core Foundation Framework:** This provides the data management and service features for the iOS apps.
- **Core Data Framework:** The data model of the model view controller app is handled using the Core Data Framework.

- **Address Book Framework:** The address book framework provides access to the contacts database of the user.
- **Core Motion Framework:** All the motion-based data on the device is accessed using core motion framework.
- **Healthkit Framework:** The health-related information of the user can be handled by this new framework.
- **Core Location Framework:** This framework provides the location and heading information to the various apps.
- **Media Services:** The media layer enables all the graphics, audio and video technology of the system. The different frameworks are:
- **UIKit Graphics:** This provides support for designing images and animating the view content.
- **Core Graphics Framework:** This provides support for 2-D vector and image based rendering and is the native drawing engine for iOS apps.
- **Core Animation:** The Core Animation technology optimizes the animation experience of the apps.
- **Media Player Framework:** This framework provides support for playing playlists and enables the user to use their iTunes library.
- **AV Kit:** This provides various easy to use interfaces for video presentation.

Cocoa Touch: The cocoa touch layer provides the following frameworks:

- **EventKit Framework:** This shows the standard system interfaces using view controllers for viewing and changing calendar related events.
- **GameKit Framework:** This provides support for users to share their game related data online using Game center.
- **MapKit Framework:** This provides a scrollable map which can be included into the app user interface.

Android Architecture

Android operating system is a stack of software components which is roughly divided into five sections and four main layers as shown below in the architecture diagram.

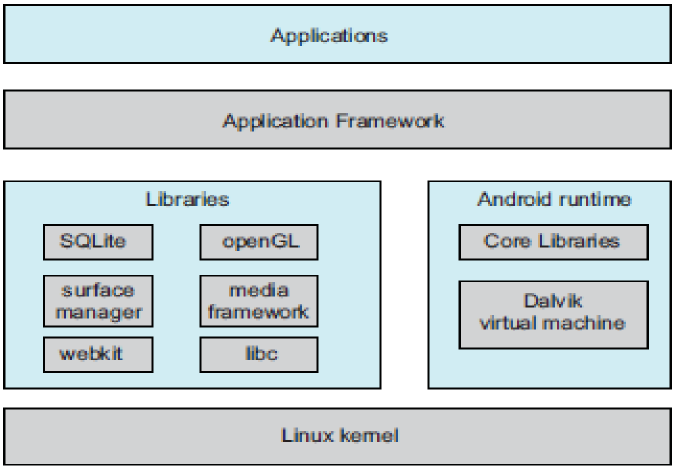


Fig 5.7 Android Architecture

Linux kernel: At the bottom of the layers is Linux kernel. This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc. Also, the kernel handles networking along with interfacing to peripheral hardware.

Libraries: On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

Android Libraries: This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

A summary of some key core Android libraries available to the Android developer is as follows:

- **android.app**: Provides access to the application model and is the cornerstone of all Android applications.
- **android.content**: Facilitates content access, publishing and messaging between applications and application components.
- **android.database** : Used to access data published by content providers and includes SQLite database management classes.
- **android.opengl**: A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os**: Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.text**: Used to render and manipulate text on a device display.
- **android.view**: The fundamental building blocks of application user interfaces.
- **android.widget**: A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit**: A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based core libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called Dalvik Virtual Machine which is a kind of Java Virtual Machine specially designed and optimized for Android.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is fundamental in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.

The Android framework includes the following key services

- **Activity Manager:** Controls all aspects of the application lifecycle and activity stack.
- **Content Providers**: Allows applications to publish and share data with other applications.
- **Resource Manager:** Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- **Notifications Manager**: Allows applications to display alerts and notifications to the user.
- **View System:** An extensible set of views used to create application user interfaces.

Applications

All the Android application are available at the top layer. If we write our application that is to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.