File System Advanced Programming Topics

Developer

Contents

Listings

About Advanced File System Topics

This document supplements the information in *File System Programming Guide* by describing procedures that are related to the file system but not central to using it.

At a Glance

This document describes advanced techniques for manipulating files.

File Mapping is an Efficient Way to Read Large Files

File mapping is the process of mapping disk sectors associated with a file into the virtual memory space of a process. Mapping is most appropriate when you plan to read small portions of a large file frequently and in random order. Mapping just the portions you need into memory is much more efficient than reading or rereading those sections over and over from disk. Mapping yields little benefit for files you plan to read sequentially anyway.

Relevant chapter: "Mapping Files Into Memory" (page 5)

Directories Can Have Localized Names

You can provide localized names for any user-visible directories that your code creates. Localized names improve the experience for users by showing directory names in the language they understand. The Finder automatically localizes the names of many system directories but you must provide the appropriate localizations for any custom directories you create.

Relevant chapter: "Localizing the Name of a Directory" (page 10)

Prerequisites

File System Programming Guide is a prerequisite for reading this document. If you have not yet read that document, you should at least understand the content in the first two chapters, which describe the file system organization and techniques for accessing files and directories.

Mapping Files Into Memory

File mapping is the process of mapping the disk sectors of a file into the virtual memory space of a process. Once mapped, your application accesses the file as if it were entirely resident in memory. As you read data from the mapped file pointer, the kernel pages in the appropriate data and returns it to your application.

Although mapping files can offer tremendous performance advantages, it is not appropriate in all cases. The following sections explain when file mapping can help you and how you go about doing it in your code.

Choosing When to Map Files

When deciding whether or not to map files, keep in mind that the overall goal is to reduce transfers between disk and memory. File mapping can help you in some cases, but not all. The more of a file you map into memory, the less useful file mapping becomes.

Before you map any files into memory, make sure you understand your typical file usage patterns. Instruments can help you identify where your application accesses files and how long those operations take. For any operations that are taking longer than expected, you can then look at your code to determine if file mapping might be of use.

File mapping is effective in the following situations:

- You have a large file whose contents you want to access randomly one or more times.
- You have a small file whose contents you want to read into memory all at once and access frequently. This technique is best for files that are no more than a few virtual memory pages in size.
- You want to cache specific portions of a file in memory. File mapping eliminates the need to cache the data at all, which leaves more room in the system disk caches for other data.

You should not use file mapping in the following situations:

- You want to read a file sequentially from start to finish only once.
- The file is several hundred megabytes or more in size. (Mapping large files into memory fills memory quickly and may lead to paging, which would negate the benefits of mapping the file in the first place.)

For large sequential read operations, you are better off disabling disk caching and reading the file into a small memory buffer.

File Mapping Caveats

Even in situations where you think file mapping is ideal, there are still some caveats that may apply. In particular, you may not want to map files in the following situations:

- The file is larger than the available contiguous virtual memory address space. This is less of an issue for 64-bit applications but can be an issue for 32-bit applications.
- The file is located on a removable drive.
- The file is located on a network drive.

When randomly accessing a very large file, it's often a better idea to map only a small portion of the file at a time. The problem with mapping large files is that the file consumes active memory. If the file is large enough, the system might be forced to page out other portions of memory to load your file. Mapping more than one file into memory just compounds this problem.

For files on removable or network drives, you should avoid mapping files altogether. If you map files on a removable or network drive and that drive is unmounted, or disappears for another reason, accessing the mapped memory can cause a bus error and crash your program. If you insist on mapping these types of files, be sure to install a signal handler in your application to trap and handle the bus error condition. Even with the signal handler installed, your application's current thread may block until it receives a timeout from trying to access a network file. This timeout period can make your application appear hung and unresponsive and is easily avoided by not mapping the files in the first place.

Mapping a file on the root device is always safe. (If the root device is somehow removed or unavailable, the system cannot continue running.) Note that the user's home directory is not required to be on the root device.

File Mapping Example

Listing 1-1 demonstrates memory mapping using the BSD-level `mmap` and `munmap` functions. The mapped file occupies a system-determined portion of the application's virtual address space until `munmap` is used to unmap the file. For more information about these functions, see `mmap` and `munmap`.

Listing 1-1 Mapping a file into virtual memory

```
void ProcessFile( char * inPathName )
{
    size_t dataLength;
    void * dataPtr;
    if( MapFile( inPathName, &dataPtr, &dataLength ) == 0 )
    {
        //
        // process the data and unmap the file
        //

        // . . .

        munmap( dataPtr, dataLength );
    }
}


// MapFile
// Return the contents of the specified file as a read-only pointer.
//
// Enter:inPathName is a UNIX-style "/"-delimited pathname
//
// Exit:    outDataPtra     pointer to the mapped memory region
//          outDataLength   size of the mapped memory region
//          return value    an errno value on error (see sys/errno.h)
//                          or zero for success
//
int MapFile( char * inPathName, void ** outDataPtr, size_t * outDataLength )
{
    int outError;
    int fileDescriptor;
    struct stat statInfo;

    // Return safe values on error.
    outError = 0;
    *outDataPtr = NULL;
    *outDataLength = 0;

    // Open the file.
    fileDescriptor = open( inPathName, O_RDONLY, 0 );
    if( fileDescriptor < 0 )
    {
       outError = errno;
    }
    else
    {
        // We now know the file exists. Retrieve the file size.
        if( fstat( fileDescriptor, &statInfo ) != 0 )
        {
            outError = errno;
        }
        else
        {
            // Map the file into a read-only memory region.
            *outDataPtr = mmap(NULL,
                                statInfo.st_size,
                                PROT_READ,
                                0,
                                fileDescriptor,
                                0);
            if( *outDataPtr == MAP_FAILED )
            {
                outError = errno;
            }
            else
            {
                // On success, return the size of the mapped file.
                *outDataLength = statInfo.st_size;
            }
        }

        // Now close the file. The kernel doesn’t use our file descriptor.
        close( fileDescriptor );
    }


    return outError;
}
```

Localizing the Name of a Directory

If your application installs any custom support directories, you can provide localized versions for your directory names. Providing localized names is not required and should be done only for directories whose names you know in advance. It should not be done for any user-specified directories. When the appropriate preferred language is selected, the Finder displays the localized directory names you provide.

> **Note:** Localized directory names appear only when the filename extension hiding is enabled in Finder preferences. Localized names do not appear until the next time the user logs in.

To provide a localized display name for a directory, do the following:

1. Add the extension `.localized` to the directory name.
2. Create a subdirectory inside the directory called `.localized`. (You must create this directory programmatically or using the Terminal application.)
3. Inside the `.localized` subdirectory, create a strings files for each localization you support.

Each strings file is a Unicode text file that contains the appropriately localized name of the directory. The name of each strings file should be an appropriate two-letter language code followed by the `.strings` extension. For example, a localized Release Notes directory with English, Japanese, and German localizations would have the following directory structure:

```
Release Notes.localized/
    .localized/
        en.strings
        de.strings
        ja.strings
```

Inside each strings file, include a single string entry to map the nonlocalized directory name to the localized name. When specifying the original directory name, do not include the `.localized` extension you just added. For example, to map the name "Release Notes" to a localized directory name, each strings file would have an entry similar to the following:

```
"Release Notes" = "Localized name";
```

For information on creating a strings file, see *Internationalization Programming Topics* .

Note: System-defined directories, such as `/System`, `/Library`, and the default directories in each user's home directory, use a localization scheme different from the one described here. For these directories, the presence of an empty file with the name `.localized` causes the system to display the directory with a localized name. Do not delete the `.localized` file from any these directories.

Document Revision History

This table describes the changes to *File System Advanced Programming Topics* .

Date	Notes
2011-06-06	New document describing advanced techniques for working with files and directories.