Kernel and Memory Management of MacOS

Shailesh Agrawal
PRN: 22010348
Roll.no: 311006, Division: A

***Abstract*—The lowest layer of OS X includes the kernel, drivers, and BSD portions of the system and is based primarily on open source technologies. OS X extends this low-level environment with several core infrastructure technologies that make it easier for you to develop software and this paper depicts the memory the executives in a working framework and it will show the fundamental design of division in a working framework and essential of its assignment**

I. INTRODUCTION

Mac OS is a series of graphical user interface-based operating systems developed by Apple Inc. (formerly Apple Computer, Inc.) for their Macintosh line of computer systems. Mac OS is credited with popularizing the graphical user interface. This is the operating system that runs on Macintosh computers.The Mac OS (currently OSX) has been around since the first Macintosh was introduced in 1984. Since then, it has been continually updated and many new features have been added to it. Each major OS release is signified by a new number (i.e. Mac OS 8, Mac OS 9). Since the core of the Mac OS was nearly decades old, Apple decided to completely revamp the operating system. In March of 2001, Apple introduced a completely new version of the Mac OS that was written from the ground up. The company dubbed it "Mac OS X," correctly pronounced "Mac OS 10." Unlike earlier versions of the Mac OS, Mac OS X is based on the same kernel as Unix and has many advanced administrative features and utilities. Though the operating system is much more advanced than earlier versions of the Mac OS, it still has the same ease-of-use that people have come to expect from Apple software.

Versions of Mac OS can be divided into two families:

- The Mac OS Classic family, which was based on Apple's own code.
- The OS X operating system, developed from Mac OS Classic family, and NeXTSTEP, which was UNIX based.

MacOS has supported three major processor architectures, beginning with PowerPC-based Macs in 1999.In 2006, Apple transitioned to the Intel architecture with a line of Macs using Intel Core processors. In 2020, Apple began the Apple silicon transition, using self-designed, 64-bit ARM-based Apple M1 processors on the latest Macintosh computers. The current version of macOS brought a number of new capabilities to provide a more stable and reliable platform than its predecessor, the classic MacOS.For example, pre-emptive multitasking and memory protection improved the system's ability to run multiple applications simultaneously without them interrupting or corrupting each other.

II. MEMORY MANAGMENT

Mac OS X include a fully-integrated virtual memory system that you cannot turn off; it is always on. Both systems also provide up to 4 gigabytes of addressable space per 32-bit process. In addition, OS X provides approximately 18 exabytes of addressable space for 64-bit processes. Even for computers that have 4 or more gigabytes of RAM available, the system rarely dedicates this much RAM to a single process.

To give processes access to their entire 4 gigabyte or 18 exabyte address space, OS X uses the hard disk to hold data that is not currently in use. As memory gets full, sections of memory that are not being used are written to disk to make room for data that is needed now. The portion of the disk that stores the unused data is known as the backing store because it provides the backup storage for main memory.

A. Virtual Memory

Virtual memory allows an operating system to escape the limitations of physical RAM. The virtual memory manager creates a logical address space (or "virtual" address space) for each process and divides it up into uniformly-sized chunks of memory called *pages*. The processor and its memory management unit (MMU) maintain a *page table* to map pages in the program's logical address space to hardware addresses in the computer's RAM. When a program's code accesses an address in memory, the MMU uses the page table to translate the specified logical address into the actual hardware memory address. This translation occurs automatically and is transparent to the running application.

In OS X, the virtual memory system often writes pages to the backing store. The *backing store* is a disk-based repository containing a copy of the memory pages used by a given process. Moving data from physical memory to the backing store is called *paging out* (or "swapping out"); moving data from the backing store back in to physical memory is called *paging in* (or "swapping in").

In OS X and in earlier versions of iOS, the size of a page is 4 kilobytes. In later versions of iOS, A7- and A8-based systems expose 16-kilobyte pages to the 64-bit userspace backed by 4-kilobyte physical pages, while A9 systems expose 16-kilobyte pages backed by 16-kilobyte physical pages. These sizes determine how many kilobytes the system reads from disk when a page fault occurs. *Disk thrashing* can occur when the system spends a disproportionate amount of time handling page faults and reading and writing pages, rather than executing code for a program.

The logical address space of a process consists of mapped regions of memory. Each mapped memory region contains a known number of virtual memory pages. The kernel associates a *VM object* with each region of the logical address space. The kernel uses VM objects to track and manage the resident and nonresident pages of the associated

regions. A region can map to part of the backing store or to a memory-mapped file in the file system. Each VM object contains a map that associates regions with either the default pager or the vnode pager. The *default pager* is a system manager that manages the nonresident virtual memory pages in the backing store and fetches those pages when requested. The *vnode pager* implements memory-mapped file access. 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.

In addition to mapping regions to either the default or vnode pager, a VM object may also map regions to another VM object. The kernel uses this self referencing technique to implement *copy-on-write* regions. Copy-on-write regions allow different processes (or multiple blocks of code within a process) to share a page as long as none of them write to that page. When a process attempts to write to the page, a copy of the page is created in the logical address space of the process doing the writing. From that point forward, the writing process maintains its own separate copy of the page, which it can write to at any time. Copy-on-write regions let the system share large quantities of data efficiently in memory while still letting processes manipulate those pages directly (and safely) if needed. These types of regions are most commonly used for the data pages loaded from system frameworks.

Each VM object contains several fields, as shown in Table

Table: Fields of the VM object

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

B. Wired Meomry

Wired memory (also called *resident* memory) stores kernel code and data structures that must 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. For example, an application that creates threads and ports implicitly allocates wired memory for the required kernel resources that are associated with them.

C. Paging Out Process

In OS X, when the number of pages in the free list dips below a computed 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:

- If a page in the active list is not recently touched, it is moved to the inactive list.
- If a page in the inactive list is not recently touched, the kernel finds the page's VM object.
- If the VM object has never been paged before, the kernel calls an initialization routine that creates and assigns a default pager object.
- The VM object's default pager attempts to write the page out to the backing store.
- 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.

D. Paging In Process

The final phase of virtual memory management moves pages into physical memory, either from the backing store or from the file containing the page data. A memory access fault initiates the page-in process. A memory access fault occurs 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 on disk, 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.

III. KERNEL

The kernel provides many enhancements for OS X. These include *preemption*, *memory protection*, enhanced performance, improved networking facilities, support for both Macintosh (Extended and Standard) and non-Macintosh (UFS, ISO 9660, and so on) file systems, object-oriented APIs, and more.

In contrast, OS X is a *preemptive multitasking* environment. In OS X, the kernel provides enforcement of cooperation, scheduling processes to share time (preemption). This supports real-time behavior in applications that require it.

In OS X, processes do not normally share memory. Instead, the kernel assigns each *process* its own *address space*, controlling access to these address spaces. This control ensures that no application can inadvertently access or modify another application's memory (protection). Size is not an issue; with the virtual memory system included in OS X, each application has access to its own 4 GB address space.

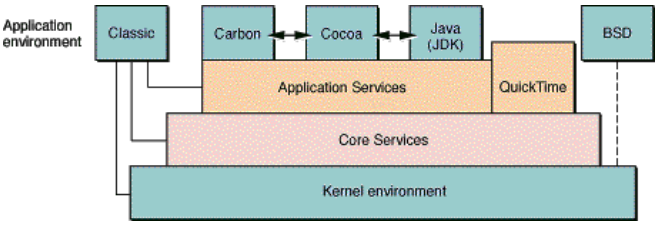
Viewed together, all applications are said to run in user space, but this does not imply that they share memory. User space is simply a term for the combined address spaces of all user-level applications. The kernel itself has its own address space, called kernel space. In OS X, no application can directly modify the memory of the system software (the kernel).

A. *Darwin:*

The OS X kernel is an *Open Source* project. The kernel, along with other core parts of OS X are collectively referred to as *Darwin*. Darwin is a complete operating system based on many of the same technologies that underlie OS X.

Below figure, shows the relationship between Darwin and OS X. Both build upon the same kernel, but OS X adds Core Services, Application Services and QuickTime, as well as the *Classic*, *Carbon*, *Cocoa*, and Java (JDK) application environments. Both Darwin and OS X include the BSD command-line application environment; however, in OS X, use of environment is not required, and thus it is hidden from the user unless they choose to access it.

Fig. 1. Darwin and OS X

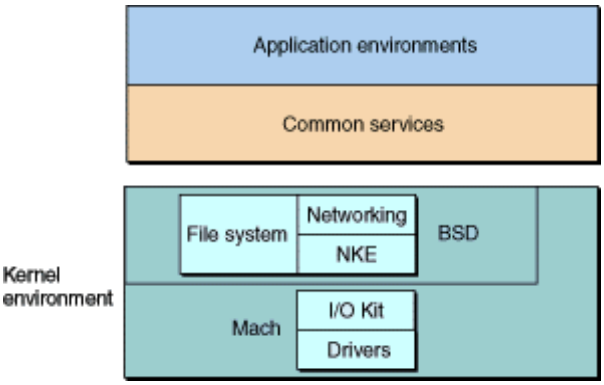


Darwin technology is based on *BSD*, Mach 3.0, and Apple technologies. Best of all, Darwin technology is Open Source technology. Darwin is based on proven technology from many sources. A large portion of this technology is derived from FreeBSD, a version of 4.4BSD that offers advanced networking, performance, security, and compatibility features. Other parts of the system software, such as Mach, are based on technology previously used in Apple's MkLinux project, in OS X Server, and in technology acquired from NeXT. Much of the code is platform-independent. All of the core operating-system code is available in source form.

B. *Architecture:*

The foundation layer of Darwin and OS X is composed of several architectural components, as shown in below figure. Taken together, these components form the *kernel environment*.

Fig. 2. Mac OS Kernel Architecture



In OS X, however, the kernel environment contains much more than the Mach kernel itself. The OS X kernel environment includes the Mach kernel, BSD, the I/O Kit, file systems, and networking components. These are often referred to collectively as the kernel. Each of these components is described briefly in the following sections. For further details, refer to the specific component chapters or to the reference material listed in the bibliography.

Because OS X contains three basic components (Mach, BSD, and the I/O Kit), there are also frequently as many as three APIs for certain key operations. In general, the API chosen should match the part of the kernel where it is being used, which in turn is dictated by what your code is attempting to do. The remainder of this chapter describes Mach, BSD, and the I/O Kit and outlines the functionality that is provided by those components.

C. Mach:

Mach manages processor resources such as CPU usage and memory, handles scheduling, provides memory protection, and provides a messaging-centered infrastructure to the rest of the operating-system layers. The Mach component provides

- untyped interprocess communication (*IPC*)
- remote procedure calls (*RPC*)
- scheduler support for symmetric multiprocessing (*SMP*)
- support for *real-time* services
- virtual memory support
- support for *pagers*
- modular architecture

D. BSD:

Above the Mach layer, the BSD layer provides "OS personality" APIs and services. The BSD layer is based on the BSD kernel, primarily *FreeBSD*. The BSD component provides

- file systems
- networking (except for the hardware device level)
- UNIX security model
- syscall support
- the BSD process model, including process IDs and signals
- FreeBSD kernel APIs
- many of the *POSIX* APIs
- kernel support for pthreads (POSIX threads)

E. Networking:

OS X networking takes advantage of BSD's advanced networking capabilities to provide support for modern features, such as Network Address Translation (*NAT*) and *firewalls*. The networking component provides

- 4.4BSD TCP/IP stack and socket APIs
- support for both IP and DDP (AppleTalk transport)
- multihoming
- routing
- multicast support
- server tuning
- packet filtering
- Mac OS Classic support (through filters)

F. File System:

OS X provides support for numerous types of file systems, including *HFS, HFS+, UFS, NFS, ISO 9660*, and others. The default file-system type is HFS+; OS X boots (and "roots") from HFS+, UFS, ISO, NFS, and UDF. Advanced features of OS X file systems include an enhanced Virtual File System (*VFS*) design. VFS provides for a layered architecture (file systems are *stackable*). The file system component provides

- *UTF-8* (Unicode) support
- increased performance over previous versions of Mac OS.

ACKNOWLEDGMENT

I would like to express my special thanks to our teacher "Mrs. Minal Deshmukh" for their able guidance and support in completing my case study. I would also like to say thanks to my friends for helping me to get needful information for my case study on the topic.

REFERENCES

[1] https://techcrunch.com/2017/10/01/apple-open-sourced-the-kernel-of-ios-and-macos-for-arm-processors/amp/

[2] https://docstore.mik.ua/orelly/unix3/mac/ch07_01.

[3] https://github.com/apple/darwin-

[4] https://apple.stackexchange.com/questions/72033/how-does-memory-management-work-in-mac-os-x

[5] https://computer.howstuffworks.com/macs/mac-os-x2.htm

[6] https://developer.apple.com/documentation/kernel