INSIDE MAC OS X

Performance

March 2001

Contents

CHAPTER 1

About This Book

This book is a detailed guide to the fundamentals of low-level performance on Mac OS X. It offers techniques, guidelines, as well as tool documentation, to allow you to enable your code for maximum performance on Mac OS X.

Why Read This Book

Inside Mac OS X: Performance is intended for all developers who want to make code run faster on Mac OS X. *Performance* covers two essential topics: enhancing your program to achieve maximum performance under the Mac OS X system architecture, and using the Mac OS X development tools to analyze your code. Whether you have a performance problem or you just want to make your code as efficient as possible, this book is your guide.

Prior to reading this book, you should read *Inside Mac OS X: System Overview*. See "Other Apple Publications" (page 11). You should also be familiar with at least one high-level language. Knowledge of BSD or another UNIX operating system is helpful, particularly where the development tools are concerned, but it is not assumed.

One performance topic not covered in this book is the selection of a suitable algorithm for a particular task. See "Further Investigations" (page 10) for more information.

Making Your Code Faster on Mac OS X

Here is a quick road map describing each chapter:

- "Managing Memory" (page 13) describes memory allocation on Mac OS X and the benefits that can be gained through judicious management of virtual memory and by taking full advantage of the Mac OS X memory allocation and copying facilities.
- "Accessing the File System" (page 27) describes characteristics of some of the many different file systems supported by Mac OS X and generic optimizations you can make for all of them.
- "Optimizing Carbon Programs for Mac OS X" (page 35) describes opportunities for improving code ported from previous versions of Mac OS. Carbon on Mac OS X introduces changes to event handling, the file system, and the runtime execution model. This chapter tells you how to use these changes to your best advantage.
- "Building Efficient C, C++, and Java Programs" (page 53) describes language-specific optimizations and guidelines for C, C++ and Java development.
- "Analyzing Performance" (page 61) describes the Mac OS X performance tools. These tools can help you gain a better understanding of the way your code executes and manipulates data. They can also be used to diagnose known performance problems.
- "Organizing Your Executable File" (page 117) describes the methods used by the development tools to organize the contents of executable files, and how you can change the layout of your code and data in memory to your benefit.

Further Investigations

There are many relevant performance resources available to Mac OS X developers. This section lists just a few.

Important Topics Covered Elsewhere

One of the most critical performance-related choices an application developer must make is the selection or design of an algorithm suited to the task at hand. There are several exhaustive reference books on this subject. The standard reference work is the multivolume set *The Art of Computer Programming*, by Donald E. Knuth (Addison-Wesley, 1998, ISBN 0-201-14854-19). A more concise guide is Robert Sedgewick's *Algorithms* (Addison-Wesley, 1988, ISBN 0-201-06672-6).

The standard textbook for overall system architecture design (including performance issues) is *Computer Architecture: A Quantitative Approach*, by John L. Hennessey and David A. Patterson (Morgan Kaufmann, 1990, ISBN 1-55860-329-8).

Proper use of the PowerPC G4 AltiVec instruction set can massively decrease execution time required by computationally expensive code. For more information, see these websites:

- http://developer.apple.com/hardware/
- http://www.altivec.org
- http://www.motorola.com/SPS/PowerPC/AltiVec/facts.html

Other Apple Publications

Performance is one of the books in the *Inside Mac OS X* series. You can obtain other books in this series using Apple's print-on-demand arrangement with fatbrain.com, or on the web as PDF files.

- PDF documents are available at Apple's developer documentation website http//developer.apple.com/techpubs/macosx/
- Printed copies may be ordered from fatbrain.com at http://www.fatbrain.com/documentation/apple/

Other Information

The Design and Evolution of C++, by Bjarne Stroustrup (Addison-Wesley,1994,ISBN 0-201-54330-3) is a design rationale for the C++ language. It contains valuable nuggets on the intended, performance-conscious use of newer C++ language constructs such as exceptions and templates.

The Graphics Programming Black Book, by Michael Abrash (Coriolis Group, 1997, ISBN 1-57610-174-6), is a down-to-earth guide to the methodology of code optimization, disguised as a graphics programming book.

C H A P T E R 2

Managing Memory

This chapter contains an overview of virtual memory as implemented by the Mac OS X kernel. Beyond that, it offers techniques for allocating and copying memory efficiently.

On a well-tuned operating system equipped with virtual memory, the main system performance bottleneck is the I/O bandwidth used to read a section of virtual memory into physical memory (or write a section of physical memory out to disk). Therefore, reducing memory I/O activity is key to getting the maximum possible performance out of Mac OS X.

Virtual Memory Theory

Virtual memory allows an operating system to escape the limitations of physical RAM. A virtual memory manager creates a logical address space (or "virtual" address space) that is larger than the installed physical memory (RAM) and divides it up into uniformly-sized chunks of memory called **pages**. Each page in the logical address space has a corresponding page on the disk, in a special file known as the **backing store**. The system then populates the computer's physical memory with the pages currently in use to give the illusion that the entire logical address space is made up of real memory.

There are two key features of the processor and its memory management unit (MMU) that you must grasp in order to understand how virtual memory works. The first is the **page table**, a table that maps all logical pages into their corresponding physical pages. When the processor accesses a logical address, the MMU uses the page table to translate the access into a physical address, which is the address that's actually passed to the computer's memory subsystem.

Virtual memory also utilizes the processor's ability to invoke a handler when a page translation fails. When the processor accesses a logical address that is not in physical memory, the processor stops executing normal code and starts executing the special code to handle this **page fault**. The page fault handler is responsible for finding a free page of physical memory, reading the contents of the page from the backing store into the physical page, and then changing the page table so the page appears to be at the correct logical address. If no free page is available, a page currently in memory is released. If that page contains modified data, it is first written to the backing store. This process is known as **paging**.

Moving data from physical memory to disk is called **paging out** (or "swapping out"); moving data from disk to physical memory is called **paging in** (or "swapping in"). In both Mac OS 9 and Mac OS X, the size of a page is 4 kilobytes. Every time a page fault occurs, 4 kilobytes is read from disk. Extended periods of paging activity tend to reduce performance; such activity is sometimes called **disk thrashing**.

Reading from disk is much slower than reading directly from RAM, just as reading from RAM is always slower than reading directly from CPU cache. As noted previously, a page fault causes a read from disk. Thus, the primary goal of every performance-conscious developer on a system with virtual memory must be the minimization of page faults.

One way to think of virtual memory is that it uses the real memory in the computer as a cache for the entire logical address space. Real memory can be accessed quickly, while memory that's paged out takes a long time to access. Thus, a page fault is analogous to a cache miss. Like all caches, virtual memory relies on locality of reference for speed.

Virtual Memory on Mac OS 9

On Mac OS 9, all processes share a global, fixed-size 32-bit address space. When virtual memory is enabled, a backing store file is created on a user-specified volume. The size of this file, and thus the amount of available virtual memory, is fixed at a user-specified setting. The file serves as a fixed-size container for all physical RAM plus all additional virtual RAM. On a system with 64 MB of physical RAM, a 65 MB virtual memory setting offers only one megabyte of virtual RAM.

Additionally, because of limitations of the Mac OS 9 multitasking architecture, real-time processes (such as audio/video playback) may be starved for execution time during even short periods of paging activity. Because of these and other software compatibility issues, users of Mac OS 9 are allowed to disable virtual memory. This is not necessary in Mac OS X.

Virtual Memory on Mac OS X

On Mac OS X, each process has its own **sparse** 32-bit virtual address space, dynamically growable up to a limit of four gigabytes. Additional swap file space is allocated on demand from dynamically created files stored in the root file system.

When your program reads, writes, or executes at a particular address, the corresponding page is **mapped** by the kernel. This mapped area is referred to as a **region**. The virtual address space of a process consists of mapped regions of memory. Each region of memory in Mac OS X represents a specific set of virtual memory pages. A region has specific attributes controlling such things as inheritance (portions of the region may be mapped from “parent” regions), write-protection, and whether it is “wired” (that is, it cannot be paged out). Regions, being containers of pages, are **page-aligned**, meaning the starting address of the region is also the starting address of a page and the ending address also defines the end of a page.

The kernel associates a **VM object** with each region of the virtual address space. The kernel uses the VM object for tracking and managing the resident and nonresident pages of the memory region. Each VM object maps either a portion of virtual memory in the backing store through the default pager or a portion of a file-mapped file through the vnode pager.

The **default pager** is a system manager that maps the nonresident virtual memory pages to backing store and fetches those pages when requested.

The **vnode pager** implements file mapping. The vnode pager uses the paging mechanism to provide a window on a specified range of a file, allowing you to read and write the contents of that range by reading and writing the mapped range of memory.

VM objects may point to a pager or to another VM object. The kernel uses this self referencing to implement a form of page-level sharing known as **copy-on-write**. Copy-on-write allows multiple blocks of code (including different processes) to share a page as long as none write to the page. If one process writes to the page, a new, writable copy of the page is created in the address space of that process. This allows efficient copying of large quantities of data. See also "Copying Memory Efficiently" (page 22).

Each VM object contains several fields, as shown in Table 2-1.

Table 2-1 Fields of the VM object

Field	Description
Resident pages	A list of the pages of this region that are currently resident in physical memory.
Size	The size of the region, in bytes.
Pager	The pager responsible for tracking and handling the pages of this region in backing store.
Shadow	Used for copy-on-write optimizations.
Copy	Used for copy-on-write optimizations.
Attributes	Flags indicating the state of various implementation details.

If the VM object is involved in a copy-on-write (`vm_copy`) operation, the shadow and copy fields may point to other VM objects. Otherwise both fields are usually `NULL`.

Page Lists in the Kernel

The kernel maintains and queries three system-wide lists of physical pages of memory:

- Active list—Pages currently resident in, and mapped to, physical memory and recently accessed.
- Inactive list—Pages currently resident in physical memory but not recently accessed. These pages contain valid data but may be unmapped at present.
- Free list—Unmapped pages (no longer containing valid data) of physical memory available to the system at no cost; also known as the “free pool.” These pages are not associated with any address space or VM object, and they are thus free for immediate use.

When the number of pages on the free list falls below a threshold (determined by the size of physical memory), the pager attempts to balance the queues. It does this by pulling pages from the inactive list. If the page has been accessed recently, it is placed on the end of the active list (reactivated). If the page has not been recently accessed, but the page has been written to, the contents of this physical page are paged out to the associated backing store. If the page was neither access recently nor written to and is not permanently resident (wired), it is stolen (any current virtual mappings to it are destroyed) and added to the free list. Once the free list size exceeds the target threshold, the pager rests.

The kernel moves pages from the active list to the inactive list if they are not accessed; it moves pages from the inactive list to the active list on a soft fault (see “Paging Virtual Memory In” (page 19)). When virtual pages are swapped out, the associated physical pages are placed in the free list. Also, when processes explicitly free memory, the kernel moves the affected pages to the free list.

Allocating and Accessing Virtual Memory

Applications usually allocate memory using the `malloc` routine. When a program allocates memory via `malloc`, the system routine `vm_allocate` may be invoked. Through this routine the kernel performs a series of initializations:

1. It maps a range of memory in the virtual address space of this process by creating a **map entry**; the map entry is a simple structure that defines the starting and ending addresses of the region.
2. The range of memory is backed by the default pager. On the first access to a page, the default pager fills the page with zeros.
3. The kernel creates and initializes a VM object, associating it with the map entry.

At this point there are no pages resident in physical memory and no pages in backing store. Everything is virtual.

When a program accesses the region, by reading or writing to a specific address in it, a fault occurs because that address has not been mapped to physical memory. The kernel also recognizes that the VM object has no backing store for the page on which this address occurs. The kernel performs the following steps for each page fault:

1. It acquires a page from the free list and fills it with zeros.
2. It inserts a reference to this page in the VM object's list of resident pages.
3. It maps the virtual page to the physical page by filling in a data structure called the **pmap**. The pmap contains the page table used by the processor (or by a separate memory management unit) to map a given virtual address to the actual hardware address.

Paging Virtual Memory Out

The kernel continuously compares the number of physical pages in the free list against a threshold value. When the number of pages in the free list dips below this threshold, the kernel swaps out pages of memory that have not been accessed recently, thereby reclaiming physical pages for the free list. The kernel then iterates all resident pages in the active and inactive lists, performing the following steps:

1. If a page in the active list is not recently touched, it is moved to the inactive list.
2. If a page in the inactive list is not recently touched, the kernel finds the page's VM object.
3. If the VM object has never been paged before, the kernel calls an initialization routine that creates and assigns a default pager object.
4. The VM object's default pager attempts to write the page out to backing store.
5. If the pager succeeds, the kernel frees the physical memory occupied by the page and moves the page from the inactive to the free list.

Paging Virtual Memory In

The final phase of virtual memory management moves pages in backing store back into physical memory (paging in). Memory access faults initiate page-in activity. Memory access faults occur when code tries to access data at a virtual address that is not mapped to physical memory. There are two kinds of faults:

- **Soft fault**: The page of the referenced address is resident in physical memory but is currently not mapped into the address space of this process.
- **Hard fault**: The page of the referenced address is not in physical memory but has been swapped out to backing store (or is available from a mapped file). This is also known as a page fault.

When any fault occurs, the kernel finds the map entry for the accessed region; from the map entry it locates the VM object. The kernel then goes through the VM object's list of resident pages.

- If the page is in the list of resident pages, a soft fault is generated. The kernel maps the region in the virtual address space to physical memory. The page will be marked as having been accessed recently. If the fault was a write (not a read), the page will also be marked as having been written to.
- If the page is not in the list of resident pages, a hard fault is generated. The VM object's pager finds the page in the backing store (if the pager is the default pager) or from a file-mapped file (if the pager is the vnode pager). After making the necessary virtual-to-physical mapping, moves the page into physical memory. It also puts the page in the active page list.

Shared Memory

Shared memory can be written to or read from two or more processes. Shared memory can be inherited from a parent process, created by a shared memory server, or explicitly created by an application for export to other applications. Uses for shared memory include

- sharing large resources such as icons or sounds
- fast communication between one or more processes

Shared memory is fragile. If one program corrupts a section of shared memory, all programs that reference that shared memory are corrupted.

Wired Memory

Wired memory (also called **resident** memory) is always in physical RAM and cannot be paged to disk. Applications, frameworks, and other user-level software cannot allocate wired memory. However, they can affect how much wired memory exists at any time. There is memory overhead associated with each kernel resource expended on behalf of a program. Table 2-2 lists some of these wired-memory costs.

Table 2-2 Wired memory generated by user-level software

Resource	Wired Memory Used by Kernel
Process	16 kilobytes
Thread	blocked in a continuation—5 kilobytes; blocked—21 kilobytes
Mach port	116 bytes
Mapping	32 bytes
Library	2 kilobytes plus 200 bytes for each task that uses it
Memory region	160 bytes

Note: These measurements will change with each new Mac OS X release. They are provided here to give you a rough estimate of the relative cost of system resource usage.

As you can see, each thread created, each subprocess forked, and each library linked contributes to the resident footprint of the system.

In addition to the memory generated through user-level requests, such kernel entities as VM objects, the virtual memory buffer cache, and I/O buffers also add to wired memory. Wired data structures are also associated with each physical page as well as a virtual-to-physical-memory hash table both of which scale with the amount of physical memory. Consequently, when you add memory to a system the wired memory increases even if nothing else changes. When the computer is first booted into the Finder, with no other applications running, wired memory consumes approximately 14 megabytes of a 64 megabyte system and 17 megabytes of a 128 megabyte system.

Wired memory is not immediately released back to the free list when it is no longer valid. Instead it is "garbage collected" when the free-page count falls below the threshold that triggers paging out.

Allocating, Copying, and Freeing Memory

Mac OS X implements a very fast allocation library that provides standard `malloc`, `calloc`, `realloc`, and `free` routines (among others). If you are currently using your own implementation of `malloc`, or one provided by your development environment, or perhaps Carbon's `NewPtr` and `NewHandle` family of routines, consider moving to the standard system `malloc` if possible.

The minimum gain for replacing a custom `malloc` implementation with the system `malloc` is shrinking your application's code by a few virtual memory pages with a resultant decrease in paging time; the maximum is much speedier memory management.

The source code for the system `malloc` library is available in the `gen` subproject of the Darwin `libc` project. For more information, see http://www.opensource.apple.com/projects/darwin/

Allocating Zero-Initialized Memory

Cross-platform code often allocates memory with `malloc` and then fills it with bytes containing a value of zero using `memset`. The problem with this approach is that the kernel allocates memory lazily, creating pages in memory only when you access them for the first time. When you call `memset` (after `malloc`) to zero the pages, you write to each page, thereby forcing all pages to be mapped into memory. This can be very expensive, especially if it requires other pages to be paged out first.

To allocate memory initially filled with zeroes, use the standard C function `calloc`. This approach allows the kernel to reserve the required virtual address space and let the virtual memory system allocate and zero pages only when they are accessed.

Understanding Malloc

`malloc` and related routines `calloc` and `realloc` use the kernel primitive routine `vm_allocate` to allocate memory. What `malloc` then does with the allocated memory differs according to the size of the allocation:

- **Small allocations**. For allocations less than a few virtual memory pages, `malloc` suballocates requested amounts from a list (or "pool") of free blocks of increasing size. If the pool is not yet allocated, malloc calls `vm_allocate` to allocate it. The granularity of the blocks `malloc` returns is 16 bytes. So if you ask for 4 bytes, `malloc` consumes 16 bytes, and if you ask for 24 bytes, `malloc` will consume 32 bytes. Any small blocks you deallocate (with `free`) are added back to the pool and are reused on a "best fit" basis. Small allocations, by nature smaller than a single page, cannot be page-aligned.
- **Large allocations**. For allocations greater than a few pages, `malloc` uses `vm_allocate` to obtain a block of the requested size. `vm_allocate` does not cause memory to be actually mapped in when allocated. Instead, `vm_allocate` creates an address range for the requested block and the virtual-memory system lazily maps in pages as addresses in that block are accessed. Memory allocated this way is guaranteed to be page-aligned, but not zero-filled.

Copying Memory Efficiently

There are two basic memory copying approaches in Mac OS X: physically copying the memory (`BlockMoveData` and `memcpy`) or marking the memory "copy-on-write" (`vm_copy`). Each is suited to certain situations.

`BlockMoveData` and `memcpy` copy memory by reading the bytes from a source block and writing the bytes to a destination block. These functions touch both the source and destination memory, and as a result the kernel must page in both of these address ranges. These functions are a good way to copy data when

- the size of the block copied is small (under 16 kilobytes)
- the source or destination block is not page aligned
- the source and destination blocks overlap

The other alternative for copying blocks of memory is the kernel routine `vm_copy`. In contrast to `memcpy`, `vm_copy` does not touch any real memory. The function performs the "copy" by changing the virtual-memory mapping data structure to indicate that the destination address range should really be a copy-on-write of the source address range. No data is actually moved until the destination buffer is actually modified.

The vm_copy alternative is best in these situations:

- The size of the memory to be copied is greater than 16 kilobytes.
- The source and destination are both page-aligned buffers. This is guaranteed if the blocks are allocated with malloc, but you must be sure to pass in the start address of the block, not an address within the block. NewPtr, NewHandle, and all other Mac OS 9 memory allocation calls do not necessarily return page-aligned blocks.

Finally, keep in mind the importance of releasing (via the free system routine) all memory that you have allocated with malloc, calloc, or realloc. Neglecting to release memory causes memory leaks, which have a direct impact on performance. To help track down memory leaks, use the MallocDebug application ("Debugging Allocations With MallocDebug" (page 72)) or the leaks command-line tool ("leaks Memory Leak Finder" (page 109)).

Using Multiple Malloc Zones

All memory blocks are contained within a malloc heap (commonly referred to as a **zone**). All allocations made using the malloc function occur within the standard malloc zone, which is created when malloc is first called. Although the practice does not have any practical benefit for most programs, you can create additional malloc zones and allocate memory in a specific zone.

Zones have the advantage of allowing blocks with similar access patterns or lifetimes to be placed together, theoretically minimizing wasted space or paging activity. Zones are created and destroyed dynamically, so you can allocate many objects in a zone and then destroy the zone to free them all.

For most developers, however, zones fail to deliver a performance advantage, and you should avoid them unless you need to either track a set of memory blocks separately from other allocations or free many memory blocks quickly.

Debugging Allocations With Malloc

`malloc` provides debugging features to help you track down memory smashers, heap corruption, references to freed memory, and buffer overruns. These options (listed in Table 2-3) are enabled by creating an environment variable with the name of the option before executing your program on the command-line. Except for `MallocCheckHeapStart` and `MallocCheckHeapEach`, the value to which you set the environment variable is ignored. See "Quick Command-Line Primer" (page 63) for more information on using environment variables).

Table 2-3 Malloc environment variables

Variable	Description
`MallocStackLogging`	If set, `malloc` remembers the function call stack at the time of each allocation. This information is purged when the block is released with `free`.
`MallocStackLoggingNoCompact`	Like `MallocStackLogging`, but retains the function call stack when the block is released.
`MallocScribble`	If set, `free` sets each byte of every released block to the value `0x55`.
`MallocGuardEdges`	If set, `malloc` adds guard pages before and after large allocations.
`MallocDoNotProtectPrelude`	Fine-grain control over the behavior of `MallocGuardEdges`: If set, `malloc` does not place a guard page at the head of each large block allocation.
`MallocDoNotProtectPostlude`	Fine-grain control over the behavior of `MallocGuardEdges`: If set, `malloc` does not place a guard page at the tail of each large block allocation.
`MallocCheckHeapStart`	Set this to the number of allocations before `malloc` will begin validating the heap. If not set, `malloc` does not validate the heap.
`MallocCheckHeapEach`	The number of allocations before `malloc` should validate the heap. If not set, `malloc` does not validate the heap.

Deferring Memory Allocation

Every memory allocation has a cost. Applications often allocate memory during initialization and then use it later—or sometimes not at all during a given session. You can easily improve on this costly approach by deferring the allocation to the first time the memory is needed. To accomplish this with a minimum of code modification, do these two things:

- Turn your global variable into a static variable so it no longer can be accessed directly by code in other modules.
- Create an accessor to access the static variable and allocate and initialize the buffer for it upon the first invocation.

Listing 2-1 gives an example of this technique.

Listing 2-1 Lazy allocation of memory through an accessor

```
MyGlobalInfo * GetGlobalBuffer()
{
    static MyGlobalInfo * sGlobalBuffer = NULL;
    if ( sGlobalBuffer == NULL )
        {
            sGlobalBuffer = malloc( sizeof( MyGlobalInfo ) );
        }
        return sGlobalBuffer;
}
```

Then call this accessor whenever you need to access this “global” data.

Note: This code is not safe in the presence of multiple threads. More than one thread could call this function simultaneously, causing the memory to be allocated more than once. To make it threadsafe, add a semaphore lock before the `if`-statement and unlock after the `if`-statement, and be sure that the code used to initialize the memory (if needed) is located inside the `if`-statement.

Not allocating memory until it is actually needed, as in the example in Listing 2-1, is a general performance technique. For example, if you have a collection sized for a maximum number of entries, you can change it into a dynamically grown list or array so only the memory that is actually needed is allocated.

You can also defer allocation of memory using file mapping. See "Reading Large Files With File Mapping" (page 29) for more information.

C H A P T E R 3

Accessing the File System

This chapter is intended to provide an understanding of the impact of common I/O operations on your application's performance. The user-perceived speed of any operating system is crucially constrained by the speed of the hard disk. Unfortunately, the hard disk is probably the slowest fixed-memory device attached to most computers. Also, because networking can be completely transparent to the user in Mac OS X, the likelihood that your application will be manipulating files on a network, or even running from a distant network server, is extremely high.

Speed of Hard Disk Drives

Hard disks, when used with real-world multitasking operating systems, can be much slower than is commonly supposed. Apple has determined that "modern" disks running under Mac OS X are able to sustain *random*, real-world read operations and write operations that are page-sized (4096 bytes) at approximately the following rates:

- 5400 rpm—60 pages per second (240 KB/sec)
- 7200 rpm—80 pages per second (320 KB/sec)
- 10000 rpm—100 pages per second (400 KB/sec)

These same drives may be able to sustain **sequential** reads and writes approaching 30 megabytes per second (30 MB/s).

Disks accessed over the network add unpredictable networking latency and transfer protocol overhead to these statistics.

General I/O Guidelines

Although there are no absolutes in a complex world, there are some basic guidelines that can be applied to reduce the I/O throughput of your program, and thus enhance its performance. As with all such improvements, it is important to measure the performance of the code being optimized before and after optimization to ensure that it actually gets faster.

- Reading is typically cheaper than writing data.
- Defer any I/O operation 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.
- 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.

Caching Data in RAM

Reading data from disk and storing it in allocated memory can be more expensive than releasing that memory, later reclaiming the memory, and recreating the data from disk. A cache in RAM is subject to the normal overhead of the virtual memory system and therefore may eventually be swapped out and then later paged in again. Thus, a cache of *n* bytes may cost 2*n* bytes worth of I/O activity. Measuring the effects of such a RAM cache on a computer with a small amount of physical RAM (such as 64 megabytes) is important, to see the difference in an environment where paging is more likely to occur.

Reading Large Files With File Mapping

Mapping a large file into memory (rather than performing a number of sequential read operations on that file) can significantly enhance I/O performance. Why is this? Reading a file into a buffer can directly incur up to three I/O operations:

- paging out enough modified pages to allow for the allocation of the read buffer
- reading bytes from the file into the buffer
- later, paging out the buffer memory

Additionally, any pages paged out will likely need to be paged in again.

Mapping a file, on the other hand, causes the file to be treated as an extension of virtual memory: Conceptually, the mapped file is simply another swap file mapped to a range of memory. Thus, reading from the file can incur only two I/O operations directly: paging in the desired section of the file and paging out any modified pages needed to make room for the page in memory. (See "Virtual Memory on Mac OS X" (page 15) for more information on the operation of the virtual memory manager.)

The pages of a file are maintained in a shareable state in memory. If the system runs low on memory, the pages are purged, but no I/O operation is incurred; if the data is referenced again, it is paged in again from the file.

Additionally, your code is often smaller with file mapping, because the file is accessed through a pointer, like all random-access memory, and no file system calls need be used.

The BSD routines `mmap` and `munmap` map and unmap files, respectively. Listing 3-1 demonstrates the use of `mmap`. The mapped file occupies virtual address space until `munmap` is used to unmap the file (or until the application is terminated).

Listing 3-1 File mapping

```
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 "/"-delimited pathname
//
//  Exit:   outDataPtra      pointer to the mapped memory region
//          outDataLength    size of the mapped memory region
//          return value     either an errno error condition
//                           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;
}
```

Tracking File System Changes

Many applications have cause to poll the file system for changes.

- Applications with Finder-style file list views (as seen in Sherlock 2's lower pane) need to synchronize with changes made by the user in the Finder.
- Document synchronization, in which the application watches for filename modifications and changes the document's window title accordingly, or closes a document window when the associated file is moved to the Trash.
- "Folder watching, in which an application processes files as they are dropped into a specified folder.

Polling the file system is bad for performance. Among other issues, polling uses I/O bandwidth excessively, and polling also tends to fill low-level file system caches with entries that will likely cause many cache misses until they are purged.

Synchronizing Files on Window Activation

Applications should synchronize file state only when the associated document window becomes active. Synchronizing the file state involves, for example, updating the window title and proxy icon if the document's filename or file type is changed by another application. Information synchronized from the file should *not* be updated at regular timed intervals. In Cocoa applications, implement this behavior in the `windowDidBecomeMain` method of the window delegate. In Carbon applications, implement this behavior in a Carbon Event Manager `kWindowActivateEvent` event handler.

Reducing the Folder-Watching Interval

Applications watching folders for the creation of new files should keep their polling intervals high—from five to fifteen seconds, higher being better for system performance.

Using Pathnames

Mac OS X's native file system manager implements the POSIX file system API, which requires applications to specify the location of files using BSD-style forward-slash-delimited pathnames. For best possible performance on all file systems, BSD path names are the recommended way to access files.

Although Apple recommends moving resources into data files in the application package `Resources` directory, it is possible to access the resource fork of a file on an HFS Plus volume by adding the suffix:`/..namedfork/rsrc` to the end of the file pathname. Because this doesn't work on other file systems, notably UFS, and because it requires you to parse the resource fork structure directly, this technique is not recommended.

See *Inside Mac OS X: System Overview* for more about application packages.

Optimizing Carbon Programs for Mac OS X

Although your Carbon application can largely benefit from running on Mac OS X without significant changes, following the guidelines detailed in this chapter will help you avoid potential performance trouble spots. These guidelines are all compatible with Mac OS 9, except where noted.

Carbon and the Mach-O Executable Format

If you have a Carbon application that is based on the Code Fragment Manager Preferred Executable Format (PEF), you should consider switching to the Mach-O executable format, for several reasons:

- On Mac OS X, the libraries that implement the Carbon environment use the Mach-O executable format. Mach-O executables use a calling convention different from that used by PEF executables. Calls made to or from PEF code fragments must be translated at runtime. While the translation overhead is small, it is altogether unnecessary if you are using Mach-O.
- Apple's Mac OS X development environment supports only Mach-O. Whether or not you use Apple's development environment for Mac OS X, the Mac OS X performance tools are significantly easier to use with Mach-O executables than with PEF.
- Mach-O was designed and optimized for use with the Mac OS X virtual memory system.
- Mach-O executables can directly call other Mach-O shared libraries and BSD API routines in the kernel.

- Mach-O supports just-in-time binding, where a link to a function is resolved when that function is first called. All links in a PEF-based application (and all PEF libraries it links to) must be resolved when the application is launched.
- Mach-O is not supported on Mac OS 9, but using Mach-O does not require you to abandon Mac OS 9 as a delivery platform. You can build an application package that runs a PEF binary on Mac OS 9 and a Mach-O binary on Mac OS X. This allows you to optimize your executable for each operating system that you wish to support. For more information, see the section on application packages in *Inside Mac OS X: System Overview*.

See "Overview of the Mach-O Executable Format" (page 117) for more detailed information about Mach-O.

Carbon and Mac OS X Events

For maximum performance potential, you should consider using the Carbon Event Manager routines to handle user interface events.

Adopting the Carbon Event Manager

The advantages of full adoption of the Carbon Event Manager include these:

- Applications that use the classic Event Manager must contain code to perform basic event routing and focus management tasks. The Carbon Event Manager does all of this work for you, freeing you to write code for the tasks unique to your application.
- The Carbon Event Manager is designed to allow easy extensibility. No longer do you have to define complex extension mechanisms to allow plug-ins to handle basic user interface events.
- Because less code is required to implement basic event-handling tasks, your executables are smaller and faster.

- Because you install event handling routines only for events you wish to receive, on the targets that will receive them, the Carbon Event Manager can optimize event posting. Events are posted to the system event queue only if the application requires them. With the classic Event Manager routines, the system must post events that applications often ignore.
- The Carbon Event Manager is designed to encourage better overall system performance by replacing polling with blocking. This concept is detailed further in the next section.

Avoiding Polling Behavior

Polling is the process of repeatedly checking a resource to learn whether it has changed state. For example, an application might poll a network connection to see if new data is available for reading. Another example is the classic Event Manager routine `GetNextEvent`, used by many older applications to poll for user interaction events.

Polling is inherently inefficient because your program must continuously spend CPU time to repeatedly discover whether the value of some element of system state has changed. Typically, the state does not change more often than the program can poll. For example, a program might wait for the mouse button to be pressed. With polling, the program checks the state of the mouse several hundred times a second. This uses up a large amount of the computer's total available CPU time.

With **blocking**, when the system changes a piece of state that programs might be interested in, the system sends a notification about the change. Interested programs can block until the notification is received. For example, a program waiting for the mouse button to be pressed might call a function that blocks until the mouse button is pressed. The program still knows when the mouse button is pressed, but CPU time that would otherwise be wasted checking for the mouse button is free for other programs to use.

A blocked thread is a thread that is currently suspended. It does not execute and occupies no system resources other than a small amount of memory. Typically, a thread is unblocked (resumed) when a low-level event, message, or notification is received. At that point, the thread is permitted to execute and consume system resources (such as CPU time) again.

To eliminate the most common polling scenarios (such as the mouse button example provided above), Mac OS X provides modern programming interfaces that support blocking, such as the Carbon Event Manager.

The following sections detail changes you may need to make to your code to eliminate polling for user interface events. See also "Tracking File System Changes" (page 32) for information on avoiding file system polling.

Tracking the Mouse With WaitNextEvent

Some third-party application frameworks keep track of the current mouse location by specifying a region consisting of a single pixel in the `mouseRgn` parameter of `WaitNextEvent`. This causes a mouse-moved event to be generated every time the user moves the mouse, though in most cases the application does not need to know that the mouse has moved. This is extremely wasteful of CPU time.

You should instead specify a region defining the boundaries beyond which you must know whether the mouse has moved. Another alternative is to use a Carbon event timer to periodically find the object located under the mouse.

Tracking the Mouse on Mouse Down

In Mac OS 9, tracking the mouse when the mouse button is pressed usually involves polling the mouse in a tight loop, as demonstrated in Listing 4-1 (page 39).

Mac OS 9 Event Manager functions such as `StillDown` and `WaitMouseUp` were designed to be used in a polling loop, rather than a blocking loop. Both of these routines return immediately, which forces your application to call them repeatedly even while the mouse location does not change. This is extremely wasteful of CPU time, but, using the Mac OS 9 Event Manager, there is no other simple way to track the mouse in a modal fashion.

To eliminate this performance drain, the Carbon Event Manager function `TrackMouseLocation` was created. When called, `TrackMouseLocation` blocks the current thread until the state of the mouse changes. The CPU is free to perform other tasks while `TrackMouseLocation` is waiting for the mouse. Listing 4-2 (page 40) demonstrates the use of `TrackMouseLocation`.

To see the performance benefit of switching to `TrackMouseLocation`, run these two code samples with either a Terminal window running `top` or the CPU Monitor application visible in the background. You should notice a large difference in the `CPU time %` column.

Listing 4-1 Modal mouse tracking prior to Carbon

```
void TrackMarquee( Point inStartPoint, Rect * outRect )
{
    Pattern                 pattern = { 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55 };
    Point                   mouse;
    Rect                    rect;
    Rect                    previousRect = {0,0,0,0};
    GrafPtr                 port;

    GetPort(&port);

    PenPat( &pattern );
    PenMode(srcXor);
    PenSize( 2, 2 );

    SetRect( &rect, inStartPoint.h, inStartPoint.v, inStartPoint.h, inStartPoint.v );

    while( StillDown() )
    {
        GetMouse(&mouse);
        SetMobiusRect(&rect, inStartPoint.h, inStartPoint.v, mouse.h, mouse.v);

        if( ! EqualRect( &rect, &previousRect ) )
        {
            // erase previous
            FrameRect( &previousRect );

            // draw next
            FrameRect( &rect );

            previousRect = rect;

            // flush window buffer to screen
            QDFlushPortBuffer(port, NULL);
        }

    }

    // erase final rect
    FrameRect( &rect );

    // clean up the grafport
    NormalizeThemeDrawingState();

    *outRect = rect;

}
```

Listing 4-2 Modal mouse tracking in Carbon

```
void TrackMarquee( Point inStartPoint, Rect * outRect )
{
    Pattern                 pattern = { 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55 };
    Point                   mouse;
    Rect                    rect;
    Rect                    previousRect = {0,0,0,0};
    GrafPtr                 port;
    OSStatus                status;
    MouseTrackingResult     trackingResult;

    GetPort(&port);

    PenPat( &pattern );
    PenMode(srcXor);
    PenSize( 2, 2 );

    SetRect( &rect, inStartPoint.h, inStartPoint.v, inStartPoint.h, inStartPoint.v );

    mouse = inStartPoint;

    do
    {

        SetMobiusRect(&rect, inStartPoint.h, inStartPoint.v, mouse.h, mouse.v);
        if( ! EqualRect( &rect, &previousRect ) )
        {
            // erase previous
            FrameRect( &previousRect );

            // draw next
            FrameRect( &rect );

            previousRect = rect;

            // flush window buffer to screen
            QDFlushPortBuffer(port, NULL);

        }

        // get next mouse location
        status = TrackMouseLocation( port, &mouse, &trackingResult );
        if( status != noErr )
        {
            break;
        }

    } while( trackingResult != kMouseTrackingMouseReleased );

    // erase final rect
    FrameRect( &rect );

    // clean up the grafport
    NormalizeThemeDrawingState();

    *outRect = rect;

}
```

Watching for Modifier Keys

Applications commonly poll the keyboard (using a function such as `GetKeys`) to learn the current state of the modifier keys (Shift, Option, Control, and Command.) To help applications remove this behavior, the Carbon Event Manager sends modifier-key state change events to registered event handlers of class `kEventClassKeyboard` and type `kEventRawKeyModifiersChanged`. Listing 4-3 demonstrates the use of this event type.

Listing 4-3 Receiving modifier-key events

```
int main(void)
{
    OSStatus             status;
    EventHandlerUPP      eventHandlerUPP;
    EventHandlerRef      eventHandlerRef;
    EventTypeSpec        eventTypes[] = { {kEventClassKeyboard,
                                              kEventRawKeyModifiersChanged} };

    //
    // Install Apple event handler for Quit events
    //
    status = AEInstallEventHandler( kCoreEventClass,
                                     kAEQuitApplication,
                                     NewAEEventHandlerUPP(QuitAppleEventHandler),
                                     0,
                                     false);
    require_noerr( status, NoAppleEvents );

    //
    // Install the modifier-key-changed handler
    //
    eventHandlerUPP = NewEventHandlerUPP(ModifierKeysChangedEventHandler);
    require( eventHandlerUPP != NULL, NoEventHandler );

    status = InstallApplicationEventHandler(  eventHandlerUPP,
                                              1,
                                              &eventTypes[0],
                                              0,
                                              &eventHandlerRef);
    require_noerr( status, InstallEventHandlerFailed );

    //
    // Set the cursor to the arrow (on Mac OS 9 and earlier)
    // and start running the Carbon Event Manager event loop
    //
    InitCursor();

    RunApplicationEventLoop();

InstallEventHandlerFailed:
NoEventHandler:
NoAppleEvents:
NoMenuBar:
    return 0;
}

//
// ModifierKeysChangedEventHandler
//
//  Watch for modifier key-down events
//
static pascal OSStatus
ModifierKeysChangedEventHandler(EventHandlerCallRef, EventRef event, void *)
{
    UInt32              modifiers;
    OSStatus            result = eventNotHandledErr;

    static Boolean      optionWasDown = false;

    //
    // Use the Speech Manager to speak the string “down” when the Option key is pressed
    // When the Option key is released, speak the string “up”
    //
    result = GetEventParameter(   event,
                                  kEventParamKeyModifiers,
                                  typeUInt32,
                                  NULL,
                                  sizeof(UInt32),
                                  NULL,
                                  &modifiers);

    if( result == noErr )
    {
        if( ((modifiers & optionKey) == optionKey)
            && (! optionWasDown) )
        {
            SpeakString( "\pdown" );
            optionWasDown = true;
        }
        else if( optionWasDown )
        {
            SpeakString( "\pup" );
            optionWasDown = false;
        }
    }

    return result;
}

static pascal OSErr QuitAppleEventHandler(const AppleEvent *, AppleEvent*, long)
{
    //
    // Quit the Carbon Event Manager event loop.
    //
    QuitApplicationEventLoop();
    return noErr;
}
```

Carbon and the Mac OS X File System

Mac OS X can be used to access files on a variety of file systems and volume formats, including those listed in Table 4-1 (page 45). Although the primary volume format is HFS Plus, Mac OS X can also boot from a disk formatted with the UFS file system.

Future versions of Mac OS X may be bootable with other volume formats as well. This section provides guidelines for good performance when accessing files on any file system.

Table 4-1 File systems supported by Mac 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 Mac OS X.
UFS	Unix File System. A variant of the BSD "Fast File System."
WebDAV	Used for directly accessing files on the web.
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.
Samba	Used for sharing files with Microsoft Windows SMB file servers.
AFP	AppleTalk Filing Protocol. The Mac OS 9 file sharing standard.
NFS	Network File System. A commonly-used BSD file sharing standard.

Adopting the HFS Plus API

Every file system stores a different set of **metadata**, data associated with a file but not part of the file itself. The definition of metadata includes attributes such as Macintosh file type information, BSD-style file access permissions, and creation and modification dates. Metadata not accounted for in the design of the file system must be either stored inefficiently or calculated expensively at runtime.

For example, a single call to `PBGetCatInfoSync` returns Finder file type information from a file or folder. On HFS and HFS Plus volumes, there is no extra cost for this metadata because it is stored in the file's catalog node, and thus is read into memory along with the name of the file. Other file systems, however, must compute some of the seldom-used fields using expensive I/O operations. For example:

- The object count (“valence”) for directories and volumes requires a recursive subdirectory iteration on most file systems.
- Finder information, such as type and creator codes, might require opening and reading a separate file.

During the initial design of the HFS Plus API, it was observed that, although all of this metadata can be valuable to access, applications tend not to use all of it at the same time. To improve efficiency with alternative file systems, the HFS Plus API was designed to allow only the information actually required by the client application to be retrieved, thereby eliminating the potential overhead of unneeded calculation or read operations.

The HFS Plus API additionally provides efficient routines to iterate through the contents of directories (see “Iterating Over the Contents of a Directory” (page 46)).

Iterating Over the Contents of a Directory

The example in Listing 4-4 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. If you need to scan a directory repeatedly in order to watch for changes (for example, new files added or removed to a directory), see also“Tracking File System Changes” (page 32).

Listing 4-4 Fast directory iteration

```
int main(void)
{
    OSStatus     outStatus;
    FSSpec       spec;
    FSRef        folderRef;

    printf("begin file iteration!\n");
    fflush( stdout );

    //
    // Get the currently running application’s parent folder,
    // make it into an FSRef, and iterate it
    //
    outStatus = FSMakeFSSpec( 0, 0, "\p", &spec );
    if( outStatus == noErr )
    {
        outStatus = FSpMakeFSRef( &spec, &folderRef );
        if( outStatus == noErr )
        {
            outStatus = IterateFolder( &folderRef );
        }
    }

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

    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)
                 );
    }
}
```

Converting Pathnames to File System References

The routines that convert a pathname to an `FSSpec` structure or an `FSRef` structure (`FSPathMakeFSSpec` and `FSPathMakeFSRef`, respectively) must perform conversions that take a long time to complete. Consider caching the returned file system reference in memory to avoid calling these routines often.

Carbon and Mac OS X Graphics

All drawing into windows on Mac OS X is double-buffered unless you explicitly request otherwise. When you draw content into the graphics port (`GrafPort`) of a window, you are actually drawing into the offscreen drawing buffer associated with the window. The content being drawn does not appear onscreen until `QDFlushPortBuffer` is called.

`QDFlushPortBuffer` is called by the Carbon Event Manager whenever

- an event is retrieved (either `WaitNextEvent` is called or an event handler returns control to the Carbon Event Manager)
- a Human Interface Toolbox routine that must draw implicitly (such as `TEIdle` or `TEClick`) is called

The buffer is *not* flushed when QuickDraw drawing routines (such as `LineTo`, `FrameRect`, and `CopyBits`) are called or when controls are drawn. Generally, applications don't need to flush the port, because the port is flushed at event retrieval time.

Flushing the Port

Many small port buffer flushes generally take significantly more time to complete than one large port buffer flush. The best thing to do is to wait for the system to draw at event loop time.

If you cannot wait for the system to flush the port, the best tactic is to wait until many small flushes have accumulated and then flush the port buffer. Avoid flushing after every call to `FrameRect` or `LineTo` or `CopyBits`. Instead, flush when all content is drawn.

Buffering Windows With Offscreen Drawing

As noted above, the window is buffered in Mac OS X, unless you have chosen otherwise. If your application is maintaining an offscreen graphics world (GWorld) for each window or otherwise buffering the window contents during drawing, be sure to disable or conditionalize that code, because it serves no purpose on Mac OS X other than to occupy memory and slow down window drawing.

Rendering Controls

When changing the attributes of a large number of controls, consider using `SetControlVisibility` on the root control to prevent redundant drawing. All Control Manager functions that alter the appearance of a control immediately cause the control to be redrawn. Although the port is not flushed until `QDFlushPortBuffer` is called, rendering controls still takes time, especially given the computationally expensive nature of the Aqua user interface.

C H A P T E R 5

Building Efficient C, C++, and Java Programs

Mac OS X supports a wide variety of languages and development tools. This chapter details possible performance issues pertinent to C, C++, and Java.

Automated Code Optimization

Most development environments implement several levels of automated code optimization. Typical among these are the optimization of generated binary code and **dead code stripping**, which is the process of removing unused and unreferenced variables, functions, and methods from the output executable file.

Enabling Compiler Optimization

One standard, easy way to optimize your code is to enable compiler optimization. The standard Mac OS X compiler is `gcc`, and it has several command-line options to control the general level of optimization, summarized in Table 5-1.

The primary benefit of compiler optimization is that it makes your code smaller, which reduces the amount of code that must be loaded into memory at any one time. Although reduction in size is generally preferable to an increase in execution speed (because of the impact on virtual memory), you should try different options and see what works best for you.

Table 5-1 Primary compiler optimization parameters

Parameter	Optimization performed
`-O` or `-O1`	Optimize to reduce code size and execution time.
`-O2`	Perform most optimizations not requiring a space-time trade off. Thus the compiler does not perform loop unrolling or function inlining.
`-O3`	Optimize for speed as opposed to size. May add aggressive function inlining.
`-Os`	Optimize for size as opposed to speed. This option is very similar to `-O2`.
`-O0` or no `-O` option given	Do not optimize. (The first character after the dash is the letter O and the second is the digit zero.)
`-fcoalesce-rtti`	C++ only. Enable coalescing of runtime type information. See "Coalescing Runtime Type Information" (page 59) for more information.

Note that, for each of the `-O` options, the first character after the dash is the letter O, not the digit zero.

In Project Builder, the `-O` parameters are specified in the Optimization Level pop-up menu in the Build Settings pane, as Level 1, Level 2, and Level 3 for `-O1`, `-O2`, and `-O3`, respectively. To specify other parameters in Project Builder, place them in the `$OTHER_CFLAGS` variable in the Build Settings pane.

If you specify multiple `-O` options, the last such option takes precedence.

Dead Code Stripping

Dead code stripping is typically performed by a development environment's static linker. The standard Mac OS X tools do not currently perform dead code stripping. If you include in your project a large file containing a collection of utilities, all of them are compiled and linked into your code, regardless of whether or not you actually use them all. Consequently, your code occupies more memory than is necessary.

The most effective solution is simply to scatter load your executable, as described in "Improving Locality of Reference" (page 126). The scatter loading process moves unused code and data to the same set of virtual memory pages, removing them from the effective memory footprint of your executable.

An alternative is to carefully examine your code and remove (or conditionalize) unused functions, globals, macros, and other code that your program doesn't actually use.

Inline Functions

Inlining is an optimization that allows the compiler to integrate the code of simple functions into their callers, thereby eliminating the overhead of function call setup and return. Inlining can be performed automatically by the compiler, or selectively by you, the developer, through use of the `inline` and `static` keywords.

These are the rules of thumb for inlining:

- An inline function should be (at most) two or three statements long.
- Good candidates for inlining are functions that increment, set, or return a single variable, or call a single function.

The `gcc` compiler only tries to inline functions that are declared `inline`, and only if optimization is enabled. For C programs, inline functions are not inlined unless they are declared `static` as well as `inline`.

With the `-finline-functions` option, the compiler aggressively tries to inline all functions even if they are not declared `inline`. This mechanism is known to have problems that could result in incorrect code generation. You should be especially careful to test the generated code if you use this option. Note that aggressive inlining can also result in a considerable increase in code size. Normally, the `-O3` option turns on the `-finline-functions` option; however, this has been disabled due to the problems mentioned here.

If you want to keep the compiler from expanding any functions inline, specify the `-fno-inline` option.

For C++ code, the compiler is not always able to inline functions that are declared `inline` in the class declaration, so local instances of these functions may be generated, even if the functions are not used.

C++ Performance Notes

C++ was designed with the performance goal of adding object-oriented features to C without incurring a cost for code that does not use those features. As such, using the C++ compiler options to compile ANSI C code should not generate an executable requiring significantly more execution time than without C++. However, both the exception-handling and runtime type information (RTTI) core language features do each increase the size of the executable by small amounts, even if the feature is not used.

In a nutshell, these are the Mac OS X C++ performance guidelines:

- Exceptions do not greatly increase the size of your executable, nor do exceptions incur a performance penalty at runtime.
- Still, you can achieve a small decrease in executable size by turning exceptions off by specifying the `-fno-exceptions` parameter to `gcc`.
- `throw` is expensive. Only use exceptions for exceptional cases, never as an alternative return mechanism.
- Use the `-fcoalesce-rtti` parameter to `gcc` to create only one global copy of each RTTI symbol.
- If you build a C++ library or framework and export RTTI information to clients, you cannot use `-fcoalesce-rtti`. Instead, create a trigger function to create one global copy of the RTTI symbols.
- If you do not use RTTI and you do not use exceptions, specify the `-fno-rtti` parameter to `gcc`.
- Avoid exception specifications (`void foo() throw(OSErr) { throw resNotFound; }`). They incur a significant performance penalty.

- Ensure that each of your concrete (non-abstract) classes contains at least one virtual function that is not marked `inline`. This avoids proliferation of duplicate v-tables and of RTTI symbols when `-fcoalesce-rtti` is not specified. See "Using Trigger Functions" (page 58) for more information.

Note that you cannot successfully link together object files compiled with different exception handling and RTTI settings.

The remainder of this section contains in-depth discussion of the above issues.

Understanding C++ Exceptions

When your code throws an exception, the compiler must unwind the program's stack, calling destructors for stack-allocated objects contained in each of the unwound stack frames, and jump to an exception handler—a `catch` clause—or abort the program if no exception handler is found.

In order to unwind the stack and invoke destructors for the unwound stack frames, the code used to support exception handling must have detailed information about the contents of each stack frame. Many compilers insert code into the program's functions to build this information at runtime, but this code has major costs in both execution time and executable size.

Mac OS X `gcc` uses an alternative method called **zero-runtime-overhead exception handling**. With this method, the compiler generates the exception-handling tables at compile time. Zero-runtime-overhead exception handling is cheap and fast. Your code doesn't incur runtime overhead until an exception is thrown.

The caveat is that the compiler generates exception-handling information for every function in a C++ program. Because an exception can be thrown through a routine that does not itself either catch or throw exceptions, the compiler must generate tracking information for all functions that call other functions. Because the compiler enables exceptions by default (as required by the ANSI C++ standard), your executable may be slightly larger than it needs to be. You can use the `-fno-exceptions` compiler option to disable exception support and make your executable slightly smaller.

Also, be aware that a `throw` statement is much more expensive than a `return` statement. Exceptions should be used only in exceptional cases, when a genuine error condition has been discovered. If your code is designed to throw exceptions during normal operation, your program is slower than it could be. You should avoid using exception handling as an alternate return value mechanism.

Using Trigger Functions

There are two constructs associated with classes in C++: the v-table and the RTTI symbol.

"V-table" is short for **virtual function table**. A v-table is an array of pointers to virtual methods and support code. Every C++ object containing one or more virtual methods contains a v-table.

The RTTI symbol is a data structure that uniquely identifies the class with which it is associated. The `dynamic_cast` keyword and the `typeid` function both use the RTTI symbol. Every class contains an RTTI symbol.

The compiler must decide where in the generated code to place both the RTTI symbol and the v-table. If the compiler cannot decide, it makes a separate copy of the v-table and RTTI symbol in each object file that the class is referenced in. This can lead to a large amount of space wasted on duplicate information. To help the compiler decide (and thus avoid this waste of space), C++ uses the concept of a **trigger function**.

A trigger function is a function that identifies a location for the compiler to place the data associated with a class definition, including the RTTI symbol and the v-table. The compiler generates the v-table and RTTI symbol associated with the class when it finds the trigger function definition in your source file. When the compiler sees other references to this class, it knows that the class data is placed at the location of the trigger function, and doesn't need to generate an additional v-table or RTTI symbol.

The `gcc` compiler considers the first virtual function (not pure virtual, not marked `inline`) listed in the class declaration to be the trigger function. Functions in base classes do not count. If you declare the trigger function in the header, but never implement the trigger function in a source file, the v-table and RTTI symbols are never defined.

Most concrete (non-abstract) classes with v-tables do contain at least one virtual function that is not marked `inline`, but watch out for subclasses that define no virtual functions of their own and also for classes with only `inline` virtual functions.

Coalescing Runtime Type Information

The easiest method for removing duplicate RTTI symbols is to allow the tools to do it for you. The `-fcoalesce-rtti` parameter to `gcc` eliminates redundant RTTI symbols, regardless of trigger functions.

If you build a C++ library or framework with `-fcoalesce-rtti` enabled, your clients cannot use RTTI-based features, because the RTTI coalescing mechanism causes the RTTI symbols to be made private, no longer exported to clients that link to your library.

Java Performance Notes

The performance of a Java application is often largely determined by the core libraries it is based on. Additionally, there are some unavoidable issues with standard implementations of Java. Standard libraries commonly manifest all of these problems:

- Overuse of synchronization. Thread synchronization overhead is very expensive. Make sure the `synchronized` keyword is applied at as fine a granularity as possible. Be aware that `static synchronized` methods require locking the entire class. In Java 1.2 and later, synchronization overhead can sometimes be eliminated by using thread-local variables rather than singleton `static` globals. See the Java SDK documentation on `java.lang.ThreadLocal` for more information.
- Allocation of many small objects. Library interfaces are often designed to return small objects in Java because it's convenient, and, with garbage collection, there is no need for the client to explicitly deallocate the memory. However, heap allocation for many tiny objects is expensive. In C and C++, objects can be explicitly allocated on the stack to remove the overhead of allocation; in Java, they can't.

- Overuse of exceptions. Exception handling in Java is very slow. Exception guidelines for C++ also apply to Java: Use exceptions only for exceptional cases.
- Storing each class in an individual `.class` file. Classes should be grouped together in `.jar` files. Opening and closing many small `.class` files is more expensive than opening and closing one large `.jar` file. Grouping classes in `.jar` files allows the class loader to efficiently use file mapping. See also "Reading Large Files With File Mapping" (page 29).

Analyzing Performance

Apple provides a comprehensive suite of performance analysis tools for Mac OS X. These tools can measure and diagnose many aspects of system and program performance, from memory usage and memory leaks to file-system access patterns and paging activity.

Using the Performance Tools

Some of the performance tools are applications (with graphical user interfaces) and others are command-line utilities that must be invoked from the Terminal application or, less invasively, through `ssh` or `telnet` from a remote machine.

Most of the graphical applications have online help available through the Help menu. All of the command-line tools have **man pages**, accessed on the command-line by typing "`man` *tool name*".

It's worthwhile to note key statistics and compare them over time as you fine-tune your application, framework, or other executable. You can thereby spot regressions or improvements in your executable's performance.

Finding Problems

The performance tools can help you debug problems and provide a mental model of the inner workings of your code, giving you the information you need to make your program run faster.

For example, your program appears to be using system resources at an inordinate rate. Perhaps excessive disk thrashing is occurring; maybe the response time of the user interface is very slow. You may be leaking memory, or your code may be spending time spinning when it could be blocking instead.

- Use the `top` tool to discover possible excess usage of resources.
- To trace perceivable freeze behavior to a source cause, the `gdb` debugger can be used to examine your application at runtime.
- To find memory leaks, use MallocDebug or the `leaks` tool.

Exploring Your Code

Understanding the runtime flow of your code, its purpose and meaning, is crucial to both improving performance and shortening debugging time.

- The `top` tool is useful for determining possible excessive use of resources. `vmmap` provides you with a high-level display of the layout of your application's virtual address space.
- Sampler allows you to quickly figure out which functions use more overall execution time than others. The `gprof` tool can be used for complete recording and timing of your program's function call tree.
- QuartzDebug can display the amount of memory allocated to windows and provides options for discovering excessive drawing to the screen.

MallocDebug and ObjectAlloc provide fine-grained allocation tracking, allowing you to gain a complete understanding of the allocation behavior of your application.

- ObjectAlloc tracks the historical, repetitive allocations, recording data at runtime.
- MallocDebug allows you to collect a snapshot of the memory allocated by your program at a given moment in time. MallocDebug also provides powerful debugging features.

Although the performance applications are not currently AppleScript-scriptable, most allow you to export performance data for analysis and manipulation by scripting and command-line tools

Quick Command-Line Primer

Many of the performance tools run only on the BSD command line. This section provides some brief usage information about the command line, just enough for basic debugging. To learn more about facilities available at the command line, see "Further Investigations" (page 10).

The Terminal application is Mac OS X's gateway to the BSD command-line shell. Each window in Terminal contains a complete shell execution context, separate from all other shell execution contexts. The shell itself is an interactive programming language interpreter. Like the C language, the shell programming language offers control structures and variables of different scope. Different shells feature slightly different syntax and abilities. While you can use any shell of your choice, the examples in this book assume that you are using the standard Mac OS X shell, `tcsh`, or else a shell supporting syntax similar to `tcsh`.

To run a program in the shell, you must type the complete pathname of the program's executable file (and then press the Return key). You do not need to type the full name of programs that can be found in the directories listed in the shell's `PATH` variable, including most of the command-line tools you will read about in this chapter.

To launch application packages, you can either use the `open` tool (`open MyApp.app`) or launch the application by directly typing the pathname of the executable file, usually something like `MyApp.app/Contents/MacOS/MyApp`.

Some of the tools and techniques referenced in this chapter require the use of environment variables. Environment variables are variables inherited by all executables executed in the shell's context. For example, to execute a debugger called `FooDebugger` that requires the environment variable `DebugFunction` set to the name of your function, type these lines:

```
setenv DebugFunction yourFunctionName
/LocationOfFooDebugger/FooDebugger
```

Because each shell is a separate execution context, variables you set in one copy of a shell are not set in another copy of a shell. Thus, if you open two Terminal windows and set an environment variable in one window, programs executed from the other Terminal window do not get the new environment variable.

You can use the standard BSD shell output redirection notation `command > file` to log the output of command-line tools to a file.

Code Profiling With Sampler

Sampler is an application that analyzes a program's running behavior and its allocation of memory. Sampler stops the program periodically to examine its function call stack. Sampler then displays the functions that were most frequently seen while sampling was taking place. This information can help you locate functions consuming large chunks of CPU time and functions in which excessive memory allocations are occurring. You can thereby find spots in your code where execution time or allocation size is more than you expect, and then improve your code to reduce running time or memory usage.

Sampler works on any executable. It does not require code changes, recompilation, or the linking of special "profiling" libraries.

What Sampler Does

To analyze those parts of a program that are frequently called when that program is running, Sampler performs a statistical sampling of the target program over a period of time. At *n*-millisecond intervals within that period, Sampler takes a snapshot of the stack of each thread, where *n* is a user-specified value greater than or equal to 10 milliseconds.

At the end of the sampling period, Sampler takes the collected data and shows the call graph of the program in the Call Graph browser; for each function listed in this browser it displays the number of samples where the program was executing that function in the call graph. By sampling the running behavior of your program, Sampler helps you identify which routines your application is spending most of its time in, and hints at functions that may be good candidates for tuning.

Each time it samples, Sampler records the function call stack—the hierarchy of functions called to reach that point in the execution. (Call stacks are also called "stack traces" or "stack crawls," and are usually displayed in debuggers to show how you reached the current function.) Next to the name of each function, Sampler displays the count of samples where the functions at the top of the stack match a specific chain of functions. The Call Graph browser combines call stacks with the same functions at the top of the call stack. This resulting tree roughly provides a call graph of the program, displaying which functions are called by `main`, and which functions are called by the callees of `main`. The call graph represents the observed running behavior of your program, not just an analysis of what might happen.

Sampler's form of performance analysis complements other methods such as CPU-usage analysis or profiling (see "gprof Code Profiler" (page 106)). Although measuring CPU time can show you how much time your application spends running, sampling can be more effective because it includes time spent when the application was blocked and waiting for system resources. Unlike the `gprof` tool, Sampler does not require you to recompile your program.

However, also unlike `gprof`, Sampler only shows where it found the program executing when it did one of its periodic examinations. Profiling shows perfect call graph information, as gathered by the profiling instrumentation code. Profiling records exactly which functions were called from a given function, which, combined with the timing data, may help you to determine which bits of code are expensive. With Sampler, functions that take little time to execute and do not call other functions may not end up in the call stacks unless they are executing at the moment a sampling is taken. On average, however, Sampler should see these functions in proportion to the time the function takes to run. You can improve Sampler's accuracy with shorter sampling intervals and longer sampling sessions.

Other Sampler Modes

Sampler can display information other than frequently executed functions:

- memory allocations
- file-system usage information
- calls to specific functions

Watching for Allocations

When it analyzes the allocations made by a program, Sampler records the amount of memory allocated and the call stack each time your program calls an allocation routine (`malloc`, for example). With this information, you see not only where you allocate memory and how much memory is being allocated, but also identify the context when `malloc` was being called.

Sampler's version of memory allocation analysis differs from MallocDebug's in that it doesn't require a special debugging library to be inserted in the context of the application being debugged, and thus doesn't have problems with `setuid` applications. Unlike MallocDebug, it detects leaked cycles and structures. However, because Sampler doesn't understand Carbon handles, it misses memory leaks in blocks allocated with `NewHandle`.

See also "leaks Memory Leak Finder" (page 109), and "Debugging Allocations With MallocDebug" (page 72).

Watching File System Usage

In file system usage mode, Sampler watches for calls to the POSIX file system functions `access`, `creat`, `close`, `fcntl`, `flock`, `link`, `lstat`, `open`, `read`, `stat`, `truncate`, and `write`. As in allocation mode, Sampler records the function call stack, then combines the call information into a call tree to show all the code paths your program takes as it accesses the disk.

This functionality is extremely useful for understanding how your application accesses the disk. 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. From the call tree data, you can see the functions that perform reads, and you can avoid cases where you reread the same data in multiple points in the code. For example, you can discover how many times your program opens files. If the number of calls to `open` is higher than you expect, you can then examine the function call stacks to understand why your program calls `open` excessively.

This feature is similar to the `fs_usage` tool, which displays all the disk accesses being performed by a specific process; it lists the action, the time taken to complete, and some of the parameters passed to the function. Sampler improves on this functionality by presenting the data in a graphical browser, and recording the call tree to see where the function is being called. However, Sampler does not identify the parameters passed to the `open` call, as `fs_usage` does. Sampler also does not display the time taken by each operation to execute, as `fs_usage` does.See "fs_usage File System Access Analysis Tool" (page 97) for more details.

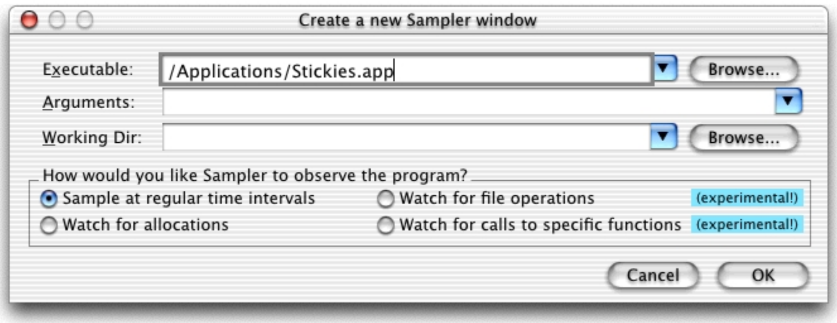
Watching for Calls to Specific Functions

This mode can tell you the number of times a specific function is called and the functions it is called from. Sampler allows you to specify a set of functions you care about, and see how many times and from where they are called. As with watching for allocations or watching for file operations, Sampler watches for calls to your specified functions and notes the call stack showing from where the function is called. When sampling is done, it merges all the call stacks into a call tree viewable with the Call Graph browser.

Using Sampler

After launching Sampler, choose either the New or Attach command from the File menu. New allows you to launch a new program. Attach allows you to start examining a program that is already running. Either command asks you to select an application to examine and a mode, as seen in Figure 6-1.

Figure 6-1 Creating a new Sampler document


Create a new Sampler window

Executable: /Applications/Stickies.app

Arguments:

Working Dir:

How would you like Sampler to observe the program?

[✓ Sample at regular time intervals]
Watch for allocations
Watch for file operations (experimental!)
Watch for calls to specific functions (experimental!)

Sampler's execution frequency window is used to discover the routines your program is spending most of its time in. Figure 6-2 shows this window.

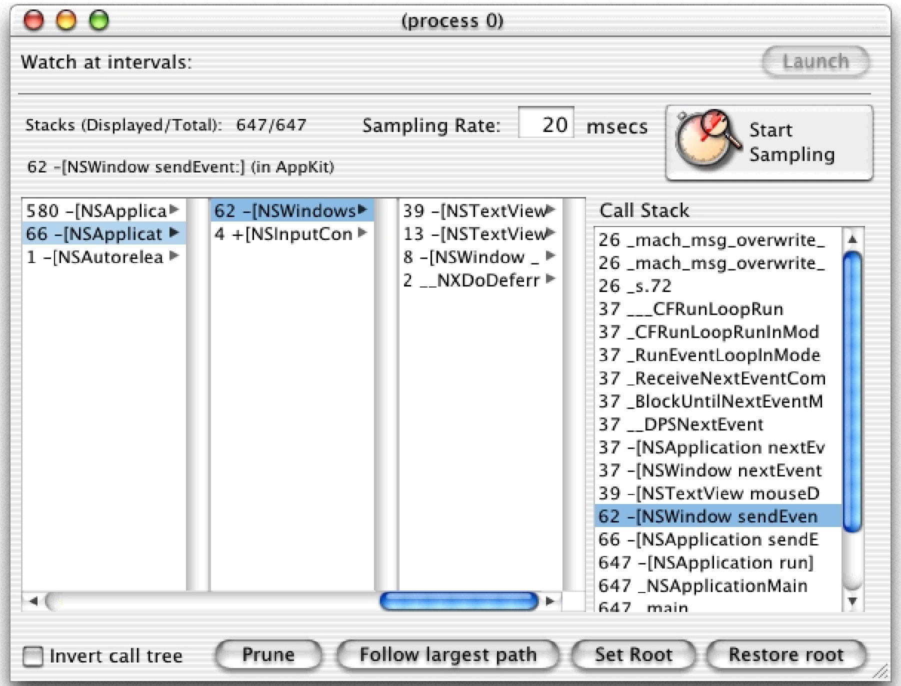
To profile a program, follow this procedure:

1. Enter the full path to the program in the Executable field, or click the Browse button to select the program in an Open dialog.

2. If you want to run the executable with command-line arguments, enter them in the Arguments field.
3. In the Working Directory field, enter the path where the program will read and write files and where Sampler places temporary files. By default, this is the `/tmp` directory. You can also click the Browse button and navigate to the working directory if it is not `/tmp`.
4. Click OK. The execution frequency window will now appear.
5. Enter the sampling frequency rate (in milliseconds) in the Sampling Rate field if you want a different sampling rate than is currently displayed (20 milliseconds is the default; generally you should keep this value between 10 and 50 milliseconds for optimum sampling).
6. If you want to sample the program as it launches, click the Launch & Sample button. The button title changes to Stop Sampling.

   If you want to samples the program during a certain activity, click the Launch button at the top-right corner of the window. The Launch & Sample button title changes to Start Sampling. Press the Start Sampling button to begin sampling.
7. Perform the activities you wish to sample. When you're done, click Stop Sampling.

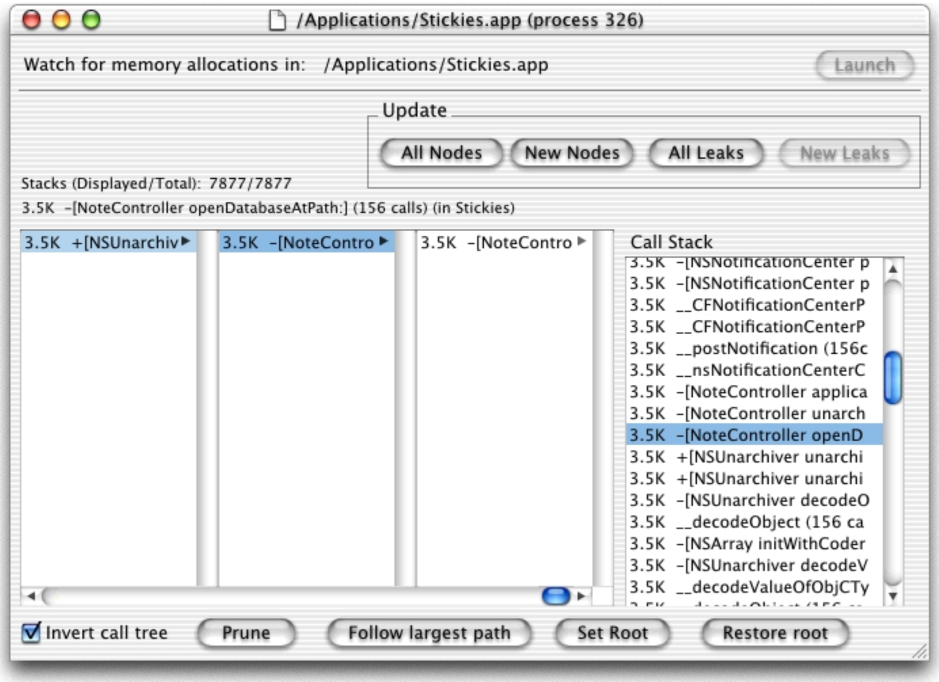
Figure 6-2 Sampler's execution frequency window



When you press the Stop Sampling button, Sampler shows the call-frequency information in the Call Graph browser and Call Stack list. For information related to using these views, see "Interpreting the Call Graph Browser and the Call Stack" (page 71).

Windows for the other modes are extremely similar to each other and quite similar to the execution frequency window, except that they are missing the Start/Stop Sampling button and the Sampling Rate field. Figure 6-3 shows the allocation window.

Figure 6-3 Sampler's allocation window



Stacks (Displayed/Total): 7877/7877

3.5K -[NoteController openDatabaseAtPath:] (156 calls) (in Stickies)

3.5K +[NSUnarchiv ▸ | 3.5K -[NoteContro ▸ | 3.5K -[NoteContro ▸

```
Call Stack
3.5K  -[NSNotificationCenter p
3.5K  -[NSNotificationCenter p
3.5K  __CFNotificationCenterP
3.5K  __CFNotificationCenterP
3.5K  __postNotification (156c
3.5K  __nsNotificationCenterC
3.5K  -[NoteController applica
3.5K  -[NoteController unarch
3.5K  -[NoteController openD
3.5K  +[NSUnarchiver unarchi
3.5K  +[NSUnarchiver unarchi
3.5K  -[NSUnarchiver decodeO
3.5K  __decodeObject (156 ca
3.5K  -[NSArray initWithCoder
3.5K  -[NSUnarchiver decodeV
3.5K  __decodeValueOfObjCTy
```

Instead of Start/Stop Sampling, there is a Launch button. When you press this button to launch the program, sampler collects function call stacks as specified by the selected mode. However, it does not immediately display anything. To display information in the Call Graph browser, click the All Nodes button. (In allocation mode, click the All Leaks button to see memory leaks). If you want to display function calls made over a period of program activity, exercise the program, then click New Nodes. (In allocation mode, to see if any new memory leaks occurred over that same period, click New Leaks.)

Note that you can attach a window to a program that is already running by choosing the Attach command from the File menu.

All Sampler windows have several fields and controls in common. These allow you to

- invert the call graph to start at the leaf nodes of the tree (the Invert Call Tree checkbox)
- generate a printable report (a textual representation of the call graph)
- exclude stacks from display to remove irrelevant entries or entries you've already examined (the Prune button)
- change the root of the call graph to a different function to allow you to focus on only the call tree you care about (the Follow Largest Path button)
- select the largest stack in terms of execution frequency or allocations

Interpreting the Call Graph Browser and the Call Stack

All of Sampler's windows use call stacks as the primary unit of analysis. The coalesced call stacks are presented as call graphs (or call trees) in the Call Graph browser. If you navigate through the data with the Call Graph browser, you can find out how many times a routine is called or how much memory is allocated during samples with similar call stacks.

The number next to a function represents the number of call stacks where that particular chain of functions was seen. For example:

- If `main` has a count of 100, then there were a hundred stack traces with `main` at the top. This number doesn't indicate how many times a function was called. Many scenarios are possible; for example, `main` called `foo` once, and `foo` called `malloc` one hundred times, or `main` called `foo` ten times, and `foo` called `malloc` ten times each time it was called.
- If you need to know how many times `foo` was called (and know it's being called only a small number of times), you can use a debugger to set a breakpoint on function `foo` and see how many times the breakpoint is hit. Alternately, you can use Sampler's user-defined function-watching mode (see "Watching for Calls to Specific Functions" (page 67)).
- If in the Call Graph browser you select `main`, then `createList`, then `addElement`, and it shows that 75 out of 100 samples were found in such a call stack, you know that 75% of your application's time is spent in calling `addElement` when it was called by `createList`. Thus you know not only where significant computation or allocations were done, but perhaps why the functions were called.

Limitations of Sampler

Sampler's results aren't comprehensive. Because Sampler is based on a statistical sampling when threads are preempted, it doesn't identify all calls or how many times a function was called, just those instances it has observed. Thus it might not report small, quickly executing functions, and longer running routines can appear more often. Leaf functions might appear in the output because the application had been preempted during those functions, not necessarily because the functions were executing for a long time. To improve the data Sampler generates, use longer sample sessions and shorter sample intervals. To see a complete picture of the function call graph, use the `gprof` code profiling tool ("gprof Code Profiler" (page 106)) instead.

Sampler may adversely affect the behavior of the computer during time-critical operations, or may be impossible to use if the target program runs on the entire screen. In these cases, you can use the related `sample` command-line tool, which requires fewer system resources and can be used from a telnet session. See the section "sample CPU and Memory Analysis Tool" (page 111) for information on using the `sample` tool.

The `sample` tool is also useful for diagnosing the reason your program has hung during normal execution.

Debugging Allocations With MallocDebug

The MallocDebug application is useful for inspecting how a program uses memory and for finding memory leaks in programs. MallocDebug shows the currently allocated blocks of memory, organized by the call stack at the time of allocation. No prior instrumentation of the program is necessary. MallocDebug can help you immediately identify how much allocated memory your application consumes, where that memory was allocated from, and which functions allocated large amounts of memory. It gathers data on Carbon, Core Foundation, and Cocoa allocations as well as `malloc` allocations. MallocDebug also analyzes the program to find allocated memory that is not referenced elsewhere in the program, thus helping you find leaks and track down exactly where the memory was allocated.

MallocDebug is intended for answering questions about your application's memory usage, such as these:

- How much memory is my program using at this point in its execution?
- How much memory gets created by this operation?
- Am I leaking memory? How much?
- What's the memory overhead of a specific call?
- Am I overrunning any buffers?

MallocDebug includes a number of refinements and supporting tools:

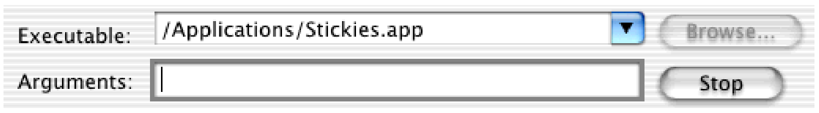
- It provides a hex-dump view for examining raw memory.
- It enables you to mark off any period of execution for analysis.
- It allows you to export performance data for detailed examination or for further analysis and refinement by command-line tools. The export feature gives you the freedom to look at or summarize the data in the form most relevant to your executable.

Using MallocDebug

After launching MallocDebug, the main window appears. There are three basic sections in the MallocDebug window. The **executable launcher** (Figure 6-4) occupies the top of the window, the **call stack browser** (Figure 6-5 (page 74)) is the main focus of the window, and the **memory buffer browser** (Figure 6-9 (page 79)) is at the bottom of the window.\

To use MallocDebug, you need to select and launch the application (or other executable) you wish to analyze.

Figure 6-4 Launching an executable in MallocDebug



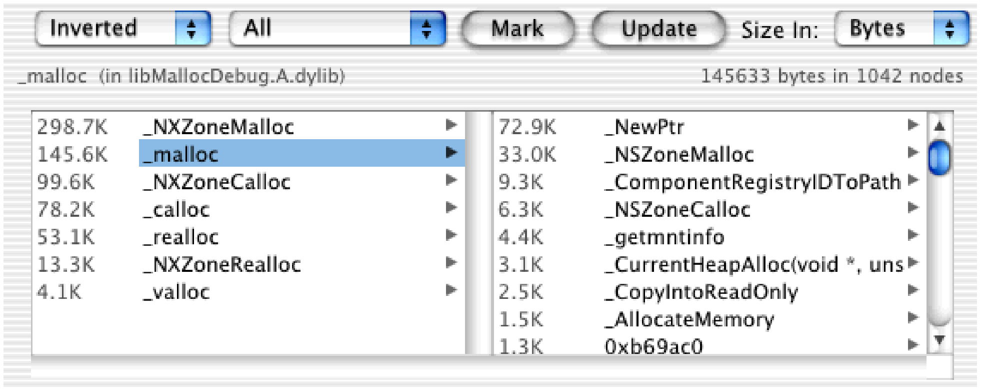
1. Enter the full path to the program in the Executable field, or click the Browse button and select the program using the file-system browser.
2. If you want to run the executable with command-line arguments, enter them in the Arguments field.
3. Click the Launch button.

MallocDebug launches the program and performs an initial query about memory usage. Further updates occur whenever you press the Update button.

The Call Stack Browser

The main focus of memory analysis in MallocDebug is the call stack browser, shown in Figure 6-5.

Figure 6-5 MallocDebug call stack browser

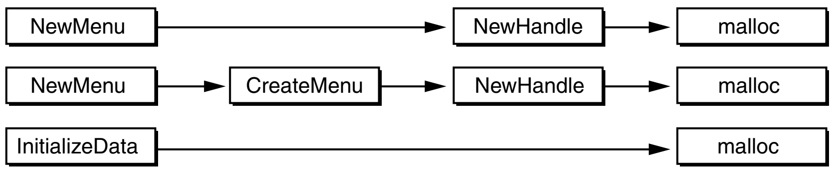


_malloc (in libMallocDebug.A.dylib) 145633 bytes in 1042 nodes

298.7K	_NXZoneMalloc	72.9K	_NewPtr
145.6K	_malloc	33.0K	_NSZoneMalloc
99.6K	_NXZoneCalloc	9.3K	_ComponentRegistryIDToPath
78.2K	_calloc	6.3K	_NSZoneCalloc
53.1K	_realloc	4.4K	_getmntinfo
13.3K	_NXZoneRealloc	3.1K	_CurrentHeapAlloc(void *, uns
4.1K	_valloc	2.5K	_CopyIntoReadOnly
		1.5K	_AllocateMemory
		1.3K	0xb69ac0

MallocDebug gathers data using its own `malloc` library. Over a period, as depicted in Figure 6-6, it identifies and gathers each function call stack leading to an allocation.

Figure 6-6 Function call stacks gathered at runtime

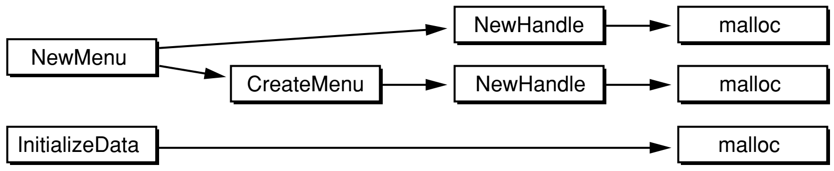


It coalesces these call stacks into a call tree by overlapping equivalent sequences of functions and presents this information in the call stack browser. Figure 6-7 illustrates the structure of a standard call tree.

The call stack browser has three viewing modes, which you choose in the left-most pop-up menu:

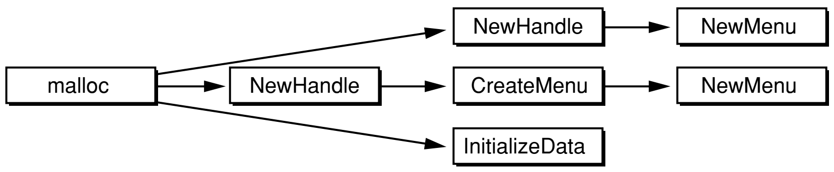
Standard mode presents each call stack hierarchically from the function at the top of the stack (for instance, `main`) to the function that performs the allocation: `malloc`, `calloc`, and so on. Each element of the browser shows the amount of memory that has been allocated in the call stack involving that method or function.

Figure 6-7 View of function call tree in standard mode



Inverted mode reverses the hierarchy of standard mode and shows the call tree from the allocation functions to the bottom of each stack. This mode is useful for highlighting the ways in which specific allocation functions are called. By seeing all the calls to `malloc` or `NewHandle` or the Core Foundation allocators, you can more easily detect wasteful patterns in lower-level libraries. Use inverted mode if you're working on a low-level framework or if you want to focus on how you're calling `malloc` in your own code. Figure 6-8 depicts the call stack in inverted mode.

Figure 6-8 View of function call tree in inverted mode



Flat mode shows memory usage for every method and function of an application in a single list, sorted by allocated amount. All of the instances of a function call are collapsed into one browser item corresponding to that function. A function's memory use includes the sum of all the allocations performed in that function and all allocations performed in functions that it calls. This allows you to see the total amount of memory allocated by every function, not just those at the top or bottom of the call stack.

The display mode pop-up menu (located to the right of the viewing-mode pop-up menu) affects the type of allocations that are displayed in the call stack browser. You have several options:

- **All**. Gives you the call trees for all currently allocated buffers in your application.
- **New**. Displays functions and methods in your application in which allocation has occurred since a specified point in time. The contents of the call stack browser thus provide an indication of the memory usage during a period of program execution. See "Taking a Snapshot of Memory Usage" (page 77) for the related procedure.
- **All Leaks, Definite Leaks, Possible Leaks**. These items display a call tree showing leaked memory blocks in your program. For further discussion of these display modes, see "Looking for Memory Leaks" (page 77).
- **Trashed**. Displays a call tree that shows allocated buffers in your application that have been written to incorrectly, either overrunning or underrunning the allocated buffer. The list of allocations, which gives you a more detailed picture of memory usage, indicates when memory is trashed. If the program has written past the end of a buffer, a right arrow (>) appears by the buffer. Similarly, if the application has written before the start of a buffer, a left arrow (<) appears by the buffer. For more on MallocDebug's memory-detail features, see "Analyzing Raw Memory" (page 79).

Taking a Snapshot of Memory Usage

When you launch a program with MallocDebug, you first see the allocation activity that goes on during launch time. Each time the Update button is pressed, MallocDebug shows memory usage at the current point in time. Often you want to measure the memory usage of your program during some other segment of program execution—for example, opening a document. And you want to exclude all other allocations from the measurement. To take this memory snapshot, complete the following steps:

1. Press the Mark button.
2. Exercise a portion of your program.
3. Select the New item from the second pop-up list.

MallocDebug shows the buffers allocated since the mark was set. Note that MallocDebug displays only the buffers that are still currently allocated, so you will see only those buffers allocated since the mark that haven't been freed.

Looking for Memory Leaks

Memory leaks are blocks of memory that have not been freed by the program, but that the program no longer references. Memory leaks waste both space and time. They waste space by filling up pages of memory, and they waste time by causing unnecessary paging activity. If a page contains both valid and leaked memory blocks, the leaked memory blocks occupy space that could be used to hold valid memory blocks. Another page must be allocated to hold the valid memory which could have used the space occupied by the leaked memory.

There are two ways an application can leak memory allocated by `malloc`. First, the application can allocate memory, embed it into data structures, then forget to free the memory when it is no longer needed. Because the memory is still referenced, MallocDebug cannot automatically detect this sort of leak. One way to find such problems is to watch for functions where the total memory allocated grows constantly. You can also use the Display New mode to see what memory is allocated, but not freed, during an operation. Choose New from the display mode pop-up menu and click the Mark button to set a starting point. Perform an operaion that should have no impact on memor, such as opening and closing a window. After the action, click the Update button. This shows you any memory allocated but not freed during the operation. Some of these allocations may be long-lived objects that happened to be created during the operation. However, if a significant amount of memory is left allocated, or if objects that should only exist while the window is open remain allocated, then you probably have memory leaks.

The second type of leak occurs when an application allocates memory, embeds it in a data structure, then gets rid of all pointers to the structure without deleting the memory. For example, a program could allocate a new object and assign its address to a pointer, then assign `0` or `NULL` to that pointer, trashing the pointer to the allocation. The block of memory is then left unreferenced. Because no pointers reference the allocated block, the memory is guaranteed not to be used and is obviously taking up space for no purpose. This sort of leak can be detected automatically by MallocDebug.

MallocDebug uses a conservative garbage detector for detecting unreferenced blocks of memory. When you choose the Leaks, Definite Leaks, or Possible Leaks pop-up menu items, MallocDebug searches through your program's memory for pointers to each block allocated by `malloc`. Any block that is not referenced is marked as a memory leak. By noting this wasted memory and changing your code to free the memory when it is no longer needed, you can reduce the memory footprint of your application.

The three modes of leak detection (all, definite, and possible) use slightly different criteria for identifying leaks.

- If you choose Definite Leaks, MallocDebug finds all blocks allocated by `malloc` where no pointers exist to any part of the buffer. Because this memory is not referenced within the application, it is certainly wasted.
- If you choose Possible Leaks, MallocDebug finds all blocks where no pointers exist to the beginning of the block, but pointers exist into the middle of the block. Pointers into the middle of the block could be random values or stale pointers, implying that the block is wasted. Alternatively, the application might only reference the block via pointers into the middle of the buffer. This is a common method for implementing objects in a procedural language such as C; the first few bytes of the buffer store type information, and clients pass around pointers into the middle of the buffer where the actual data starts. In such a case, the buffer is not leaked even though no pointers exist to the start of the buffer. Because MallocDebug cannot determine if an arbitrary value in memory is a pointer into the middle of the buffer or is just a random value, you may need to examine the allocations found by Possible Leaks to determine if they may truly be leaks, or if they are merely buffers that your code is referencing through pointer arithmetic.

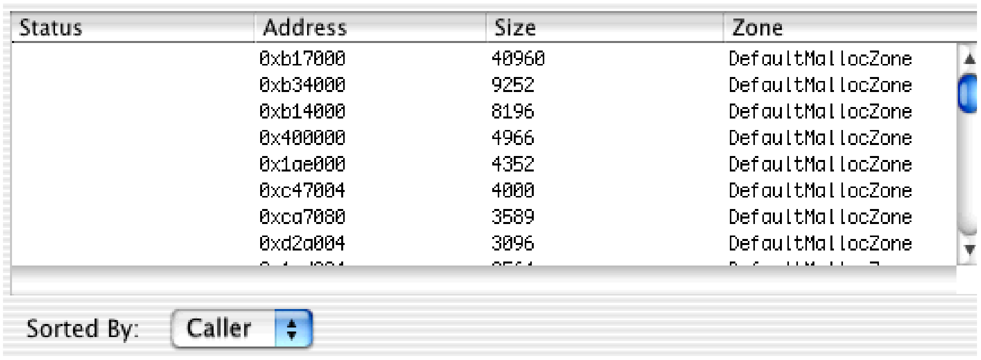
- If you choose All Leaks, MallocDebug shows all blocks allocated by `malloc` where no pointers exist to the beginning of the buffer. Its results are the union of the results from Definite Leaks and Possible Leaks.

MallocDebug's garbage collection algorithm might not detect some leaks. See "Limitations of MallocDebug" (page 82) for an explanation. Leak detection, however, does work correctly in the presence of Carbon handles. Carbon handles point at the allocated buffer, allowing the buffer to be moved in memory without the user's pointer becoming invalid. For a buffer created with `NewHandle` to be leaked, no pointers must exist to the handle; the pointer from the handle to the buffer is ignored.

Analyzing Raw Memory

When you select an allocation buffer (functions or methods) in the call stack browser, the memory buffer browser (as shown in Figure 6-9 (page 79)) might show one or more lines of data. Each line in this list represents a block of memory allocated in the currently selected function or in a function eventually called by that function. An inspector allows you to examine hexadecimal values in each buffer, including special byte patterns that MallocDebug places there to help identify bad memory usage.

Figure 6-9 MallocDebug memory buffer browser



Status	Address	Size	Zone
	0xb17000	40960	DefaultMallocZone
	0xb34000	9252	DefaultMallocZone
	0xb14000	8196	DefaultMallocZone
	0x400000	4966	DefaultMallocZone
	0x1ae000	4352	DefaultMallocZone
	0xc47004	4000	DefaultMallocZone
	0xca7080	3589	DefaultMallocZone
	0xd2a004	3096	DefaultMallocZone

Sorted By: Caller

Each line in the memory buffer browser gives the address of the buffer, its size in bytes, and the zone it was allocated from (for more information on zones, see "Using Multiple Malloc Zones" (page 23)). If the buffer is trashed, it also shows whether the bytes before the block are trashed (<) or the bytes after the block are trashed (>). Lines in the browser are sorted according to the item selected in the pop-up list below the browser:

- **Caller** sorts the memory buffers according to the functions or methods in the call graph making the memory allocation calls.
- **Time** sorts the memory buffers according by the order in which the allocation event occurred.
- **Zone** sorts the memory buffers by the zones from which they were allocated.
- **Size** sorts the memory buffers by byte size.

By double-clicking a line in the memory buffer browser, you bring up the Memory View window for that buffer of memory. This window allows you to inspect the contents of the buffer.

MallocDebug helps to catch problems such as memory leaks and trashed memory by writing certain hexadecimal patterns into the hex dump displayed in the memory buffer inspector. It overwrites freed memory with `0X55` and it guards against memory overruns and memory underruns by writing `0xDEADBEEF` and `0xBEEFDEAD`, respectively, at the beginning and end of each allocated buffer.

The memory buffer inspector can be particularly helpful for determining why an object is leaking. For example, if a string is being leaked, the text of the string might indicate where it was created. If an event structure is leaked, you might be able to identify the type of event from the contents of memory and thus find the corresponding event-handling code responsible for the leak.

Filtering Call Trees

You can temporarily hide items in the call tree by using the Prune button in the call stack browser or the Prune command in the Graph menu. When you select an item in the call tree and press the Prune Path button, the item is removed from the call tree. The Restore Path command in the Graph menu will restore all entries. You can also open the Filter Panel (via the Filter Panel command in the Graph menu) which shows all the items pruned from the graph. By selecting elements in the Filter panel and pressing the Restore button in the panel, you can selectively bring pruned elements back into the graph.

Other Techniques Using MallocDebug

MallocDebug has many other capabilities not discussed here:

- Attaching to a running process (the process must have been linked with the MallocDebug instrumented malloc library—either the statically linked version (`libMallocDebug.a`) or the shared library version (`libMallocDebug.A.dylib`).
- Streamlining the traversal of the call tree by traversing only the largest allocation buffers or by traversing the call tree until there are multiple buffers at any one point in the call tree.
- Examining zone usage.
- Categorizing allocations by creating mappings based on allocation zone, path through call tree, or text string. You can store the data gathered from such mappings and generate reports from it.

For information on using these features of MallocDebug, consult the Help menu while running MallocDebug.

Evaluating MallocDebug Problem Reports

Some of the reports that MallocDebug presents identify obvious problems—leaks, buffer overruns, and references to freed memory—that you should fix immediately. To improve your program's overall allocation behavior, you can use MallocDebug's detailed accounting of memory usage to explore the memory usage of your program. This can allow you to identify wasted memory allocations or strange allocation patterns, in turn allowing you to optimize your program's use of memory.

- Don't ignore small buffers, because they could be the root of a huge allocation graph.
- Look at allocation patterns during specific intervals of typical program use, especially where you suspect memory usage might be a problem.
- The inverted viewing mode for the call stack browser can sometimes yield results faster than the standard or flat modes, by displaying more interesting call stacks.
- Keep track of important statistics so you can compare a program's performance between tunings.

Limitations of MallocDebug

Some issues you may run into when running MallocDebug are described in the following section.

Allocated Memory Reporting

MallocDebug shows the *current* amount of allocated memory at a given point in a program's execution; it does not show the *total* amount of allocated memory. Memory that has been freed is not shown.

To see memory that your program has allocated and freed, use the `malloc_history` tool. See "malloc_history Allocation Debugging Tool" (page 115) for more information.

Crashing Under MallocDebug

If a program crashes under MallocDebug, a diagnostic message is printed to the console that explains why the program crashed. Listing 6-1 gives an example of MallocDebug's crash diagnostic message.

Listing 6-1 Diagnostic output from crashing under MallocDebug

```
MallocDebug: Target application attempted to read address 0x55555555, which can't be read.
MallocDebug: MallocDebug trashes freed memory with the value 0x55,
MallocDebug: strongly suggesting the application or a library is referencing
MallocDebug: memory it already freed.
MallocDebug: MallocDebug can't do anything about this, so the app's just going to have
to be terminated.
MallocDebug: libMallocDebug cannot help the application recover from this error,
MallocDebug: so we'll just have to shut down the application.
MallocDebug: *************************************************
MallocDebug: THIS IS A BUG IN THE PROGRAM BEING RUN UNDER MALLOC DEBUG,
MallocDebug: NOT A BUG IN MALLOC DEBUG!
MallocDebug: *************************************************
```

Usually a crash results from subtle memory problems, such as dereferencing freed memory or dereferencing pointers found outside an allocated buffer. Check suspected buffers of memory with the memory-buffer inspector (see "Analyzing Raw Memory" (page 79)). If your program is referencing memory at `0x55555555`, then it is referencing freed memory.

Important
Because the types of bugs detected by MallocDebug normally result in subtle, random crashes and data structure corruption, fixing these crashers should be your top priority.

Missing Leaks

MallocDebug's leaked memory analysis can sometimes miss leaks. Because the MallocDebug garbage detector cannot know which values in memory are pointers, it is possible that an integer has the same value as a pointer to a given node. In this case that node doesn't show up as a leak, even though it really is. (This is why the garbage detector is called *conservative*.)

Stale pointers could also exist in freed buffers, or in other leaked data structures. For example, two leaked data structures might both point to each other, but are not referenced by the rest of the program. MallocDebug currently cannot detect such leaks; circularly linked leaked structures are not detected; leaked tree-like structures only note the root node as leaked because it is the only node not referenced by another pointer. Because a single leaked buffer might be the start of a large data structure, a small leak could be the outward sign of a huge set of leaked data structures.

It's important to keep in mind that all leaks reported by MallocDebug are true leaks, and the problems noted above are very rare in practice.

Programs Calling setuid or setgid

For security reasons, the operating system does not allow programs running `setuid` (set the user id at execution) or `setgid` (set the group id at execution) to have new libraries—such as the heap debugging library used by MallocDebug—loaded into them. As a result, MallocDebug cannot display information about these programs if they are not run by that user or a member of that group.

If you want to examine a `setuid` or `setgid` program with MallocDebug, you have two options:

- Use MallocDebug on a copy of the program without the `setuid` or `setgid` permissions set. This approach may not work if the permissions are needed to access files normally not accessible by you.
- Run MallocDebug while logged in as the user who owns the file, or use the `su` tool to log in as another user. Note that you must run your program by calling the executable file directly in the latter case since the `open` tool runs the program as if it was launched by the user who logged in.

Simple Command-Line Programs

Simple programs run from the command line that do not use the System framework must statically link `malloc` routines into the executable. As a result, MallocDebug cannot insert its heap debugging library into the program at runtime. You can run MallocDebug on such a command-line program if you explicitly link the program with `/usr/lib/libMallocDebug.a`. The specially linked application should behave normally when run from the command line and can be launched or attached to from MallocDebug.

Setting Environment Variables

MallocDebug does not contain support for setting environment variables in the environment in which it executes your application. The easiest workaround for this limitation is to set those variables in MallocDebug's environment. To do this, set the environment variables on the command line (in Terminal), and then run MallocDebug by calling the executable directly (not using the `open` command). Applications run by MallocDebug inherit MallocDebug's environment, and thus the variables you set on the command line. See also "Quick Command-Line Primer" (page 63).

Observing Allocations With ObjectAlloc

ObjectAlloc allows you to observe memory allocation activity in an application. It retains a history of allocations and deallocations, allowing you to identify repetitive allocation behavior and overall allocation trends.

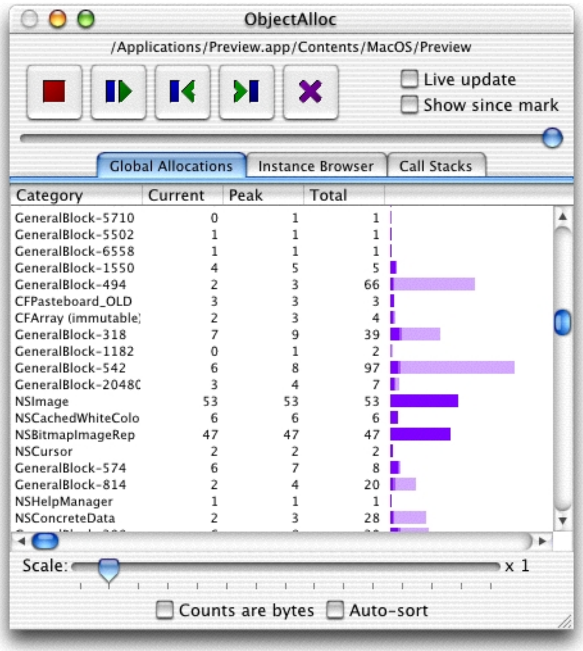
The information displayed by ObjectAlloc is recorded by an allocation statistics facility built into the Core Foundation framework. When this facility is active, every allocation and deallocation is recorded as it happens. For Objective-C objects, copy, retain, release, and autorelease are recorded.

Using ObjectAlloc

At launch, ObjectAlloc asks you for an application to inspect. Select an application and ObjectAlloc's window appears.

When you're ready to begin gathering data, click the Start button (the button with the green arrowhead or play symbol). (The Start button becomes the Stop button, which you can then use to stop gathering data.) ObjectAlloc launches the application and display memory allocations as they occur, as shown in Figure 6-10.

Figure 6-10 ObjectAlloc window



Category	Current	Peak	Total
GeneralBlock-5710	0	1	1
GeneralBlock-5502	1	1	1
GeneralBlock-6558	1	1	1
GeneralBlock-1550	4	5	5
GeneralBlock-494	2	3	66
CFPasteboard_OLD	3	3	3
CFArray (immutable)	2	3	4
GeneralBlock-318	7	9	39
GeneralBlock-1182	0	1	2
GeneralBlock-542	6	8	97
GeneralBlock-2048C	3	4	7
NSImage	53	53	53
NSCachedWhiteColo	6	6	6
NSBitmapImageRep	47	47	47
NSCursor	2	2	2
GeneralBlock-574	6	7	8
GeneralBlock-814	2	4	20
NSHelpManager	1	1	1
NSConcreteData	2	3	28

To view allocation data from the past, you must press either the Stop or Pause buttons to stop or pause collection of memory data. You can then use the Step Backward and Step Forward buttons to single-step through the allocation history, or you can drag the slider at the top of the window (the furthest-left position of the slider is launch time, the furthest-right position of the slider is the most recent data).

The Mark button sets a mark. You can use the “Show since mark” checkbox to toggle the display between events that have occurred since the mark was set and events that have occurred since the application was launched.

Due to the sheer quantity of information being processed, continuously updating ObjectAlloc’s display can noticeably slow the system down. If the “Live update” checkbox is not selected, the display is updated only when the ObjectAlloc window is activated or deactivated.

ObjectAlloc’s main window contains three tabs, Global Allocations, Instance Browser, and Call Stacks, each displaying complementary data.

Browsing Global Allocations

The Global Allocations tab contains a table with a listing of all memory blocks ever allocated in the application. The Category column shows the type of the memory block—either an Objective-C class name or a Core Foundation object name. If ObjectAlloc cannot deduce type information for the block, it uses “GeneralBlock-” followed by the size of the block (in bytes).The Current column shows the number of blocks of each type allocated but not (yet) released. The Peak column shows the largest number of blocks of each type that existed at any given time. The Total column shows the total number of blocks of each type that have been allocated, including blocks that have since been released.

The histogram bars to the right of the Total column are graphical representations of the three columns: the dark portion of the bar indicates the Current value, the middle portion of the bar is the additional number under Peak, and the complete length of the bar indicates the value under Total. The Scale slider controls the number of objects represented by each pixel in a bar (the actual number is shown to the right of the bar).

The “Counts are bytes” checkbox changes the numbers in the Current, Peak, and Total columns to reflect the number of bytes allocated (per object type) instead of the number of objects allocated (per object type).

Browsing Object Instances

The Instance Browser tab lists each type of block. Clicking a block type displays a list of all instances of that block. Clicking the address of a block instance displays a list of all allocation events pertaining to that block. If the block has not yet been freed, the contents of the block are displayed in the bottom pane of the ObjectAlloc window. Clicking an event brings up a textual description of the event, including a function call stack.

Browsing Call Stacks

The Call Stacks tab displays a table of each block type along with the number of instances (Count) and the number of bytes allocated to those instances (Size). The furthest-right column of this table contains the first item of a hierarchical function call stack. Clicking the disclosure triangle displays the next level of the function call stack. When the function call stack is open, it displays the location of each allocation.

The Descend Unique Path button discloses the selected function call stack to the deepest function shared by each instance's function call stack.

The Descend Max Path button discloses the selected function call stack to the deepest function in the stack.

Limitations of ObjectAlloc

The function call stacks collected by ObjectAlloc are not guaranteed to be accurate. In practice, this problem is rarely seen.

QuartzDebug for Debugging Graphics

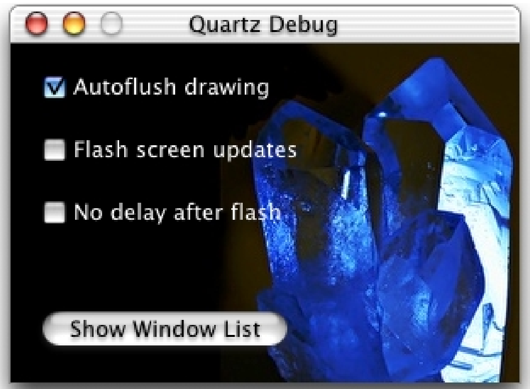
QuartzDebug is a debugging interface for the Quartz graphics system.

Upon launch, the QuartzDebug options window (Figure 6-11) appears. It contains three debugging checkboxes (all initially deselected) and a Show Window List button.

The "Autoflush drawing" checkbox flushes the contents of a CoreGraphics graphics context after each drawing operation.

When "Flash screen updates" is selected, regions of the screen that are about to be updated are painted yellow, followed by a brief pause, followed by the actual screen update. This allows you to see screen updates as they occur. The pause allows you to see the region in yellow; without it, the screen would be updated immediately, possibly faster than you can perceive it. To turn off the pause, select the No delay after flash option.

Figure 6-11 QuartzDebug options window



Quartz Debug

[✓] Autoflush drawing
[ ] Flash screen updates
[ ] No delay after flash

Use the Show Window List button to display a text window containing a static snapshot of the system-wide window list. The list identifies the owner of each window and the memory the window occupies. This is useful for understanding the impact of buffered windows on your application's memory footprint.

Listing 6-2 shows a sample window list and Table 6-1 explains the meaning of each column in the window list.

This list is a snapshot, not updated automatically. To update the snapshot, either press the Show Window List button again, or choose Show Window List from the File menu. Note that you will need to scroll the window list window down to see the new snapshot.

Listing 6-2 QuartzDebug window list

```
CID  WID kBytes  Type     Visible   Backing Shared   Fade Bps Level Rect                Name
==== === ======= ======== ========= ======= ======== ==== === ===== =====================
=====================
560f  6c   56.3  Buffered OffScreen  Meshed  Private 100% 32     20 { 159,  22, 213,  65}
QuartzDebug.app
560f  6b    8.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 853,  63,  17}
QuartzDebug.app
3803  63  160.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 742, 320, 128}
Clock.app
6223  5e  100.3I Buffered OffScreen  Opaque  Private 100% 32     20 {   0,   0,1152,  22}
TextEdit.app
3803  59    4.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 851,  19,  19}
Clock.app
6223  57    4.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 851,  19,  19}
TextEdit.app
6223  56 2052.3  Buffered  Obscured  Meshed  Private 100% 32      0 {  41, 174, 810, 631}
TextEdit.app
3403  52    4.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0,   9,   5}
Dock.app
3803  4f  100.3I Buffered OffScreen  Opaque  Private 100% 32     20 {   0,   0,1152,  22}
Clock.app
560f  4e 2068.8  Buffered  Obscured  Meshed  Private 100% 32      0 {  96, 100, 835, 606}
QuartzDebug.app
3103  2b  943.9  Buffered  Obscured  Meshed  Private 100% 32      0 { 491, 155, 543, 427}
Finder.app
560f  2a  100.3  Buffered  Obscured  Opaque  Private 100% 32     20 {   0,   0,1152,  22}
QuartzDebug.app
560f  29    4.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 851,  19,  19}
QuartzDebug.app
560f  28  298.1  Buffered  Obscured  Meshed  Private 100% 32      0 { 803,  64, 300, 222}
QuartzDebug.app
3403  27    4.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0,   9,   5}
Dock.app
3803  24  128.3  Buffered OffScreen  Meshed  Private 100% 32      0 {   0, 614, 128, 256}
Clock.app
3803  23   64.3  Buffered OffScreen  Opaque   Shared  58% 32     21 { 591, 290, 128, 128}
Clock.app
   0  22    4.3  Buffered  Obscured  Meshed  Private 100% 32    -20 {   0,  22,   1,  16}
```

```
3103  21  100.3I Buffered OffScreen  Opaque  Private 100% 32     20 {   0,   0,1152,  22}
Finder.app
3103  1f  943.9  Buffered  Obscured  Meshed  Private 100% 32      0 {  50,  86, 543, 427}
Finder.app
3403  1c  292.3  Buffered  Obscured  Meshed  Private 100% 32     10 {  76, 796,1000,  74}
Dock.app
3403  1b    4.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0,   9,   5}
Dock.app
3403  1a    4.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0,   9,   5}
Dock.app
3403  19   80.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0, 128, 128}
Dock.app
3403  18   80.3  Buffered  Obscured  Meshed   Shared 100% 32     11 {   0,   0, 128, 128}
Dock.app
3403  17   80.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0, 128, 128}
Dock.app
3403  16   80.3  Buffered  Obscured  Meshed  Private 100% 32     11 {   0,   0, 128, 128}
Dock.app
```

Table 6-1 QuartzDebug window list columns

Column	Description
`CID`	The connection ID of the window. Used internally by the window server. Typically, the connection ID is the same for all windows owned by a process.
`WID`	The ID of the window itself.
`kBytes`	The amount of memory occupied by the window buffer and other large data structures. Specified in kilobytes. The letter `I` is appended to the size if the buffer is invalid (in need of an update).
`Type`	`Buffered` windows are buffered in shared memory. All graphics operations are recorded in the backing buffer and drawn to screen by the window server as necessary. Only the portions of a `Retained` window that are obscured by other windows are saved in the buffer. This results in some memory savings, but disables translucency. Graphics operations in `NonRetained` windows are not recorded at all.

Table 6-1 QuartzDebug window list columns (continued)

Column	Description
`Visible`	The visibility of the window. `Obscured` windows are partially or completely covered by other windows. `Offscreen` windows are hidden from view. `Onscreen` windows are visible and not obscured by other windows in front of them.
`Backing`	Indicates the window buffer's image type. `Meshed` buffers are arrays of pixel quadlets, each individual quadlet containing the red, green, blue, and alpha (transparency) channel values for a pixel. `Planar` buffers are arrays of red, green, and blue triplets, with the alpha values for each pixel stored in a separate array. `Opaque` window buffers contain no alpha channel. Note that the window buffer includes the window's title bar and frame (or, in Carbon terms, "structure region").
`Shared`	`Private` windows can only be modified by the application specified in the `Name` column. `Shared` windows can be manipulated by multiple applications.
`Fade`	Opacity of the window. Opacity is separate from the window's alpha channel. Ranges from 0% to 100%, where 0% indicates a completely transparent window; 100% indicates a completely opaque window.
`Bps`	Depth of the window's buffer (the number of bits per pixel).
`Level`	The window level. Windows at higher levels can never be placed visually below windows at lower levels. Values from `LONG_MIN` + 1 to `LONG_MAX` - 16 are supported.
`Rect`	Screen-relative coordinates of the window (in pixels).
`Name`	The name of the application that owns the window.

top Process Examination Tool

Syntax: `top` [`-u`] [`-w`] [`-k`] [`-l` *count*] [`-s` *interval*] [`-e | -d`]`|` `-a`] [*num_procs*]

The `top` tool displays a periodically sampled set of statistics on system usage. It operates in various modes, but by default shows CPU utilization and memory usage for each process in the system. The default sampling interval is one second.

Listing 6-3 shows a typical statistical output.

Listing 6-3 Typical output of “top”

```
Processes:  36 total, 2 running, 34 sleeping... 81 threads
Load Avg:  0.24, 0.27, 0.23     CPU usage:  12.5% user, 87.5% sys, 0.0% idle
SharedLibs: num =   77, resident = 10.6M code, 1.11M data, 4.75M LinkEdit
MemRegions: num = 1207, resident = 16.4M + 4.94M private, 22.2M shared
PhysMem:  16.0M wired, 25.8M active, 48.9M inactive, 90.7M used, 37.2M free
VM:  476M + 39.8M   6494(6494) pageins, 0(0) pageouts

  PID COMMAND      %CPU   TIME       #TH  #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  318 top            0.0%  0:00.36   1     23     13   172K   232K   380K  1.31M
  316 zsh            0.0%  0:00.08   1     18     12   168K   516K   628K  1.67M
  315 Terminal       0.0%  0:02.25   4    112     50  1.32M  3.55M  4.88M  31.7M
  314 CPU Monito     0.0%  0:02.08   1     63     35   896K  1.34M  2.14M  27.9M
  313 Clock          0.0%  0:01.51   1     57     38  1.02M  2.01M  2.69M  29.0M
  312 Dock           0.0%  0:03.72   2     77     78  2.18M  2.28M  3.64M  30.0M
  311 Finder         0.0%  0:07.68   4     86    171  7.96M  9.15M  15.1M  52.1M
  308 pbs            0.0%  0:01.37   4     76     40   928K   684K  1.77M  15.4M
  285 loginwindow    0.0%  0:07.19   2     70     58  1.64M  1.93M  3.45M  29.6M
  282 cron           0.0%  0:00.00   1     11     14    88K   228K   116K  1.50M
  245 sshd           0.0%  0:02.48   1     10     15   176K   312K   356K  1.41M
  222 SecuritySe     0.0%  0:00.14   2     21     24   476K   828K  1.29M  3.95M
  209 automount      0.0%  0:00.03   2     13     20   336K   748K   324K  4.36M
  200 nfsiod         0.0%  0:00.00   1     10     12     4K   224K    52K  1.22M
  199 nfsiod         0.0%  0:00.00   1     10     12     4K   224K    52K  1.2
[...]
```

Interpreting the Output of top

The top tool displays periodically updated statistics on CPU usage, memory usage (in various categories), resource usage (such as threads and ports), and paging events. It has two modes, a CPU and memory utilization mode and a event-counting mode (the events being paging and low-level system calls). The former mode is the default. By adjusting the size of your window, you can change the number of processes and columns displayed.

In its header area, top displays periodically updated statistics on global state. This information includes load averages, total process and thread counts, and total memory, broken down into various categories such as private, shared, wired, and free. It also includes global information concerning the system frameworks.

Table 6-2 describes the columnar data that appears in the CPU and memory utilization mode using the -w parameter. See Table 6-4 (page 96) for a listing of all the parameters.

Table 6-2 "top -w" output: the wide CPU and memory utilization mode

Column	Description
`PID`	The BSD process ID
`COMMAND`	The name of the executable or application package. (Note that Code Fragment Manager applications are named after the native process that launches them, `LaunchCFMApp`.)
`%CPU`	The percentage of CPU cycles consumed during the interval on behalf of this process (both kernel and user space).
`TIME`	The amount of CPU time consumed by this process (*minute*:*seconds.hundreths*) since it was launched.
`#TH`	The number of threads owned by this process.
`#PRTS (delta)`	The number of Mach port objects owned by this process. The delta value, which is enabled by the `-w` parameter, is relative to the value first displayed when `top` was launched.
`#MREG`	The number of memory regions.
`VPRVT (delta)`	The private address space currently allocated (with `-w` parameter only).

Table 6-2 “top -w” output: the wide CPU and memory utilization mode (continued)

Column	Description
`RPRVT (delta)`	The resident private memory.
`RSHRD`	The resident shared memory (as represented by the resident page count of each memory object).
`RSIZE (delta)`	The total resident memory as real pages that this process currently has associated with it. Some may be shared by other processes. The delta value, which is enabled by the `-w` parameter, is relative to the previous sample.
`VSIZE (delta)`	The total address space currently allocated, including shared memory. The delta value, which is enabled by the `-w` parameter, is relative to the value first displayed when `top` was launched.

The `RPRVT` data (for resident private pages) is a measure of how much real memory is used for pages that are referenced only by this task. The pages could be in a shared library but there are no other links to anything in the shared library. The `RSHRD` column (for resident shared pages) shows the resident pages of all the shared mapped files or memory objects that are shared with other tasks.

Note: `top` does not provide a separate count of the number of pages in shared libraries that are mapped into the task.

`top` reports memory usage of windows in the “shared memory” category because window buffers are shared with the window server.

Table 6-3 shows the columns displayed in the event-counting mode, which is requested with either the `-e`, `-d`, or `-a` parameter.

Table 6-3 “top -d” output: event-counting mode

Column	Description
`PID`	The BSD process ID.
`COMMAND`	The name of the executable or application package. (Note that Code Fragment Manager applications are named after the native process that launches them, `LaunchCFMApp`.)
`%CPU`	The percentage of CPU cycles consumed during the interval on behalf of this process (both kernel and user space).
`TIME`	The amount of CPU time consumed by this process (*minute*:*seconds*.*hundreths*) since it was launched.
`PGINS`	The number of page-ins, requests for pages from a pager (each page-in represents a 4 kilobyte I/O operation).
`FAULTS`	The total number of page faults.
`COWS`	The number of faults that caused a page to be copied (generally caused by copy-on-write faults).
`MSENT`	The number of Mach messages sent by the process.
`MRCVD`	The number of Mach messages received by the process.
`BSD`	The number of BSD system calls made by the process.
`MACH`	The number of Mach system calls made by the process.
`CSW`	The number of context switches to the process (the number of times the process has been given time to run by the kernel’s scheduler).

top Command-Line Parameters

The `top` tool has a number of command-line parameters to control the values it displays and how it displays those values, as shown in Table 6-4.

Table 6-4 "top" command-line parameters

Parameter	Description
`-u`	Sorts processes by CPU usage, starting with the highest consumer first.
`-w`	Generates additional columns in the output producing a much wider display of data. The additional columns include `VPRVT` and `RALIAS` and the delta columns for `#PRTS`, `RSIZE`, and `VSIZE`. This parameter is ignored if event-counting mode is enabled (parameters `-e`, `-d`).
`-k`	Reports the memory object map for the kernel task (PID 0). This feature is optional because it is fairly expensive to traverse the object maps and the kernel task may have a huge number of entries. This parameter is ignored if event-counting mode is enabled (parameters `-e`, `-d`).
[`-l` *count*]	Enables logging mode, suitable for storing the output to a file. Normally, `top` modifies the statistics in place on the screen. With logging mode, `top` prints new statistics at every sampling interval. *count* is the number of times to output statistics. The default is 1, meaning only one sample is logged.
`-s` *interval*	Changes the sampling interval to *interval* seconds. By default, `top` updates its output at one-second intervals.
`-e`	Enables event-counting mode, where the counts reported are absolute counters for paging and messaging activity, starting from the time the process was initially launched. The `-w` and `-k` parameters are ignored when running in event-counting mode.

Table 6-4 “top” command-line parameters (continued)

Parameter	Description
`-d`	Enables delta event-counting mode, similar to the `-e` parameter, but with counts reported as deltas relative to the previous sample. The `-w` and `-k` parameters are ignored when running in event-counting delta mode.
`-a`	Shows accumulated events from the time `top` is run. Additionally, the `%CPU` value is calculated as the average over the time `top` is running, thus indicating where the CPU usage goes over the span of an activity.
num_procs	Limits the number of processes displayed (output lines) to this value. The display is limited to the number of rows that your Terminal window is currently displaying. If *num_procs* is not specified, the number of rows that can fit in your Terminal window is used. Unless the `-u` parameter is specified, processes are displayed in descending PID order, on the assumption that you will want to see the most recently-launched processes at the top of the list. `top` will adapt at runtime to Terminal window size changes, displaying a greater or lesser number of processes according to the new size.

fs_usage File System Access Analysis Tool

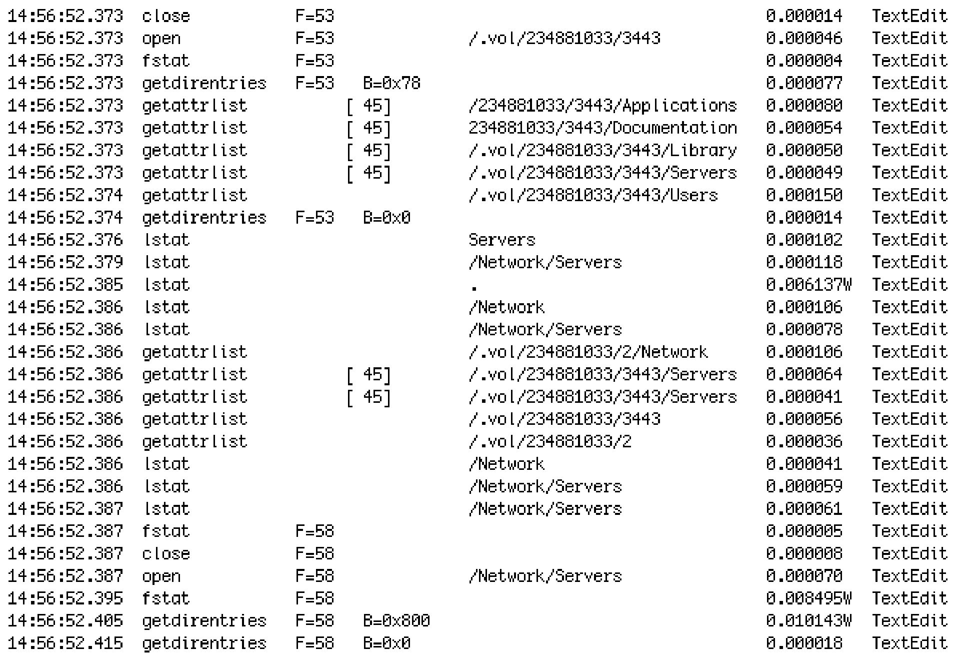
Syntax: `fs_usage` [`-e`] [`-w`] [*pid* | *proc_name* …]

`fs_usage` 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. You can limit the statistics, however, to include or exclude a specified list of processes.

Identifying the patterns used by your application to access files can suggest optimizations. For example, a slow-launching application might be trying to read from preferences stored on a network file server. Try watching for operations that take a long time to complete, and see which files are being accessed.

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 su` may be required instead).

Figure 6-12 Example of “fs_usage -w” output



```
14:56:52.373  close          F=53                                                0.000014   TextEdit
14:56:52.373  open           F=53              /.vol/234881033/3443              0.000046   TextEdit
14:56:52.373  fstat          F=53                                                0.000004   TextEdit
14:56:52.373  getdirentries  F=53   B=0x78                                       0.000077   TextEdit
14:56:52.373  getattrlist          [ 45]       /234881033/3443/Applications      0.000080   TextEdit
14:56:52.373  getattrlist          [ 45]       234881033/3443/Documentation      0.000054   TextEdit
14:56:52.373  getattrlist          [ 45]       /.vol/234881033/3443/Library      0.000050   TextEdit
14:56:52.373  getattrlist          [ 45]       /.vol/234881033/3443/Servers      0.000049   TextEdit
14:56:52.374  getattrlist                      /.vol/234881033/3443/Users        0.000150   TextEdit
14:56:52.374  getdirentries  F=53   B=0x0                                        0.000014   TextEdit
14:56:52.376  lstat                            Servers                           0.000102   TextEdit
14:56:52.379  lstat                            /Network/Servers                  0.000118   TextEdit
14:56:52.385  lstat                            .                                 0.006137W  TextEdit
14:56:52.386  lstat                            /Network                          0.000106   TextEdit
14:56:52.386  lstat                            /Network/Servers                  0.000078   TextEdit
14:56:52.386  getattrlist                      /.vol/234881033/2/Network         0.000106   TextEdit
14:56:52.386  getattrlist          [ 45]       /.vol/234881033/3443/Servers      0.000064   TextEdit
14:56:52.386  getattrlist          [ 45]       /.vol/234881033/3443/Servers      0.000041   TextEdit
14:56:52.386  getattrlist                      /.vol/234881033/3443              0.000056   TextEdit
14:56:52.386  getattrlist                      /.vol/234881033/2                 0.000036   TextEdit
14:56:52.386  lstat                            /Network                          0.000041   TextEdit
14:56:52.386  lstat                            /Network/Servers                  0.000059   TextEdit
14:56:52.387  lstat                            /Network/Servers                  0.000061   TextEdit
14:56:52.387  fstat          F=58                                                0.000005   TextEdit
14:56:52.387  close          F=58                                                0.000008   TextEdit
14:56:52.387  open           F=58              /Network/Servers                  0.000070   TextEdit
14:56:52.395  fstat          F=58                                                0.008495W  TextEdit
14:56:52.405  getdirentries  F=58   B=0x800                                      0.010143W  TextEdit
14:56:52.415  getdirentries  F=58   B=0x0                                        0.000018   TextEdit
```

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.

Interpreting fs_usage's Output

fs_usage generates a lot of data, continuously and with millisecond granularity. The output is not updated in place (as with, say, top); instead, each new line of data is appended to existing data. This aspect of the tool makes it especially worthwhile to redirect the output to a file and to run the tool during a specific activity.

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 6-5 describes these columns.

Table 6-5 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 or PAGE_IN	The name of the called file-system routine or a page-in.
3	A=0x45e2a000	Fault address. If the prior column is PAGE_IN, 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]	File offset specified to lseek or ftruncate. or The number of bytes requested by the call. or If the call results in an error, the error number (an errno value, see the header file errno.h) is displayed between brackets.

Table 6-5 Columns of “fs_usage” output (continued)

Column Number	Example	Description
5	`/Network`	The final 28 bytes of the pathname of the file accessed.
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`.)

fs_usage Command-Line Parameters

The `fs_usage` tool parameters are described in Table 6-6.

Table 6-6 “fs_usage” command-line parameters

Parameter	Description
`-e`	Generates output that excludes sampling of either the running `fs_usage` tool or the processes with the specified process IDs or command names. In the latter case, you must explicitly specify `fs_usage` to exclude it.
pid or *proc_name*	One or more process IDs or commands identifying the processes to be sampled. If there are multiple processes, separate each ID or command with a space. If the `-e` parameter is specified, the processes identified by process ID or command name are excluded from the sampling. Without the `-e` parameter, only the specified processes are sampled.
`-w`	Forces all columns to be displayed regardless of the current Terminal window size. This is useful when redirecting output to a file.

vmmap Memory Visualization Tool

Syntax: `vmmap` [`-d` *seconds*] *pid*

`vmmap` displays the virtual memory regions allocated in a specified process, helping a programmer understand how memory is being used, and what the purposes of memory at a given address are. For each region, `vmmap` describes the starting address, size of the region in kilobytes, read/write permissions for the page, sharing mode for the page, and the purpose of the pages.

The size of the virtual memory region represents the virtual memory pages reserved, but not necessarily allocated. For example, using the `vm_allocate` system call reserves the pages, but physical memory won't be allocated for the page until the memory is actually touched. A memory-mapped file may have a virtual memory page reserved, but the pages are not instantiated until a read or write happens. Thus, this size may not correctly describe the application's true memory usage.

The protection mode describes the access restrictions of the memory region. A memory region contains separate flags for reading, writing, and executing. Each virtual memory region has a current permission, and a maximum permission. In the line for a virtual memory region, the current permission is displayed first, the maximum permission second. For example, The first page of an application (starting at address `0x00000000`) permits neither reads, writes, or execution (`---`), ensuring that any reads or writes to address `0`, or dereferences of a `NULL` pointer immediately cause a bus error. Pages representing an executable always have the execute and read bits set (`r-x`). The current permissions usually do not permit writing to the region. However, the maximum permissions allow writing so that the debugger can request write access to a page to insert breakpoints. Permissions for executables appear as `r-x/rwx`, the first set of bits indicating the current permissions and the second set indicating the maximum..

The sharing mode describes whether pages are shared between processes and what happens when pages are modified. Private pages (`PRV`) are pages visible only to this process. They are allocated as they are written to and can be paged out to disk. Copy-on-write (`COW`) pages are shared by multiple processes (or shared by a single process in multiple locations). When the page is modified, the writing process then receives its own copy of the page. Empty (NUL) sharing implies that the page does not really exist in physical memory. Aliased (ALI) and shared (SHM) memory are shared between processes.

The sharing mode typically describes the general mode controlling the region. For example, as copy-on-write pages are modified, they become private to the application. Even with the private pages, the region is still copy-on-write until all pages become private. Once all pages are private, then the share mode would change to private.

The far left column names the purpose of the memory: __TEXT segment (which usually contains read-only code and data), __DATA segment (which usually contains data that is both readable and writable), and how the memory was allocated (via malloc, on the stack, and so forth). For regions loaded from binaries, the far right shows the library loaded into the memory.

Some lines in the output of vmmap describe submaps. A submap is a shared set of virtual memory page descriptions that the operating system can reuse between multiple processes. The memory between 0x70000000 and 0x80000000, for example, is a submap containing the most common dynamic libraries. Submaps minimize the operating system's memory usage by representing the virtual memory regions only once. Submaps can either be shared by all processes (machine-wide) or local to the process (process-only). If the contents of a machine-wide submap are changed—for example, the debugger makes a section of memory for a dynamic library writable so it can insert debugging traps—then the submap becomes local, and the kernel allocates memory to store the extra copy.

Listing 6-4 gives an example of vmmap output.

Listing 6-4 Typical output of vmmap

```
==== Non-writable regions for process 313
__PAGEZERO                        0 [   4K] ---/--- SM=NUL ...ts/MacOS/Clock
__TEXT                         1000 [  40K] r-x/rwx SM=COW ...ts/MacOS/Clock
__LINKEDIT                     e000 [   4K] r--/rwx SM=COW ...ts/MacOS/Clock
                              90000 [   4K] r--/r-- SM=SHM
                            340000 [3228K] r--/rwx SM=COW 00000100 00320...
                            789000 [3228K] r--/rwx SM=COW 00000100 00320...
Submap                 70000000-7fffffff         r--/r-- machine-wide submap
__TEXT                     70000000 [ 916K] r--/r-x SM=COW ...sions/B/System
```

```
__LINKEDIT                 700e5000 [ 264K] r--/r-- SM=COW ...sions/B/System
__TEXT                     70150000 [ 620K] r--/r-x SM=COW ...CoreFoundation
__LINKEDIT                 701eb000 [ 168K] r--/r-- SM=COW ...CoreFoundation
__TEXT                     70220000 [  12K] r--/r-x SM=COW ...ns/A/DesktopDB
__LINKEDIT                 70223000 [   4K] r--/r-- SM=COW ...ns/A/DesktopDB
__TEXT                     70230000 [ 692K] r--/r-x SM=COW ...cdsa_utilities
__LINKEDIT                 702dd000 [ 180K] r--/r-- SM=COW ...cdsa_utilities
__TEXT                     704a0000 [2860K] r--/r-x SM=COW ...s/A/CarbonCore
__LINKEDIT                 7076b000 [ 240K] r--/r-- SM=COW ...s/A/CarbonCore
[...data omitted...]
==== Writable regions for process 313
__DATA                         b000 [   4K] rw-/rwx SM=PRV ...ts/MacOS/Clock
__OBJC                         c000 [   8K] rw-/rwx SM=COW ...ts/MacOS/Clock
                                f000 [   4K] rw-/rwx SM=COW
                               10000 [ 252K] rw-/rwx SM=ZER
MALLOC_USED(DefaultMallocZone_   4f000 [  12K] rw-/rwx SM=PRV
MALLOC_USED(DefaultMallocZone_   52000 [  36K] rw-/rwx SM=COW
MALLOC_USED(DefaultMallocZone_   5b000 [  12K] rw-/rwx SM=PRV
MALLOC_USED(DefaultMallocZone_   5e000 [  28K] rw-/rwx SM=COW
MALLOC_USED(DefaultMallocZone_   65000 [   4K] rw-/rwx SM=ZER
MALLOC_USED(DefaultMallocZone_   66000 [   8K] rw-/rwx SM=COW
MALLOC_USED(DefaultMallocZone_   68000 [   4K] rw-/rwx SM=ZER
[...data omitted...]
Submap                 85fd1000-85feffff         r--/r-- process-only submap
__DATA                     85ff0000 [  32K] rw-/rw- SM=COW ...A/CoreGraphics
Submap                 85ff8000-86a0ffff         r--/r-- process-only submap
__DATA                     86a10000 [   4K] rw-/rw- SM=PRV ...libCGATS.dylib
Submap                 86a11000-86e4ffff         r--/r-- process-only submap
__DATA                     86e50000 [   8K] rw-/rw- SM=COW ...ns/A/PrintCore
Submap                 86e52000-8fffffff         r--/r-- process-only submap
                            a0002000 [  32K] rw-/rw- SM=SHM
                           a000b000 [16384K] rw-/rwx SM=NUL
                            a122e000 [  68K] rw-/rw- SM=SHM
                            a18c6000 [  68K] rw-/rw- SM=SHM
                            a18ea000 [ 204K] rw-/rw- SM=SHM
                            a1e73000 [ 168K] rw-/rw- SM=SHM
STACK[0](?)                 bff80000 [ 508K] rw-/rwx SM=PRV
STACK[0]                    bffff000 [   4K] rw-/rwx SM=PRV

==== Legend
SM=sharing mode:
    COW=copy_on_write PRV=private NUL=empty ALI=aliased
    SHM=shared ZER=zero_filled S/A=shared_alias

==== Summary for process 313
ReadOnly portion of Libraries: Total=27420KB resident=12416KB(45%)
swapped_out_or_unallocated=15004KB(55%)
Writable regions: Total=21632KB written=536KB(2%) resident=1916KB(9%)
swapped_out=0KB(0%) unallocated=19716KB(91%)
```

Interpreting vmmap's Output

The columns of `vmmap` output have no headings. Instead you can interpret the type of data in each column by its format. Table 6-7 describes these columns.

Table 6-7 Columns of vmmap output

Column Number	**Example**	**Description**

1	`__LINKEDIT`
	or
	`MALLOC`
	or
	`STACK`
		The name of a Mach-O segment.
		or
		Allocated memory (via `malloc`) off the heap.
		or
		Stack memory.

2	`(ObjC_0x46230)`	The zone used for allocation.
3	`4eee000`	Address of memory region.
4	`[ 124K]`	Size of the region, in kilobytes

Table 6-7 Columns of vmmap output (continued)

Column Number	Example	Description
5	`rw-/rwx`	Access permissions. Specifies protection/maximum protection for the region. The first bit is read protection, second is write protection, third is execute protection. The process does not have permission to read, write, or execute in a region for which those permissions are displayed as a dash (`-`).
6	`SM=PRV`	Sharing mode for the region, either `COW` (copy-on-write), `PRV` (private), `NUL` (empty), `ALI` (aliased), or `SHM` (shared).
7	`...ts/MacOS/Clock`	The end of the pathname identifying the executable mapped into this region of virtual memory. If the region is stack or heap memory, nothing is displayed in this column.

A few facts about the data in these columns:

- As noted earlier, the displayed size of a virtual memory region includes both allocated and unallocated pages. Consequently, the size might be larger than the actual memory footprint of a process, because the process could request a block of virtual memory but only access portions of it, and the virtual memory manager allocates only pages that have been accessed.
- Pages representing parts of a Mach-O executable file are usually not writable.
- The first page (`__PAGEZERO`, starting at address `0x00000000`) has no permissions set. This ensures that any reference to a `NULL` pointer immediately causes an error. The page just before the stack is similarly protected so that stack overflows will cause the app to crash immediately.

Delta Information

If you specify the `-d` parameter (plus an interval in seconds), `vmmap` takes two snapshots of virtual-memory usage—one at the beginning of a specified interval and the other at the end—and displays the differences. It shows three sets of differences:

- individual differences

- regions in the first snapshot that are not in the second
- regions in the second snapshot that are not in the first

vmmap Command-Line Parameters

Parameters to the `vmmap` tool are described in Table 6-8.

Table 6-8 "vmmap" command-line parameters

Parameter	Description
`-d` *seconds*	Specifies that `vmmap` display delta information (differences) between the start and end of the period defined by *seconds*. See "Delta Information" (page 105) for more information.
pid or *proc_name*	A process ID or name identifying the target process.

gprof Code Profiler

Syntax: `gprof` [`-a`] [`-b`] [`-s`] [`-S`] [`-z`] [`-x`] [(`-e` *name* | `-E` *name* | `-f` *name* | `-F` *name*) [*executable file* [*gmon.out* ...]]

Given profiling data collected at runtime, `gprof` produces an execution profile of a program. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (`gmon.out` by default), which is created by a program compiled and linked with the `-pg` parameter. The symbol table in the executable is correlated with the call graph profile file. If more than one profile file is specified, the `gprof` output shows the sum of the profile information in the given profile files.

`gprof` is useful for many purposes, including

- cases where the Sampler application doesn't work, such as command-line tools or applications that quit after a short period of time
- generating a full call graph to understand all the code that might be called in a given program
- optimization of code locality (for more information, see "Improving Locality of Reference" (page 126))

gprof Command-Line Parameters

Parameters to the `gprof` tool are described in Table 6-9.

Table 6-9 "gprof" command-line parameters

Parameter	Description
`-a`	Suppresses display of functions declared `static`. The static function's profiling information is absorbed by the function located immediately before it in the executable file.
`-b`	Suppresses display of the description of each field in the profile.
`-e` *name*	Suppresses the display of the profile entry for the routine *name* and all its descendants (unless they have other ancestors that aren't suppressed). Any number of `-e` arguments may be passed on a single command line, but only one routine name may be listed for each argument.
`-E` *name*	Suppresses the display of the profile entry for routine *name* (and its descendants) as `-e`, above, and also excludes the time spent in that routine (and its descendants) from the total and percentage time computations.
`-f` *name*	Displays the graph profile entry of only the specified routine name and its descendants. Any number of `-f` arguments may be passed on a single command line, but only one routine name may be listed for each argument.

Table 6-9 “gprof” command-line parameters (continued)

Parameter	Description
`-F` *name*	Displays the profile entry of only the routine name and its descendants (as `-f`, above) and also uses only the times of the displayed routines in total time and percentage computations. Any number of `-F` arguments may be passed on a single command line, but only one routine name may be listed for each argument. The `-F` parameter overrides the `-E` parameter.
`-s`	Produces a profile file called `gmon.sum`, which represents the sum of the profile information in all specified profile files. This summary profile file may be passed to subsequent executions of `gprof` to accumulate profile data across several runs of an executable.
`-S`	Produces four ordering files for use with the linker’s `-sectorder` parameter: `gmon.order` is an ordering based on a “closest is best” algorithm, `callf.order` is based on call frequency, `callo.order` is based on call order, and `time.order` is based on time spent in each routine.
`-z`	Displays routines that have zero usage (as indicated by call counts and accumulated time).
`-x`	Suppresses the generation of the `gmon.order` file when using the `-S` parameter.

Limitations of gprof .order Files

`.order` files contain only those functions that were called or sampled. For library functions to appear correctly in the order file, a `whatsloaded` file produced by the linker should exist in the working directory.

`-S` does not work with executables that have already been linked with an order file.

Production of the `gmon.order` file can take a long time—it can be suppressed with the `-x` parameter.

Filenames will be missing for

- files compiled without the `-g` parameter
- routines generated from assembly-language source

- executables that have had their debugging symbols removed (as with the `strip` tool)

leaks Memory Leak Finder

Syntax: `leaks` [`-cycles`] [`-nocontext`] [`-nostacks`] [`-exclude` *function*] *pid*

`leaks` examines a specified process for buffers allocated by `malloc` that are not referenced by the program. Such buffers waste memory; removing them can reduce swapping and memory usage.

For each leaked buffer allocated by `malloc`, `leaks` displays the address of the leaked memory and its size. If `leaks` can determine that the object is an instance of an Objective-C or Core Foundation object, it also specifies the name of the object. If the `-nocontext` option is not specified, `leaks` then displays a hexadecimal dump of the contents of the memory. If the `MallocStackLogging` environment variable is set, `leaks` finally displays a stack trace describing where the buffer was allocated.

For more information on setting environment variables, see "Quick Command-Line Primer" (page 63). For more information on `malloc`'s debugging options, see "Debugging Allocations With Malloc" (page 24).

leaks Command-Line Parameters

Parameters to the `leaks` tool are described in Table 6-10.

Table 6-10 "leaks" command-line parameters

Parameter	**Description**
pid	The BSD process identifier.
`-cycles`	Causes `leaks` to use a different leak-finding algorithm which may return different results. Results are displayed in a manner which helps you identify the source of a leak. It displays connected groups of function nodes, with the root node first.
`-nocontext`	Ccauses `leaks` to withhold a hex dump of the leaked memory. Although this information can be useful for recognizing the contents of the buffer and understanding why it might be leaked, it can also provide overwhelming detail.
`-exclude` *function*	Allows you to ignore leaks. Any allocations that were called from the function *function* are excluded from `leaks` output.
`-nostacks`	If the call stack information is being displayed, and you wish to suppress it, the `-nostacks` option causes `leaks` to turn off display of the call stack.

Limitations of leaks

Memory allocated with Carbon's `NewHandle` function and then leaked will not be noted by `leaks`. Thus, running `leaks` on a Carbon application shows only a subset of all possible leaks. The leaks reported are always true leaks.

MallocDebug (see "Debugging Allocations With MallocDebug" (page 72)) correctly finds leaked blocks allocated using `NewHandle`, and permits easier browsing of leaked blocks. However, MallocDebug does not detect leaks in circularly-linked structures or identify groups of leaked, connected nodes; the pointer analysis in `leaks` can correctly identify such leaks.

sample CPU and Memory Analysis Tool

Syntax: `sample` *pid duration* [*interval*] [`-mayDie`]

The `sample` tool analyzes a program's running behavior and then prints a report. It stops the program periodically, recording the function call stack each time, and then computes and displays the functions that were most frequently executing during the interval the program was examined. This information can help you locate functions consuming large chunks of CPU time. You can thereby find spots in your code where execution time or allocation size is more than you expect, and then improve your code to reduce running time or memory usage. You can also use `sample` to understand what's going on when a program appears hung or to optimize a program in a manner similar to profiling.

The `sample` tool is the command-line equivalent of the Sampler application. It is less invasive than the application and can be run from a `telnet` or `ssh` session. See the section "Code Profiling With Sampler" (page 64) for common background information, analytical approaches, and caveats.

Here is an example of running `sample` at the command line:

```
> sample Desktop 3 20
```

sample Command-Line Parameters

The parameters to the `sample` tool are described in Table 6-11.

Table 6-11 "sample" command-line parameters

Parameter	Description
pid	The process ID (PID) or the name of the process to be analyzed.
duration	The duration (in seconds) of the sampling session.
interval	The interval between samplings (in microseconds). The default is 10 microseconds.
`-mayDie`	Causes `sample` to read the symbol information in the process immediately. Use this parameter if the process is short-lived and likely to terminate soon.

To have `sample` properly analyze a process, you must start the process to be analyzed with a full path instead of a relative path.

Note that sample stops sampling as soon as the process terminates, so you can use the `-mayDie` option with a long *duration* to sample a short-lived process.

sc_usage System Call Statistics Tool

Syntax: `sc_usage` *pid* [`-c` *codefile*] [`-e`] [`-l`] [`-s` *interval*]

The `sc_usage` tool displays an ongoing sample of system call and page fault usage statistics for a given process. As new system calls are made, it adds to the list as they are generated by the application being watched. The counts displayed are the cumulative totals since `sc_usage` was launched and the delta changes for this sample period.

`sc_usage` shows a number of things other than the number of occurrences for each type of system call, among them the following:

- the amount of CPU time consumed
- the absolute time a process is waiting
- the pathnames for blocked system calls (per thread)
- 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-filled 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

Be aware that the `mach_msg_overwrite_trap` kernel routine will always be the system call with the greatest amount of CPU time used, since most processes accomplish interapplication communication by blocking on it.

Listing 6-5 shows some typical `sc_usage` output.

Listing 6-5 Typical sc_usage output

```
Finder     5 preemptions    7 context switches    1 thread     11:18:02
    0 faults 12 system calls

TYPENUMBER CPU_TIME   WAIT_TIME
-------------------------------------------------------------------------
System  Idle   0:07.345( 0:01.067)
System  Busy   0:00.825( 0:00.014)
Finder  Usermode  0:00.109

mach_msg_overwrite_trap   73(6) 0:06.299   0:01.866( 0:00.301) W
read  2    0:00.000
open  4    0:00.000
close 4    0:00.000
sigprocmask613(6) 0:00.003
fcntl 1    0:00.000
gettimeofday22    0:00.000
statfs      13    0:00.040
fstatfs      1    0:00.000
fstat 1    0:00.000
```

```
lstat52    0:00.018
getdirentries2    0:00.000
lseek 1    0:00.000
getattrlist 54    0:00.021
```

ps Process Listing Tool

The `ps` tool can show a simple snapshot of process resource usage. It is not generally recommended for use as a performance measurement tool. "top Process Examination Tool" (page 92) should be used instead.

pagestuff Mach-O File Page Analysis Tool

Syntax: `pagestuff` [`-a`][`-p`] *file* [*page number*…]

`pagestuff` displays information about the specified logical pages of a file conforming to the Mach-O executable format. For each specified page of code, symbols (function and static data structure names) are displayed. All pages in the `__TEXT, __text` section are displayed if no page numbers are given.

Table 6-12 "pagestuff" command-line parameters

Parameter	Description
`-a`	Displays all pages of the file. All other arguments are ignored.

Table 6-12 “pagestuff” command-line parameters

Parameter	Description
`-p`	Prints a list of the sections of the specified Mach-O file, with offsets and lengths. All other arguments are ignored.
file	Pathname of a Mach-O executable.
page number…	Numbers of one or more logical pages to display. Page numbers are separated by spaces.

heap Memory Block Listing Tool

Syntax: `heap` *pid*

`heap` lists memory blocks allocated with the `malloc` system call that have been allocated in the address space of the specified process. Any Objective-C objects found are also described, sorted by class.

MallocDebug (see “Debugging Allocations With MallocDebug” (page 72)) also provides information about memory blocks allocated with `malloc`, but `heap`, being a small command-line tool, is much less invasive (needing fewer system resources).

malloc_history Allocation Debugging Tool

Syntax: `malloc_history` *pid [address]* [`-all_by_size`] [`-all_by_count`]

`malloc_history` displays function call stacks that indicate the exact location of calls to allocation functions such as `malloc` and `free` by the specified process.

If an address is specified, `malloc_history` lists the function call stack for calls to `malloc` and `free` that allocate and free the buffer located at that address.

The `-all_by_size` and `-all_by_count` parameters list function call stacks for all allocations. Frequent allocations from the same point in the program (that is, the same call stack) are grouped together. and output presented either from largest allocations to smallest, or from most allocations to least.

The call stacks are collected by the standard Mac OS X allocation library when the environment variable `MallocStackLogging` is set to `1`. To record allocations by address, the environment variable `MallocStackLoggingNoCompact` must also be set to `1`.

For information on using environment variables, see "Quick Command-Line Primer" (page 63).

Be sure to also see ("Debugging Allocations With MallocDebug" (page 72)) for information on tracking memory leaks with MallocDebug.

Organizing Your Executable File

This chapter describes some of the features of the Mach-O executable format (the primary file format for executable code on Mac OS X) and offers strategies for reorganizing Mach-O executables for optimal layout in memory. The goal of this optimization is reducing paging I/O activity, directly resulting in improved runtime performance.

Overview of the Mach-O Executable Format

Mach-O is the executable format of native binaries compiled from C code (or object-oriented variants such as C++ and Objective-C) on Mac OS X. This format determines the order in which the code and data of the executable are read into memory. The ordering of code and data has implications for memory usage and paging activity and thus directly affects the performance of your program.

A Mach-O binary is organized into segments. Each segment contains one or more sections. Code or data of different types goes into each section. Segments always start on a page boundaries, but sections are not page-aligned. The size of a segment is the count of all the bytes of the sections it contains, which is then rounded up to the next virtual memory page boundary (4096 bytes, or 4 kilobytes). Thus, the minimum size of a segment is 4 kilobytes, and thereafter it is sized at 4 kilobyte increments.

The segments and sections of a Mach-O executable are named according to their intended use. The convention for segments is all-uppercase letters preceded by double underscores; the convention for sections is all-lowercase letters preceded by double underscores. There are several possible segments within a Mach-O executable, but only three of them are of interest in relation to performance: the `__TEXT` segment, the `__DATA` segment, and the `__LINKEDIT` segment.

The __TEXT Segment: Read Only

The `__TEXT` segment is a read-only area containing executable code and constant data. By convention, the compiler tools create every executable file with a read-only `__TEXT` segment. Because the segment is read-only, the kernel can map the `__TEXT` segment directly from the executable into memory just once. When the segment is mapped into memory, it can be shared among all processes interested in its contents. (This is primarily the case with frameworks and other shared libraries.) The read-only attribute also means that the pages that make up the `__TEXT` segment never have to be saved to backing store. If the kernel needs to free up physical memory, it can discard one or more `__TEXT` pages and re-read them from disk when they are needed.

Major sections in the `__TEXT` segment are summarized in Table 7-1.

Table 7-1 Major sections in the __TEXT segment

Section	Description
`__text`	The compiled machine code for the executable
`__const`	The general constant data for the executable
`__cstring`	Literal string constants (quoted strings in source code)
`__picsymbol_stub`	Position-independent code stub routines used by the dynamic linker (`dyld`).

The `__TEXT` segment can contain other sections, and some of the sections listed above might not appear in an executable.

The __DATA Segment: Read/Write

The __DATA segment contains the nonconstant data for an executable. This segment is both readable and writable. Because it is writable, the __DATA segment of a framework or other shared library is logically copied for each process linking with the library. When memory pages such as those making up the __DATA segment are readable and writable, the kernel marks them copy-on-write; therefore when a process writes to one of these pages, that process gets its own private copy of the page.

The __DATA segment has a number of sections, some of which are used only by the dynamic linker. Table 7-2 summarizes the sections of the __DATA segment.

Table 7-2 Major sections of the __DATA segment

Section	Description
`__data`	Initialized global variables (for example `int a = 1;` or `static int a = 1;`).
`__const`	Constant data needing relocation (for example, `char * const p = "foo";`).
`__bss`	Uninitialized static variables (for example, `static int a;`).
`__common`	Uninitialized external globals (for example, `int a;` outside function blocks).
`__dyld`	A placeholder section, used by the dynamic linker.
`__la_symbol_ptr`	"Lazy" symbol pointers. Symbol pointers for each undefined function called by the executable.
`__nl_symbol_ptr`	"Non lazy" symbol pointers. Symbol pointers for each undefined data symbol referenced by the executable.

Mach-O Performance Implications

The composition of the __TEXT and __DATA segments of a Mach-O executable file has a direct bearing on performance. The techniques and goals for optimizing these segments are different. However, they have as a common goal: greater efficiency in the use of memory.

Most of a typical Mach-O file consists of executable code, which occupies the `__TEXT`, `__text` section. As noted in "The __TEXT Segment: Read Only" (page 118), the `__TEXT` segment is read-only and is mapped directly to the executable file. Thus, if the kernel needs to reclaim the physical memory occupied by some `__text` pages, it does not have to save the pages to backing store and page them in later. It only needs to free up the memory and, when the code is later referenced, read it back in from disk. Although this is cheaper than swapping—because it involves one disk access instead of two—it can still be expensive, especially if many pages have to be recreated from disk.

One way to improve this situation is through improving your code's locality of reference through procedure reordering, as described in "Improving Locality of Reference" (page 126). This technique groups methods and functions together based on the order in which they are executed, how often they are called, and the frequency with which they call one another. If pages in the `__text` section group functions logically in this way, it is less likely they have to be freed and read back in multiple times. For example, if you put on one or two pages all functions of your application that perform launch-time initialization tasks, the pages do not have to be recreated after those initializations have occurred.

Unlike the `__TEXT` segment, the `__DATA` segment can be written to and thus the pages in the `__DATA` segment are not shareable. The nonconstant global variables in frameworks can have an impact on performance because each process that links with the framework gets its own copy of these variables. The main solution to this problem is to move as many of the nonconstant global variables as possible to the `__TEXT`,`__const` section by declaring them `const`. "Reducing the Number of Sharable Pages" (page 121) describes this and related techniques. This is not usually a problem for applications because the `__DATA` section in an application is not shared with other applications.

The compiler stores different types of nonconstant global data in different sections of the `__DATA` segment. These types of data are uninitialized static data and symbols consistent with the ANSI C notion of "tentative definition" that aren't declared `extern`. Uninitialized static data is in the `__bss` section of the `__DATA` segment. Tentative-definition symbols are in the `__common` section of the `__DATA` segment.

The ANSI C and C++ standards specify that the system must set these types of variables to zero. (Other types of uninitialized data are left uninitialized.) Because uninitialized static variables and tentative-definition symbols are stored in separate sections, the system needs to treat them differently. But when variables are in different sections, they are more likely to end up on different memory pages and thus can be swapped in and out separately, making your code run slower. The solutions to these problems, as described in "Reducing the Number of Sharable Pages" (page 121), is to consolidate the nonconstant global data in one section of the `__DATA` segment.

Reducing the Number of Sharable Pages

Although limiting your nonconstant data is a performance gain for all types of software, it is especially beneficial to frameworks. As noted in "Overview of the Mach-O Executable Format" (page 117), the data in the `__DATA` segment of Mach-O binaries is writable and thus shareable (via copy-on-write). Being shareable in a framework's dynamic shared library means that nonconstant global data can potentially be replicated among all processes linking with the framework. This section describes the steps you can take to remedy this situation.

Declaring Data as const

The easiest way to make the `__DATA` segment smaller is to move as much data as possible into the `__TEXT` segment by declaring it `const`. Most of the time, it's easy to mark data as constant. For example, if you're never going to modify the elements in an array, you should change its declaration from this:

```
int fibonacci_table[8]; = {1, 1, 2, 3, 5, 8, 13, 21};
```

to this:

```
const int fibonacci_table[8]; = {1, 1, 2, 3, 5, 8, 13, 21};
```

Remember to mark pointers as constant (when appropriate). In this example, the strings `"a"` and `"b"` are constant, but the array pointer `foo` is not:

```
static const char *foo[] = {"a", "b"};
foo[1] = "c";        // NOT OK: foo[1] is constant.
foo = {"c", "d"};    // OK:     foo is not constant.
```

You need to add the `const` keyword to the pointer to make the pointer constant. Here, both the array and its contents are constant:

```
static const char *const foo[] = {"a", "b"};
foo[1] = "c";       // NOT OK: foo[1] is constant.
foo = {"c", "d"};   // NOT OK: foo is now constant.
```

Sometimes you may want to rewrite your code to separate out the constant data. This example contains an array of structures in which only one field is declared `const`. Because the entire array isn't declared `const`, it is stored in the `__DATA` segment.

```
struct {
    const char *imageName;
    NSImage *image;
} images[100] = {
    {"FooImage", nil},
    // ...
    // and so on
}
```

To store as much of this data as possible in the `__TEXT` segment, create two parallel arrays, one marked constant and one not:

```
const char *const imageNames[100] = { "FooImage", /* . . . */ };
NSImage *imageInstances[100] = { nil, /* . . . */ };
```

If an uninitialized data item contains pointers, the compiler can't store the item in the `__TEXT` segment. Strings end up in the `__TEXT` segment's `__cstring` section but the rest of the data item, including the pointers to the strings, ends up in the `__DATA` segment's `const` section. In this example, `daytimeTable` would end up split between the `__TEXT` and `__DATA` segments, even though it's constant:

```
struct daytime {
    const int value;
    const char *const name;
};

const struct daytime daytimeTable[]; = {
    {1, "dawn"},
    {2, "day"},
    {3, "dusk"},
    {4, "night"}
};
```

To place the whole array in the __TEXT segment, you must rewrite this structure so it uses a fixed-size char array instead of a string pointer, like this:

```
struct daytime {
    const int value;
    const char name[6];
};

const struct daytime daytimeTable[] = {
    {1, {‘d', 'a', 'w', 'n', '\0'}},
    {2, {'d', 'a', 'y', '\0'}},
    {3, {'d', 'u', 's', 'k', '\0'}},
    {4, {'n', 'i', 'g', 'h', 't', '\0'}}
};
```

Unfortunately, there's no good solution if the strings are of widely varying sizes, because this solution would leave a lot of unused space.

The array is split onto two segments because the compiler always stores constant strings in the __TEXT segment's __cstring section. If the compiler stored the rest of the array in the __DATA segment's __data section, it's possible that the strings and the pointers to the strings would end up on different pages. If that happened, the system would have to update the pointers to the strings with the new addresses, and it can't do that if the pointers are in the __TEXT segment, because the __TEXT segment is marked read-only. So the pointers to the strings, and the rest of the array along with it, must be stored in the __DATA segment. It's placed in the const section, which is reserved for data declared const that couldn't be placed in the __TEXT segment.

Initializing Static Data

As pointed out in "Overview of the Mach-O Executable Format" (page 117), the compiler stores uninitialized static data in the __bss section of the __DATA segment. Spreading an executable's nonconstant global data across several sections makes it more likely that this data will be stored on different memory pages and thus will have to be swapped in and out separately.

The problem of uninitialized static data is easy to solve. Just change this:

```
static int x;
static short conv_table[128];
```

To this:

```
static int x = 0;
static short conv_table[128] = {0};
```

By initializing the static variables, you cause them to move to the `__DATA`, `__data` section in the Mach-O executable.

Important
You should follow this tactic only if you have a *small* (less than a page) amount of static data. Initialized data must be read from disk the first time the data is accessed. If the `__DATA`, `__data` page on which the data is stored is already in memory, there is no additional read required. If, by switching from the `__bss` section to the `__data` section, you add another page to the `__data` section, you've added the expense of another read from disk.

Avoiding Tentative-Definition Symbols

The compiler puts duplicate symbols in another place in the `__DATA` segment: the `__common` section (see "Overview of the Mach-O Executable Format" (page 117)). The problem here is the same as with uninitialized static variables. If an executable's nonconstant global data is distributed among several sections, it is more likely that this data will be on different memory pages; consequently, the pages may have to be swapped in and out separately. The goal for the `__common` section is the same as that for the `__bss` section: to eliminate it from your executable if you have a small amount of it.

A common source of tentative-definition symbols is definitions (that is, implementations) of symbols in header files. Typically headers contain declarations, some of which require definitions found in an implementation file. But definitions appearing in header files can result in that code or data appearing in every implementation file that includes the header file. The solution to this problem is to ensure that header files as much as possible contain only declarations, not definitions.

Header files can also contain explicit definitions of symbols, such as definitions of constants requiring construction. These definitions are compiled into static instances of the symbol in every object file. To avoid this, declare these symbols in the header file with the `extern` keyword and initialize them in an implementation file.

You can also get tentative-definition symbols when you accidentally import the same header file twice or when you use the same name for variables in two or more header files. If the variables with the same name are meant to be the same variable, declare one of them `extern`. Otherwise, give them different names.

Analyzing Mach-O Executables

You have several tools at your disposal to find out how much memory your nonconstant data is occupying. These tools can report on various aspects of data usage.

While your application or framework is running, use `pagestuff` and `size -m` to see how big your various data sections are. Some things to look for:

- To find executables with lots of nonconstant data, check for files with large `__data` sections in the `__DATA` segment. To narrow down the largest source of this data, you can use the `size` command on each object file making up the executable until you find the offending one.
- Check for variables and symbols in the `__bss` and `__common` sections that can be removed or moved to the `__data` section.
- To locate data that, although declared constant, the compiler can't treat as constant, check for executables or object files with a `__const` section in the `__DATA` segment.

Some of the bigger consumers of memory in the `__DATA` segment are fixed-size global arrays initialized but not declared `const`. You can sometimes find these tables by searching your source code for "`[] = {`".

You can also let the compiler help you find where arrays can be made constant. Put `const` in front of all the initialized arrays you suspect might be read-only and recompile. If an array is not truly read-only, it will not compile. Remove the offending `const` and retry.

Improving Locality of Reference

One of the most important improvements you can make to your application's performance is to reduce the number of virtual memory pages that might have to be cleared from memory and later read back in (these pages are usually referred to as the **working set**). The process of reducing the working set is called **scatter loading** or improving locality of reference.

Scatter loading places the blocks of code for individual methods or functions in an optimized order, independent of the source file they came from or their position in the source file. It allows the kernel to keep an active application's most frequently referenced executable pages in memory. On average, this can speed up an application's execution time by about 30 percent overall.

You should generally wait until very late in the development cycle to scatter load your application, since code tends to get moved around during development, which tends to invalidate prior profiling results.

Reordering the __text Section

As described in "Overview of the Mach-O Executable Format" (page 117), the `__TEXT` segment holds the actual code and other read-only portions of your program. The compiler tools, by convention, place procedures from your Mach-O object files (with extension `.o`) in the `__text` section of the `__TEXT` segment.

As your program runs, pages from the `__text` section are loaded into memory on demand, as routines on these pages are used. Code is linked into the `__text` section in the order in which it appears in the source file, and source files are linked in the order in which they are listed on the linker command line (or in the order specifiable in Project Builder). Thus, code from the first object file is linked from start to finish, followed by code from the second and subsequent files.

However, this is rarely the optimal order. For example, say that certain methods or functions in your code are invoked repeatedly, while others are seldom used. Reordering the procedures to place frequently used code at the beginning of the `__text` section minimizes the average number of pages your application uses and thereby reduces paging activity.

As another example, say that all the objects defined by your code are initialized at the same time. Because the initialization routine for each class is defined in a separate source file, the initialization code is ordinarily distributed across the `__text` section. By contiguously reordering initialization code for all classes, you reduce the number of pages that need to be read in, enhancing initialization performance. The application requires just the small number of pages containing initialization code, rather than a larger number of pages, each containing a small piece of initialization code.

Simple Procedure Reordering

Depending on the size and complexity of your application, you can pursue a strategy for ordering code that best improves the performance of your executable. Like most performance tuning, the more time you spend measuring and retuning your procedure order, the more memory you save. You can easily obtain a good first-cut ordering by running your application and sorting the routines by call frequency. The steps for this strategy are as follows:

1. Build a profile version of your application. This step generates an executable containing symbols used in the profiling and reordering procedures.
2. Run and use the application to create a set of profile data. Perform a series of functional tests, or have someone use the program for a test period.

   Important
   For best results, focus on the most typical usage pattern. Avoid using all the features of the application or the profile data might become diluted. For example, focus on launch time and the time to activate and deactivate your main window. Do not bring up ancillary windows.
3. Create order files. Order files list procedures in optimized order. The linker uses order files to reorder procedures in the executable.
4. Run the linker using the order files. This creates an executable with procedures linked into the `__text` section as specified in the order file.

These steps for this basic ordering are described in detail in the following sections.

Step 1: Compile and Link the Source Files

Compile and link your program using the `-pg` and `-g` options. The `-pg` option adds hooks for profiling the application and for listing procedure calls in the order file (next step). The `-g` option creates symbol tables with source-file references for use by the debugger; this option is used by the `gprof -S` option to add source file names to the order file (step 3).

If you want to reorder static library procedures along with those in object files, use the linker's `-whatsloaded` option to create a file of all loaded procedures in the project directory. The section "Creating a Default Order File" (page 132) describes this option.

To setup a Project Builder project for profiling, add a new "Profiling" build style, as detailed in these steps:

1. Click the Targets tab of the project window.
2. From the Project menu, choose New Build Style. The new build style will appear in the Build Styles pane.
3. Name the new build style Profiling and click it. An empty Build Settings table appears in the lower-right pane of the project window.
4. Click on the white area inside the table, and press Return. A new build setting value appears.
5. Name the build setting `OTHER_LDFLAGS` and set its value (double-click the value column) to `-pg -g`.
6. Create an identical build setting named `OTHER_CFLAGS` and set its value to `-pg -g`.
7. Click the furthest-left column of the Profiling build style item in the Build Styles pane. The Profiling build style should now have a checkmark in that column.
8. Use the "clean" button at the top of the Project Builder window to remove previous build results.

If your program contains assembly-language code, you might need to take additional steps to set up for profiling. See the section "Reordering Assembly Language Code" (page 139) for more information.

Step 2: Run and Use the Program

During program use, the profiling runtime records a profile of each routine called. The resultant profiling data is placed in a file named `gmon.out` when the program exits. Note that each time you quit the program, a new `gmon.out` file is created, overwriting the old file. If you want to keep the profile, it's a good idea to rename it before restarting the program for the next profiling run.

The simplest way to profile a program is to exercise it through a test suite, or to have someone use the program for a few consecutive days. These techniques generate large sets of profile data that you can then use to generate a procedure ordering file. In more advanced strategies, you might profile particular operations to accumulate many profile data sets, then generate a number of procedure order files from these sets, and then combine those order files into a final order file.

Step 3: Run gprof to Create Order Files

An order file contains an ordered sequence of lines, each line consisting of a source file name and a symbol name, separated by a colon with no other white space. Each line represents a block to be placed in a section of the executable. If you modify the file by hand, you must follow this format exactly so the linker can process the file. If the object file *name*:*symbol* name pair is not exactly the name seen by the linker, it tries its best to match up the names with the objects being linked.

The lines in an order file for procedure reordering consist of an object filename and procedure name (function, method, or other symbol). The sequence in which procedures are listed in the order file represents the sequence in which they are linked into the `__text` section of the executable.

To create an order file from the profiling data generated by using a program, run `gprof` using the `-S` option (see the man page for `gprof (1)`). For example,

```
gprof -S MyApp.profile/MyApp gmon.out
```

The -S option produces four mutually exclusive order files:

gmon.order	Ordering based on a "closest is best" analysis of the profiling call graph. Calls that call each other frequently are placed close together.
callf.order	Routines sorted by the number of calls made to each routine, largest numbers first.
callo.order	Routines sorted by the order in which they are called.
time.order	Routines sorted by the amount of CPU time spent, largest times first.

You should try using each of these files to see which provides the largest performance improvement, if any. See "Using pagestuff to Examine Pages on Disk" (page 133) for a discussion of how to measure the results of the ordering.

These order files contain only those procedures used during profiling. The linker keeps track of missing procedures and links them in their default order after those listed in the order file. Static names for library functions are generated in the order file only if the project directory contains a file generated by the linker's -whatsloaded option; see "Creating a Default Order File" (page 132) for details.

The order file omits the names of files not compiled with the -g option, assembly files, and stripped executable files. If your order file contains such references, you can either edit the file to add the filenames or delete the references so the procedures can be linked in default order.

The gprof -S option doesn't work with executables that have already been linked using an order file.

To preserve the order of routines in a particular object file, use the special symbol .section_all. For example, if the object file foo.o comes from assembly source and you want to link all of the routines without reordering them, delete any existing references to foo.o and insert the following line in the order file:

```
foo.o:.section_all
```

This option is useful for object files compiled from assembly source, or for which you don't have the source.

For more information about gprof, see the section "gprof Code Profiler" (page 106).

Step 4: Link the program with the order file

Once you've generated an order file, you can link the program using the `-sectorder` and `-e start` options:

```
cc -o outputFile inputFile.o … -sectorder __TEXT __text orderFile -e start
```

To use an order file with a Project Builder project, either add a build setting called `SECTORDER_FLAGS` to the Deployment build style, or set the `SECTORDER_FLAGS` variable in the Expert Build Settings table of the Build Settings tab of the Target Settings pane. In either case, the value of the setting should be `-sectorder __TEXT __text` *orderFile*.

If any *inputFile* is a library rather than an object file, you may need to edit the order file before linking to replace all references to the object file with references to the appropriate library file. Again, the linker does its best to match names in the order file with the sources it is editing.

With these options, the linker constructs the executable file *outputFile* so the contents of the `__TEXT` segment's `__text` section are constructed from blocks of the input files' `__text` sections. The linker arranges the routines in the input files in the order listed in *orderFile*.

As the linker processes the order file, it places the procedures whose object-file and symbol-name pairs aren't listed in the order file into the `__text` section of *outputFile*. It links these symbols in the default order. Object-file and symbol-name pairs listed more than once always generate a warning, and the first occurrence of the pair is used.

By default, the linker prints a summary of the number of symbol names in the linked objects that are not in the order file, the number of symbol names in the order file that are not in the linked objects, and the number of symbol names it tried to match that were ambiguous. To request a detailed listing of these symbols, use the `-sectorder_detail` option.

The linker's `-e start` option preserves the executable's entry point. The symbol `start` (note the lack of a leading "_") is defined in the C runtime shared library `/usr/bin/crt1.o`; it represents the first text address in your program when you link normally. When you reorder your procedures, you have to use this option to fix the entry point. Another way to do this is to make the line `/usr/lib/crt1.o:start` or `/usr/lib/crt1.o:section_all` the first line of your order file.

Procedure Reordering for Large Programs

For many programs, the ordering generated by the steps just described brings a substantial improvement over unordered procedures. For a simple application with few features, such an ordering represents most of the gains to be had by procedure reordering. However, larger applications and other large programs often benefit greatly from additional analysis. While the order files based on call frequency or the call graph are a good start, you can use your knowledge of the structure of your application to further reduce the virtual-memory working set.

Creating a Default Order File

If you want to reorder an application's procedures using techniques other than those described above, you may want to skip the profiling steps and just start with a default order file that lists all the routines of your application. Once you have a list of the routines in suitable form, you can then rearrange the entries by hand or by using a sorting technique of your choice. You can then use the resulting order file with the linker's `-sectorder` option as described in "Step 4: Link the program with the order file" (page 131).

To create a default order file, first run the linker with the `-whatsloaded` option:

```
cc -ooutputFile inputFile.o -whatsloaded > loadedFile
```

This creates a file, *loadedFile*, that lists the object files loaded in the executable, including any in frameworks or other libraries. The `-whatsloaded` option can also be used to make sure that order files generated by `gprof -S` include names for procedures in static libraries.

Using the file *loadedFile*, you can run `nm` with the `-onjls` options and the `__TEXT __text` argument:

```
nm -onjls __TEXT __text `cat loadedFile` > orderFile
```

The content of the file *orderFile* is the symbol table for the text section. Procedures are listed in the symbol table in their default link order. You can rearrange entries in this file to change the order in which you want procedures to be linked, then run the linker as described in "Step 4: Link the program with the order file" (page 131).

Using pagestuff to Examine Pages on Disk

The `pagestuff` tool helps you measure the effectiveness of your procedure ordering by telling you which pages of the executable file are likely to be loaded in memory at a given time. This section briefly describes how to use this tool; for more information, see "pagestuff Mach-O File Page Analysis Tool" (page 114).

The `pagestuff` tool prints out the symbols on a particular page of executable code. The following is the syntax for the command:

`pagestuff` *filename* [*pageNumber* `| -a`]

The output of `pagestuff` is a list of procedures contained in *filename* on page *pageNumber*. To view all the pages of the file, use the `-a` option in place of the page number. This output allows you to determine if each page associated with the file in memory is optimized. If it isn't, you can rearrange entries in the order file and link the executable again to maximize performance gains. For example, move two related procedures together so they are linked on the same page. Perfecting the ordering may require several cycles of linking and tuning.

Grouping Routines According to Usage

Why generate profile data for individual operations of your application? The strategy is based on the assumption that a large application has three general groups of routines:

- **Hot routines** run during the most common usage of the application. These are often primitive routines that provide a foundation for the application's features (for example, routines for accessing a document's data structures) or routines that implement the core features of an application, such as routines that implement typing in a word processor. These routines should be clustered together in the same set of pages.
- **Warm routines** implement specific features of the application. Warm routines are usually associated with particular features that user performs only occasionally (such as launching, printing, or importing graphics). Because these routines are used reasonably often, cluster them in the same small set of pages so they will load quickly. However, because there are long periods when users aren't accessing this functionality, these routines should not be located in the hot category.

- **Cold routines** are rarely used in the application. Cold routines implement obscure features or cover boundary or error cases. Group these routines together to avoid wasting space on a hot or warm page.

At any given time, you should expect most of the hot pages to be resident, and you should expect the warm pages to be resident for the features that the user is currently using. Only very rarely should a cold page be resident.

To achieve this ideal ordering, gather a number of profile data sets. First, gather the hot routines. As described above, compile the application for profiling, launch it, and use the program. Using `gprof -S`, generate a frequency sorted order file called `hot.order` from the profile data.

After creating a hot order file, create order files for features that users occasionally use, such as routines that only run when the application is launched. Printing, opening documents, importing images and using various non-document windows and tools are other examples of features that users use occasionally but not continually, and are good candidates for having their own order files. Naming these order files after the feature being profiled (for example, `feature.order`) is recommended.

Finally, to generate a list of all routines, build a "default" order file `default.order` (as described in "Simple Procedure Reordering" (page 127)).

Once you have these order files, you can combine them with a utility named `unique`, (Listing 7-1). This program simply combines files by removing duplicate lines, retaining the ordering of the original data. In our example you would generate your final order file with this command line:

```
unique hot.order feature1.order ... featureN.order default.order >
final.order
```

Of course, the real test of the ordering is the amount by which paging I/O is reduced. Run your application, use different features, and examine how well your ordering file is performing under different conditions. You can use the `top` tool (among others) to measure paging performance (see "Analyzing Performance" (page 61)).

Listing 7-1 “unique“

```
//
//  unique
//
//  A command for combining files while removing
//  duplicate lines of text. The order of other lines of text
//  in the input files is preserved.
//
//  Build using this command line:
//
//  cc -ObjC -O -o unique -framework Foundation Unique.c
//
//  Note that “unique” differs from the BSD command “uniq” in that
//  “uniq” combines duplicate adjacent lines, while “unique” does not
//  require duplicate lines to be adjacent. “unique” is also spelled
//  correctly.
//

#import <stdio.h>
#import <string.h>
#import <Foundation/NSSet.h>
#import <Foundation/NSData.h>

#define kBufferSize 8*1024

void ProcessFile(FILE *fp)
{
    char buf[ kBufferSize ];

    static id theSet = nil;

    if( theSet == nil )
    {
        theSet = [[NSMutableSet alloc] init];
    }

    while( fgets(buf, kBufferSize, fp) )
    {
        id      dataForString;
        dataForString = [[NSData alloc]  initWithBytes:buf
                                         length:strlen(buf)];

        if( ! [theSet containsObject:dataForString] )
        {
            [theSet addObject:dataForString];
            fputs(buf, stdout);
        }
    }
}

int main( int argc, char *argv[] )
{
    int     i;
    FILE *  theFile;
    int     status = 0;

    if( argc > 1 )
    {
        for( i = 1; i < argc; i++ )
        {
            if( theFile = fopen( argv[i], "r" ) )
            {
                ProcessFile( theFile );
                fclose( theFile );
            }
            else
            {
                fprintf( stderr, "Could not open ‘%s’\n", argv[i] );
                status = 1;
                break;
            }
        }
    }
    else
    {
        ProcessFile( stdin );
    }

    return status;
}
```

Finding That One Last Hot Routine

After reordering you will usually have a region of pages with cold routines that you expect to be rarely used, often at the end of your text ordering. However, one or two hot routines might slip through the cracks and land in this cold section. This is a costly mistake, because using one of these hot routines now requires an entire page to be resident, a page that is otherwise filled with cold routines that are not likely to be used.

Check that the cold pages of your executable are not being paged in unexpectedly. Look for pages that are resident with high-page offsets in the cold region of your application's text segment. If there is an unwanted page, you need to find out what routine on that page is being called. One way to do this is to profile during the particular operation that is touching that page, and use the `grep` tool to search the profiler output for routines that reside on that page. Alternatively, a quick way to identify the location where a page is being touched is to run the application under the `gdb` debugger and use the Mach call `vm_protect` to disallow all access to that page:

```
(gdb) p vm_protect(task_self(), startpage, vm_page_size, FALSE, 0);
```

After clearing the page protections, any access to that page causes a memory fault, which breaks the program in the debugger. At this point you can simply look at the function call stack (using the `bt` command) to learn why the routine was being called.

Reordering Other Sections

You can use the `-sectorder` option of the linker to order blocks in most of the sections of the executable. Sections that might occasionally benefit from reordering are literal sections, such as the `__TEXT` segment's `__cstring` section, and the `__DATA` segment's `__data` section.

Reordering Literal Sections

The lines in the order file for literal sections can most easily be produced with the `ld` and `otool` tools. For literal sections, `otool` creates a specific type of order file for each type of literal section:

- For C string literal sections, the order-file format is one literal C string per line (with ANSI C escape sequences allowed in the C string). For example, a line might look like

  ```
  Hello world\n
  ```

- For 4-byte literal sections, the order-file format is one 32-bit hex number with a leading 0x per line with the rest of the line treated as a comment. For example, a line might look like

  ```
  0x3f8ccccd (1.10000002384185790000e+00)
  ```

- For 8-byte literal sections, the order file line consists of two 32-bit hexadecimal numbers per line separated by white space each with a leading 0x, with the rest of the line treated as a comment. For example, a line might look like:

  ```
  0x3ff00000 0x00000000 (1.00000000000000000000e+00)
  ```

- For literal pointer sections, the format of the lines in the order file represents the pointers, one per line. A literal pointer is represented by the segment name, the section name of the literal pointer, and then the literal itself. These are separated by colons with no extra white space. For example, a line might look like:

  ```
  __OBJC:__selector_strs:new
  ```

- For all the literal sections, each line in the order file is simply entered into the literal section and appears in the output file in the order of the order file. No check is made to see if the literal is in the loaded objects.

To reorder a literal section, first create a "whatsloaded" file using the `ld -whatsloaded` option as described in section "Creating a Default Order File" (page 132). Then, run `otool` with the appropriate options, segment and section names, and filenames. The output of `otool` is a default order file for the specified section. For example, the following command line produces an order file listing the default load order for the `__TEXT` segment's `__cstring` section in the file `cstring_order`:

```
otool -X -v -s __TEXT __cstring `cat whatsloaded` > cstring_order
```

Once you've created the file `cstring_order`, you can edit the file and rearrange its entries to optimize locality of reference. For example, you can place literal strings used most frequently by your program (such as labels that appear in your user interface) at the beginning of the file. To produce the desired load order in the executable, use the following command:

```
cc -o hello hello.o -sectorder __TEXT __cstring  cstring_order
```

Reordering Data Sections

There are currently no tools to measure code references to data symbols. However, you might know a program's data-referencing patterns and might be able to get some savings by separating data for seldom-used features from other data. One way to approach `__data` section reordering is to sort the data by size so small data items end up on as few pages as possible. For example, if a larger data item is placed across two pages with two small items sharing each of these pages, the larger item must be paged in to access the smaller items. Reordering the data by size can minimize this sort of inefficiency. Because this data would normally need to be written to the virtual-memory backing store, this could be a major savings in some programs.

To reorder the `__data` section, first create an order file listing source files and symbols in the order in which you want them linked (order file entries are described at the beginning of "Step 3: Run gprof to Create Order Files" (page 129)). Then, link the program using the `-sectorder` command-line option:

```
cc -o <outputfile> <inputfile>.o -sectorder __DATA __data <orderfile> -e start
```

To use an order file with a Project Builder project, either add a build setting called `SECTORDER_FLAGS` to the Deployment build style, or set the `SECTORDER_FLAGS` variable in the Expert Build Settings table of the Build Settings tab of the Target Settings pane. In either case, the value of the setting should be `-sectorder __DATA __data` *orderFilename*.

Reordering Assembly Language Code

Some additional guidelines to keep in mind when reordering routines coded in assembly language:

- temporary labels in assembly code

  Within hand-coded assembly code, be careful of branches to temporary labels that branch over a non temporary label. For example, if you use a label that starts with "L" or a *d* label (where *d* is a digit), as in this example

  ```
  foo: b 1f
      ...
  bar: ...
  1:   ...
  ```

The resulting program won't link or execute correctly, because only the symbols `foo` and `bar` make it into the object file's symbol table. References to the temporary label `1` are compiled as offsets; as a result, no relocation entry is generated for the instruction `b 1f`. If the linker does not place the block associated with the symbol `bar` directly after that associated with `foo`, the branch to `1f` will not go to the correct place. Because there is no relocation entry, the linker doesn't know to fix up the branch. The source-code change to fix this problem is to change the label `1` to a nontemporary label (`bar1` for example). You can avoid problems with object files containing hand-coded assembly code by linking them whole, without reordering.

- the pseudo-symbol `.section_start`

  If the specified section in any input file has a non-zero size and there is no symbol with the value of the beginning of its section, the linker uses the pseudo symbol `.section_start` as the symbol name it associates with the first block in the section. The purpose of this symbol is to deal with literal constants whose symbols do not persist into the object file. Because literal strings and floating-point constants are in literal sections, this not a problem for Apple compilers. You might see this symbol used by assembly-language programs or non-Apple compilers. However, you should not reorder such code and you should instead link the file whole, without reordering (see"Step 4: Link the program with the order file" (page 131)).

Index