Memory Usage Performance Guidelines

Tools > Performance

2006-06-28

Contents

Figures, Tables, and Listings

Introduction to Memory Usage Performance Guidelines

Analyzing your memory usage is one way to fix other performance problems you may have, such as an increased code footprint or slow code. Efficient memory usage is critical to performance in any application. Increased memory usage not only increases the in-memory footprint of your application, it also increases the time spent allocating and manipulating that memory.

Organization of This Document

This programming topic includes the following articles:

- "Memory Management in Mac OS X" (page 9) introduces terminology and provides a high-level overview of the Mac OS X virtual memory system.
- "Optimizing Your Memory Allocations" (page 15) describes the best techniques for allocating, initializing, and copying memory.
- "Examining Memory Allocation Patterns" (page 23) describes the tools and techniques for analyzing your application's memory usage.
- "Finding Memory Leaks" (page 35) describes the tools and techniques for finding memory leaks in your application.
- "Enabling the Malloc Debugging Features" (page 39) describes the environment variables used to enable malloc history logging. You must set some of these variables before using some of the memory analysis tools.
- "Viewing Virtual Memory Usage" (page 43) describes the tools and techniques for analyzing your application's in-memory footprint.

Memory Management in Mac OS X

Efficient memory management is an important aspect of writing high performance code in Mac OS X code. Tuning your memory usage can reduce both your application's memory footprint and the amount of CPU time it uses. In order to properly tune your code though, you need to understand something about how Mac OS X manages memory.

Unlike earlier versions of Mac OS, Mac OS X includes a fully-integrated virtual memory system that you cannot turn off. It is always on, providing up to 4 gigabytes of addressable space per 32-bit process and approximately 18 exabytes of addressable space for 64-bit processes. However, few machines have this much dedicated RAM for the entire system, much less for a single process. To compensate for this limitation, the virtual memory system uses hard disk storage to hold data not currently in use. This hard disk storage is sometimes called the "swap" space because of its use as storage for data being swapped in and out of memory.

Note: Unlike most UNIX-based operating systems, Mac OS X does not use a preallocated swap partition for virtual memory. Instead, it uses all of the available space on the machine's boot partition.

The following sections introduce terminology and provide a brief overview of the Mac OS X virtual memory system. For more detailed information on how the Mac OS X virtual memory system works, please see *Kernel Programming Guide*.

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**, which is 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.

If the translation from a logical page address to a physical address fails, a **page fault** occurs. The virtual memory system invokes a special page-fault handler to stop executing the current code and respond to the fault. The page-fault handler finds a free page of physical memory, transfers the data from the backing store to the physical page, and then updates the page table so that the page now appears to be at the correct logical address. If no free pages are available in physical memory, the handler must first release an existing page. If that page contains modified data, the handler writes its contents to the backing store before releasing it. 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, the system reads 4 kilobytes from disk. Extended periods of paging activity reduce performance significantly; 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 the CPU caches. Because a page fault involves reading data from disk, and potentially writing data to disk as well, reducing the number of occurring page faults can have a significant improvement on overall system performance.

Virtual Memory in Mac OS X

In Mac OS X, each process has its own sparse 32-bit or 64-bit virtual address space. For 32-bit processes, each process has an address space that can grow dynamically up to a limit of four gigabytes. For 64-bit processes, the address space can grow dynamically up to a limit of approximately 18 exabytes. As an application uses up space, the virtual memory system allocates additional swap file space on the root file system.

The virtual address space of a process consists of mapped regions of memory. Each region of memory in the process 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). Because regions contain a given number of pages, they 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 to track and manage the resident and nonresident pages of that region. A region can map either to an area of memory in the backing store or to a specific file-mapped file in the file system.

The VM object maps regions in the backing store through the default pager and maps file-mapped files 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 directly into a file. This mechanism lets you read and write portions of the file as if they were located in memory.

A VM object 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 that page. If one process writes to the page, a new, writable copy of the page is created in the address space of the process doing the writing. This mechanism allows the system to copy large quantities of data efficiently.

Each VM object contains several fields, as shown in Table 1 (page 10).

Table 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.

Field	Description
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 memory pages:

- The **active list** contains pages that are currently mapped into memory and have been recently accessed.
- The **inactive list** contains pages that are currently resident in physical memory but have not been accessed recently. These pages contain valid data but may be released from memory at any time.
- The **free list** contains pages of physical memory that are not associated with any address space of VM object. These pages are available for immediate use by any process that needs them.

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 a page has been accessed recently, it is reactivated and placed on the end of the active list. If an inactive page contains data that has not been written to the backing store recently, its contents must be paged out to disk before it can be placed on the free list. If an inactive page has not been modified 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 12)). 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. This routine finds free space on an existing page or allocates new pages using `vm_allocate` to create space for the new memory block. Through the `vm_allocate` routine, the kernel performs a series of initialization steps:

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.
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 the backing store. Everything is mapped virtually within the system. 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 then performs the following steps for each page fault:

1. It acquires a page from the free list and fills it with zeroes.
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 reclaims physical pages for the free list by swapping inactive pages out of memory. To do this, the kernel 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 the 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 the backing store back into physical memory. A memory access fault initiates the page-in process. 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:

- A **soft fault** occurs when the page of the referenced address is resident in physical memory but is currently not mapped into the address space of this process.
- A **hard fault** occurs when the page of the referenced address is not in physical memory but is swapped out to backing store (or is available from a mapped file). This is what is typically known as a page fault.

When any type of fault occurs, the kernel locates the map entry and VM object for the accessed region. The kernel then goes through the VM object's list of resident pages. If the desired page is in the list of resident pages, the kernel generates a soft fault. If the page is not in the list of resident pages, it generates a hard fault.

For soft faults, the kernel maps the physical memory containing the pages to the virtual address space of the process. The kernel then marks the specific page as active. If the fault involved a write operation, the page is also marked as modified so that it will be written to backing store if it needs to be freed later.

For hard faults, the VM object's pager finds the page in the backing store or from the file-mapped file, depending on the type of pager. After making the appropriate adjustments to the map information, the pager moves the page into physical memory and places the page on the active list. As with a soft fault, if the fault involved a write operation, the page is marked as modified.

Shared Memory

Shared memory is memory that can be written to or read from by 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 the following:

- 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, any programs that also use that memory share the corrupted data.

Wired Memory

Wired memory (also called **resident** memory) stores kernel code and data structures that should never be paged out 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 (page 13) lists some of the wired-memory costs for user-generated entities.

Table 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 wired memory generated through user-level requests, the following kernel entities also use wired memory:

- VM objects
- the virtual memory buffer cache
- I/O buffer caches
- drivers

Wired data structures are also associated with the physical page and map tables used to store virtual-memory mapping information, Both of these entities scale with the amount of available 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 becomes invalid. Instead it is "garbage collected" when the free-page count falls below the threshold that triggers page out events.

Optimizing Your Memory Allocations

Memory is an important resource for your application so it's important to take the time to examine your application's memory allocation patterns and make changes as necessary.

You can gather a history of your allocations using the Sampler program or using the `malloc_history` command-line tool. For more information on analyzing your memory usage, see "Examining Memory Allocation Patterns" (page 23).

Memory Allocation in Mac OS X

Mac OS X implements a highly-tuned, threadsafe allocation library, providing standard implementations of the `malloc`,`calloc`, `realloc`, and `free` routines, among others. If you are allocating memory using older routines such as `NewPtr` or `NewHandle`, you should change your code to use `malloc` instead. The end result is the same since most legacy routines are now wrappers for `malloc` anyway.

If you are using a custom `malloc` implementation, you should consider moving to the system-supplied `malloc` routines. The Mac OS X `malloc` implementation is highly optimized and fully supports the Apple-provided memory analysis tools. Moving to Apple's implementation not only gains you the ability to analyze your memory, it lets you remove your custom code from your executable, thus reducing your application footprint.

The following sections provide some details on how the Mac OS X allocation library handles large and small allocations. This information can help you identify the costs associated with each type of allocation. Note that although the following sections talk about the behaviors of the `malloc` routine, those behaviors also apply to routines such as `calloc` and `realloc`.

Allocating Small Memory Blocks

For allocations of less than a few virtual memory pages, `malloc` suballocates the requested amount from a list (or "pool") of free blocks of increasing size. Any small blocks you deallocate using the `free` routine are added back to the pool and reused on a "best fit" basis. The memory pool is itself is comprised of several virtual memory pages and allocated using the `vm_allocate` routine.

The granularity of any block returned by `malloc` is 16 bytes. Any blocks you allocate will be at least 16 bytes in size or comprised of a block that is a multiple of 16. Thus, if you request 4 bytes, `malloc` returns a block of 16 bytes. If you request 24 bytes, `malloc` returns a block of 32 bytes.

Note: By their nature, allocations smaller than a single virtual memory page in size cannot be page aligned.

Allocating Large Memory Blocks

For allocations greater than a few virtual memory pages, `malloc` uses the `vm_allocate` routine to obtain a block of the requested size.The `vm_allocate` routine assigns an address range to the new block in the virtual memory space of the current process but does not allocate any physical memory. Instead, the `malloc` routine pages in the memory for the allocated block as it is used.

The granularity of large memory blocks is 4096 bytes, the size of a virtual memory page. If you are allocating a large memory buffer, you should consider making it a multiple of this size.

Note: Large memory allocations are guaranteed to be page-aligned.

For large allocations, you may find that it makes sense to allocate virtual memory using `vm_allocate` directly. The example in Listing 1 (page 16) shows how to use the `vm_allocate` function.

Listing 1 Allocating memory with vm_allocate

```
void* AllocateVirtualMemory(size_t size)
{
    char*           data;
    kern_return_t   err;

    // In debug builds, check that we have
    // correct VM page alignment
    check(size != 0);
    check((size % 4096) == 0);

    // Allocate directly from VM
    err = vm_allocate(  (vm_map_t) mach_task_self(),
                        (vm_address_t*) &data,
                        size,
                        VM_FLAGS_ANYWHERE);

    // Check errors
    check(err == KERN_SUCCESS);
    if(err != KERN_SUCCESS)
    {
        data = NULL;
    }

    return data;
}
```

Allocating Memory in Batches

If your code allocates multiple, identically-sized memory blocks, you can use the `malloc_zone_batch_malloc` function to allocate those blocks all at once. This function offers better performance than the equivalent series of calls to `malloc` to allocate the same memory. Performance is best when the individual block size is relatively small—less than 4K in size. The function does its best to allocate all of the requested memory but may return less than was requested. When using this function, check the return values carefully to see how many blocks were actually allocated.

Batch allocation of memory blocks is supported in Mac OS X version 10.3 and later. For information, see the `/usr/include/malloc/malloc.h` header file.

About Memory Zones

A **zone** is a variable-size range of virtual memory from which `malloc` allocates blocks. All allocations made using the `malloc` function occur within the standard malloc zone, which is created when `malloc` is first called by your application. You can create additional malloc zones and allocate memory in a specific zone.

Note: The term zone is synonomous with the terms heap, pool, and arena in terms of memory allocation using the `malloc` routines.

Zones have the advantage of allowing blocks with similar access patterns or lifetimes to be placed together, theoretically minimizing wasted space or paging activity. 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, or you have measured a specific case where zones will help.

For information on how to use multiple zones in an application, see "Using Multiple Malloc Zones" (page 18)

Tips for Allocating Memory

When it comes time to allocate memory for your program, there are other considerations you should make. The following sections provide guidelines on when and how to allocate memory.

Deferring Memory Allocations

Every memory allocation has a performance cost. That cost is measured by the time it takes to allocate the memory and the space occupied by the memory. If you do not need a particular block of memory right away, you should consider deferring its allocation until the first time you actually need it. Once allocated, you can then use it and delete it or cache it for later use.

Applications often allocate memory during initialization and then use that memory later—or sometimes not at all during a given session. Not only does this cause the application to pay an up-front cost for allocating the memory but it does so needlessly. You can easily improve on this costly approach by deferring the allocation to the first time the memory is needed.

For most operations, you can easily arrange your code to use a block of memory right after you allocate it. But if your application uses global variables, you need another way to ensure the memory is there when you need it, but not before. To accomplish this with a minimum of code modification, do the following:

- Turn any global variables into static variables so that they are inaccessible to other code modules.

- Create a public accessor function to access the static variable and allocate and initialize the buffer for it upon the first invocation.

Listing 2 (page 18) gives an example of this technique. Code modules that want to access the global buffer call the function to access the pointer.

Listing 2 Lazy allocation of memory through an accessor

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

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 around the `if` statement and any required initialization code.

Initializing Memory

Memory allocated using `malloc` is not guaranteed to be initialized with zeroes. Instead of using `memset` to initialize the memory, a better choice is to use the `calloc` routine to allocate the memory in the first place.

When you call `memset` right after `malloc`, the virtual memory system must map the corresponding pages into memory in order to zero-initialize them. This operation can be very expensive and wasteful, especially if you do not use the pages right away.

The `calloc` routine reserves the required virtual address space for the memory but waits until the memory is actually used before initializing it. This approach alleviates the need to map the pages into memory right away. It also lets the system initialize pages as they're used, as opposed to all at once.

Using Multiple Malloc Zones

All memory blocks are contained within a malloc zone (also referred to as a malloc heap). All allocations made by `malloc` function occur within the default malloc zone of the current process, which is created when `malloc` is first called. Although it is generally not recommended, you can create additional zones if measurements show there to be potential performance gains. For example, if the effect of releasing a large number of temporary (and isolated) objects is slowing down your application, you could allocate them in a zone instead and simply deallocate the zone.

Basic support for zones is defined in `/usr/include/malloc/malloc.h`. Use the `malloc_create_zone` function to create a custom malloc zone or the `malloc_default_zone` function to get the default zone for your application. To allocate memory in a particular zone, use the `malloc_zone_malloc`, `malloc_zone_calloc`, `malloc_zone_valloc`, or `malloc_zone_realloc` functions. To release the memory in a custom zone, call `malloc_destroy_zone`.

Warning: You should never deallocate the default zone for your application.

If you are a Cocoa developer, you can also use the `NSCreateZone` function to create a custom malloc zone and the `NSDefaultMallocZone` function to get the default zone for your application. To create new objects in a custom zone, use the `allocWithZone:` class method, which is available to all subclasses of NSObject. If your class does not descend from NSObject, use the `NSAllocateObject` function to allocate the memory for your new instances. For more information, see the function descriptions in *Foundation Framework Reference*.

If you are creating objects (or allocating memory blocks) in a custom malloc zone, you can simply free the entire zone when you are done with it, instead of releasing the zone-allocated objects or memory blocks individually. When doing so, be sure your application data structures do not hold references to the memory in the custom zone. Attempting to access memory in a deallocated zone will cause a memory fault and crash your application.

Cache Temporary Buffers

If you have a highly-used function that allocates a large temporary buffer for some calculations, you might want to consider alternative ways to allocate that buffer. Instead of creating a new block of memory each time it's called, your function could instead cache a buffer initially and reuse that buffer during subsequent invocations. If your function needs a variable buffer space, you can always grow the buffer as needed. For multi-threaded applications, you can attach the buffer pointer to your thread's context. For single-threaded applications, you can just store the pointer in a global variable.

Caching buffers eliminates much of the overhead for functions that regularly allocate and free large blocks of memory. However, this technique is only appropriate for functions that are called frequently. Also, you should be careful not to cache too many large buffers. Caching buffers does add to the memory footprint of your application. You should be sure to gather metrics for your program with and without the caches to see which yields better performance.

Release Your Memory

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 releasememory causes memory leaks, which have a direct impact on performance. To help track down memory leaks, use the MallocDebug application or the `leaks` command-line tool. Both of these tools are described in "Examining Memory Allocation Patterns" (page 23).

Using Handles in Carbon

If you have existing code from Mac OS 9 that you are porting to Mac OS X, you can achieve some performance gains by simplifying your handle-related code. The benefit offered by handles in Mac OS 9 is no longer relevant in applications built for Mac OS X. In particular, there is no need to compact the memory blocks referenced by handles. As a result, your handles never move and there is no need to lock them when you want to access their contents.

If you have code that makes calls to `HLock`, `HUnlock`. `HSetState`, or `HGetState`, you can either conditionally compile that code out for Mac OS X or you can remove the code entirely. The only exception to this rule is cases where your code calls the `SetHandleSize` function, which can potentially move a handle if more space is required. If your code needs to access a handle that might be resized at some point, you should lock the handle first.

Copying Memory

There are two main approaches to copying memory in Mac OS X: direct and delayed. For most situations, the direct approach offers the best overall performance. However, there are times when using a delayed-copy operation has is benefits. The goal of the following sections is to introduce you to the different approaches for copying memory and the situations when you might use those approaches.

Copying Memory Directly

The direct copying of memory involves using a routine such as `memcpy` or `memmove` to copy bytes from one block to another. Both the source and destination blocks must be resident in memory at the time of the copy. However, these routines are especially suited for the following situations:

- the size of the block you want to copy is small (under 16 kilobytes).
- you intend to use either the source or destination right away.
- the source or destination block is not page aligned.
- the source and destination blocks overlap.

If you do not plan to use the source or destination data for some time, performing a direct copy can decrease performance significantly for large memory blocks. Copying the memory directly increases the size of your application's working set. Whenever you increase your application's working set, you increase the chances of paging to disk. If you have two direct copies of a large memory block in your working set, you might end up paging them both to disk. When you later access either the source or destination, you would then need to load that data back from disk, which is much more expensive than using `vm_copy` to perform a delayed copy operation.

Note: If the source and destination blocks overlap, you should prefer the use of `memmove` over `memcpy`. Both implementations handle overlapping blocks correctly in Mac OS X, but the implementation of `memcpy` is not guaranteed to do so.

Delaying Memory Copy Operations

If you intend to copy many pages worth of memory, but don't intend to use either the source or destination pages immediately, then you may want to use the `vm_copy` routine. Unlike `memmove` or `memcpy`, `vm_copy` does not touch any real memory. It modifies the virtual memory map to indicate that the destination address range is a copy-on-write version of the source address range.

The `vm_copy` routine is more efficient than `memcpy` in very specific situations. Specifically, it is more efficient in cases where your code does not access either the source or destination memory for a fairly large period of time after the copy operation. The reason that `vm_copy` is effective for delayed usage is the way the kernel handles the copy-on-write case. In order to perform the copy operation, the kernel must remove all references to the source pages from the virtual memory system. The next time a process accesses data on that source page, a soft fault occurs, and the kernel maps the page back into the process space as a copy-on-write page. The process of handling a single soft fault is almost as expensive as copying the data directly.

Copying Small Amounts of Data

If you need to copy a small blocks of non-overlapping data, you should prefer `memcpy` over any other routines. For small blocks of memory, the GCC compiler can optimize out this routine and replace it with inline instructions to copy the data by value. The compiler may not optimize out other routines such as `memmove` or `BlockMoveData`.

Copying Data to Video RAM

When copying data into VRAM, use the `BlockMoveDataUncached`function instead of functions such as `bcopy`. The `bcopy` routine uses cache-manipulation instructions that may cause exception errors. The kernel must fix these errors in order to continue, which slows down performance tremendously.

Examining Memory Allocation Patterns

Examining your application's memory allocation patterns can help reveal algorithms that may not be the most efficient in their memory use. If you see a large number of allocations occurring during a loop, you may decide to go back and change the allocations to occur outside of the loop. This kind of reduction can have a significant increase in application performance.

Apple provides several tools for examining your memory usage. The MallocDebug tracks the location of memory allocations by recording the application call stack whenever a memory-related function is called. The `malloc_history` tool does many of the same things as MallocDebug but from a command-line interface. You can use these tools to look for unexpectedly large allocations or allocations that were made but are no longer needed.

If you are writing a Cocoa application, you should use the ObjectAlloc program and `heap` tool to see which Objective-C objects your program creates. The ObjectAlloc program is good for finding problems involving allocation trends, retain/release problems, or other problems involving object allocations. Similarly, the `heap` tool helps you examine the objects currently in use by a program.

Debugging Allocations With MallocDebug

The MallocDebug application provides tools for inspecting your program's memory use and for finding memory leaks. MallocDebug shows currently allocated blocks of memory, organized by the call stack at the time of allocation. You can use MallocDebug to determine how much memory your application allocates, where it allocates that memory, and which functions allocated large amounts of memory. It gathers data from the Carbon Memory Manager, Core Foundation object allocations, Cocoa object allocations, and `malloc`allocations.

MallocDebug does not require prior instrumentation of the program—that is, you don't need to link with special libraries or call special functions. Instead, MallocDebug launches your application using its own instrumented version of the `malloc` library calls.

Note: The custom malloc library used by MallocDebug may hold on to memory blocks longer than normal for analysis purposes. As a result, you should not try to gather metrics regarding the size of your program's memory footprint while running it under MallocDebug.

MallocDebug includes a number of features you can use to refine your memory analysis:

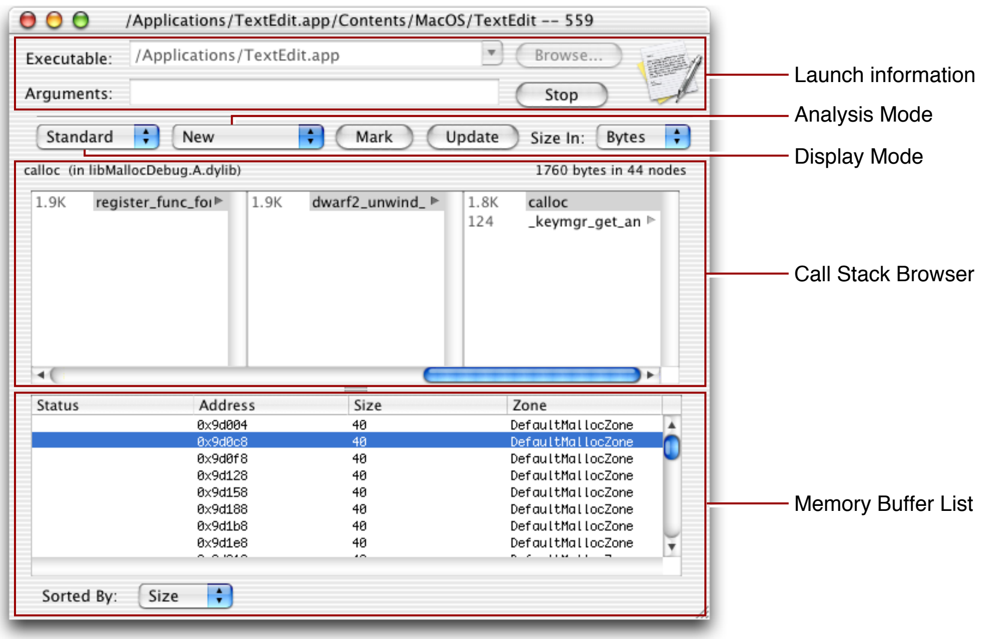
- It provides a hex-dump view for examining raw memory.
- It allows you to mark off any period of execution for analysis.
- It allows you to export performance data for detailed examination or for further analysis and refinement by command-line tools. The export feature gives you the freedom to look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

For information on how to use MallocDebug to identify memory leaks in your program, see "Finding Memory Leaks" (page 35).

Using MallocDebug

After launching MallocDebug, the main window appears (Figure 1 (page 24)). There are three basic sections in the MallocDebug window. Information about the launched program is at the top of the window. The center portion displays the call stack browser. The bottom portion displays the memory buffer browser.

Figure 1 MallocDebug main window



Executable: /Applications/TextEdit.app
Arguments:
Standard | New | Size In: Bytes

calloc (in libMallocDebug.A.dylib) 1760 bytes in 44 nodes

```
1.9K  register_func_foı ▸   1.9K  dwarf2_unwind_ ▸   1.8K  calloc
                                                     124   _keymgr_get_an ▸
```

Status	Address	Size	Zone
	0x9d004	40	DefaultMallocZone
	0x9d0c8	40	DefaultMallocZone
	0x9d0f8	40	DefaultMallocZone
	0x9d128	40	DefaultMallocZone
	0x9d158	40	DefaultMallocZone
	0x9d188	40	DefaultMallocZone
	0x9d1b8	40	DefaultMallocZone
	0x9d1e8	40	DefaultMallocZone

Sorted By: Size

To start a new MallocDebug session, you must select and launch the application you want to analyze by doing the following:

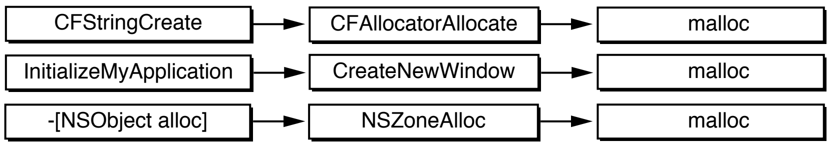
1. Enter the full path to the program in the Executable field, or click the Browse button and select the program using the file-system browser.
2. If you want to run the executable with command-line arguments, enter them in the Arguments field.
3. Click the Launch button.

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 (see Figure 1 (page 24)). This browser shows you where memory allocations occurred by gathering stack snapshots whenever one of the malloc library routines was encountered. Figure 2 (page 25) shows a sample set of data for calls to the `malloc` routine.

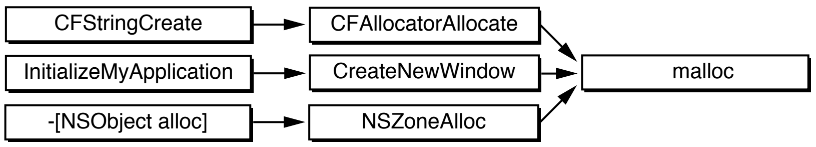
Figure 2 Function call stacks gathered at runtime



MallocDebug coalesces the call stack information it gathers into a call tree by overlapping equivalent sequences of functions. It then presents this information in the call stack browser. The call stack browser has three display modes: standard, inverted, and flat. Each display mode presents the data in a different way to help you identify trends. You can choose which mode you want from the left-most pop-up menu and toggle back and forth as needed.

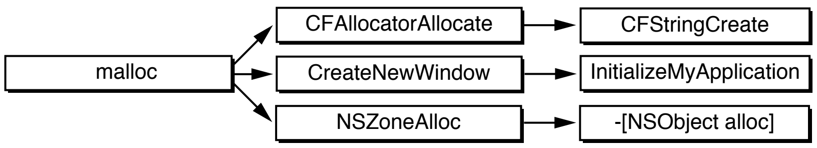
Standard mode presents each call stack hierarchically from the function at the top of the stack (for instance, `main`) to the function that performs the allocation: `malloc`, `calloc`, and so on. Each element of the browser shows the amount of memory that has been allocated in the call stack involving that method or function. Figure 3 (page 25) illustrates the structure of the call stack in standard mode.

Figure 3 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 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 4 (page 25) illustrates the structure of the call stack in inverted mode.

Figure 4 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 a specific function and every function it called, not just those at the top or bottom of the call stack.

The analysis 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 Allocations** Gives you the call trees for all currently allocated buffers in your application.
- **Allocations since Mark** Displays functions and methods in which an allocation has occurred since launch time or the last mark. See "Taking a Snapshot of Memory Usage" (page 26) for information on how to display allocations over a specific period of time.
- **Leaks** This item displays a call tree showing leaked memory blocks in your program. For further discussion of this analysis mode, see "Finding Memory Leaks" (page 35).
- **Overruns/underruns** Displays a call tree with a list of buffers that were written to incorrectly, caused by writing to memory before or after the buffer boundary. If the program wrote past the end of a buffer, a right arrow (>) appears by the buffer. Similarly, if the application wrote 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 26).

Taking a Snapshot of Memory Usage

When you launch a program with MallocDebug, the main window displays the allocation activity that occurred during launch time. When you click the Update button, MallocDebug shows memory usage up to the current point in time. If you want to display allocations from a particular point in time, you can do the following:

1. Press the Mark button.
2. Exercise a portion of your program.
3. Select the "Allocations from mark" item from the analysis mode 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 you clicked the Mark button that have not been freed.

Analyzing Raw Memory

When you select an allocation buffer in the call stack browser, the memory buffer list (shown in Figure 1 (page 24)) might show one or more lines of data. Each line in this list represents a block of memory allocated by the currently selected function or by a function eventually called by that function. Each line contains the address of the buffer, its size (in bytes), and the zone in which it was allocated. Double-clicking one of these lines opens the Memory Viewer Panel window, which you can use to inspect the contents of memory at that location.

If code attempts to write before the start or past the end of a buffer, the memory buffer list shows an appropriate indicator in the Status column. If bytes were written before the buffer, the column displays a less-than < character. If bytes were written after the buffer, the column contains a greater-than > character. Use the popup menu below the list to sort the list contents.

MallocDebug helps you catch some types of problems by writing certain hexadecimal patterns into the hex values displayed in the Memory Viewer Panel window. It overwrites freed memory with `0X55` and it guards against writing beyond a block's boundaries by putting the values `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.

Evaluating MallocDebug Problem Reports

Some of the reports that MallocDebug presents identify obvious problems that you should fix immediately. Some of these problems include leaks, buffer overruns, and references to freed memory. Other problems are more subjective in nature and require you to make a determination as to whether there is a problem.

To improve your program's overall allocation behavior, use MallocDebug's detailed accounting of memory usage to explore the memory usage of your program. This can help you identify wasted memory allocations or unexpected allocation patterns and thus optimize your program's memory usage. As you analyze your memory allocations, consider the following items:

- Don't ignore small buffers. A small leaked buffer might itself contain references to larger buffers, which then also become leaks. (The `leaks` tool is better at reporting leaks of this nature.)
- Look at allocation patterns during specific intervals of typical program use, especially where you suspect memory usage might be a problem.
- The inverted display mode for the call stack browser can sometimes yield results faster because it shows which routines are actually calling `malloc`. The normal display mode is better for seeing memory allocations in particular modules of your code.
- Keep track of important statistics, such as private memory usage and total allocated memory, so you can compare them against previous measurements.

Limitations of MallocDebug

The following section describes some of the issues you may run into when running MallocDebug.

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 memory allocated by the program during its entire span of execution. Memory that has been freed is not shown.

To see memory that your program has allocated and freed, use the `malloc_history` tool. See "Tracking Memory Allocations With malloc_history" (page 29) 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 1 (page 28) gives an example of MallocDebug's crash diagnostic message.

Listing 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 referencing 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 26)). If your program is referencing memory at `0x55555555`, then it is referencing freed memory.

Important: You should always investigate and fix bugs that cause your program to crash. Subtle problems may indicate a design flaw that could cost more time to fix later.

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 load new libraries, such as the heap debugging library used by MallocDebug. As a result, MallocDebug cannot display information about these programs unless they are run by the target user or by a member of the target 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.

Running Under libMallocDebug

If you're writing a simple program that runs from the command-line, you may need to statically link the `malloc` routines into your executable before MallocDebug can attach to your program. Most programs link to the System framework, which is instrumented for use by MallocDebug. If your program does not use this framework, you can explicitly link your program with the `/usr/lib/libMallocDebug.a` library. (If you are running in Mac OS X 10.3.9 or later, you can also execute the command `set env DYLD_INSERT_LIBRARIES /usr/lib/libMallocDebug.A.dylib` from Terminal to attach your program to `libMallocDebug` .) You should not notice any difference in your program's allocation behavior when linking with this library.

If you do link your application to `libMallocDebug` , you should be aware of the following caveats:

- If your code runs in versions of Mac OS X prior to 10.4, you must need to set the `DYLD_FORCE_FLAT_NAMESPACE` environment variable to force the linker to use the malloc routines in `libMallocDebug`. If you are running in Mac OS X v10.4 or later, you do not need to set this variable.
- When running your program in Mac OS X v10.4 or later under gdb with libMallocDebug installed, your program automatically drops into the debugger when `libMallocDebug` detects that memory has been corrupted by a `malloc` or `free` call. To continue running your program, execute the command " `set $PC+=4` " in gdb then continue.
- If your program runs on a version of Mac OS X prior to version 10.3.9, you may need to execute the command " `set start-with-shell 0` " in gdb to debug your program with `libMallocDebug` ..
- In Mac OS X v10.4 and later, child processes created by a `fork` and `exec` now properly inherit the `DYLD_INSERT_LIBRARIES` environment variable setting. Thus, if the parent is running under `libMallocDebug` , so will the child.

Setting Environment Variables

MallocDebug does not contain any built-in mechanism for setting environment variables. You can work around this limitation by setting your environment variables from Terminal and then launching MallocDebug from there. When launched in this manner, your application inherits the Terminal environment, including any environment variables.

Do not launch MallocDebug from Terminal using the `open` command. Instead, run the MallocDebug executable directly. The executable is located in the MallocDebug.app application bundle, usually in the `MallocDebug.app/Contents/MacOS` directory.

Tracking Memory Allocations With malloc_history

The `malloc_history` tool displays backtrace data showing exactly where your program made calls to the `malloc` and `free` functions. If you specify an address when calling `malloc_history`, the tool tracks memory allocations occurring at that address only. If you specify the `-all_by_size` or `-all_by_count` options, the tool displays all allocations, grouping frequent allocations together.

Before using the `malloc_history` tool on your program, you must first enable the malloc library logging features by setting the `MallocStackLogging` to `1`. You may also want to set the `MallocStackLoggingNoCompact` environment variable to retain information about freed blocks. For more information on these variables, see "Enabling the Malloc Debugging Features" (page 39).

The `malloc_history` tool is best used in situations where you need to find the previous owner of a block of memory. If you determine that a particular data is somehow becoming corrupted, you can put checks into your code to print the address of the block when the corruption occurs. You can then use `malloc_history` to find out who owns the block and identify any stale pointers.

The `malloc_history` tool is also suited for situations where Sampler or MallocDebug cannot be used. For example, you might use this tool from a remote computer or in situations where you want a minimal impact on the behavior of your program.

For more information on using `malloc_history`, see the man pages.

Observing Allocations With ObjectAlloc

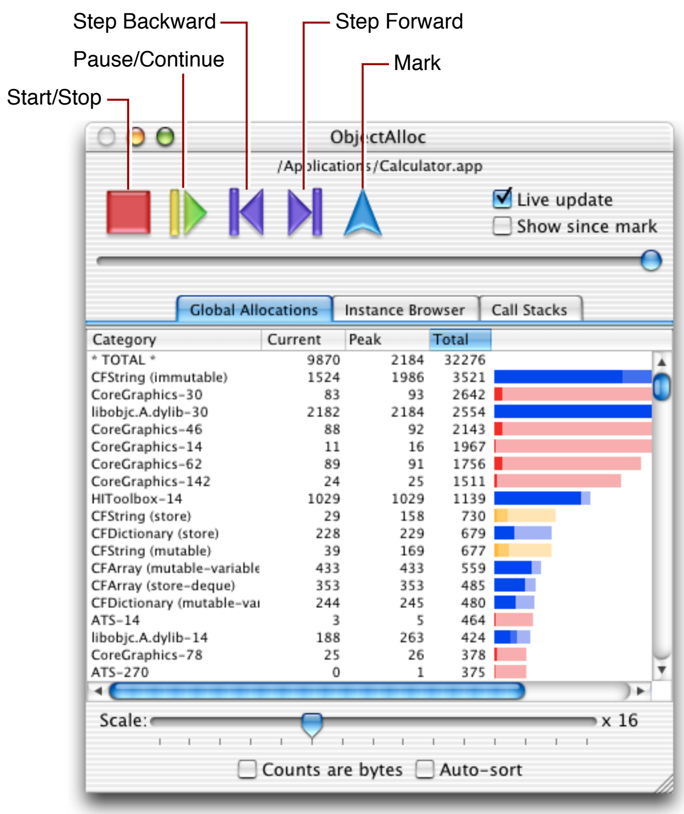
ObjectAlloc allows you to observe memory allocation activity in an application. It shows you the allocations in terms of the number of objects created, rather than where the allocations occurred. Its real-time histograms allow you to directly perceive changes and trends in object counts. It also retains a history of allocations and deallocations, allowing you to identify 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

When you first launch ObjectAlloc, it asks you to choose the application you want to inspect. After selecting the application, ObjectAlloc displays the main window so that you can launch the application and start gathering data. The main window contains several buttons for controlling the launch and execution of the application, as well as for setting mark points from which to examine data. Figure 5 (page 31) shows the ObjectAlloc main window.

Figure 5 ObjectAlloc window



/Applications/Calculator.app

[✓] Live update
[ ] Show since mark

Category	Current	Peak	Total
* TOTAL *	9870	2184	32276
CFString (immutable)	1524	1986	3521
CoreGraphics-30	83	93	2642
libobjc.A.dylib-30	2182	2184	2554
CoreGraphics-46	88	92	2143
CoreGraphics-14	11	16	1967
CoreGraphics-62	89	91	1756
CoreGraphics-142	24	25	1511
HIToolbox-14	1029	1029	1139
CFString (store)	29	158	730
CFDictionary (store)	228	229	679
CFString (mutable)	39	169	677
CFArray (mutable-variable	433	433	559
CFArray (store-deque)	353	353	485
CFDictionary (mutable-var	244	245	480
ATS-14	3	5	464
libobjc.A.dylib-14	188	263	424
CoreGraphics-78	25	26	378
ATS-270	0	1	375

Scale: x 16

[ ] Counts are bytes [ ] Auto-sort

When you click the Start button, ObjectAlloc launches the application and starts displaying object allocation data as it occurs. You can use the buttons along the top of the window to control the gathering and display of collection data. If you want to view past allocations, you must pause or stop the data gathering process before using the controls to step forward or backward through the allocation history. You can use the slider control just under the buttons to watch changes in allocation counts over time.

The Mark button sets a starting point from which to watch allocations. This starting point is a convenience that lets you view allocations over a more recent period of time. Use the "Show since mark" checkbox to toggle between displaying events that occurred since the mark was set and displaying events that occurred since the application was launched.

Note: Due to the sheer quantity of information being processed, updating the ObjectAlloc main window can slow the system down noticeably. Uncheck the "Live update" checkbox to update the display only when the ObjectAlloc window is activated or deactivated.

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, including allocation, retain, release, autorelease, and free events. 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 tree. 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.

Interpreting ObjectAlloc Data

Here are some general guidelines for interpreting the data reported by ObjectAlloc:

- Does the number of instantiated objects match your expectations? If not, you might be creating more temporary objects than anticipated.
- Examine the relationships between different objects. If you allocate one custom object, did you expect to see several other objects created along with it?
- Examine the colored bars in the allocation graph. Large differences between the peak and current allocations (or between the total and current allocations) indicate a surge in allocations. This could reflect a large number of autoreleased objects needing to be released earlier from a loop.

Examining Heaps With the heap Tool

The `heap` tool displays a snapshot of the memory allocated by the malloc library and located in the address space of a specified process. For Cocoa applications, this tool identifies Objective-C objects by name; For both memory blocks and objects, the tool organizes the information by heap, showing all items in the same heap together.

The `heap` tool provides much of the same information as the ObjectAlloc application, but does so in a much less intrusive manner. You can use this tool from a remote session or in situations where the use of ObjectAlloc might slow the system down enough to affect the resulting output.

Finding Memory Leaks

Memory leaks are blocks of allocated memory that the program no longer references. Memory leaks are bugs and should always be fixed. Leaks waste space by filling up pages of memory with inaccessible data and waste time due to extra paging activity. Leaked memory eventually forces the system to allocate additional virtual memory pages for the application, the allocation of which could have been avoided by reclaiming the leaked memory.

The malloc library can only reclaim the memory you tell it to reclaim. If you call `malloc` or any routine that allocates memory, you must balance that call with a corresponding `free`. A typical memory leak occurs when a developer forgets to deallocate memory for a pointer embedded in a data structure. If you allocate memory for embedded pointers in your code, make sure you release the memory for that pointer prior to deallocating the data structure itself.

Another typical memory leak example occurs when a developer allocates memory, assigns it to a pointer, and then assigns a different value to the pointer without freeing the first block of memory. In this example, overwriting the address in the pointer erases the reference to the original block of memory, making it impossible to release.

Apple provides the MallocDebug application and `leaks` command-line tool for automatically tracking down memory leaks. You can also track down leaks manually using other analysis tools, but that task falls under the category of finding memory problems in general and is covered in "Examining Memory Allocation Patterns" (page 23). The following sections describe the MallocDebug and `leaks` tools and show you how to use them to track down memory leaks.

Finding Leaks With MallocDebug

The MallocDebug application is a graphical tool for diagnosing all types of memory problems. MallocDebug includes a memory-leak analysis tool that you can use to identify leaks in your program. The interface for MallocDebug displays potential leaks using a call-graph structure so that you can easily locate the function that generated the leak.

Performing a Global Leak Analysis

MallocDebug uses a conservative garbage-collection algorithm for detecting leaks. This algorithm searches the program's memory for pointers to each malloc-allocated block. Any block that is not referenced at all by the program is marked as a leak.

To initiate a leak search in MallocDebug, do the following:

1. Launch MallocDebug.

2. Open a new window and select the executable you want to examine.

3. Click the Launch button.

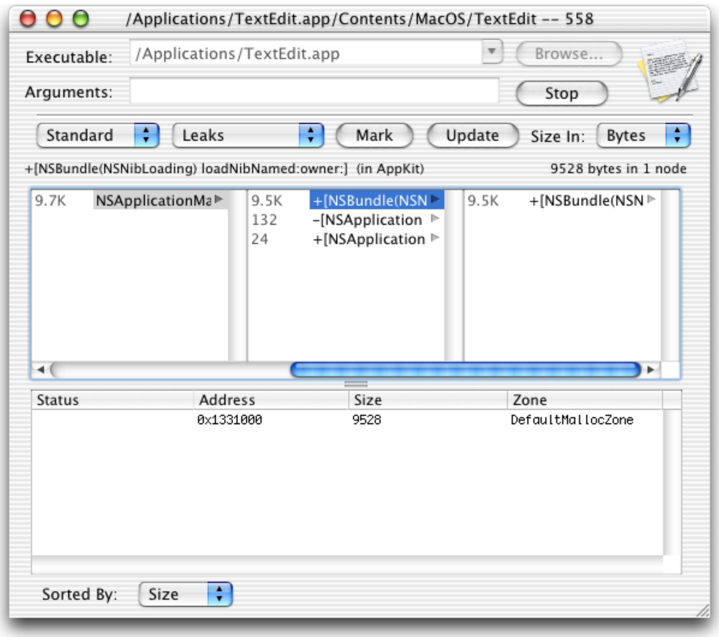
4. Exercise the application features to build its memory profile.

5. In MallocDebug, select “Leaks” from the analysis popup menu to display the memory leaks in your application.

6. Use the call-graph data in the browser to find where the memory was allocated.

Figure 1 shows the MallocDebug main window with the Leaks option selected for viewing. When you select any of the leak-related options from this popup menu, MallocDebug initiates its leak-detection analysis. It then displays the results of the analysis in the browser window. Each entry in the browser includes the amount of memory leaked from that function.

Figure 1 MallocDebug main window


/Applications/TextEdit.app/Contents/MacOS/TextEdit -- 558

Executable: /Applications/TextEdit.app

Arguments:

Standard | Leaks | Size In: Bytes

+[NSBundle(NSNibLoading) loadNibNamed:owner:] (in AppKit) 9528 bytes in 1 node

```
9.7K  NSApplicationMa    9.5K  +[NSBundle(NSN     9.5K  +[NSBundle(NSN
                         132   -[NSApplication
                         24    +[NSApplication
```

Status	Address	Size	Zone
	0x1331000	9528	DefaultMallocZone

Sorted By: Size

The leak analysis performed by MallocDebug identifies all memory that has been leaked since the application was launched. “Finding Leaks for Specific Features” (page 36) describes a way you can use MallocDebug to isolate memory leaks in your application.

Finding Leaks for Specific Features

The leak analysis tools in MallocDebug perform a global search for leaks in your program. However, there are other types of leaks that MallocDebug cannot identify. These are leaks caused by your code allocating a block and then not freeing it. You must identify these leaks yourself using the MallocDebug sampling features. To find these leaks, do the following:

1. In the MallocDebug window, select "Allocations from mark" from the analysis popup button.

2. Click the Mark button.

3. Exercise the target feature of your application.
4. In MallocDebug, click the Update button to display the memory allocated since the Mark button was clicked.
5. Look for any newly-allocated buffers. These may be buffers your code forgot to free after it was done with them.

Using the leaks Tool

The `leaks` command-line tool searches the virtual memory space of a process for buffers allocated by `malloc` but no longer referenced. For each leaked buffer it finds, `leaks` displays the following information:

- the address of the leaked memory
- the size of the leak (in bytes)
- the contents of the leaked buffer

If `leaks` can determine that the object is an instance of an Objective-C or Core Foundation object, it also displays the name of the object. If you do not want to view the contents of each leaked buffer, you can specify the `-nocontext` option when calling `leaks`. If the `MallocStackLogging` environment variable is set and you are running your application in `gdb`, `leaks` displays a stack trace describing where the buffer was allocated. For more information on `malloc` debugging options, see "Enabling the Malloc Debugging Features" (page 39).

The `leaks` tool has some advantages over MallocDebug when it comes to detecting leaks in complex data structures. For example, the `leaks` tool correctly handles leaks in circularly linked structures. It can also identify leaks in groups of connected nodes. MallocDebug may not correctly identify leaks in these types of structures.

Note: The leaks command-line tool is located in `/usr/bin`.

Finding Leaked Autoreleased Objects

If a Cocoa object is autoreleased without an autorelease pool in place, Xcode sends an a message to the console warning you that the object is just leaking. Even if you are not writing a Cocoa application, it is possible to see this same type of console warning. The implementation of many Cocoa classes is based on Core Foundation types. If your application uses Core Foundation, it is possible that the leaks are occurring as a result of calls to that framework.

To find memory leaks of this type, use the debugger to put a breakpoint on the `_NSAutoreleaseNoPool` function. This function is declared in `NSDebug.h` in the Foundation framework. When the debugger reaches that function, you should be able to look at the stack crawl and see what piece of code caused the leak.

Tips for Improving Leak Detection

The following guidelines can help you find memory leaks quickly in your program. Most of these guidelines are intended to be used with the `leaks` tool but some are also applicable for use with MallocDebug.

- Run `leaks` during unit testing. Because unit testing exercises all code paths in a repeatable manner, you are more likely to find leaks than you would be if you were testing your code in a production environment.
- Enable the `MallocScribble` and `MallocPreScribble` environment variables before running your leak tests. For more information, see "Enabling the Malloc Debugging Features" (page 39).
- Use the `-exclude` option of `leaks` to filter out leaks in functions with known memory leaks. This option helps reduce the amount of extraneous information reported by `leaks`.
- If `leaks` reports a leak intermittently, set up a loop around the target code path and run the code hundreds or thousands of times. This increases the likelihood of the leak reappearing more regularly.
- Run your program against `libgmalloc.dylib` in `gdb`. This library is an aggressive debugging malloc library that can help track down insidious bugs in your code. For more information, see the `libgmalloc` man page.
- For Cocoa applications, if you fix a leak and your program starts crashing, your code is probably trying to use an already-freed object or memory buffer. Set the `NSZombieEnabled` environment variable to 1 to find messages to already freed objects.

Most unit testing code executes the desired code paths and exits. Although this is perfectly normal for unit testing, it creates a problem for the `leaks` tool, which needs time to analyze the process memory space. To fix this problem, you should make sure your unit-testing code does not exit immediately upon completing its tests. You can do this by putting the process to sleep indefinitely instead of exiting normally.

Enabling the Malloc Debugging Features

The malloc library provides debugging features to help you track down memory smashing bugs, heap corruption, references to freed memory, and buffer overruns. You enable these debugging options through a set of environment variables. With the exception of `MallocCheckHeapStart` and `MallocCheckHeapEach`, the value for most of these environment variables is ignored. To disable a variable from Terminal, use the `unsetenv` command. Table 1 lists some of the key environment variables and describes their basic function. For a complete list of variables, see the `malloc` man page.

Table 1 Malloc environment variables

Variable	Description
`MallocStackLogging`	If set, `malloc` remembers the function call stack at the time of each allocation.
`MallocStackLogging-NoCompact`	This option is similar to `MallocStackLogging` but makes sure that all allocations are logged, no matter how small or how short lived the buffer may be.
`MallocScribble`	If set, `free` sets each byte of every released block to the value `0x55`.
`MallocPreScribble`	If set, `malloc` sets each byte of a newly allocated block to the value `0xAA`. This increases the likelihood that a program making assumptions about freshly allocated memory fails.
`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.
`MallocDoNotProtect-Postlude`	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 variable to the number of allocations before `malloc` will begin validating the heap. If not set, `malloc` does not validate the heap.
`MallocCheckHeapEach`	Set this variable to the number of allocations before `malloc` should validate the heap. If not set, `malloc` does not validate the heap.

The following example enables stack logging and heap checking before running an application. The value for `MallocCheckHeapStart` is set to 1 but is irrelevant and can be set to any value you want. You could also set these variables from you shell's startup file.

```
% setenv MallocStackLogging 1
% setenv MallocCheckHeapStart 1000
% setenv MallocCheckHeapEach 100
% ./my_tool
```

If you want to run your program in `gdb`, you can set environment variables from the Xcode debugging console using the command `set env`, as shown in the following example:

```
% gdb
(gdb) set env MallocStackLogging 1
(gdb) run
```

Some of the performance tools require these options to be set in order to gather their data. For example, the `malloc_history` tool can identify the allocation site of specific blocks if the `MallocStackLogging` flag is set. This tool can also describe the blocks previously allocated at an address if the `MallocStackLoggingNoCompact` environment variable is set. The `leaks` command line tool will name the allocation site of a leaked buffer if `MallocStackLogging` is set. See the man pages for `leaks` and `malloc_history` for more details.

Detecting Double Freed Memory

The malloc library reports attempts to call `free` on a buffer that has already been freed. If you have set the `MallocStackLoggingNoCompact` option set, you can use the logged stack information to find out where in your code the second `free` call was made. You can then use this information to set up an appropriate breakpoint in the debugger and track down the problem.

The malloc library reports information to `stderr`.

Detecting Heap Corruption

To enable heap checking, assign values to the `MallocCheckHeapStart` and `MallocCheckHeapEach` environment variables. You must set both of these variables to enable heap checking. The `MallocCheckHeapStart` variable tells the malloc library how many `malloc` calls to process before initiating the first heap check. Set the second to the number of `malloc` calls to process between heap checks.

The `MallocCheckHeapStart` variable is useful when the heap corruption occurs at a predictable time. Once it hits the appropriate start point, the malloc library starts logging allocation messages to the Terminal window. You can watch the number of allocations and use that information to determine approximately where the heap is being corrupted. Adjust the values for `MallocCheckHeapStart` and `MallocCheckHeapEach` as necessary to narrow down the actual point of corruption.

Detecting Memory Smashing Bugs

To find memory smashing bugs, enable the `MallocScribble` variable. This variable writes invalid data to freed memory blocks, the execution of which causes an exception to occur. When using this variable, you should also set the `MallocStackLogging` and `MallocStackLoggingNoCompact` variables to log the location of the exception. When the exception occurs, you can then use the `malloc_history` command to track down the code that allocated the memory block. You can then use this information to track through your code and look for any lingering pointers to this block.

Viewing Virtual Memory Usage

Mac OS X provides the `top`, `vm_stat`, and `vmmap` command-line utilities for viewing statistics about virtual memory usage. The information returned by these tools ranges from summary information about all the system processes to detailed information about a specific process.

For information on how to use the `top` tool, please see *Performance Overview* in Performance Documentation. The sections that follow provide information on using `vmmap` and `vm_stat`.

Viewing Virtual Memory Statistics

The `vm_stat` tool displays high-level statistics about the current virtual memory usage of the system. By default, `vm_stat` displays these statistics once, but you can specify an interval value (in seconds) to update these statistics continuously. For information on the usage of this tool, see the man pages.

Listing 1 shows an example of the output from `vm_stat`.

Listing 1 Output of vm_stat tool

```
Mach Virtual Memory Statistics: (page size of 4096 bytes)
Pages free:                     3194.
Pages active:                  34594.
Pages inactive:                17870.
Pages wired down:               9878.
"Translation faults":        6333197.
Pages copy-on-write:           81385.
Pages zero filled:           3180051.
Pages reactivated:            343961.
Pageins:                       33043.
Pageouts:                      78496.
Object cache: 66227 hits of 96952 lookups (68% hit rate)
```

Viewing Mach-O Code Pages

The `pagestuff`tool 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.

Listing 2 shows part of the output from `pagestuff` for the TextEdit application. This output is the result of running the tool with the `-a` option, which prints information about all of the executable's code pages. It includes the virtual address locations of each page and the type of information on that page.

Listing 2 Partial output of pagestuff tool

```
File Page 0 contains Mach-O headers
File Page 1 contains Mach-O headers
File Page 2 contains contents of section (__TEXT,__text)
Symbols on file page 2 virtual address 0x3a08 to 0x4000
File Page 3 contains contents of section (__TEXT,__text)
Symbols on file page 3 virtual address 0x4000 to 0x5000
File Page 4 contains contents of section (__TEXT,__text)
Symbols on file page 4 virtual address 0x5000 to 0x6000

...

File Page 22 contains contents of section (__TEXT,__cstring)
File Page 22 contains contents of section (__TEXT,__literal4)
File Page 22 contains contents of section (__TEXT,__literal8)
File Page 22 contains contents of section (__TEXT,__const)
Symbols on file page 22 virtual address 0x17000 to 0x17ffc
File Page 23 contains contents of section (__DATA,__data)
File Page 23 contains contents of section (__DATA,__la_symbol_ptr)
File Page 23 contains contents of section (__DATA,__nl_symbol_ptr)
File Page 23 contains contents of section (__DATA,__dyld)
File Page 23 contains contents of section (__DATA,__cfstring)
File Page 23 contains contents of section (__DATA,__bss)
File Page 23 contains contents of section (__DATA,__common)
Symbols on file page 23 virtual address 0x18000 to 0x18d48
 0x00018000 _NXArgc
 0x00018004 _NXArgv
 0x00018008 _environ
 0x0001800c ___progname
...
```

In the preceding listing, if a page exports any symbols, those symbols are also displayed by the `-a` option. If you want to view the symbols for a single page, pass in the desired page number instead of the `-a` option.

Viewing Virtual Memory Regions

The `vmmap` tool displays the virtual memory regions allocated for a specified process. This tool can help a programmer understand the purpose of memory at a given address and how that memory is being used. For each virtual-memory region, `vmmap` displays the type of page, the starting address, region size (in kilobytes), read/write permissions, sharing mode, and the purpose of the pages in that region.

Sample Output From vmmap

Listing 3 (page 44) shows some sample output from the `vmmap` tool. This example is not a full listing of the tool's output but is an abbreviated version showing the primary sections.

Listing 3 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/w/Clock
                    90000 [   4K] r--/r-- SM=SHM
                   340000 [3228K] r--/rwx SM=COW 00000100 00320...
                   789000 [3228K] r--/rwx SM=COW 00000100 00320...
Submap           90000000-9fffffff r--/r-- machine-wide submap
__TEXT           90000000  [ 932K] r-x/r-x SM=COW /usr/lib/libSystem.B.dylib
__LINKEDIT       900e9000   [ 260K] r--/r-- SM=COW /usr/lib/libSystem.B.dylib
__TEXT           90130000 [ 740K] r-x/r-x SM=COW .../Versions/A/CoreFoundation
__LINKEDIT       901e9000 [ 188K] r--/r-- SM=COW .../Versions/A/CoreFoundation
__TEXT           90220000 [2144K] r-x/r-x SM=COW .../Versions/A/CarbonCore
__LINKEDIT       90438000 [ 296K] r--/r-- SM=COW .../Versions/A/CarbonCore

[...data omitted...]

==== Writable regions for process 606
__DATA             18000 [   4K] rw-/rwx SM=PRV /Contents/MacOS/TextEdit
__OBJC             19000 [   8K] rw-/rwx SM=COW /Contents/MacOS/TextEdit
MALLOC_OTHER       1d000 [ 256K] rw-/rwx SM=PRV
MALLOC_USED(DefaultMallocZone_0x5d2c0)     5d000 [ 256K] rw-/rwx SM=PRV
                   9d000 [ 372K] rw-/rwx SM=COW 33320000 00000020 00000000 00001b84...
VALLOC_USED(DefaultMallocZone_0x5d2c0)     ff000 [  36K] rw-/rwx SM=PRV
MALLOC_USED(CoreGraphicsDefaultZone_0x10  108000 [ 256K] rw-/rwx SM=PRV
MALLOC_USED(CoreGraphicsRegionZone_0x148  148000 [ 256K] rw-/rwx SM=PRV

[...data omitted...]

Submap           a000b000-a012ffff r--/r-- process-only submap
__DATA           a0130000 [  28K] rw-/rw- SM=COW .../Versions/A/CoreFoundation
Submap           a0137000-a021ffff r--/r-- process-only submap
__DATA           a0220000 [  20K] rw-/rw- SM=COW .../Versions/A/CarbonCore
Submap           a0225000-a048ffff r--/r-- process-only submap
__DATA           a0490000 [  12K] rw-/rw- SM=COW .../IOKit.framework/Versions/A/IOKit
Submap           a0493000-a050ffff r--/r-- process-only submap
__DATA           a0510000 [  36K] rw-/rw- SM=COW .../Versions/A/OSServices
                 b959e000 [   4K] rw-/rw- SM=SHM
                 b95a0000 [   4K] rw-/rw- SM=SHM
                 b9630000 [ 164K] rw-/rw- SM=SHM
                 b965a000 [ 896K] rw-/rw- SM=SHM
                 bff80000 [ 504K] rw-/rwx SM=ZER
STACK[0]         bfffe000 [   4K] rw-/rwx SM=PRV
                 bffff000 [   4K] rw-/rwx SM=PRV
__DATA           c000c000 [   4K] rw-/rwx SM=PRV .../Versions/A/ApplicationEnhancer
STACK[1]         f0001000 [ 512K] rw-/rwx SM=PRV
                 ff002000 [12272K] rw-/rw- SM=SHM

==== 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%)
```

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

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 1 (page 46) describes these columns.

Table 1 Column descriptions for vmmap

Column Number	Example	Description
1	`__TEXT`, `__LINKEDIT`, `MALLOC_USED`, `STACK`, and so on	The purpose of the memory. This column can contain the name of a Mach-O segment or the memory allocation technique.
2	`(DefaultMallocZone_-0x5d2c0)`	If present, the zone used for allocation.
3	`4eee000`	The virtual memory address of the region.
4	`[ 124K]`	The size of the region, measured in kilobytes
5	`rw-/rwx`	Read, write and execution permissions for the region. The first set of flags specifies the current protection for the region. The second set of values specifies the maximum protection for the region. If an entry contains a dash (`-`), the process does not have the target permission.
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.

Column 1 identifies the purpose of the memory. A `__TEXT` segment contains read-only code and data. A `__DATA` segment contains data that may be both readable and writable. For allocated data, this column shows how the memory was allocated, such as on the stack, using `malloc`, and so on. For regions loaded from a library, the far right column shows the name of the library loaded into memory.

The size of the virtual memory region (column 4) represents the total size reserved for that region. This number may not reflect the actual number of memory pages allocated for the region. For example, calling `vm_allocate` reserves a set of memory pages but does not allocate any physical memory until the pages are actually touched. Similarly, a memory-mapped file may reserve a set of pages, but the system does not load pages until a read or write event occurs on the file.

The protection mode (column 5) describes the access restrictions for the memory region. A memory region contains separate flags for read, write, and execution permissions. Each virtual memory region has a current permission, and a maximum permission. In the output from `vmmap`, the current permission appears first followed by the maximum permission. Thus, if the permissions are "`r--/rwx`" the page is currently read-only but allows read, write, and execution access as its maximum allowed permissions. Typically, the current permissions do not permit writing to a region. However, these permissions may change under certain circumstances. For example, a debugger may request write access to a page in order to set a breakpoint.

Note: Pages representing part of a Mach-O executable 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 cause the application to crash immediately.

The sharing mode (`SM=` field) tells you whether pages are shared between processes and what happens when pages are modified. Private pages (`PRV`) are visible only to the process and are allocated as they are used. Private pages can also 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. However, the region containing those private pages is still copy-on-write until all pages become private. Once all pages are private, the sharing mode changes to private.

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. For example, the memory between `0x90000000` and `0xAFFFFFFF` 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 be 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.

Document Revision History

This table describes the changes to *Memory Usage Performance Guidelines*.

Date	Notes
2006-06-28	Clarified where to get the leaks tool.
2005-07-07	Updated information related to using libMallocDebug and malloc zones.
2005-04-29	Fixed some minor bugs. Added new sections on batch allocation of memory and finding leaks of autoreleased objects.
	Added tips for detecting leaks more quickly.
	Document title changed. Old title was *Memory Performance*.
2003-07-25	Added Carbon-specific performance tips.
2003-05-15	First revision of this programming topic. Some of the information appeared in the document *Inside Mac OS X: Performance*.

Index