

## Chapter 11

# Command-Line Debugging Tools

Over the years Unix programmers have created many command-line tools to help them find problems with their code. Because Mac OS X has Unix at its core, you have access to all these tools as well. I do not recommend starting with the command-line tools. You will find it much easier initially to use Xcode's debugger or Instruments to find problems in your code. When you need to dig deeper to find a nasty bug, try the tools in this chapter.

## A Command Line Primer

For those of you who have used only graphical user interfaces, the Unix command line in Mac OS X may seem intimidating, but this section introduces you to enough of the Unix environment for you to use the tools covered in this chapter. To reach the command line, run the Terminal application. It should be in your Applications folder under Utilities. After launching Terminal a Unix shell window opens, and you can start working in the command line.

## Executing Commands as root

Several of the tools I cover in this chapter require you to run them as the `root` superuser, but you don't have to switch to `root`. The `sudo` command allows you to execute any Unix command as though you were `root`. Supply the command you want to execute or the program you want to run.

```
sudo Command
```

After running the `sudo` command you will be prompted for a password. Enter the password for your user name. The command executes after you enter your password.

## Navigating Directories

If your Mac is like mine, it has thousands of directories (folders) for applications, source code, software development kits, and programming documentation. Navigating these directories is more difficult to do from the command line than from the Finder. Two UNIX commands help you navigate your computer's directories: `ls` and `cd`.

## 2 Chapter 11: Command-Line Debugging Tools

The `ls` command lists the files and subdirectories inside the current directory. Without the `ls` command you would have to memorize the contents of your computer's folders to do any navigation.

The `cd` command changes the current directory. To change to a subdirectory of the current directory, enter the subdirectory name. Suppose the current directory has a subdirectory `Games` that holds your favorite games. If you wanted to move to the `Games` directory, you would type.

```
cd Games
```

To move through several levels of subdirectories at once, use slashes to separate the subdirectories. Suppose you wanted to move to the directory where you have the game `Halo`. You could execute the `cd` command twice, once to move to the `Games` directory and once to move to the `Halo` directory. Or you could move directly to the `Halo` directory with one `cd` command.

```
cd Games/Halo
```

To move back one directory, use `..` as the directory name. If you were in the `Halo` directory and wanted to move back to the `Games` directory, you would type.

```
cd ..
```

If you have a leading slash in the directory name, the directory moves relative to the hard disk on which you installed Mac OS X. To move to your `Developer` directory where the `Xcode Tools` reside, you would type.

```
cd /Developer
```

If one of your directories has a space in it, put the directory name in quotes.

```
cd "Source Code"
```

Without the quotes, the computer treats the directory name `Source Code` as two separate arguments, and you will get an error message "Too many arguments".

## Getting Help

If you get stuck in the command line, Unix manual pages are your friends. There are manual pages for virtually every Unix system command and command-line tool. They describe what the command (or tool) does, the arguments it takes, and the available options. To read a manual page, use the `man` command.

```
man CommandName
```

To learn about the options available for the `ls` command, type.

```
man ls
```

Most manual pages are too large to fit in a shell window so only the first part of the manual page appears in your window. The bottom line of your window will have a colon with the cursor next to it to tell you there's more text to read. To see the next line of text, press the down arrow key or the Return key. To see the next page (the amount of text that will fit in the window) of text, press the space bar. Press the up arrow key to see previously viewed parts of the manual page that are no longer visible in the window.

## NOTE

You can read Unix manual pages in Xcode. Choose Help > Open man page.

## Finding Your Application's Process ID

Some of the tools in this chapter require you to supply your application's process ID. Every running application in Mac OS X has a process ID, which is a number that uniquely identifies the application to the operating system. You have no way of knowing your application's process ID until you launch it.

To find your program's process ID, open a new shell window by choosing File > New Shell. Run the `top` tool by typing the word `top`. `top` lists all the currently running processes with information about each process, such as how much CPU time and memory they are using. When you run `top`, look at the left side of the shell window to see a column with the heading PID. This column lists each running application's process ID. Find your application's process ID and be ready to use it when running the debugging tools in this chapter.

## fs\_usage

The `fs_usage` tool presents a real-time listing of file system calls and page faults. Before running `fs_usage`, make your shell window as wide as possible; the wider the window, the more information `fs_usage` returns. You must use the `sudo` command to run `fs_usage`.

## Reporting File Manager Routines

Cocoa and Carbon programs use the File Manager to work with files. The File Manager functions call the low-level file system routines that `fs_usage` reports. If you use Cocoa or Carbon in your program, you want `fs_usage` to report the File Manager function calls as well as the low-level routines. Use the debug runtime library in your program to display the File Manager functions your program calls. To tell Xcode to use the debug runtime library, run Xcode and open your project.

1. Select Executables from the Groups and Files list. The list of executable files appears in the window.
2. Select the executable name of the program you want to profile.
3. Click the Info button to open the inspector for the executable file.
4. Click the General tab.
5. You should see a pop-up menu with the label Use suffix when loading frameworks. Choose debug from the menu.
6. Clean your project by choosing Build > Clean.
7. Build your project by choosing Build > Build.

## Running `fs_usage`

If you supply no arguments to `fs_usage`,

```
sudo fs_usage
```

It displays information on all running processes. Running `fs_usage` on all running processes is usually a bad idea. `fs_usage` displays each file system call as it occurs, and Mac OS X applications make a surprisingly large number of file system calls, making it hard to keep up with all the information `fs_usage` reports. In most cases the only process you're interested in is your program. To limit `fs_usage`'s reporting to your application, supply either the application's process ID or its name.

```
sudo fs_usage ProcessID  
sudo fs_usage AppTitle
```

When you run `fs_usage` on your application, the shell window seems to lock up with no data appearing in it. The lack of activity reflects the fact that you're in the Terminal application, which means your application isn't doing anything at the moment. Switch back to your application, do some things in it, then switch back to Terminal. Your shell window should be packed with data from `fs_usage`.

## What `fs_usage` Tells You

Assuming you made your window wide enough, `fs_usage` provides the following data for each system call:

- A timestamp of when the system call occurred.
- The name of the system call. Table 11.1 lists the most common system calls.
- Additional information that depends on the nature of the system call. Make sure your shell window is wide or the additional information will not appear. Look at Table 11.2 for the types of additional information `fs_usage` provides.
- The pathname of the file the system call accessed.
- The amount of time, in seconds, the program spent in the system call. If the listing has a W next to it, the amount of time includes the time spent waiting for a file operation to finish.
- The name of the process that made the system call. The process name is useful only if you're viewing multiple processes with `fs_usage`.

If you use the debug runtime library for your program, `fs_usage` lists the high-level File Manager function calls Cocoa and Carbon programs use to access files. For the File Manager function calls, `fs_usage` displays the parameters passed to each function instead of the extra information in Table 11.2.

**Table 11.1 Common System Calls**

System Call	Description
CACHE_HIT	The computer found the data it needs inside one of its caches. Cache hits are good because the computer doesn't have to go to RAM or disk to retrieve the data.
PAGE_IN	The operating system moved data from disk to memory.
PAGE_OUT	The operating system moved data from memory to disk.
read	Read data from a file.
write	Write data to a file.
open	Open a file.
close	Close a file.
lseek	Move to a particular place in a file.
fstat	Retrieve information about an open file such as its size, the last time it was accessed, and the last time its contents changed.
stat	Retrieves the same information as <code>fstat</code> , but the file does not have to be open.

<code>lstat</code>	Retrieves the same information as <code>fstat</code> , but the file does not have to be open. If the file refers to a symbolic link, <code>lstat</code> returns information about the link, not the file to which the link refers.
<code>getattrlist</code>	Returns a list of attributes for a file system object such as a volume, directory, file, or file fork. Some information <code>getattrlist</code> returns for a file are its name, its size, the number of forks, and the size of each fork. Mac files have two forks: a data fork and a resource fork.
<code>getdirentries</code>	Returns information about the files and subdirectories inside a given directory.
<code>getdirentriesattr</code>	Returns a list of attributes for multiple directories. It combines the work of <code>getattrlist</code> and <code>getdirentries</code> .
<code>mmap</code>	Maps a file into memory.

**Table 11.2 Extra Information `fs_usage` Provides**

Information	Prefix	Description
Address	A=	For cache hits, page ins, and page outs, the address tells you where the cache hit, page in, or page out occurred in memory.
Bytes	B=	For file reads and writes, the number of bytes the operating system read from or wrote to the file. For page ins and page outs, the number of bytes that were paged in from disk or paged out to disk. <code>fs_usage</code> reports the number of bytes as a hexadecimal number.
Disk Block Number	D=	For file reads and writes, the physical disk block number being read from or written to.
File Descriptor	F=	An integer that identifies an open file. <code>fs_usage</code> reports a file descriptor for most of the file operations in Table 11.1.
Offset	O=	For <code>lseek</code> operations, the offset tells you how far the operating system moved from the start of the file.
Select Return	S=	The return value of the select system call. A value of 0 means the select timed out.
Error Number	[ ]	If a file operation produced an error, the error number appears in brackets.

## fs\_usage Options

The `fs_usage` tool provides three options. You can use multiple options in one command. The following command:

```
fs_usage -w -f network AppTitle
```

Combines the `-w` and `-f` options to limit `fs_usage`'s output to network system calls and tells `fs_usage` to wrap its output to a second line if the window isn't wide enough to display all the output on one line.

### -e Option

The `-e` option allows you to exclude certain processes from the `fs_usage` report. Supply a list of process ID numbers or application names.

```
fs_usage -e 449 465 508
```

On Mac OS X the `-e` option does not help much because there are too many processes running. As a test, I started up my Mac, launched the Terminal application, and ran `top`. `top` reported 39 running processes when I had only two applications running: the Finder and Terminal. To look at a few applications, supply the process ID numbers of the applications you're interested in viewing instead of using the `-e` option with every process you don't want to see.

### -f Option

Normally `fs_usage` displays every system call it finds. The `-f` option filters `fs_usage`'s output. Supply one of three filtering modes: network, file system, and cache hit. With the network filtering mode,

```
fs_usage -f network AppTitle
```

`fs_usage` displays only network system calls. With the file system filtering mode,

```
fs_usage -f filesys AppTitle
```

`fs_usage` displays only file system calls.

If you want `fs_usage` to report cache hits, you must use the `-f` option and cache hit mode.

```
fs_usage -f cachehit AppTitle
```

## -w Option

`fs_usage` usually supplies as many columns of data as will fit in your shell window; the wider the window, the more columns of data `fs_usage` shows. When you use the `-w` option, `fs_usage` displays all the available information, wrapping the output to the next line if it won't fit on one line. The `-w` option saves you from having to resize the window to see all of `fs_usage`'s output.

```
fs_usage -w AppTitle
```

## sc\_usage

The `sc_usage` tool maintains a running list of system calls and page faults for a given application. You must use the `sudo` command to run `sc_usage`. Supply either a process ID or the name of a currently running application to `sc_usage`.

```
sudo sc_usage ProcessID
sudo sc_usage AppTitle
```

When you run `sc_usage` on your application, the chances are high no data will appear initially because you're in the Terminal application instead of your application. Switch to your application and do some things in it to see some of the data `sc_usage` records.

## What sc\_usage Tells You

`sc_usage` provides its output differently than `fs_usage`. `fs_usage` has one listing for each individual event while `sc_usage` has one listing for each type of system call along with the number of calls. If your application has 25 cache hits, `fs_usage` displays 25 `CACHE_HIT` listings while `sc_usage` displays one `cache_hit` listing with a count of 25.

By default `sc_usage` updates its data every second, replacing the old information with the new. This behavior differs from `fs_usage`, which adds each system call to the output immediately. The `sc_usage` output is more useful to view while your program's running rather than saving and viewing later.

`sc_usage` provides its output in three areas. At the top is summary information about the program being monitored. In the center is a list of the system calls the program made along with information about the calls. At the bottom is a list of blocked system calls along with information about each call.



## Program Summary Information

At the start of the report is summary information about the program being monitored. `sc_usage` reports eight pieces of summary information.

- The program's name.
- The number of preemptions. A preemption occurs when the operating system interrupts your program to give time to another program. Preemptions are good as a Mac user. They allow you to simultaneously write source code in Xcode, download a file in Safari, and listen to a CD in iTunes.
- The number of page faults. A page fault occurs when the operating system can't find a page of memory in physical memory.
- The number of context switches. A context switch occurs when the operating system switches to another program. A major context switch occurs when you switch applications. If you're in Safari and switch to iTunes, a major context switch changes the foreground process from Safari to iTunes. A minor context switch occurs when the operating system gives time to a program running in the background. If you're surfing the Internet on Safari and listening to music with iTunes, a minor context switch gives iTunes the time it needs to play the music you're listening to.
- The number of system calls your program made.
- The number of threads in the program.
- The current time. It's always important to know what time it is.
- The elapsed time, the amount of time `sc_usage` has been monitoring your program.

When you look at the summary information, remember that the numbers of preemptions, page faults, context switches, and system calls `sc_usage` reports are the numbers that occurred during the last sampling period. If you run your program and switch to `sc_usage`, you could see your preemptions, page faults, context switches, and system calls trickle down to zero. The trickling occurs because `sc_usage`'s default sampling period is one second. If you're examining `sc_usage`'s results, your application won't have much activity in the previous second.

## System Call List

After the program summary information is a list of each system call your application makes. `sc_usage` tells you the following information for each system call:

- The system call name.
- The number of times your program made the system call.
- The amount of time your program spent in the system call.
- The amount of time the call has been waiting.

When reporting the number of times your program made a system call and the amount of time the call has been waiting, there may be two values, one of which is in parentheses.

```
500 (41)
```

The listing says your program made this system call 500 times since launching `sc_usage`. 41 of the 500 calls occurred during the last sampling period. The first three listings in the system call list are.

- System Idle, which tells you the amount of time the operating system is idle.
- System Busy, which tells you the amount of time the operating system is busy.
- Usermode, which tells you the amount of time spent in your program.

After the System Idle, System Busy, and Usermode listings comes the page fault system calls. Table 11.3 lists these calls.

After the page fault system call listings come the rest of the system call listings. You can see some common system calls in Table 11.1.

**Table 11.3 Page Fault Types**

Page Fault Type	Description
cache_hit	The operating system found the page of memory in the computer's cache.
page_in	The operating system moved the page from disk to physical memory.
zero_fill	The operating system created the page of memory and filled it with zeroes.
copy_on_write	The operating system copied the memory page from another page of memory.

## sc\_usage Options

`sc_usage` has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
sudo sc_usage -l -s 10 AppTitle
```

Combines the `-l` and `-s` options. It tells `sc_usage` to turn off continuous updating of data and to use a sampling interval of ten seconds.

## -c Option

The `-c` option lets you specify a code file that contains the system calls you want `sc_usage` to report your program making. You can view the default code file at `/usr/share/misc/trace.codes`. Supply the path to the code file when using the `-c` option.

```
sudo sc_usage -c CodeFile AppTitle
```

## -e Option

The `-e` option sorts the output data by the call count, the number of times your application made the system calls. By default `sc_usage` sorts the listings by the amount of time your program spent in each system call.

```
sudo sc_usage -e AppTitle
```

`sc_usage` sorts the system calls on a first come, first served basis. When your program makes a system call for the first time, the call appears at the bottom of the list. If your program makes multiple system calls for the first time in a given sampling period, `sc_usage` sorts the calls based on the number of times your program made them, and this order never changes. Suppose you have two system calls, A and B. If during the first sampling period, your program calls A 10 times and B 20 times, B appears ahead of A. If your program calls A 100 times the next period and doesn't call B, B still appears ahead of A even though A has been called more than B.

## -E Option

Running `sc_usage` with the `-E` option launches your program first. Use the `-E` option when you want to see the system calls your program makes when it starts. Supply a path to your program's executable file and any runtime arguments your program needs to run. For Mac OS X application bundles, the executable file resides three directories inside the application bundle.

```
> Application Bundle
  > Contents
    > MacOS
      > Executable File
```

If you're in the directory where the application bundle resides, you would launch the program with the following command:

```
sudo sc_usage -E AppTitle.app/Contents/MacOS/AppTitle
```

## -l Option

The `-l` option turns off the continuous updating of data. Every time `sc_usage` updates its data, it appends the output to the shell window. If you use the default sampling rate of one second and watch your program for one minute, you would end up with 60 reports in the shell window. Use the `-l` option if you want a record of all system calls your application makes.

## -s Option

The `-s` option lets you specify how often you want `sc_usage` to update its output. Supply the number of seconds you want in the sampling period; the default period is one second. The following command tells `sc_usage` to update its output every five seconds:

```
sudo sc_usage -s 5 AppTitle
```

## vmmap

The `vmmap` tool gives you a map of the virtual memory the operating system reserved for your program. It's the command-line equivalent of the VM Tracker instrument in Instruments. Supply a process ID number or program name when running `vmmap`.

```
vmmap ProcessID  
vmmap AppTitle
```

## What vmmap Tells You

The report `vmmap` generates has three sections.

- The non-writable memory regions, which consist mostly of shared libraries to which you linked your application. Shared libraries being non-writable is a good thing. If you write an application that uses QuickTime, you don't want your application to be able to overwrite the memory QuickTime is using.
- The writable memory regions.
- A summary report of the virtual memory map.

When you look at the report, keep in mind that it's telling you about the memory the operating system reserved for your application. It does not necessarily mean your application is using all of that memory. When the operating system launches your program, it reserves a large chunk of memory, more than most programs need. From this large chunk of reserved

memory, the operating system allocates smaller chunks when your application needs them. The `vmmap` report gives you a map of the initial large chunk of reserved memory. You have to do some digging to discover how much of the reserved memory your program uses.

## Non-Writable Memory Regions

The report for the non-writable memory regions your application reserves has one listing for each memory region. `vmmap` tells you the following information for each region:

- The purpose of the memory in the region.
- The starting and ending addresses of the memory region. If you're debugging your program, you can look at the starting address in the debugger to view the memory contents.
- The size of the region. It appears in brackets.
- The permissions for the memory region.
- The sharing mode of the memory region.
- Additional data that depends on the memory region. Some regions have no additional data to display. For some regions `vmmap` shows the pathname of the executable file for the memory region. Most of these pathnames will be libraries linked to your program. For some regions `vmmap` shows the contents of the memory in the region. The amount of data that appears in the window depends on the width of your window. Wider windows reveal more memory contents and a larger portion of pathnames.

### Region Purpose

The region purpose of a memory region gives you a general idea of the region's contents. The first non-writable memory region for every application has the name `__PAGEZERO` as its purpose. This region is protected; accessing it will crash your program.

Non-writable memory regions have six common purposes.

- `__TEXT` regions contain executable code or constant data like the constants your application declares.
- `__DATA` regions contain data, which comes as a shock to you. The data is read-only for a non-writable memory region, but a writable memory region would contain data that your application could both read and write.
- `__LINKEDIT` regions contain raw data the dynamic linker uses, such as symbol table entries. These regions generally follow `__TEXT` regions.
- `__OBJC` regions contain data that the Objective-C runtime library uses. Only Objective-C programs have these regions.

- Shared memory regions are shared by multiple applications. System libraries like Cocoa and Carbon are the major source of shared memory regions.
- Mapped file regions are memory mapped files, where the operating system maps part of a file into a program's virtual address space. Memory mapped files let multiple programs read from and write to the same file.

One not so common purpose deserves some explanation. Guarded memory regions, which show up as `STACK GUARD` in a `vmmap` report, prevent out of order accesses. Normally, the computer allows operations to be performed out of order. Suppose your program is currently performing a series of integer calculations, followed by some floating-point calculations. If the CPU performed the operations in order, the floating-point unit would be idle until the series of integer calculations finished. To make use of the idle floating-point unit, the CPU performs the floating-point calculations while it performs the series of integer calculations. The floating-point calculations are out of order operations in this situation. Out of order operations are normally good; they allow your program to run efficiently. Sometimes accessing memory out of order can wreak havoc with your program. Memory that controls I/O devices is especially vulnerable to out of order memory accesses. Making vulnerable memory regions guarded prevents bad things from happening in your program.

### Permissions

The permissions for a memory region tell you how you can access the region. `vmmap` lists two sets of permissions for each memory region. The first set lists the permissions for your application, and the second set lists the maximum permissions. The most common use of maximum permissions is running your program in a debugger. In a non-writable memory region the most common permission is.

`r--/rwx`

Which means your application can read memory in this region, but cannot write to memory or execute memory in this region. Your application will never have write permission in non-writable memory regions. The only way a memory region can have execute permission is if the operating system loaded a piece of executable code in the region. An application with maximum permission can read, write, and execute memory in this memory region. The most common situation to use maximum permissions to write to memory in a non-writable region is a debugger inserting a breakpoint in an application.

If you look at a `vmmap` report, you can see the first listing, `__PAGEZERO`, has six dashes in the permissions area, which means nobody can read, write, or execute memory there. If you've done any work with pointers, you know that accessing null pointers will crash your program. Denying permission to access `__PAGEZERO` ensures a crash if you do anything with a null pointer.

## Sharing Modes

The sharing mode of a memory region tells you how the region interacts with other programs. A memory region can have one of seven possible sharing modes.

- Copy on write (COW) regions can be shared at multiple locations in your application or shared by multiple applications. When your application modifies the memory of a copy on write region, it receives a copy of the region, and the page of memory your application modified becomes private. When all of the pages of a memory region become private, the region becomes private as well.
- Private (PRV) regions are visible only to your application.
- Empty (NUL) regions do not exist in physical memory.
- Aliased (ALI) regions point to another memory region.
- Shared (SHM) regions are shared by multiple applications.
- Zero filled (ZER) regions have had their contents cleared and filled with zeroes.
- Shared alias (S/A) regions combine the characteristics of aliased and shared memory regions.

## Writable Memory Regions

`vmmap` provides the same types of information for writable memory regions as it does for non-writable ones: purpose, starting address, ending address, size, permission, sharing mode, and possible additional data. When you look at the permissions for a writable memory region, you will see most of the listings give your application write access, which is what you expect for a writable memory region. If a writable memory region shows no additional data, the memory in the region is heap memory or stack memory.

Writable memory regions rarely have the `__TEXT` and `__LINKEDIT` purposes that are common in non-writable regions. Common purposes for writable memory regions include.

- `__DATA` regions contain data (surprise) that your application can both read from and write to.
- `MALLOC` regions have been allocated using the function `malloc()`.
- `MALLOC_TINY` regions consist of small memory allocations, allocations 512 bytes and smaller.
- `MALLOC_LARGE` regions consist of large memory allocations. Apple's documentation says large memory allocations are at least 4 pages in size, 16KB. But I saw `MALLOC_LARGE` regions that were 4KB when I ran `vmmap`.
- `MALLOC_FREE` regions were allocated earlier, used, and freed when your program was finished with the memory.
- `VM_ALLOCATE` regions are regions that were allocated with the `vm_allocate()` function. This function allocates virtual memory regions. `malloc()` allocates memory from these virtual memory regions.



- **STACK** regions contain stack memory, which is where the computer places the parameters of every function your application calls.
- **ATS** regions involve the Apple Type Services (ATS) framework, which deals with fonts. If your program works with text, chances are high you'll have ATS memory regions.
- **\_\_OBJC** regions contain data that the Objective-C runtime library uses.
- **\_\_LOCK** regions are locked, which means the operating system can't move them when memory runs low.

You might be wondering where your program calls `malloc()` if you didn't call it in your code. When you use the `new` operator in a C++ program to create an object, you're indirectly calling `malloc()`. When you use Cocoa methods to create things like arrays, strings, and dictionaries, you're calling `malloc()` indirectly.

## Summary Report

For non-writable memory regions, `vmmap`'s summary displays the following data:

- The total amount of memory reserved for your application.
- The amount of resident memory, the memory that is in physical RAM.
- The amount of memory that was either swapped out or unallocated. Swapped-out memory is memory that was in RAM, but moved to disk because other programs needed the RAM.

Remember that shared libraries your application links to comprise most of the read-only memory regions. Other applications use these libraries as well so seeing a high amount of resident memory does not necessarily mean your application is a memory hog.

For writable memory regions, `vmmap`'s summary tells you the following information:

- The total amount of memory reserved for your application.
- The amount of memory the computer wrote in your application.
- The amount of resident memory.
- The amount of swapped-out memory. Swapped-out memory can mean one of two things: you're switching to other applications or your program uses a lot of memory.
- The amount of unallocated memory.

The sum of resident memory and swapped-out memory is the best measurement of the amount of memory your program uses.

At the end of the summary report, `vmmap` breaks down the memory map by region purpose and tells you the total amount of virtual memory for each purpose.



## vmmap Options

`vmmap` has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
vmmap -w -submap ProcessID
```

Tells `vmmap` to print wide output and to print submap information in the output.

### -d Option

When you use the `-d` option, `vmmap` takes two snapshots of your program's virtual memory map. It takes the first snapshot immediately and takes the second one after a period of time that you specify. After taking the snapshots, `vmmap` reports three pieces of information.

- The memory regions that changed in the second snapshot.
- Memory regions in the first snapshot that aren't in the second.
- Memory regions in the second snapshot that aren't in the first.

The `vmmap` report has one difference listing for each memory region that changed between the first snapshot and the second. The difference listing starts with the text `Diff`, then displays the region as both snapshots recorded it. The usual cause of a difference in a memory region is a change in sharing mode.

After showing the memory regions that changed, `vmmap` lists the memory regions that appear in only one of the snapshots. It starts with the regions that appear only in the first snapshot then lists the regions that appear only in the second snapshot. Non-writable memory regions appear before writable ones.

Using the `-d` option is relatively simple. Supply the number of seconds you want `vmmap` to wait before taking the second snapshot. The following example waits 30 seconds:

```
vmmap -d 30 AppTitle
```

### -w Option

The `-w` option displays wide output for each memory region. Wide output allows you to see a greater amount of additional data for each region.

```
vmmap -w AppTitle
```

### **-resident Option**

When you run `vmmap` with the `—resident` option, it lists the virtual and resident size of each memory region. The resident size tells you how much of the region is in physical memory.

```
vmmap —resident AppTitle
```

### **-pages Option**

When you run `vmmap` with the `—pages` option, it lists the size of each memory region in pages instead of bytes. Memory pages on PowerPC Macs are 4KB. The default page size is also 4KB on Intel Macs, but pages can also be 2MB and 4 MB.

```
vmmap —pages ProcessID
```

### **-interleaved Option**

When you run `vmmap` with the `—interleaved` option, the report does not separate the memory regions into writable and non-writable regions. It lists the memory regions in order of their starting address, with the lowest memory regions appearing first.

```
vmmap —interleaved ProcessID
```

### **-submap Option**

The `—submap` option tells `vmmap` to print information about memory submaps. A *submap* is an area of memory the operating system can use in multiple applications. A Cocoa runtime library may reside in a submap so any running Cocoa application can use it.

In a `vmmap` report all listings below a submap listing belong to that particular submap until another submap listing appears.

```
vmmap —submap AppTitle
```

## -allSplitLibs Option

The `—allSplitLibs` option tells `vmmap` to print information about all shared system split libraries. The default behavior is to print information about only the libraries your program loads. *Shared system split libraries* are dynamic libraries that multiple programs share. Examples of these libraries are the Cocoa and Carbon libraries. The `—allSplitLibs` option generates a lot of unused split lib listings for non-writable regions.

```
vmmap -allSplitLibs AppTitle
```

## heap

The `heap` tool lists the objects allocated in your application's heap. Supply a process ID or an application name when running `heap`.

```
heap ProcessID
heap AppTitle
```

## heap's Output

The output `heap` generates comes in three levels. First, `heap` reports a summary for each memory zone. The summary tells you the following information for each zone:

- The size of the zone.
- The number of nodes allocated.
- The amount of memory allocated.
- The largest unused block of memory in the zone.

After reporting each memory zone, `heap`'s summary reports the total number of nodes allocated with the total amount of allocated memory. What does it mean when a node is allocated? It means your program made a memory allocation. One node equals one memory allocation.

Second, `heap` reports the sizes of each memory allocation. For each memory zone `heap` tells you the number of nodes allocated along with the size listings. In each size listing, `heap` reports the size of the memory allocation with the number of allocations in brackets. The following listing:

```
24KB[ 7 ]
```

Says your program made seven memory allocations of 24KB in the memory zone.

`heap` sorts the size listings by allocation size, with the largest allocations coming first. After listing the information for each zone, `heap` reports the same information for all zones combined.

Finally, `heap` reports the objects your program allocated. The report starts with a total number of Objective-C classes and CTypes (Core Foundation types) allocated on the heap for the whole application. Then for each memory zone, it lists the objects that were allocated in that zone. For each class `heap` tells you the following information:

- The number of objects of the class your program allocated.
- The amount of memory allocated.
- The average amount of memory allocated, which is the total amount of memory divided by the number of objects.
- The class name. Memory not part of a class has the class name non-object.
- The type of class. Common class types are ObjC, C++, and CType.
- The binary, which in most cases is the framework or library the class is part of.

`heap` sorts the listings by the number of allocations, with the highest numbers appearing first in the list.

## heap Options

`heap` has several options to customize the output it displays. You can combine multiple options when running `heap`. The following command tells `heap` to use both the `-guessNonObjects` and `-showSizes` options:

```
heap -guessNonObjects -showSizes AppTitle
```

### -guessNonObjects Option

The `-guessNonObjects` option tells `heap` to search the memory of each object for pointers to non-objects, blocks of memory allocated by `malloc()`. The `heap` output has one listing for each of these memory blocks. The block listings show the Objective-C object, with the offset from the start of the object in brackets. The following listing:

```
NSCFArray[ 36 ]
```

References a block of memory that is located 36 entries from the start of the array.

## **-sumObjectFields Option**

Like the `-guessNonObjects` option, the `-sumObjectFields` option tells `heap` to search the memory of each object for pointers to non-objects. When `heap` finds a pointer to a non-object, `heap` adds the size of the non-object to the object's listing.

## **-showSizes Option**

When `heap` lists the objects in a memory zone, it has one listing for each class. The listing for a class contains all the memory allocations of that class. If you use the `-showSizes` option, `heap` creates a listing for each memory allocation size in every class.

Suppose your Cocoa applications allocates strings of length 32, 64, 96, and 128. `heap` normally combines all the string allocations into one listing for a memory zone. If you pass the `-showSizes` option to `heap`, it prints separate listings for the 32, 64, 96, and 128-byte string allocations.

## **-addresses Option**

When you supply the `-addresses` option to `heap`, it prints the address of each object on the heap. Tell `heap` what objects to print. Supplying all tells `heap` to print the address of each object on the heap.

```
heap -addresses all AppDelegate
```

If you don't want to print the address of each object on the heap, you can supply a regular expression that tells `heap` what objects to print. The following command prints the address of every `NSCFDictionary` object found on the heap:

```
heap -addresses NSCFDictionary AppDelegate
```

If your regular expression consists of something more than a single class name, place the expression in single quotes. The following command prints the address of every Cocoa object found on the heap:

```
heap -addresses 'NS.*' AppDelegate
```

## leaks

The `leaks` program checks your application for memory leaks, which is something Instruments can do as well. `leaks` is easier to use for command-line applications.

### Running leaks

You can run `leaks` without switching to `root`. Supply either a process ID or application name.

```
leaks AppTitle
leaks ProcessID
```

Setting the environment variable `MallocStackLogging` to 1 turns on call stack recording, which tells `leaks` to display the call stack for each memory leak. Turning on call stack recording is a good idea. If you have a memory leak in your program, the first thing you want to know is where you're leaking memory so you can fix the leak in your code. By turning on call stack recording, `leaks` can tell you where you're leaking memory. Refer to the section "Setting Environment Variables" in Chapter 5, "Debugging with Xcode", for instructions on setting environment variables.

### What leaks Tells You

`leaks` provides two pieces of information about your program. The first piece of information is a summary that reports the following information:

- The number of nodes your application allocated, which is the number of memory allocations your program made.
- The amount of memory your program allocated.
- The number of memory leaks.
- The total amount of leaked memory.

The second piece of information is a listing for each memory leak that `leaks` found. Each listing reports the following information:

- The address of the leaked memory.
- The size of the leak.
- A memory dump of the leak. For memory leaks 128 bytes and smaller, `leaks` shows the contents of the leaked memory. For memory leaks larger than 128 bytes, `leaks` displays the first 128 bytes of leaked memory.
- If the leak occurs in a Core Foundation or Cocoa class, `leaks` reports the name of the class.
- If you set the `MallocStackLogging` environment variable to 1, `leaks` displays the call stack of functions leading to the memory leak.

## leaks Options

`leaks` has three options you can supply. You can use multiple options. The following command:

```
leaks -nocontext -nostacks
```

Tells `leaks` to conceal the memory dump and call stack in the individual memory leak listings.

### -nocontext Option

The `-nocontext` option tells `leaks` to suppress the memory dump. Use the `-nocontext` option if you don't care about the contents of the memory your program is leaking.

```
leaks -nocontext AppTitle
```

### -nostacks Option

Setting the environment variable `MallocStackLogging` to 1 tells `leaks` to display the call stack for each leak. The `-nostacks` option tells `leaks` to suppress the call stack display. I'm not sure why you would want to suppress the call stack display; the call stack display helps you discover where the memory leaks are in your code.

```
leaks -nostacks ProcessID
```

## **-exclude Option**

When you use the `—exclude` option, `leaks` reports the summary information: the amount of memory allocated, the number of memory leaks, and the total amount of leaked memory. If the function you supply appears in the call stack of a memory leak, `leaks` does not create a listing for that memory leak. Use the `—exclude` option to keep functions you already know leak memory and functions falsely accused of leaking memory from creating unnecessary memory leak listings.

```
leaks —exclude FunctionName ProcessID
```

## **malloc\_history**

As its name suggests, `malloc_history` supplies a history of every memory allocation your application makes using the malloc library. The `ObjectAlloc` instrument in Instruments provides similar information, but `malloc_history` works better with command-line programs and makes saving the results to a text file easier.

To run `malloc_history` on your application, you must turn on the malloc library's debugging capabilities by setting the environment variable `MallocStackLogging` to 1. While you're setting environment variables, you may also want to set the variables `MallocStackLoggingNoCompact` and `MallocScribbling` to 1. Setting the `MallocStackLoggingNoCompact` environment variable allows you to run `malloc_history` on a specific memory address. Setting the `MallocScribbling` environment variable causes the operating system to overwrite memory your application frees. Overwriting freed memory helps you detect memory smashers, places in your code that overwrite memory. Refer to the section "Setting Environment Variables" in Chapter 5, "Debugging with Xcode", for instructions on setting environment variables.

## **Running malloc\_history**

There are two ways to run `malloc_history` on an application.

```
malloc_history ProcessID —all_by_size  
malloc_history ProcessID —all_by_count
```

`malloc_history` displays the memory allocations your application made in the shell window with one listing for each combination of allocation size and call stack. A listing tells you the following information:



- The thread.
- The number of memory allocations.
- The size of each allocation.
- The call stack for the allocation.

The only difference between `-all_by_size` and `-all_by_count` is how `malloc_history` sorts the listings. `-all_by_size` sorts by allocation size, with the largest sizes appearing first while `-all_by_count` sorts by allocation count, with the largest counts appearing first.

## Running `malloc_history` on a Specific Memory Area

`malloc_history` normally lists every memory allocation your program makes. You can tell `malloc_history` to report allocations made in a particular memory area by supplying a memory address to `malloc_history`.

```
malloc_history ProcessID MemoryAddress
```

`malloc_history` provides a history of all memory allocations that manipulated memory at the address you specified. For each memory allocation, `malloc_history` reports the size of the allocation (It says `arg =`), the thread, and the call stack.

To run `malloc_history` on a memory address instead of an entire program, you must set the environment variable `MallocStackLoggingNoCompact` to 1.

## Showing All Allocation Events

If you run `malloc_history` with the `-all_events` option,

```
malloc_history ProcessID -all_events
```

`malloc_history` lists every memory allocation and free event. Using the `-all_events` option gives you much more output than the `-all_by_size` and `-all_by_count` options.