File System Events Programming Guide

Darwin > File Management

2008-03-11

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

Contents

Listings

Introduction

The file system events API provides a way for your application to ask for notification when the contents of a directory hierarchy are modified. For example, your application can use this to quickly detect when the user modifies a file within a project bundle using another application.

It also provides a lightweight way to determine whether the contents of a directory hierarchy have changed since your application last examined them. For example, a backup application can use this to determine what files have changed since a given time stamp or a given event ID.

You should read this document if your application works with a large number of files—particularly if your application works with large hierarchies of files.

Organization of This Document

This document is organized into the following chapters:

- "Technology Overview" (page 9)—describes the file system events API and explains how it works at a high level.
- "Using the File System Events API" (page 11)—explains how to use the file system events API, from creating an event stream to writing a handler, including code examples to help you quickly get started.
- "File System Event Security" (page 21)—describes the security features of the file system events API.
- "Kernel Queues: An Alternative to File System Events" (page 23)—explains the kernel queues mechanism, describes when it may be appropriate to use the kernel queues API in lieu of the file system events API, and includes a brief code example to show you how to use it.

Technology Overview

The File System Events API is a new technology available in Mac OS X v10.5 and later. With it, you can register for notification of any changes that occur to a directory hierarchy or the contents thereof.

The file system events mechanism consists of three parts: kernel code that passes raw event notification to user space through a special device (also used by spotlight), a daemon which filters this stream and sends out notifications, and a persistent database which stores a record of all changes throughout time.

When your application registers for notification, the file system events daemon will post a notification every time that any file in the monitored directory hierarchy changes. It will also post a notification if the directory itself is modified (for example, if its permissions change or a new file is added). The important point to take away is that the granularity of notifications is at a directory level. It tells you *only* that something in the directory has changed, but does not tell you *what* changed.

In addition to collapsing notifications for changes to multiple files into a single, per-directory notification, the file system events daemon also coalesces multiple notifications on a given directory if they occur in a short period of time. You will always receive at least one notification after the last change is made. Beyond that, the temporal granularity can be as coarse or as fine as you desire; you choose the minimum time between events when you register for notification.

To better understand this technology, you should first understand what it is not. It is not a mechanism for registering for fine-grained notification of filesystem changes. It was not intended for virus checkers or other technologies that need to immediately learn about changes to a file and preempt those changes if needed. The best way to support these is through a kernel extension that registers for interest in changes at the VFS level.

The file system events API is also not designed for finding out when a particular file changes. For such purposes, the kqueues mechanism is more appropriate.

The file system events API is designed for passively monitoring a large tree of files for changes. The most obvious use for this technology is for backup software. Indeed, the file system events API provides the foundation for Apple's backup technology.

Another good use for the file system events API is providing consistency guarantees for applications that store data as a loose collection containing a project and dozens or hundreds of related files potentially scattered across the disk—DVD Studio Pro, for example. When one of the related files is modified by another application, your application needs to know about it so that it can choose how to incorporate the change into the project.

Of course, this use can also potentially be satisfied through the use of kqueues, but file system events provide the convenience of being able to see changes that occurred even while your application was not running, which can make consistency checking easier when your application opens a project initially.

Using the File System Events API

The File System Events API consists of several distinct groups of functions. You can obtain general information about volumes and events by using functions that begin with `FSEvents`. You can create a new event stream, perform operations on the stream, and so on using functions that begin with `FSEventStream`.

The life cycle of a file system events stream is as follows:

1. The application creates a stream by calling `FSEventStreamCreate` or `FSEventStreamCreateRelativeToDevice`. The stream initially has a retain count of 1. If desired, you can increment this count by calling `FSEventStreamRetain`.
2. The application schedules the stream on the run loop by calling `FSEventStreamScheduleWithRunLoop`.
3. The application tells the file system events daemon to start sending events by calling `FSEventStreamStart`.
4. The application services events as they arrive. The API posts events by calling the callback function specified in step 1.
5. The application tells the daemon to stop sending events by calling `FSEventStreamStop`.
6. If the application needs to restart the stream, go to step 3.
7. The application unschedules the event from its run loop by calling `FSEventStreamUnscheduleFromRunLoop`.
8. The application invalidates the stream by calling `FSEventStreamInvalidate`.
9. The application releases its reference to the stream by calling `FSEventStreamRelease`.

These steps are explained in more detail in the sections that follow.

Adding Include Directives

Before you use the file system event stream API, you must include the Core Services framework as follows:

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

When you compile, you must include the Core Services Framework by adding it to your target in Xcode or by adding the flag `-framework CoreServices` to your linker flags on the command line or in a Makefile.

Creating an Event Stream

The file system events API supports two types of event streams: per-disk event streams and a per-host event streams. Before you can create a stream, you must decide which type of stream to create: a per-host event stream or a per-disk event stream. You can create these streams by calling the functions `FSEventStreamCreate` and `FSEventStreamCreateRelativeToDevice`, respectively.

A per-host event stream consists of events whose IDs are increasing with respect to other events on that host. These IDs are guaranteed to be unique with one exception: if additional disks are added from another computer that was also running Mac OS X v10.5 or later, historical IDs may conflict between these volumes. Any new events will automatically start after the highest-numbered historical ID for any attached drive.

A per-disk event stream, by contrast, consists of events whose IDs are increasing with respect to previous events on that disk. It does not have any relationship with other events on other disks, and thus you must create a separate event stream for each physical device that you wish to monitor.

In general, if you are writing software that requires persistence, you should use per-disk streams to avoid any confusion due to ID conflicts. By contrast, per-host streams are most convenient if you are monitoring for changes in a directory or tree of directories during normal execution, such as watching a queue directory.

Note: Because disks can be modified by computers running earlier versions of Mac OS X (or potentially other operating systems), you should treat the events list as advisory rather than a definitive list of all changes to the volume. If a disk is modified by a computer running a previous version of Mac OS X, the historical log is discarded.

For example, backup software should still periodically perform a full sweep of any volume to ensure that no changes fall through the cracks.

If you are monitoring files on the root file system, either stream mechanism will behave similarly.

For example, the following snippet shows how to create an event stream:

```
    /* Define variables and create a CFArray object containing
       CFString objects containing paths to watch.
     */
    CFStringRef mypath = CFSTR("/path/to/scan");
    CFArrayRef pathsToWatch = CFArrayCreate(NULL, (const void **)&mypath, 1,
NULL);
    void *callbackInfo = NULL; // could put stream-specific data here.
    FSEventStreamRef stream;
    CFAbsoluteTime latency = 3.0; /* Latency in seconds */

    /* Create the stream, passing in a callback, */
    stream = FSEventStreamCreate(NULL,
        &myCallbackFunction,
        callbackInfo,
        pathsToWatch,
        kFSEventStreamEventIdSinceNow, /* Or a previous event ID */
        latency,
        kFSEventStreamCreateFlagNone /* Flags explained in reference */
    );
```

Once you have created an event stream, you must schedule it on your application's run loop. To do this, call `FSEventStreamScheduleWithRunLoop`, passing in the newly-created stream, a reference to your run loop, and a run loop mode. For more information about run loops, read *Run Loops*.

If you don't already have a run loop, you will need to devote a thread to this task. After creating a thread using your API of choice, call `CFRunLoopGetCurrent` to allocate an initial run loop for that thread. Any future calls to `CFRunLoopGetCurrent` will return the same run loop.

For example, the following snippet shows how to schedule a stream, called `stream`, on the current thread's run loop (not yet running):

```
    FSEventStreamRef stream;
    /* Create the stream before calling this. */
    FSEventStreamScheduleWithRunLoop(stream, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);
```

The final step in setting up an event stream is to call `FSEventStreamStart`. This function tells the event stream to begin sending events. Its sole parameter is the event stream to start.

Once the event stream has been created and scheduled, if your run loop is not already running, you should start it by calling `CFRunLoopRun`.

Handling Events

Your event handler callback must conform to the prototype for `FSEventStreamCallback`. The parameters are described in the reference documentation for the `FSEventStreamCallback` data type.

Your event handler receives three lists: a list of paths, a list of identifiers, and a list of flags. In effect, these represent a list of events. The first event consists of the first entry taken from each of the arrays, and so on. Your handler must iterate through these lists, processing the events as needed.

For each event, you should scan the directory at the specified path, processing its contents as desired. Normally, you need to scan only the exact directory specified by the path. However, there are three situations in which this is not the case:

- If an event in a directory occurs at about the same time as one or more events in a subdirectory of that directory, the events may be coalesced into a single event. In this case, you will receive an event with the `kFSEventStreamEventFlagMustScanSubDirs` flag set. When you receive such an event, you must recursively rescan the path listed in the event. The additional changes are not necessarily in an immediate child of the listed path.
- If a communication error occurs between the kernel and the user-space daemon, you may receive an event with either the `kFSEventStreamEventFlagKernelDropped` or `kFSEventStreamEventFlagUserDropped` flag set. In either case, you must do a full scan of any directories that you are monitoring because there is no way to determine what may have changed.

> **Note:** When an event is dropped, the `kFSEventStreamEventFlagMustScanSubDirs` flag is also set. Thus, it is not necessary to explicitly check for the dropped event flags when determining whether to perform a full rescan of a path. The dropped event flags are provided purely for informational purposes.

- If the root directory that you are watching is deleted, moved, or renamed (or if any of its parent directories are moved or renamed), the directory may cease to exist. If you care about this, you should pass the flag `kFSEventStreamCreateFlagWatchRoot` when creating the stream. In this case, you will receive an event with the flag `kFSEventStreamEventFlagRootChanged` and an event ID of zero (`0`). In this case, you must rescan the entire directory because it may not exist.

  If you need to figure out where the directory moved, you should open the root directory with `open(1)`, then pass `F_GETPATH` to `fcntl(2)` to find its current path. See the manual page for `fcntl(2)` for more information.

- If the number of events approaches 2^64, the event identifier will wrap around. When this happens, you will receive an event with the flag `kFSEventStreamEventFlagEventIdsWrapped`. Fortunately, at least in the near term, this is unlikely to occur in practice, as 64 bits allows enough room for about one event per eraser-sized region on the Earth's surface (including water) and would require about 2000 exabytes (2 million million gigabytes) of storage to hold them all. However, you should still check for this flag and take appropriate action if you receive it.

As part of your handler, you may sometimes need to obtain a list of paths being watched by the current event stream. You can obtain that list by calling `FSEventStreamCopyPathsBeingWatched`.

Sometimes, you may wish to monitor where you are in the stream. You might, for example, choose to do less processing if your code is slipping significantly behind. You can find out the latest event included in the current batch of events by calling `FSEventStreamGetLatestEventId` (or by examining the last event in the list). You can then compare this with the value returned by `FSEventsGetCurrentEventId`, which returns the highest numbered event in the system.

For example, the following code snippet shows a very simple handler.

```
void mycallback(
    ConstFSEventStreamRef streamRef,
    void *clientCallBackInfo,
    size_t numEvents,
    void *eventPaths,
    const FSEventStreamEventFlags eventFlags[],
    const FSEventStreamEventId eventIds[])
{
    int i;
    char **paths = eventPaths;

    // printf("Callback called\n");
    for (i=0; i<numEvents; i++) {
        int count;
        /* flags are unsigned long, IDs are uint64_t */
        printf("Change %llu in %s, flags %lu\n", eventIds[i], paths[i],
eventFlags[i]);
   }
}
```

Note: If you passed the flag `kFSEventStreamCreateFlagUseCFTypes` when creating the stream, you should cast the `eventPaths` value to a `CFArrayRef` object.

Using Persistent Events

One of the most powerful features of file system events is their persistence across reboots. This means that your application can easily find out what happened since a particular time or a particular event in the distant past. By doing so, you can find out what files have been modified even when your application is not running. This can greatly simplify tasks such as backing up modified files, checking for changed dependencies in multi-file projects, and so on.

To work with persistent events, your application should regularly store the last event ID that it processes. Then, when it needs to go back and see what files have changed, it only needs to look at events that occurred after the last known event. To obtain all events since a particular event in the past, you pass the event ID in the `sinceWhen` argument to `FSEventStreamCreate` or `FSEventStreamCreateRelativeToDevice`.] [On a per-device basis, you can also easily use a timestamp to determine which events to include. To do this, you must first call `FSEventsGetLastEventIdForDeviceBeforeTime` to obtain the last event ID `sinceWhen` argument to `FSEventStreamCreateRelativeToDevice`.

On a per-device basis, you can also easily use a time stamp to determine which events to include. To do this, you must first call `FSEventsGetLastEventIdForDeviceBeforeTime` to obtain the last event ID for that device prior to the specified time stamp. You then pass the resulting value to `FSEventStreamCreateRelativeToDevice`. This is described further in "Special Considerations for Per-Device Streams" (page 18).

When working with persistent events, a commonly-used technique is to combine file system event notifications with a cached "snapshot" of the metadata of files within the tree. This process is described further in "Building a Directory Hierarchy Snapshot" (page 15).

Building a Directory Hierarchy Snapshot

File system events tell you that something in a given directory changed. In some cases, this is sufficient—for example, if your application is a print or mail spooler, all it needs to know is that a file has been added to the directory.

In some cases, however, this is not enough, and you need to know precisely what changed within the directory. The simplest way to solve this problem is to take a snapshot directory hierarchy, storing your own copy of the state of the system at a given point in time. You might, for example, store a list of filenames and last modified dates, thus allowing you to determine which files have been modified since the last time you performed a backup.

You do this by iterating through the hierarchy and building up a data structure of your choice. As you cache this metadata, if you see changes during the caching process, you can reread the directory or directories that changed to obtain an updated snapshot. Once you have a cached tree of metadata that accurately reflects the current state of the hierarchy you are concerned with, you can then determine what file or files changed within a directory or hierarchy (after a file system event notification) by comparing the current directory state with your snapshot.

Important: To avoid missing changes, you must start monitoring the directory *before* you start scanning it. Because of the inherently non-deterministic latency in any notification mechanism on a multitasking operating system, it may not always be obvious whether the action that triggered an event occurred before or after a nested subdirectory was scanned. To guarantee that no changes are lost, it is best to always rescan any subdirectory that is modified during scanning rather than taking a time stamp for each subdirectory and trying to compare those time stamps with event time stamps.

Mac OS X provides a number of APIs that can make this easier. The `scandir(3)` function returns an array of directory entries that you can quickly iterate through. This is somewhat easier than reading a directory manually with `opendir(3)`, `readdir(3)`, and so on, and is slightly more efficient since you will always iterate through the entire directory while caching anyway.

The binary tree functions `tsearch(3)`,, `tfind(3)`, `twalk(3)`, and `tdelete(3)` can simplify working with large search trees. In particular, binary trees are an easy way of quickly finding the cached file information from a particular directory. The following code snippet demonstrates the proper way to call these functions:

Listing 2-1 Using the tsearch, tfind, twalk, and tdelete API.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>
#include <search.h>

int array[] = { 1, 17, 2432, 645, 2456, 1234, 6543, 214, 3, 45, 34 };
void *dirtree;

static int cmp(const void *a, const void *b) {
    if (*(int *)a < *(int *)b) return -1;
    if (*(int *)a > *(int *)b) return 1;
    return 0;
}

void printtree(void);

/* Pass in a directory as an argument. */
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i< sizeof(array) / sizeof(array[0]); i++) {
        void *x = tsearch(&array[i], &dirtree, &cmp);
        printf("Inserted %p\n", x);
    }

    printtree();

    void *deleted_node = tdelete(&array[2], &dirtree, &cmp);
    printf("Deleted node %p with value %d (parent node contains %d)\n",
        deleted_node, array[2], **(int**)deleted_node);

    for (i=0; i< sizeof(array) / sizeof(array[0]); i++) {
        void *node = tfind(&array[i], &dirtree, &cmp);
        if (node) {
```

```
            int **x = node;
            printf("Found %d (%d) at %p\n", array[i], **x, node);
        } else {
            printf("Not found: %d\n", array[i]);
        }
    }
    exit(0);
}

static void printme(const void *node, VISIT v, int k)
{
    const void *myvoid = *(void **)node;
    const int *myint = (const int *)myvoid;
    // printf("x\n");
    if (v != postorder && v != leaf) return;
    printf("%d\n", *myint);
}

void printtree(void)
{
    twalk(dirtree, &printme);
}
```

Two unusual design decisions in this API can make it tricky to use correctly if you haven't used it before on other UNIX-based or UNIX-like operating systems:

- The `tsearch(3)` and `tdelete(3)` functions take the address of the tree variable, not the tree variable itself. This is because they must modify the value stored in the tree variable when they create or delete the initial root node, respectively.

  Even though `tfind(3)` does not modify the value of the root, it still takes the address of the root as its parameter, not the root pointer itself. A common mistake is to pass in the `dirtree` pointer. In fact, you must pass in `&dirtree` (the address of the `dirtree` pointer).

  > **Note:** Despite the seeming consistency, the `twalk(3)` function does *not* take the address of the root, so the ampersand is not needed, and indeed, will cause a crash if you use it.

- The values passed to the callback by `twalk(3)` and the values returned by `tfind(3)` and `tsearch(3)` are the address where the pointer to the data is stored, not the data value itself. Because this code passed in the address of an integer, it is necessary to dereference that value twice—once for the original `address-of` operator and once to dereference the pointer to that pointer that these functions return.

  Unlike the other functions, however, the function `tdelete(3)` does not return an address within the tree where the data is stored. This is because the data is no longer stored in the tree. Instead, it returns the *parent* node of the node that it deleted.

The POSIX functions `stat(2)` and `lstat(2)` provide easy access to file metadata. These two functions differ in their treatment of symbolic links. The `lstat` function provides information about the link itself, while the `stat` function provides information about the file that the link points to. Generally speaking, when working with file system event notifications, you will probably want to use `lstat`, because changes to the underlying file will not result in a change notification for the directory containing the symbolic link to that file. However, if you are working with a controlled file structure in which symbolic links always point within your watched tree, you might have reason to use `stat`.

For an example of a tool that builds a directory snapshot, see the *Watcher* sample code.

Cleaning Up

When you no longer need a file system event stream, you should always clean up the stream to avoid leaking memory and descriptors. Before cleaning up, however, you must first stop the run loop by calling `FSEventStreamStop`.

Next, you should call `FSEventStreamInvalidate`. This function unschedules the stream from all run loops with a single call. If you need to unschedule it from only a single run loop, or if you need to move the event stream between two run loops, you should instead call `FSEventStreamUnscheduleFromRunLoop`. You can then reschedule the event stream, if desired, by calling `FSEventStreamScheduleWithRunLoop`.

Once you have invalidated the event stream, you can release it by calling `FSEventStreamRelease`. When the retain count reaches zero, the stream will be freed.

There are three other cleanup-related functions that you should be aware of under certain circumstances. If your application needs to make certain that the file system has reached a steady state prior to cleaning up the stream, you may find it useful to flush the stream. You can do this with one of two functions: `FSEventStreamFlushAsync` and `FSEventStreamFlushSync`.

When flushing events, the synchronous call will not return until all pending events are flushed. The asynchronous call will return immediately, and will return the event ID (of type `FSEventStreamEventId`) of the last event pending. You can then use this value in your callback function to determine when the last event has been processed, if desired.

The final function related to cleaning up is `FSEventsPurgeEventsForDeviceUpToEventId`. This function can only be called by the root user because it destroys the historical record of events on a volume prior to a given event ID. As a general rule, you should never call this function because you cannot safely assume that your application is the only consumer of event data.

If you are writing a specialized application (an enterprise backup solution, for example), it may be appropriate to call this function to trim the event record to some reasonable size to prevent it from growing arbitrarily large. You should do this *only* if the administrator explicitly requests this behavior, however, and you should *always* ask for confirmation (either before performing the operation or before enabling any rule that would cause it to be performed at a later time).

Special Considerations for Per-Device Streams

In addition to the considerations described in "Handling Events" (page 13), streams created with `FSEventStreamCreateRelativeToDevice`, per-device streams have some special characteristics that you should be aware of:

- All paths are relative to the root of the volume that you are monitoring, *not* relative to the system root. This applies to both the path used when creating the stream and to any path that your callback receives as part of an event.
- Device IDs may not remain the same across reboots (particularly with removable devices). It is your responsibility to ensure that the volume you are looking at is the right one by comparing the UUID.

In addition to the functions provided for systemwide streams, you can obtain the UUID for the device associated with a stream by calling `FSEventStreamGetDeviceBeingWatched`.

You can obtain the unique ID for a device by calling `FSEventsCopyUUIDForDevice`. If this unique ID is different than the one obtained from a previous run, this can mean many things. It could mean that the user has two volumes with the same name, that the user has reformatted the volume with the same name, or that the event IDs have been purged for the volume. In any of these cases, any previous events for the volume do not apply to this particular volume, but they may still be valid for another volume.

If you find that the UUID for a volume matches what was stored on a previous run, but the event ID is lower than the last version you stored, this may mean that the user restored a volume from a backup, or it may mean that the IDs have wrapped around or have been purged. In either case, any stored events you may have for the device are invalid.

Finally, if you are using persistent events, you can also use the function `FSEventsGetLastEventIdForDeviceBeforeTime` to find the last event prior to a time stamp. This event ID is persistent, and can be particularly useful for performing incremental backups.

The time format used is a `CFAbsoluteTime` value, which is measured in seconds since January 1, 2001. For other timestamp formats, you must convert them to this format as follows:

- If you are writing a Cocoa application, you should use an `NSDate` object to perform any conversions, then use `CFDateGetAbsoluteTime` to obtain the corresponding `CFAbsoluteTime` value. (You can transparently pass an `NSDate` object as a `CFDateRef`.)
- If you are starting with a POSIX timestamp in a non-Cocoa application, you should subtract `kCFAbsoluteTimeIntervalSince1970` from the value to convert to a `CFAbsoluteTime` value. Be sure to always use timestamps based on GMT.
- If you are working with a legacy Carbon timestamp in a non-Cocoa application, you would subtract `kCFAbsoluteTimeIntervalSince1904`. Be sure to always use timestamps based on GMT.

For more information about date and time types, you should read *Dates and Times Programming Guide for Core Foundation*.

File System Event Security

The file system events API poses an interesting challenge for security. Because it provides the file system path leading up to changed content, storing that information in a persistent database, it represents a new avenue for information leaks, albeit only of the names of directories.

The file system events API mediates this concern in two ways: permissions and prevention.

File System Permissions and File System Events

The most obvious security concern related to file system events is one of privacy. If Bob can see a list of events from changes to Alice's home directory, Bob might see things that Alice does not want him to see. For example, Alice might have a directory name that coincides with the code name of an unreleased Apple product.

To prevent this potential security leak, users do not receive any events unless the user can reach the modified directory through standard file system permissions.

Note: As a side effect, event IDs presented to a file system events client will not necessarily be consecutive even if the user is monitoring all events on all directories beginning at the root. Only applications running as the root user can be guaranteed to receive all events.

Deleted Files and File System Events

When files or directories are deleted, these events look just like any other file system event. This means that directory names will linger on your computer even after a file is deleted.

It is not possible to remove individual records programmatically. The only way to remove prior entries in the database is to purge all entries prior to a particular entry ID. You can do this by calling `FSEventsPurgeEventsForDeviceUpToEventId` in your application.

While the gzipped data is stored in a series of files in the `.fseventsd` directory at the root level of each volume (accessible only by the root user), you should never work with the data directly, as the format of these files may change at any time.

Preventing File System Event Storage

In some cases, the contents of a volume are sufficiently secret that it is not appropriate to log them. To disable logging on a per-volume basis (for creating a backup volume, for example), you must do the following:

- Create a `.fsevents` directory at the top level of the volume.
- Create an empty `no_log` file in that directory.

So if your volume is mounted at `/Volumes/MyDisk`, you would create an empty file called `/Volumes/MyDisk/.fsevents/no_log`.

Kernel Queues: An Alternative to File System Events

The kernel queues API provides a way for an application to receive notifications whenever a given file or directory is modified in any way, including changes to the file's contents, attributes, name, or length. Your application can also receive notification if you are watching a block or character device and access is revoked through a call to `revoke(2)`.

The kernel queues API also provides a way to monitor child processes and find out when they call `exit(3)`, `fork(2)`, `exec(3)`, and so on. This use of kernel queues is beyond the scope of this document. For more information about kernel queues and processes, you should read the FreeBSD documentation for kernel queues. You can find links to this documentation at http://people.freebsd.org/~jmg/kq.html.

Choosing an Event Mechanism

File system events are intended to provide notification of changes with directory-level granularity. For most purposes, this is sufficient. In some cases, however, you may need to receive notifications with finer granularity. For example, you might need to monitor only changes made to a single file. For that purpose, the kernel queue (kqueue) notification system is more appropriate.

If you are monitoring a large hierarchy of content, you should use file system events instead, however, because kernel queues are somewhat more complex than kernel events, and can be more resource intensive because of the additional user-kernel communication involved.

Using Kernel Queues

The kernel queues (kqueue) and kernel events (kevent) mechanism is extremely powerful and flexible, allowing you to receive a stream of kernel-level events (including file modifications) and to define a set of filters that limit which events are delivered to your application.

To use kernel queues, you must do four things:

- Create a kernel event queue by calling `kqueue`. This function returns a file descriptor for a newly allocated event queue.
- Open a file descriptor for each file that you wish to watch.
- Create a list of events to watch for. To do this, use the `EV_SET` to fill in the fields of a kernel event structure. The prototype is as follows:

  ```
  EV_SET(&kev, ident, filter, flags, fflags, data, udata);
  ```

  The first argument, `kev`, is the address of the structure itself. The second, `ident`, contains a file descriptor for the file you are watching.

The third argument, `filter`, contains the name of the kernel filter whose results you want to see. For example, you can use `EVFILT_VNODE` to monitor vnode operations on the file.

The remaining arguments are all specific to a particular filter and are described in the manual page for `kevent(2)`.

- Call `kevent` in a loop. This function monitors the kernel event queue for events and stores them in a buffer that you provide. The prototype is as follows:

```
int kevent(int kq, const struct kevent *changelist,
     int nchanges, struct kevent *eventlist,
     int nevents, const struct timespec *timeout);
```

Its arguments are (in order) the queue file descriptor, the list of events to watch for (from the previous step), the number of events in that list, temporary storage space for the resulting event data, the size of that storage, and a timeout.

On success, the `kevent` function returns the number of events returned. If the timeout expires before any event occurs, it returns 0. Depending on the nature of the error, errors may be reported either as an event with the `EV_ERROR` flag set and the system error stored in the `data` field or by returning -1 with the error stored in `errno`.

A Brief Example

Listing A-1 is a brief example that shows how to monitor a single file using kernel queues. For a more complex example that monitors directories, look at the *FileNotification* sample code.

Listing A-1 Watch a File Using Kernel Queues

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>
#include <errno.h>
#include <string.h>
#include <inttypes.h>

#define NUM_EVENT_SLOTS 1
#define NUM_EVENT_FDS 1

char *flagstring(int flags);

int main(int argc, char *argv[])
{
    char *path = argv[1];
    int kq;
    int event_fd;
    struct kevent events_to_monitor[NUM_EVENT_FDS];
    struct kevent event_data[NUM_EVENT_SLOTS];
    void *user_data;
```

```
    struct timespec timeout;
    unsigned int vnode_events;

    if (argc != 2) {
        fprintf(stderr, "Usage: monitor <file_path>\n");
        exit(-1);
    }

    /* Open a kernel queue. */
    if ((kq = kqueue()) < 0) {
       fprintf(stderr, "Could not open kernel queue.  Error was %s.\n", strerror(errno));
    }

    /*
       Open a file descriptor for the file/directory that you
       want to monitor.
     */
    event_fd = open(path, O_EVTONLY);
    if (event_fd <=0) {
        fprintf(stderr, "The file %s could not be opened for monitoring.  Error was
%s.\n", path, strerror(errno));
        exit(-1);
    }

    /*
       The address in user_data will be copied into a field in the
       event.  If you are monitoring multiple files, you could,
       for example, pass in different data structure for each file.
       For this example, the path string is used.
     */
    user_data = path;

    /* Set the timeout to wake us every half second. */
    timeout.tv_sec = 0;        // 0 seconds
    timeout.tv_nsec = 500000000;    // 500 microseconds

    /* Set up a list of events to monitor. */
    vnode_events = NOTE_DELETE |  NOTE_WRITE | NOTE_EXTEND |
  NOTE_ATTRIB | NOTE_LINK | NOTE_RENAME | NOTE_REVOKE;
    EV_SET( &events_to_monitor[0], event_fd, EVFILT_VNODE, EV_ADD | EV_CLEAR,
vnode_events, 0, user_data);

    /* Handle events. */
    int num_files = 1;
    int continue_loop = 40; /* Monitor for twenty seconds. */
    while (--continue_loop) {
        int event_count = kevent(kq, events_to_monitor, NUM_EVENT_SLOTS, event_data,
num_files, &timeout);
        if ((event_count < 0) || (event_data[0].flags == EV_ERROR)) {
            /* An error occurred. */
            fprintf(stderr, "An error occurred (event count %d).  The error was %s.\n",
 event_count, strerror(errno));
            break;
        }
        if (event_count) {
            printf("Event %" PRIdPTR " occurred.  Filter %d, flags %d, filter flags %s,
 filter data %" PRIdPTR ", path %s\n",
                event_data[0].ident,
```

```
                event_data[0].filter,
                event_data[0].flags,
                flagstring(event_data[0].fflags),
                event_data[0].data,
                (char *)event_data[0].udata);
        } else {
            printf("No event.\n");
        }

        /* Reset the timeout.  In case of a signal interrruption, the
           values may change. */
        timeout.tv_sec = 0;        // 0 seconds
        timeout.tv_nsec = 500000000;    // 500 microseconds
    }
    close(event_fd);
    return 0;
}

/* A simple routine to return a string for a set of flags. */
char *flagstring(int flags)
{
    static char ret[512];
    char *or = "";

    ret[0]='\0'; // clear the string.
    if (flags & NOTE_DELETE) {strcat(ret,or);strcat(ret,"NOTE_DELETE");or="|";}
    if (flags & NOTE_WRITE) {strcat(ret,or);strcat(ret,"NOTE_WRITE");or="|";}
    if (flags & NOTE_EXTEND) {strcat(ret,or);strcat(ret,"NOTE_EXTEND");or="|";}
    if (flags & NOTE_ATTRIB) {strcat(ret,or);strcat(ret,"NOTE_ATTRIB");or="|";}
    if (flags & NOTE_LINK) {strcat(ret,or);strcat(ret,"NOTE_LINK");or="|";}
    if (flags & NOTE_RENAME) {strcat(ret,or);strcat(ret,"NOTE_RENAME");or="|";}
    if (flags & NOTE_REVOKE) {strcat(ret,or);strcat(ret,"NOTE_REVOKE");or="|";}

    return ret;
}
```

For More Information

For more information about kernel queues, see the manual page for `kqueue(2)`), the *FileNotification* sample code, and the FreeBSD documentation for kernel queues at http://people.freebsd.org/~jmg/kq.html..

Document Revision History

This table describes the changes to *File System Events Programming Guide*.

Date	Notes
2008-03-11	Fixed typos.
2007-10-31	Fixed minor errors in description of how to create event streams.
2007-05-10	Introduces File System Events, a technology for finding out when file data and metadata are modified.

Document Revision History