# File-System Performance Guidelines

Developer

# Contents

# Figures, Tables, and Listings

# Introduction to File-System Performance Guidelines

Accessing file data is one of the biggest bottlenecks to performance on any computer system. Most computers are capable of executing millions of instructions before the hard drive heads are even in position and ready to read a piece of data. As a result, it is imperative that you examine your application's file-system interactions and do what you can to improve them.

## Organization of This Document

This programming topic contains the following articles:

- "File-System Performance Tips" (page 6) provides some general tips for improving your application's file-related code.
- "Overview of OS X File Systems" (page 11) provides a brief overview of OS X file-system performance and how it can impact your application.
- "Examining File-System Usage" (page 13) describes techniques for analyzing your application's file-system interactions.
- "Mapping Files Into Memory" (page 20) describes techniques for minimizing the work done when reading files into memory.
- "Iterating Directory Contents" (page 25) provides an example of how to iterate directories efficiently.
- "Resolving Domain Names" (page 31) describes better-performing alternatives to getting network-based information.
- "Tracking File-System Changes" (page 33) describes the approach your application should take when monitoring the file system for changes to individual files and directories.

# File-System Performance Tips

Given the nature of disk-based storage, the file system can be a significant bottleneck to code. The following sections provide tips on how you can minimize this bottleneck to improve the performance of your code.

## General I/O Guidelines

What follows are some basic recommendations for reducing the I/O activity of your program, and thus enhancing its performance. As with all recommendations, it is important to measure the performance of the code being optimized before and after optimization to ensure that it actually gets faster.

- Minimize the number of file operations you perform. For more information, see "Minimize File-System Access" (page 7).
- Group several small I/O transfers into one large transfer. A single write of eight pages is faster than eight separate single-page writes, primarily because it allows the hard disk to write the data in one pass over the disk surface. For more information, see "Choosing an Optimal Transfer Buffer Size" (page 10).
- Perform sequential reads instead of seeking and reading small blocks of data. The kernel transparently clusters I/O operations, which makes sequential reads much faster.
- Avoid skipping ahead in an empty file before writing data. The system must write zeroes into the intervening space to fill the gap. For more information, see "Be Aware of Zero-Fill Delays" (page 9).
- Reading is typically cheaper than writing data.
- Defer any I/O operations until the point that your application actually needs the data.
- Use the preferences system to capture only user preferences (such as window positions and view settings) and not data that can be inexpensively recomputed.
- Do not assume that caching file data in memory will speed up your application. Storing file data in memory improves speed until that memory gets swapped out to disk, at which point you pay the price for accessing the disk once again. Strive to find an appropriate balance between reading from disk and caching in memory. For more information, see "Cache Files Selectively" (page 8).

## Avoid Making Assumptions

Be careful about making assumptions that a particular file operation will be fast. Something as simple as reading a preferences file might still take a long time if the file is located on a busy network server. If the server crashes, reading the file can take even longer. Always analyze your application with the available tools to find the actual performance problems.

For more information about measuring file access performance, see "Examining File-System Usage" (page 13).

## Minimize File-System Access

Moving data from a local file system into memory takes a significant amount of time. File-system access times are generally measured in milliseconds, which corresponds to several millions of clock cycles spent waiting for data to be fetched from disk. And if the target file system is located on a server halfway around the world, network latency increases the delay in retrieving the data. Because of these factors, you should strive to reduce your application's dependence on files as much as possible.

To find out where your application is accessing the file system, use the `fs_usage` tool. This tool reports any file-system interactions and includes information about how long those interactions take. See "Examining File-System Usage" (page 13) for more information.

## Use Modern File APIs

If you are migrating legacy code to OS X, you should update your file-related code to use more modern APIs. Modern routines that use the `FSRef` data type offer much better performance than the older `FSSpec`-based routines. The reason is that modern routines were written with Unicode and a wide spectrum of file systems in mind and were thus optimized for those environments. Older routines require additional manipulation to work on non-HFS file systems and in non-Roman languages.

If your application requires the maximum possible performance from the file system, consider using BSD function calls to transfer data. For most application developers, this step is unnecessary because the performance of both the Carbon and Cocoa routines is quite acceptable for most uses. However, you might consider using the BSD routines if you are writing a file-system utility or an application that spends a lot of time interacting with the file system.

The BSD layer implements the POSIX routines to `open`, `close`, `read`, and `write` files. You can also use the `fcntl` routine to control the current file-system settings and perform other operations.

## Cache Files Selectively

Disk caching can be a good way to accelerate access to file data, but its use is not appropriate in every situation. Caching increases the memory footprint of your application and if used inappropriately can be more expensive than simply reloading data from the disk.

Caching is most appropriate for files you plan to access multiple times. If you have files you only intend to use once, you should either disable the caches or map the file into memory.

### Disabling File-System Caching

When reading data that you are certain you won't need again soon, such as streaming a large multimedia file, tell the file system not to add that data to the file-system caches. By default, the system maintains a buffer cache with the data most recently read from disk. This disk cache is most effective when it contains frequently used data. If you leave file caching enabled while streaming a large multimedia file, you can quickly fill up the disk cache with data you won't use again. Even worse is that this process is likely to push other data out of the cache that might have benefited from being there.

Carbon applications can tell the File Manager not to cache data by passing the `kFSNoCacheBit` option to `FSReadFork` or similar functions. (In versions of OS X prior to 10.4, this option is specified using the `noCacheBit` flag instead.) Applications can also call the BSD `fcntl` function with the `F_NOCACHE` flag to enable or disable caching for a file.

**Note:** When reading uncached data, it is recommended that you use 4K-aligned buffers. This gives the system more flexibility in how it loads the data into memory and can result in faster load times.

### Using Mapped I/O

If you intend to read data randomly from a file, you can improve performance in some situations by mapping that file directly into your application's virtual memory space.File mapping is a programming convenience for files you want to access with read-only permissions. It lets the kernel take advantage of the virtual memory paging mechanism to read the file data only when it is needed. You can also use file mapping to overwrite existing bytes in a file; however, you cannot extend the size of file using this technique. Mapped files bypass the system disk caches, so only one copy of the file is stored in memory.

For more information about mapping files into memory, see "Mapping Files Into Memory" (page 20).

## Be Aware of Zero-Fill Delays

For security reasons, file systems are supposed to zero out areas on disk when they are allocated to a file. This behavior prevents data leftover from a previously deleted file from being included with the new file.

The OS X HFS Plus file system has always implemented this zero-fill behavior. However, in OS X version 10.1 a new technique was introduced to improve the performance of this operation. For both reading and writing operations, the system delays the writing of zeroes until the last possible moment. When you close a file after writing to it, the system writes zeroes to any portions of the file your code did not touch. When reading from a file, the system writes zeroes to new areas only when your code attempts to read from that area or when it closes the file. This delayed-write behavior avoids redundant I/O operations to the same area of a file.

If you notice a delay when closing your files, it is likely because of this zero-fill behavior. Make sure you do the following when working with files:

- Write data to files sequentially. Gaps in writing must be filled with zeros when the file is saved.
- Do not move the file pointer past the end of the file and then close the file.
- Truncate files to match the length of the data you wrote. For scratch files you plan to delete, truncate the file to zero-length.

## Reuse Computed Path Information

Converting pathname information from one form to another is often an expensive operation. If your code converts back and forth between pathnames, `FSSpec` structures, `FSRef` structures, or `CFURL` structures, you might want to consider caching the resulting data structures. The best time to cache is when you know you are going to need that same structure again. Reusing file-related data structures minimizes the interactions your program has with the file system.

## Use CFNetwork Services

The CFNetwork services provide modern APIs for accessing network-based services, such as those related to HTTP and Bonjour. If you are currently using Open Transport, URLAccess, or other legacy APIs to access network resources, you should move your code to these new services.

## Use Concurrent Asynchronous I/O

OS X version 10.4 and later implements true asynchronous I/O operations in Carbon File Manager routines. In previous versions of the Carbon File Manager, asynchronous I/O operations were offloaded to a separate thread, which queued I/O requests and performed them sequentially. Now, changes to the kernel allow those same operations to be performed in parallel.

In versions of OS X prior to 10.4, if you want to perform truly asynchronous I/O requests, you must add the `kFSAllowConcurrentAsyncIO` bit to the `positionMode` parameter when calling `PBReadForkAsync` or `PBWriteForkAsync`.

## Choosing an Optimal Transfer Buffer Size

When reading data from the disk to a local buffer, the buffer size you choose can have a dramatic effect on the speed of the operation. If you are working with relatively large files, it does not make sense to allocate a 1K buffer to read and process the data in small chunks. Instead, it is advisable to create a larger buffer (say 128K to 256K in size) and read much or all of the data into memory before processing it. The same rules apply for writing data to the disk: write data as sequentially as you can using a single file-system call.

# Overview of OS X File Systems

The following sections discuss the file systems supported by OS X and the impact they can have on application performance.

## Supported File Systems

OS X supports a variety of file systems and volume formats, including those listed in Table 1. Although the primary volume format is HFS Plus, OS X can also boot from a disk formatted with the UFS file system. Future versions of OS X may be bootable with other volume formats as well.

**Table 1** File systems supported by OS X

| File System | Description |
|---|---|
| HFS | Mac OS Standard file system. Standard Macintosh file system for older versions of Mac OS. |
| HFS Plus | Mac OS Extended file system. Standard Macintosh file system for OS X. |
| UFS | Unix File System. A variant of the BSD "Fast File System." |
| WebDAV | Used for directly accessing files on the web. For example, iDisk uses WebDAV for accessing files. |
| UDF | Universal Disk Format. The standard file system for all forms of DVD media (video, ROM, RAM and RW) and some writable CD formats. |
| FAT | The MS-DOS file system, with 16- and 32-bit variants. |
| SMB/CIFS | Used for sharing files with Microsoft Windows SMB file servers. |
| AFP | AppleTalk Filing Protocol. The primary network file system for all versions of Mac OS. |
| NFS | Network File System. A commonly-used BSD file sharing standard. OS X supports NFSv2 and NFSv3 over TCP and UDP. |
| FTP | A file system wrapper for the standard Internet File Transfer Protocol. |

## Accessing File-System Data

Every file system stores **metadata** about the files in the file system. This metadata describes the file but is not part of the file itself. The metadata for a file can include attributes such as Mac OS file type information, BSD-style file access permissions, and creation and modification dates. Because of the differences in how file systems store this data, accessing metadata can be a potentially expensive operation on some file systems.

It's important to realize that if a piece of data is not immediately present in the file system, that information might have to be calculated. Retrieving file-system information is a time-consuming operation as it is, but if the information must be calculated or read separately from disk, it becomes even more time-consuming. The valence of a directory—the number of items in that directory—is a typical example of information that must be calculated on most file systems.

When calling file-system routines, you should always carefully consider what information you actually need and request only that information. For example, a single call to `PBGetCatInfoSync` returns Finder file type information from a file or folder. On HFS and HFS Plus file systems, the penalty for retrieving this metadata is minimal because it is stored in the file's catalog node and read into memory along with the file name. However, on other file systems, this data may have to be read separately, incurring another read operation. Instead of `PBGetCatInfoSync`, you should have used `FSGetCatalogInfo` or `PBGetCatalogInfoSync` and specified exactly which pieces of information you wanted.

# Examining File-System Usage

OS X comes with several tools for examining how your application uses the file system. You can sample your application to see what file-system calls it makes. You can watch the file-system calls at the system level and you can examine overall file-system statistics.

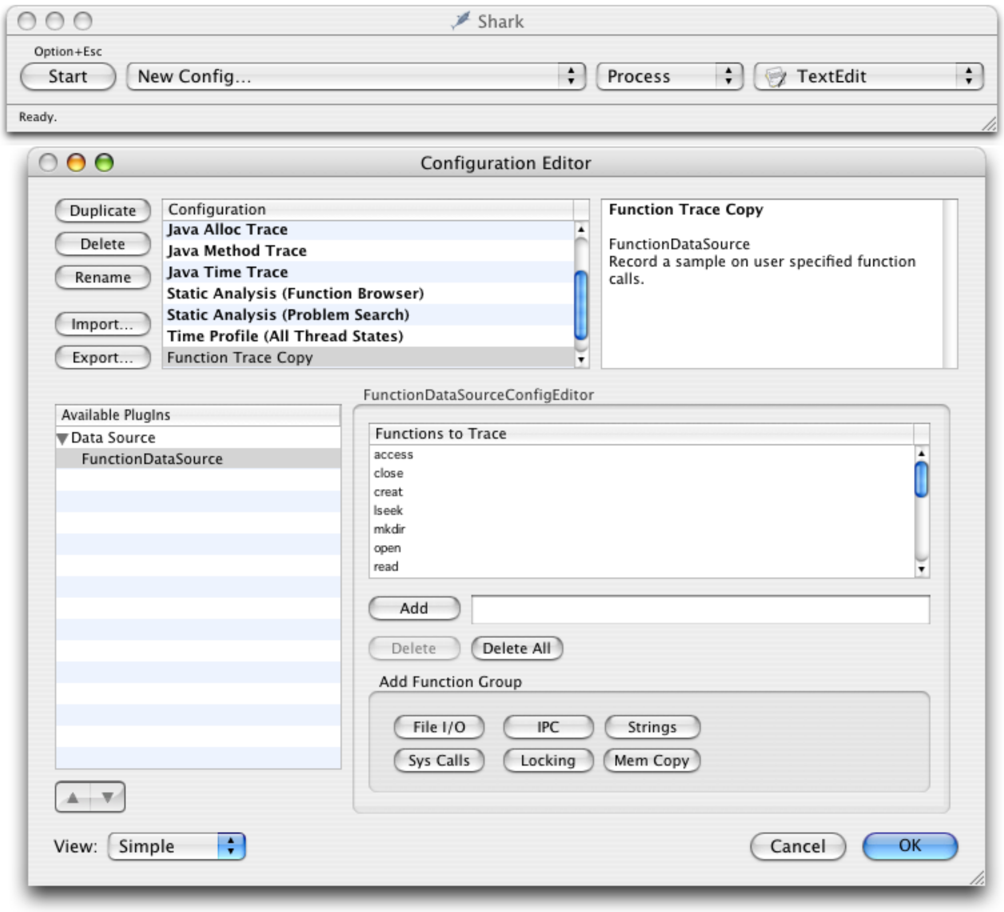
As you gather information, consider using multiple tools to gather your data from different angles. Both Shark and `fs_usage` can provide a lot of information about your application's file-system interactions, but that information is typically very complimentary. Seeing the same behavior in different ways can provide you with better data for isolating the real problems.

## Sampling File-System Usage Selectively

The Shark application lets you selectively find out what file system-related function calls your application makes. When you set up your session configuration, you can tell Shark exactly which function calls you want it to watch. You can specify high-level or low-level calls in your application. Shark even defines a set of default POSIX file I/O functions to watch. This list of functions includes `access`, `close`, `creat`, `lseek`, `mkdir`, `open`, `read`, `readv`, `rename`, `rmdir`, `truncate`, `unlink`, `write`, `writev`, `getattrlist`, `setattrlist`, `getdirentries`, and `getdirentriesattr`. A list of system calls includes all the file I/O calls plus `fcntl`, `flock`, `fstat`, `fsync`, `link`, `lstat`, `lstatv`, and `stat`.

Figure 1 shows the Shark Configuration Editor window, which you display by selecting New Config from the sampling configuration popup menu. Before you begin your session, you must create a configuration that includes the functions you want to trace. Once you select the functions, click OK to dismiss the Configuration Editor window. Shark adds your new configuration to the sampling configuration popup menu. Select it and begin sampling.

**Figure 1** Sampling file I/O with Shark



Ready.

| Configuration |
|---|
| Java Alloc Trace |
| Java Method Trace |
| Java Time Trace |
| Static Analysis (Function Browser) |
| Static Analysis (Problem Search) |
| Time Profile (All Thread States) |
| Function Trace Copy |

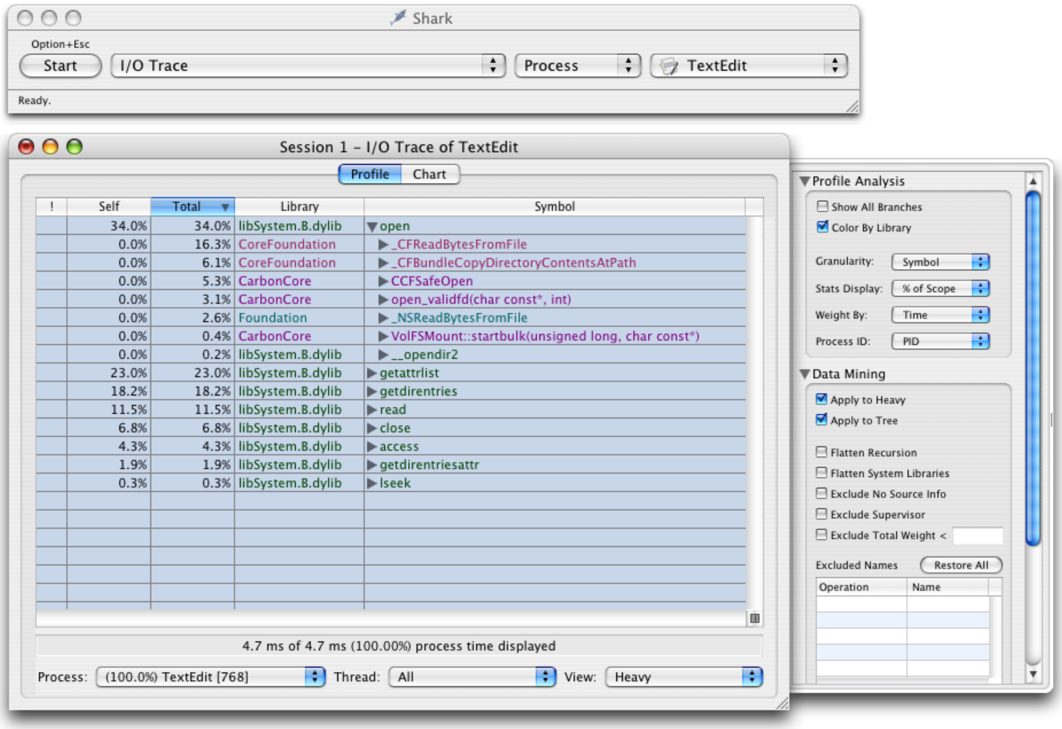
**Function Trace Copy**

FunctionDataSource
Record a sample on user specified function calls.

| Functions to Trace |
|---|
| access |
| close |
| creat |
| lseek |
| mkdir |
| open |
| read |

Disk accesses tend to be slow operations, so minimizing unneeded reads, caching data, or minimizing the number of files examined can improve your application's performance. When you finish sampling, Shark displays the sample window in heavy view mode to highlight the functions you were tracking (Figure 2). You can expand each function to see the points from which it was called.

**Figure 2** File I/O samples for TextEdit



| ! | Self | Total | Library | Symbol |
|---|---|---|---|---|
| | 34.0% | 34.0% | libSystem.B.dylib | ▼open |
| | 0.0% | 16.3% | CoreFoundation | ▶ _CFReadBytesFromFile |
| | 0.0% | 6.1% | CoreFoundation | ▶ _CFBundleCopyDirectoryContentsAtPath |
| | 0.0% | 5.3% | CarbonCore | ▶ CCFSafeOpen |
| | 0.0% | 3.1% | CarbonCore | ▶ open_validfd(char const*, int) |
| | 0.0% | 2.6% | Foundation | ▶ _NSReadBytesFromFile |
| | 0.0% | 0.4% | CarbonCore | ▶ VolFSMount::startbulk(unsigned long, char const*) |
| | 0.0% | 0.2% | libSystem.B.dylib | ▶ __opendir2 |
| | 23.0% | 23.0% | libSystem.B.dylib | ▶ getattrlist |
| | 18.2% | 18.2% | libSystem.B.dylib | ▶ getdirentries |
| | 11.5% | 11.5% | libSystem.B.dylib | ▶ read |
| | 6.8% | 6.8% | libSystem.B.dylib | ▶ close |
| | 4.3% | 4.3% | libSystem.B.dylib | ▶ access |
| | 1.9% | 1.9% | libSystem.B.dylib | ▶ getdirentriesattr |
| | 0.3% | 0.3% | libSystem.B.dylib | ▶ lseek |

4.7 ms of 4.7 ms (100.00%) process time displayed

Process: (100.0%) TextEdit [768] Thread: All View: Heavy

▼Profile Analysis
- ☐ Show All Branches
- ☑ Color By Library

Granularity: Symbol
Stats Display: % of Scope
Weight By: Time
Process ID: PID

▼Data Mining
- ☑ Apply to Heavy
- ☑ Apply to Tree
- ☐ Flatten Recursion
- ☐ Flatten System Libraries
- ☐ Exclude No Source Info
- ☐ Exclude Supervisor
- ☐ Exclude Total Weight <

Excluded Names Restore All

| Operation | Name |
|---|---|

Shark does not identify the parameters passed to any functions, nor does it display the time taken by each operation to execute. If you need this sort of information, you should use `fs_usage` in addition to or instead of Shark. See "Analyzing File Interactions in Detail" (page 15) for more details.

## Analyzing File Interactions in Detail

The `fs_usage` tool presents an ongoing display of system-call usage statistics related to file-system activity, including page-ins and errors. By default this includes all current processes running on the system, including `fs_usage` itself. However, you can limit the statistics gathering to include or exclude a specified list of processes.

The `fs_usage` tool is well suited for the following operations:

- detecting redundant file operations
- discovering what files your application touches during launch

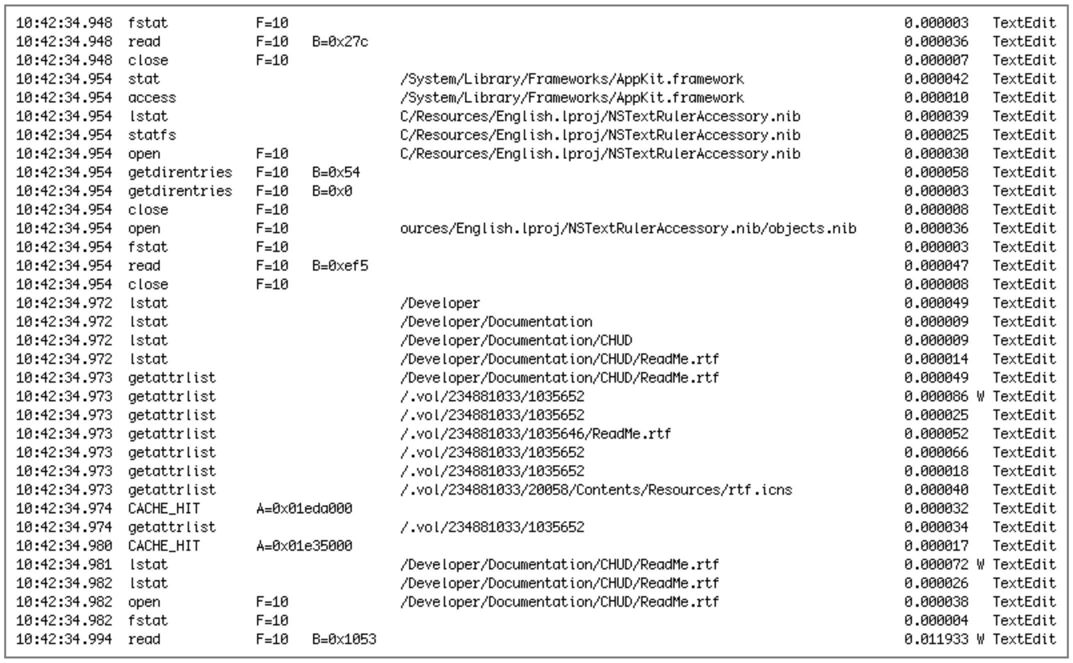
- discovering which files are taking a long time to read
- discovering where bad file-related calls are being made

You can also use `fs_usage` to identify the file-access patterns used by your application. Examining these patterns might point out places where you could optimize your code's behavior. For example, a slow-launching application might be trying to read from preferences stored on a network file server. Rather than read the preferences from the server each time, you might decide to cache those preferences locally and write them back to the server as needed.

**Important:** File-system activity information is subject to access controls. The kernel does not allow you to access information through `fs_usage` unless you are logged in as the root user (or logged in at a Terminal window using the `su` command—on some systems, `sudo` may be required instead).

The `fs_usage` tool formats its output according to the size of your window. A narrow window displays fewer columns of data. Use a wide window for maximum data display. The `–w` parameter forces all columns to be displayed regardless of the size of the window. Figure 3 shows the output of `fs_usage` using the `–w` parameter.

**Figure 3** Example of "fs_usage -w" output



```
10:42:34.948  fstat           F=10                                                                0.000003   TextEdit
10:42:34.948  read            F=10   B=0x27c                                                      0.000036   TextEdit
10:42:34.948  close           F=10                                                                0.000007   TextEdit
10:42:34.954  stat                              /System/Library/Frameworks/AppKit.framework                 0.000042   TextEdit
10:42:34.954  access                            /System/Library/Frameworks/AppKit.framework                 0.000010   TextEdit
10:42:34.954  lstat                             C/Resources/English.lproj/NSTextRulerAccessory.nib          0.000039   TextEdit
10:42:34.954  statfs                            C/Resources/English.lproj/NSTextRulerAccessory.nib          0.000025   TextEdit
10:42:34.954  open            F=10              C/Resources/English.lproj/NSTextRulerAccessory.nib          0.000030   TextEdit
10:42:34.954  getdirentries   F=10   B=0x54                                                       0.000058   TextEdit
10:42:34.954  getdirentries   F=10   B=0x0                                                        0.000003   TextEdit
10:42:34.954  close           F=10                                                                0.000008   TextEdit
10:42:34.954  open            F=10              ources/English.lproj/NSTextRulerAccessory.nib/objects.nib   0.000036   TextEdit
10:42:34.954  fstat           F=10                                                                0.000003   TextEdit
10:42:34.954  read            F=10   B=0xef5                                                      0.000047   TextEdit
10:42:34.954  close           F=10                                                                0.000008   TextEdit
10:42:34.972  lstat                             /Developer                                                  0.000049   TextEdit
10:42:34.972  lstat                             /Developer/Documentation                                    0.000009   TextEdit
10:42:34.972  lstat                             /Developer/Documentation/CHUD                               0.000009   TextEdit
10:42:34.972  lstat                             /Developer/Documentation/CHUD/ReadMe.rtf                    0.000014   TextEdit
10:42:34.973  getattrlist                       /Developer/Documentation/CHUD/ReadMe.rtf                    0.000049   TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/1035652                                     0.000086 W TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/1035652                                     0.000025   TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/1035646/ReadMe.rtf                          0.000052   TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/1035652                                     0.000066   TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/1035652                                     0.000018   TextEdit
10:42:34.973  getattrlist                       /.vol/234881033/20058/Contents/Resources/rtf.icns           0.000040   TextEdit
10:42:34.974  CACHE_HIT       A=0x01eda000                                                        0.000032   TextEdit
10:42:34.974  getattrlist                       /.vol/234881033/1035652                                     0.000034   TextEdit
10:42:34.980  CACHE_HIT       A=0x01e35000                                                        0.000017   TextEdit
10:42:34.981  lstat                             /Developer/Documentation/CHUD/ReadMe.rtf                    0.000072 W TextEdit
10:42:34.982  lstat                             /Developer/Documentation/CHUD/ReadMe.rtf                    0.000026   TextEdit
10:42:34.982  open            F=10              /Developer/Documentation/CHUD/ReadMe.rtf                    0.000038   TextEdit
10:42:34.982  fstat           F=10                                                                0.000004   TextEdit
10:42:34.994  read            F=10   B=0x1053                                                     0.011933 W TextEdit
```

## Interpreting the Output of fs_usage

The `fs_usage` tool continuously generates a large amount of data with millisecond granularity. The output is not updated in place (as with, say, `top`); instead, each new line of data is appended to the existing data. When running `fs_usage` for very brief periods of time or during a very specific activity, viewing the information in the Terminal window is possible but time consuming. In most cases, you will probably want to redirect the output of `fs_usage` to a file so that you can go back and examine it later or run it through a script.

The columns of `fs_usage` output have no headings and are separated by spaces. You can interpret the type of data in each column by its format. Table 1 (page 17) describes these columns. If you run the tool without the `-w` option, some of these columns may be missing.

**Table 1** Columns of "fs_usage" output

| Column Number | Example | Description |
|---|---|---|
| 1 | `14:56:52.386` | Timestamp, giving the time of day when the call occurred. In wide mode, this field has millisecond granularity. |
| 2 | `fstat`, `CACHE_HIT`, or `PAGE_IN` | The operation that was detected. Usually, this is the name of a file-system routine or a specific system event, such as a page-in. |
| 3 | `A=0x45e2a000` | Fault address. If the prior column is `CACHE_HIT`, `PAGE_IN` or another system event, this specifies the address being faulted. |
| 3 | `F=58` | File descriptor associated with the call described in the second column (for example, `fstat` or `open`); in this example, `58` is the file descriptor. |
| 4 | `O=0x5000` or `B=0x78` or `[45]` | This column can contain one of three values. It can contain the file offset specified to `lseek` or `ftruncate` (shown as `O=0x5000`). It can contain the number of bytes requested by the call (shown as `B=0x78`). Finally, if the call results in an error, it contains an `errno` value between brackets (see the header file `errno.h` for a list of error codes). |
| 5 | `/Network` | The pathname of the file accessed. This value may be truncated but will always display the end of the pathname. Carbon developers should read Technical Q&A QA1113: The "/.vol" Directory and "volfs" for additional information on how to interpret Carbon File Manager calls. |

| Column Number | Example | Description |
|---|---|---|
| 6 | `0.000459W` | Elapsed time (in microseconds) spent in the system call. A `W` after the time indicates that the process was scheduled out during this file activity (probably because it was waiting for a disk or network I/O operation to complete). In this case, the elapsed time includes the wait time. |
| 7 | `TextEdit` | The name of the executable or application package that made the system call. (Note that Code Fragment Manager applications are named after the native process that launches them, `LaunchCFMApp`.) |

### Viewing Carbon File Manager Calls

Carbon and Cocoa applications can obtain additional information from `fs_usage` using the `DYLD_IMAGE_SUFFIX` environment variable. Setting this variable to the value "`_debug`" causes the dynamic linker to use the debug version of the Carbon libraries. Running `fs_usage` against these libraries causes the tool to display the name of the Carbon File Manager routine that was called in addition to the underlying system routine.

**Note:** Because the NSFileManager class in Cocoa uses the Carbon File Manager for its underlying file manipulations, this technique works for Cocoa applications as well.

## Gathering System Call Statistics with sc_usage

The `sc_usage` tool displays an ongoing sample of system statistics for a given process, including the number of system calls and page faults. The tool adds new system calls to the list as they are generated by the application being watched. The counts displayed are both the cumulative totals since `sc_usage` was launched and the delta changes for this sample period. The `sc_usage` tool also displays the following information:

- the amount of CPU time consumed by the process and by each routine
- the absolute time during which the process is waiting
- the cumulative time a thread has been blocked (identified by number)
- the current scheduling priority for the thread
- the number of page-ins, copy-on-write operations, zero-fill faults, and faults that hit in the page cache
- global state, including the number of preemptions, context switches, threads, faults, and system calls found during the sampling period

Listing 1 (page 19) shows some sample `sc_usage` output for the TextEdit application.

**Listing 1** Sample output from sc_usage

```
TextEdit          0 preemptions    0 context switches    1 thread     13:23:55
                  0 faults         0 system calls                      0:00:30

TYPE                            NUMBER        CPU_TIME   WAIT_TIME
------------------------------------------------------------------------------
System         Idle                                      0:05.643( 0:00.965)
System         Busy                                      0:00.285( 0:00.038)
TextEdit       Usermode                       0:00.029

zero_fill                         17          0:00.000   0:00.000

mach_msg_trap                    213          0:00.003   0:02.944( 0:01.003) W
gettimeofday                       4          0:00.000
mk_timer_create                    9          0:00.000
mk_timer_destroy                   9          0:00.000
mk_timer_arm                      19          0:00.000
mk_timer_cancel                    3          0:00.000
mach_port_insert_member           13          0:00.000
mach_port_extract_membe           13          0:00.000
vm_deallocate                     17          0:00.000   0:00.000
```

Be aware that the `mach_msg_trap` kernel routine will always be the system call with the greatest amount of CPU time used. This call indicates that the application is blocked and waiting for something to happen, such as a system event.

# 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.

Another thing to remember about mapped files is that they share the process space with system libraries, your application code, and allocated memory. Most applications have around 2 gigabytes of addressable memory, depending on the number of libraries they load. In order to map a file, there must be an available address range big enough to fit the file. Finding this much space can be difficult if your application's virtual memory space is fragmented or you attempt to map a very large file.

Before you map any files into memory, make sure you understand your typical file usage patterns. Tools such as Shark and `fs_usage` 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 fills virtual memory space quickly. In addition, your program may not have the available space if it has been running for a while or its memory space is fragmented.)

For large sequential read operations, you are better off disabling disk caching and reading the file into a small memory buffer. See "Cache Files Selectively" (page 8) for more information.

## 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. Files whose size is several hundred megabytes or more fall into this category.
- 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 can occupy a significant portion of your application's virtual address space. The address space for a single process is currently limited to 4 gigabytes, with some portions of that space reserved for various system frameworks and libraries. If you try to map a very large file, you might find there isn't enough room to map the entire file anyway. This problem can also occur if you map too many files into your process space.

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.

## Mapping Resource Files

Mapping your data fork-based resource files into memory is often a good idea. Resource files typically contain frequently-used data that your application needs to operate. Because of its usefulness, OS X includes a mechanism to map resources automatically. To enable this mechanism, add the following lines to your `Info.plist` file:

```
<key>CSResourcesFileMapped</key>
<true/>
```

The CFBundle resource file functions (`CFBundleOpenBundleResourceMap` and `CFBundleOpenBundleResourceFiles`) check for the `CSResourcesFileMapped` key before opening a resource file. If this key is present and set to true, the functions map the resource file into memory. The resource data is mapped read-only, so you cannot write to the file or any of its resources directly. For example, the following will cause an memory access exception if the `PICT` resource comes from a mapped resource file:

```
PicHandle picture = (PicHandle)GetResource('PICT', 128);
(**picture).rect = myRect; // crash here attempting to write
                           // to read-only memory
```

## File Mapping Example

Listing 1 (page 22) demonstrates the BSD routines `mmap` and `munmap` to map and unmap files. The mapped file occupies a system-determined portion of the application's virtual address space until `munmap` is used to unmap the file.

**Listing 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;
}
```

# Iterating Directory Contents

Iterating the file system should be avoided whenever possible. Iterating the file system reads in metadata for a large number of files and fills up the system disk caches with data that will likely be used only once. If you must iterate directories repeatedly, consider caching the results from previous iterations to avoid repeated calls to the `stat` function.

When iterating over a large number of files, your best choice is to use the more efficient low-level routines provided by both Carbon and the BSD system. The following sections describe the basic techniques for using these routines.

## Iterating Directories in Carbon

The example in Listing 1 (page 25) demonstrates how to use an HFS Plus bulk iterator to efficiently scan the contents of a directory. It does not descend into subdirectories, but you can open as many bulk iterators as necessary to handle recursive iteration. For information on scanning a directory repeatedly to watch for changes, "Tracking File-System Changes" (page 33).

**Listing 1** Fast directory iteration

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

// Forward declarations.
OSStatus IterateFolder( FSRef * inFolder );
void DoSomethingWithThisObject( const FSCatalogInfo * inCatInfo );

int main(void)
{
    OSStatus    outStatus;
    FSRef       folderRef;

    printf("begin file iteration!\n");
    fflush( stdout );
    //
    // Get the current user’s documents folder,
    // make it into an FSRef, and iterate it
    //
   outStatus = FSFindFolder(kUserDomain, kDocumentsFolderType, false, &folderRef);
    if( outStatus == noErr )
    {
        outStatus = IterateFolder( &folderRef );
    }

    printf( "final error status is (#%d)\n", (int)outStatus );
    return 0;
}

OSStatus IterateFolder( FSRef * inFolder )
{
    OSStatus outStatus;

    // Get permissions and node flags and Finder info
    //
    // For maximum performance, specify in the catalog
    // bitmap only the information you need to know
    FSCatalogInfoBitmap kCatalogInfoBitmap = (kFSCatInfoNodeFlags |
                                              kFSCatInfoFinderInfo);

    // On each iteration of the do-while loop, retrieve this
    // number of catalog infos
    //
    // We use the number of FSCatalogInfos that will fit in
    // exactly four VM pages (#113). This is a good balance
    // between the iteration I/O overhead and the risk of
    // incurring additional I/O from additional memory
    // allocation
```

```
const size_t kRequestCountPerIteration = ((4096 * 4) / sizeof(FSCatalogInfo));
FSIterator iterator;
FSCatalogInfo * catalogInfoArray;

// Create an iterator
outStatus = FSOpenIterator( inFolder, kFSIterateFlat, &iterator );

if( outStatus == noErr )
{
    // Allocate storage for the returned information
    catalogInfoArray = (FSCatalogInfo *) malloc(sizeof(FSCatalogInfo) *
                    kRequestCountPerIteration);

    if( catalogInfoArray == NULL )
    {
        outStatus = memFullErr;
    }
    else
    {
        // Request information about files in the given directory,
        // until we get a status code back from the File Manager
        do
        {
            ItemCount actualCount;

          outStatus = FSGetCatalogInfoBulk(iterator, kRequestCountPerIteration,
                            &actualCount, NULL, kCatalogInfoBitmap,
                            catalogInfoArray, NULL, NULL, NULL );

            // Process all items received
            if( outStatus == noErr || outStatus == errFSNoMoreItems )
            {
                UInt32  index;

                for( index = 0; index < actualCount; index += 1 )
```

```
                    {
                        // Do something interesting with the object found
                        DoSomethingWithThisObject( &catalogInfoArray[ index ] );
                    }
                }



            }
            while( outStatus == noErr );

            // errFSNoMoreItems tells us we have successfully processed all
            // items in the directory -- not really an error
            if( outStatus == errFSNoMoreItems )
            {
                outStatus = noErr;
            }

            // Free the array memory
            free( (void *) catalogInfoArray );
        }
    }

    FSCloseIterator(iterator);

    return outStatus;
}

void DoSomethingWithThisObject( const FSCatalogInfo * inCatInfo )
{
    if( (inCatInfo->nodeFlags & kFSNodeIsDirectoryMask) == kFSNodeIsDirectoryMask
 )
    {
        printf( "Found a folder\n" );
    }
    else
    {
        FInfo * theFinderInfo;
        OSType type;

        theFinderInfo = (FInfo *)&inCatInfo->finderInfo[0];
        type = theFinderInfo->fdType;

        printf( "Found a file (type %c%c%c%c)\n",
                (char) ((type & 0xFF000000) >> 24),
                (char) ((type & 0x00FF0000) >> 16),
                (char) ((type & 0x0000FF00) >> 8),
                (char) (type & 0x000000FF)
                 );
    }
}
```

## Traversing Directories in BSD

A reasonably fast way to traverse a directory hierarchy is to use the BSD-level `fts` routines. Like the Carbon catalog iterators, these routines let you walk a file hierarchy and examine each file and directory. Unlike the Carbon catalog iterators, you cannot use the `fts` routines to retrieve a file's Finder type or a directory's valence. To gather information of that nature, you must call additional routines, which adds an additional expense.

**Note:** Gathering valence information for directories on non-HFS Plus volumes is very expensive and should be avoided whenever possible. HFS Plus volumes cache valence information in the file catalog, but other file systems must physically count the number of files and directories in the target directory.

For detailed information on how to use the BSD `fts` routines, see the `fts(3)` man page entry.

## Searching Directory Catalogs

If you need to search for information on a hard disk, consider using the Carbon `FSCatalogSearch` function. This function is optimized for reading the disk catalog information and is significantly faster than iterating the directories yourself and searching for the information.

## Improving Iteration Memory Usage

Whenever you iterate the contents of a directory, you should be careful to check your virtual memory usage in `top`. If you notice your memory usage increasing during the iteration cycle, you may want to use Shark, MallocDebug, or ObjectAlloc to investigate your allocation patterns further. A significant increase in the number of resident memory pages during your iteration could cause paging to occur in low-memory situations.

If you find your memory allocation or usage increasing during an iteration, you should examine ways to reduce your overall memory consumption. Rather than allocating new memory for storing data, try to recycle existing memory blocks or eliminate specific allocations altogether. The example in Listing 1 (page 25) shows an efficient way to maintain a low memory footprint for Carbon-based iterators. For Cocoa applications using an NSFileManager object to walk the items in a directory, consider adding an NSAutoreleasePool inside your iteration loop to reclaim any released objects.

# Resolving Domain Names

With the advent of IPv6, you may be tempted to use the `getaddrinfo` function to perform name or IP address resolution. However, use of this function can lead to extreme performance problems in some situations, such as large NetInfo networks. The `getaddrinfo` function is very general in nature and can cause multiple network operations to occur.

## Using CFNetwork Routines

The CFNetwork family of routines provides a convenient way to do simple name-to-IP lookups. These routines support the fast lookup of host names and do so in a non-blocking fashion. More importantly, they retrieve precisely the information you need.

Beginning with OS X version 10.3, you can also use the CFHost family of routines to perform lookups. CFHost supports hostname lookup in both IPv4 and IPv6 automatically, so you do not need to rewrite your code to support both. As with the CFNetwork routines, CFHost routines are fast and non-blocking.

For more information on CFNetwork and CFHost, see the Core Foundation reference.

## Using BSD routines

If you are performing a simple name-to-IP lookup, you can also use the BSD functions `getipnodebyname` and `getipnodebyaddr` instead of `getaddrinfo`. In OS X version 10.2 and later, these functions are threadsafe, reentrant, and take advantage of the networking subsystem's IP address caching capabilities to improve performance. You can also use `getipnodebyname` to resolve domain names instead of `getipnodebyname`. For example, if you have code such as the following:

```
struct hostent * hostentry;


hostentry = gethostbyname(name);
    /* ...do something with the hostentry structure*/
```

You can replace it with the following IPv6-savvy code:

```
struct hostent * hostentry;


hostentry = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
    /* ...do something with the hostentry structure*/
freehostent(hostentry);
```

The new code will handle both IPv4 and IPv6 addresses. Note that you must call the `freehostent` function to release the `hostent` data structure when you are finished with it.

# Tracking File-System Changes

Many applications need to watch the file system for changes:

- Applications with Finder-style file list views need to update the file lists based on changes made by the user in the Finder or in another application.
- Document-based applications should watch for filename modifications and change the document's window title accordingly. They might also want to close a document window when the associated file is moved to the Trash.
- A small number of applications need to watch a particular directory and process files dropped onto that directory.

The problem with any of these tasks is that the most common solution to the problem is to poll the operating system. Unfortunately, as is explained in *Performance Overview* , applications should never poll the system for information. In particular, polling the file system uses an excessive amount of I/O bandwidth and degrades system performance. It also tends to fill low-level file-system caches with less-useful information.

## Carbon Notifications

Instead of polling the system, your application should wait for system events and then synchronize information as appropriate. For example, when the application or document window becomes active, you can update the window title appropriately. For Cocoa applications, the NSDocument class implements this behavior for you. For Carbon applications, you can implement this behavior in a Carbon Event Manager `kWindowActivateEvent` event handler

For more global changes, the Carbon File Manager provides the `FNNotify` and `FNSubscribe` functions that allow applications to receive notifications whenever another application explicitly publishes changes to a directory. However, this service is strictly voluntary for both applications and does not provide notifications over the network. You can use it to supplement the file synchronization strategy mentioned above, but you cannot rely on it alone. For information on how to use the `FNNotify` and `FNSubscribe` family of functions see the *File Manager Reference* .

## Cocoa Notifications

For Cocoa developers, the NSWorkspace class provides behavior similar to that provided by FNNotify and FNSubscribe. Your application can register to receive workspace notifications and use them to update files when changes occur. For information on how to send and receive notifications in cocoa, see "Receiving Workspace Notifications" in *Workspace Services Programming Topics* . See also the `NSWorkspace` class documentation.

## Kernel Notifications

The kqueue mechanism in BSD provides another way to be notified of system changes. Using this mechanism you can request notifications when specific events occur or when a specific condition becomes true. You can use this to monitor files and other system entities such as ports and processes.

When you only want to track changes on a file or directory, be sure to open it using the `O_EVTONLY` flag. This flag prevents the file or directory from being marked as open or in use. This is important if you are tracking files on a removable volume and the user tries to unmount the volume. With this flag in place, the system knows it can dismiss the volume. If you had opened the files or directories without this flag, the volume would be marked as busy and would not be unmounted.

For more information about kqueues and kevents, see the `kqueue` man page.

# Document Revision History

This table describes the changes to *File-System Performance Guidelines* .

| Date | Notes |
|---|---|
| 2005-07-07 | Updated cache flag information. |
| 2005-04-29 | Replaced Sampler examples with Shark examples. Updated zero-fill guidelines. Updated caching guidelines. Changed "Rendezvous" to "Bonjour." Changed title from *File-System Performance* . |
| 2004-08-31 | Updated the list of tips for improving file system performance. |
| 2004-01-29 | Fixed a bug in file system iteration sample code. |
| 2003-07-25 | Updated tips and fixed some minor bugs for OS X v10.3. |
| 2003-05-15 | First revision of this programming topic. Some of the information appeared in the document *Inside OS X: Performance* . |