ANDROID ARCHITECTURE

Android architecture contains different number of components to support any android device needs. Android software contains an open-source Linux Kernel having collection of number of C/C++ libraries which are exposed through an application framework services.

Among all the components Linux Kernel provides main functionality of operating system functions to smartphones and Dalvik Virtual Machine (DVM) provide platform for running an android application.

The main components of android architecture are following:-

- Applications
- Application Framework
- Android Runtime
- Platform Libraries
- Linux Kernel

Pictorial representation of android architecture with several main components and their sub components –

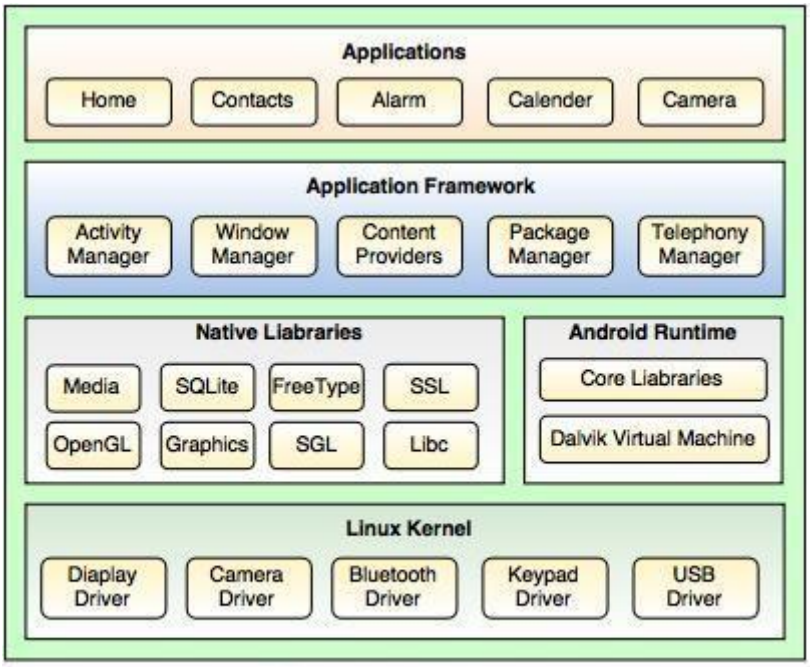


Fig. Android Architecture

Applications –

Applications is the top layer of android architecture. The pre-installed applications like home, contacts, camera, gallery etc and third party applications downloaded from the play store like chat applications, games etc. will be installed on this layer only. It runs within the Android run time with the help of the classes and services provided by the application framework.

Application framework –

Application Framework provides several important classes which are used to create an Android application. It provides a generic abstraction for hardware access and also helps in managing the user interface with application resources. Generally, it provides the services with the help of which we can create a particular class and make that class helpful for the Applications creation.

It includes different types of services activity manager, notification manager, view system, package manager etc. which are helpful for the development of our application according to the prerequisite.

Application runtime –

Android Runtime environment is one of the most important part of Android. It contains components like core libraries and the Dalvik virtual machine(DVM). Mainly, it provides the base for the application framework and powers our application with the help of the core libraries.

Like Java Virtual Machine (JVM), **Dalvik Virtual Machine (DVM)** is a register-based virtual machine and specially designed and optimized for android to ensure that a device can run multiple instances efficiently. It depends on the layer Linux kernel for threading and low-level memory management. The core libraries enable us to implement android applications using the standard JAVA or Kotlin programming languages.

Platform libraries –

The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.

- **Media** library provides support to play and record an audio and video formats.
- **Surface manager** responsible for managing access to the display subsystem.

SGL and **OpenGL** both cross-language, cross-platform application program interface (API) are used for 2D and 3D computer graphics.

SQLite provides database support and **FreeType** provides font support.

Web-Kit This open source web browser engine provides all the functionality to display web content and to simplify page loading.

- **SSL (Secure Sockets Layer)** is security technology to establish an encrypted link between a web server and a web browser.

Linux Kernel –

Linux Kernel is heart of the android architecture. It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.

The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture. It is responsible for management of memory, power, devices etc.

The features of Linux kernel are:

- **Security:** The Linux kernel handles the security between the application and the system.
- **Memory Management:** It efficiently handles the memory management thereby providing the freedom to develop our apps.
- **Process Management:** It manages the process well, allocates resources to processes whenever they need them.
- **Network Stack:** It effectively handles the network communication.
- **Driver Model:** It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

UNIT V –CASE STUDIES

1. LINUX

History

- 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
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code

The Linux Kernel

Version 0.01 was released at May 1991

- No networking
- Ran only on 80386-compatible Intel processors and on PC hardware
- Extremely limited device-drive support

Linux 1.0 (March 1994) included these new features:

- Support UNIX's standard TCP/IP networking protocols
- BSD-compatible socket interface for networking programming
- Device-driver support for running IP over an Ethernet
- Extra hardware support

Version 1.2 (March 1995) was the final PC-only Linux kernel

Linux 2.0

- Version 2.0 was released in June 1996
- Support for multiple architectures, including a fully 64-bit native Alpha port
- Support for multiprocessor architectures
- Improved memory-management code
- Improved TCP/IP performance
- Support for internal kernel threads
- Preemptive kernel

- 64-bit memory support

The Linux System

Linux uses many free tools developed as part of

1. Berkeley's BSD operating system
    - socket interface
    - Networking tools (e.g., traceroute…)
2. MIT's X Window System
3. Free Software Foundation's GNU project

    -bin-utilities, gcc, gnu libc…

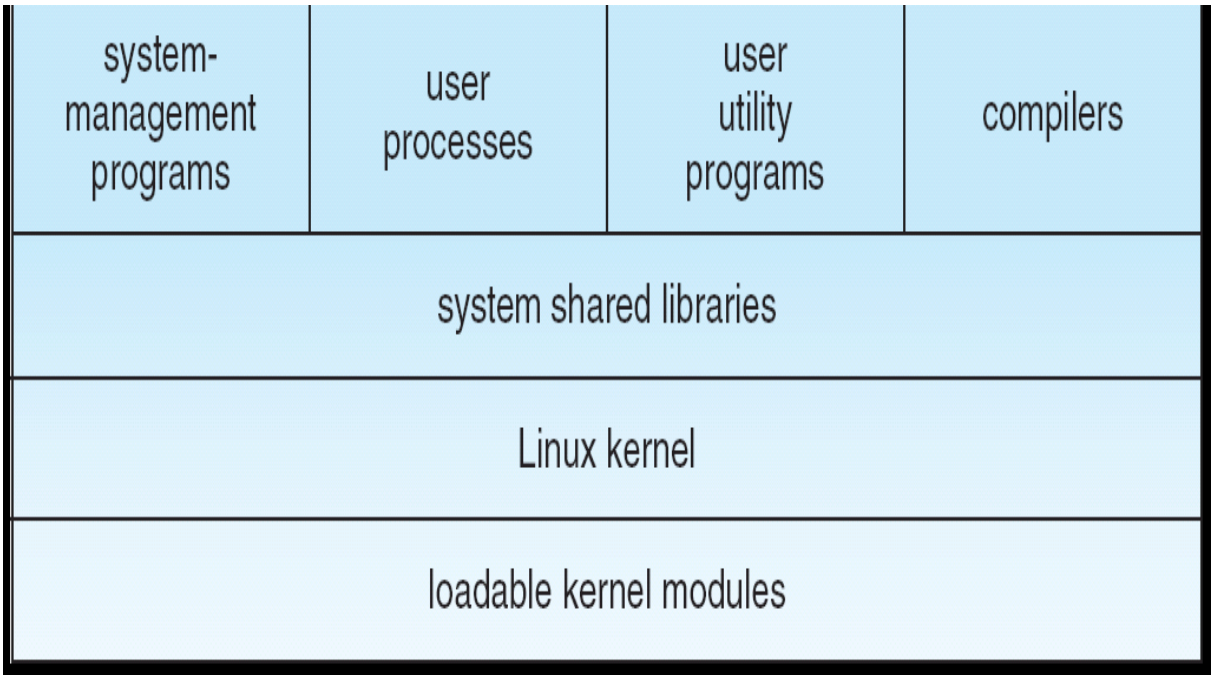
❖ Linux used to developed by individual, now also by big cooperators IBM, Intel, Red hat, Marvell, Microsoft...

❖ Main Linux repository: www.kernel.org

Design Principles

- Linux is a multiuser, multitasking system
- Linux is UNIX compatible
- its file system adheres to traditional UNIX semantics
- it fully implements the standard UNIX networking model
- its API adheres to the SVR4 (System V Release 4, or **SVR4)** UNIX semantics
- it is POSIX-The Portable Operating System Interface (**POSIX**) compliant
- Linux supports a wide variety of architectures
- Main design goals are speed, efficiency, and standardization

Components of a Linux System



1. Kernel.

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

2. 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.
- The most important system library is the **C library**, known as libc. In addition to providing the standard C library, libc implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.

3. 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.

- 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.

Linux Kernel

- The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to run processes, and it provides system services to give arbitrated and protected access to hardware resources.

System libraries

- The system libraries provide many types of functionality. At the simplest level, they allow applications to make system calls to the Linux kernel.
- Making a system call involves transferring control from unprivileged user mode to privileged kernel mode;

System Utilities

- The Linux system includes a wide variety of user-mode programs—both system utilities and user utilities.
- The system utilities include all the programs necessary to initialize and then administer the system, such as those to set up networking interfaces and to add and remove users from the system.

Kernel Modules

- The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand.
- These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run.
- Kernel code that can be compiled, loaded, and unloaded independently, it allows a Linux system to be set up with standard minimal kernel, other components loaded as modules

- A kernel module can implement a device driver, a file system, or a networking
- protocol.
- The module support under Linux has four components:

  1. The **module-management system** allows modules to be loaded into memory and to communicate with the rest of the kernel.

  2. The **module loader and unloader**, which are user-mode utilities, work with the module-management system to load a module into memory.

  3. The **driver-registration system** allows modules to tell the rest of the kernel that a new driver has become available.

  4. A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

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, and 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 autoprobes from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

FILE SYSTEM

A *file system* handles the persistent storage of data files, apps, and the files associated with the operating system itself. Therefore, the file system is one of the fundamental resources used by all processes.

1. IOS FILE SYSTEM

The iOS file system is geared toward apps running on their own. To keep the system simple, users of iOS devices do not have direct access to the file system and apps are expected to follow this convention.

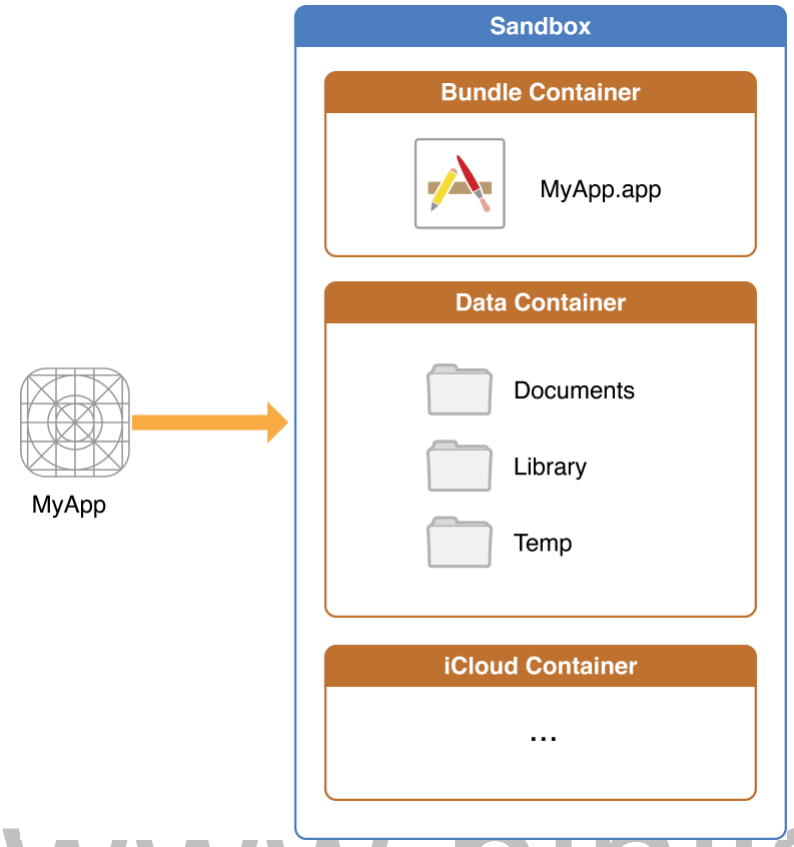
iOS Standard Directories: Where Files Reside

For security purposes, an iOS app's interactions with the file system are limited to the directories inside the app's sandbox directory. During installation of a new app, the installer creates a number of container directories for the app inside the sandbox directory. Each container directory has a specific role.

The bundle container directory holds the app's bundle, whereas the data container directory holds data for both the app and the user. The data container directory is further divided into a number of subdirectories that the app can use to sort and organize its data. The app may also request access to additional container directories—for example, the iCloud container—at runtime.

These container directories constitute the app's primary view of the file system. Figure shows a representation of the sandbox directory for an app.

Figure An iOS app operating within its own sandbox directory



The following Table lists some of the more important subdirectories inside the sandbox directory and describes their intended usage. This table also describes any additional access restrictions for each subdirectory and points out whether the directory's contents are backed up by iTunes and iCloud.

Table 1-1 Commonly used directories of an iOS app

Directory	Description
AppName.app	This is the app's bundle. This directory contains the app and all of its resources. You cannot write to this directory. To prevent tampering, the bundle directory is signed at installation time. Writing to this directory changes the signature and prevents your app from launching. You can, however, gain read-only access to any resources stored in the apps bundle. For more information, see the *Resource Programming Guide*

	The contents of this directory are not backed up by iTunes or iCloud. However, iTunes does perform an initial sync of any apps purchased from the App Store.
Documents/	Use this directory to store user-generated content. The contents of this directory can be made available to the user through file sharing; therefore, this directory should only contain files that you may wish to expose to the user. The contents of this directory are backed up by iTunes and iCloud.
Documents/Inbox	Use this directory to access files that your app was asked to open by outside entities. Specifically, the Mail program places email attachments associated with your app in this directory. Document interaction controllers may also place files in it. Your app can read and delete files in this directory but cannot create new files or write to existing files. If the user tries to edit a file in this directory, your app must silently move it out of the directory before making any changes. The contents of this directory are backed up by iTunes and iCloud.
Library/	This is the top-level directory for any files that are not user data files. You typically put files in one of several standard subdirectories. iOS apps commonly use the Application Support and Caches subdirectories; however, you can create custom subdirectories. Use the Library subdirectories for any files you don't want exposed to the user. Your app should not use these directories for user data files. The contents of the Library directory (with the exception of the Caches subdirectory) are backed up by iTunes and iCloud.

	For additional information about the Library directory and its commonly used subdirectories, see The Library Directory Stores App-Specific Files.
tmp/	Use this directory to write temporary files that do not need to persist between launches of your app. Your app should remove files from this directory when they are no longer needed; however, the system may purge this directory when your app is not running. The contents of this directory are not backed up by iTunes or iCloud.

An iOS app may create additional directories in the Documents, Library, and tmp directories. You might do this to better organize the files in those locations

2. ANDROID FILE SYSTEM.

(i) Flash Memory Android File System

1. exFAT

Originally created by Microsoft for flash memory, the exFAT file system is not a part of the standard Linux kernel. However, it still provides support for Android devices in some cases. It stands for Extended File Allocation Table.

2.F2FS

Users of Samsung smartphones are bound to have come across this type of file system if they have been using the smartphone for a while. F2FS stands for Flash-Friendly File System, which is an Open Source Linux file system. This was introduced by Samsung 4 years ago, in 2012.

3. JFFS2

It stands for the Journal Flash File System version 2. This is the default flash file system for the Android Open Source Project kernels. This version of Android File System has been around since the Android Ice Cream Sandwich OS was released. JFFS2 has since replaced the JFFS.

4. YAFFS2

It stands for Yet Another Flash File System version 2. Funny as the name might sound like, it is actually a serious business! It has not been a part of the AOSP for a while now and is rarely found in Android smartphones. However, it does tend to make a few appearances every now and then.

(ii) Media-based Android File System

1. EXT2/EXT3/EXT4

Ext, which stands for the EXTended file systems, are the standards for the Linux file system. The latest out of these is the EXT4, which has now been replacing the YAFFS2 and the JFFS2 file systems on Android smartphones.

2. MSDOS

Microsoft Disk Operating System is known to be one of the oldest names in the world of Operating Systems, and it helps FAT 12, FAT 16 and FAT 32 file systems to run.

3. VFAT

An extension to the aforementioned FAT 12, FAT 16 and FAT 32 file systems, the VFAT is a kernel module seen alongside the MSDOS module. External SD cards that help expand the storage space are formatted using VFAT.

(iii) Pseudo File Systems

1. CGroup

Cgroup stands for Control Group. It is a pseudo file system which allows access and meaning to various kernel parameters. Cgroups are very important for the Android File System as the Android OS makes use of these control groups for user accounting and CPU Control.

2. Rootfs

Rootfs acts as the mount point, and it is a minimal file system. It is located at the mount point "/".

3. Procfs

Usually found mounted at the /proc directory. The procfs file system has files which showcase the live kernel data. Sometimes this file system also reflects a number of kernel

data structures. These number directories are reflective of the process IDs for all the currently running tasks.

4. Sysfs

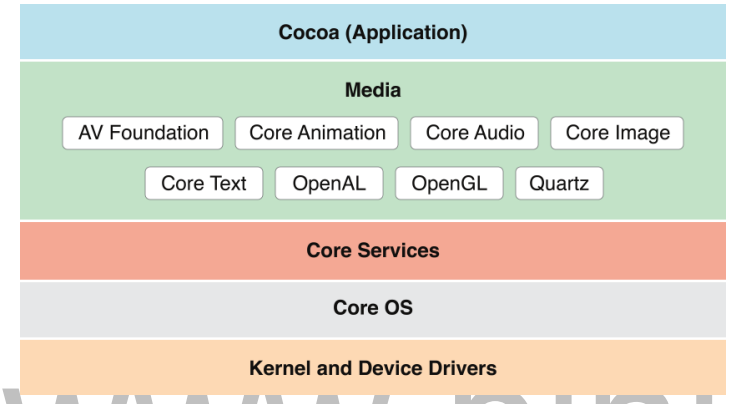
Usually mounted on the /sys directory. The sysfs file system helps the kernel identify the devices. Upon identifying a new device, the kernel builds an object in sys/module/ directory. There are various other elements stored inside the /sys/ folder which helps the kernel communicate with various Android File Systems.

5. Tmpfs

A temporary file system, tmpfs is usually mounted on /dev directory. Data on this is lost when the device is rebooted.

IOS - MEDIA LAYER

Beautiful graphics and high-fidelity multimedia are hallmarks of the OS X user experience. Take advantage of the technologies of the Media layer to incorporate 2D and 3D graphics, animations, image effects, and professional-grade audio and video functionality into your app.



Supported Media Formats

OS X supports more than 100 media types, covering a range of audio, video, image, and streaming formats. Table 3-1 lists some of the more common supported file formats.

Table 3-1 Partial list of formats supported in OS X

Image formats	PICT, BMP, GIF, JPEG, TIFF, PNG, DIB, ICO, EPS, PDF
Audio file and data formats	AAC, AIFF, WAVE, uLaw, AC3, MPEG-3, MPEG-4 (.mp4, .m4a), .snd, .au, .caf, Adaptive multi-rate (.amr)
Video file formats	AVI, AVR, DV, M-JPEG, MPEG-1, MPEG-2, MPEG-4, AAC, OpenDML, 3GPP, 3GPP2, AMC, H.264, iTunes (.m4v), QuickTime (.mov, .qt)
Web streaming protocols	HTTP, RTP, RTSP

Graphics Technologies

A distinctive quality of any OS X app is high-quality graphics in its user interface. And on a Retina display, users are more aware than ever of your app's graphics.

The simplest, most efficient, and most common way to ensure high-quality graphics in your app is to use the standard views and controls of the AppKit framework, along with pre-rendered images in different resolutions. In this way, you let the system do the work of rendering the app's UI appropriately for the current display.

Occasionally, you might need to go beyond off-the-shelf views and simple graphics. In these situations, you can take advantage of the powerful OS X graphics technologies. The following sections describe some of these technologies; for summaries of all technologies see Media Layer Frameworks.

Graphics and Drawing

OS X offers several system technologies for graphics and drawing. Many of these technologies provide support for making your rendered content look good at different screen resolutions. To learn how to make sure that your app looks good on a high-resolution display, see *High Resolution Guidelines for OS X*.

Cocoa Drawing

It supports drawing in both standard and custom color spaces and it supports content manipulations using graphics transforms. Drawing calls made from Cocoa are composited along with all other Quartz 2D content. You can even mix Quartz 2D drawing calls (and drawing calls from other system graphics technologies) with Cocoa calls in your code.

The AppKit framework is described in AppKit. For more information on how to draw using Cocoa features, see *Cocoa Drawing Guide*.

Metal

Metal provides the lowest-overhead access to the GPU, enabling you to maximize the

graphics and compute potential of your app. With a streamlined API, precompiled shaders, and support for efficient multi-threading, Metal can take your game or graphics app to the next level of performance and capability.

The Metal framework is described in *Metal Programming Guide*, *Metal Shading Language Guide*, and the associated references.

Other Frameworks for Graphics and Drawing

In addition to AppKit (specifically, its Cocoa drawing interface), there are several other important frameworks for graphics and drawing. By design, Cocoa drawing integrates well with the other graphics and drawing technologies listed next.

Core Graphics (CoreGraphics.framework). Core Graphics (also known as *Quartz 2D*) offers native 2D vector- and image-based rendering capabilities that are resolution- and

device-independent. These capabilities include path-based drawing, painting with transparency, shading, drawing of shadows, transparency layers, color management, antialiased rendering, and PDF document generation. The Core Graphics framework is in the Application Services umbrella framework.

- **Core Animation.** Core Animation enables your app to create fluid animations using advanced compositing effects. It defines a hierarchical view-like abstraction that mirrors a hierarchy of views and is used to perform complex animations of user interfaces. Core Animation is implemented by the Quartz Core framework (QuartzCore.framework) (for more information, see Core Animation).

- **SpriteKit** (SpriteKit.framework). SpriteKit provides the tools and methods for creating and rendering and animating textured images, or sprites. You use graphical editors for creating sprites, and then use those sprites in scenes that simulate game physics. In addition to sprites, you can add lights, emitters, and different kinds of fields to scenes. SpriteKit animates your scene and calls back to your code for events such as collisions. To learn more, see Sprite Kit.

- **Scene Kit** (SceneKit.framework). Scene Kit provides a high-level, Objective-C graphics API that you can use to efficiently load, manipulate, and render 3D scenes. Powerful and easy-to-use Scene Kit integrates well with Core Animation and SpriteKit, allowing you to use built-in materials or custom GLSL shaders to render your 3D scenes (for more information, see Scene Kit).

- **Metal** *Metal.framework* provides extremely low-overhead access to the capabilities of modern GPUs and enables high-performance 2D and 3D graphics, and parallel

computational tasks. A more flexible and efficient alternative to OpenGL and OpenCL, Metal is intended for use by games and graphics-intensive applications that require fine-grained control over the GPU. To learn more about Metal, see *Metal Programming Guide* and *Metal Shading Language Guide*

- MetalKit MetalKit.framework provides libraries of commonly needed functions and classes to reduce the overall time for developing a Metal application. For more information, see *MetalKit Framework Reference* and *MetalKit Functions Reference*.

- **OpenGL** (OpenGL.framework). OpenGL is an open, standards-based technology for creating and animating real-time 2D and 3D graphics. It is primarily used for games and other apps with real-time rendering needs. To learn more about OpenGL in OS X, see OpenGL.

- **GLKit** (GLKit.framework). GLKit provides libraries of commonly needed functions and classes that reduce the effort required to create shader-based apps or to port existing apps that rely on fixed-function vertex or fragment processing provided by earlier versions of OpenGL ES or OpenGL. To learn more about the GLKit framework, see GLKit.

Text, Typography, and Fonts

OS X provides extensive support for advanced typography for Cocoa apps. With this support, your app can control the fonts, layout, typesetting, text input, and text storage when managing the display and editing of text. For the most basic text requirements, you can use the text fields, text views, and other text-displaying objects provided by the AppKit framework.

There are two technologies to draw upon for more sophisticated text, font, and typography needs: the Cocoa text system and the Core Text API.

- **Cocoa text system.** AppKit provides a collection of classes, known as the *Cocoa text system*, that together provide a complete set of high-quality typographical services.
- **Core Text** (CoreText.framework). The Core Text framework contains low-level interfaces for laying out Unicode text and handling Unicode fonts.

Images

Both AppKit and Quartz let you create objects that represent images (NSImage and CGImageRef) from various sources, draw these images to an appropriate graphics context, and even composite one image over another according to a given blending mode. Beyond the native capabilities of AppKit and Core Graphics, you can do other things with images using the following frameworks:

- **Image Capture Core** (ImageCaptureCore.framework). The Image Capture Core framework enables your app to browse locally connected or networked scanners and cameras and images.
- **Core Image.** Core Image is an image processing technology that allows developers to process images with system-provided image filters, create custom image filters, and detect features in an image.
- **Image Kit.** Image Kit is built on top of the Image Capture Core framework.
- **Image I/O** (ImageIO.framework). The Image I/O framework helps you read image data from a source and write image data to a destination. Sources and destinations can be URLs, Core Foundation data objects, and Quartz data consumers and data providers.

Color Management

ColorSync is the color management system for OS X. It provides essential services for fast, consistent, and accurate color reproduction, proofing, and calibration. It also provides an interface for accessing and managing systemwide settings for color devices such as displays, printers, cameras, and scanners.

Printing

OS X implements printing support using a collection of APIs and system services that are available to all app environments.

Table 3-2 Features of the OS X printing system

Feature	Description

AirPrint	Users can print to an AirPrint-enabled printer on their network without having to use a third-party driver.
CUPS	Common UNIX Printing System (CUPS), an open source architecture used to handle print spooling and other low-level features, provides the underlying support for printing.
Desktop printers	Desktop printers offer users a way to manage printing and print jobs from the Dock or desktop.
Fax support	Fax support means that users can fax documents directly from the Print dialog.
Native PDF support	PDF as a native data type lets any app easily save textual and graphical data to the device-independent PDF format.
PostScript support	PostScript support allows apps to use legacy third-party drivers to print to PostScript Level 2–compatible and Level 3–compatible printers and to convert PostScript files directly to PDF.
Print preview	The print preview capability lets users see documents through a PDF viewer app prior to printing.
Printer discovery	Printer discovery enables users to detect, configure, and add to printer lists those printers that implement Bluetooth or Bonjour.
Raster printers (support for)	This support allows apps to print to raster printers using legacy third- party drivers.
Speedy spooling	Speedy spooling enables apps that use PDF to submit PDF files directly to the printing system instead of spooling individual pages.

Audio Technologies

OS X includes support for high-quality audio recording, synthesis, manipulation, and playback. The frameworks in the following list are ordered from high level to low level, with the AV Foundation framework offering the highest-level interfaces you can use. When choosing an audio technology, remember that higher-level frameworks are easier to use and, for this reason, are usually preferred. Lower-level frameworks offer more flexibility and control but require you to do more work.

- **AV Foundation** (AVFoundation.framework). AV Foundation supports audio playback, editing, analysis, and recording.
- **OpenAL** (OpenAL.framework). OpenAL implements a cross-platform standard API for 3D audio.
- **Core Audio** (CoreAudio.framework). Core Audio consists of a set of frameworks that provide audio services that support recording, playback, synchronization, signal processing, format conversion, synthesis, and surround sound.

Video Technologies

When choosing a video technology, remember that the higher-level frameworks simplify your work and are, for this reason, usually preferred.

The frameworks in the following list are ordered from highest to lowest level, with the AV Foundation framework offering the highest-level interfaces you can use.

- **AVKit** (AVKit.framework). AV Kit supports playing visual content in your application using the standard controls.
- **AV Foundation** (AVFoundation.framework). AV Foundation supports playing, recording, reading, encoding, writing, and editing audiovisual media.
- **Core Media** (CoreMedia.framework). Core Media provides a low-level C interface for managing audiovisual media. With the Core Media I/O framework, you can create plug-ins that can access media hardware and that can capture video and mixed audio and video streams.
- **Core Video** (CoreVideo.framework). Core Video provides a pipeline model for digital video between a client and the GPU to deliver hardware-accelerated video processing while allowing access to individual frames.

LINUX - FILE SYSTEMS

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of files by hiding the implemen-tation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system — ext3.

The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file. A **file object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

int open(. . .) — Open a file.

ssize t read(. . .) — Read from a file. ssize t write(. . .) — Write to a file.

int mmap(. . .) — Memory-map a file.

The complete definition of the file object is specified in the struct file operations, which is located in the file /usr/include/linux/fs.h. An implementation of the file object

(for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's read() operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file

A process cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O.

It also remembers the permissions (for example, read or write) requested when the file was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a single inode object. Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique file-system/inode number pair, and it

finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio.h). For example, the file /usr/include/stdio.h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio.h. Each of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situ-ation in which a process wishes to open the file with the pathname /usr/include/stdio.h using an editor.

Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root — /. The operating system must then read through this file to obtain the inode for the file include. It must continue this process until it obtains the inode for the file stdio.h. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

The Linux ext3 File System

The standard on-disk file system used by Linux is called **ext3**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB.

The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs)**. A later redesign to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2)**. Further development added journaling capabilities, and the system was renamed the **third extended file system (ext3)**.

Linux kernel developers are working on augmenting ext3 with modern file-system features such as extents. This new file system is called the **fourth extended file system (ext4)**. The rest of this section discusses ext3, however, since it remains the most-deployed Linux file system. Most of the discussion applies equally to ext4.Linux's ext3 has much in common with the BSD Fast File System (FFS). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection.

As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries. In turn, each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext3 and FFS lie in their disk-allocation policies. In FFS,

the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext3 does not use fragments at all but performs all its allocations in smaller units. The default block size on ext3 varies as a function of the total size of the file system. Supported block sizes are
1, 2, 4, and 8 KB.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware.

A block-sized I/O request size is too small to maintain good performance, so ext3 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.

The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. (Note that modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.)

When allocating a file, ext3 must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides for nondirectory files. Directory files are not kept together but

rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext3 tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group. When extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext3 from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found.

Once the next block to be allocated has been found by either bit or byte search, ext3 extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 5.7 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be.

The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks. Furthermore, allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it. Thus, before allocating, we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

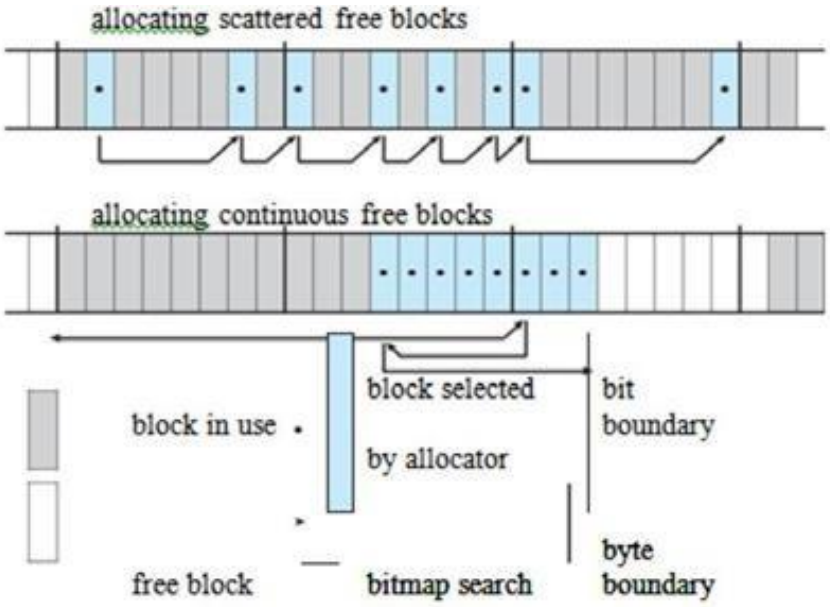


Figure 5.7 ext3 block-allocation policies.

Journaling

The ext3 file system supports a popular feature called **journaling**, whereby modifications to the file system are written sequentially to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed.

Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read – write heads, thereby decreasing head contention and seek times.

If the system crashes, some transactions may remain in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system recovers. The

transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted — that is, it was not committed before the system crashed.
Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems may perform some operations faster than non-journaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O.

Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion. Due to this performance improvement, ext3 can be configured to journal only metadata and not file data.

The Linux Process File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather provides an interface to some other functionality. The Linux **process file system**, known as the /proc file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A /proc file system is not unique to Linux. SVR4 UNIX introduced a /proc file system as an efficient interface to the kernel's process debugging support. Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process

on the current system. A listing of the file system reveals one directory per process, with the directory name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a /proc file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The /proc file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files.

For example, in the past, the traditional UNIX ps command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from /proc.

The /proc file system must implement two things: a directory structure and the file contents within. Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the /proc file system must define a unique and persistent inode number for each directory and the associated files.

Once such a mapping exists, the file system can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory inode. When data are read from one of these files, the /proc file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits in size, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global rather than process-specific information. Separate global files exist in /proc to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. Users can open an access channel to a device in the same way they open any other file — devices can appear as objects within the file system.

The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 5.8 illustrates the overall structure of the device-driver system.

Block devices

Include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. For example, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

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

They 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.

Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must provide functionality to

ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a block represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a buffer. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list.

It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the lists.

Linux kernel version 2.6 introduced a new I/O scheduling algorithm. Although a simple elevator algorithm remains available, the default I/O scheduler is now the **Completely Fair Queueing (CFQ)** scheduler. The CFQ I/O scheduler is fundamentally different from elevator-based algorithms. Instead of sorting requests into a list, CFQ maintains a set of lists by default, one for each process. Requests originating from a process go in that process's list. For example, if two processes are issuing I/O requests, CFQ will maintain two separate lists of requests, one for each process. The lists are maintained according to the C-SCAN algorithm.

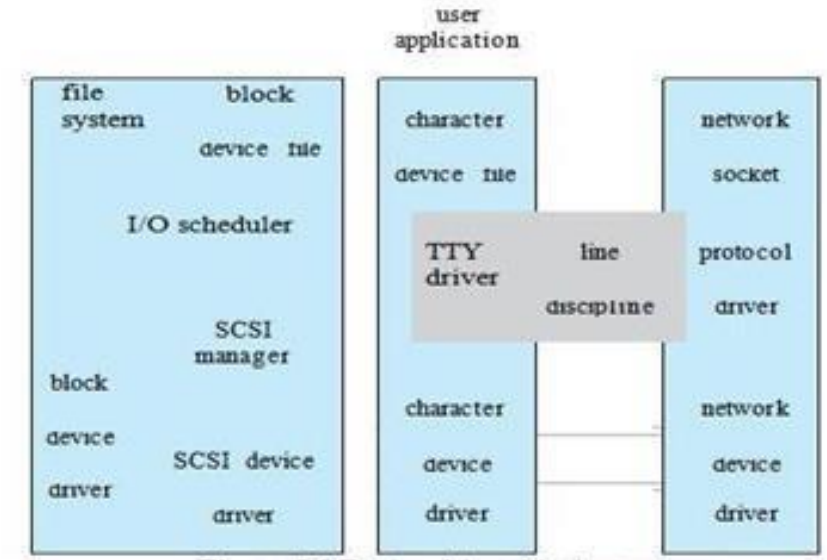


Figure 5.8 Device-driver block structure.

Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of tty struct structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the tty discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal.

This job is complicated by the fact that several such processes may be running simultaneously, and the tty line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line.

These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

LINUX - 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: the process's identity, environment, and context

Process Identity

Process ID (PID) - The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process

Credentials - Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files

Personality - Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls

Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX

Namespace – Specific view of file system hierarchy

Most processes share common namespace and operate on a shared file-system hierarchy

But each can have unique file-system hierarchy with its own root directory and set of mounted file systems

Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:

The **argument vector** lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself.

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

- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software.
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

Process Context

- The (constantly changing) state of a running program at any point in time
- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process
- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far
- The **file table** is an array of pointers to kernel file structures
  - When making file I/O system calls, processes refer to files by their index into this table, the **file descriptor** (**fd**)
- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files
  - The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive
- The **virtual-memory context** of a process describes the full contents of the its private address space

Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent

Both are called ***tasks*** by Linux

A distinction is only made when a new thread is created by the clone() system call

- fork() creates a new task with its own entirely new task context
- clone() creates a new task with its own identity, but that is allowed to share the data structures of its parent

  - Using clone() gives an application fine-grained control over exactly what is shared between two threads

flag	meaning
FS	File-system information is shared.
VM	The same memory space is
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is

- **LINUX -MEMORY MANAGEMENT**
- Memory management under Linux has two components.
- The first deals with allocating and freeing physical memory ‘s, groups of s, and small blocks of RAM.
- The second handles virtual memory, which is memory- mapped into the address space of running processes.

In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process’s virtual memory in response to an exec() system call.

Management of Physical Memory

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

ZONE DMA

ZONE DMA32

ZONE NORMAL

ZONE HIGHMEM

These zones are architecture specific.

For example, on the Intel x86-32 architec-ture, certain ISA (industry standard architecture) devices can only access the lower 16 MB of physical memory using DMA.

On these systems, the first 16 MB of physical memory comprise ZONE DMA. On other systems, certain devices can only access the first 4 GB of physical memory, despite supporting 64-bit addresses. On such systems, the first 4 GB of physical memory comprise ZONE DMA32. ZONE HIGHMEM (for “high memory”) refers to physical memory that is not mapped into the kernel address space.

For example, on the 32-bit Intel architecture (where 2^{32} provides a 4-GB address space), the kernel is mapped into the first 896-MB of the address space; the remaining memory is referred to as **high memory** and is allocated from ZONE HIGHMEM. Finally, ZONE NORMAL comprises everything else — the normal, regularly mapped s.

Whether an architecture has a given zone depends on its constraints. A modern, 64-bit architecture such as Intel x86-64 has a small 16 MB ZONE DMA (for legacy devices) and all the rest of its memory in ZONE NORMAL, with no “high memory”.

The relationship of zones and physical addresses on the Intel x86-32 architecture is shown in Figure 5.3. The kernel maintains a list of free s for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

The primary physical-memory manager in the Linux kernel is the **allocator**. Each zone has its own allocator, which is responsible for allocating and freeing all physical s for the zone and is capable of allocating ranges of physically contiguous s on request.

The allocator uses a buddy system (Section 9.8.1) to keep track of available physical s. 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 . Figure 5.3 shows an example of buddy-heap allocation.

A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

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

Figure 5.3 Relationship of zones and physical addresses in Intel x86-32.

Ultimately, 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 allocator. However, kernel functions do not have to use the basic allocator to reserve memory.

Several specialized memory-management subsystems use the underlying allocator to man-age their own pools of memory. 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 cache, used for caching s belonging to files.

Many components of the Linux operating system need to allocate entire s on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes.

Analogous to the C language's malloc() function, this kmalloc() service allocates entire physical s on demand but then splits them into smaller pieces. The kernel maintains lists of s in use by the kmalloc() service. Allocating memory involves determining the appropriate list and either taking the first free piece available on the list or allocating a new and splitting it up.

Memory regions claimed by the kmalloc() system are allocated permanently until they are freed explicitly with a corresponding call to kfree(); the kmalloc() system cannot reallocate or reclaim these regions in response to memory shortages.

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 s.

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.

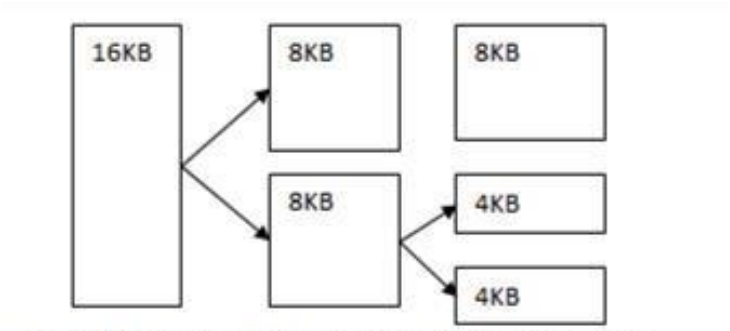


Fig: 5.4 Splitting of memory in the buddy system

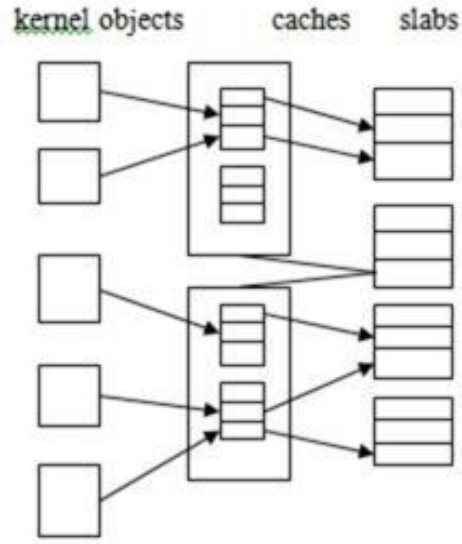


Fig 5.5 Slab allocation in Linux

Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing inodes stores instances of inode structures, and the cache representing process descriptors stores instances of process descriptor structures.

The relationship among slabs, caches, and objects is shown in Figure 5.5. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

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.

For example, a 12-KB slab (made up of three contiguous 4-KB s) could store six 2-KB objects. Initially, all the objects in the cache are marked as free. When a new object for a

kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct task struct, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.

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 s and assigned to a cache; memory for the object is allocated from this slab.

Two other main subsystems in Linux do their own management of physical s: the cache and the virtual memory system. These systems are closely related to each other. The **cache** is the kernel's main cache for files and is the main mechanism through which I/O to block devices is performed.

File systems of all types, including the native Linux disk- based file systems and the NFS networked file system, perform their I/O through the cache. The cache stores entire s of file contents and is not limited to block devices.

It can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with each other because reading a of data into the cache requires mapping s in the cache using the virtual memory system.

Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space accessible to each process. It creates s of virtual memory on demand and manages loading those s from disk and swapping them back out to disk as required.

Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of s. The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space.

In this view, the address space consists of a set of non overlapping regions, each region representing a continuous, -aligned subset of the address space.

Each region is described internally by a single vm area struct structure that defines the properties of the region, including the process's read, write, and execute permissions in the region as well as information about any files associated with the region.

The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware tables for the process. The -table entries identify the exact current location of each of virtual memory, whether it is on disk or in physical memory.

The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a that is not currently present in the tables.

Each vm area struct in the address-space description contains a field pointing to a table of functions that implement the key - management functionality for any given virtual memory region. All requests to read or write an unavailable are eventually dispatched to the appropriate handler in the function table for the vm area struct, so that the central memory- management routines do not have to know the details of managing each possible type of memory region.

Virtual Memory Regions

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the backing store for the region, which describes where the s for the region come from.

Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory region. Such a region represents **demand-**

zero memory: when a process tries to read a in such a region, it is simply given back a of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file. Whenever the process tries to access a within that region, the table is filled with the address of a within the kernel's cache corresponding to the appropriate offset in the file.

The same of physical memory is used by both the cache and the process's tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space.

Any number of processes can map the same region of the same file, and they will all end up using the same of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either ***private*** or ***shared***. If a process writes to a privately mapped region, then the r detects that a copy-on-write is necessary to keep the changes local to the process.

In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

Lifetime of a Virtual Address Space

The kernel creates a new virtual address space in two situations: when a process runs a new program with the exec() system call and when a new process is created by the fork() system call. The first case is easy. When a new program is executed, the process is given a new, completely empty virtual address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with fork(), involves creating a complete copy of the ēxistiṉg process's virtual address space. The kernel copies the parent process's vm area struct descriptors, then creates a new set of tables for the child.

The parent's tables are copied directly into the child's, and the reference count of each covered is incremented. Thus, after the fork, the parent and child share the same physical s of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any s to which the parent process has written within such a

region are private, and subsequent changes to these s by either the parent or the child must not update the in the other process's address space.

When the - table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these s, the two processes share the same of physical memory.

However, if either process tries to modify a copy-on-write, the reference count on the is checked. If the is still shared, then the process copies the 's contents to a brand-new of physical memory and uses its copy instead.

This mechanism ensures that private data s are shared between processes whenever possible and copies are made only when absolutely necessary.

Swapping and Paging

An important task for a virtual memory system is to relocate s of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging — the movement of individual s of virtual memory between physical memory and disk.

Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which s to write out to disk and when to write them.

Second, the **paging mechanism** carries out the transfer and s data back into physical memory when they are needed again.

Linux's **out policy** uses a modified version of the standard clock (or second-chance) algorithm. Under Linux, a multiple-pass clock is used, and every has an ***age*** that is adjusted on each pass of the clock.

The age is more precisely a measure of the 's youthfulness, or how much activity the has seen recently. Frequently accessed s will attain a higher age value, but the age of infrequently accessed s will drop toward zero with each pass. This age valuing allows the r to select s to out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system.

Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out s to continuous runs of disk blocks for improved performance.

The allocator records the fact that a has been d out to disk by using a feature of the tables on modern processors: the -table entry's -not-present bit is set, allowing the rest of the -table entry to be filled with an index identifying where the has been written.

Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The -table entries that map to these kernel s are marked as protected, so that the s are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions.

The first is a static area that contains -table references to every available physical of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all s allocated by the normal allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. - table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow kernel code to use this virtual memory.

The vmalloc() function allocates an arbitrary number of physical s of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The vremap() function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the exec() system call. This exec() call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program.

The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an exec() system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed.

Older Linux kernels understood the a.out format for binary files — a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF** format, now supported by most current UNIX implementations. ELF has a number of advantages over a.out, including flexibility and extendability. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and a.out binary formats in a single running system.

Mapping of Programs into Memory

Under Linux, the binary loader does not load a binary file into physical memory. Rather, the s of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given will a fault result in the loading of that into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several -aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

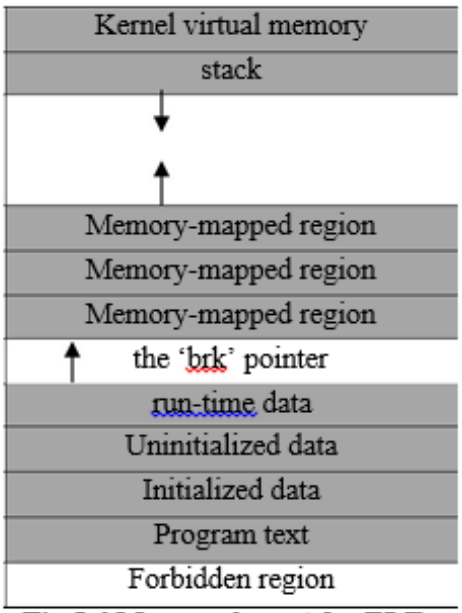


Fig 5.6 Memory layout for ELF programs

Figure 5.6 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in its own privileged region of virtual memory inaccessible to normal user-mode programs.

The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the arguments and environment variables given to the program in the exec() system call. The other regions are created near the bottom end of virtual memory.

The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, brk,

that points to the current extent of this data region, and processes can extend or contract their brk region with a single system call— sbrk().

Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions must also be loaded.

In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows that to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains.

The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: they are compiled into **position-independent code (PIC)**, which can run at any address in memory.

LINUX- 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 While scheduling is normally thought of as the running and interrupting of processes, Linux, scheduling also includes the running of the various kernel tasks
    - Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver

Process Scheduling

- Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes. The other is designed for real-time tasks, where absolute priorities are more important than fairness.
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as *O*(1)
    - Real-time range 0 to 99
    - nice value 20 to 19
    - Had challenges with interactive performance
- Version 2.6 introduced **Completely Fair Scheduler** (**CFS**)

CFS

- Eliminates traditional, common idea of time slice
- Instead all tasks allocated portion of processor's time
- CFS calculates how long a process should run as a function of total number of tasks
- *N* runnable tasks means each gets 1/*N* of processor's time
- Then weights each task with its nice value
    - Smaller nice value -> higher weight (higher priority)
- CFS then runs each process for a "time slice" proportional to the process's weight divided by the total weight of all runnable processes.

The **time slice** is the length of time — the ***slice*** of the processor that a process is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes, respectively.

A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ.

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.

To calculate the actual length of time a process runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each process receives a proportion of the processor's time. How long that allotment is depends on how many other processes are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority- based scheduling algorithms.

Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing processes. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first- Served (FCFS) and round-robin.

In both cases, each process has a priority in addition to its scheduling class. The scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.

Linux's real-time scheduling is soft rather than hard real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a process becomes runnable and when it actually runs.

Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways.

A running program may request an operating- system service, either explicitly via a system call or implicitly for example, when a fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption.

This fact relates to the idea of critical sections portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel

synchronization involves much more than just process scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a non preemptive kernel, meaning that a process running in kernel mode could not be preempted even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel

The Linux kernel provides spinlocks and semaphores (as well as reader – writer versions of these two locks) for locking in the kernel.

On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption.

That is, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

Single processor	Multiple processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

- Linux uses an interesting approach to disable and enable kernel pre-emption.

It provides two simple kernel interfaces preempt disable() and preempt enable(). In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock.

To enforce this rule, each task in the system has a thread-info structure that includes the field preempt count, which is a counter indicating the number of locks being held by the task.

The counter is incremented when a lock is acquired and decremented when a lock is released.

If the value of preempt count for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to preempt disable().

Spinlocks along with the enabling and disabling of kernel preemption are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades.

To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half.

The ***top half*** is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run.

The ***bottom half*** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred

until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system.

Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it.

At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

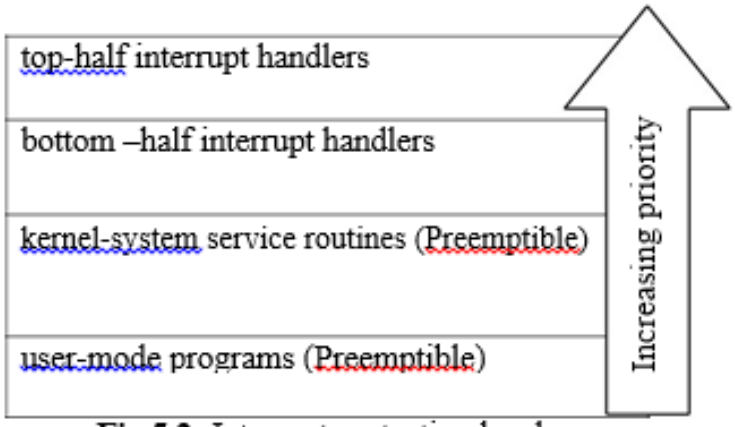


Fig 5.2: Interrupt protection levels

Figure 5.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for “big kernel lock”) was created to allow multiple processes (running on different processors) to be active in the kernel concurrently.

However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and processes. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel’s data structures.

Such spinlocks are described in Section 18.5.3. The 3.0 kernel provides additional SMP enhancements, including ever- finer locking, processor affinity, and load-balancing algorithms.

MOBILE OS - IOS AND ANDROID

In The Anatomy of an iPhone 4 we looked at the hardware that is contained within an iPhone 4 device. When we develop apps for the iPhone Apple does not allow us direct access to any of this hardware. In fact, all hardware interaction takes place exclusively through a number of different layers of software that act as intermediaries between the application code and device hardware. These layers make up what is known as an operating system. In the case of the iPhone, this operating system is known as iOS.

In order to gain a better understanding of the iPhone development environment, this chapter will look in detail at the different layers that comprise the iOS operating system and the frameworks that allow us, as developers, to write iPhone applications.

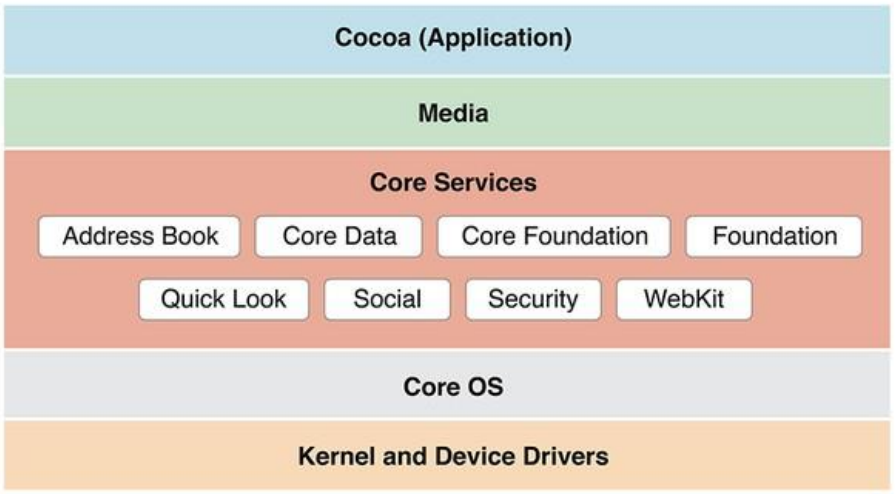
IPHONE OS BECOMES IOS

Prior to the release of the first iPad in 2010, the operating system running on the iPhone was referred to as iPhone OS. Given that the operating system used for the iPad is essentially the same as that on the iPhone it didn't make much sense to name it iPad OS. Instead, Apple decided to adopt a more generic and non-device specific name for the operating system.

Given Apple's predilection for names prefixed with the letter 'i' (iTunes, iBookstore, iMac etc) the logical choice was, of course, iOS. Unfortunately, iOS is also the name used by Cisco for the operating system on its routers (Apple, it seems, also has a predilection for ignoring trademarks). When performing an internet search for iOS, therefore, be prepared to see large numbers of results for Cisco's iOS which have absolutely nothing to do with Apple's iOS.

An Overview of the iOS 4 Architecture

As previously mentioned, iOS consists of a number of different software layers, each of which provides programming frameworks for the development of applications that run on top of the underlying hardware. These operating system layers can be presented diagrammatically as illustrated in the following figure:



some diagrams designed to graphically depict the iOS software stack show an additional box positioned above the Cocoa Touch layer to indicate the applications running on the device.

In the above diagram we have not done so since this would suggest that the only interface available to the app is Cocoa Touch. In practice, an app can directly call down any of the layers of the stack to perform tasks on the physical device.

That said, however, each operating system layer provides an increasing level of abstraction away from the complexity of working with the hardware. As an iOS developer you should, therefore, always look for solutions to your programming goals in the frameworks located in the higher level iOS layers before resorting to writing code that reaches down to the lower level layers.

In general, the higher level of layer you program to, the less effort and fewer lines of code you will have to write to achieve your objective. And as any veteran programmer will tell you, the less code you have to write the less opportunity you have to introduce bugs.

Now that we have identified the various layers that comprise iOS 4 we can now look in more detail at the services provided by each layer and the corresponding frameworks that make those services available to us as application developers.

The Cocoa Touch Layer

The Cocoa Touch layer sits at the top of the iOS stack and contains the frameworks that are most commonly used by iPhone application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API (as found on Apple desktop and laptop computers) and has been extended and modified to meet the needs of the iPhone. The Cocoa Touch layer provides the following frameworks for iPhone app development:

UIKit Framework (UIKit.framework)

The UIKit framework is a vast and feature rich Objective-C based programming interface. It is, without question, the framework with which you will spend most of your time working. Entire books could, and probably will, be written about the UIKit framework alone. Some of the key features of UIKit are as follows:

- User interface creation and management (text fields, buttons, labels, colors, fonts etc) Application lifecycle management
- Application event handling (e.g. touch screen user interaction) Multitasking
- Wireless Printing
- Data protection via encryption
- Cut, copy, and paste functionality
- Web and text content presentation and management
- Data handling
- Inter-application integration
- Push notification in conjunction with Push Notification Service
- Local notifications (a mechanism whereby an application running in the background can gain the user's attention)
- Accessibility
- Accelerometer, battery, proximity sensor, camera and photo library interaction. Touch screen gesture recognition
- File sharing (the ability to make application files stored on the device available via iTunes) Blue tooth based peer to peer connectivity between devices
- Connection to external displays

Map Kit Framework(MapKit.framework)

If you have spent any appreciable time with an iPhone then the chances are you have needed to use the Maps application more than once, either to get a map of a specific area or to generate driving directions to get you to your intended destination.

The Map Kit framework provides a programming interface that enables you to build map based capabilities into your own applications. This allows you to, amongst other things, display scrollable maps for any location, display the map corresponding to the current geographical location of the device and annotate the map in a variety of ways.

Push Notification Service

The Push Notification Service allows applications to notify users of an event even when the application is not currently running on the device. Since the introduction of this service it has most commonly been used by news based applications.

Typically when there is breaking news the service will generate a message on the device with the news headline and provide the user the option to load the corresponding news app to read more details. This alert is typically accompanied by an audio alert and vibration of the device. This feature should be used sparingly to avoid annoying the user with frequent interruptions.

Message UI Framework (MessageUI.framework)

The Message UI framework provides everything you need to allow users to compose and send email messages from within your application. In fact, the framework even provides the user interface elements through which the user enters the email addressing information and message content. Alternatively, this information can be pre-defined within your application and then displayed for the user to edit and approve prior to sending.

Address Book UI Framework (AddressUI.framework)

Given that a key function of the iPhone is as a communications device and digital assistant it should not come as too much of a surprise that an entire framework is dedicated to the integration of the address book data into your own applications. The primary purpose of the framework is to enable you to access, display, edit and enter contact information from the iPhone address book from within your own application.

Game Kit Framework (GameKit.framework)

The Game Kit framework provides peer-to-peer connectivity and voice communication between multiple devices and users allowing those running the same app to interact. When this feature was first introduced it was anticipated by Apple that it would primarily be used in multi-player games (hence the choice of name) but the possible applications for this feature clearly extend far beyond games development.

iAd Framework (iAd.framework)

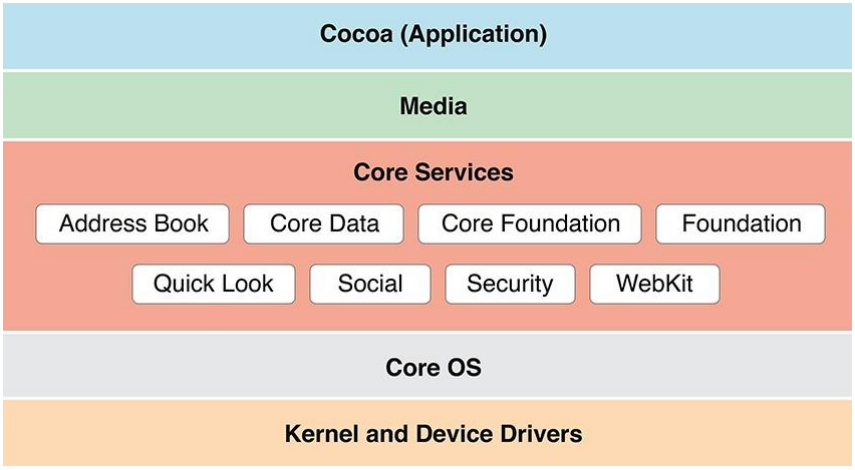
The purpose of the iAd Framework is to allow developers to include banner advertising within their applications. All advertisements are served by Apple's own ad service.

Event Kit UI Framework

The Event Kit UI framework was introduced in iOS 4 and is provided to allow the calendar events to be accessed and edited from within an application.

1. THE IOS CORE SERVICES LAYER

The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built and consists of the following frameworks.



Address Book framework (AddressBook.framework)

The Address Book framework provides programmatic access to the iPhone Address Book contact database allowing applications to retrieve and modify contact entries.

CFNetwork Framework (CFNetwork.framework)

The CFNetwork framework provides a C-based interface to the TCP/IP networking protocol stack and low level access to BSD sockets. This enables application code to be written that works with HTTP, FTP and Domain Name servers and to establish secure and encrypted connections using Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

Core Data Framework (CoreData.framework)

This framework is provided to ease the creation of data modeling and storage in Model-View-Controller (MVC) based applications. Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data in an application.

Core Foundation Framework (CoreFoundation.framework)

The Core Foundation is a C-based Framework that provides basic functionality such as data types, string manipulation, raw block data management, URL manipulation, threads and run loops, date and times, basic XML manipulation and port and socket communication.

Additional XML capabilities beyond those included with this framework are provided via the libXML2 library. Though this is a C-based interface, most of the capabilities of the Core Foundation framework are also available with Objective-C wrappers via the Foundation Framework.

Core Media Framework (CoreMedia.framework)

The Core Media framework is the lower level foundation upon which the AV Foundation layer is built. Whilst most audio and video tasks can, and indeed should, be performed using the higher level AV Foundation framework, access is also provided for situations where lower level control is required by the iOS application developer.

Core Telephony Framework (CoreTelephony.framework)

The iOS Core Telephony framework is provided to allow applications to interrogate the device for information about the current cell phone service provider and to receive notification of telephony related events.

EventKit Framework (EventKit.framework)

An API designed to provide applications with access to the calendar and alarms on the device.

Foundation Framework (Foundation.framework)

The Foundation framework is the standard Objective-C framework that will be familiar to those that have programmed in Objective-C on other platforms (most likely Mac OS X). Essentially, this consists of Objective-C wrappers around much of the C-based Core Foundation Framework.

Core Location Framework (CoreLocation.framework)

The Core Location framework allows you to obtain the current geographical location of the device (latitude and longitude) and compass readings from with your own applications. The method used by the device to provide coordinates will depend on the data available at the time the information is requested and the hardware support provided by the particular iPhone model on which the app is running (GPS and compass are only featured on recent models). This will either be based on GPS readings, Wi-Fi network data or cell tower triangulation (or some combination of the three).

Mobile Core Services Framework (MobileCoreServices.framework)

The iOS Mobile Core Services framework provides the foundation for Apple's Uniform Type Identifiers (UTI) mechanism, a system for specifying and identifying data types.

A vast range of predefined identifiers have been defined by Apple including such diverse data types as text, RTF, HTML, JavaScript, PowerPoint .ppt files, PhotoShop images and MP3 files.

Store Kit Framework (StoreKit.framework)

The purpose of the Store Kit framework is to facilitate commerce transactions between your application and the Apple App Store. Prior to version 3.0 of iOS, it was only possible to charge a customer for an app at the point that they purchased it from the App Store. iOS 3.0 introduced the concept of the "in app purchase" whereby the user can be given the option make additional payments from within the application. This might, for example, involve implementing a subscription model for an application, purchasing additional functionality or even buying a faster car for you to drive in a racing game.

SQLite library

Allows for a lightweight, SQL based database to be created and manipulated from within your iPhone application.

System Configuration Framework (SystemConfiguration.framework)

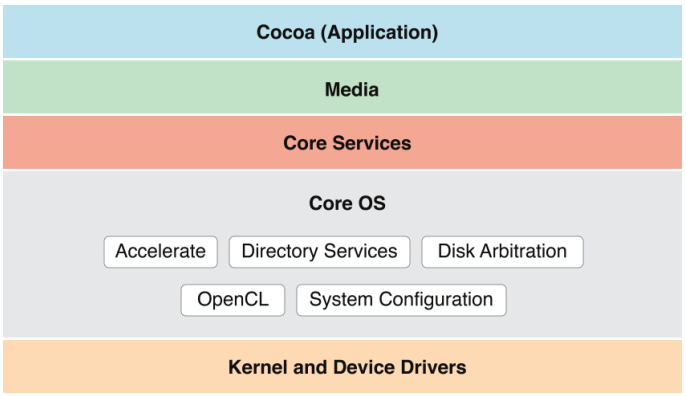
The System Configuration framework allows applications to access the network configuration settings of the device to establish information about the "reachability" of the device (for example whether Wi-Fi or cell connectivity is active and whether and how traffic can be routed to a server).

Quick Look Framework (QuickLook.framework)

One of the many new additions included in iOS 4, the Quick Look framework provides a useful mechanism for displaying previews of the contents of files types loaded onto the device (typically via an internet or network connection) for which the application does not already provide support. File format types supported by this framework include iWork, Microsoft Office document, Rich Text Format, Adobe PDF, Image files, public.text files and comma separated (CSV).

THE IOS CORE OS LAYER

The Core OS Layer occupies the bottom position of the iOS stack and, as such, sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.



Accelerate Framework (Accelerate.framework)

Introduced with iOS 4, the Accelerate Framework provides a hardware optimized C-based API for performing complex and large number math, vector, digital signal processing (DSP) and image processing tasks and calculations.

External Accessory framework (ExternalAccessory.framework)

Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone via the 30-pin dock connector or wirelessly via Bluetooth.

Security Framework (Security.framework)

The iOS Security framework provides all the security interfaces you would expect to find on a device that can connect to external networks including certificates, public and private keys, trust policies, keychains, encryption, digests and Hash-based Message Authentication Code (HMAC).

System (LibSystem)

As we have previously mentioned, the iOS is built upon a UNIX-like foundation. The System component of the Core OS Layer provides much the same functionality as any other UNIX like operating system. This layer includes the operating system kernel (based on the

Mach kernel developed by Carnegie Mellon University) and device drivers. The kernel is the foundation on which the entire iOS is built and provides the low level interface to the underlying hardware. Amongst other things the kernel is responsible for memory allocation, process lifecycle management, input/output, inter-process communication, thread management, low level networking, file system access and thread management.

As an app developer your access to the System interfaces is restricted for security and stability reasons. Those interfaces that are available to you are contained in a C-based library called LibSystem. As with all other layers of the iOS stack, these interfaces should be used only when you are absolutely certain there is no way to achieve the same objective using a framework located in a higher iOS layer.