1 Welcome to Mac OS X

- Origins of Mac OS X
- Macintosh user interface
- Mac OS user interface
- Mac OS X UNIX underpinnings
- Mac OS X system architecture

You're never too old to become younger.

—Mae West

The Macintosh burst onto the personal computing scene in January 1984, instantly changing the way people view and interact with personal computers. Arguably, no other product has affected our perception of personal computers, or how we expect them to look and operate, more than the Macintosh.

In this chapter, we'll look at the Mac OS X at the user and architectural levels. This introduction provides some background on the Macintosh user interface, discusses the Mac OS X interface, and concludes with a discussion of the Mac OS X architecture and system components. Section 1.4 contains some terms and concepts associated with operating systems. Appendix D, "A brief history of UNIX," gives a brief overview of UNIX and operating system concepts.

1.1 Introduction

The Macintosh was separated from other personal computers of the day by its uncomplicated graphical user interface (GUI) and ease of use. The designers of the Macintosh accomplished this differentiation by using real-world metaphors for user interface elements, direct feedback for user actions, and a consistent user interface shared between and among applications. A central theme of the Macintosh is that the user is in charge of the computer, not the other way around; the system should always respond to the user's needs and actions. These design principles have spawned a user community that is vehemently loyal to the Macintosh and expects its applications to behave in a consistent manner.

From a user's point of view, the Macintosh has always been an elegant system that is simple to use and easy to understand. This is no accident: Macintosh developers have a highly acute sense of computer-user interaction and user interface design, and take great pride in producing software that respects the way people work and use their computers. Macintosh programmers are as concerned about user interfaces issues as program features or the computational aspects of a program. If users love Macintoshes for their elegance and simplicity, programmers love them because they are uncomplicated, well designed, and great deal of fun to program.

1.1.1 Origins of Mac OS X

In March 2001, Apple released a new generation operation system for the Macintosh platform called Mac OS X (*X* is pronounced "ten"). Many innovations and developments led to its creation. In the mid-1990s, Apple began work on its next generation operating system, called *Copland*. Copland attempted to address some of the problems associated with Apple's then-current operating system, Mac OS. The Mac OS had always excelled in its user interface and ease of use, but it was falling behind other personal computer operating systems in performance, features, and stability. For various reasons, Copland never panned out; in 1996 the project was cancelled.

Also in 1996, Apple purchased NeXT computer and began work on another operating system named *Rhapsody*. The foundation of Rhapsody was NeXTSTEP, the operating system Apple acquired from NeXT computer. NeXTSTEP was a BSD-like operating system based on a Mach kernel, which Apple engineers modified for Rhapsody. Over time, Rhapsody's design and features evolved first into Mac OS X Server and then Mac OS X.

Mac OS X represents a fundamental departure from past Apple operating systems, merging the best features of the traditional Mac OS with the rock-solid reliability of UNIX. At the core of the system is *Darwin*, an open source UNIX-based operating system built on Mach 3.0 and 4.4BSD; it supplies the UNIX underpinnings for Mac OS X. On top of Darwin, Apple engineers layered various Macintosh services that give the system its Macintosh character and functionality. On top of all this sits a brand new user interface, called *Aqua*.

At one level, the system is a UNIX box, providing access to all the familiar command-line tools and commands, as well as a wealth of open-source software and programs including Apache, MySQL, Perl, and GNU software. In addition, free implementations of X Window can be run under OS X, permitting local and remote access to a wealth of X Window–based systems and applications. At another level, the system is a Macintosh; you can run native Mac OS X as well as older Macintosh application.

Figure 1.1 shows an OS X machine running a variety of Mac OS X, UNIX, and older Macintosh software.

Another interesting feature is the renewed viability of the Macintosh platform within the scientific, engineering, and research communities. Many people in these areas have had a bias toward using a Macintosh, but because of the limitations of the Mac OS, have moved to other platforms to run simulations and conduct research. You can now run simulations and develop computationally

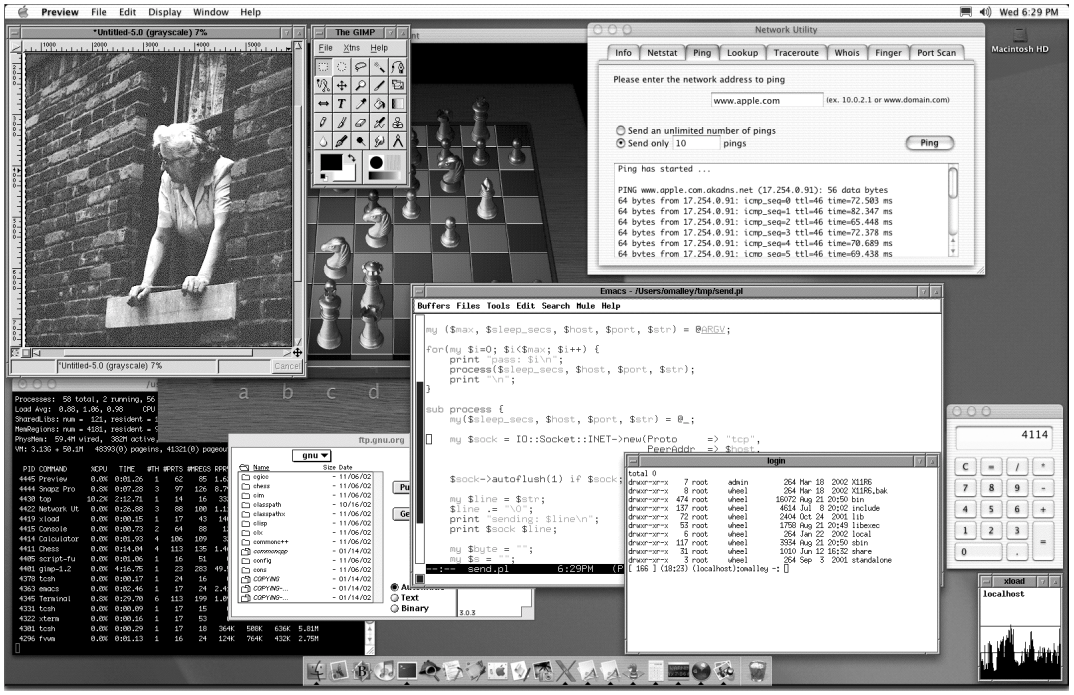

Figure 1.1 An example of Mac OS X running UNIX (text and X Window based), Mac OS X, and Mac Classic software

intensive software on the platform; in many cases, you only need to recompile the source code for the UNIX-based program under Mac OS X.

These are truly interesting times for Macintosh users, as well as those moving to Mac OS X from other UNIX-based platforms.

1.2 The Macintosh user interface

When people make the transition to the Macintosh from other systems like UNIX, often the first thing they notice is how simple and logical the interface is and how easily they can learn to use the system. As a friend, and long-time UNIX user, pointed out to me, when he's using a Macintosh he spends less time working the levers of the operating system and more time getting work done. The reasons include Apple's understanding of user needs and the company's insistence on developers following a set of interface guidelines when building Macintosh applications.

In the mid-1980s, Apple came up with some fundamental principles for how the Macintosh and its applications should look and feel: the Macintosh Human Interface Guidelines. The goal was to present users with a powerful, consistent system that was easy to use and that had an uncomplicated user interface. These design goals centered on the user being in charge of the computer and advocated techniques such as direct feedback for user actions, use of real-world metaphors for user interface elements, and a consistent user interface shared between and among applications. (Remember, these were the days when most personal computers ran MS-DOS and users interacted with the system using a command prompt and text-based interfaces.)

For example, imagine you were developing an application and working on its user interface. One method would be to design your application's interface from scratch according to your own preferences, or possibly base it on a similar program's interface and make appropriate modifications. Now imagine if developers built all applications this way. The result would be applications that look and behave very differently and implement common operations in dissimilar ways. The consequence for users would be an uneven user experience and constant relearning of tasks when moving to new applications.

Macintosh programmers did things differently. Instead of designing and laying out their applications' user interface any way they wished, they followed the guidelines Apple provided them; this process ensured that applications maintained the Macintosh look and feel. In addition, Apple's toolbox routines did much of the work of supporting that interface—for most developers, breaking the guidelines involved more work than following them. At first this programming approach was quite a shift, and it probably would not have succeeded if the guidelines had not been well thought out or did not make sense. Luckily, Apple employed some smart, experienced people who cared a great deal about how users interact with computers. The Macintosh Human Interface Guidelines became a cornerstone for user interface development on the Macintosh, and most applications were judged and evaluated based on these principles.

The consequences of these guidelines are applications that implement interface elements and standard operations in a consistent way, enabling users to easily translate their current knowledge to new programs. Over the years, the interface guidelines have grown as new technologies and interface components have been added to the Macintosh system. Today, the Aqua Human Interface Guidelines (http://developer.apple.com/techpubs/macosx/Essentials/AquaHIGuidelines) describe how to construct user interfaces for Mac OS X applications. To a degree,

the Aqua guidelines are another extension of the original interface guidelines, addressing new features of the Mac OS X user interface.

The most important lesson to take from this discussion is that Apple has put a lot of time and thought into how Macintosh applications should look and behave. The company has produced an excellent set of rules and recommendations for constructing contemporary user interfaces, and developers should read, understand, and follow them when developing Macintosh applications. Try to envision the programs you write for Mac OS X as being members of a complete, well-thought-out system where certain rules exists to promote the user experience. Your application should exist within this context, and not as a separate entity.

1.3 The Mac OS X user interface

The strength of the Macintosh has always been its user interface and ease of use. The new Mac OS X Aqua interface maintains the tradition of intelligent, easy-to-use Macintosh user interfaces, but sports a distinctive, liquid-like look, as well as many new and advanced interface components and features. Figure 1.2 shows an example of the Aqua user interface.

The Aqua interface continues to use real-world metaphors to represent computer resources. Navigating and using the system is simple because you are already familiar with many of these concepts. Overall, the Aqua user interface is simple and intuitive compared to UNIX desktops and window managers such as GNOME (http://www.gnome.org), KDE (http://www.kde.org), and fvwm (http://www.fvwm.org). As a result, you will require little upfront information to begin using the system.

1.3.1 The desktop

The Mac OS X desktop is analogous to a real office desk, which functions as your primary workspace and repository of information. A program called the *Finder* works with the system software to provide users with file management and process invocation functions, and presents and manages the desktop.

1.3.2 Menus

Under Aqua, an application displays its menu bar at the top of the screen. This is different from Windows or UNIX environments, where the menu bar appears at the top of each application window. The items in the menu bar are ordered as follows (from left to right): Apple menu, application menu, application-defined menus, window menu, help menu, and menu status bar items (see figure 1.3).

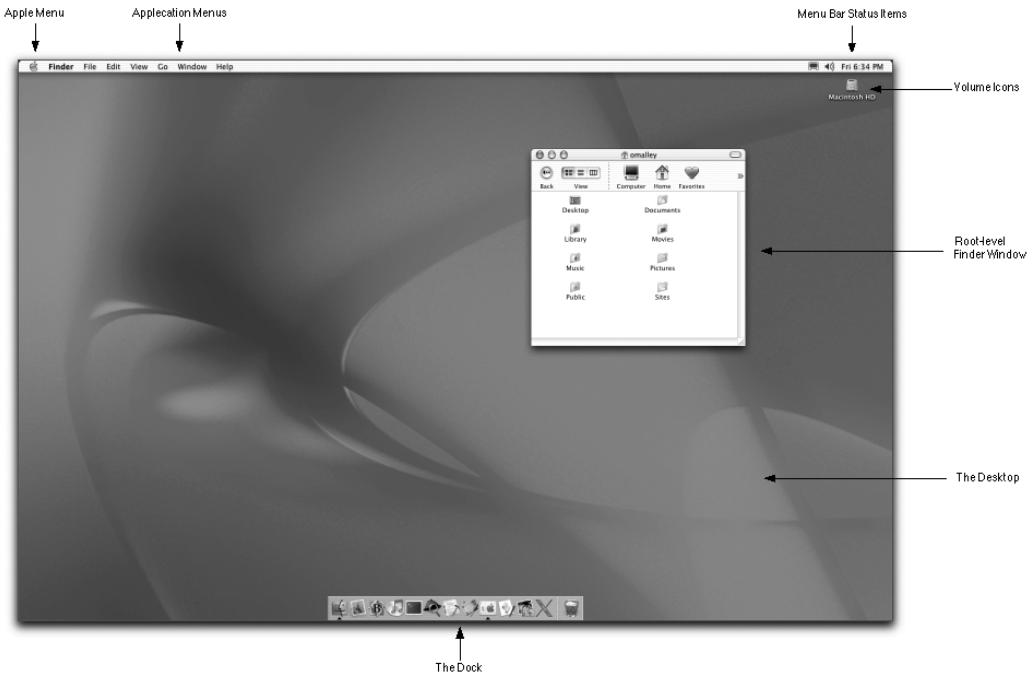

Figure 1.2 Aqua, the user interface for Mac OS X, builds on many features of the original Macintosh user interface. However, it has an entirely new look and feel, as well as many new features.

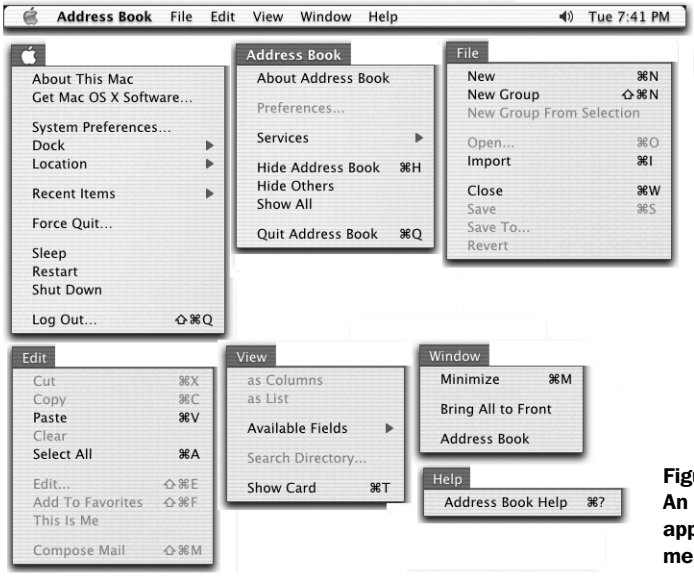

Figure 1.3
An example of a Mac OS X application's (Address Book) menu bar and menu items

First is the Apple menu, a system-wide menu whose contents do not change. Its commands permit users to perform tasks that operate on the system as a whole and are independent of any particular application. Commands support accessing system preferences, restarting and shutting down the computer, and logging off the current session.

Next is the Application menu, which holds items that apply to a specific application. Menu items include the application's preferences, services provided by other applications, and the Quit option. The menu name is bold, so it stands out from the other menus.

The next set of menus is application defined, but it typically includes the following menus, in this order: File and Edit, application-defined menus (possibly including View), Window and Help. They perform these functions:

- The File menu implements operations for document management such as opening, creating, and printing documents.
- The Edit menu contains commands for editing application documents and sharing application data over the clipboard.
- The View menu holds commands enabling users to change or alter the view of an application's current window.
- The Window menu lists currently open windows as well as window operations.
- The Help menu provides access to application help.
- Status items appear as the final, rightmost menu item and display information about system services, enabling quick access to system settings.

NOTE *Clipboard* is a Macintosh term for a common shared data holder used by the applications to temporarily hold data or to transfer data from one application to another. On the Macintosh, terms like *copy*, *cut*, and *paste* describe editing operations. For example, after you highlight an item in a document, you can perform a cut, which moves the selected item from the document to the clipboard; a copy, which copies the selected item to the clipboard; or a paste, which copies the item on the clipboard to the desired location.

1.3.3 The Dock

The *Dock*, located at the bottom of the screen in Figure 1.2, is a small toolbar that provides a standard, system-supplied location for you to organize commonly

accessed items such as applications, documents, and other information. It also aids in maneuvering between running applications.[1]

You add items by dragging their icons to the Dock; you remove items by dragging them off the Dock. Clicking an icon will bring it to the foreground, launching it first if it is not already running. A triangle next to an application icon indicates that the application is running. The Dock also holds the familiar Macintosh Trash icon, which collects files waiting to be deleted from the system. You can customize the Dock's appearance and behavior through the System Preference program, located in /Application.

1.3.4 Window layering

The original Mac OS imposed a window-layering scheme that placed all application windows conceptually on a single layer. This meant that if you were using one application and you clicked a window from another application, all of that application's windows came to the foreground. Mac OS X implements a different window-layering model: windows within an application are independent of one another, and can therefore be interleaved with windows from different applications.

Imagine you have two applications running, each with several visible windows. Under Mac OS X, only the window you click comes to the foreground, enabling windows from different applications to be interspersed. The result is more information simultaneously visible at a time and fewer visible transitions between applications. Perceptually, the new window-layering scheme blurs the boundaries between applications, causing you to feel as if you are interacting with the system as a whole, rather than with individual applications. (By the way, clicking the application's icon on the Dock will bring all of the application's windows to the foreground.)

1.3.5 Dialog boxes

Past Macintosh operating systems used two main types of dialog boxes: modal and modeless. A *modal* dialog box forces you to work within the mode of the dialog box only; once the dialog box is open, the only way to interact with another part of the system is to close the dialog box. Conversely, a *modeless* dialog box does not force you to interact only with it; you can simultaneously use the modeless dialog box and other parts of the system.

---

[1] Bruce Tognazzini, a noted expert on user interfaces design, has written an interesting column called "Top 10 Reasons the Apple Dock Sucks" that discusses his objections to the Dock. Check it out at http://www.asktog.com/columns/044top10docksucks.html.

Figure 1.4
Mac OS X Sheets seem fixed, or attached, to an application's document or window. They simplify identifying the owner of the Sheet.

A *Sheet* is a Mac OS X implementation of a modal dialog box. When an application displays a Sheet, it appears attached to the application's document or window (see figure 1.4). Because it attaches to its creator, you can always tell what program element the Sheet belongs to. See the Aqua Human Interface guidelines for more information about Sheets (http://developer.apple.com/techpubs/macosx/Essentials/AquaHIGuidelines/AHIGDialogs/index.html).

1.3.6 Drawers

Drawers are child windows that appear to slide out from their parent. This is another interface element that permits you to access frequently used application features or information without requiring the application to display the Drawers throughout the life of the application. To see Drawers in action, open the Mail application (located in /Applications) and click the Mailbox icon. The mailboxes for your mail accounts will slide in and out from the parent window as you click the icon (see figure 1.5).

1.3.7 Keyboard navigation

The Macintosh has traditionally been a point-and-click interface: users interact with the system using a mouse. Over the years, the system has included increasing support for system navigation through the keyboard at both the Finder and application levels. Aqua carries on this tradition by providing more keyboard options you can use to navigate the system.

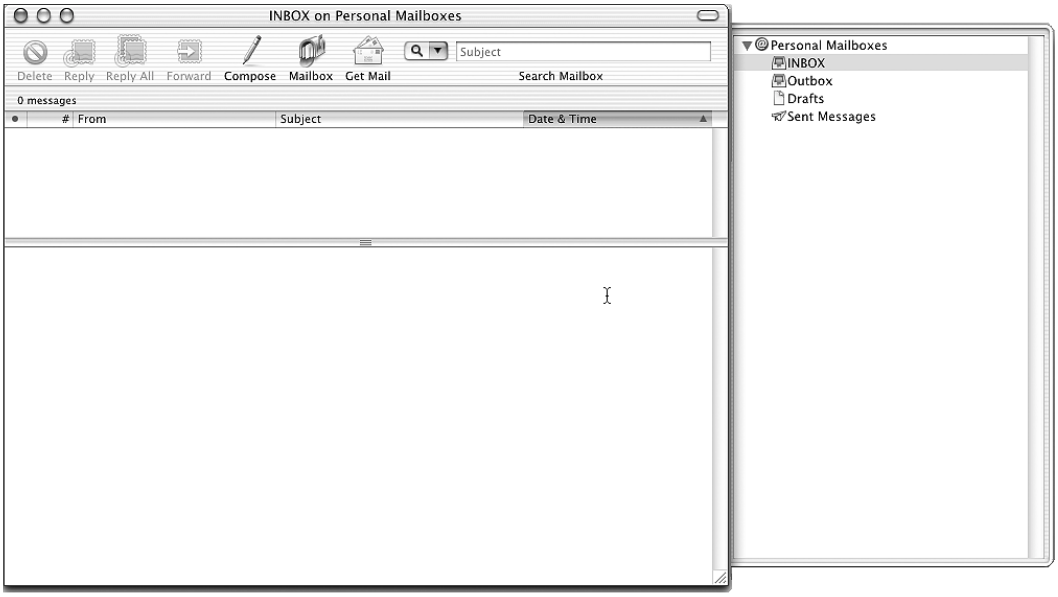

Figure 1.5 Drawers slide out from their parent window, enabling access to frequently used application features or information.

To take full advantage of the keyboard, open the System Preference program, select the Keyboard pane, select the Full Keyboard Access tab, and make sure the Turn On Full Keyboard Access checkbox is checked. The Use Control With menu enables you to change the keys associated with each command. Now, you can use the keyboard to select interface elements such as application menus and the Dock.

1.3.8 Other interface features

Mac OS X includes lots of other interface features, including transparent windows and menus that let you see through a window or menu to what is behind it. The appearance of icons and lists has improved, and there's a new help system and a new system font.

1.4 The Mac OS X architecture

From a user's point of view, the Mac OS X system is its user interface, applications, and services. For developers, however, the interface is simply a facade; behind it exists the Mac OS X operating system, a complex web of software that handles the interactions between user requests and computing resources.

The heart of this system software is the *kernel*. The kernel provides the operating system's basic computing services such as interrupt handling, processor and memory management, and process scheduling. Two types of kernels form the basis for most operating systems: the *monolithic* kernel and the *microkernel*. A monolithic kernel encapsulates nearly all the operating system layers within one program, which runs in kernel space. A microkernel implements a subset of operating system services, runs in kernel space, and is much smaller than the monolithic kernel. Additional services, implemented on top of the kernel as user programs (running in user space), export well-defined interfaces and communication semantics. To perform a service that resides outside of kernel space, the kernel communicates with the user-level service through message passing. Generally, a monolithic kernel is faster but larger than a microkernel.

The original Mac OS was more a collection of cooperating system services, whose design did not divide neatly into user and kernel domains. In addition, its handling of critical operating system tasks such as memory management and process management was showing its age, which led Apple to look into alternatives for its future OS. For example, most of us are familiar with operating systems that use preemptive multitasking and fixed-process scheduling policies. Under UNIX, one policy is for the process scheduler to divide CPU time into time slices, assigning each process a quantum of CPU time. If the running process has not terminated by the end of its quantum, the operating system will switch processes by preempting the running process and activating the next.

Contrast this to Mac OS, which implemented a scheduling called *cooperative multitasking*. It works as follows: when you run a program, the operating system loads the program into memory, schedules it for execution on the CPU, and runs the program only when the currently running program surrenders the CPU. It is the responsibility of each program, not the operating system, to occasionally hand over the CPU to allow other programs to run. As you can imagine, this scheduling is suboptimal, because one rogue program can monopolize the CPU and disallow others from running. Mac OS X is built on UNIX, and therefore uses preemptive multitasking; the kernel manages process-scheduling policies.

Another difference between Mac OS X and earlier Macintosh systems is memory management. Mac OS did not enforce memory protection of the system or application partitions. Applications were free to write to memory outside their own address space and could potentially take down other applications, as well as the entire system. Under Mac OS X, this is not possible: accessing memory outside a program's address space will result in a segment fault and the process will dump core, but it will not take down the operating system or other processes with it.

1.4.1 Architecture layers

The Mac OS X architecture is composed of several layers, each responsible for different system services. It's important to keep in mind that Mac OS X is built on top of a UNIX-based kernel, which provides the system with its plumbing (core services) and supports the various application layers with which the user interacts. It's useful to view Mac OS X as two systems, one built on the other (see figure 1.6).

At the core of Mac OS X is Darwin, an open source operating system based on Mach 3.0 and 4.4BSD. Darwin is a complete operating system that does not require higher-level Macintosh components to run. The Darwin system has two overall components: the kernel environment and the BSD emulation layer. The kernel environment provides core operating system services; the emulation layer supplies the system with the BSD user environment, or operating system personality. In fact, you can install Darwin on a PowerPC or x86 machine and use it as a stand-alone BSD-like system.

Macintosh-specific system components, built on top of the Darwin kernel environment, give Mac OS X its Macintosh character and services. Think of Darwin

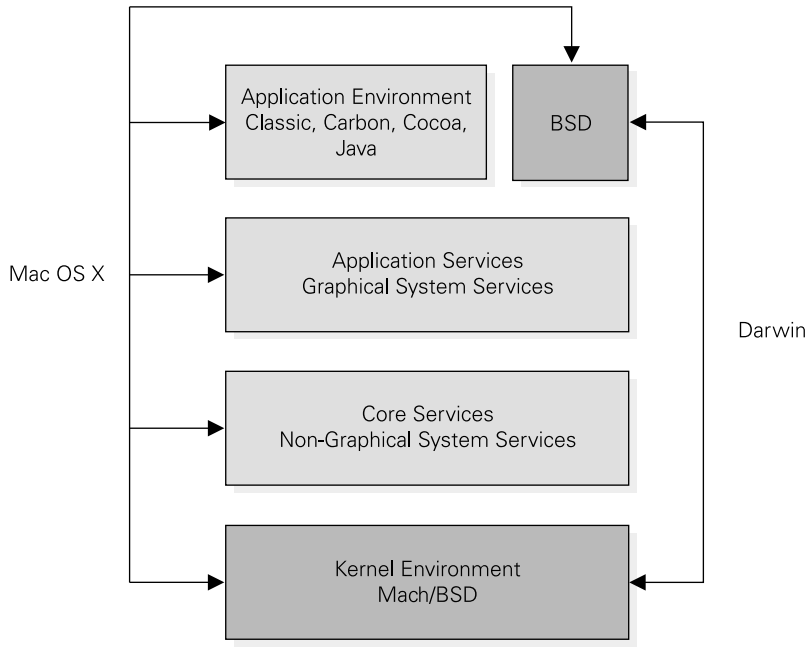

Figure 1.6 Mac OS X is a series of software layers, each providing services for the layer above it.

as the BSD-based operating system core and the Macintosh components as putting the Mac into OS X. This classification enables you to see that Mac OS X is built on top of Darwin, and that Darwin is a complete UNIX system within itself.

Let's begin with a brief overview of the Mac OS X system components:

- The lowest layer is the Mach/BSD-based kernel, called the *kernel environment*. It provides the system with core operating system services such as processor and memory management, file systems, networking, and device access and control.
- The *Core Services layer* implements a central set of non-graphical routines that various Macintosh APIs access. This layer includes facilities for application interaction with file systems, threads, and memory, and provides routines for manipulating strings, accessing local and remote resources through URLs, and XML parsing.
- Above the Core Services layer is the *Application Services layer*. Application services supply programs running within the application environment (except BSD) with user interface, windowing, and graphical support, including support for drawing graphical elements on the display, event handling, printing, and window management. This layer includes the Mac OS X window manager.
- The *Application Environment,* like Applications Services, is composed of the different application environments that give the system its user-level environment. Currently, Carbon, Cocoa, Classic, Java, and BSD form this layer, each as a separate application environment. Each provides a distinct runtime environment in which to run programs and interact with the lower layers of the operating system. For example, when you run a Mac OS X Cocoa program, you are in the Cocoa application environment; when you run a Mac OS program, you are interacting with the Classic application environment.
- Above the application environment is *Aqua*, the Mac OS X user interface. Aqua gives the Mac OS X system and programs their look and feel.

Now, let's look at each system layer and its components in more detail.

1.4.2 The kernel environment

The kernel environment supplies Mac OS X with its core operating system services. This layer is composed of two sublayers: the Mach kernel and the BSD layer, which encloses Mach (see figure 1.7). Within these layers are five primary components: Mach, the I/O Kit, BSD, the file system, and networking.

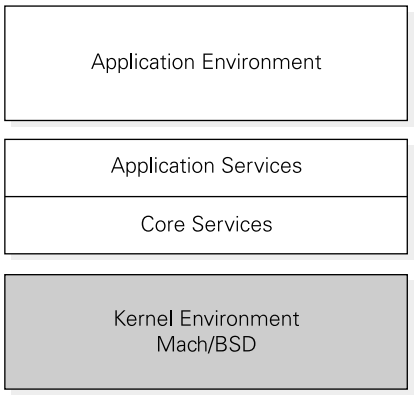

Figure 1.7
The Mac OS X kernel environment supplies the system with its core operating system services.

Mach

At its core, Mac OS X uses the Mach 3.0 microkernel (Mach 3.0 + OSF/Apple enhancements). The Mach portion of the kernel environment is responsible for managing the processor and memory (including virtual memory and memory protection), preemptive multitasking, and handling messaging between operating system layers. Mach also controls and mediates access to the low-level computing resources. It performs the following tasks:

- Provides IPC infrastructure and policies (through ports and port rights), as well as methods (message queues, RPCs, and locks) enabling operating system layers to communicate
- Manages the processor by scheduling the execution and preemption of threads that make up a task
- Supports SMP (symmetric multiprocessing)
- Handles low-level memory management issues, including virtual memory

Keep in mind that Mach is policy neutral, meaning that it has no knowledge of things like file systems, networking, and operating system personalities.

Historically, Mach implements a very small set of core system services in the kernel address space, communicating with additional services in user space through well-defined interfaces and communication semantics. The kernel implementation for Darwin integrates many of these user-space services into the kernel space.

There is a fundamental difference between how a UNIX monolithic kernel and Mach kernel use and implement processes and threads. In a UNIX kernel, the basic level of scheduling is the process, not the thread. All threads within the process are bound by the scheduling priority of the process and are not seen by

the kernel as schedulable entities. For example, if the operating system suspends a process, all its threads are also suspended.

Contrast this with Mach. Mach divides the concept of a UNIX process into two components: a task and a thread. A task contains the program's execution environment (system resources minus control flow) and its threads. With Mach, the thread is the basic unit of scheduling, as opposed to a UNIX process, which uses the process as the scheduling unit. Under Mach, scheduling priority is handled on a per-thread basis: the operating system coordinates and schedules threads from one or many tasks, not on a per-process level.

I/O Kit

The I/O Kit is an object-oriented framework for developing Mac OS X drivers, implemented in a subset of C++. Developing device drivers is a specialized task, requiring detailed knowledge, experience, and highly specific code. The I/O Kit attempts to increase code reuse and reduce the learning curve of driver development by providing programmers with a framework that encapsulates basic device driver functionality in base classes, which are extended to implement specific device drivers. Conceptually, this approach is very similar to application frameworks and class libraries. The I/O Kit infrastructure enables true plug and play, as well as dynamically loaded and unloaded drivers and dynamic device management.

BSD

Another component of the Darwin kernel environment is its implementation of BSD, which is based on 4.4BSD. The BSD kernel component sits on top of the modified Mach kernel, running in the kernel's address space. This component provides networking services, file systems, security policies, the application process model (process management and signals), the FreeBSD kernel API, and the POSIX API for supporting user space applications. It also provides applications with the BSD interface into the core services of the OS by wrapping the Mach primitives.

The traditional, or pure, microkernel design places many of these BSD components (such as file systems and networking) within user space, not kernel space. Darwin is not a pure microkernel. To address performance concerns, designers modified the kernel by placing some BSD system modules within the kernel space, traditionally reserved for Mach.

File system

Darwin's file system infrastructure is based on an enhanced virtual file system (VFS) and includes support for HFS (hierarchical file system), HFS+ (hierarchical file system plus) , UFS (UNIX file system) , NFS (network file system) , and ISO 9660.

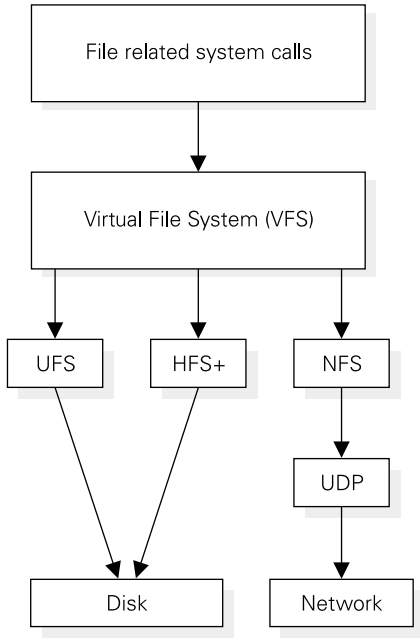

Figure 1.8
The Darwin kernel implements a Virtual File System (VFS) that translates a file-related system call into the matching call for the appropriate file system.

VFS is a kernel-level component that provides an abstract view of the physical file systems through a common interface. VFS accepts file-related system calls (open, close, read, and write) and translates them into the appropriate calls for the target file system (see figure 1.8). VFS is often referred to as supporting *stacks* of file systems (stackable), because it can interact with and add many kinds of file systems and supports augmenting existing file systems with custom code that supplies various services (such as encryption or mirroring).

Networking

Darwin's networking infrastructure is based on 4.4BSD. It includes all the features you'd expect from a BSD-derived system, such as routing, the TCP/IP stack, and BSD-style sockets. This component lives in the BSD layer of the kernel.

Kernel Extensions (KEXTs) and Network Kernel Extensions (NKEs)

Kernel Extensions (KEXTs) give developers the ability to access internal kernel data structures and add functionality to the kernel. KEXTs are dynamically loaded into kernel space without recompiling or relinking the kernel. Because KEXTs run within the kernel, a misbehaving module can potentially bring the system to its knees.

Network Kernel Extensions (NKEs) are a special instance of KEXTs. They permit developers to hook into the networking layers of the kernel and implement new features or modify existing functionality. Like KEXTs, they are dynamically loaded into kernel space and do not require recompiling or relinking of the kernel to execute.

Collectively, these components provide the core services for Darwin, and by extension, Mac OS X. A complete Darwin system adds a BSD emulation or application environment on top of this core layer, providing the userland commands and execution environment you are accustomed to in a BSD system. A complete Darwin system (core layer and BSD application environment) is a BSD-based UNIX implementation that is more than capable of running as a stand-alone operating system. You can run Darwin on a PowerPC or x86 compatible system and install it from either source code or a binary.

NOTE Remember, Darwin is an open-source project, and it is being actively developed; all source code for the operating system is available at no charge. Apple also supports several mailing lists devoted to Darwin development issues.

1.4.3 Core Services layer

The Core Services layer sits above the kernel and is responsible for non-graphical system services (see figure 1.9). Common operations are not coded into each Macintosh API (Carbon and Cocoa); instead, the Core Services layer implements a single code base that the various Macintosh APIs access. Developers use the Carbon and Cocoa APIs to construct Macintosh applications. These services are implemented in the following components:

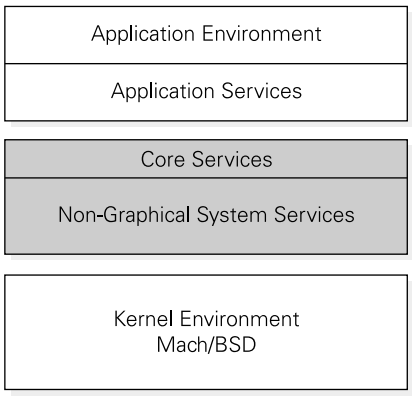

Figure 1.9
The Core Services layer (the software layer above the kernel) provides common, non-graphical routines for the Macintosh APIs (Carbon and Cocoa).

- *Carbon Managers*—A set of services, grouped under various managers, that implement routines providing applications with access to system resources and services. Managers exist for file manipulation (File Manager), text operations (Text Encoding Conversion Manager), memory management (Memory Management Utilities), and thread operations (Thread Manager). For example, when an application requires memory services, it calls a memory allocation routine located in the memory manager; this routine subsequently invokes the kernel-level system calls to manage the actual memory allocation.
- *Core Foundation*—A library that provides many low-level system services such as internationalization, string preferences, and XML services. A handy feature of the Core Foundation is its XML facilities, which include a full-fledged XML parser that implements both tree (DOM) and callback (SAX) based XML parsing.
- *Open Transport*—A single set of routines that offer transport independence and that access the underlying network protocols. Application programs interact with Open Transport through its API to perform network operations such as connecting to and receiving data from other machines. Open Transport uses the networking primitives supplied by the BSD kernel environment code.

1.4.4 Application Services layer

The next layer, called Application Services, supplies the system with the graphical services to construct user interfaces and windowed environments, as well as perform drawing operations, printing, and low-level event forwarding (see figure 1.10).

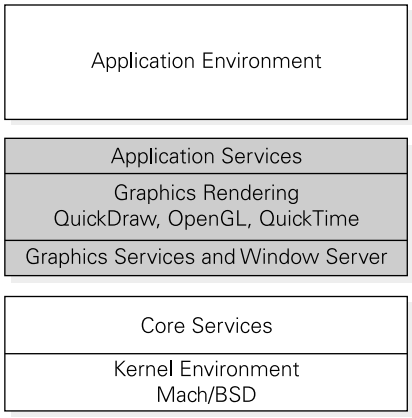

Figure 1.10
The Application Services layer supplies Mac OS X applications with graphics routines and graphics rendering using QuickDraw, OpenGL, and QuickTime.

The main component of this layer is Quartz. The term *Quartz* collectively defines the primary display technologies for Mac OS X. Quartz is composed of two layers: the core graphics services and the rendering libraries.

Core graphics services implement the Mac OS X window server and provide window management as well as event- and cursor-handling services. This sublayer does not actually render objects; the graphics-rendering sublayer that sits on top of the core services contains the following rendering libraries, which perform the graphic-rendering operations:

- *Core Graphics Rendering library*—Performs two-dimensional operations. The Core Graphics Rendering library is used for drawing and rendering using the PDF path (vector) based drawing model.
- *QuickDraw*—Performs two-dimensional operations. QuickDraw is the fundamental graphics display system for the traditional Macintosh OS; it is used to perform traditional Macintosh graphic operations.
- *OpenGL*—Renders three-dimensional operations.
- *QuickTime*—Renders multimedia and digital video in many encoding formats.
- *PDF*—(Developed by Adobe Systems.) Specifies a file format whose files are sharable across platforms. Because the Core Graphics Rendering library uses PDF for vector graphics representation, Mac OS X programs can output files in PDF format—users don't need to buy and install Adobe Acrobat. The printing system is based on this rendering model, as well.

Building these technologies into the Application Services layer provides applications with strong graphics support at the operating system level.

1.4.5 Application Environment layer

Next in this architecture is the Application Environment layer, which provides Mac OS X users with a setting in which to build and run applications (see figure 1.11). This layer, sometimes referred to as the Software Emulation layer, typically contains application emulation environments for implementations of various operating systems. In fact, you can emulate almost any operation system at this layer, including Solaris, Windows, or MS-DOS. Currently, five application environments ship with Mac OS X: Classic, Carbon, Cocoa, Java, and BSD.

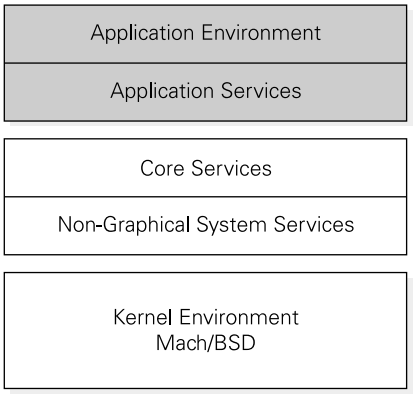

Figure 1.11
The application environment provides a setting for users to run programs. Mac OS X ships with Classic, Carbon, Cocoa, Java, and the BSD application environments.

Classic

The Classic application environment provides a setting for running programs written for Mac OS 9 and earlier. Because Apple does not endorse developing new applications for Mac OS 9, this mode's primary purpose is to support running legacy Macintosh programs. To use Classic mode, your machine must have Mac OS 9.1 or greater installed, which is the default on a typical Mac OS X machine. Therefore, a conventional Mac OS X machine will have both Mac OS X and Mac OS 9.1 installed (under Jaguar, it's version 9.2.2).

There are various approaches to running more than one operating system on a single machine. One method involves setting up a *dual boot* machine. To set up a dual boot machine, you install different operating systems on a single machine and choose the operating system you wish to run at system startup. This method is popular among users of Intel-based UNIX distributions, and it is required to run Linux/BSD and Windows on a single machine.

Another approach is *software emulation*. In this case, you run a software emulator under the host operating system that translates calls of the emulated operating system into the language of the host. This technique permits you to run different operating systems on your machine as long as you have the appropriate emulator. For example, on the Macintosh, a product called Virtual PC (http://www.connectix.com/index_mac.html) enables you to run the Windows operating system and software on your Macintosh. In addition, MacMAME (Multi-Arcade Machine Emulator) is an arcade emulator that lets you run and play your older arcade games on your Macintosh (http://emulation.net/mame).

Under Mac OS X, Classic mode is not emulated as described so far, because Classic instructions are not translated. As Sánchez pointed out:

> The Classic environment in Mac OS X creates a virtual machine inside of Mac OS X, which boots a largely unmodified version of Mac OS 9. Applications that are built for Mac OS 9 and have not been "Carbonized" run in this environment. The Classic environment replaces the hardware abstraction layer in Mac OS 9 with a series of shims that pass requests to parts of Mac OS X. For example, a memory request in Mac OS 9 is fulfilled by a memory request in the Darwin kernel. Mac OS 9 can thereby use resources managed by Mac OS X.[2]

Carbon

Carbon is a set of APIs developers can use to write applications that run under both Mac OS X and early versions of the Mac OS. The original intent of Carbon was to help developers move existing applications from Mac OS to Mac OS X.

Developers write Carbon applications in C and C++. Once an application is "Carbonized," you can run the same binary on your Mac OS X machine as on machines running Mac OS 8.1 or later.

The current Carbon API is a redesigned version of the Mac OS Toolbox. This Toolbox, originally located in the ROM and later in a file loaded by the boot loader in pre-Mac OS X systems, is a set of functions that programs access to construct the graphical elements of a program and interact with core system components. The Toolbox gave the Mac OS its unique appearance and feel, and was a fundamental element of all Macintosh programming. The Carbon API adds many new features to support the architectural changes imposed by Mac OS X. In addition, the API is much smaller, because its designers removed many Mac OS API calls.

Cocoa

Cocoa is an object-oriented environment for developing native Mac OS X applications. Cocoa provides developers with a complete component framework that greatly simplifies and facilitates the development of Mac OS X applications. Apple recommends that developers use Cocoa when writing new applications for Mac OS X.

The etymology of Cocoa begins with NeXT computer and its NeXTSTEP operating system. NeXTSTEP shipped with a set of tools and libraries called *frameworks*

---

[2] Wilfredo Sánchez, "The Challenges of Integrating the Unix and Mac OS Environments" (paper presented at the USENIX 2000 Annual Technical Conference, Invited Talks, San Diego, June 19, 2000), http://www.mit.edu/people/wsanchez/papers/USENIX_2000.

for application development. These NeXTSTEP development tools were subsequently called OpenStep, and are now called Cocoa.

Cocoa applications are currently written in one of two languages: Java and Objective-C. This may seem strange to UNIX developers who are used to developing code in languages such as C, C++, Perl, Python, and Ruby; some may even consider this limitation a reason not to develop Cocoa applications. Resist this temptation. True, many of us would prefer to use Perl or C++ as our main development language when building Cocoa applications, but any programmer who is comfortable with C or C++ can easily get the basics of Objective-C in a few days and be writing useful application in a few weeks.

In addition, some projects are attempting to bring other languages to Cocoa, including Perl, Python, and Ruby. It may just be a matter of time before your favorite language meets Cocoa.[3]

Java

The Java application environment enables development and execution of Java programs and applets. This environment supports the most recent Java Development Kit (JDK) and virtual machine, so programs developed within this environment are portable to virtual machines running on other systems. You can use Java to write applications and applets as well as Cocoa-based applications, although Objective-C is the language of choice for Cocoa development. Apple has made a strong commitment to Java on the Macintosh, so Java developers can rest assured that Java implementations and tools will be available under Mac OS X for years to come.

BSD

The BSD command environment enables users to interact with the system as a BSD workstation, typically through the Terminal application; functionally a shell. This environment supports the BSD tool set, commands, and utilities, and cumulatively provides users with a BSD-derived environment. In fact, the BSD environment and kernel environment form the complete Darwin system. This application environment enables traditional UNIX developers and users to make a smooth transition to the Mac OS X environment by providing them with the accustomed shell, tools, and command set. I for one spend most of my time in the Terminal application using Mac OS X as a BSD-based workstation.

---

[3] The PyObjC project has released a version that enables Python developers to talk to Objective-C objects from Python (http://sourceforge.net/projects/pyobjc/). See chapter 8 for more details.

1.4.6 Aqua

The top layer of the Mac OS X architecture is the Aqua user interface. Aqua is a combination interface implementation and specification that defines recommended user-interface design practices for Mac OS X applications. Think of Aqua as providing guidelines for how applications should look and behave within Mac OS X. These guidelines, documented in the Aqua Human Interface Guidelines, tell developers how to construct a Mac OS X user interface, including the proper layout of dialog boxes and window items' menu structures.

1.5 Summary

You now have a basic understanding of Macintosh user interface principles, as well as Mac OS X's user interface and design. As you can imagine, this chapter is just the tip of the iceberg. If you are interested in this aspect of the Mac OS X system, I encourage you to look at the references in the "Resources" section at the back of this book, and to explore the many online and printed sources that exist on this topic.

In chapter 2, you will learn more about the UNIX side of Mac OS X. You'll see how to accomplish common UNIX tasks under both the Mac OS X command-line interface and the Aqua interface.