

three hybrid systems: the Apple Mac OS X operating system and the two most prominent mobile operating systems—iOS and Android.

2.7.5.1 Mac OS X

The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure 2.16, it is a layered system. The top layers include the *Aqua* user interface (Figure 2.4) and a set of application environments and services. Notably, the *Cocoa* environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications. Below these layers is the *kernel environment*, which consists primarily of the Mach microkernel and the BSD UNIX kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as *kernel extensions*). As shown in Figure 2.16, the BSD application environment can make use of BSD facilities directly.

2.7.5.2 iOS

iOS is a mobile operating system designed by Apple to run its smartphone, the *iPhone*, as well as its tablet computer, the *iPad*. iOS is structured on the Mac OS X operating system, with added functionality pertinent to mobile devices, but does not directly run Mac OS X applications. The structure of iOS appears in Figure 2.17.

Cocoa Touch is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices. The fundamental difference between Cocoa, mentioned earlier, and Cocoa Touch is that the latter provides support for hardware features unique to mobile devices, such as touch screens. The *media services* layer provides services for graphics, audio, and video.

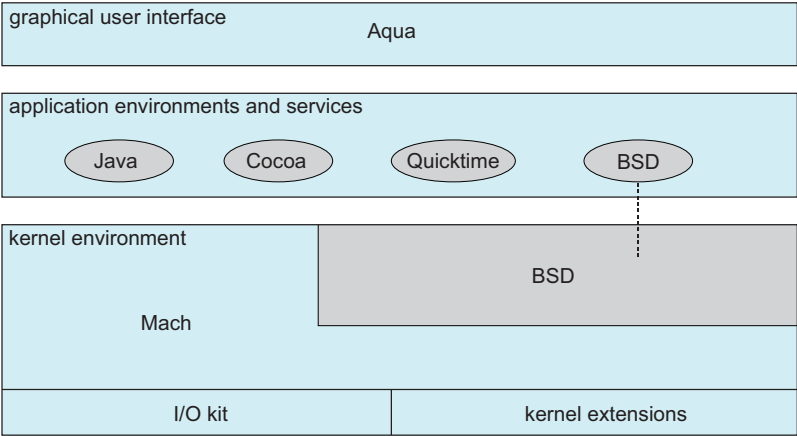


Figure 2.16 The Mac OS X structure.

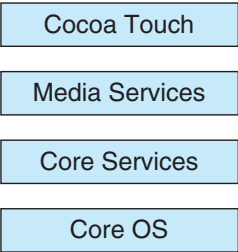


Figure 2.17 Architecture of Apple’s iOS.

The **core services** layer provides a variety of features, including support for cloud computing and databases. The bottom layer represents the core operating system, which is based on the kernel environment shown in Figure 2.16.

2.7.5.3 Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.18.

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications. At the bottom of this software stack is the Linux kernel, although it has been modified by Google and is currently outside the normal distribution of Linux releases.

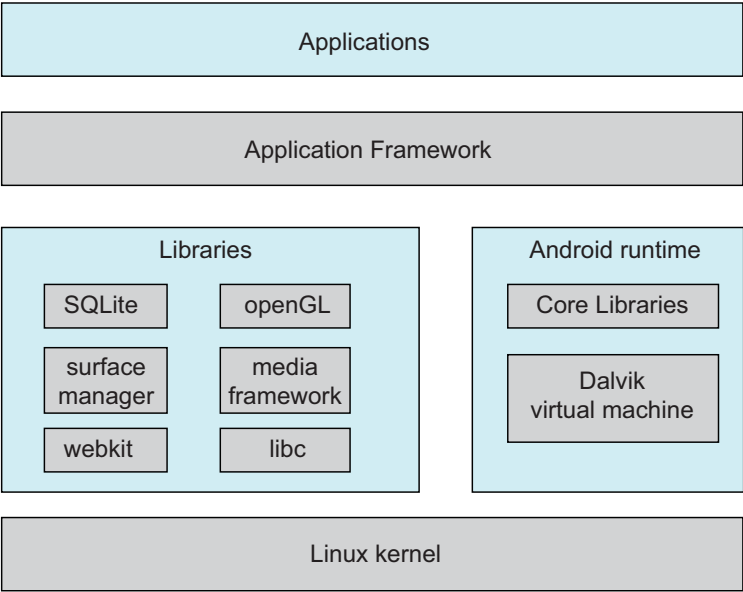


Figure 2.18 Architecture of Google’s Android.

Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.

The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

2.8 Operating-System Debugging

We have mentioned debugging frequently in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

2.8.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system operators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the “core” in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel’s memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash