Multiple User Environments

Mac OS X

2005-07-07

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

Simultaneously published in the United States and Canada.

Contents

Figures

Introduction to Multiple User Environments

This programming topic provides background information about the multiple user environment of Mac OS X. It also provides guidelines on how to write software to support this type of environment, including ways you may need to change your existing Mac OS X applications.

Mac OS X has always supported the use of a single machine by multiple users. Initially, this usage was exclusive; only one user at a time could log in to the console and use the machine. In version 10.3, Mac OS X introduced a feature called **fast user switching** that lets multiple login sessions run concurrently on the same machine. With this feature, one user at a time is active on the machine while the other user's sessions continue to run in the background.

Prior to the introduction of fast user switching, developers could rely on the fact that only one user at a time was active on the system console. This meant that applications could make some assumptions about the availability of resources. Unfortunately, some of these assumptions may cause applications to fail in a fast user switching environment. If you are developing applications to run in Mac OS X, you should examine your designs and make sure they take multiple simultaneous users into account.

Note: Mac OS X has always supported multiple simultaneous users through secure shell (`ssh`) connections. Each `ssh` connection runs in its own login session. See "Root and Login Sessions" (page 9) for more information about sessions and how they affect processes.

Organization of This Document

This programming topic contains the following articles:

- "Root and Login Sessions" (page 9) provides advanced material for daemon developers that describes the organization of the Mac OS X process space and how that organization impacts applications.
- "Supporting Fast User Switching" (page 13) provides general guidelines for application developers on how to make your application work in a fast user switching environment.
- "User Switch Notifications" (page 19) shows you how to handle the notifications that occur when the console user changes.

If you want to know more about the login/logout process, are writing a daemon or startup item, or want to know more about the daemons that run in the root session, see *System Startup Programming Topics*.

Root and Login Sessions

From early on, Mac OS X was designed to be a secure operating system. One aspect of this security is the control exercised over inter-process communication. Processes are loosely gathered into groups and associated with either the root session or a login session. The `mach_init` program, also known as the **bootstrap server**, assigns each new process to an appropriate session based on factors such as who created the process and when.

Most root-level processes are placed in the **root session**. The root session is the first session to be created and the last to be destroyed. Only one root session ever exists on the system, and it is where most boot-time processes and daemons live. Processes in the root session are allowed to provide services to all users of the system. For example, `lookupd` and the `mDNSResponder` process both run in the root session. These processes are user-independent and vend basic services to anyone on the system.

Processes launched by a user or for a user's benefit live in a **login session**. Each login session is associated with an authenticated user. The system may have multiple login sessions active at any given time, but each one is something of an island to the associated user. Console login sessions include processes such as the Finder and Dock; however, remote login connections contain only shell-level processes. Most of the communication between different login sessions is restricted by the bootstrap server. Communication is still possible but generally requires creating an explicit, and trusted, connection.

Important: The content of this article is intended primarily for developers writing daemons and other low-level processes; other developers may find the information interesting but should not need it in the general course of development. These sections also assume some knowledge of the Mach kernel environment, which is described in *Kernel Programming* in Darwin Documentation.

Securing the User Space

By organizing processes into root and login sessions, the `mach_init` process is able to create a more trustworthy environment for users. Requests for Mach ports go through the `mach_init` process, which routes the requests to the appropriate target. The login session acts like a set of walls around a user's processes, limiting port requests to those in the current login session or root session. Doing so prevents accidental or intentional attempts by processes in a different session to obtain access to the ports in the current login session. In a sense, login sessions are a lightweight firewall around the processes they represent.

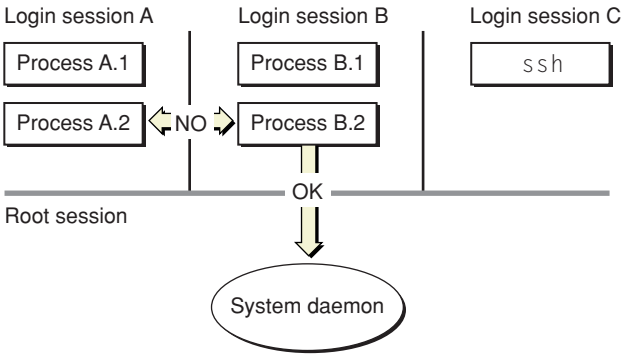
Thus, login sessions give users an assurance that processes launched by other users do not interfere with their own processes. For example, a malicious user might write a program that pretends to be a known user service and use that program to gather information from other users. However, unless the malicious user has administrative access to the machine, the program runs in the login session of the malicious user. Because other users cannot see the program, they are protected from its effects.

Communicating Across Login Sessions

At boot time, the kernel launches the `mach_init` process, whose job is to handle lookup requests for Mach ports. As part of each request, `mach_init` also ensures that processes do not attempt to cross login session bounds illegally. This is not to say that crossing session bounds is completely forbidden. There are situations where it is appropriate and even necessary. For example, applications can use BSD sockets, shared files, shared memory, or distributed notifications to communicate with processes in other sessions; however, doing so requires cooperation between both sessions and involves a certain level of trust.

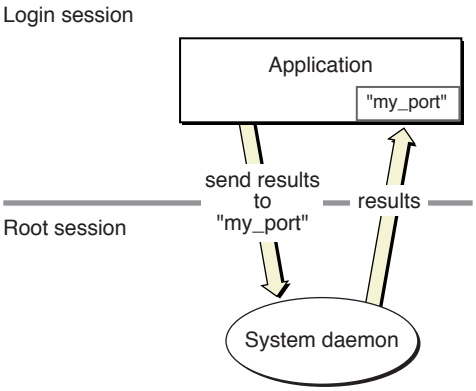
Conceptually, you can think of the root session as a parent session for all of the login sessions that follow. However, the parent-child analogy ends there. Processes in the root session have no inherent knowledge of processes in any of the active login sessions. Conversely, processes in login sessions do have knowledge of processes in the root session and can access them as needed for appropriate services. User processes in one login session do not have any inherent access to processes in other login sessions, though. Figure 1 illustrates this relationship.

Figure 1 Root and login session relationships



Processes in the root session are not completely blind to processes in login sessions. When a user process requests a service from a daemon running in the root session, the process typically provides a port address over which to return the results. Because it has an explicit Mach port to talk to, the daemon can now send the results of the request directly back to the user process. Figure 2 illustrates this behavior.

Figure 2 Communicating with user processes



Session Lifespans

As you might expect, the root session is the first to be created on the system. It is also the last to be destroyed when the system is shut down. The root session is the logical place in which to run daemons and system services that apply to all users. However, writing a daemon to run in the root session precludes the use of many higher-level system frameworks that require the presence of the window server. (For a list of frameworks you can use, see "Frameworks Available in the Root Session" (page 12).)

When a user logs in, either through the login window or `ssh`, the system creates a new login session. The login session remains in existence until all processes belong to it are terminated. When a user logs out, the system attempts to terminate the processes in that user's login session. When the last process in a login session dies, the system closes out the login session and reclaims its memory.

Note: If a user process daemonizes itself prior to logout, it can live past the end of the user logout and prolong the existence of the login session. Some system services, such as the Apache web server, do this to avoid being shut down during a logout. Processes that survive the user logout continue to run in the login session. However, they must be able to run without the existence of the window server, which is terminated when the user logs out.

Applications can defer a user logout for various reasons, which are described in *System Startup Programming Topics*. Per-user services are shut down automatically and are not given the chance to abort the logout procedure.

Identifying Login Sessions

Each time a user is authenticated with the system, the Security layer of the system creates a unique ID to identify the user's login session. This ID is the security session ID, often referred to simply as the **session ID**. Applications can use the session ID to distinguish among resources allocated in different login sessions.

Session IDs are not persistent between user logins. Each session ID is valid only for the duration of the current login session, and it is valid for all processes in that login session. If a user logs out and logs back in, a different session ID is asssigned to the new login session.

Note: Session IDs should not be confused with setsid sessions created by calling the `setsid` function. The two values are distinct and used for different purposes. The security session ID encompasses all processes launched within a given security session and may include multiple setsid sessions.

You should use session IDs to prevent namespace collisions with objects in other login sessions. If an application used a common name for a session-specific shared memory region, the same application in another session would likely encounter an error when trying to create that memory region using the same name. Including the session ID in the memory region name can prevent these kinds of errors.

For information on how to get session IDs, see "Getting Login Session Information" (page 14).

How Login Sessions Affect Developers

If you are writing applications for end users, the existence of login sessions requires more thought in your design. You must remember that multiple instances of your application may be running simultaneously and write your code to handle potential resource conflicts. If you need to communicate with processes in other login sessions, you need to use BSD sockets, distributed notifications, or some other form of trusted connection.

If you are writing *only* kernel code, root and login sessions are largely irrelevant. However, driver developers frequently need to write programs that run either in the kernel or in the root session and communicate with programs in one or more login sessions. For example, a driver developer might want to configure a sound driver based on the active user's preferences. In this case, it would be necessary to write an agent program to run in each login session and communicate user-level changes to the driver.

Frameworks Available in the Root Session

Many system frameworks depend on the window server for part of their implementations. They either use the window server for drawing graphics, for tracking notifications, or for coordinating with the system in other ways. For most applications, this is not a problem at all and is actually necessary to implement some behaviors. However, if you are writing a daemon or other type of program to run in the root session, there is no window server process with which to communicate. As a result, many higher level frameworks cannot be used at all.

Supporting Fast User Switching

Fast user switching lets users share a single machine without having to log out every time they want to access their user account. Users share physical access to the machine, including the same keyboard, mouse, and monitor. However, instead of logging out, a new user simply logs in and switches out the previous user.

Processes in a switched-out login session continue running as before. They can continue processing data, communicating with the system, and drawing to the screen buffer as before. However, because they are switched out, they do not receive input from the keyboard and mouse. Similarly, if they were to check, the monitor would appear to be in sleep mode. As a result, it may benefit some applications to adjust their behavior while in a switched-out state to avoid wasting system resources.

While fast user switching is a convenient feature for users, it does provide several challenges for application developers. Applications that rely on exclusive access to certain resources may need to modify their behavior to live in a fast user switching environment. For example, an application that stores temporary data in `/tmp` may run into problems when a second instance running under a different user tries to modify the same files in that directory.

Guidelines for Supporting Fast User Switching

To support fast user switching, there are certain guidelines you should follow in developing your applications. Most of these guidelines describe safe ways to identify and share system resources. The summary of these guidelines is as follows:

- Incorporate session ID information into the name of any entity that appears in a global namespace, including the following:
  - File names
  - Shared memory regions
  - Semaphores
  - Named pipes
- Accept and handle user switch notifications. See "User Switch Notifications" (page 19) for more information.
- Do not assume you have exclusive access to any resource, especially the following:
  - TCP/IP ports
  - Hardware devices
- Do not assume there is only one instance of a per-user service running.
- Use file-level or range-level locks when accessing files.

Tagging application-specific resources with a session ID is necessary to distinguish them from similar resources created by applications in a different session. The Security layer of the system assigns a unique ID to each login session. Incorporating this ID into cache file or temporary directory names can prevent namespace collisions when creating these files. See "Getting Login Session Information" (page 14) for information on how to get the session ID.

Obtaining shared system resources, such as TCP/IP ports, introduces other problems in a fast user switching environment. For example, if you have a chat connection, do you hold onto the port or give it up when a new user is switched in? How you handle these situations depends on the resource involved and the design of your application. Regardless of how you handle it, you need to know when the user switch occurs, and for that you should read the section "User Switch Notifications" (page 19), which describes the notifications available to applications.

Getting Login Session Information

If your application creates a file, shared memory region, or other object that lives in a global namespace, you should brand that object with your unique session ID.

Mac OS X provides access to session information from two different frameworks. The Security framework includes a function for retrieving the session ID along with the basic information about the login session. The Core Graphics framework provides additional information about the login session, including the name of the user and whether the login process has completed.

Guidelines for Using Session IDs

Because session IDs are unique to a session, use them to identify resources that are session-specific, but which may still be shared by entities in same login session. For example, if your application creates a shared memory region, you could include the session ID in the name of that region. Other applications (including copies of the same application) would be able to access this memory region using the shared name and session information.

You can use session IDs in directory names to group temporary files together and make them easier to find. However, it is still recommended that you use the `mktemp` family of functions to generate random names for the files in that directory.

Using the Security Framework

The Security framework can be linked into any program, whether it resides in the root session or in a login session. This framework includes the function `SessionGetInfo` for getting basic information about the current session, including the session ID and some session attributes.

To get the session ID of the current session, pass the value `callerSecuritySession` as the first parameter to `SessionGetInfo`. If you happen to have the session ID of a different login session, you can use it to get information about the other session. The parameters returned by `SessionGetInfo` are mostly informational and cannot be modified; thus they have no consequences on the security of the login session.

The following example shows you how to call `SessionGetInfo` to get the session ID for the current root or login session.

```
#include <Security/Security.h>

OSStatus error;
SecuritySessionId mySession;
SessionAttributeBits sessionInfo;

error = SessionGetInfo(callerSecuritySession, &mySession, &sessionInfo);
```

The session ID itself is an integer value that persists for the duration of the login session; see "Identifying Login Sessions" (page 11) for additional information. The attribute bits parameter is an integer bitmask value that provides additional information about the session, such as whether it is the root session and whether it has an available console. See the header files for a description of the available bit constants.

Using the Core Graphics Framework

The Core Graphics framework relies on the presence of the window server and thus is available only to applications running in a login session. This framework includes the `CGSessionCopyCurrentDictionary` function for getting information about the current login session, including the user ID and name.

Note: The user ID value returned by `CGSessionCopyCurrentDictionary` should not be confused with the session ID returned by the Security framework. The user ID is a fixed value assigned by the system to differentiate among users on the system. The session ID uniquely identifies the session and may differ between logins of the same user.

The following example shows you how to get login session information using the `CGSessionCopyCurrentDictionary` function. This function returns a dictionary with several keys that you can read to get the information you need.

```
#include <CoreFoundation/CoreFoundation.h>
#include <ApplicationServices/ApplicationServices.h>

CFStringRef shortUserName;
CFNumberRef userUID;
CFBooleanRef userIsActive;
CFBooleanRef loginCompleted;

int MyCGGetSessionInfo
{
    CFDictionaryRef sessionInfoDict;

    sessionInfoDict = CGSessionCopyCurrentDictionary();
    if (sessionInfoDict == NULL)
    {
        printf("Unable to get session dictionary.");
        return(1);
    }

    shortUserName = CFDictionaryGetValue(sessionInfoDict,
                                         kCGSessionUserNameKey);
    userUID = CFDictionaryGetValue(sessionInfoDict,
                                         kCGSessionUserIDKey);
    userIsActive = CFDictionaryGetValue(sessionInfoDict,
                                         kCGSessionOnConsoleKey);
    loginCompleted = CFDictionaryGetValue(sessionInfoDict,
```

```
                                        kCGSessionLoginDoneKey);
}
```

Distributed Notifications

Prior to the introduction of fast user switching, distributed notifications sent using either the Core Foundation or Cocoa interfaces were delivered to any process that registered as an observer. With the introduction of fast user switching in Mac OS X 10.3, the existing interfaces have changed to limit distribution to registered processes in the current login session. This change should not affect the behavior of most applications. For those applications that might be affected, new interfaces have been added for distributing notifications across login session boundaries.

Posting Notifications to All Sessions

If you need to send a distributed notification to other processes, you should generally do so only to processes in the current login session. However, there may be cases where you need to send a notification to all processes, regardless of their corresponding login session. For example, changes to system-global settings affect all applications that rely on those settings.

In Core Foundation, the `CFNotificationCenterPostNotificationWithOptions` function allows you to post notifications to all login sessions. To use this function for global notifications, pass the `kCFNotificationPostToAllSessions` constant to the `options` parameter.

In Cocoa, the `postNotificationName:object:userInfo:options:` method of the NSDistributedNotificationCenter object allows you to post notifications to all login sessions. To use this function for global notifications, pass the `NSNotificationPostToAllSessions` constant to the `options` parameter.

The use of global notifications should be minimized whenever possible. If you need to identify the login session that originated a notification, put its session ID into the user information dictionary of the notification. For more information on getting the session ID, see "Getting Login Session Information" (page 14).

Handling Distributed Notifications

The handler for a global notification must be prepared to handle that notification even when it comes from a different login session. If your handler assumes the notification originated in the same login session, your application could experience problems in a fast user switched environment. Fortunately, existing methods for distributing notifications limit the scope of delivery to the current login session. Applications that need to send global notifications must be modified to send those notifications explicitly with interfaces introduced in Mac OS X version 10.3.

If you set up your application to receive a particular global notification, keep in mind that you may not need to handle all instances of that notification. For example, a daemon in the root session might use a global notification to communicate with a process in the active session. In this situation, use of a global notification is preferable because the daemon does not have any knowledge of programs in the login sessions. By broadcasting to all sessions, the notification is guaranteed to reach the active session. Applications in inactive sessions would simply ignore the notification.

Disabling Multiple Session Support

If the need arises, you can tell Mac OS X not to run your application in more than one login session. This option should be avoided if possible; however, if you are unable to get your application running suitably in multiple login sessions, you might consider using it.

To disable multiple-session support for your application, include the LSMultipleInstancesProhibited key in your application's information property list. If this key is present and set to true, Launch Services denies attempts to launch a second instance of your application, in any login session. This key works both for launching your application across multiple sessions and for launching a second instance of your application in the same session. In both cases, Launch Services returns an error to the calling process indicating the reason for the failure.

For details on how to use the LSMultipleInstancesProhibited key, see "Property List Key Reference" in *Runtime Configuration Guidelines*.

User Switch Notifications

When a user switch occurs, Mac OS X generates events for all interested applications. Events are sent to applications in a login session whenever the login session is activated or deactivated. If a login session is not being activated or deactivated, it receives no events. You can use the activation events to perform the following kinds of tasks:

- Halt or restart sound playback
- Halt or restart animations
- Give up or acquire shared resources
- Put your application into a quiescent state to improve overall system performance

Carbon Notifications

For Carbon applications, user switch notifications come through the Carbon Event Manager. Applications can register to receive the `kEventSystemUserSessionActivated` and `kEventSystemUserSessionDeactivated` events if they want to know when a switch occurs. The following example shows you how to register for these events:

```
pascal OSStatus switchEventsHandler (EventHandlerCallRef nextHandler,
                                     EventRef switchEvent,
                                     void* userData)
{
    if (GetEventKind(switchEvent)== kEventSystemUserSessionDeactivated)
    {
        // Perform deactivation tasks here.
    }
    else
    {
        // Perform activation tasks here.
    }

    return noErr;
}

void SwitchEventsRegister()
{
    EventTypeSpec switchEventTypes[2];
    EventHandlerUPP switchEventHandler;

    switchEventTypes[0].eventClass = kEventClassSystem;
    switchEventTypes[0].eventKind = kEventSystemUserSessionDeactivated;
    switchEventTypes[1].eventClass = kEventClassSystem;
    switchEventTypes[1].eventKind = kEventSystemUserSessionActivated;
```

```
    switchEventHandler = NewEventHandlerUPP(switchEventsHandler);
    InstallApplicationEventHandler(switchEventHandler, 2,
                                   switchEventTypes, NULL, NULL);
}
```

Cocoa Notifications

For Cocoa applications, user switch notifications come through the NSWorkspace shared notification center. The user-switch events themselves are defined in NSWorkspace and are `NSWorkspaceSessionDidBecomeActiveNotification` and `NSWorkspaceSessionDidResignActiveNotification`. To register for these notifications, your application (or its delegate) would provide a handler method and then register that handler by adding itself as an observer of the notification. The following sample methods illustrate how to do this:

```
- (void) switchHandler:(NSNotification*) notification
{
    if ([[notification name] isEqualToString:
                NSWorkspaceSessionDidResignActiveNotification])
    {
        // Perform deactivation tasks here.
    }
    else
    {
        // Perform activation tasks here.
    }
}

// Register the handler
- (void) applicationDidFinishLaunching:(NSNotification*) aNotification
{
    [[[NSWorkspace sharedWorkspace] notificationCenter]
            addObserver:self
            selector:@selector(switchHandler:)
            name:NSWorkspaceSessionDidBecomeActiveNotification
            object:nil];

    [[[NSWorkspace sharedWorkspace] notificationCenter]
            addObserver:self
            selector:@selector(switchHandler:)
            name:NSWorkspaceSessionDidResignActiveNotification
            object:nil];
}
```

Event Timing

User switch notifications are sent to applications at the same time the switch occurs. Because the switch occurs relatively quickly, this is normally not a problem. However, it is possible for an application to receive its activation event before other applications have received their deactivation events. This could lead to potential race conditions between applications releasing and acquiring shared resources.

To avoid race conditions, applications in the session being deactivated should continue to release any shared resources as soon as possible. Applications in the session being activated should delay the acquisition of any shared resources until those resources are actually used. Not only can this help avoid potential race conditions, it can also improve overall system performance. If your application needs a particular resource right away but encounters errors while trying to acquire it, set a timer and try to acquire the resource again a short time later.

Shutdown Notifications

Shutdown and restart sequences do not generate any special notifications to switched out login sessions. Without fast user switching enabled, the normal sequence is to notify the user's applications about the impending termination and give them a chance to abort the sequence. However, with fast user switching enabled, only the applications for the active user are prompted. Applications in switched out login sessions are killed without the chance to save any changes; otherwise, there is the potential that a process in one of those sessions could hang the system.

If other users are logged in, the system warns the initiator of a shutdown or restart sequence that those users might lose unsaved changes. This warning only appears when other users are logged in. The user is prompted for an administrative password to ensure that there is a good reason to shutdown or restart the machine. As long as the user has a valid administrator password, the sequence proceeds.

For more information about the shutdown and restart sequence for the active login session, see *System Startup Programming Topics*.

Document Revision History

This table describes the changes to *Multiple User Environments*.

Date	Notes
2005-07-07	Corrected code example that demonstrates the way to use the security framework.
2004-08-31	Removed references to missing tech note.
2004-03-26	Fixed minor bugs.
	Removed references to non-existent tech notes.
2003-09-18	First version of *Multiple User Environments*.