Recent Additions for OS X

Catching and reporting file system events

Andrew S. Downs
andrew@downs.ws

Abstract

Not so long ago, patching the file system provided an opportunity to receive notification about operations on files, including creation and deletion. OS X requires a new approach. The Virtual File System concept allows for stacking of file system modules. A VFS can relay file operation details to an application in user space, which is then responsible for informing the user.

Overview

The goal of this paper is to lay out the steps necessary to create a piece of software that responds to file creation "events" by placing an alias to the new file in a well-known location. This paper touches on various aspects of OS X programming, including: Virtual File System (VFS) implementation, crossing the kernel-user space boundary, and the Carbon Toolbox.

To accomplish the goal, we will explore the steps in reverse order, from the easiest to the most difficult.

Informing the user

Although this paper focuses on aliases as the primary means of user feedback, there are other possibilities, including log files, callbacks and Apple Events. These were discussed in some detail under Mac OS Classic in [Downs99]. Most of those approaches should still work in modified form.

Creating an alias

The Alias Manager under Carbon contains functions that convert from full POSIX paths to file references, and vice versa. Depositing the alias in a convenient location is an added bonus. This implementation deposits it on the desktop.

Listing 1 illustrates a function that should run as part of a user space application. The `path` argument is a UNIX path of the form:

```
/rootdir/subdir/filename
```

This path is used to create a FSRef, then an alias, and finally a FSSpec (in the guise of obtaining the catalog info.) The remainder of the code behaves as it did back in System 7.

```
void CreateAliasFromPath(
  UInt8 * path ) {
  AliasHandle alias;
  OSErr err;
  OSStatus status;
  FSRef ref;
  FSSpec theFSSpec, aliasSpec;
  FInfo theInfo;
  short fileRef, i = 0;

  // Turn the POSIX path into a
  // FSRef. Then create an
  // alias and FSSpec.
  status = FSPathMakeRef( path,
    &ref, NULL );

  err = FSNewAliasMinimal( &ref,
     &alias );

  err = FSGetCatalogInfo ( &ref,
    kFSCatInfoNone, NULL, NULL,
    &theFSSpec, NULL );

  FSpGetFInfo( &theFSSpec,
    &theInfo );

  if ( theInfo.fdType ==
    'APPL' ) {
      theInfo.fdType =
     'adrp';
```

```
    }

    // Feel free to substitute a
    // Pascal-style string copy
    // call here.
    while ( i <
      theFSSpec.name[ 0 ] ) {
      aliasSpec.name[ i ] =
        theFSSpec.name[ i ];
      i++;
    }

    // The desktop is where the
    // alias will get deposited.
    // The FSSpec for the alias
    // file is now complete.
    err = FindFolder(
      kOnSystemDisk,
      kDesktopFolderType,
      kCreateFolder,
      &aliasSpec.vRefNum,
      &aliasSpec.parID );

    // Create and populate a file
    // with the alias.
    FSpRstFLock( &aliasSpec );
    FSpDelete( &aliasSpec );
    FSpCreateResFile( &aliasSpec,
      theInfo.fdCreator,
      theInfo.fdType,
      smSystemScript );

    FSpGetFInfo( &aliasSpec,
      &theInfo );

    theInfo.fdFlags |= 0x8000;
    FSpSetFInfo( &aliasSpec,
      &theInfo );
    fileRef = FSpOpenResFile(
      &aliasSpec, fsCurPerm );

    if ( fileRef == -1 ) return;

    AddResource( ( Handle )alias,
      rAliasType, 0,
      theFSSpec.name );

    CloseResFile( fileRef );
}
```

Listing 1. Creating an alias file from a path.

That wasn't too hard. But how do we obtain the path? Let's look at a BSD mechanism for transporting such data out of the kernel.

Crossing from BSD to user space

To facilitate sharing of data with processes outside of the kernel, BSD supports a mechanism called a sysctl. A sysctl is a registered function that exists in the kernel, but is accessible from outside the kernel. A sysctl is callable by name, and has an associated handler function residing in the kernel that can accept arguments and return values. Refer to the Kernel Programming OS X document for additional details.

The neat thing is that kernel-resident code can register new sysctl functions, named as we see fit and returning values we define. We need to set up and register a sysctl that returns a file path. The approach is similar to registering a new Gestalt handler. See Figure 1.

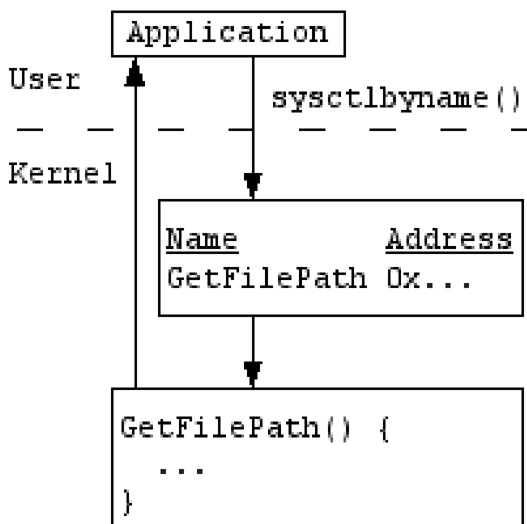

Figure 1. A sysctl resides in the kernel but is callable from user space. The handler function is what actually gets invoked.

Before we look at the code, let's briefly discuss the sysctl flow of information. Ideally the data flow could be unidirectional, from the file system to the client app in user space. New file info would get "pushed." Unfortunately, this is not easy to do in the BSD system. As a compromise, I resorted to the second most evil form of IPC, polling.

(In my opinion a shared file is worse.) Here the client app polls the file system, periodically requesting the file data. So this is a pull, rather than push, model.

The sysctl illustrated in Listing 2 places the outgoing path information in a char * variable. This variable corresponds to an argument passed by the caller that is of appropriate type and size to hold the returned value.

Here, the handler retrieves the last file path and returns it to the caller. A slightly more advanced model would allow for multiple file paths to be stored, returning the next oldest path to the caller when requested. A circular queue or fixed array of strings would be useful here. To keep things simple, the caller does not keep track of what files or indices it has received, it simply requests the next one.

This leads to a couple of possibilities:

1. Have the file system (which contains the handler) recycle slots in the path array as new files are created, EVEN IF the client has not requested the ones being overwritten. This means the client will always receive the most recent file paths, but may miss some older ones.

2. Have the file system stop adding new file paths until the client catches up. The client may miss some newer files.

I would choose the first option (don't wait for the user space app), though the program complexity is about the same for each.

```
// The variable of interest to
// the caller. It will get
// dynamically set within the
// file system code.
char * namePtr = NULL;

// Declaration of the handler.
static int myhandler
  SYSCTL_HANDLER_ARGS;

// This macro expansion is
// defined in sys/sysctl.h, as
// well as Kernel Programming.
SYSCTL_PROC( _hw, OID_AUTO,
getCreatePath,
CTLTYPE_STRING|CTLFLAG_RW,
&namePtr, 0, &myhandler, "A",
"Return path to new file." );

// This function gets called to
// set up the sysctl when the
// file system loads.
void doLoad( void ) {
  // Kernel-safe memory
  // allocation.
  MALLOC( namePtr, char *, 256,
    M_TEMP, M_WAITOK );

  // Register the sysctl.
  if ( namePtr != NULL )
    sysctl_register_oid(
    &sysctl__hw_getCreatePath );
}

void doUnload( void ) {
  if ( namePtr != NULL ) {
    // Kernel-safe freeing.
    FREE( namePtr, M_TEMP );

    // Unregister the sysctl.
    sysctl_unregister_oid(
    &sysctl__hw_getCreatePath );
  }
}

// The actual handler.
static int myhandler
  SYSCTL_HANDLER_ARGS {
  int error;

  // The req param gets
  // generated by the macro
  // expansion. This call
  // "returns" the value in
  // namePtr (the path) as well
  // as its length.
  error =
    SYSCTL_OUT( req, namePtr,
      strlen( namePtr ) );

  return error;
}
```

Listing 2. Returning file path to the client.

The next listing shows the invocation of the sysctl by the user space app. This

can be wrapped in a block that polls by sleeping for some duration before calling the sysctl one or more times.

```
void GetFilePath( void ) {
  // Call the sysctl by name.
  char *name="hw.getCreatePath";

  // Store the path data coming
  // back from the sysctl.
  char path[ 256 ] = "";
  int retval = 0;

  // Size of buffer being passed
  // to sysctl. On return
  // contains length of string.
  size_t len = 256;

  retval = sysctlbyname( name,
    &path, &len, NULL, 0 );

  if ( retval == noErr )
    CreateAliasFromPath( path );
}
```

Listing 3. Calling the sysctl.

Although the sysctl wasn't quite as simple as creating an alias, it may feel familiar because of its similarities to the Gestalt mechanism.

Now it's time to figure out how to generate the path data.

Virtual File Systems

The Virtual File System (VFS) mechanism provides a flexible approach for adding functionality to existing file systems. A VFS is a module containing a set of functions that can be dynamically added to a "stack" of known operations on a file system and, underneath, an actual device. See Figure 2.

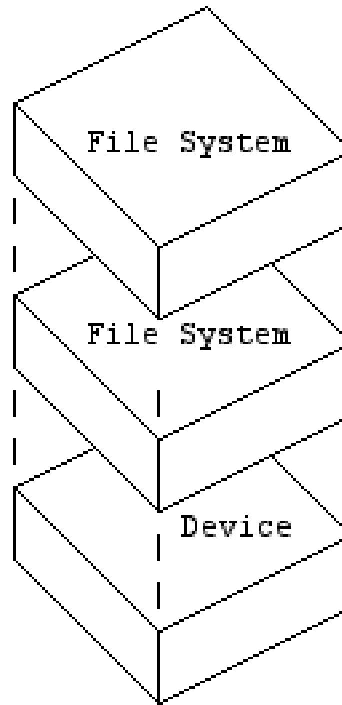

Figure 2. VFS stacking.

Concept

A VFS gets mounted like any other file system. However, as an abstraction layer its functions get hooked into the stack of operations for the file system at the specified mount point.

Most of the functions in a VFS will eventually call through to the underlying layer (another VFS or the actual device.) Some should not; for examples see the file `null_vnops.c`. Figure 3 illustrates this call chain concept.

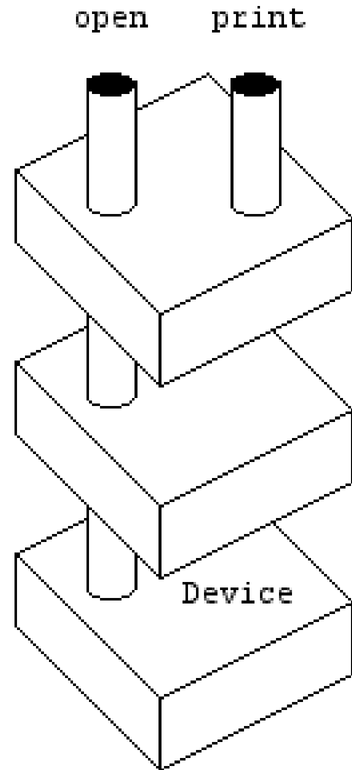

Figure 3. VFS function call chain. Most functions, such as open, will call the next lower-layer, but some (e.g. print) will not.

The VFS implementation discussed in this section reads and retains a copy of some of the information passed between layers, then calls the underlying layer via a `bypass()` routine. It does not modify any expected file system behavior.

Implementation

A VFS implements a set of functions supporting a range of operations on files (e.g. create, open) and the file system itself (e.g. mount, unmount). This particular file system watches for calls to open and lookup. It determines path information for the file in question, and then stores that data until requested (via the sysctl described previously.) One goal in this part of the code is to reduce the amount of time spent in the kernel and file system, so not much is done aside from generating the path info.

This implementation suffers from an inability to dynamically determine the current directory during a file operation. This makes it necessary to track directory changes separately, an activity fraught with peril, and almost guaranteed to break in the future. But for now, it is readily available and simple to setup.

Listing 4 simply shows the calls to the sysctl setup and teardown functions discussed earlier. These occur during file system mounting and unmounting.

```
// This header contains the
// doLoad/doUnload prototypes.
#include "sysctlnullfs.h"

// Mount null layer.
int nullfs_mount(
  struct mount *mp,
  char *path,
  caddr_t data,
  struct nameidata *ndp,
  struct proc *p ) {
  // snip
  doLoad();
  return (0);
}

// Free reference to null layer.
int nullfs_unmount(
  struct mount *mp,
  int mntflags,
  struct proc *p ) {
  // snip
  doUnload();
  return (0);
}
```

Listing 4. Calling the sysctl setup and teardown functions.

Listing 5 shows the lookup function, which gets invoked during the path resolution process. For example, when copying a file from one directory to another, a series of lookups occur that check for the existence of:
- the source dir

- the source file
- the destination dir
- any intermediate dirs
- the destination file

This last check is presumably to make sure the destination file does not already exist prior to the copy.

```
int null_lookup(
  struct vop_lookup_args *ap ) {
  int error;
  int dir = 0;

  if ( namePtr != NULL ) {
    // An absolute path is a
    // tip-off that a directory
    // change is in progress.
    if ( ap->a_cnp->cn_pnbuf[0]
      == '/' ) {
      // Store the dir in a
      // separate variable.
      strcpy( currDir,
        ap->a_cnp->cn_pnbuf );
      strcat( currDir, "" );
      // Flag this as a dir
      // change.
      dir = 1;
    }
    // Check flag set in open
    // handler.
    else if ( isOpen == 1 ) {
      // An open() followed by a
      // lookup() signals file
      // creation.
      strcpy( currFile,
        ap->a_cnp->cn_pnbuf );
      strcat( currFile, "" );
    }

    // Reset flag.
    isOpen = 0;

    // Zero-out path.
    strcpy( namePtr, "" );

    // If this is not a dir
    // change in progress, build
    // the full path.
    if ( dir == 0 ) {
      strcpy( namePtr,
        currDir );
      len = strlen( namePtr );

      // Path leading up to the
      // filename must end with
      // a slash.
      if ( namePtr[ len ]
        != '/' )
          strcat( namePtr, "/" );

        // Append the file name to
        // the path. Path is
        // now complete.
        strcat( namePtr,
          currFile );
      }
    }

  // snip
  error = null_bypass(ap);

  return error;
}
```

Listing 5. Logging directory changes and file lookups.

```
int null_open(
  struct vop_open_args *ap ) {
  int result;
  // snip
  result = (null_bypass(ap));

  // Set flag. lookup() should
  // follow open().
  isOpen = 1;

  return result;
}
```

Listing 6. Flagging the open operation.

A special case involves a "cd" to the root dir. Listing 7 illustrates the handling of this situation.

```
int null_access(
  struct vop_access_args  *ap ) {
  struct mount *tempMount;

  tempMount =
    ( struct mount *)
      (ap->a_vp->v_mount );

   if ( tempMount > 0 ) {
    // Set path to root of fs.
    strcpy( currDir,
      tempMount->
      mnt_stat.f_mntonname );
    strcat( currDir, "" );
  }
```

Listing 7. Handle cd to root dir.

Mounting and unmounting

The VFS module must be built as a kernel extension (KEXT). It can then be loaded and the file system mounted manually using the following shell commands:

```
shell% kextload -v nullfs.kext
shell% ./mount_null /docs /tmpfs
```

A file copy operation will result in the creation of an alias to the new file:

```
shell% cp notes.txt notescpy.txt
```

Unmounting the file system is the reverse of the above:

```
shell% umount /tmpfs
shell% kextunload nullfs.kext
```

Conclusion

This project is by no means complete or robust. It suffers from an inability to determine the current directory. If file system operations are threaded this could result in the generation of an incorrect path.

Another nice-to-have feature would be a tie-in with volfs. the OS X file system that provides path translation between Carbon and BSD. Layering over volfs and intercepting those calls would probably eliminate the need for tracking the directory, though we still need to know about specific file operations.

Still, many of the concepts remain useful and necessary in order to accomplish the goal of relaying file operations to the user.

Bibliography

[Apple02] Apple Computer, Inc. QA 1113: The "/.vol" directory and "volfs". February, 2002.

[Bovet01] Bovet, Daniel P. and Mario Cesati. Understanding the Linux Kernel. O'Reilly & Associates, Inc., 2001.

[Downs99] Downs, Andrew. Watching the File System. Presented at MacHack 14, June 1999.

[Giampaolo99] Giampaolo, Dominic. Practical File System Design. Morgan Kaufmann Publishers, Inc., 1999.

[McKusick96] McKusick, Marshall et al. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley Longman, Inc., Boston, MA. 1996.

[Stevens92] Stevens, W. Richard. Advanced Programming in the UNIX Environment. Addison-Wesley Publishing Company, Reading, MA. 1992.