File Manager Extensions Files on nets,shared programs and multitasking

#include <Files.h>

File access modes that support the three original **File Manager** access modes (read/write, read only, or "anything available") are inadequate for dealing with situations other than those involving a single writer and/or multiple readers. With the increasing use of local area networks and multitasking operating systems, Apple created new access modes based on the ability to deny access.

The new "deny modes" create four ways to look at a file, called: "browsing," "exclusive access," "single writer with multiple readers," and "multiple writers."

Browsing is the same as read-only access, except that many readers can have access to the same file at the same time.

Exclusive access gives one user at at time read/write access. Any attempt to open an exclusive access file that is already open will fail.

Single writer with multiple readers lets one person read and write to a file while many users can see what it looked like as of the last time it was changed. Range locking prevents the file from being browsed while the writer is engaged in updating it.

Multiple writers, or shared access lets several users read and make changes to a file, although "range locking" is used to keep users from accessing information while it is changing.

Mode Translation

The most widespread use of the new modes is generally in an AppleShare environment. To ensure that old applications work over networks, AppleShare's external file system translates old, single-system permissions into the new fileserver-environment permissions. AppleShare (and only AppleShare) adopts a "single writer or multiple readers, but not both" rule of file access as the basic permissions level for all files. This keeps applications from damaging each other's files and avoids confusing the file's users since it also means that the only time more than one user can simultaneously access a file is if they both have read-only permission.

Under the new **File Manager** extensions, standard HFS permissions are translated: read only becomes browsing (read/deny write); read/write becomes write only, and whatever's available(read/write if possible--otherwise read only)becomes either exclusive(read/write/deny read/deny write) or browsing but not both; and the old shared read/write becomes the new shared access (read/write, no denial of permission).

Not only are individual files identified on the basis of their access modes, but another category of user-access restraint, called "access privileges", is assigned by a system's owner on a folder-by-folder basis. Access privileges define a user as: the folder's "owner", a member of a "group"; "everyone" (meaning all users who can access the file server); "see folders," (which lets the user see other folders within a folder); "see files" (which lets users see

the icons and open the applications made available by "see folders); and "make changes" (which lets users create, modify, rename or delete the files and folders).

Network Considerations

Aside from additional complications of ownership, connection to a network imposes requirements in the way you have to deal with other aspects of writing a program. You can't assume, for example, that a needed resource will be available and your program has to be ready with an appropriate error message should an operation fail. Among other things, files on a network can be removed by one user without other users' knowledge. Files can exist invisibly so users may find that they're attempting to create file names that have already been taken. Users can find that they are denied access to a file, either because another user has already opened it, or because they don't have the proper access privileges. Failures in the network or on the file server can result in failures when users try to read or write to a network dependent file. In all of these cases, users need to be told why they're not being allowed to do whatever it is they're attempting.

A check on the availability of resources will let your program search for and reserve needed elements before it tries to use them. This is especially important if your application performs such functions as creating temporary files. If the temporary file names you want are already in existence, you need to either rename your files or tell the user that there are problems ahead.

A networked environment also means that applications and data will be shared among more than one user. In the case of data file sharing, you'll need to make sure that updates can only be done by one user at a time. For applications, you'll want to make sure that simultaneous use is normal and natural, with each user having full access to all program functions all the time.

Byte-Range Locking

With range locking you can let users have exclusive control of parts of files whenever they have those files open. The **PBLockRange** call is specifically designed to let one user change a file while keeping others away for the duration of the update. The ioPosMode field in **PBLockRange**'s HParamBlockRec uses bits 0 and 1 to position the start of the range you want reserved. The call **PBUnlockRange** frees that portion of the file once the user's modifications are complete. Before calling **PBUnlockRange**, however, you have to reset the ioPosOffset field since the Read and Write calls the user made while working on the file will have changed it .

For file updates, your application needs to: 1) determine the range of bytes affected by the update; 2) call **PBLockRange**; 3) call **PBUnlockRange**. If a user's lock request fails on successive attempts, your application should post an error message indicating that the file is already being used.

If an update is going to be appended to the end of a file, your application should lock a range that includes the logical end-of-file and the last byte the file can possibly address, and then write into that area By taking up the entire range, your application makes sure that two users can't append data simultaneously.

Conversely, when a user's making a change that cuts data from a file, your application should lock the entire file until the cut is completed. Thus, user's

aren't put in the situation of using data that somebody else is deleting.

Program Sharing

Programs that allow multiple users can either allow two or more users to change data in the same file (multi-user) or allow more than one copy of the application to run simultaneously (multi-launch). Along with single-user and single-launch applications, the full range of permutations looks like:

	Program Sharing: Single Launch	Program Sharing: Multiple Launch
One User	One user at a time can launch and use a program and modify files.	Many users can launch and use one program but only one at a time can change a file.
Many Users	One user can launch and use a program but many users can modify the same files.	Many users can launch and use one program, as well as make changes to the same file at the same time.

For organizational purposes, you'll want to make sure that your program coordinates the many users of a single file, includes an update mechanism so that each user knows when somebody else makes a change, and use byte-range locking so that, for instance, only one user at a time can change a particular section of a file.

The requirements are different for multiple launch programs in that you'll want to use either ResEdit or FEdit to set the multi-launch or shared bit in the program's Finder data.

You may also want to give some thought to just how many multiple launches you want to allow. If you set a limit on the number of multiple launches you'll have to have some way of counting the users as they launch and quit. Since launch/quit counting schemes can be complex, Apple recommends that you at least make things easier for yourself. For example, require that any temporary files associated with the program be in the same folder as the program. This way, the application simply counts the number of temporary files (all in the same folder) and bars further launches when your pre-established maximum limit is reached.

Networking Hints

1) Make sure your application tries out the new open calls and checks for a paramErr indicating that the file doesn't exist on the file server. What you'll need to do in that case is open the file with an old-style call. A network-oriented open call to a local volume yields an error indicating that the local system can't handle a network call.

2) Let users know what kind of access they have. This lets the application respond correctly to errors under the error reporting mechanism in use between the file server and an application running on the user's machine.

3) Use the ROM **Scrap Manager** to access the Scrapbook to let different applications share the resources in the scrap.

4) Keeping program segmentation swapping low is especially important on a network when applications are launched from a file server. Dynamic swapping over the network exaggerates the effect of such swapping and lowers file server performance.

Among the list of things not to do:

1) Don't have your programs write to themselves since information specific to one user will be saved in an area that is accessible by all users. You'll wind up with users overwriting each others' data.

2) Don't use the **Resource Manager** to structure a multi-user program's data in a resource fork since the **Resource Manager** always assumes it has sole authority to modify a file. More than one write-access path in a resource fork leaves each of them with no way of letting the others know that a change took place. Programs that use resource files for document storage can't share data. You'll have to come up with another storage scheme if you want your program to be functional as a multi-user or multi-launch network application.

3) Don't let your application close a file while changes are being made to a copy of it on a local workstation. Keep the application open at all times so other users won't be making changes on top of the first user's changes.

4) If your program gives fixed names to temporary files and more than one user is creating temporary files, you'll wind up with an attempt to create duplicate file names.

5) Use the **File Manager** calls to access file and volume control blocks (or other system data structures) instead of trying to directly manipulate them in memory. If you don't use the **File Manager**, you won't be giving the file system a chance to perform its updates. You could easily run into situations where misinformation is propagated concerning the data on file server volumes.

6) Don't try to use an allocation function to set the logical end-of-file. AppleShare doesn't support it. Use SetEOF instead to extend a file.

Networking Calls

Calls that support shared environments are installed on startup volumes by an 'INIT' resource patch in the AppleShare file. As a result, the only volumes that will support the shared environment calls are those whose System Folders contain the AppleShare file. Further, the patch only handles external file system volumes. Using the shared environment routines on local volumes

produces an error.

If you want to find out if your HFS supports shared environment routines, the easiest way to find out is to call PBHGetVolParms for the volume in question. If your parameters are correct and you still get a paramErr, you can assume that the volume doesn't support the networking routines. If you get back a noErr, you can explore the volume further to see if it supports all of the other networking functions you want to use.