CIS 31 Project

Goals

- Describe the Operating System assigned to your team.
- Explain ALL the design goals and chosen implementation strategies.
- Explain KEY tables and queues used by the Operating System.

In this project, we studied Mac OS X operating system, developed by Apple Inc. These are the different parts of the OS that we studied in detail:

1. Mac OS X Overview - Jun Li
2. Process Management - Rashmi Omprakash Baheti
3. Main Memory Management - John Michael Kilcrease
4. The File Management Scheme - Tetiana Gushchyk
5. GUI Overview - Dong Nguyen
6. ? - Johnny Phan (may have dropped the course)

We dig deeper into each of these areas in rest of the document.

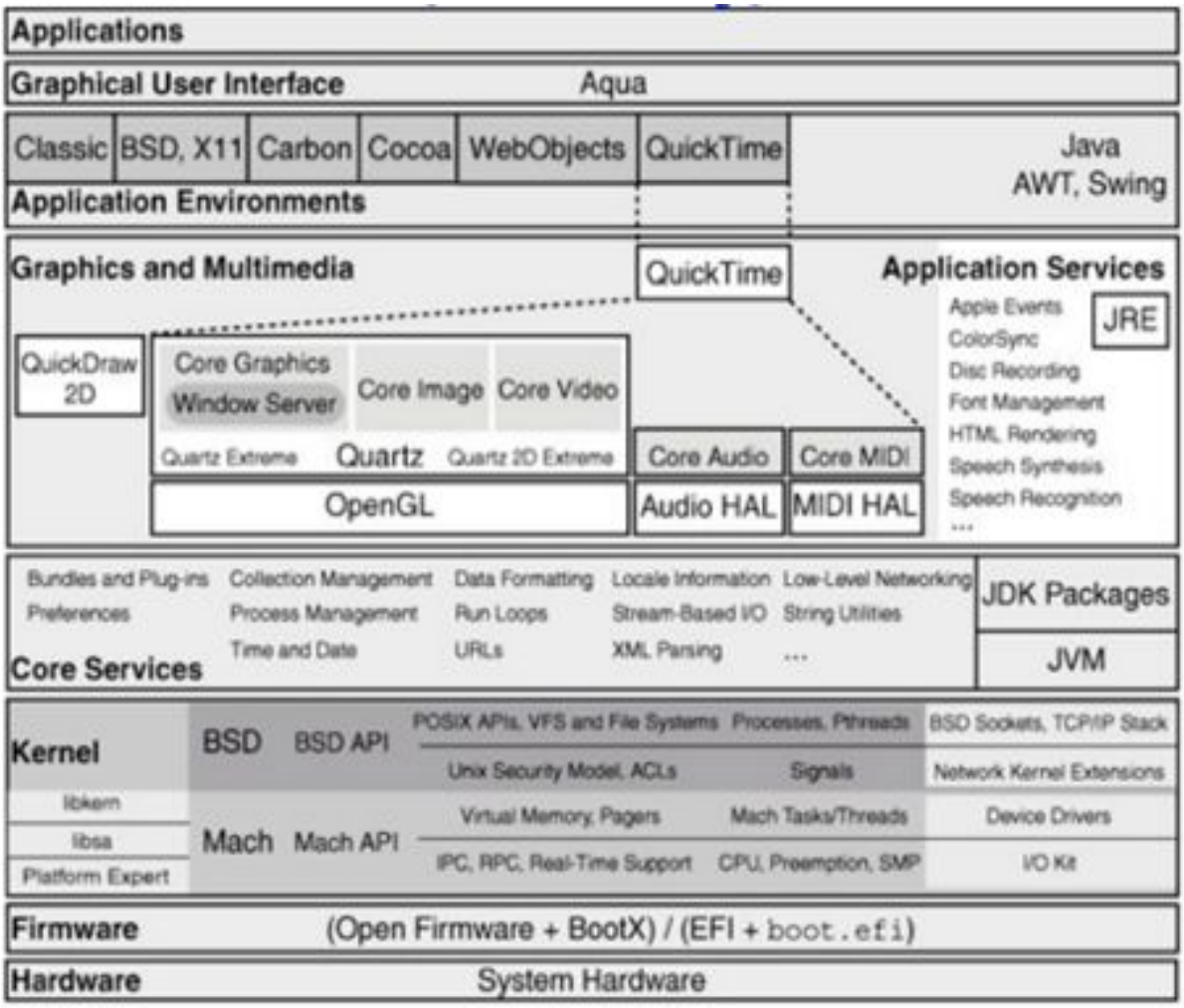
1. Mac OS X Overview

Jun Li

Mac OS X is a mix of several technologies that differ not only in what they do but also in where they came from, which philosophies they represent, and how they are implemented.

Figure1 shows a layered view of the important components of the Mac OS X architecture. The picture is approximate since it is impractical if not impossible to divide various components into cleanly separated layers. Sometimes there is overlap between the layers.

We will mostly focus on Kernel layer.



Major Components

The Mac OS X kernel is called xnu. In the simplest sense, xnu could be viewed as having a Mach-based core, a BSD-based operating system personality, and an object-oriented runtime environment for drivers and other kernel extensions. The Mach component is based on Mach 3, whereas the BSD component is based on FreeBSD 5. We can divide the Mac OS X kernel into the following components:

1. Mach: the services layer.
2. BSD: the primary system programming interface provider.
3. The I/O Kit: the runtime environment for drivers.
4. libkernan: in-kernel library.
5. libsaan: in-kernel library that is normally used only during early system startup.
6. The Platform Expert: the hardware abstraction module.
7. Kernel extensions: various I/O Kit families, the majority of loadable device drivers, and some non-I/O Kit extensions.

Mach may be considered the core of xnu. Mach provides critical low-level services that are transparent to applications. System aspects that Mach is responsible for include the following:

1. Hardware abstraction to some extent.
2. Processor management, including symmetric multiprocessing and scheduling.
3. Preemptive multitasking, including support for tasks and threads.
4. Virtual memory management, including low-level paging, memory protection, sharing, and inheritance.
5. Low-level IPC mechanisms that are the basis for all messaging in the kernel.
6. Real-time support that allows time-sensitive applications (e.g., media applications such as GarageBand and iTunes) to have latency-bounded access to processor resources.
7. Kernel debugging support.
8. Console I/O.

BSD in the context of Mac OS X is not the case that a well-defined BSD kernel runs within xnu, whether as a single Mach task or otherwise. Whereas some BSD-derived portions in xnu are similar to their original forms, other portions are quite different, since they were made to coexist with non-BSD entities such as the I/O Kit and Mach. Some aspects that BSD (or BSD-style code) is responsible for include the following:

1. BSD-style process model.
2. Signals.
3. User IDs, permissions, and basic security policies.
4. POSIX APIs.
5. Asynchronous I/O APIs (AIO).
6. BSD-style system calls.
7. TCP/IP stack, BSD sockets, and firewalling.

8. Network Kernel Extensions (NKEs), a type of kernel extension for making the BSD networking architecture fit into xnu.
9. The virtual file system (VFS) layer and numerous file systems, including a file-system-independent VFS-level journaling mechanism.
10. System V and POSIX interprocess communication mechanisms.
11. In-kernel cryptographic framework.
12. A system notification mechanism based on FreeBSD's kqueue/kevent mechanism, which is a system wide service enabling notifications between applications, and from the kernel to applications.
13. The fsevents file system change notification mechanism that is used by the Spotlight search technology.
14. Access control lists (ACLs) and the kauth authorization framework.
15. Various synchronization primitives.

The I/O Kit is a object-oriented device driver framework in xnu, which uses a restricted subset of C++ as its programming language. The I/O Kit's implementation consists of kernel-resident C++ libraries (libkern and IOKit) and a user-space framework. The I/O Kit's runtime architecture is modular and layered. It provides an infrastructure for capturing, representing, and maintaining relationships between the various hardware and software components that are involved in I/O connections. In this manner, the I/O Kit presents abstractions of the underlying hardware to the rest of the system. For example, the abstraction of a disk partition involves dynamic relationships between numerous I/O Kit classes: the physical disk, the disk controller, the bus that the controller is attached to, and so on. The device driver model provided by the I/O Kit has several useful features, such as the following:

1. Extensive programming interfaces, including interfaces for applications and user-space drivers to communicate with the I/O Kit.
2. Numerous device families such as ATA/ATAPI, FireWire, Graphics, HID, Network, PCI, and USB.
3. Object-oriented abstractions of devices.
4. Plug-and-play and dynamic device management ("hot-plugging").
5. Power management.
6. Preemptive multitasking, threading, symmetric multiprocessing, memory protection, and data management.
7. Dynamic matching and loading of drivers for multiple bus types.
8. A database for tracking and maintaining detailed information on instantiated objects (the I/O Registry).
9. A database of all I/O Kit classes available on a system (the I/O Catalog).
10. Interfaces for applications and user-space drivers to communicate with the I/O Kit.
11. Driver stacking.

The libkern library provides commonly needed services to drivers, it also contains classes that are generally useful for kernel software development. In particular, it defines the OSObject class, which is the

root base class for the Mac OS X kernel. OSObject implements dynamic typing and allocation features for supporting loadable kernel modules. The following are examples of the functionality provided by libkern:

1. Dynamic allocation, construction, and destruction objects, with support for a variety of built-in object types such as Arrays, Booleans, and Dictionaries.
2. Atomic operations and miscellaneous functions such as bcmp(), memcmp(), and strlen().
3. Functions for byte-swapping.
4. Provisions for tracking the number of current instances for each class.
5. Mechanisms that help alleviate the C++ fragile base-class problem

libsa is an in-kernel support library essentially an in-kernel linker used during early system startup for loading kernel extensions. It provides a minimal runtime environment for kernel extension, including:

1. Simple memory allocation.
2. Binary searching.
3. Sorting.
4. Miscellaneous string-handling functions.
5. Symbol re-mangling.
6. A dependency graph package used while determining kernel extension dependencies.
7. Decompression of compressed kernels and verification of checksums

The Platform Expert is an object essentially a motherboard-specific driver that knows the type of platform that the system is running on. The I/O Kit registers a nub for the Platform Expert at system initialization time. An instance of the IOPlatformExpertDevice class becomes the root of the device tree. The root nub then loads the correct platform-specific driver, which further discovers the busses present on the system, registering a nub for each bus found. The I/O Kit loads a matching driver for each bus nub, which in turn discovers the devices connected to the bus, and so on. The Platform Expert abstraction provides access to a wide variety of platform-specific functions and information, such as those related to:

1. Constructing device trees.
2. Parsing certain boot arguments.
3. Identifying the machine, which includes determining processor and bus clock speeds.
4. Accessing power management information.
5. Retrieving and setting system time.
6. Retrieving and setting console information.
7. Halting and restarting the machine.
8. Accessing the interrupt controller.
9. Creating the system serial number string.
10. Saving kernel panic information.
11. Initializing a "user interface" to be used in case of kernel panics.
12. Reading and writing the nonvolatile memory (NVRAM).
13. Reading and writing the parameter memory (PRAM).

Kernel extensions are dynamically loaded as needed. Most standard kernel extensions are targeted for the I/O Kit, but there are exceptions such as certain networking-related and file-system-related kernel extensions

Interaction Between Mach and BSD

Among these 7 components of Mac OS X kernel, the most important ones are Mach and BSD. Both components work closely to presents a cohesive and consistent picture to the end user. Certain kernel functionality has a lower-level implementation in one portion of the kernel with higher-level abstraction layers in another portion.
For example, the traditional process structure (struct proc), which is the primary kernel data structure that represents a UNIX process, is contained in the BSD portion, as is the u-area.

However, strictly speaking, in Mac OS X, a BSD process does not correspond to exactly one Mach task, which contains one or more Mach threads. Consider the example of the fork() system call, which is the only way to create a new process on a UNIX system.

In Mac OS X, Mach tasks and threads are created and manipulated using Mach calls, which user programs typically do not use directly. The BSD-style fork() implementation in the kernel uses these Mach calls to create a task and a thread.

Additionally, it allocates and initializes a process structure that is associated with the task. From the standpoint of the caller of fork(), these operations occur atomically, with the Mach and BSD-style data structures remaining in sync. Therefore, the BSD process structure acts as Unix "glue" in Mac OS X. Similarly, BSD's unified buffer cache (UBC) has a back-end that hooks into Mach's virtual memory subsystem.

Design Philosophy and Benefits

From a high-level standpoint, Mac OS X may be seen as consisting of three classes of technologies: those that originated at Apple, those that originated at NeXT, and "everything else." The latter consists mostly of third-party open source software.

On the one hand, such confluence makes it somewhat hard to clearly visualize the structure of Mac OS X and might even be a stumbling block for the new Mac OS X programmer.

On the other hand, Mac OS X programmers have a rather colorful environment to give vent to their creative fervors. The end user is the bigger beneficiary, enjoying a range of software that is not seen on any other single platform.

In particular, Mac OS X provides the benefits of a typical Unix system, while maintaining the traditional ease of use of a Macintosh. The Mac OS X Unix environment is standard enough so that most portable

Unix software such as the GNU suite and X Window applications runs easily. Mac OS X is often dubbed a mass-market Unix system, and yet, traditionally non-Unix, mainstream software, such as Microsoft Office and the Adobe Creative Suite, is available natively for Mac OS X.

2. Process Management

Rashmi Omprakash Baheti

Mac OS has evolved tremendously since its inception. Today's OS X is the fusion of Mac OS Classic and NeXTSTEP. The fusion of two operating systems resulted in a new OS that has become far more popular than the both of them combined. Sierra is Apple's latest incarnation of OS X. The UNIX-like core of Mac OS X is called Darwin, which is comprised of the XNU kernel and UNIX shell environment. The core of XNU is the Mach microkernel.

In this section, we discuss the process management aspect of Mac OS X or its Mach microkernel.

Task & Threads

OS X is a UNIX-certified operating system. But, it does not recognize the notion of a process as UNIX does. It employs a slightly different approach, using the concepts of the more lightweight tasks rather than processes. UNIX uses a top-down approach, in which a process is divided into one or more threads. Mach uses a bottom-up approach in which the fundamental unit is a thread, and one or more threads are contained in a task.

A Task

- is the basic unit of resource allocation and contains a collection of resources such as paged virtual address space, IPC space, exception handlers, credentials, file descriptors, protection state, signal management state, and statistics.
- has no life on its own.
- exists to serve as a container of one or more threads. A multithreaded process is implemented as a Mach task containing multiple Mach threads.
- contains some number of threads, all of which share the task's resources.

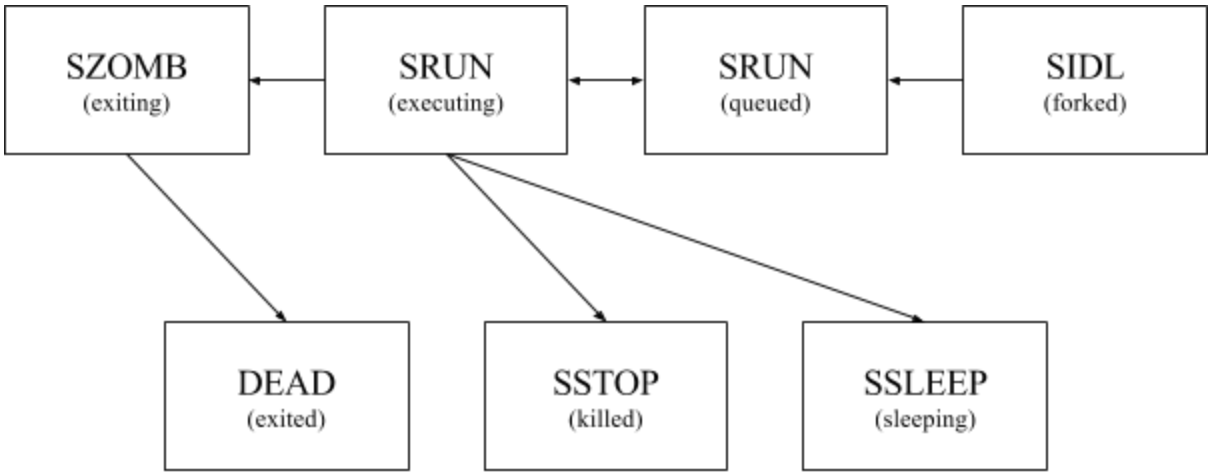
A thread

- is the the basic computational entity in Mach and executes within the context of exactly one task.
- has access to all of the elements of the containing task.
- contains information such as scheduling priority, scheduling policy, processor usage statistics etc.
- is the fundamental schedulable entity. Each thread is scheduled preemptively and independently of other threads, whether they are in the same task or in any other task.
- tracks the CPU it is bound to so that it gets scheduled to execute on the same CPU each time.
- is significantly cheaper to create or destroy than a task.

Mach represents the kernel by a task as well. The kernel task contains threads which functionalities such as bootstrapping, scheduling, exception handling, networking, and file system I/O. When a Mac OS X systems boots, kernel task is the first task that is created and kernel bootstrap thread is the first thread that is created. Thus, Mac OS X kernel is a monolithic kernel containing components such as Mach, BSD, and the I/O Kit, all running as groups of threads in a single task in the same address space.

Mac OS X is a preemptive multitasking system. A thread can get preempted at any given time, whether or not the thread is ready for it. A running thread can also voluntarily yield the CPU to a particular thread through a mechanism called Handoff. This feature is very useful in Mach, given that it is a message-passing kernel, and messages pass between threads. Due to Handoff, the messages can be processed with minimal latency, rather than opportunistically waiting for the next time the message-processing thread, sender or receiver, gets scheduled.

Following diagram shows the state transition diagram of Mac OS X.



Process State Transition Diagram

Mac OS X has a limit on the total number of tasks and threads that can be concurrently running at any given point of time. The following command reveals these limits.

$sysctl kern.num_tasks kern.num_taskthreads kern.num_threads
kern.num_tasks: 2048
kern.num_taskthreads: 2048
kern.num_threads: 10240

The default values of these limits indicate that Mac OS X permits only 2048 tasks or 10240 threads to be running concurrently at any given point of time.

Scheduling

Mach's thread scheduling is highly extensible, and actually allows changing the algorithms used for thread scheduling.

The scheduling algorithm used on a Mac OS machine can be checked using following command. ‘multiq’ is the default priority-based scheduling algorithm.

$sysctl kern.sched
kern.sched: multiq

Mac OS X scheduler has some unique features that are not found in any other popular operating system. Following is a list of some of these unique features.

- The scheduler schedules only Mach threads and no other higher-level entities.
- The scheduler does not use the knowledge that two or more threads may belong to the same task to select between them. In theory, such knowledge could be used to minimize the overhead of context switching. But, it also makes a scheduling algorithm very complicated.
- Mach uses the same scheduler for both multiprocessor and uniprocessor machines.
- The scheduler supports handoff scheduling, wherein a thread can directly yield the processor to another thread without fully involving the scheduler. As a result of a handoff, the current thread’s remaining quantum is given to the new thread to be scheduled.
- The scheduler supports continuation scheduling, wherein a thread gets resumed with a new stack and given continuation function after the specified event happens. This makes context switching much faster since the saving and loading of registers can be omitted. This feature used in many places around the kernel.
- The Mac OS X scheduler supports multiple scheduling policies.

The default time quantum used by the scheduler is 10 milliseconds. The following command reveals this information.

$sysctl kern.clockrate
kern.clockrate: { hz = 100, tick = 10000, tickadj = 2, profhz = 100, stathz = 100 }

The default tick value of 10000 microseconds or 10 milliseconds wakes up the scheduler every 10 milliseconds to take scheduling decisions.

The default scheduler is priority-based i.e. scheduler takes into account the priorities of runnable threads to take its scheduling decisions. Each thread has a base priority. The scheduling priority of a thread is computed using this base priority and the thread's recent processor usage. This allows scheduler to balance processor usage across available threads. Scheduling priority of a thread thus decreases when it is using the CPU too much, and increases when it is not getting enough CPU. Threads can thus migrate between priority levels. This migration is however within a priority band. The default base priority for user threads is 31, whereas the minimum kernel priority is 80. Consequently, kernel threads are substantially favored over user threads.

A fundamental data structure maintained by the scheduler is a run queue. Each run queue structure represents a priority queue of runnable threads and contains an array of 128 doubly linked lists, one

corresponding to each priority level. The priority levels are divided into four bands according to their characteristics, as described in following table.

Priority Band	Characteristics
Normal	Normal application thread priorities.
System high priority	Threads whose priority has been raised above normal threads.
Kernel mode only	Reserved for threads created inside the kernel that need to run at a higher priority than all user space threads.
Real-time threads	Threads whose priority is based on getting a well-defined fraction of total clock cycles, regardless of other activity (e.g. an audio player application).

In addition to the priority of a thread, the 'role' of a task also directly affects the frequency with which threads get scheduled. Following are the task roles supported by Mac OS X.

- TASK_FOREGROUND_APPLICATION
    - This is intended for a normal UI-based application meant to run in the foreground from the UI's standpoint.
- TASK_BACKGROUND_APPLICATION
    - This is intended for a normal UI-based application meant to run in the background from the UI's standpoint.
- TASK_CONTROL_APPLICATION
    - This designates a task as the UI-based control application.
- TASK_GRAPHICS_SERVER
    - This is intended for a window management server.

While a thread cannot explicitly control its own scheduling, Mach offers several scheduling policies to help user programs to achieve desired class of service. User program can retrieve and set scheduling policies on its threads using pthread_getschedparam() and pthread_setschedparam() functions of Pthreads API. Following are the scheduling policies supported by Mac OS X.

- THREAD_STANDARD_POLICY
    - This is the standard scheduling policy and is the default for timesharing threads. In this policy, threads running long-running computations are fairly assigned approximately equal CPU.
- THREAD_EXTENDED_POLICY
    - This is an extended version of the standard policy. In this policy, a hint designates a thread as long-running or non-long-running. In the former case, this policy is identical to THREAD_STANDARD_POLICY. In the latter case, threads run at a fixed priority as

long as its processor usage does not exceed an unsafe limit. After exceeding the unsafe limit, scheduler temporarily demotes a non-long-running thread to being a long-running or time-sharing thread.

- THREAD_PRECEDENCE_POLICY
    - This policy allows threads within a task to be designated as more or less important relative to each other.
- THREAD_TIME_CONSTRAINT_POLICY
    - This is a real-time scheduling policy intended for threads with real-time constraints on their execution. Under this policy, a thread can specify to the scheduler that it needs a certain fraction of processor time. The scheduler favors a real-time thread over all other kind of threads.

Interrupts

Interrupt handling in Mac OS X is very similar to how it happens in other popular operating systems. The kernel sets the predefined entry points in a one-dimensional array which stores function pointers to interrupt handlers. To handle an interrupt, the CPU jumps to the function pointer and executes the function after switching to supervisor mode. Rather than install separate handlers individually for every interrupt, Mac OS X installs one handler for all the traps, and have that handler jump according to a predefined table.

The Mac OS X kernel implements a mechanism called Asynchronous Software Traps (ASTs) which is a software equivalent of the low-level hardware trap mechanisms. When a processor is about to return from an interrupt context, including returns from system calls, it checks for AST, and takes a trap if it finds one. The kernel's clock interrupt handler also periodically checks for ASTs. Thus, these software traps are both initiated and handled by software. Using ASTs the kernel responds to out-of-band events requiring attention. ASTs are crucial for kernel operations. Thread scheduler makes heavy use of the ASTs.

3. Main Memory Management

John Michael Kilcrease

Original Design

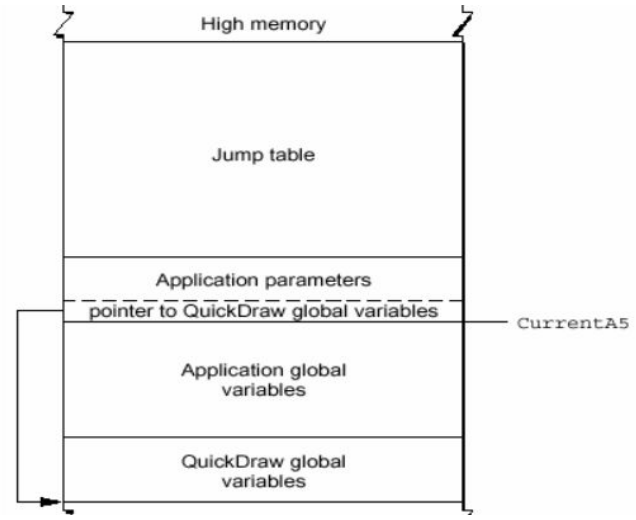
Originally Mac OS was designed for a computer with Motorola 68000 hardware. At the time MacOS was created for machines that could only run a single program at a time and there was no secondary storage or virtual memory, so there was very limited space. Under these circumstances they creates a very simple design that used two partitions that took up all the memory, the system partition got what it needed and the application partition got whatever was left. As time passes and multiprogramming came into existence the simple design created difficulties because of the more advanced requirements needed to execute multiple applications. Some of the other weaknesses of the Mac OS was its compaction could cause invalid handles and it did not have protected memory.

System Partition

The system partition consists of 2 parts. The lowest memory is for the smaller part which is allocated specifically for system global variables, global variables are used for things like how long the system has been running since it was turned on, or the information required for currently running applications. The System partition needs to keeps track of the top, bottom, and limit line of the currently running application. The second part of the system partition is the system heap, it is simply a space of memory allocated for exclusive use by the operating system for any components or information it needs.

Application Partition

The application partition is made up of three parts, A5 World which exists at the top of allocated memory and is a set size, the application stack which starts just below A5 World, and the application heap which starts at the bottom of allocated memory.



A5 World starts at the highest allocated memory slot and descends in memory. The A5 in the name comes from the register the microprocessor puts the pointer for this area in, the A5 register. A5 World contains four specific kinds of data: application global variables, application QuickDraw global variables, application parameters, and the application's jump table. The top pointer that the system partition keeps is between the application global variables and the application parameters, the pointer is called CurrentA5. The system partition uses the CurrentA5 pointer to access all information in A5 World. The A5 World size is set and will never increase or decrease throughout the life of the application although the size of the global variables and the jump table depend on the specific application.

Application Stack

Starting at the bottom of the A5 World and continuing downward is the application stack. The application stack is always contiguous and follows LIFO, it does not fragment. When a routine is called a stack frame is put on the stack, the stack frame contains the routine's parameters, local variables, and return address. The application stack does not have a size limit so it grow until it runs into space that is allocated to the application heap and corrupt the data. There is a task called stack sniffer that is constantly checking if the stack has grown over the heap. If it has the task will cause a system error.

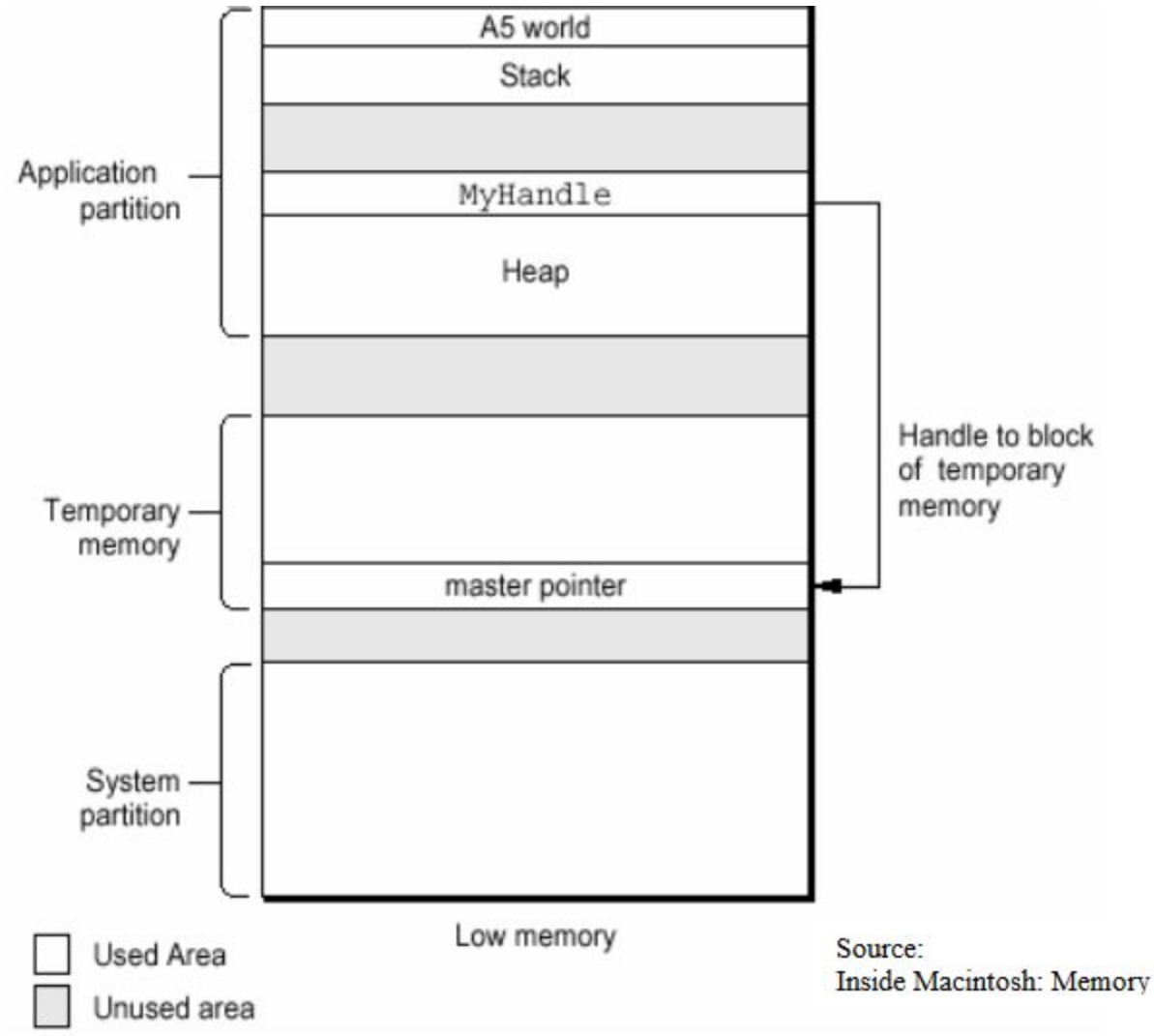
Application Heap

The application heap is the last part of the application partition and it starts at the lowest allocated memory location and works its way up. The bottom of the applications memory is another one of the pointers contained in the system partition and it is called ApplZone. Unlike the application stack it does have a limit to how far upward it can grow, the limit is kept in the system partition as ApplLimit. Memory in the application heap portion does fragment as memory is allocated and deallocated so it requires compression when all the open spaces are too small to receive new data. A part of the Heap cannot be compressed when it is a non-relocatable block or when it is a locked block. Mac OS had an issue with locked blocks because it did not count the number of locks a block had, so if it was locked 6 times and 1 of those locks released it would also release the other 5 locks. This creates problems as the other 5 things that put a lock on it are not expecting it to be moved. All blocks are either non-relocatable or relocatable. To reference a non-relocatable block only requires a pointer, pointing to the first bit. The pointer can be stored in A5 World, on the application stack, or in the application heap. To reference a relocatable block MacOS uses a scheme known as double indirect. The memory manager has a non-relocatable block in the heap called the master pointer block. Each pointer in the master pointer block points to a the start of a block in the heap, each master pointer block contains 64 pointers. The non-relocatable blocks are not directly accessed through the master pointer in the double indirection scheme, instead they are accessed to a handle, a handle is a double pointer that points to the pointer in the master pointer block.

Temporary Memory

When multitasking started requiring the space not used by the system partition to be divided amongst multiple applications instead of just a single application, the applications had to be given a size. The size was determined by what the application was programmed to ask for. If the application needed more memory than it originally asked for it would have ask the operating system for temporary memory. Temporary memory is additional memory that is not always attached to the block where the application is.

The operating system can allocate the temporary memory from free memory in any holes. It is meant to be allocated for only a short time so the application can finish what it needs to do.



Virtual Memory

MacOS did get virtual memory and it used multiple swap files that would be placed on the root partition during the default installation for paging. Virtual memory could be manually disabled easily in the control panel because it was not as integral to the systems operation. It got added in a much later installations. Users could also define page size. Virtual memory operated transparently to most applications unless they required knowing whether it was virtual memory because of their circumstances.

4. The File Management Scheme

Tetiana Gushchyk

The HFS Plus file system (or simply HFS+) is the preferred and default volume format on Mac OS X. The term HFS stands for Hierarchical File System, which replaced the flat Macintosh File System (MFS) used in early Macintosh operating systems. HFS remained the primary volume format for Macintosh systems before Mac OS 8.1, which was the first Apple operating system to support HFS+. Also called the Mac OS X Extended volume format, HFS+ is architecturally similar to HFS but provides several important benefits over the latter (HFS was largely single threaded and supported only 16-bit allocation blocks).

Features of HFS+

- Support for files up to 2^{63} bytes in size
- Unicode-based file/directory name encoding, with support for names containing up to 255 16-bit Unicode characters
- Metadata journaling through the kernel's VFS-level journaling mechanism
- Multiple mechanisms to allow one file system object to refer to another: aliases, hard links, and symbolic links
- Native support for access control lists (ACLs), with ACLs being stored as extended attributes
- Unix-style file permissions
- Journaling
- Dynamic resizing
- Dynamic defragmentation

Timestamps

HFS+ maintains its dates as a count of seconds from January 1, 1904, GMT, as an unsigned integer. This choice of start time is rather peculiar, as computers as we know them didn’t exist back then. Even UNIX dates are relative to the “epoch” (January 1, 1970). As a result, despite using a UInt32, the last possible date is February 6, 2040, 06:28:15 GMT.

Case Sensitivity

File systems are defi ned as case-insensitive or case-sensitive, depending on whether they consider letter uppercase/lowercase when comparing filenames. Additionally, while a file system may be case insensitive, it may still opt to be case-preserving — i.e., create files in the exact case passed to it, and maintain that case in all further operations on that file. HFS+ is case-insensitive, but case-preserving. OS X supports a newer variant, HFSX, which can be made case-sensitive, as well.

Hot Files

An interesting and quite unique feature of HFS+ is its dynamic adaptation to handle frequently accessed files. HFS+ keeps a temperature measurement on each file. The temperature is computed as the number of bytes divided by the fi le size (as a uint32_t, so it is always rounded down). This calculation is inversely

proportional to the file size, so it favors small files, whose contents are read very frequently. Those “hot” files exceeding a certain HFC_MINIMUM_TEMPERATURE are added to a special B-Tree in the metadata zone, which maintains up to HFC_MAXIMUM_FILE_COUNT entries, and their blocks are moved into the metadata zone as well.

Journaling

HFS+ supports journaling of metadata, including volume data structures, wherein metadata-related file system changes are recorded to a log file (the journal) that is implemented as a circular on-disk buffer. The primary purpose of a journal is to ensure file system consistency in the case of failure. Certain file system operations are semantically atomic but may result in considerable I/O internally. For example, creating a file, which involves adding the file's thread and file records to the Catalog B-Tree, will cause one or more disk blocks to be written. If the tree needs balancing, several more blocks will be written. If a failure occurs before all changes have been committed to physical storage, the file system will be in an inconsistent state perhaps even irrecoverably so. Journaling allows related modifications to be grouped into transactions that are recorded in a journal file. Then related modifications can be committed to their final destinations in a transactional manner either all of them or none at all. Journaling makes it easier and significantly faster to repair the volume after a crash, because only a small amount of information that contained in the journal needs to be examined.

Space Allocation

Space on an HFS+ volume is allocated to files in fundamental units called allocation blocks. For any given volume, its allocation block size is a multiple of the storage medium's sector size (i.e., the hardware addressable block size). Common sector sizes for disk drives and optical drives are 512 bytes and 2KB, respectively. The default allocation block size is 4KB.
An allocation block cannot be shared (split) between two files or even between forks of the same file. BSD's UFS (including the Mac OS X implementation) employs another unit of allocation besides a block: a fragment. A fragment is a fraction of a block that allows a block to be shared between files. When a volume contains a large number of small files, such sharing leads to more efficient use of space, but at the cost of more complicated logic in the file system.

An extent is a range of contiguous allocation blocks. An extent descriptor contains a pair of numbers: the allocation block number where the range starts and the number of allocation blocks in the range. For example, the extent descriptor { 100, 10 } represents a sequence of 10 consecutive allocation blocks, beginning at block number 100 on the volume.

An eight-element array of HFS+ extent descriptors constitutes an extent record. HFS+ uses an extent record as an inline extent list for a file's contents that is, up to the first eight extents of a file are stored as part of the file's basic metadata. For a file that has more than eight extents, HFS+ maintains one or more additional extent records, but they are not kept inline in the metadata.

B-Trees

HFS+ uses B-Trees to implement its critical indexing data structures that make it possible to locate both file content and metadata residing on a volume.

A B-Tree is a generalization of a balanced binary search tree. Whereas a binary tree has a branching factor of two, a B-Tree can have an arbitrarily large branching factor. This is achieved by having very large tree nodes. A B-Tree node may be thought of as an encapsulation of many levels of a binary tree. Having a very large branching factor leads to very low tree height, which is the essence of B-Trees: They are exceptionally suited for cases where the tree structure resides on an expensive-to-access storage medium, such as a disk drive. The lower the height, the fewer the number of disk accesses required to perform a B-Tree search. B-Trees offer guaranteed worst-case performance for common tree operations such as insertion, retrieval, and deletion of records. The operations can be implemented using reasonably simple algorithms, which are extensively covered in computing literature.

Motivation for B-Trees

The most fundamental concept in any file system is the mechanism used to store and retrieve the files. A file system needs a mechanism that answers several run-time needs:

- Searches: Since the primary goal of a file system is to locate files, it must be able to retrieve files in the most efficient manner possible. Since the number of files tends to be very large, this calls for sub-linear time — O(n) simply isn't scalable for millions of files. Searches are often hierarchical, as files are put into folders, and folders are put into subfolders still. ‰ Insertions: Though relatively less frequent than locating files, from time to time files are added to the file system. This translates into an insertion of a file entry.
- Updates: As files are renamed, moved, and deleted, the mechanism must be flexible enough not to become fragmented. This type of fragmentation, referred to as index fragmentation, occurs in cases where fi le indices, commonly sequential, become sparse as a result of files being moved to some other location, or deleted.
- Random access: Though most files are read sequentially, from start to finish, a user or process can always ask to jump around in a file, out of order, commonly by using the lseek(2) system call. A fi le system is fully flexible if, once a file is located, its blocks on disk can be freely accessed, and can be sought through effi ciently. Every file system favors writing files contiguously, but this is not always a simple matter. When contents are frequently added or removed from a file, it is only a matter of time before block fragmentation ensues, as the file allocation on disk simply cannot be kept contiguous, and the file has to extend to other blocks.

HFS+ specifically uses a variant of B+ Trees, which themselves are B-Tree variants. In a B+ Tree, all data resides in leaf (external) nodes, with index (internal) nodes containing only keys and pointers to subtrees. Consequently, index and leaf nodes can have different formats and sizes. Moreover, the leaf nodes, which are all at the same (lowest) level in the balanced tree, [10] are chained together from left to right in a linked list to form a sequence set. Whereas the index nodes allow random searching, the list of leaf nodes can be used for sequential access to the data. Note that since data corresponding to a key can be found only in a leaf node, a B+ Tree search starting from the root node always ends at a leaf node.

The Structure of an HFS+ Volume

Besides regular files and directories, an HFS+ volume contains the following entities:

- Reserved areas appear at the beginning and end of the volume. The volume header contains a variety of information about the volume, including the locations of the volume's other key data structures.

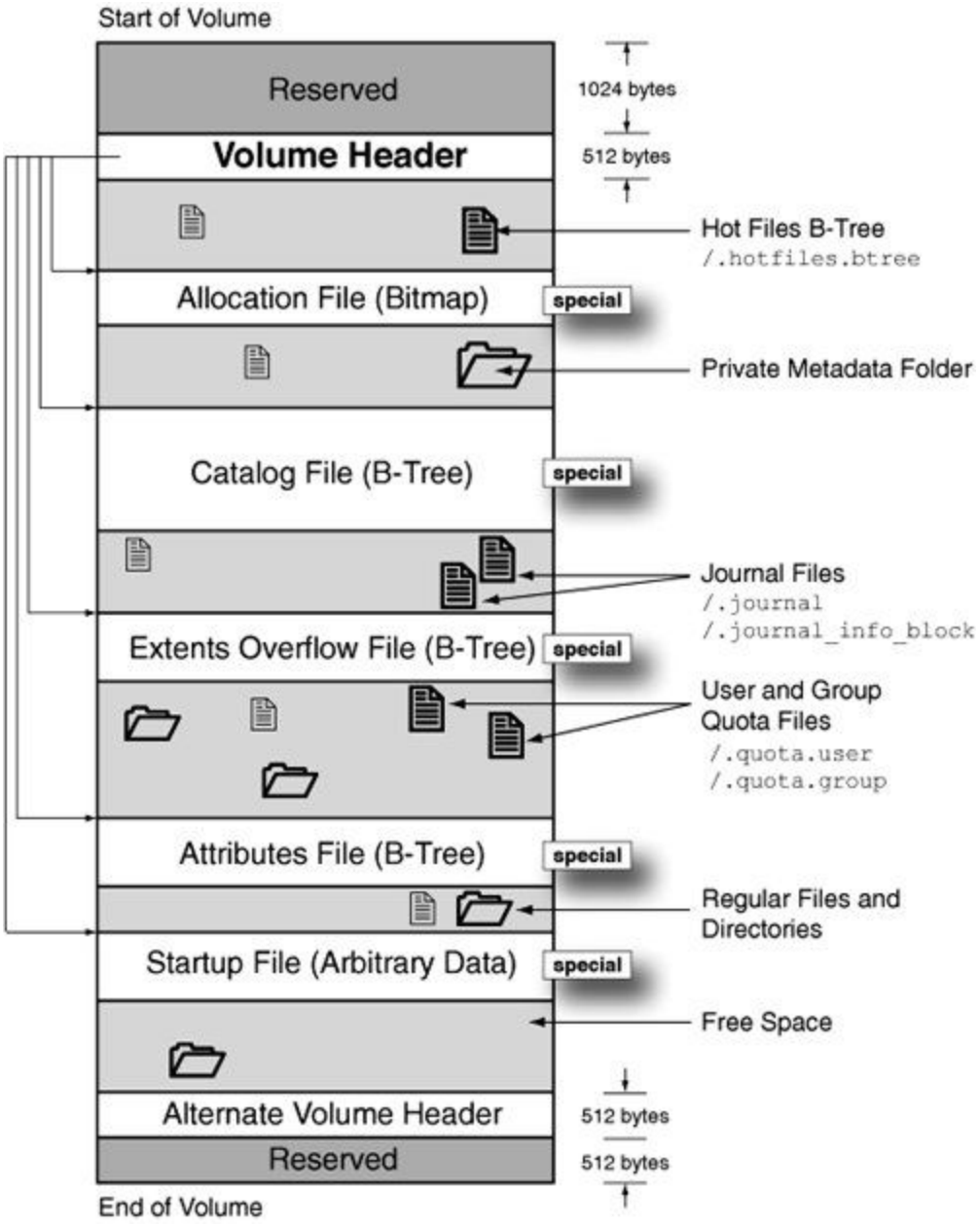
- The alternate volume header is a copy of the volume header. It is located near the end of the volume.
- The Catalog B-Tree stores the basic metadata for files and directories, including the first extent record (i.e., up to the first eight extents) for each file. The file system's hierarchical structure is also captured in the Catalog B-Tree through records that store parent-child relationships between file system objects.
- The Extents Overflow B-Tree stores overflow (additional) extent records of files that have more than eight extents
- The Attributes B-Tree stores extended attributes for files and directories.
- The Allocation file is a bitmap containing a bit for each allocation block, indicating whether the block is in use or not.
- The private metadata folder is used for implementing hard links and for storing files that are deleted while they are open
- The Startup file is meant to contain arbitrary information that an operating system might use to boot from an HFS+ volume.
- Journal files are used to hold information about the file system journal
- Quota files are used to hold information pertaining to volume-level user quotas and group quotas

The most critical structure of an HFS+ volume is the 512-byte volume header, which is stored at a 1024-byte offset from the start of the volume immediately after the first reserved area. The information contained in the volume header includes locations of various other important data structures. Unlike the volume header, these other structures do not have a predefined, fixed location the volume header serves as a starting point for the operating system (or other entities, such as a disk utility) while accessing the volume. A copy of the volume header the alternate volume header is stored at a 1024-byte offset from the end of the volume, immediately before the last reserved area. Disk and file system repair utilities typically make use of this copy.

The HFS+ volume header contains a fork-data structure for each of the five special files tagged as "special" three B-Trees, a bitmap, and an optional Startup file. Since a fork-data structure contains the fork's total size and the initial set of the file's extents, these files can all be accessed starting from the volume header. An HFS+ implementation reads the volume header and the appropriate special files to provide access to the user data (i.e., files, folders, and attributes) contained on a volume. The special files are not user-visible. They also do not contribute to the file count maintained in the volume header.

The Allocation file tracks whether an allocation block is in use. It is simply a bitmap, containing a bit for each allocation block on the volume. If a given block is either holding user data or assigned to a file system data structure, the corresponding bit is set in the Allocation file. Thus, each byte in this file tracks eight allocation blocks. Note that being a file, the Allocation file itself consumes an integral number of allocation blocks. One of its nice properties is that it can be grown or shrunk, allowing flexibility in manipulation of a volume's space. This also means that the last allocation block assigned to the Allocation file may have unused bits such bits must be explicitly set to zero by the HFS+ implementation.

The Catalog file describes the hierarchy of files and folders on a volume. It acts both as a container for holding vital information for all files and folders on a volume and as their catalog. HFS+ stores file and folder names as Unicode strings represented by HFSUniStr255 structures, which consist of a length and a 255-element double-byte Unicode character array.



The first two logical sectors (1024 bytes) and the last logical sector (512 bytes) of a volume are reserved. Although Mac OS X does not use these areas, they were used by earlier Mac OS versions.

5. GUI Overview

Dong Nguyen

Aqua is the name for a set of guidelines that describe look and behavior of Mac OS X GUI elements. The Interface Builder, a recommended technology in implementing OS X's GUI, assists programmers in laying out their application's GUI in accordance with the interface guidelines.
Thanks to the provided guidelines and tools, many of the Mac OS X applications/programs are intuitive and easy to use because they will feel familiar to the user.

The Aqua guidelines also allows for a very visually appealing interface by supporting various sizes visual objects. For example, Mac OS X allows for up to 256x256 pixels icons. A user can use the Icon Browser application, a part of Apple Developer Tools, to view icons and use the Icon Composer window to create his/her own icons for use — providing users with more customization of their personal computing system.

Mac OS X provides user with a simple and sleek GUI, preventing the cluttering of the user's screen from unneeded widgets and apps. Of course, the user can later choose to install any apps that he/she may need freely. The simple GUI does well to not overwhelm the general user and helps make it clear that the user is the one in control. Many industries are now following in Apple's design philosophy, planning their products GUI to be much more simple to appeal to the general populace.

An example of the Mac OS X desktop
(http://switchtoamac.com/guides/mac-os-x/desktop/the-mac-os-x-desktop.html)

Mac OS X also introduced Dashboard, an environment used to run widgets. The Dashboard resides on the desktop and is hidden from view until activated by the user. Once it is activated, the Dashboard allows for easy access to any widgets or applications that is included on the Dashboard. The default key to access the Dashboard is the F12 key.

- The user can simply mouse over any widget and left-click to access it.
- By holding the mouse button down, the user can drag and move widgets to reposition it however he/she desires.
- Typical widgets included by default may include the Calculator, Calendar, Clocks, Sticky Notes, Weather Reports, and News Reports.
- The user can select the plus or minus sign (+/-) on the bottom left of the screen to add or remove any widget from the Dashboard.

An example of a typical Dashboard on Mac OS X
(https://support.apple.com/en-us/HT201738)

The Dock takes a place on the Mac OS X desktop and is a convenient way for users to access commonly used applications. If the Dock is not visible, the user can hover his/her mouse near the Dock position (usually the bottom) to make it appear, or choose Dock > Turn Hiding Off from the Apple menu.

An example of a Mac OS X Dock
(https://support.apple.com/en-us/HT201730)

- Dragging an app off the Dock and hold it for a couple seconds will remove it from the Dock.
- Dragging an app onto a Dock will place the app on the Dock.
- The Dock may also hold any suspended app until they are resumed or ended by the user.

The Finder is the software that manages user files and can be found either on the Dock or in the menu bar at the top of the desktop screen. It is provided with the Mac OS X that helps users to find and organize files within the system.

The Finder icon

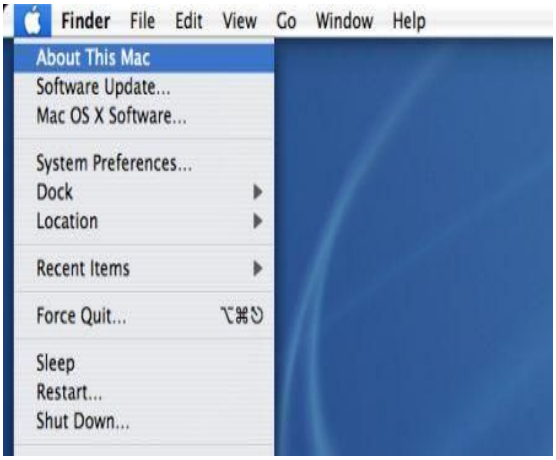
To open a window, click on the Finder; it may also show any windows that may be hidden behind other opened apps. A user may resize, minimize, maximize, or close a window once it is opened.

This details only but a few important parts of the Mac OS X general user interface. The current OS X is capable of many more general functions and operations, some of which will be described in the next section of general usage.

MAC OS X General Usage

A new user to MAC OS X will find it a bit different than conventional Windows OS but will quickly realize that most general operations will remain the same. The following pages contain information on general usage that may differ in different OS X integrations.

The menu bar near the top of the user's screen is an integral part in using the Mac OS X. Many key functions can be found on the menu bar including powering off your computer and updating the OS. More or less options will be shown on the menu bar depending on the currently opened app. For example, a compiler app may create a Build option as well as a Run option on the menu bar while non-compiler apps will not have those options. Below is a simplified explanation of what a few of these options contain.



As the name suggests, the File option contains many file-oriented operations such as open, save, etc.

The Edit option contains basic text editing operations such as cut, copy, and paste and more advanced versions of text editing such as formatting and templates.

The Help option will contain information and manuals/ regarding whichever app is opened and is currently in used. If no app is opened, it will have information on the system itself

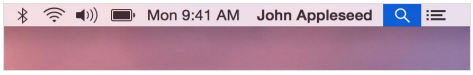
Basic menu bar for Mac OS X

Knowing the layouts and different functions of the various menu bars associated with each app will require time and practice; however, because Mac OS X applications follow the Aqua guidelines, it should not take too long before a user can feel familiar using a brand-new software. Below are some of the basic shortcuts using the keyboard that will help users quickly perform operations that are implemented in almost every Mac OS X application.

Shortcut	Description
Command-X	**Cut** the selected item and copy it to the Clipboard.
Command-C	**Copy** the selected item to the Clipboard. This also works for files in the Finder.
Command-V	**Paste** the contents of the Clipboard into the current document or app. This also works for files in the Finder.
Command-Z	**Undo** the previous command. You can then press Command-Shift-Z to **Redo**, reversing the undo command. In some apps, you can undo and redo multiple commands.
Command-A	**Select All** items.
Command-F	**Find** items in a document or open a Find window.
Command-G	**Find Again**: Find the next occurrence of the item previously found. To find the previous occurrence, press Command-Shift-G.
Command-H	**Hide** the windows of the front app. To view the front app but hide all other apps, press Command-Option-H.
Command-M	**Minimize** the front window to the Dock. To minimize all windows of the front app, press Command-Option-M.
Command-N	**New:** Open an new document or window.
Command-O	**Open** the selected item, or open a dialog to select a file to open.
Command-P	**Print** the current document.
Command-S	**Save** the current document.
Command-W	**Close** the front window. To close all windows of the app, press Command-Option-W.
Command-Q	**Quit** the app.

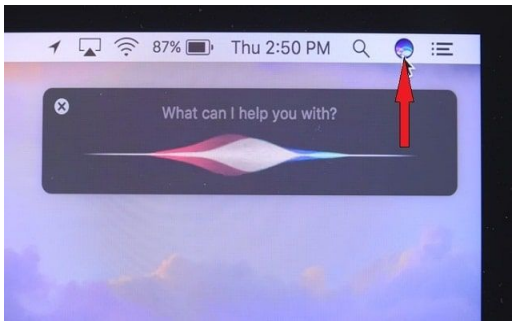
Common Mac OS X keyboard shortcuts
(https://support.apple.com/en-us/HT201236)

There are many more keyboard shortcuts that have not been included in favor of space. They are easily found by going to Apple's main website and viewing the support section. Users who have had experienced in other operating systems may find these and a few others very familiar and easy to be accustomed to as they follow the same general principles.
File operations in Mac OS X is like that of other operating systems. A user can use Spotlight, an application built in the Mac OS X that is the icon of a magnifying glass, to search for files within the system. The user can rename, move, sort, or open the files once they are found.
The Spotlight app can be found on the top right of the screen.



Spotlight application location on Mac OS X

The user can also use a voice recognition software known as Siri to perform daily tasks without having to manually execute the operations themselves. Formerly a part of only the iOS devices, present-day Mac OS X also includes the personal artificial assistant. By saying certain phrases to Siri, the AI will perform the operations necessary to complete a task. For example, the user can work on a document while vocally asking Siri to open a webpage for viewing.



Siri can be found either on the menu bar or on the Dock, represented by a distinct icon or the microphone icon.

To ask Siri one question, the user can hold down the Command key and Spacebar until Siri executes.

To learn about Siri's capabilities, the user can ask the AI "What can you do?"

Siri is a convenient and hands-free way for users to effectively multi-task without having to suspend their work. Present day Siri is capable for finding files, keep users notified of important events and/or schedules, seamlessly search the internet for information, and much more.

Users who prefer to not use the voice command can instead interact with Siri via typing in the keyboard directly. All the same functionalities Siri provides can still be provoked.

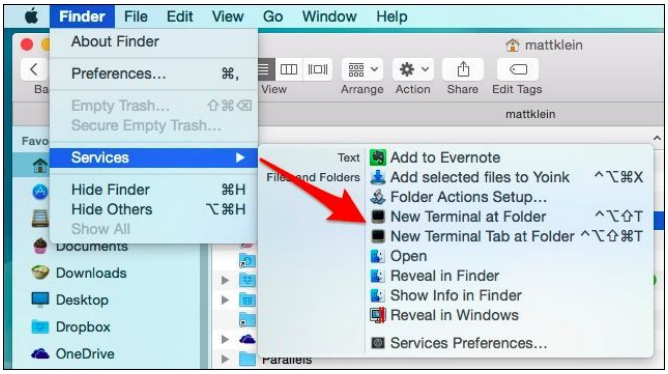
Users that feel that Siri is too intrusive can choose to disable the AI. This is done by going to System Preferences > Siri (or ask Siri to open Siri preferences.) In this setting panel, Siri can be turned on or off, change its spoken language, or turn off voice feedback for a silent system.

Siri provides users with a very streamlined and simple interface to perform general tasks, many of which is programmed into the AI itself. Thanks to Siri, Mac OS X is a very user friendly operating system for both new and experienced users alike. New users will not need to learn too many new operations and experienced users does not need to do trivial operations to get things done.

MAC OS X Command Line Interface

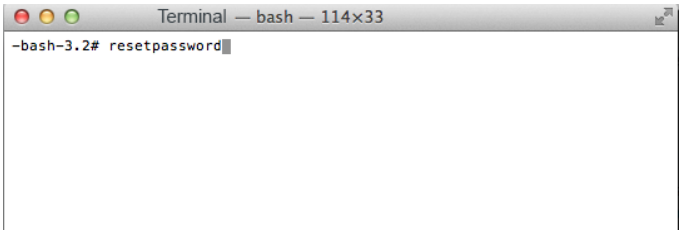
For the more tech-savvy and developers, Mac OS X also includes a command line interface/interpreter. Called the Terminal, the command line interface is used to control the UNIX based operating system underneath Mac OS X.

The Terminal is Apple's version of the Bash shell, a Unix command line environment. Other types of shells can be run besides the Bash shell, but the user will need to install them. Commands may differ from shells to shells, so a user must know which to use in different situations.



The Terminal is an application like any other, included within the Mac OS X's Utilities folder in Applications.

The terminal can also be opened by clicking on Finder in the menu bar > Services > New terminal...



When it is executed, the Terminal will show in the title bar:

- The name of the current user
- The current shell
- The window size of the terminal.

The Terminal will usually open at the top level of the user's Home directory. Opening the Terminal via the Finder and choosing New Terminal at Folder will open a new Terminal at the current opened folder. Because commands are executed relative to the Terminal's location, opening a Terminal in a folder will

save the user from needing to include the path to the folder every time a file within the folder is referenced. The user can also change the working directory of an already opened Terminal window by using the the change directory command (*cd*.)

Using the Terminal is straightforward. The user types in a command and press Enter, and the Terminal responds by executing the command. Some commands will provide visual feedbacks to the user in the form of text in the Terminal while other commands may silently process.

A command will include different elements depending on its specifications. A typical command will contain the command itself and argument(s) to modify a command's behavior.

Example: *ls -l *.txt*

- *ls* is the command to print files within the current directory
- *-l* is the argument that modifies the original **ls** behavior. In this case, the files are formatted to print in a list
- **.txt* is another argument. For this command, only files with the .txt extension will be printed to the command line interface.

Users should be careful of case sensitivities when using the Terminal as one wrong character could prevent a command from properly working. As with most operating systems, basic commands will not negatively impact the Terminal or the working system when something goes wrong; however, when dealing with more powerful commands that requires administrative or super-user privileges, extra care should be taken by the user when issuing commands. As a rule of thumb, a user should only use any commands that he/she can predict the output to and should research new commands before using them.

Because Mac OS X uses the Bash shell by default, many Unix commands will work in the Mac OS X Terminal. Below is a few basic and commonly used commands for general users:

UNIX Directory Commands

Command	What It Does
ls	Lists the names of the files in the working directory. For more complete information, use ls –alF (.
cd directoryname	Changes the working directory to the one you named.
cd ..	Brings you up one directory level.
cd	Returns you to your home directory.
pwd	Displays the pathname of the current directory.
mkdir newdirectoryname	Makes a new directory.
rmdir directoryname	Removes (deletes) an empty directory.

Working with Files

Command	What It Does
cp filename1 filename2	Copies a file.
chmod	Changes permissions for access to a file. Study the man page before using this one.
diff	Compares two files line by line (assumes text).
more filename	Displays a text file one page at a time. Press the spacebar to see the next page; press Q to quit. The man command works through more.
mv filename1 filename2	Moves a file or changes its name.
rm filename	Removes (deletes) a file.

Some shortcuts to the Mac OS X Terminal/command line interface:

- For arguments that are specific files that requires a user to type in a path to the file, the user can simply find the file in the Finder and drag and drop it into the Terminal window. Ther Terminal will extract the file's path and place it in the command without having the user type.

- Previous commands can be easily rewritten to the Terminal by pressing the Up Arrow on the keyboard until the desired command is found. The user can then modify the command arguments and press Enter to execute it.
- Pressing both the Command and C buttons will interrupt a command and stop its execution. Useful when the user needs to quit from a long running/frozen command.
- If a user wants to see a list of commands, he/she can hold down the Escape key and press the 'Y' key when the Terminal asks for permissions to print the commands. A list of commands will be printed along with their functionalities. Pressing Spacebar will load more commands, pressing 'Q' will exit the list of commands and return to the normal Terminal.

Since Mac OS X Terminal is built on the Unix command line interface, a user can view the built in manual to Unix commands within the Terminal itself. To open the manual on a specific command, the user can use the command *man*. The syntax is: *man [command]*, in which *command* is the command that the user wants more information on.

The Terminal coupled with the consistent and sleek graphical user interface provides Mac OS X users with two different choices on how to use the operating system, drawing more and more users both young and old. Although Mac OS X is not known to be easily customizable by everyday general users, many owners of the operating system would agree that it is simple to use and does its job very well.