Creating Launch Daemons and Agents

If you are developing daemons to run on OS X, 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 OS X, `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. In addition, `launchd` can start jobs at timed intervals.

Although `launchd` supports 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 launched on demand, you will have to handle these dependencies in another way, such as by using the legacy startup item mechanism.

Launching Custom Daemons Using launchd

With the introduction of `launchd` in OS X v10.4, an effort was made to improve the steps needed to launch and maintain daemons. What `launchd` provides is a harness for launching 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. 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 process of making a daemon 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.

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 user agents.
3. It launches any user agents that requested to be running all the time.
4. As requests for a particular service arrive, it launches the corresponding user agent 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.

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 5-1 lists the required and recommended keys for all daemons.

The property list file is structured the same for both daemons and agents. You indicate whether it describes a daemon or agent by the directory you place it in. Property list files describing daemons are installed in `/Library/LaunchDaemons`, and those describing agents are installed in `/Library/LaunchAgents` or in the `LaunchAgents` subdirectory of an individual user's `Library` directory. (The appropriate location for executables that you launch from your job is `/usr/local/libexec`.)

Table 5-1 Required and recommended property list keys

Key	Description
`Label`	Contains a unique string that identifies your daemon to `launchd`. (required)
`ProgramArguments`	Contains the arguments used to launch your daemon. (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. (required if your daemon was designed to be launched by `inetd`; otherwise, must not be included)
`KeepAlive`	This key specifies whether your daemon launches on-demand or must always be running. It is recommended that you design your daemon to be launched on-demand.

For more information: For a complete listing of the keys, see the `launchd.plist` manual page.

For sample configuration property lists, look at the files in `/System/Library/LaunchDaemons/`. These files are used to configure many daemons that run on OS X.

Writing a “Hello World!” launchd Job

The following simple example launches a daemon named `hello`, passing `world` as a single argument, and instructs launchd to keep the job running:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.hello</string>
    <key>ProgramArguments</key>
    <array>
        <string>hello</string>
        <string>world</string>
    </array>
    <key>KeepAlive</key>
    <true/>
</dict>
</plist>
```

In this example, there are three keys in the top level dictionary. The first is `Label`, which uniquely identifies the job. when. The second is `ProgramArguments` which has a value of an array of strings which represent the tokenized arguments and the program to run. The third and final key is `KeepAlive` which indicates that this job needs to be running at all times, rather than the default launch-on-demand behavior, so launchd should always try to keep this job running.

Listening on Sockets

You can also include other keys in your configuration property list file. For example, if your daemon monitors a well-known port (one of the ports listed in `/etc/services`), add a `Sockets` entry as follows:

```
<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>
```

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 the same, except the string contains a number instead of a name. For example:

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

Debugging launchd Jobs

There are some options that are useful for debugging your launchd job.

The following example enables core dumps, sets standard out and error to go to a log file, and instructs launchd to temporarily increase the debug level of its logging while acting on behalf of your job (remember to adjust your syslog.conf accordingly):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.sleep</string>
    <key>ProgramArguments</key>
    <array>
        <string>sleep</string>
        <string>100</string>
    </array>
    <key>StandardOutPath</key>
    <string>/var/log/myjob.log</string>
    <key>StandardErrorPath</key>
    <string>/var/log/myjob.log</string>
    <key>Debug</key>
```

```
    <true/>
    <key>SoftResourceLimits</key>
    <dict>
        <key>Core</key>
        <integer>9223372036854775807</integer>
    </dict>
    <key>HardResourceLimits</key>
    <dict>
        <key>Core</key>
        <integer>9223372036854775807</integer>
    </dict>
</dict>
</plist>
```

Running a Job Periodically

The following example creates a job that is run every five minutes (300 seconds):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.touchsomefile</string>
    <key>ProgramArguments</key>
    <array>
        <string>touch</string>
        <string>/tmp/helloworld</string>
    </array>
    <key>StartInterval</key>
    <integer>300</integer>
</dict>
</plist>
```

Alternately, you can specify a calendar-based interval. The following example starts the job on the 7th day of every month at 13:45 (1:45 pm). Like the Unix cron subsystem, any missing key of the `StartCalendarInterval` dictionary is treated as a wildcard—in this case, the month is omitted, so the job is run every month.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.touchsomefile</string>
```

```
    <key>ProgramArguments</key>
    <array>
        <string>touch</string>
        <string>/tmp/helloworld</string>
    </array>
    <key>StartCalendarInterval</key>
    <dict>
        <key>Minute</key>
        <integer>45</integer>
        <key>Hour</key>
        <integer>13</integer>
        <key>Day</key>
        <integer>7</integer>
    </dict>
</dict>
</plist>
```

Monitoring a Directory

The following example starts the job whenever any of the paths being watched have changed:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.watchhostconfig</string>
    <key>ProgramArguments</key>
    <array>
        <string>syslog</string>
        <string>-s</string>
        <string>-l</string>
        <string>notice</string>
        <string>somebody touched /etc/hostconfig</string>
    </array>
    <key>WatchPaths</key>
    <array>
        <string>/etc/hostconfig</string>
    </array>
</dict>
</plist>
```

An additional file system trigger is the notion of a queue directory. The launchd daemon starts your job whenever the given directories are non-empty, and it keeps your job running as long as those directories are not empty:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.mailpush</string>
    <key>ProgramArguments</key>
    <array>
        <string>my_custom_mail_push_tool</string>
    </array>
    <key>QueueDirectories</key>
    <array>
        <string>/var/spool/mymailqdir</string>
    </array>
</dict>
</plist>
```

Emulating inetd

The launchd daemon emulates the older `inetd`-style daemon semantics if you provide the `inetdCompatibility` key:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//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.telnetd</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/libexec/telnetd</string>
    </array>
    <key>inetdCompatibility</key>
    <dict>
        <key>Wait</key>
        <false/>
    </dict>
    <key>Sockets</key>
    <dict>
        <key>Listeners</key>
        <dict>
            <key>SockServiceName</key>
            <string>telnet</string>
```

```
            <key>SockType</key>
            <string>stream</string>
        </dict>
    </dict>
</dict>
</plist>
```

Behavior for Processes Managed by launchd

Processes that are managed by `launchd` must follow certain requirements so that they interact properly with `launchd`. This includes launch daemons and launch agents.

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.
- 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.
- Daemons and agents that are installed globally must be owned by the root user. Agents installed for the current user must be owned by that user. All daemons and agents must not be group writable or world writable. (That is, they must have file mode set to `600` or `400`.)

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 (rather than an error) when processing requests.
- Register the sockets and file descriptors used by your daemon in your `launchd` configuration property list file.
- If your daemon advertises a socket, check in with `launchd` as part of your daemon initialization. For an example implementation of the check-in process, see *SampleD*.
- During check-in, get the launch dictionary from `launchd`, extract and store its contents, and then discard the dictionary. Accessing the data structure used for the dictionary is very slow, so storing the whole 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 *avoid* 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.

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 when idle, 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.

Network Availability

If your daemon depends on the network being available, this cannot be handled with dependencies because network interfaces can come and go at any time in OS X. To solve this problem, you should use the network reachability functionality or the dynamic store functionality in the System Configuration framework. This is documented in *System Configuration Programming Guidelines* and *System*

Configuration Framework Reference. For more information about network reachability, see Determining Reachability and Getting Connected in *System Configuration Programming Guidelines*.

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*.

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 Logins

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, as described in Technical Q&A QA1133.

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.

The daemon may manually request that an extension be loaded. To do this, run `kextload` with the appropriate arguments using `exec` or variants thereof. I/O Kit kernel extensions should not be loaded with `kextload`; the I/O Kit will load them automatically when they are needed.

> **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.

Alternatively, our daemon may wait for a kernel service to be available. To do this, you should first register for service change notification. This 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 *IOKit Fundamentals*, *I/O Kit Framework Reference* (user space reference), and *Kernel Framework Reference* (kernel space reference).

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 Mac Developer Library as the SampleD sample code project.

The *Daemons and Agents* technical note provides additional information about how `launchd` daemons and agents work under the hood.

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/.
