Programación de eventos del sistema de ficheros

Acerca de este documento

En muchas ocasiones es útil para una aplicación saber cuándo otra aplicación modifica algún fichero del sistema de ficheros. Ejemplo de aplicaciones que pueden necesitar esta información serian un antivirus o una aplicación que necesita conocer cuándo han sido modificado externamente alguno de los múltiples ficheros que componen un proyecto. Este reportaje explica las APIs que proporciona Mac OS X para detectar estos eventos y actuar el consecuencia.

Antes de leer este reportaje es importante conocer los conceptos de lenguaje y de programación que se explican en los tutoriales "El lenguaje Objective-C para programadores C++ y Java" y "Programación Cocoa con Foundation Framework". Podrá encontrar estos tutoriales publicados en MacProgramadores.

Nota legal

Este reportaje ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a cualquier persona interesada en conocer la programación de eventos del sistema de ficheros, y las ventajas que aporta, a bajarse o imprimirse este tutorial.

Madrid, Junio 2009

Para cualquier aclaración contacte con:
fernando@DELITmacprogramadores.org

Tabla de contenido

1. Introducción 4
2. File System Events API 5
 2.1. El ciclo de vida 6
 2.2. Crear un stream 6
 2.3. Registrar el stream en el bucle de sondeo 8
 2.4. Flags de callback 10
 2.5. Eventos persistentes 11
 2.6. Consideraciones de privacidad 12
3. Kernel Events API 16
 3.1. Usar kernel events 16
 3.2. Campos del kernel event 17
 3.3. Pedir kernel events 18
4. Metainformación de directorio 23
 4.1. Información de fichero 23
 4.2. Directorios UNIX 24
 4.3. Operaciones sobre un directorio 26
 4.4. Recorrer directorios en Objective-C 28

1. Introducción

Para la programación de eventos del sistema de ficheros, Mac OS X proporciona dos APIs. Ambas APIs permiten al kernel enviar a las aplicaciones que lo hayan solicitado eventos relativos al sistema de ficheros.

La primera es la **File System Events API**. Esta API se recomienda cuando queremos tener un control de alto nivel que nos permita saber cuándo algún fichero situado en un directorio de una jerarquía de directorios ha sido modificado. Una ventaja de esta API es que permite conocer cambios en los ficheros de una jerarquía de ficheros que se hayan producido en periodos de tiempo en los que nuestra aplicación no estaba ejecutando. Al ejecutar de nuevo nuestra aplicación, ésta podrá preguntar qué cambios se han producido durante su ausencia. Por ejemplo, una aplicación de backup que implemente backups incrementales necesita conocer los ficheros que han sido modificados desde la última ver que se ejecutó.

La File System Events API no proporciona información concreta sobre el cambio que se ha producido, de hecho no indica qué fichero ha sido modificado, únicamente indica en qué directorio se encontraba el fichero modificado. Si queremos tener un control más preciso del cambio producido debemos usar la **Kernel Events API**. Esta API está destinada a detectar cambios en un determinado fichero o directorio, con lo que si queremos detectar cambios en toda una jerarquía de directorios consumimos más recursos que con la File System Events API. A cambio, la Kernel Events API proporciona información más precisa del cambio producido.

Este reportaje estudia por separado cada una de estas APIs. En general, debemos elegir la API que mejor se adapte a nuestras necesidades dependiendo de la aplicación concreta que estemos desarrollando. Para detectar cambios externos a los ficheros de un proyecto posiblemente sea más adecuada la File System Events API. Para implementar un antivirus que esté siempre ejecutando podemos obtener información más precisa de los cambios producidos mediante la Kernel Events API.

2. File System Events API

File System Events API es una tecnología introducida en Mac OS X 10.5. Este mecanismo costa de tres componentes:

1. El *kernel* es el encargado de proporcionar todos los accesos al sistema de ficheros. Además el kernel de Mac OS X implementa un mecanismo de mensajes a las aplicaciones interesadas en conocer eventos del sistema de ficheros.
2. Un *demonio* llamado `fseventsd` es el que se encarga de recibir las notificaciones del kernel y de informar a las aplicaciones interesadas en conocer estos eventos. `fseventsd` implementa un mecanismo de filtrado para informar a las aplicaciones de los cambios producidos sólo en determinadas jerarquías del sistema de ficheros. Spotlight no se registra como cliente de `fseventsd`, sino que se comunica directamente con el kernel.
3. Las *aplicaciones* se registran para recibir notificaciones cuando se producen cambios en el sistema de ficheros. Las notificaciones se reciben a través del bucle de sondeo (run loop).

Es importante recordar que en la File System Events API el nivel de granularidad es el directorio. Es decir, sólo se nos informa de que algo ha cambiado en un directorio, pero no se nos informa de qué fichero es el que ha cambiado en ese directorio. Además, cuando una aplicación se registra para recibir este tipo de eventos, la aplicación debe especificar una latencia, de forma que si se producen varios cambios en el mismo directorio durante ese intervalo de tiempo, la aplicación sólo recibirá una notificación. Este mecanismo reduce el número de mensajes que se envían a las aplicaciones registradas.

Dado que las notificaciones las recibimos cuando el cambio ya se ha producido, File System Events API no es una tecnología adecuada para implementar un antivirus. En este caso es necesario programar una extensión del kernel que detecte peticiones de cambio en el sistema de ficheros a nivel de VFS. Si lo que queremos es detectar cambios en un determinado fichero o directorio es mejor utilizar la Kernel Events API que veremos en el apartado 3. La principal ventaja de la File System Events API es que nos permite conocer cambios que se han producido incluso cuando nuestra aplicación no estaba ejecutando.

La File System Events API está implementada por un conjunto de funciones de Core Services Framework. Para enlazar con este framework debemos de usar la opción `-framework CoreServices` de GCC, o bien añadir este framework al proyecto Xcode. Los nombres de todas estas funciones empiezan por `FSEvent`.

2.1. Ciclo de vida

Por cada directorio que una aplicación quiera vigilar, debe crear un **file system event stream**. El tipo `FSEventStreamRef` es el que representa estos objetos. El ciclo de vida de los file system event stream es el siguiente:

1. La aplicación crea un stream usando la función `FSEventStreamCreate()` o `FSEventCreateRelativeToDevice()`.
2. La aplicación registra el stream en un bucle de sondeo. Para ello utiliza la función `FSEventStreamScheduleWithRunLoop()`.
3. La aplicación llama a `FSEventStreamStart()` para pedir al demonio `fseventsd` que empiece a enviar eventos.
4. File System Events API envía los eventos que se vayan produciendo a la función de callback que registramos en el paso 1.
5. La aplicación llama a `FSEventStreamStop()` para pedir al demonio `fseventsd` que deje de enviar eventos.
6. La aplicación llama a `FSEventStreamUnscheduleFromRunLoop()` para desregistrar al stream del bucle de sondeo.
7. La aplicación llama a `FSEventStreamInvalidate()` para invalidar el stream.
8. La aplicación llama a `FSEventStreamRelease()` para decrementar la cuenta de referencias del stream, y de esta forma liberar su memoria.

En los siguientes apartados vamos a explicar con más detalle cada uno de estos pasos.

2.2. Crear un stream

Los file system event streams pueden ser de dos tipos: **Relativos al host** o **relativos a un dispositivo**. Para crearlos se utilizan respectivamente las funciones:

```
FSEventStreamRef FSEventStreamCreate(
    CFAllocatorRef allocator,
    FSEventStreamCallback callback,
    FSEventStreamContext *context,
    CFArrayRef pathsToWatch,
    FSEventStreamEventId sinceWhen,
    CFTimeInterval latency,
    FSEventStreamCreateFlags flags);

FSEventStreamRef FSEventStreamCreateRelativeToDevice(
    CFAllocatorRef allocator,
    FSEventStreamCallback callback,
    FSEventStreamContext *context,
    dev_t deviceToWatch,
    CFArrayRef pathsToWatchRelativeToDevice,
```

```
    FSEventStreamEventId sinceWhen,
    CFTimeInterval latency,
    FSEventStreamCreateFlags flags);
```

Ambas funciones devuelven un objeto de tipo `FSEventStreamRef`. La primera función crea un stream para todo el sistema de ficheros mientras que la segunda crea un stream para un determinado dispositivo o partición de disco. El parámetro `deviceToWatch` indica el dispositivo que queremos purgar. Para obtener este parámetro debemos usar la primitiva `stat()`.

Normalmente, en el parámetro `allocator` se pasa o bien `NULL` o bien `kCFAllocatorDefault`. Ambos parámetros sirven para usar el gestor de memoria dinámica por defecto.

El parámetro `callback` es un puntero a una función callback con el siguiente prototipo:

```
typedef void (*FSEventStreamCallback)(
    ConstFSEventStreamRef streamRef,
    void* clientCallBackInfo,
    size_t numEvents,
    void* eventPaths,
    const FSEventStreamEventFlags eventFlags[],
    const FSEventStreamEventId eventIds[]);
```

El valor que pongamos en `context` es el valor que recibirá la función de callback en el parámetro `clientCallBackInfo`. La función de callback puede recibir más de un evento. El parámetro `numEvents` sirve para indicar el número de eventos que se están enviando a la función de callback.

Entre `FSEventStreamCreate()` y `FSEventCreateRelativeToDevice()`, la principal diferencia está en que la primera recibe paths absolutos a vigilar en el parámetro `pathsToWatch` (p.e. `"/Volumes/autor/prueba.txt"`), mientras que la segunda recibe un dispositivo a vigilar en `deviceToWatch`, y paths relativos a este dispositivo en `pathsToWatchRelativeToDevice`. Es decir, si el dispositivo está montado en `"/Volumes/autor"`, el path relativo sería `prueba.txt`.

En el parámetro `latency` ambas funciones reciben un `double` con el número mínimo de segundos entre eventos. Esto permite reducir el número de veces que se ejecuta la función callback, así como no enviar eventos para ficheros temporales que se crean y destruyen antes de cumplirse el periodo de latencia. Un valor de entre 3.0 segundos y 5.0 segundos resulta apropiado en la mayoría de las ocasiones.

Recuérdese que al final del ciclo de vida del stream debemos liberarlo con las funciones `FSEventStreamInvalidate()` y `FSEventStreamRelease()`.

En resumen, la forma de crear y destruir un stream sería:

```
// Creamos el stream
NSArray* paths = [NSArray arrayWithObject:@"/Users/flh"];
CFTimeInterval latencia = 3.0;
FSEventStreamRef stream = FSEventStreamCreate(NULL,
    RecibidoEvento, NULL, paths,
    kFSEventStreamEventIdSinceNow,latencia,0);
// Procesamos el stream
....................
// Cerramos e stream
FSEventStreamInvalidate(stream);
FSEventStreamRelease(stream);
```

La forma en que `fseventsd` lleva la cuenta de los eventos producidos en el sistema de ficheros es almacenándolos en un directorio llamado `.fseventsd` por cada dispositivo. A cada evento se le asigna un ID. El parámetro `sinceWhen` hace referencia a este ID, y permite conocer eventos del sistema de ficheros que se han producido cuando la aplicación no estaba ejecutando. En el ejemplo anterior hemos pasado `kFSEventStreamEventIdSinceNow` en este parámetro para indicar que sólo queremos ser informados de nuevos eventos.

Debemos tener en cuenta que la base de datos de eventos de fichero de `fseventsd` no se actualiza cuando se accede al dispositivo usando otros sistemas operativos o versiones de Mac OS X anteriores a la 10.5.

2.3. Registrar el stream en el bucle de sondeo

Antes de poder empezar a recibir eventos, debemos registrar el stream en un bucle de sondeo y ponerlo en ejecución con las funciones:

```
void FSEventStreamScheduleWithRunLoop(
    FSEventStreamRef streamRef,
    CFRunLoopRef runLoop,
    CFStringRef runLoopMode);

Boolean FSEventStreamStart(FSEventStreamRef streamRef);
```

La función `FSEventStreamStart()` devuelve `TRUE` si la operación tiene éxito.

Para parar y desregistrar el stream se usan las funciones:

```
void FSEventStreamStop(FSEventStreamRef streamRef);

void FSEventStreamUnscheduleFromRunLoop(
    FSEventStreamRef streamRef,
    CFRunLoopRef runLoop,
    CFStringRef runLoopMode);
```

El Listado 1 muestra un programa que muestra los eventos de fichero producidos durante 10 segundos. El programa dispone de una función callback que imprime los IDs de los eventos, los nombres de los directorios afectados y sus flags asociados.

```
#import <Foundation/Foundation.h>

void RecibidoEvento(ConstFSEventStreamRef streamRef,
    void* clientCallBackInfo,
    size_t numEvents,
    void* eventPaths,
    const FSEventStreamEventFlags eventFlags[],
    const FSEventStreamEventId eventIds[]) {
    char** paths = eventPaths;
    for (int i=0;i<numEvents;i++) {
        printf("ID:%llu directorio: %s flags:%lu\n",
               eventIds[i], paths[i], eventFlags[i]);
    }
}

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];
    // Crea un stream
    NSArray* paths = [NSArray
        arrayWithObject:@"/Users/fernando"];
    CFTimeInterval latencia = 1.0;
    FSEventStreamRef stream = FSEventStreamCreate(NULL,
        RecibidoEvento, NULL, (CFArrayRef)paths,
        kFSEventStreamEventIdSinceNow,latencia,0);
    // Registra stream
    CFRunLoopRef rl = CFRunLoopGetCurrent();
    FSEventStreamScheduleWithRunLoop(stream, rl,
        kCFRunLoopDefaultMode);
    if (FSEventStreamStart(stream))
        printf("Esperando eventos...\n");
    else {
        printf("Fallo en el registro");
    }
    // Pone en funcionamiento el bucle de sondeo
    CFTimeInterval timeout = 60.0;
    CFRunLoopRunInMode(kCFRunLoopDefaultMode, timeout, TRUE);
    // Desregistra stream
    FSEventStreamStop(stream);
    FSEventStreamUnscheduleFromRunLoop(stream, rl,
        kCFRunLoopDefaultMode);
    FSEventStreamInvalidate(stream);
    FSEventStreamRelease(stream);
    [pool drain];
    return 0;
}
```

Listado 1: Programa que muestra eventos de fichero

El programa utiliza el tipo NSArray para crear un CFArrayRef, ya que estos tipos son tipos puente y se pueden intercambiar. Por el contrario, NSRunLoop y CFRunLoopRef no son tipos puente, con lo que hay que usar las funciones de Core Foundation para crearlos.

Dado que hemos pasado el parámetro TRUE a la función CFRunLoopInMode(), el bucle de sondeo termina en cuando se produce el primer evento de fichero.

En cualquier momento podemos preguntar por los directorios que están siendo vigilados con la función:

```
CFArrayRef FSEventStreamCopyPathsBeingWatched(
    ConstFSEventStreamRef streamRef);
```

2.4. Flags de callback

La función callback del Listado 1 puede recibir en el parámetro eventFlags eventos asociados al cambio producido en cada fichero vigilado.

Normalmente sólo debemos inspeccionar los ficheros del directorio especificado en eventPaths, pero en ocasiones esta información no es tan precisa. En concreto, si recibimos el flag kFSEventStreamEventFlagMustScanSubDirs, se nos está indicando que el cambio se puede haber producido tanto en el directorio como en alguno de sus subdirectorios. Esto se puede deber, o bien a que el demonio fseventsd ha fusionado varios eventos en uno. Por ejemplo, si han cambiado ficheros en los directorios /Users/fernando/Music y /User/fernando/Movies, el demonio fseventsd puede fusionar estos eventos informando de un sólo cambio en /User/fernando, y activar este flag. Este flag también se puede activar cuando hay un problema de comunicación entre el kernel y el demonio que impide determinar con precisión qué directorio ha cambiado.

Si queremos ser informados de que un directorio (o alguno de sus padres) se borra, renombra o mueve, debemos crear el stream usando el flag kFSEventStreamCreateFlagWatchRoot. En este caso la función callback envía un evento con el flag kFSEventStreamCreateFlagWatchRootChanged activado y con el ID a cero. Como path se pasa en nombre anterior al cambio.

Cuando estamos vigilando todo el sistema de ficheros de un host, para informarnos de que un dispositivo se ha montado se nos envía un evento con el flag kFSEventStreamEventFlagMount. En este caso el path apunta al directorio montado. Análogamente, el flag kFSEventStreamEventFlagUnmount se usa para informarnos de que el dispositivo se ha desmontado.

2.5. Eventos persistentes

La principal ventaja de la File System Events API es que almacena eventos del sistema de ficheros aunque la aplicación no esté ejecutando. Cuando la aplicación arranca, puede preguntar por estos eventos. En el apartado 2.2 vimos que las funciones de creación del stream reciben el parámetro `sinceWhen` donde podemos proporcionar el ID del evento a partir del cual queremos ser informados. En el Listado 1 indicamos que queríamos ser informados sólo de los eventos que se produzcan después de crear el stream proporcionando el valor `kFSEventStreamEventIdSinceNow`.

El flag `kFSEventStreamEventFlagHistoryDone` de la función callback se activa cuando terminan de enviarse los eventos de historial anteriores a la creación del stream. Cuando se recibe este flag, el path no es válido y debe de ignorarse.

Cuando una aplicación se cierra, podemos almacenar el ID o la fecha hasta donde la aplicación ha sido informada de los eventos de fichero. Al volver a ejecutar la aplicación podemos seguir procesando eventos desde esta fecha. La siguiente función nos permite consultar cuál es el último ID anterior a una determinada fecha. Dado que cada dispositivo lleva una cuenta de IDs distinta, la consulta se debe de hacer para un determinado dispositivo.

```
FSEventStreamEventId
FSEventsGetLastEventIdForDeviceBeforeTime(
     dev_t dev,
     CFAbsoluteTime time);
```

Esta función es especialmente útil para realizar backups incrementales. También podemos obtener el último ID del host con la función:

```
FSEventStreamEventId FSEventsGetCurrentEventId(void);
```

Una consideración a tener en cuenta cuando trabajemos con streams de dispositivo es que pueden existir varios dispositivos con el mismo nombre. Por ejemplo, varios pendrives con el nombre `FER_PEN`. Para evitar que se confundan ambos dispositivos, en el directorio `.fseventsd` de cada dispositivo se almacena un fichero con el nombre `fseventsd-uuid`, el cual almacena un número único para el dispositivo. Podemos obtener este ID único con la función:

```
CFUUIDRef FSEventsCopyUUIDForDevice(dev_t dev);
```

Si nuestro programa almacena el ID del último evento hasta donde ha procesado, no debemos de almacenar el path del dispositivo (ya que puede haber varios dispositivos con el mismo nombre), sino el ID de dispositivo que devuelve esta función.

2.6. Consideraciones de privacidad

Para proteger la privacidad de los usuarios File System Events API sólo informa a los usuarios de eventos en los directorios para los que al menos tengan permiso de lectura. De esta forma un usuario no será informado sobre eventos en el sistema de ficheros de otro usuario. Esto implica que los ID de evento que recibe un usuario no sean consecutivos. El usuario root sí que recibe los ID de eventos consecutivos, ya que este usuario recibe todos los eventos.

En el apartado 2.2 hemos visto que el demonio `fseventsd` almacena logs en el directorio `/.fseventsd`. Aunque este directorio es accesible sólo por el usuario root, cualquier usuario con permiso de administración podría obtener acceso a este directorio. Esto es un problema de privacidad.

La File System Events API también introduce problemas de privacidad para los usuarios que incluso aunque trabajando bajo FileVault. Por ejemplo, imaginemos que un usuario está trabajando en un proyecto secreto y para que otros usuarios no tengan acceso a él, lo desarrolla en una cuenta protegida con FileVault. Otro usuario con permisos de administración no puede acceder a los datos encriptados con FileVault, pero sí que puede conocer los nombres de los directorios que ha creado y destruido el usuario de FileVault.

Para llevar a cabo este análisis basta con que ejecute como root un programa que crea un file system event stream con el ID a 0. Por ejemplo, para obtener información de todos los eventos del sistema de ficheros producidos dentro de un directorio bastaría con ejecutar la función `FSEventStreamCreate()` como root.

El programa del Listado 2 muestra cómo acceder al contenido de `fseventsd`. La forma de ejecutarlo sería desde root:

```
$ su -
Password: ****
# ./ls_fseventsd /Users
```

```
#import <Foundation/Foundation.h>

void RecibidoEvento(ConstFSEventStreamRef streamRef,
        void* clientCallBackInfo,
        size_t numEvents,
        void* eventPaths,
        const FSEventStreamEventFlags eventFlags[],
        const FSEventStreamEventId eventIds[]) {
    char** paths = eventPaths;
    NSMutableSet* dirs = clientCallBackInfo;
    for (int i=0;i<numEvents;i++) {
        NSString* obj_str = [NSString
                             stringWithUTF8String:paths[i]];
        if ( ! [dirs containsObject:obj_str] )
```

```
            [dirs addObject: obj_str];
    }
}

int main (int argc, const char * argv[]) {
    if (argc<2) {
        printf("Indique directorio a analizar\n");
        return 1;
    }
    NSAutoreleasePool * pool = [NSAutoreleasePool new];
    NSMutableSet* dirs = [NSMutableSet new];
    // Crea un stream
    NSArray* paths = [NSArray arrayWithObject:
        [NSString stringWithUTF8String:argv[1]]];
    CFTimeInterval latencia = 1.0;
    FSEventStreamEventId desde = 0;
    FSEventStreamContext context = {0,dirs,0,0,0};
    FSEventStreamRef stream = FSEventStreamCreate(NULL,
        RecibidoEvento, &context, (CFArrayRef)paths,
        desde,latencia,0);
    // Registra stream
    CFRunLoopRef rl = CFRunLoopGetCurrent();
    FSEventStreamScheduleWithRunLoop(stream, rl,
        kCFRunLoopDefaultMode);
    if (FSEventStreamStart(stream))
        printf("Recogiendo directorios de fseventsd...\n");
    else {
        printf("Fallo en el registro");
    }
    // Pone en funcionamiento el bucle de sondeo
    CFTimeInterval timeout = 3.0;
    CFRunLoopRunInMode(kCFRunLoopDefaultMode, timeout, FALSE);
    // Transcurrido timeout muestra
    // los directorios recogidos de fseventsd
    for (NSString* str in dirs) {
        printf("%s\n",[str UTF8String]);
    }
    // Desregistra stream
    FSEventStreamStop(stream);
    FSEventStreamUnscheduleFromRunLoop(stream, rl
        kCFRunLoopDefaultMode);
    FSEventStreamInvalidate(stream);
    FSEventStreamRelease(stream);
    [dirs release];
    [pool drain];
    return 0;
}
```

Listado 2: Programa que muestra el log de File System Events API

La única condición para poder ver los datos bajo FileVault es que el usuario de FileVault esté logado en su cuenta. Cuando no está logado los datos de log realmente siguen estando en el directorio `/.fseventsd`, pero las funciones de File System Events API no muestran estos datos.

Otro problema de privacidad es que aunque borremos un directorio, todavía quedará información sobre este directorio en el sistema de logs de File System Events API.

PAra paliar este problema, Apple proporciona la siguiente función que borra la información histórica de File System Events API:

```
Boolean FSEventsPurgeEventsForDeviceUpToEventId(
    dev_t dev,
    FSEventStreamEventId eventId);
```

Esta operación solo se puede realizar a nivel de dispositivo (no a nivel de host). La operación recibe en el parámetro `eventId` el ID de evento hasta el que queremos borrar.

El Listado 3 muestra un programa que purga el contenido del dispositivo cuyo path pasamos como argumento. El programa obtiene el último ID de evento llamando a la función `FSEventsGetLastEventIdForDeviceBeforeTime()`. Un ejemplo de ejecución para purgar los logs de un pendrive sería:

```
$ sudo ./purge_fseventsd /Volumes/FER_PEN
Se va a purgar hasta el evento 942543745
Unidad purgada correctamente
```

```
#import <CoreServices/CoreServices.h>
#include <sys/stat.h>

int main (int argc, const char* argv[]) {
    if (getuid()!=0) {
        printf("Esta llamada debe hacerla el usuario root\n");
        return 1;
    }
    if (argc<2) {
        printf("Indique dispositivo a purgar\n");
        return 1;
    }
    // Obtiene el dispositivo correspondiente a un path
    struct stat dir_info;
    int err = stat(argv[1],&dir_info);
    if (err) {
        perror("argv[1]");
        return 1;
    }
    // Obtiene el último ID de evento
    CFAbsoluteTime ahora = CFAbsoluteTimeGetCurrent();
    FSEventStreamEventId id_evento =
        FSEventsGetLastEventIdForDeviceBeforeTime(
        dir_info.st_dev,ahora);
    printf("Se va a purgar hasta el evento %llu\n",id_evento);
    // Purga
    Boolean result = FSEventsPurgeEventsForDeviceUpToEventId(
```

```
        dir_info.st_dev, id_evento);
    if (result) {
        printf("Unidad purgada correctamente\n",id_evento);
    } else {
        printf("Fallo el intento de purgar\n");
    }
    return 0;
}
```

Listado 3: Programa que purga un dispositivo

Por último, conviene conocer que podemos desactivar los logs de fseventsd creando un fichero `no_log` en el directorio `.fseventsd`. Una vez desactivados los logs, podemos borrar los ficheros de log del directorio `.fseventsd`. Por ejemplo:

```
$ cd /Volumes/FER_PEN/.fseventsd/
$ touch no_log
$ ls -la
drwxrwxrwx  1 flh  staff  4096 May 24 09:51 ./
drwxrwxrwx  1 flh  staff  4096 May 24 09:51 ../
-rwxrwxrwx  1 flh  staff     0 May 24 09:51 00000000382e1e9e*
-rwxrwxrwx  1 flh  staff    36 May 24 09:51 fseventsd-uuid*
-rwxrwxrwx  1 flh  staff     0 May 24 09:51 no_log*
$ rm 00000000382e1e9e fseventsd-uuid
```

Recuérdese que al comienzo del apartado 2 indicamos que Spotlight tiene su propio sistema de logs. Estos logs se almacenan de forma separada para cada dispositivo en un directorio llamado `.Spotlight-V100`. Podemos usar el comando `mdutil` con la opción `-s` para conocer el estado de los logs de Spotlight:

```
$ mdutil -s /Volumes/FER_PEN
/Volumes/FER_PEN:
      Indexing enabled.
```

Además podemos usar la opción `-i on|off` para activar o desactivar la indexación de dispositivo que hace Spotlight:

```
$ mdutil -i off /Volumes/FER_PEN
/Volumes/FER_PEN:
      Indexing disabled.
```

Y podemos borrar los logs de Spotlight con la opción `-E`:

```
$ mdutil -E /Volumes/FER_PEN
```

3. Kernel Events API

La Kernel Events API es una API POSIX que permite al usuario pedir al kernel notificaciones cuando un determinado evento se produce. Estos eventos están relacionados con objetos POSIX como son los ficheros, sockets, pipes, procesos o señales.

En el kernel de Mac OS X existe un conjunto de **filtros** que detectan estas condiciones. En el espacio de memoria del usuario se crea un **kernel event** (`kevent`) para acceder a estos filtros. El `kevent` está formado por un par de variables (`ident`, `filter`):

- `ident` indica el objeto del kernel a vigilar y su interpretación depende del filtro, pero frecuentemente es un descriptor de fichero o un ID de proceso.
- `filter` identifica el filtro del kernel. Existen un conjunto de filtros predefinidos que se resumen en la Tabla 1.

A diferencia de la File System Events API, no se usa una función callback para recibir los kernel events. La aplicación recibe estos eventos mediante sondeo. En concreto llama a la función `kevent()`, y ésta devuelve los eventos que hayan ocurrido. Si no existen eventos pendientes de enviar a la aplicación, el hilo de la aplicación se queda bloqueado hasta que se cumpla un timeout.

3.1. Usar kernel events

Para usar kernel events debemos hacerlo en cinco pasos:

1. Crear una **kernel queue** llamando a la función `kqueue()`. Esta función devuelve un entero que hace referencia a la cola en la que el kernel deposita los eventos que envía a la aplicación.

2. Abrir el objeto del kernel que queremos vigilar (descriptor de fichero, ID de proceso, etc). Este objeto también está representado por un entero.

3. Rellenar los campos de una estructura `kevent` por cada evento que queramos registrar. Para rellenar esta estructura se suele usar el macro `EV_SET()`.

4. Llamar a la función `kevent()` en un bucle. A esta función la pasamos un array de objetos `kevent` con los eventos que estamos interesados en vigilar. La función nos devuelve información sobre los eventos que se hayan producido.

5. Cerrar los kernel events y la kernel queue. Al llamar a `close()` sobre los descriptores de fichero se cierra su respectivo kernel event. Al llamar a `close()` sobre la kernel queue se cierra la cola.

3.2. Campos del kernel event

La estructura `kevent` tiene los siguientes campos:

```
struct kevent {
    uintptr_t ident;       /* identifier for this event */
    short     filter;      /* filter for event */
    u_short   flags;       /* action flags for kqueue */
    u_int     fflags;      /* filter flag value */
    intptr_t  data;        /* filter data value */
    void      *udata;      /* opaque user data identifier */
};
```

Para rellenar una estructura `kevent` se suele usar el macro:

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

`ident` contiene objeto del kernel a vigilar (p.e. descriptor de fichero).

`filter` indica el filtro del kernel que detecta el evento. Los filtros predefinidos se resumen en la Tabla 1.

Filtro	Descripción
`EVFILT_READ`	Permite saber cuándo hay datos pendientes de leer en un socket o pipe.
`EVFILT_WRITE`	Permite saber cuándo es posible escribir en un socket o pipe así como el espacio disponible en su buffer de escritura.
`EVFILT_VNONE`	Permite vigilar un fichero o directorio y nos informa cuándo se produce un evento.
`EVFILT_PROC`	Nos permite conocer cuándo se produce un evento sobre un proceso: exit, folk, exec, signal o reap.
`EVFILT_SIGNAL`	Nos permite conocer cuándo se ha enviado una señal a nuestro proceso. Coexiste con las operaciones `signal()` y `sigaction()`.

Tabla 1: Filtros predefinidos

`flags` contiene información adicional asociada al par (`ident`, `filter`) y sus valores más comunes se resumen en la Tabla 2.

Flag	Descripción
`EV_ADD`	Indica que se quiere añadir el evento a la cola. Se

	puede volver a añadir un evento ya añadido a la cola, en cuyo caso se sobrescribe el evento con los nuevos parámetros.
EV_ENABLE	Habilita el evento. Si el flag EV_ADD está activo no es necesario usar este flag ya que por defecto al añadir también se habilita (si el flag EV_DISABLE no está activo).
EV_DISABLE	Deshabilita el evento
EV_DELETE	Elimina el evento de la cola. Al cerrar un descriptor de fichero también se elimina el evento de la cola.
EV_ONESHOT	Hace que el evento se produzca sólo una vez. Una vez producido se elimina de la cola.

Tabla 2: Flags de kernel event

fflags (filter flags) contiene flags específicos de cada filtro. Para el filtro EVFILT_VNONE los fflags indican los eventos de fichero en los que estamos interesados: NOTE_WRITE, NOTE_DELETE, NOTE_RENAME, etc. Para el filtro EVFILT_PROC los fflags indican las operaciones a monitorizar: NOTE_EXIT, NOTE_FOLK, NOTE_EXEC, etc.

data contiene datos específicos para cada filtro.

udata (user data) permite almacenar un puntero a algún dato que luego recibimos cuando se produce el evento.

3.3. Pedir kernel events

El programador de aplicaciones debe crear un array de elemento kevent con los eventos que desea registrar en la kernel queue e inicializar cada uno de ellos con el macro EV_SET(). Una vez tengamos este array, lo registramos en la cola llamando a la función:

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

changelist es un parámetro de entrada con los eventos que queremos registrar en la cola. Su longitud se da en nchanges. La función kevent() devuelve el número de eventos que se hayan recibido. Sus valores se depositan en el parámetro de salida eventlist. La capacidad del array eventlist se da en el parámetro de entrada nevents.

Si no hay eventos pendientes de entregar a la aplicación, la llamada a kevent() se queda bloqueada el tiempo dado en timeout. Si timeout es NULL, la llamada se queda indefinidamente bloqueada.

3.4. Monitorizar ficheros y directorios

El Listado 4 muestra un programa que monitoriza eventos sobre un fichero o directorio durante 20 segundos. La forma de ejecutarlo sería:

```
$ ./monitoriza /Users/fernando
Monitorizando "/Users/fernando" 20 segundos...
Evento ident: 4, filtro -4, flags EV_ADD|EV_CLEAR, filter flags NOTE_WRITE|NOTE_ATTRIB, filter data 0, user data /Users/fernando
```

Tenga en cuenta que Kernel Events API detecta los eventos producidos en el fichero o directorio dado (`/Users/fernando` en este ejemplo), pero no en sus subdirectorios.

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

#define NUM_FDS 1
#define NUM_EVENTOS 1

#define CONCATENA_FLAG(var,valor) if (var & (valor))
{strcat(str,or);strcat(str,#valor);or="|";}

char* strflags(int flags) {
    static char str[512];
    char* or = "";
    str[0] = '\0';
    CONCATENA_FLAG(flags,EV_ADD);
    CONCATENA_FLAG(flags,EV_ENABLE);
    CONCATENA_FLAG(flags,EV_DISABLE);
    CONCATENA_FLAG(flags,EV_DELETE);
    CONCATENA_FLAG(flags,EV_RECEIPT);
    CONCATENA_FLAG(flags,EV_ONESHOT);
    CONCATENA_FLAG(flags,EV_CLEAR);
    CONCATENA_FLAG(flags,EV_EOF);
    CONCATENA_FLAG(flags,EV_ERROR);
    return str;
}

char* strfflags(int fflags) {
    static char str[512];
    char* or = "";
    str[0] = '\0';
    CONCATENA_FLAG(fflags,NOTE_DELETE);
    CONCATENA_FLAG(fflags,NOTE_WRITE);
```

```
    CONCATENA_FLAG(fflags,NOTE_EXTEND);
    CONCATENA_FLAG(fflags,NOTE_ATTRIB);
    CONCATENA_FLAG(fflags,NOTE_LINK);
    CONCATENA_FLAG(fflags,NOTE_RENAME);
    CONCATENA_FLAG(fflags,NOTE_REVOKE);
    return str;
}

int main (int argc, const char * argv[]) {
    if (argc!=2) {
        fprintf(stderr, "Use: monitoriza <path_fichero>\n");
        return 1;
    }
    // Crea una kernel queue
    int kq;
    if ( (kq = kqueue()) < 0) {
        fprintf(stderr,"Error creando kernel queue: %s\n",
            strerror(errno));
        return 2;
    }
    // Abre un descriptor de fichero para el fichero
    // a monitorizar
    const char* path_fichero = argv[1];
    int fd = open(path_fichero, O_EVTONLY);
    if (fd<=0) {
        fprintf(stderr, "Error abriendo fichero: %s\n",
            strerror(errno));
        return 3;
    }
    // Fija el filtro en filter, la operación a realizar por
    // el filtro en flags y la lista de eventos a monitorizar
    // en fflags (filter flags)
    unsigned int filter = EVFILT_VNODE;
    unsigned int flags = EV_ADD | EV_CLEAR;
    unsigned int fflags = NOTE_DELETE | NOTE_WRITE |
        NOTE_EXTEND | NOTE_ATTRIB | NOTE_LINK | NOTE_RENAME |
        NOTE_REVOKE;
    // En udata (user data) guardamos el path
    void* udata = (void*)path_fichero;
    // Rellena el kernel event
    struct kevent change_list[NUM_FDS];
    EV_SET(&change_list[0],fd,filter,flags,fflags,0,udata);
    // Monitoriza 20 segundos
    printf("Monitorizando \"%s\" 20 segundos...\n",
        path_fichero);
    int sondeos = 40;
    while (--sondeos>0) {
        struct timespec timeout;
        timeout.tv_sec = 0; // Segundos
        timeout.tv_nsec = 500000000; // 500 microsegundos
        struct kevent event_list[NUM_EVENTOS];
        int n_eventos = kevent(kq,change_list,NUM_FDS,
            event_list,NUM_EVENTOS,&timeout);
        if ( n_eventos<0 || event_list[0].flags&EV_ERROR ) {
            fprintf(stderr,
```

```
                "Error recobiendo evento numero %d: %s\n",
                strerror(errno));
            break;
        }
        if (n_eventos>0) {
            printf("Evento ident: %d, filtro %d, flags %s, "
            "filter flags %s, filter data %x, user data %s\n",
                   event_list[0].ident,
                   event_list[0].filter,
                   strflags(event_list[0].flags),
                   strfflags(event_list[0].fflags),
                   event_list[0].data,
                   event_list[0].udata);
        }
    }
    close(fd);
    printf("Cerrado %s",path_fichero);
    close(kq);
    return 0;
}
```

Listado 4: Programa que monitoriza un fichero o directorio

3.5. Monitorizar un proceso

El Listado 5 muestra un programa que usa el filtro `EVFILT_PROC` para detectar la terminación de un proceso cuyo PID pasamos como argumento. La forma de ejecutarlo sería:

```
$ ps -xc | grep Safari
 1933 ??         0:00.30 Safari
$ fin_proceso 1933
El proceso 1933 termino
```

La función `kevent()` devuelve un `kevent` en el parámetro de salida `event_list`. Si el proceso no existe cuando ejecutamos `kevent()` se activa el flag `EV_ERROR`. En caso contrario la llamada se queda bloqueada hasta que acaba el proceso.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/event.h>

#define NUM_FDS 1
#define NUM_EVENTOS 1

int main (int argc, const char * argv[]) {
    if (argc<2) {
```

```
        fprintf(stderr,"Indique proceso\n");
        return 1;
    }
    // Crea kernel queue
    int kq = kqueue();
    // Fija el filtro en filter, la operación a realizar por
    // el filtro en flags y la lista de eventos a monitorizar
    // en fflags (filter flags)
    pid_t pid = atoi(argv[1]);
    unsigned int filter = EVFILT_PROC;
    unsigned int flags = EV_ADD;
    unsigned int fflags = NOTE_EXIT;
    // En udata (user data) guardamos el path
    void* udata = 0;
    // Rellena el kernel event
    struct kevent change_list[NUM_FDS];
    EV_SET(&change_list[0],pid,filter,flags,fflags,0,udata);
    // Espera a que se produzca el  evento
    struct kevent event_list[NUM_EVENTOS];
    kevent(kq,change_list,NUM_FDS,
                         event_list,NUM_EVENTOS,NULL);
    if (event_list[0].flags & EV_ERROR)
        printf("No se puede seguir trazar el proceso %s\n",
            argv[1]);
    else
        printf("El proceso %s termino\n", argv[1]);
    close(kq);
    return 0;
}
```

Listado 5: Programa que detecta la terminación de un proceso

4. Metainformación de directorio

Para poder conocer qué cambios se han producido en un directorio es necesario almacenar metainformación sobre el estado previo del directorio. Por ejemplo, podemos almacenar una lista de ficheros y su fecha de modificación antes del último backup. De esta forma podemos determinar qué ficheros han cambiado en la jerarquía.

POSIX proporciona un conjunto de funciones que permiten iterar sobre una jerarquía de ficheros. Dado que su uso no es trivial, vamos a explicar estas funciones en este apartado. Estas funciones se pueden usar tanto cuando estamos usando la File System Events API como cuando estemos usando la Kernet Events API.

4.1. Información de fichero

El comando `stat` nos permite obtener información sobre un fichero o directorio:

```
$ stat /Users/fernando
234881026 2185274 drwxr-xr-x 30 fernando staff 0 1020 "Jun 20
18:42:47 2009" "Jun 20 18:05:34 2009" "Jun 20 18:05:34 2009"
"Mar  1 18:59:46 2009" 4096 0 0 /Users/fernando
```

Este comando se limita a llamar a la correspondiente función POSIX[1]:

```
int stat(const char* path, struct stat* buf);
```

El puntero `buf` es un parámetro de salida. Debe apuntar a una estructura `stat` en la que coloca la siguiente información del fichero:

```
struct stat {
  dev_t    st_dev;   /* device inode resides on */
  ino_t    st_ino;   /* inode's number */
  mode_t   st_mode;  /* inode protection mode */
  nlink_t  st_nlink; /* number or hard links to the file */
  uid_t    st_uid;   /* user-id of owner */
  gid_t    st_gid;   /* group-id of owner */
  dev_t    st_rdev;  /* device type, for special file inode */
  struct timespec st_atimespec; /* last access */
  struct timespec st_mtimespec; /* last data modification */
  struct timespec st_ctimespec; /* last file status change */
  off_t    st_size;   /* file size, in bytes */
  quad_t   st_blocks; /* blocks allocated for file */
  u_long   st_blksize;/* optimal file sys I/O ops blocksize */
```

[1] En realidad llamada a una variante llamada `lstat()` que explicamos más abajo.

```
  u_long   st_flags;  /* user defined flags for file */
  u_long   st_gen;    /* file generation number */
 };
```

Cuando a `stat()` le pasamos un link, devuelve información sobre el fichero apuntado por el link. La función `stat()` tiene una variante llamada `lstat()` que es similar a `stat()` excepto que cuando la pasamos un link devuelve información sobre el propio link.

Existe una versión posterior de `stat()` llamada `getattrlist()`. Esta función permite obtener más atributos sobre un fichero. Puede usar `man getattrlist` para obtener más información sobre esta función.

4.2. Directorios UNIX

En los sistemas UNIX, un directorio es un fichero con un flag de directorio[2]. Cada directorio almacena un conjunto de registros de tipo `dirent` (directory entry). Esta estructura está definida en `dirent.h`. El formato de cada uno de estos registros es:

```
struct dirent {
  ino_t d_ino;                /* file number of entry */
  __uint16_t d_reclen;        /* length of this record */
  __uint8_t  d_type;          /* file type, see below */
  __uint8_t  d_namlen;        /* length of string in d_name */
  char d_name[MAXPATHLEN];    /* maximum path length */
};
```

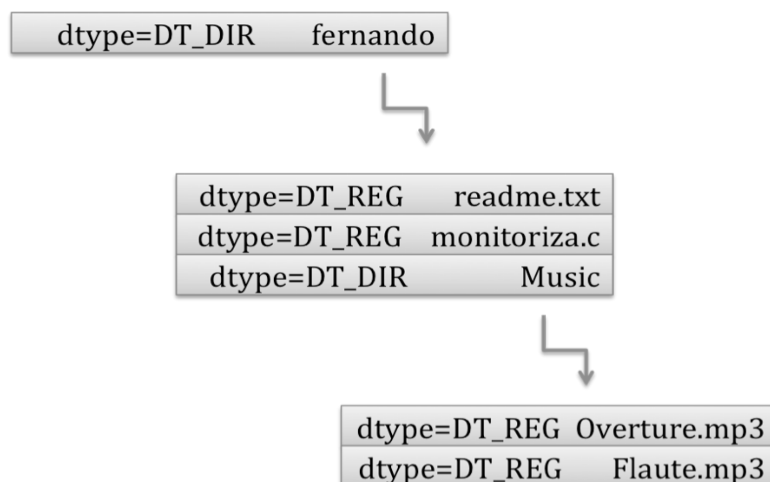

Figura 1: Estructura de directorios

[2] En concreto, en el campo `d_type` de la estructura `dirent` tiene el flag `DT_DIR`.

La Figura 1 muestra una estructura de directorios formada por varios registros `dirent`. Cuando el registro es de tipo `DT_REG`, su contenido es un fichero. Cuando su tipo es `DT_DIR` su contenido es un grupo de registros `dirent`.

Podemos obtener los registros de un directorio con la función:

```
int getdirentries(int fd, char* buf, int nbytes, long* basep);
```

El parámetro `fd` es un descriptor de fichero del directorio a leer. `buf` es un parámetro de salida donde se depositan los registros leídos. `nbytes` es el tamaño del buffer dado en `buf`. `basep` es un parámetro de salida indicando hasta qué posición se ha leído. El valor devuelto en `basep` también se puede obtener ejecutando `fseek()` sobre el descriptor de fichero `fd`.

El Listado 6 muestra un programa que imprime el contenido de los registros `dirent`. La longitud de los registros `dirent` es variable. El Listado 6 usa el campo `d_reclen` en el bucle `while` para saber la longitud de cada registro, y ir así incrementado la posición en el `buffer` con la variable `pos`.

El resultado de su ejecución podría ser:

```
$ dirent /Users/fernando
Tipo: dir  Nombre: .
Tipo: dir  Nombre: ..
Tipo: reg  Nombre: .bash_history
Tipo: reg  Nombre: .DS_Store
Tipo: dir  Nombre: Desktop
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>

int main (int argc, const char * argv[]) {
    if (argc<2) {
        fprintf(stderr, "Indique directorio\n");
        return 1;
    }
    int fd = open(argv[1],O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Error abriendo directorio %s %s",
            argv[1],strerror(errno));
        return 1;
    }
    char* buffer = malloc(64000);
    long base = 345;
    int b_leidos  = getdirentries(fd, buffer, 64000,&base);
    if (b_leidos <= 0) {
```

```
        fprintf(stderr, "Error leyendo directorio %s",
            argv[1]);
        return 1;
    }
    // Muestra los dirent
    struct dirent* dirents = (struct dirent*) buffer;
    int pos = 0;
    while (dirents->d_reclen>0) {
        if (dirents->d_type==DT_DIR)
            printf("Tipo: dir\t");
        else if (dirents->d_type==DT_REG)
            printf("Tipo: reg\t");
        else
            printf("Tipo: otro ");
        printf("Nombre: %s\n", dirents->d_name);
        pos += dirents->d_reclen;
        dirents = (struct dirent*) (buffer+pos);
    }

    free(buffer);
    close(fd);
    return 0;
}
```

Listado 6: Programa que muestra los registros `dirent` de un directorio

4.3. Operaciones sobre un directorio

Para facilitar el acceso a los elementos de un directorio, POSIX introdujo un conjunto de primitivas que vamos a estudiar en este apartado.

Lo primero que tenemos que hacer para acceder a un directorio es abrirlo con:

```
DIR* opendir(const char* dirname);
```

Esta función devuelve un puntero a una estructura de tipo `DIR`, que es la que usan el resto de primitivas.

Para iterar los elementos del directorio se utiliza la operación:

```
struct dirent* readdir(DIR* dirp);
```

Esta operación nos va devolviendo secuencialmente los `dirent` del directorio. Al llegar al último elemento devuelve `NULL`.

Si queremos obtener el descriptor de fichero del `dirent` actual podemos usar la primitiva:

```
int dirfd(DIR* dirp);
```

También podemos realizar accesos aleatorios con las operaciones:

```
void rewinddir(DIR* dirp);
void seekdir(DIR* dirp, long loc);
long telldir(DIR* dirp);
```

Cuando acabemos de recorrer el directorio debemos de liberar la memoria asociada al `DIR*` con la primitiva:

```
int closedir(DIR* dirp);
```

Otra forma de obtener un array de punteros con los elementos de un directorio es usar la operación:

```
int scandir(const char* dirname, struct dirent*** namelist,
    int (*select)(struct dirent*),
    int (*compar)(const void*, const void*));
```

La función coloca en el parámetro de salida `namelist` un array de punteros a registros `dirent`. El parámetro `select` es un puntero a una función que actúa como filtro a la hora de decidir qué `dirent` se quieren obtener. Si `select` es `NULL` se aceptan todos. El parámetro `compar` es un puntero a una función que permite ordenar los elementos. Si vale `NULL` no se ordenan. Es muy común ordenarlos con la función predefinida:

```
int alphasort(const void *d1, const void *d2);
```

Al acabar de usar `namelist` debemos liberar con `free()` la memoria tanto de los `dirent` como del array de punteros.

En el Listado 7 hemos reescrito el programa del Listado 6 usando `scandir()`. También hemos añadido un filtro con la función `se_acepta()` que descarta los ficheros o directorios cuyo nombre empieza por punto (`.`). Además hemos usado la función `alphasort()` para ordenar los nombres de los ficheros y directorios alfabéticamente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>

int se_acepta(struct dirent* d) {
    return d->d_name[0] != '.';
}

int main (int argc, const char * argv[]) {
    if (argc<2) {
        fprintf(stderr,
```

```
            "Indique directorio a inspeccionar\n");
        return 1;
    }
    struct dirent** dirents;
    int err = scandir(argv[1],&dirents,se_acepta,alphasort);
    if (err == -1) {
        fprintf(stderr, "Error en scandir %s\n",
            strerror(errno));
        return 1;
    }
    // Muestra los dirent
    for (int i=0;dirents[i]!=NULL;i++) {
        if (dirents[i]->d_type==DT_DIR)
            printf("Tipo: dir\t");
        else if (dirents[i]->d_type==DT_REG)
            printf("Tipo: reg\t");
        else
            printf("Tipo: otro ");
        printf("Nombre: %s\n", dirents[i]->d_name);
    }
    // Libera el array de punteros
    for (int i=0;dirents[i]!=NULL;i++) {
        free(dirents[i]);
    }
    free(dirents);
    return 0;
}
```

Listado 7: Enumerar directorio con `scandir()`

4.4. Recorrer directorios en Objective-C

Este apartado indica cómo podemos recorrer cómodamente un directorio con Objective-C.

La clase `NSFileManager` dispone del método:

```
- (NSDirectoryEnumerator*)enumeratorAtPath:(NSString*)path
```

El cual devuelve una enumeración recursiva de los ficheros y directorios dentro de un directorio.

La clase `NSDirectoryEnumerator` dispone del método:

```
- (NSDictionary*)directoryAttributes
```

Este método devuelve los atributos del directorio donde se empezó la enumeración.

Además podemos obtener los atributos del fichero o subdirectorio que se está recorriendo en cada momento con:

```
- (NSDictionary*)fileAttributes
```

Aunque el recorrido es recursivo, podemos indicar que no queremos seguir recorriendo los elementos del subdirectorio por el que vamos enumerando con la operación `skipDescendents` de `NSDirectoryEnumerator`:

Por ejemplo, el Listado 8 muestra un programa que recorre recursivamente todos los subdirectorios del directorio home cuyo nombre no empiece por punto (`.`).

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];
    NSDirectoryEnumerator* direnum = [[NSFileManager
        defaultManager]
            enumeratorAtPath:NSHomeDirectory()];
    NSString *nombre;
    while (nombre = [direnum nextObject]) {
        if ([nombre characterAtIndex:0] == '.') {
            [direnum skipDescendents];
        }
        else {
            printf("%s %s\n",[nombre UTF8String],
                [[[direnum fileAttributes] description]
                    UTF8String]);
        }
    }
    [pool drain];
    return 0;
}
```

Listado 8: Recorrido de subdirectorios con Objective-C