System Startup Programming Topics

General

2008-11-19

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to System Startup Programming Topics

System startup (also known as the boot process) involves a sequence of actions that prepare the computer for use by the user. This sequence includes a number of different tasks, including initializing hardware, starting system daemons, and displaying the login window. After a user logs in, the system completes an additional series of actions that sets up the computing environment for that user.

This document provides information that developers of daemons and other low-level system services need to write their code and incorporate it into the startup process. It also provides some useful information for system administrators who must manage the startup process on the computers they manage.

You should read this document if you are writing or porting software that needs to be launched at boot time or at user login time. You should also read this document if you are interested in the system logging facilities provided by Apple system log (asl) facility.

Organization of This Document

This programming topic includes the following articles:

- "The Boot Process" (page 9) describes the overall sequence that occurs when Mac OS X boots, focusing on the places where developers can customize the boot cycle.
- "Daemons" (page 17) provides background information on daemons in Mac OS X.
- "Guidelines for Creating and Launching Daemons" (page 23) provides basic guidelines for developers who want to run daemons on Mac OS X.
- "Creating launchd Daemons and Agents" (page 27) provides guidance on how to create a daemon that runs under `launchd`.
- "Creating a Startup Item" (page 35) provides information about how to create startup items to support the launching of daemons on versions of Mac OS X prior to 10.4.
- "Logging Errors and Warnings" (page 41) provides sample code for logging errors and warnings using `syslog` and `asl`.
- "Customizing Login and Logout" (page 45) provides information on how to customize the login process on older versions of Mac OS X.

The Boot Process

From the moment a user turns on a Mac OS X system to beyond the time the login window appears, Mac OS X executes a boot sequence that readies the system for use. If you provide system services to all users, you might need to execute some code during this process. The following sections explain the basic boot sequence and the places where your code can tie into it.

BootROM

When the power to a Macintosh computer is turned on, the BootROM firmware is activated. BootROM (which is part of the computer's hardware) has two primary responsibilities: it initializes system hardware and it selects an operating system to run. BootROM has two components to help it carry out these functions:

- POST (Power-On Self Test) initializes some hardware interfaces and verifies that sufficient memory is available and in a good state.
- On PowerPC-based Macintosh computers, Open Firmware initializes the rest of the hardware, builds the initial device tree (a hierarchical representation of devices associated with the computer), and selects the operating system to use.

  On Intel-based Macintosh computers, EFI does basic hardware initialization and selects which operating system to use.

If multiple installations of Mac OS X are available, BootROM chooses the one that was last selected by the Startup Disk System Preference. The user can override this choice by holding down the Option key while the computer boots, which causes Open Firmware or EFI to display a screen for choosing the boot volume.

Note: On some legacy hardware, the same version of BootROM can start either Mac OS 9 or Mac OS X. Most current hardware can start only Mac OS X.

BootX, boot.efi, and System Initialization

Once BootROM is finished and a Mac OS X partition has been selected, control passes to the BootX (PowerPC) or boot.efi (Intel) boot loader. The principal job of this boot loader is to load the kernel environment. As it does this, the boot loader draws the "booting" image on the screen.

BootX and boot.efi can be found in the `/System/Library/CoreServices` directory on the root partition. In addition, a copy of boot.efi can be found at `/usr/standalone/i386/boot.efi`.

In "exotic" boot situations such as booting from a software RAID volume, a copy of the boot loader is stored on a separate HFS+ "helper" volume to get the system started. In some versions of Mac OS X, a copy of the kernel and mkext cache are also included on the helper volume. In these cases, the booter and other components on the root volume are unused.

Note: Booting from a UFS volume is deprecated as of Mac OS X v10.5.

The boot loader first attempts to load a prelinked version of the kernel that includes all device drivers that are involved in the boot process. This prelinked kernel is located in `/System/Library/Caches/com.apple.kernelcaches`. By linking these drivers into the kernel ahead of time, boot time is reduced.

If the prelinked kernel is missing, out-of-date, or corrupt, the boot loader attempts to load that same set of device drivers all at once in the form of a single, compressed archive called an mkext cache.

If this cache is also out-of-date, missing, or corrupt, the boot loader searches `/System/Library/Extensions` for drivers and other kernel extensions whose `OSBundleRequired` property is set to a value appropriate to the type of boot (for example, local or network boot).

For more information on how drivers are loaded, see *I/O Kit Fundamentals*.

Once the kernel and all drivers necessary for booting are loaded, the boot loader starts the kernel's initialization procedure. At this point, enough drivers are loaded for the kernel to find the root device. Also from this point, on PowerPC-based Macintosh computers, Open Firmware is no longer accessible (quiesced).

The kernel initializes the Mach and BSD data structures and then initializes the I/O Kit. The I/O Kit links the loaded drivers into the kernel, using the device tree to determine which drivers to link. Once the kernel finds the root device, it roots(*) BSD off of it.

Note: As a terminology aside, the term "boot" was historically reserved for loading a bootstrap loader and kernel off of a disk or partition. In more recent years, the usage has evolved to allow a second meaning: the entire process from initial bootstrap until the OS is generally usable by an end user. In this case, the term is used according to the former meaning.

As used here, the term "root" refers to mounting a partition as the root, or top-level, filesystem. Thus, while the OS boots off of the root partition, the kernel roots the OS off of the partition before executing startup scripts from it.

Prior to Mac OS X v10.4, the remaining system initialization was handled by the `mach_init` and `init` processes. During the course of initialization, these processes would call various system scripts (including `/etc/rc`), run startup items, and generally prepare the system for the user. While many of the same scripts and daemons are still run, the `mach_init` and `init` processes have been replaced by `launchd` in Mac OS X v10.4 and later. This change means that `launchd` is now the root system process.

In addition to initializing the system, the `launchd` process coordinates the launching of system daemons in an orderly manner. Like the `inetd` process, `launchd` launches daemons on-demand. Daemons launched in this manner can shut down during periods of inactivity and be relaunched as needed. (When a subsequent service request comes in, `launchd` automatically relaunches the daemon to process the request.)

This technique frees up memory and other resources associated with the daemon, which is worthwhile if the daemon is likely to be idle for extended periods of time. More importantly, however, this guarantees that runtime dependencies between daemons are satisfied without the need for manual lists of dependencies.

Next, `launchd` starts `SystemStarter`, which starts any non-launch-on-demand daemons.

Note: While `launchd` does support non-launch-on-demand daemons, this use is not recommended. The `launchd` daemon was designed to remove the need for dependency ordering among daemons. If you do not make your daemon be launch-on-demand, you will have to handle these dependencies in another way, such as by using the legacy startup item mechanism.

For more information about launch-on-demand and `SystemStarter` daemons and how to launch them, see "Daemons" (page 17).

As the final part of system initialization, `launchd` launches `loginwindow`. The `loginwindow` program controls several aspects of user sessions and coordinates the display of the login window and the authentication of users.

Note: By default, Mac OS X boots with a graphical boot screen. For debugging the boot process, it is often useful to disable this, revealing the text console underneath. This mode is known as verbose boot mode. To enable verbose boot mode, simply hold down command-v after the boot chime.

Authenticating Users

Mac OS X requires users to authenticate themselves prior to accessing the system. The `loginwindow` program coordinates the visual portion of the login process (as manifested by the window where users enter name and password information) and the security portion (which handles the user authentication). Once a user is authenticated by the security systems, `loginwindow` begins setting up the user environment.

In two key situations, `loginwindow` bypasses the usual login prompt and begins the user session immediately. The first situation occurs when the system administrator has configured the computer to automatically log in as a specified user. The second occurs during software installation when the installer program is to be launched immediately after a reboot.

Configuring User Sessions

Immediately after the user is successfully authenticated, `loginwindow` sets up the user environment and records information about the login. As part of this process, it performs the following tasks:

- Secures the login session from unauthorized remote access. Applications that are launched remotely are not registered with the pasteboard server's (Clipboard's) port. As a result, some standard features are blocked for these processes, including copy, cut, paste, Apple events, window minimization, and other services.
- Records the login in the system's `utmp` database.
- Sets the owner and permissions for the console terminal.
- Resets the user's preferences to include global system defaults.
- Registers the pasteboard server (`pbs`) with the bootstrap port and launches `pbs`.
- Configures the mouse, keyboard, and system sound using the user's preferences.

- Sets the user's group permissions (`gid`).
- Retrieves the user record from Directory Services and applies that information to the session.
- It loads the user's computing environment (including preferences, environment variables, device and file permissions, keychain access, and so on).
- It launches the Dock, Finder, and SystemUIServer.
- It automatically launches applications specified in the Login Items pane of the Accounts System Preferences for the user.

Once the user session is up and running, `loginwindow` monitors the session and user applications in the following ways:

- It manages the logout, restart, and shutdown procedures. See "Logout Responsibilities" (page 12) for more information.
- It manages the Force Quit window, which includes monitoring the currently active applications and responding to user requests to force-quit applications and relaunch the Finder. (Users open this window from the Apple menu or by pressing Command-Option-Escape.)
- It writes any standard-error (`stderr`) output to a log file. Log files are stored in the `/Library/Logs/Console/`<*uid*>`/console.log` file, where <*uid*> is the user ID of the currently logged in user.

If the Finder, Dock, or SystemUIServer processes die for some reason, `loginwindow` automatically restarts them. In the same manner, if the `loginwindow` process dies, the `launchd` process automatically restarts it.

Logout Responsibilities

The procedures for logging out, restarting the system, or shutting down the system have similar semantics. The foreground process usually initiates these procedures in response to the user choosing an item from the Apple menu; however, a process can also initiate the procedure programmatically by sending an appropriate Apple event to `loginwindow`. The `loginwindow` program carries out the procedure, posting alerts and notifying applications to give them a chance to clean up before closing.

A typical logout/restart/shutdown procedure is as follows:

1. The user selects Log Out, Restart, or Shut Down from the Apple menu.
2. The foreground application initiates the user request by sending an Apple event to `loginwindow`. (See "Application Responsibilities" (page 13) for a list of events.)
3. The `loginwindow` program displays an alert to the user asking for confirmation of the action.
4. If the user confirms the action, `loginwindow` sends a Quit Application Apple event (`kAEQuitApplication`) to every foreground and background user process.
5. Once all processes have quit, `loginwindow` closes out the user session and continues with the action.

- For a logout action, `loginwindow` dequeues all events in the event queue, starts the logout-hook program (if one is defined), records the logout, resets device permissions and user preferences to their defaults, and quits. It is subsequently relaunched by `launchd` to handle a new login. (See "Customizing Login and Logout" (page 45) for more on `loginwindow` hooks.)
- For a restart action, `loginwindow` sets the device permissions and user preferences to their defaults and then restarts the system.
- For a shutdown action, `loginwindow` powers off the system.

Foreground processes can choose not to terminate when they receive the Quit Application event. See "Terminating Processes" (page 14) for more information.

Application Responsibilities

To initiate a logout, restart, or shutdown sequence programmatically, the foreground application must send an appropriate Apple event to `loginwindow`. Upon receipt of the event, `loginwindow` begins the process of shutting down the user session. Depending on the Apple event sent by the process, `loginwindow` may or may not post an alert dialog and give the user a chance to abort the sequence.

The following list shows the preferred Apple events for logout, restart, and shutdown procedures. These events have no required parameters.

- `kAELogOut`
- `kAEShowRestartDialog`
- `kAEShowShutdownDialog`

Upon receipt of one of these events, `loginwindow` displays an alert notifying the user of the impending action. At this point, the user may continue with the action or abort it. If the user continues with the action, `loginwindow` sends an Apple event to each application asking it to quit. See "Terminating Processes" (page 14).

In addition to the preferred Apple events, there are two additional events that tell `loginwindow` to proceed immediately with a restart or shutdown sequence:

- `kAERestart`
- `kAEShutDown`

These events proceed with the corresponding sequence without posting an alert dialog to the user. Thus, if you send one of these events to `loginwindow`, the user does not have an opportunity to abort the sequence. These events should be used sparingly, if at all.

Important: Note that if a logout, restart, or shutdown event originates from an application in the Classic environment, the event affects only the Classic environment and its applications. The rest of the user session continues running.

Terminating Processes

As part of a log out, restart, or shutdown sequence, `loginwindow` attempts to terminate all foreground and background user processes. It sends each process a Quit Application Apple event (`kAEQuitApplication`), as a courtesy, to give each process a chance to shut itself down gracefully. For foreground processes, `loginwindow` sends the event and waits for a reply. For background processes, `loginwindow` sends the event but does not wait for a reply. It terminates any lingering background processes by sending a `kill` command.

When a foreground process receives the Quit Application Apple event from `loginwindow`, it should terminate itself immediately or post an alert dialog if a user decision is required first (such as when there is an unsaved document); when that condition is resolved the application should then terminate. If the user decides to abort the termination sequence (by clicking Cancel in a Save dialog, for example) the application should respond to the event by returning a `userCancelledErr` error.

Note: Cocoa applications do not see the `kAEQuitApplication` event directly. The Application Kit notifies your application by calling its `applicationShouldTerminate:` delegate method. To abort the termination sequence, implement this method and return NSTerminateCancel; otherwise, termination of your application continues normally.

If a foreground application fails to reply or terminate itself after 45 seconds, `loginwindow` automatically aborts the termination sequence. This safeguard is to protect data in various situations, such as when an application is saving a large file to disk and is unable to terminate in the allotted time. If a foreground application is unresponsive and not doing anything, the user must use the Force Quit window to kill it before proceeding.

For user background processes that link with Carbon, Cocoa, or Java, the procedure is a little different. The `loginwindow` program notifies the process that it is about to be terminated by sending it a Quit Application Apple event (`kAEQuitApplication`) as before. Unlike foreground processes, however, `loginwindow` does not wait for a reply. It proceeds to terminate any open background processes, regardless of any returned errors.

During a user logout, `loginwindow` does not terminate processes residing in the root context. These processes reside outside the context of the user session and are terminated only during a restart or shutdown sequence. The `loginwindow` program also does not kill background processes that are independent of Carbon, Cocoa, or Java, even if they are launched from the user context. (Though launched from a user context, these processes are taken over by the system when the user logs out.) Mac OS X does not send any notifications to system processes before terminating them.

Identifying the Scope of Processes

Although the `launchd` process owns every other process on the system, a distinction can still be made between user and system processes. Startup items, daemonized processes, and any processes run prior to `loginwindow` typically run in the root context. Processes in this context provide services to all users of the system.

Processes that run within the context of an authenticated user session are *user processes*. User processes are always associated with a particular user session and are usually children of the `WindowServer` or `loginwindow` processes associated with the user login.

Note: Not all user processes are children of the `WindowServer` process. Processes launched as root, and some special system processes, are owned by the user but are children of the `launchd` process. You can use the ActivityMonitor application to determine the owner and parent of any process on the system.

The Shutdown Process

At shutdown, Mac OS X first executes the service stop routines in any `SystemStarter` startup items such as those described in "Creating a Startup Item" (page 35).

Next, as with most UNIX-based and UNIX-like operating systems, Mac OS X sends a `SIGTERM` signal to all running processes prior to shutdown. Upon receiving this signal, your daemon should quickly make an orderly shutdown.

Every reasonable attempt will be made to wait for it to exit, but it is ultimately the responsibility of your daemon to keep the amount of unsaved state information to a level that can be reasonably written to disk in the time allotted.

A few seconds later, your daemon will receive a `SIGKILL` signal and will be terminated.

Daemons

By the time a user logs in to a Mac OS X system, a number of processes are already running. Many of these processes are known as daemons. A **daemon** is a background process that provides a service to the users of the system. For example, the `cupsd` daemon coordinates printing requests while the `httpd` daemon responds to requests for web pages.

Most daemons run in the root context of the system—that is, they run at the lowest level of the system and make their services available to all user sessions. Daemons at this level continue running even when no users are logged into the system. Because of this fact, the daemon program should have no direct knowledge of users. Instead, the daemon must wait for a user program to contact it and make a request. As part of that request, the user program usually tells the daemon how to return any results.

Note: For more information about the root context and user sessions, see *Multiple User Environments*.

Communicating With Daemons

Applications communicate transparently with built-in system daemons most of the time. For example, when you use the CFNetwork API to open a network connection, the API itself may communicate with several daemons to process your request. For the most part, your program does not need to worry about the details of this communication.

Note: It is possible to communicate with many system daemons directly if needed. Most system daemons are part of the Darwin layer of Mac OS X and have well-documented communication protocols. In most cases, though, it is easier, more practical, and more reliable to let the Mac OS X libraries and frameworks handle this communication for you.

When creating a custom daemon, however, you must define the protocol for receiving requests and returning the results. To simplify matters for clients who might use your daemon, you should define your own library of routines for initiating requests and receiving results.

For example, if you created a daemon to manage database transactions, you would likely also create a library of functions for communicating with that daemon. Client applications linking to your library would then talk transparently to your daemon without the need to understand the communication mechanism itself.

There are three major classes of communication mechanisms commonly used between daemons and their clients: traditional client-server communications (including Apple events, TCP/IP, UDP, other socket and pipe mechanisms, and Mach IPC), remote procedure calls (including Mach RPC, Sun RPC, and Distributed Objects), and memory mapping (used underneath the Core Graphics APIs, among others).

In general, you should use a traditional client-server communication API. Code based on these APIs tends to be easier to understand and maintain than RPC or memory mapping designs. It also tends to be more portable to other platforms.

If both your client and daemon are written in Cocoa, and if most of your communication involves sending a message and expecting a reply, you should consider Distributed Objects, which is an RPC mechanism.

Because memory mapping requires more complex management (and represents a security risk if you are not careful about what memory pages you share), you should use memory mapping only if your client and daemon require a large amount of shared state with low latency, such as passing audio or photo data in a real-time fashion.

The details of using these communication mechanisms are outside the scope of this document, but you can find documentation elsewhere in the ADC Reference Library to help you with the details. Some related documents include:

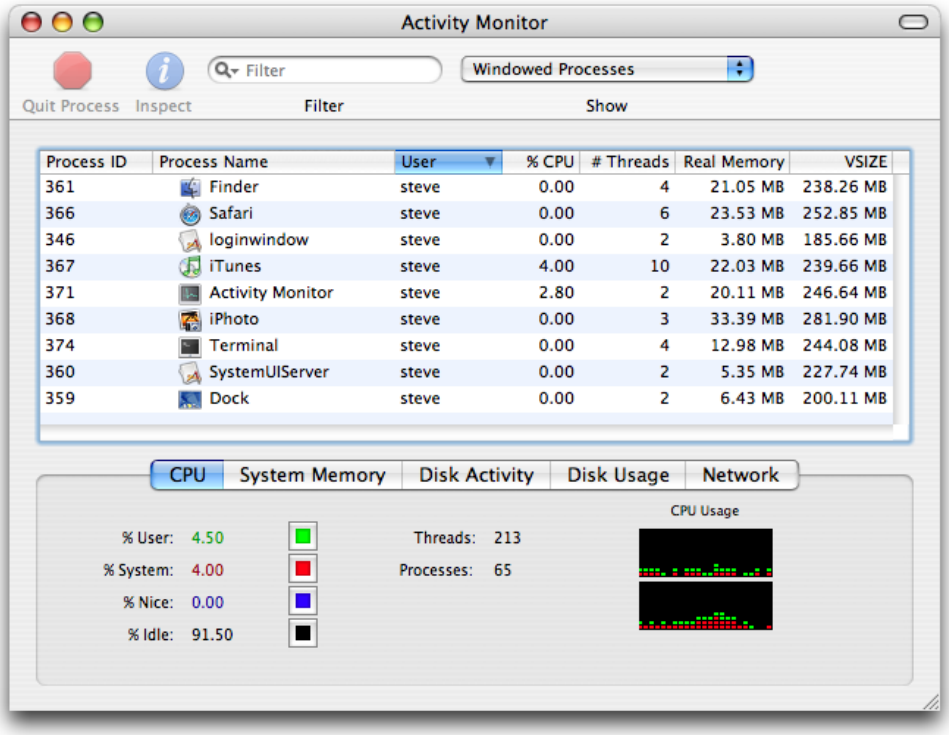
- *Apple Events Programming Guide*
- *Distributed Objects Programming Topics*
- *SharedMemory*
- *CFSocket Reference*
- *NSSocketPort Class Reference*
- UNIX Socket FAQ

Viewing the Currently Running Daemons

If you want to see the daemons currently running on your system, use the Activity Monitor application (located in `/Applications/Utilities`). This application lets you view information about all processes including their resource usage. Figure 1 (page 19) shows the Activity Monitor window and the process information.

Note: If you want to know more about the services provided by a particular daemon, consult the `man` page for that daemon. You can also view the manual pages online by reading *Mac OS X Man Pages*.

Figure 1 Processes shown in Activity Monitor



Running a Process on a Schedule

Sometimes you need to run a daemon or other background process on a timed schedule. In Mac OS X, you can do this in four ways: `cron` jobs, `at` jobs, `launchd` timed jobs, and `periodic` jobs. This section explains these methods briefly and provides links to manual pages that provide additional details.

Timed Jobs Using Periodic Jobs

If your timed job just needs to be run periodically, the easiest way to make this happen is to create a `periodic` job. To create a `periodic` job, you simply create a shell script that executes your program with the desired flags and place it in the daily, weekly, or monthly directory in `/etc/periodic`.

Mac OS X runs these jobs in different ways depending on the version of Mac OS X. In Mac OS X v10.3 and earlier, `cron` is responsible for starting periodic jobs. In v10.4 and later, `launchd` starts the jobs.

As a result, if your computer is asleep at the scheduled time, in Mac OS X v10.3 and earlier, the job does not run; in Mac OS X v10.4 and later, the job executes automatically when the computer wakes up. For this reason, you should not assume that a job will run at a particular time or on a particular day.

Note: If the computer is turned off at the scheduled time, the jobs does not run at all, regardless of what version of Mac OS X you are using.

You can configure the `periodic` tool by modifying `/etc/defaults/periodic.conf`. For more information, see the manual page for `periodic.conf`.

Timed Jobs Using cron

The `cron` daemon is the most common tool for executing a job at a particular time. Although all versions of Mac OS X support `cron` jobs, beginning in v10.4, the functionality of `cron` is largely superseded by `launchd`, and all periodic jobs built into Mac OS X are `launchd` jobs.

A `cron` job is not guaranteed to run. If the system is turned off or asleep at the designated time, the job does not execute until the designated time occurs again.

Systemwide `cron` jobs can be installed by modifying `/etc/crontab`. Per-user `cron` jobs can be installed using the `crontab` tool. The format of these `crontab` files is described in the man page for the `crontab` file format.

Because installing `cron` jobs requires modifying a shared resource (the `crontab` file), you should not programmatically add a `cron` job. However, `cron` is a very important tool for system administrators and power users.

Timed Jobs Using launchd

Beginning in Mac OS X v10.4, the preferred way to add a timed job is to use a `launchd` timed job. A `launchd` timed job is similar to a `cron` job, with two key differences:

- Each `launchd` job is described by a separate file. This means that you can add `launchd` timed jobs by simply adding or removing a file.
- If the computer is asleep at the designated time, a `launchd` job executes as soon as the computer wakes. This is similar to the behavior of `anacron` and other `cron` replacements).

To create a `launchd` timed job, you should create a configuration property list file similar to those described in "Creating a Launchd Property List File" (page 30) except that, instead of including an `OnDemand` key with a value of `true`, you specify a `StartCalendarInterval` key containing a dictionary of time values.

For example, the following property list runs the program `happybirthday` at midnight every time July 11 falls on a Sunday.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC -//Apple Computer//DTD PLIST 1.0//EN
http://www.apple.com/DTDs/PropertyList-1.0.dtd >
<plist version="1.0">
<dict>
     <key>Label</key>
     <string>com.example.happybirthday</string>
     <key>ProgramArguments</key>
     <array>
          <string>happybirthday</string>
```

```
        </array>
        <key>OnDemand</key>
        <false/>
        <key>StartCalendarInterval</key>
        <dict>
            <key>Hour</key>
            <integer>00</integer>
            <key>Minute</key>
            <integer>00</integer>
            <key>Month</key>
            <integer>7</integer>
            <key>Day</key>
            <integer>11</integer>
            <key>Weekday</key>
            <integer>0</integer>
        </dict>
    </dict>
    </plist>
```

For more information on these values, see the manual page for `launchd.plist`.

Timed Jobs Using at

The `at` daemon is disabled by default in Mac OS X to reduce power consumption. If you need to enable it, you can learn how in the manual page for `at`.

Guidelines for Creating and Launching Daemons

Mac OS X comes with several daemons that provide most of the basic system services. Examples of these services include handling network lookup requests, serving web pages, monitoring hardware devices, and gathering metadata from the file system. The clients of these services may be the operating system, client applications, or both. An example of a daemon used by the system is the `mds` daemon, which monitors the file system and initiates the gathering of metadata. The system gathers the metadata and puts it in a central repository, which client applications can then access.

Although the general steps for how to create a new daemon are beyond the scope of this document, there are some things that daemon writers need to understand before writing daemons for Mac OS X.

When Is a Custom Daemon Appropriate?

Most application developers will never need to create a daemon directly. Even those developers that need some sort of background server may not find a daemon to be the best choice in all cases.

Daemons run in the root context, which means they are unaware of the users logged on to the system. A daemon cannot initiate contact with a user process directly, although it can respond to requests made by user processes. Because they have no knowledge of users, daemons also have no access to the window server, and thus no ability to post a visual interface. Daemons are strictly background processes that respond to low-level requests.

If you need to provide user-specific services, you should create an **agent** instead of a daemon. An agent is essentially the same thing as a daemon, except that it runs in the context of a user session. Agents can communicate with other processes in the same user session and with system-wide daemons in the root context. Because they have access to the window server, agents can also post a user interface, although they should do so sparingly, if at all. Like daemons, agents should be launched automatically.

The only difference between a daemon and an agent is location: daemons are installed in `/Library/LaunchDaemons`, while agents are installed in `/Library/LaunchAgents` or in the `LaunchAgents` subdirectory of an individual user's `Library` directory.

If you find that your code provides both user-specific and user-independent services, you might want to create both a daemon and an agent. Your daemon would run in the root context and provide the user-independent services while a copy of your agent would run in each user session. The agents would then coordinate with the daemon to provide the services to each user.

For more information about the root context and user sessions, see *Multiple User Environments*.

Launching Daemons

Mac OS X provides two methods for launching daemons: startup items and `launchd` daemons. Which one you use depends largely on the versions of Mac OS X that the daemon must support.

- **Mac OS X v10.3 and earlier:** You `must` use startup items. The `launchd` service is not supported prior to v10.4.
- **Mac OS X v10.4 and later:** You can either use startup items or `launchd` daemons. Using `launchd` daemons is preferred unless you also require backwards compatibility with versions of Mac OS X prior to v10.4.

Of course, you do not necessarily have to choose one or the other. For optimal compatibility and performance, you could use both. The key is to add a command-line argument or arguments to enable or disable launchd-compliant behavior, such as launch-on-demand support.

Once you have the ability to launch your daemon in either form, you can install both a `launchd` property list and a startup item. To avoid launching your daemon twice, be sure to add code to the startup item to disable it in Mac OS X v10.4 or later. For example, the following will print "10.3 or earlier" if it is running on a version of Mac OS X prior to v10.4:

Listing 1 Conditional Startup Item Execution

```
#!/bin/sh

OSVERSION="$(sw_vers -productVersion)"
MAJOR="$(echo $OSVERSION | sed 's/\..*//')"
MINOR="$(echo $OSVERSION | sed -E 's/[0-9]+\.([0-9]+)\..*/\1/')"
PATCH="$(echo $OSVERSION | sed -E 's/[0-9]+\.[0-9]+\.([0-9]+).*/\1/')"

echo "MAJOR: $MAJOR"
echo "MINOR: $MINOR"
echo "PATCH: $PATCH"

if [ $MAJOR -eq 10 ] ; then
        if [ $MINOR -le 3 ] ; then
                echo "10.3 or earlier";
        fi
fi
```

For more information about shell scripting, read *Shell Scripting Primer*.

Launching Custom Daemons Using launchd

With the introduction of `launchd` in Mac OS X v10.4, an effort was made to improve the steps needed to launch and maintain daemons. Prior to 10.4, if you wanted to launch a custom daemon, you had to create a startup item to do so. During boot up, the system would execute your startup item, allowing you to run a script that launched your daemon. Once launched, your daemon would continue running (and continue holding on to memory and resources) until the computer was shut down or restarted, the daemon was manually shut down, or the daemon itself crashed.

What `launchd` does is provide a harness for launching and relaunching your daemon as needed. To client programs, the port representing your daemon's service is always available and ready to handle requests. In reality, the daemon may or may not be running. So, when a client sends a request to the port, `launchd` may have to launch the daemon so that it can handle the request. Once launched, the daemon can continue running or shut itself down to free up the memory and resources it holds. If a daemon shuts itself down, `launchd` once again relaunches it as needed to process requests.

In addition to the launch-on-demand feature, `launchd` provides the following benefits to daemon developers:

- Simplifies the daemonization process by handling many of the standard housekeeping chores normally associated with launching a daemon.
- Provides system administrators with a central place to manage daemons on the system.
- Supports `inetd`-style daemons.
- Eliminates the primary reason for running daemons as root. Because launchd runs as root, it can create low-numbered TCP/IP listen sockets and hand them off to the daemon.
- Simplifies error handling and dependency management for inter-daemon communication. Because daemons launch on demand, communication requests do not fail if the daemon is not launched. They are simply delayed until the daemon can launch and process them.

For more information on how to create a launch-on-demand daemon, see "Creating launchd Daemons and Agents" (page 27).

Launching Daemons with Startup Items

If your software includes a custom daemon and must support versions of Mac OS X prior to 10.4, use a startup item to launch the daemon. A startup item is a bundled shell script or executable binary that runs once when the computer first boots (see "The Boot Process" (page 9)).

If you have custom startup items, you should install them in the `/Library/StartupItems` directory. Apple startup items are located in the `/System/Library/StartupItems` directory, although most of them have been stubbed out in Mac OS X v10.4 and later and replaced by `launchd`-compliant versions. The stubbed out versions remain for the benefit of other startup items that have dependencies on them.

For information on how to create a startup item, see "Creating a Startup Item" (page 35).

Creating launchd Daemons and Agents

If you are developing daemons to run on Mac OS X version 10.4 and later, it is highly recommended that you design your daemons to be `launchd`-compliant. Using `launchd` provides better performance and flexibility for daemons. It also improves the ability of administrators to manage the daemons running on a given system.

If you are running per-user background processes for Mac OS X v10.4 and later, `launchd` is also the preferred way to start these processes. These per-user processes are referred to as user agents. A user agent is essentially identical to a daemon, but is specific to a given logged-in user and executes only while that user is logged in.

Unless otherwise noted, for the purposes of this chapter, the terms "daemon" and "agent" can be used interchangeably. Thus, the term "daemon" is used generically in this section to encompass both system-level daemons and user agents except where otherwise noted.

There are four ways to launch daemons using `launchd`. The preferred method is on-demand launching, but `launchd` can launch daemons that run continuously, and can replace `inetd` for launching `inetd`-style daemons. This chapter describes these three methods.

In addition, `launchd` can start jobs at timed intervals much like `cron` jobs. This is described in "Timed Jobs Using launchd" (page 20).

The launchd Startup Process

After the system is booted and the kernel is running, `launchd` is run to finish the system initialization. As part of that initialization, it goes through the following steps:

1. It loads the parameters for each launch-on-demand system-level daemon from the property list files found in `/System/Library/LaunchDaemons/` and `/Library/LaunchDaemons/`.
2. It registers the sockets and file descriptors requested by those daemons.
3. It launches any daemons that requested to be running all the time.
4. As requests for a particular service arrive, it launches the corresponding daemon and passes the request to it.
5. When the system shuts down, it sends a `SIGTERM` signal to all of the daemons that it started.

The process for per-user agents is similar. When a user logs in, a per-user `launchd` is started. It does the following:

1. It loads the parameters for each launch-on-demand user agent from the property list files found in `/System/Library/LaunchAgents`, `/Library/LaunchAgents`, and the user's individual `Library/LaunchAgents` directory.

2. It registers the sockets and file descriptors requested by those daemons.
3. It launches any daemons that requested to be running all the time.
4. As requests for a particular service arrive, it launches the corresponding daemon and passes the request to it.
5. When the user logs out, it sends a `SIGTERM` signal to all of the user agents that it started.

Because `launchd` registers the sockets and file descriptors used by all daemons before it launches any of them, daemons can be launched in any order. If a request comes in for a daemon that is not yet running, the requesting process is suspended until the target daemon finishes launching and responds.

If a daemon does not receive any requests over a specific period of time, it can choose to shut itself down and release the resources it holds. When this happens, `launchd` monitors the shutdown and makes a note to launch the daemon again when future requests arrive.

Important: If your daemon shuts down too quickly after being launched, `launchd` may think it has crashed. Daemons that continue this behavior may be suspended and not launched again when future requests arrive. To avoid this behavior, do not shut down for at least 10 seconds after launch.

Daemon Requirements

If you are looking to implement a daemon that supports `launchd`, there are some behaviors with which you should be familiar. Creating daemons to run under `launchd` is actually simpler than in previous versions of Mac OS X. The reason is that many tasks normally implemented in your daemon code are now handled automatically by `launchd`.

Note: The following sections describe only the changes you must make to your daemons to support `launchd`. The overall process for creating daemons is not covered.

Required Behaviors

To support `launchd`, you must obey the following guidelines when writing your daemon code:

- You must provide a property list with some basic launch-on-demand criteria for your daemon. See "Creating a Launchd Property List File" (page 30).
- You must not fork your process and have the parent process exit.
- You must not daemonize your process. This includes calling the `daemon` function, calling `fork` followed by `exec`, or calling `fork` followed by `exit`. If you do, `launchd` thinks your process has died. Depending on your property list key settings, `launchd` will either keep trying to relaunch your process until it gives up (with a "respawning too fast" error message) or will be unable to restart it if it really does die.
- Your daemon and its property list file must be owned by the root user (except for agents, which may be owned by the logged-in user), and must not be group writable or other writable.

Although they are normally part of the daemon creation process, it is worth emphasizing that forking and exiting the parent process and calling the `daemon` function must be avoided if you want to support `launchd`. The `launchd` program configures your daemon to run as a daemon before your code is ever called, so these steps are unnecessary. In addition, calling them interferes with the ability of `launchd` to launch your daemon on demand.

Recommended Behaviors

To support `launchd`, it is recommended that you obey the following guidelines when writing your daemon code:

- Wait until your daemon is fully initialized before attempting to process requests. Your daemon should always provide a reasonable response (as opposed to an error) when processing requests.
- Register the sockets and file descriptors used by your daemon in your `launchd` configuration property list file.
- Check in with `launchd` as part of your daemon initialization using the routines in `launch.h`.
- During checkin, get the launch dictionary from `launchd`, extract its contents, store those contents locally, and get rid of the dictionary. Caching the dictionary locally and accessing it frequently could hurt performance.
- Provide a handler to catch the `SIGTERM` signal.

In addition to the preceding list, the following is a list of things it is recommended you do *not* do in your code:

- Do not set the user or group ID for your daemon. Include the `UserName`, `UID`, `GroupName`, or `GID` keys in your daemon's configuration property list instead.
- Do not set the working directory. Include the `WorkingDirectory` key in your daemon's configuration property list instead.
- Do not call `chroot` to change the root directory. Include the `RootDirectory` key in your daemon's configuration property list instead.
- Do not call `setsid` to create a new session.
- Do not close any stray file descriptors.
- Do not change `stdio` to point to `/dev/null`. Include the `StandardOutPath` or `StandardErrorPath` keys in your daemon's configuration property list file instead.
- Do not set up resource limits with `setrusage`.
- Do not set the daemon priority with `setpriority`

Although many of the preceding behaviors may be standard tasks for daemons to perform, they are not recommended when running under `launchd`. The reason is that `launchd` configures the operating environment for the daemons that it manages. Changing this environment could interfere with the normal operation of your daemon.

Creating a Launchd Property List File

To run under `launchd`, you must provide a configuration property list file for your daemon. This file contains information about your daemon, including the list of sockets or file descriptors it uses to process requests. Specifying this information in a property list file lets `launchd` register the corresponding file descriptors and launch your daemon only after a request arrives for your daemon's services. Table 1 lists the required and recommended keys for all daemons.

Table 1 Required and recommended property list keys

Key	Description
`Label`	Contains a unique string that identifies your daemon to `launchd`. This key is required.
`ProgramArguments`	Contains the arguments used to launch your daemon. This key is required.
`inetdCompatibility`	Indicates that your daemon requires a separate instance per incoming connection. This causes `launchd` to behave like `inetd`, passing each daemon a single socket that is already connected to the incoming client. This key is required if your daemon was designed to be launched by `inetd`; otherwise, it must not be included.
`OnDemand`	This key specifies whether your daemon launches on-demand or must always be running. This key is recommended.

Depending on the needs of your daemon, you would also include other keys in your configuration property list file. For example, if your daemon monitors a well-known port (those listed in `/etc/services`), you would add a `Sockets` entry that looks like this:

```
<key>Sockets</key>
<dict>
    <key>Listeners</key>
    <dict>
        <key>SockServiceName</key>
        <string>bootps</string>
        <key>SockType</key>
        <string>dgram</string>
        <key>SockFamily</key>
        <string>IPv4</string>
    </dict>
</dict>
```

Note that the string for `SockServiceName` typically comes from the leftmost column in `/etc/services`. The `SockType` is one of `dgram` (UDP) or `stream` (TCP/IP).

If you need to pass a port number that is not listed in the well-known ports list, the format is basically the same, except the string contains a number instead of a name. For example:

```
<key>SockServiceName</key>
<string>23</string>
```

You can also pass additional keys to further configure your daemon. For a list of sample configuration property lists, look at the files in `/System/Library/LaunchDaemons/`. These files are used to configure many daemons that run on Mac OS X.

For additional information about the keys you can specify in your configuration property list file, see the `man` page for `launchd.plist`.

Deciding When to Shut Down

If you do not expect your daemon to handle many requests, you might want to shut it down after a predetermined amount of idle time, rather than continue running. Although a well-written daemon does not consume any CPU resources, it still consumes memory and could be paged out during periods of intense memory use.

The timing of when to shut down is different for each daemon and depends on several factors, including:

- The number and frequency of requests it receives
- The time it takes to launch the daemon
- The time it takes to shut down the daemon
- The need to retain state information

If your daemon does not receive frequent requests and can be launched and shut down quickly, you might prefer to shut it down rather than wait for it to be paged out to disk. Paging memory to disk, and subsequently reading it back, incurs two disk operations. If you do not need the data stored in memory, your daemon can shut down and avoid the step of writing memory to disk.

Special Dependencies

While `launchd` takes care of dependencies between daemons, in some cases, your daemon may depend on other system functionality that cannot be addressed in this manner. This section describes many of these special cases and how to handle them.

Disk or server availability:

If your daemon depends on the availability of a mounted volume (whether local or remote), you can determine the status of that volume using the disk arbitration framework. This is documented in *Disk Arbitration Framework Reference*.

Kernel Extensions

If your daemon requires that a certain kernel extension be loaded prior to executing, you have two options: load it yourself, or wait for it to be loaded.

Load it yourself:

The daemon may manually request that an extension be loaded. To do this, run `kextload` with the appropriate arguments using `exec` or variants thereof.

Note: The `kextload` executable must be run as root in order to load extensions into the kernel. For security reasons, it is not a setuid executable. This means that your daemon must either be running as the root user or must include a helper binary that is setuid root in order to use `kextload` to load a kernel extension.

Wait for a matching service:

Your daemon may wait for a kernel service to be available. To do this, you should first register for service change notification using the functions described in `IOKitLib`. This header is part of the I/O Kit Framework, which is further documented in *I/O Kit Framework Reference*.

After registering for these notifications, you should check to see if the service is already available. By doing this after registering for notifications, you avoid waiting forever if the service becomes available between checking for availability and registering for the notification.

Note: In order for your kernel extension to be detected in a useful way, it must publish a node in the I/O registry to advertise the availability of its service. For I/O Kit drivers, this is usually handled by the I/O Kit family.

For other kernel extensions, you must explicitly register the service by publishing a nub, which must be an instance of `IOService`.

For more information about I/O Kit services and matching, see *I/O Kit Fundamentals*, *I/O Kit Framework Reference* (user space reference), and *Kernel Framework Reference* (kernel space reference).

Network availability:

If your daemon depends on the network being available with a valid outgoing route, this cannot be handled with dependencies because network interfaces can come and go at any time in Mac OS X. To solve this problem, you should use the network reachability functionality in the system configuration framework. This is documented in *System Configuration Programming Guidelines* and *System Configuration Framework Reference*.

Non-launchd daemons:

If your daemon has a dependency on a non-launchd daemon, you must take additional care to ensure that your daemon works correctly if that non-launchd daemon has not started when your daemon is started. The best way to do this is to include a loop at start time that checks to see if the non-launchd daemon is running, and if not, sleeps for several seconds before checking again.

Be sure to set up handlers for `SIGTERM` prior to this loop to ensure that you are able to properly shut down if the daemon you rely on never becomes available.

User login:

In general, a daemon should not care whether a user is logged in, and user agents should be used to provide per-user functionality. However, in some cases, this may be useful.

To determine what user is logged in at the console, you can use the system configuration framework to register for login and logout notifications, as described in Technical Q&A QA1133.

Non-Launch-on-Demand Daemons

While most daemons should generally be run on demand, it can sometimes be useful to run a daemon continuously.

In Mac OS X v10.3 and earlier, the preferred facility for launching a daemon was a startup item. If you need to support versions of Mac OS X prior to v10.4, you should use a startup item. This is described in "Creating a Startup Item" (page 35).

In Mac OS X v10.4 and later, the preferred facility for launching daemons is `launchd`. To create a non-on-demand `launchd` daemon, you need to add some additional keys to your daemon's launchd property list file:

- **OnDemand**—set this to `false`. This will tell `launchd` that your daemon is not designed for on-demand launching.
- **RunAtLoad**—set this to `true`. This will cause your daemon to be launched as soon as launchd starts. Because of the nature of on-demand facilities for things like networking, any dependencies will automatically be launched as needed to support your daemon.

Important: Before using `launchd` to launch any daemon (on-demand or otherwise), you should read about the requirements for a `launchd` daemon in "Daemon Requirements" (page 28).

In particular, your daemon must *not* daemonize itself (`fork` and exit). If it does, `launchd` will interpret this as a failed launch and will repeatedly try to respawn it before eventually giving up.

For More Information

The manual pages for `launchd` and `launchd.plist` are the two best sources for information about `launchd`.

In addition, you can find a source daemon accompanying the `launchd` source code (available from http://www.macosforge.org/). This daemon is also provided from the ADC reference library as the SampleD sample code project.

Finally, many Apple-provided daemons support `launchd`. Their property list files can be found in `/System/Library/LaunchDaemons`. Some of these daemons are also available as open source from http://www.opensource.apple.com/ or http://www.macosforge.org/.

Creating a Startup Item

A startup item is a specialized bundle whose code is executed during the final phase of the boot process, and at other predetermined times (see "Managing Startup Items" (page 39)). The startup item typically contains a shell script or other executable file along with configuration information used by the system to determine the execution order for all startup items.

The `/System/Library/StartupItems` directory is reserved for startup items that ship with Mac OS X. All other startup items should be placed in the `/Library/StartupItems` directory. Note that this directory does not exist by default and may need to be created during installation of the startup item.

Note: The `launchd` facility is he preferred mechanism for launching daemons in Mac OS X v10.4 and higher. Unless your software requires compatibility with Mac OS X v10.3 or earlier, you should use the `launchd` facility instead of writing a startup item. For more information, see "Guidelines for Creating and Launching Daemons" (page 23).

Important: A startup item is *not* an application. You cannot display windows or do anything else that you can't do while logged in via ssh. If you want to launch an application after login, you should install a login item instead. For more information, see "Customizing Login and Logout" (page 45).

Anatomy of a Startup Item

Unlike many other bundled structures, a startup item does not appear as an opaque file in the Finder. A startup item is a directory whose executable and configuration property list reside in the top-level directory. The name of the startup item executable must match the name of the startup item itself. The name of the configuration property list is always `StartupParameters.plist`. Depending on your needs, you may also include other files in your startup item bundle directory.

```
MyStartupItem/
    MyStartupItem
    StartupParameters.plist
```

To create your startup item:

1. Create the startup item directory. The directory name should correspond to the behavior you're providing.

   Example: `MyDBServer`

2. Add your executable to the directory. The name of your executable should be exactly the same as the directory name. For more information, see "Creating the Startup Item Executable" (page 36).

3. Create a property list with the name `StartupParameters.plist` and add it to the directory. For information on the keys to include in this property list, see "Specifying the Startup Item Properties " (page 37).

Example: `MyDBServer/StartupParameters.plist`

4. Create an installer to place your startup item in the `/Library/StartupItems` directory of the target system. (Your installer may need to create this directory first.)

   Your installer script should set the permissions of the startup item directory to prevent non-root users from modifying the startup item or its contents. For more information, see "Startup Item Permissions" (page 40).

Creating the Startup Item Executable

The startup item executable can be a binary executable file or an executable shell script. Shell scripts are more commonly used because they are easier to create and modify.

If you are implementing your startup-item executable as a shell script, Mac OS X provides some code to simplify the process of creating your script. The file `/etc/rc.common` defines routines for processing command-line arguments and for gathering system settings. In your shell script, source the `rc.common` file and call the `RunService` routine, passing it the first command-line argument, as shown in the following example:

```
#!/bin/sh
. /etc/rc.common

# The start subroutine
StartService() {
    # Insert your start command below.  For example:
    mydaemon -e -i -e -i -o
    # End example.
}

# The stop subroutine
StopService() {
    # Insert your stop command(s) below.  For example:
    killall -TERM mydaemon
    sleep 10
    killall -9 mydaemon
    # End example.
}

# The restart subroutine
RestartService() {
    # Insert your start command below.  For example:
    killall -HUP mydaemon
    # End example.
}

RunService "$1"
```

The `RunService` routine looks for `StartService`, `StopService`, and `RestartService` functions in your shell script and calls them to start, stop, or restart your services as needed. You must provide implementations for all three routines, although the implementations can be empty for routines whose commands your service does not support.

If your startup-item executable contains code that might take a long time to finish, consider spawning off a background process to run that code. Performing lengthy startup tasks directly from your scripts delays system startup. Your startup item script should execute as quickly as possible and then exit.

For more information about writing shell scripts, see *Shell Scripting Primer*.

Note: Most apple-provided startup items have a test in the script to check to see if a particular variable is set to prevent automatic starting of daemons unless they are enabled (usually in the System Preference sharing pane).

To enable or disable a daemon that does not have a GUI checkbox, you must add or modify these variables directly by editing the file `/etc/hostconfig`.

Specifying the Startup Item Properties

The configuration property list of a startup item provides descriptive information about the startup item, lists the services it provides, and lists its dependencies on other services. Mac OS X uses the service and dependency information to determine the launch order for startup items. This property list is stored in ASCII format (as opposed to XML) and can be created using the Property List Editor application.

Note: In Mac OS X v10.4 and later, the dependency information for many stubbed-out system startup items is still present in case other startup items depend on it.

Table 1 (page 37) lists the key-value pairs you can include in your startup item's `StartupParameters.plist` file. Each of the listed arrays contains string values. You can use the Property List Editor application that comes with the Xcode Tools to create this property list. When saving your property-list file, be sure to save it as an ASCII property-list file.

Table 1 StartupParameters.plist key-value pairs

Key	Type	Value
Description	String	A short description of the startup item, used by administrative tools.
Provides	Array	The names of the services provided by this startup item. Although a startup item can potentially provide multiple services, it is recommended that you limit your startup items to only one service each.
Requires	Array	The services provided by other startup items that must be running before this startup item can be started. If the required services are not available, this startup item is not run.
Uses	Array	The services provided by other startup items that should be started before this startup item, but which are not required. The startup item should be able to start without the availability of these services.

For example, here is an old-style plist:

```
{
  Description     = "Software Update service";
```

```
  Provides        = ("SoftwareUpdateServer");
  Requires        = ("Network");
  Uses            = ("Network");
  OrderPreference = "Late";
  Messages =
  {
    start = "Starting Software Update service";
    stop  = "Stopping Software Update service";
  };
}
```

And here is an XML plist example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
    <dict>
        <key>Description</key>
        <string>Apple Serial Terminal Support</string>
        <key>OrderPreference</key>
        <string>Late</string>
        <key>Provides</key>
        <array>
                <string>Serial Terminal Support</string>
        </array>
        <key>Uses</key>
        <array>
                <string>SystemLog</string>
        </array>
    </dict>
</plist>
```

The service names you specify in the `Requires` and `Uses` arrays may not always correspond directly to the name of the startup item that provides that service. The `Provides` property specifies the actual name of the service provided by a startup item, and while this name usually matches the name of the startup item, it is not required to do so. For example, if the startup item launches multiple services, only one of those services can have the same name as the startup item.

If two startup items provide a service with the same name, the system runs only the first startup item it finds with that name. This is one of the reasons why your own startup items should launch only one service. If the name of only one of the services matches the name of another service, the entire startup item might not be executed and neither service would be launched.

The values of the `Requires` and `Uses` keys do not guarantee a particular launch order.

In Mac OS X v10.4 and later, most low-level services are started with `launchd`. By the time your startup item starts executing, `launchd` is running, and any attempt to access any of the services provided by a `launchd` daemon will result in that daemon starting. Thus, you can safely assume (or at least pretend) that any of these services are running by the time your startup item is called.

For this reason, with few exceptions, the `Requires` and `Uses` keys are largely irrelevant after Mac OS X v10.3 except to support service dependencies between two or more third-party startup items.

Managing Startup Items

During the boot process, the system launches the available startup items, passing a `start` argument to the startup item executable. After the boot process, the system may run the startup item executable again, this time passing it a `restart` or `stop` argument. Your startup item executable should check the supplied argument and act accordingly to start, restart, or stop the corresponding services.

Note: In general, with the exception of daemons provided with Mac OS X, the system will only run your startup script with `start` or `stop` arguments (at boot and shutdown, respectively). Users, however, may elect to use the `restart` argument.

You should not make any assumptions about the order in which daemons will be shut down.

If you want to start, restart, or stop startup items from your own scripts, you can do so using the `SystemStarter` program. To use `SystemStarter`, you must execute it with two parameters: the desired action and the name of the service provided by the startup item. For example, to restart the Apache Web server, you would execute the following command:

```
/sbin/SystemStarter restart "Web Server"
```

Important: You must have root authority to start, restart, or stop startup items.

Startup items should always respect the arguments passed in by `SystemStarter`. However, the response to those arguments is dependent on the startup item. The stop and restart options may not make sense in all cases. Your startup item could also support the restart option using the existing code for stopping and starting its service.

Displaying and Localizing a Startup Message

When your startup item starts at boot time, you may (if desired) display a message to the user. To do this, use the `ConsoleMessage` command. (You can use this command even if the computer is not starting up, but the user will not see it unless the Console application is running.)

For example:

```
ConsoleMessage "MyDaemon is running.  Better go catch it."
```

If you want to localize the message displayed when a startup item starts, you must create a series of property list files with localized versions of the strings. Each of these files must be named `Localizable.strings`, and must be in a localized project directory whose name is based on the name of a language or locale code for the desired language. These folders, in turn, must be in a folder called `Resources` in the startup item folder.

For example, you might create a tree structure that looks like this:

```
MyDaemon
    |
    |- MyDaemon
    |- StartupParameters.plist
```

```
    |- Resources
            |
            |- English.lproj
            |- French.lproj
            |- German.lproj
            |- zh_CN.lproj
```

Within each of these localizable strings files, you must include a dictionary whose keys map an English string to a string in another language. For example, the French version of the `PrintingServices` localization file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>Starting printing services</key>
        <string>Démarrage des services d'impression</string>
    </dict>
</plist>
```

Whenever the `ConsoleMessage` command is passed the string "Starting printing services", if the user's language preference is French, the user will instead see "Démarrage des services d'impression" at startup. C'est très bien!

The value of the *key* field must precisely match the english string printed by your startup item using `ConsoleMessage`.

See the manual page for `locale` for more information about locale codes.

Startup Item Permissions

Because startup items run with root authority, you must make sure your startup item directory permissions are set correctly. For security reasons, your startup item directory should be owned by root, the group should be set to wheel, and the permissions for the directory should be 755 (rwxr-xr-x). This means that only the root user can modify the directory contents; other users can examine the directory view its contents but not modify them. The files inside the directory should have similar permissions and ownership. Thus, the file listing for the Apache startup item directory is as follows:

```
./Apache:
total 16
drwxr-xr-x    4 root  wheel   136 Feb 14 14:33 .
drwxr-xr-x   21 root  wheel   714 Feb 14 15:03 ..
-rwxr-xr-x    1 root  wheel  1253 Feb 10 19:31 Apache
-rw-r--r--    1 root  wheel   152 Feb 10 19:31 StartupParameters.plist
```

Important: In Mac OS X version 10.4 and later, the system asks the user what to do about startup items with incorrect permissions. At this point, the user may choose to disable the startup item, which could have unexpected results for your software. To avoid this, be sure to set the permissions during installation.

Logging Errors and Warnings

You can use two major APIs in Mac OS X to log errors: `syslog` and `asl`. In addition, you can use a number of higher-level APIs built on top of these such as `NSLog`. However, because most daemons are written in C, you probably want to use the low-level APIs when writing code that executes at startup.

The `syslog` API is the most commonly used logging API on UNIX and Linux systems. For this reason, you should consider using it when writing cross-platform software. For software specific to Mac OS X, you should use the `asl` API because it provides more functionality.

If your daemon uses standard output or standard error to notify users of problems during startup, those error messages are not generally visible to a user. You can, of course, boot in verbose mode (by holding down Command-v at startup), but you may have a hard time reading the errors as they scroll by. To solve this problem, you should use system logging to record error conditions for later analysis by the user or system administrator.

Log Levels and Log Files

In Mac OS X (and other UNIX-based and UNIX-like operating systems), the system logger supports logging at a number of priority levels. The priority levels (and suggested uses for these levels) are:

- **Emergency (level 0):** The highest priority, usually reserved for catastrophic failures and reboot notices.
- **Alert (level 1):** A serious failure in a key system.
- **Critical (level 2):** A failure in a key system.
- **Error (level 3):** Something has failed.
- **Warning (level 4):** Something is amiss and might fail if not corrected.
- **Notice (level 5):** Things of moderate interest to the user or administrator.
- **Info (level 6):** The lowest priority that you would normally log, and purely informational in nature.
- **Debug (level 7):** The lowest priority, and normally not logged except for messages from the kernel.

The system logger in Mac OS X determines where to log messages at any given priority level based on the file `/etc/syslog.conf`.

Note: A daemon or application may mask low-priority messages before they even get to the system logger using the `setlogmask` function call. Thus, if you want to see these debugging messages in the system log, you may have to pass certain debugging flags to the daemon, regardless of how you have configured the system logger in `/etc/syslog.conf`.

Logging Errors Using the syslog API

The `syslog` API is relatively straightforward. It consists of five main functions:

```
void syslog(int priority, const char *message, ...);
void vsyslog(int priority, const char *message,
                va_list args);
void openlog(const char *ident, int logopt, int facility);
void closelog(void);
int setlogmask(int maskpri);
```

The first function you should call is `openlog`. This function associates future calls to `syslog` with a particular facility. You can find a list of facilities in the man page for `syslog`.

Note: Technically, you can call `syslog` without calling `openlog`, but as a rule, you *should* always call `openlog` to specify a facility and logging options. If you do not call `openlog` before you call `syslog`, the API will use the default facility, `LOG_USER`.

Next, you should call `syslog`. This function actually logs your message at a specified priority level. The priority levels for log messages are `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, or `LOG_DEBUG`, in decreasing order of importance. These correspond with the levels described in "Log Levels and Log Files" (page 41).

Warning: It is very important to choose an appropriate priority level for log messages. The system logger discards most low-priority messages, depending on the facility specified. To find out how the system logger decides which facilities and priority levels to log in a given log file, look in the file `/etc/syslog.conf`.

If you need to write a wrapper function for the `syslog` function, you should use the function `vsyslog` instead. This function is identical to `syslog` except that it takes a variable argument list parameter instead of a series of individual parameters.

Finally, when your program exits (or when you need to specify a different facility), you should call `closelog`. This function resets the facility and logging options to the default settings as though you had never called `openlog`.

The following code example shows how to log a simple error message:

```
#include <fcntl.h>
#include <syslog.h>

main()
{
```

```
        int cause_an_error = open("/fictitious_file", O_RDONLY, 0); // sets
errno to ENOENT
        openlog("LogIt", (LOG_CONS|LOG_PERROR|LOG_PID), LOG_DAEMON);
        syslog(LOG_EMERG, "This is a silly test: Error %m: %d", 42);
        closelog();
}
```

The flags passed to `openlog` specify the following:

- **LOG_CONS:** If the `syslogd` daemon is not running, the `syslog` function should print the message to the console.
- **LOG_PERROR:** in *addition* to normal logging, the `syslog` function should also print the message to standard error.
- **LOG_PID:** The `syslog` function should append the process ID after the process name at the beginning of the log message.

These and other flags are described in more detail in the `syslog` manual page.

In addition to the usual `printf` format flags, this command supports an additional flag, `%m`. If this flag appears in the log string, it is replaced by a string representation of the last error stored in `errno`. This is equivalent to what would be reported if you called `perror` or `strerror` directly.

Thus, the code sample above prints the following message to standard output:

```
LogIt[165]: This is a silly test: Error No such file or directory: 42
```

Then, the code sample tells the system logger to log that message. As a result, assuming you have not changed `/etc/syslog.conf`, the system logger broadcasts this message to all users:

```
Broadcast Message from user@My-Machine-Name.mycompany.com
        (no tty) at 13:28 PDT...

Jul 24 13:28:46 My-Machine-Name LogIt[601]: This is a silly test: Error No such
 file or directory: 42
```

In this example, the process ID was `601`, and the process name was `LogIt`.

For additional control over what gets logged, you can use the function `setlogmask` to quickly enable or disable logging at various levels. For example, the following code disables logging of any messages below the `LOG_EMERG` level (which is one higher than the `LOG_ALERT` level):

```
setlogmask(LOG_UPTO(LOG_ALERT));
```

You might, for example, use this function to disable logging of debug messages without recompiling your code or adding conditional statements.

Logging Errors Using the asl API

The `asl` API is short for Apple System Logger. The Apple System Logger API is very similar to `syslog` but provides additional functionality.

There are a few key differences, though; the `asl` logging API:

- Uses a text string for the facility identifier for more precise filtering of log messages
- Provides functions for querying the log files
- Is safe for use in a multithreaded environment because it provides functions for obtaining a separate communication handle for each thread

The following sample code is equivalent to the code in "Logging Errors Using the syslog API" (page 42), except that it uses `asl` for logging:

```
#include <fcntl.h>
#include <asl.h>
#include <unistd.h>

main()
{
        aslclient log_client;
        int cause_an_error = open("/fictitious_file", O_RDONLY, 0);

        log_client = asl_open("LogIt", "The LogIt Facility", ASL_OPT_STDERR);
       asl_log(log_client, NULL, ASL_LEVEL_EMERG, "This is a silly test: Error
 %m: %d", 42);
        asl_close(log_client);
}
```

A complete explanation of the additional features of the `asl` API is beyond the scope of this document. For more information, see the `asl` manual page.

Customizing Login and Logout

If you want to run custom scripts or applications when the user logs in, there are several ways to do it. You might use this feature to perform maintenance tasks or set up the operating environment for your own applications when the user first logs in.

Login Items

To launch an application each time the user logs in, use the Login items found in the Accounts system preference. Login items are appropriate for user-level applications, as opposed to applications that operate on behalf of the system. When you configure an application to be launched as a login item, you must remember that the user can disable the launching of your application via the Accounts system preference. Dependent applications must be able to respond appropriately if they detect the login-item application is not running.

Note: Beginning with Mac OS X v10.5, the recommended method for adding a login item programmatically is with the Shared File Lists API. If your application needs to be compatible with earlier versions of Mac OS X, you should instead add login items with Apple Events. The two additional methods listed here are deprecated and may cease to function with future updates to Mac OS X.

Adding Login Items with Shared File Lists

Available in Mac OS X v10.5 and later, the Shared File Lists API can be found in `LSSharedFileList.h` in `/System/Library/Frameworks/CoreServices.framework/Frameworks/LaunchServices.framework/Headers/`.

Adding Login Items with Apple Events

You can easily manipulate the list of login items for the currently logged-in user using Apple events. This technique is described in the sample code *LoginItemsAE*.

Adding Login Items with CFPreferences

Another technique for working with property lists is through the CFPreferences API in Core Foundation. Like the Apple Events technique, this only works for the current user. Unlike the Apple Events technique, however, that user does not need to be logged in on the console. Thus, this technique can be used to modify preferences for a different user by running this piece of code as that user using a setuid executable.

The following code snippet shows the basics:

```
CFArrayRef prefCFArrayRef =
CFPreferencesCopyAppValue(CFSTR("AutoLaunchedApplicationDictionary"),
CFSTR("loginwindow"));
CFMutableArrayRef tCFMutableArrayRef = CFArrayCreateMutableCopy(NULL, 0,
prefCFArrayRef);
/* Modify tCFMutableArrayRef here */
CFPreferencesSetAppValue(CFSTR("AutoLaunchedApplicationDictionary"),
tCFMutableArrayRef, CFSTR("loginwindow"));
```

For a more complete example, see the *CFPreferences* sample code. For additional documentation about the API itself, see *Core Foundation Framework Reference*.

Adding Login Items Manually

If you need to add applications manually to your list of Login items, you can modify the `~/Library/Preferences/loginwindow.plist` property-list file of the desired user using the Property List Editor. Inside this file, add the desired application to the array of applications listed under the `AutoLaunchedApplicationDictionary` key.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>AutoLaunchedApplicationDictionary</key>
        <array>
                <dict>
                        <key>Hide</key>
                        <true/>
                        <key>Path</key>

<string>/Applications/iTunes.app/Contents/Resources/iTunesHelper.app</string>
                </dict>
        </array>
        <key>BuildVersionStampAsNumber</key>
        <integer>17371360</integer>
        <key>BuildVersionStampAsString</key>
        <string>8J135</string>
        <key>SystemVersionStampAsNumber</key>
        <integer>168036096</integer>
        <key>SystemVersionStampAsString</key>
        <string>10.4.7</string>
</dict>
</plist>
```

Note: This file does not exist by default. It is created only when you change a setting from the default (for example, by adding a login item using System Preferences).

Loginwindow Scripts

Another way to run applications at login time is to launch them using a custom shell script. Mac OS X provides two options for launching scripts when the user logs in. When creating your script file, keep the following in mind:

- The permissions for your script file should include execute privileges for the appropriate users.
- Scripts launched by `loginwindow` are run as root. Therefore, you should thoroughly test your scripts before deploying them to make sure they do not adversely affect the user's system.
- In your script, the variable `$1` returns the short name of the user who is logging in.
- Other login actions wait until your hook finishes executing. Therefore, have your script do what it needs to do quickly and then exit.

Warning: Only one of these hooks can be installed at a time. For this reason, login scripts are not recommended for deployment. This information is provided as an aid to system administrators and should not be deployed in released software.

Installing Scripts Using Defaults

In Mac OS X v10.3 and later, you can use the `defaults` tool to install your login script. This is the preferred technique for installing a login script.

Create the script file and put it in a directory that is accessible to all users. In Terminal, use the following command to install the script (where `/path/to/script` is the full path to your script file):

```
sudo defaults write com.apple.loginwindow LoginHook /path/to/script
```

To remove this hook, you simply delete the property as follows:

```
sudo defaults delete com.apple.loginwindow LoginHook
```

When using the `sudo` command, you must be an administrator of the current system. When you run the preceding command, `sudo` prompts you to enter your password. You must provide a valid password before the hook can be installed.

Note: If no `plist` file exists for `com.apple.loginwindow`, this method will not work. This file (`/var/root/Library/Preferences/com.apple.loginwindow.plist`) does not exist on a fresh installation until the user changes a login window setting (such as turning on fast user switching).

If you must install startup scripts programmatically, you should consider providing a copy of this file containing the default configuration options. Then, if the file does not exist, copy that default configuration file into place before running `defaults`. Again, however, this technique is not recommended for use in applications because these hooks cannot coexist.

Installing Scripts Using Loginwindow Hooks

In Mac OS X v10.2 and v10.3, you can run your startup script by modifying the `/etc/ttys` file. In this file is the following line:

```
console "/System/Library/CoreServices/loginwindow.app/Contents/
    MacOS/loginwindow" vt100 on secure onoption="/usr/libexec/getty
    std.9600"
```

This line tells the `init` program to launch `loginwindow` on the console terminal. Into this line, you can insert additional parameters for `loginwindow` to process. Table 1 (page 48) lists the parameters that are supported by `loginwindow`.

Table 1 loginwindow parameters

Parameter	Description of value
`-LoginHook`	The full path to a script or tool to run when a user successfully logs in.
`-LogoutHook`	The full path to a script or tool to run when a user successfully logs out.
`-PowerOffDisabled`	If "YES," the Shut Down and Restart buttons in the login window are disabled; also, pressing the computer's power button quits the Finder and Dock applications but does not turn off the system. This feature prevents users from casually powering down a system that provides some shared service, such as a print server or file server. (This feature can also be controlled from the Accounts system preference.)

The `-LoginHook` and `-LogoutHook` parameters were particularly useful because they permit custom administrative, accounting, or security programs to run as part of the login and logout procedures. For example, your modified console definition in `/etc/ttys` might look similar to the following:

```
console "/System/Library/CoreServices/loginwindow.app/Contents/
    MacOS/loginwindow -PowerOffDisabled YES
    -LoginHook /Users/Administrator/Scripts/mailLoginToAdmin"
    vt100 on secure onoption="/usr/libexec/getty std.9600"
```

You should avoid using this technique to run your script in Mac OS X v10.4 and later. Instead, run your script using the `defaults` tool, as described in "Installing Scripts Using Defaults" (page 47).

Bootstrap Daemons

In Mac OS X v10.3, a mechanism similar to `launchd` was introduced to allow the launching of programs either at system startup or on a per-user basis. The process involved placing a specially formatted property list file in either the `/etc/mach_init.d` or the `/etc/mach_init_per_user.d` directory.

Important: The use of bootstrap daemons is deprecated and should be avoided entirely. Launching of daemons through this process is currently limited to some Apple-specific programs and may be removed or eliminated in a future release of Mac OS X. If you need to launch daemons, use `launchd`. If you need to launch daemons on versions of Mac OS X prior to 10.4, use a startup item.

Launchd User Agents

In Mac OS X v10.4, `launchd` was added. This is the preferred means of adding background agents on a per-user basis. These are described in more detail in "The launchd Startup Process" (page 27).

Document Revision History

This table describes the changes to *System Startup Programming Topics*.

Date	Notes
2008-11-19	Miscellaneous edits.
2007-02-08	Clarified the explanation of launchd daemons versus startup items.
2006-11-07	Added mention of prelinked kernels and helper partitions.
2006-10-03	Added information about Intel booting, launchd, and asl (logging).
2005-08-11	Updated guidance for customizing login and logout. Updated information pertaining to Startup Item property-list files.
2005-04-29	Updated for Mac OS X v10.4. Updated login/logout customization information.
	Added information about launch-on-demand daemons.
	Updated information related to the use of startup items.
2003-08-07	First revision of this programming topic. Some information in this programming topic originally appeared in *System Overview*.

Index