Getting Inside OS X

Kim Bauters

December 16, 2008

Contents

List of Figures

List of Tables

1 Introduction

Mac OS X is the default operating system on all Apple computers. Mac OS X not only runs on the consumer version of the Apple computers, but also powers the Xserve range of servers, the AppleTV and the iPhone/iPod touch. These systems differ vastly in functionality, processor architectures and performance. Yet the versions of Mac OS X that run on all these different platforms differ barely. In essence, only the graphical user interface and some of the higher-level libraries are different.

For most people, this seems irrelevant. They like (or dislike) OS X for its ease of use, the wide range of powerful applications and the relative virus- and maintenance-free nature of the system. Little do they know or care that much of the ease of use and low maintenance finds its origin in the kernel and the file system, which are two of the most fundamental parts of any operating system.

This lack of interest result in a lack of literature on the subject and a large number of persistent – and erroneous – myths. Examples include the strong believe that the kernel of Mac OS X is a microkernel, since it is based on Mach. A contradicting myth concludes that the kernel of Mac OS X is based on BSD, a monolithic kernel. Most often, these myths are harmless. Until now, no one has come up with the idea to sell a replacement kernel. However, the myth that the OS X file system requires a defragmentation tool may turn out to be costly, since such tools are few and far between and hence come with a considerable price tag. As we shall see, there is simply no need for any such tool.

The goal of this paper is to provide a thorough overview of the Darwin XNU kernel and the HFS+ file system. In the process of doing so, we will debunk all the aforementioned myths. We will show that the Darwin kernel is a monolithic kernel *based on* Mach. We will further show that this monolithic kernel enjoys the same *maintainability* as a microkernel. We will explain how the Darwin XNU kernel is capable of running BSD processes and we will show how this does not require a BSD kernel but only support for BSD system calls. We will also explain how the HFS+ file system can both proactively and retroactively prevent fragmentation and we will look at real, life systems to find out the actual degree of fragmentation. We will not stop there; the reader will be pointed towards some noteworthy technology such as universal binaries, Spotlight and how a 32-bit kernel can run 64-bit applications.

Whenever possible, the technical description will be backed up by actual real-life experiments. For this purpose, we will give a brief introduction to tools that allow to explore the kernel and the file system on a very low level. Though an elaborate explanation of these tools is well beyond the scope of this paper, we will show how these tools can be used in an easy enough fashion to confirm quite a number of conclusions in this paper.

2 History

Apple has a long-standing history in personal computing and operating system design. Before taking a deeper look at OS X, we will first skim through the eventful history of Apple and highlight those events that eventually lead to what we now know as OS X. While we could do without this history, the reader would miss out on quite a few subtleties and design decisions that found their way into OS X. Above all, the history of Apple is fun to read and it would be a shame to miss out on it.

Apple was established on April 1, 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne [16]. Their first product, the Apple I, was sold as a motherboard for the relatively low price at that time of $666,66. Though not too many units were sold, their next computer, the Apple][, proved to be a real success. As was customary in that time, the operating systems that powered these and some of the later Apple computers were simple command line user interfaces. After Steve Jobs and some other employees visited Xerox PARC in late 1979, Steve Jobs realises that all future operating systems would be using a GUI and only a couple of years later the ancestor of OS X is released into the world.

In 1983 and 1984 Apple released two personal computers for the masses that incorporated a graphical user interface.[1] These two personal computers were the Lisa and the Macintosh and both were powered by Motorola 68k processors. Even though the Lisa was driven by Lisa OS and the Macintosh by System 1, both operating systems tend to have a lot in common. In fact, the Macintosh was introduced as a cheap successor to the Lisa. Most noteworthy was of course the GUI. Key features included the desktop metaphor, the use of a mouse pointer, the cut-and-paste metaphor, the ability to move and delete icons from the desktop and a menu bar at the top of the screen that is identical for every program. This was in an era when the console was omnipresent and the mouse was seen as a futuristic input device.

Apple did not simply introduce a GUI, but already made some noteworthy performance improvements for drawing the GUI. One of these improvements is actually influenced by the SmallTalk [7] system. After one of the employees of Apple had seen a presentation he noticed that SmallTalk only redrew those parts of the screen that had previously been obscured by the window that the user was now moving around. The truth is that SmallTalk did not have such a feature and simply redrew the entire screen. Nevertheless, the idea of using regions on the screen and only redrawing the regions that are modified has since been present in almost every GUI, including Lisa OS and System 1.

From a technical point of view, these operating systems were very basic. They could only run a single program at a time, they had no protected mem-

[1] Obviously, this development was based on the research by SRI and later by Xerox PARC[23] on graphical user interfaces. This is not at all strange, since most of the early Apple employees were former Xerox PARC engineers.

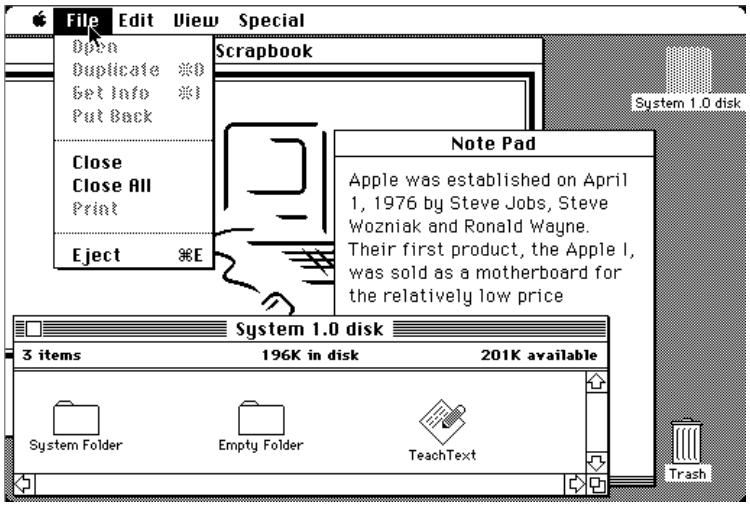


Figure 1: The user interface of the System 1 operating system.

ory and they had a flat file system, with only flawed support for directories[2]. Basically, the OS was nothing but a kernel from a command line OS with an added GUI. Nevertheless, the file system included key concepts such as resource forks, names with a length of 255 characters and storage of metadata which was needed for the GUI. Needless to say that everything was built to support the GUI, which was the most important aspect of these personal computers and their main selling point. This would remain so for the years to come, caused by the lack of any noteworthy competition and the strong market dominance they enjoyed by being the first to market a GUI OS.

Fast forward to mid 1991, when System 7 was released. Under the hood, a lot had been added, though most of the additions had severe limitations. The first addition, cooperative multitasking, allowed multiple programs to run at the same time but relied on the programs themselves to hand the control back to the system in order to move to the next program. This made the system notoriously unstable, especially when using some of the more devious programs who tried to monopolise the system. Another addition was virtual memory using paging which gave the program a continuous range of memory addresses, but without the needed memory protection. As such, a program could easily, and often unwillingly, write into the memory space of another program with unwanted crashes or necessary manual reboots as a consequence. Other additions included the support for aliases (somewhat like links, in Windows terms), support for MMU, the possibility for system extensions, the introduction of TrueType fonts, a true 32-bit address space and AppleScript, which was a high-level language to allow easy automation of tasks.

[2]Folders could only be added to the root – whereas *adding* might be the wrong word: one folder was always empty; as soon as you used this folder a new empty folder was created.

A remarkable achievement of System 7 was that this operating system, as of version 7.1.2, ran on both the 68k architecture of the old Motorola processors and on the new PowerPC architecture of the new IBM processors. The new PowerPC processors were introduced in a new range of Macintosh computers that were launched around 1994. Running a single version of an operating system on two entirely different architectures is no simple task to accomplish. The latest versions of Linux and Windows still require you to install a 64-bit operating system on a 64-bit processor and a 32-bit version of the operating system on a 32-bit processor, even when the processor architecture is basically the same. What Apple did in System 7.1.2 was to allow a single version of their operating system to be installed on two different architectures. Old 68k programs were still supported on the PowerPC architecture through transparent emulation of the code. Most remarkable however was the introduction of fat binaries [15]. All programs that were compiled as fat binaries were capable of running on both the old 68k architecture and the new PowerPC architecture without the need for emulation or the need for another version of the program. This is one of the few operating systems to date with such a feature. Remarkably, the best-known other systems that had a similar feature (*i.e.* Mach and NeXTStep) contributed directly to the development of OS X, as we shall see shortly.

In the years to come, the competition began to grow and started scooping away a lot of the customers who were drawn to more stable operating systems. One of these competitors was Microsoft, who introduced Windows NT in 1993 and Windows 95 in 1995. When Windows NT was released, System 7 paled in comparison. Essentially, Windows NT was written from scratch and showed off some of the most modern technologies. Windows NT was not only easy to use, it also offered rock solid stability. More and more people switched to a Windows operating system and Apple started to look feverishly for a new operating system that could hold its own against Windows NT and Windows 95.

A great number of research projects were underway at that time, most of which had already started in 1989. One of these projects was Project Red [4].[3] This project aimed at building an entirely new operating system with a true microkernel at the core. Later on, this project was absorbed by another project (called Star Trek) that tried to migrate the kernel to the x86 architecture. This endeavour succeeded, in part thanks to the help of Novell [16], but eventually the project was cancelled due to the performance superiority that PowerPC processors enjoyed over x86 processors at that time.

[3]Project Red and project Pink? And yes, there was also project Blue. In 1989, when Apple started to feel the competition rise, the engineers were called together. Each engineer was to propose a number of ideas that might find their way into future versions of System 6. Three colours of index cards were handed out: blue, red and pink. Ideas on the blue notes were ideas that could quite easily be implemented and (most) of these ideas were implemented in System 7. Ideas on red notes were already quite complicated ideas, but ones that may be achievable in a number of years. Ideas on pink notes were the most extravagant and experimental ideas. Obiously, project Red was the team of engineers who worked on implementing the ideas that were found on the red index cards. Almost all of the ideas that were proposed on these index cards eventually found their way into OS X.

Another project was Project Pink.[3] Pink was destined to be an experimental operating system. Apple and IBM founded the company Taligent in 1992 to support the development of this system [5]. TalOS became an object-oriented system based on the Mach kernel [1]. Eventually, TalOS did became a true, ground-up object oriented environment, but an environment that ran under Windows 95. As Apple realised that also this project was not what they were looking for, it ended up being taken over by IBM [5].

Yet another project was Copland, which started around 1994 [22]. Copland was to become System 8. Copland would run on top of a microkernel named Nukernel and it would have quite a number of new features. It would have a HAL (*H*ardware *A*bstraction *L*ayer), support for symmetric multiprocessing with preemtive multitasking and virtual memory support with memory protection. Furthermore, the idea was taken on to include the networking support and the file system as extensions of the kernel to improve performance. Other niceties would include a search engine which could search through both metadata as well as content and a system that supported live updates (as such, key components of the system could be updated without the need to restart or even close down any running programs). Sadly, also this project was eventually cancelled and System 7.7 was simply renamed System 8, with a few minor additions that had already been successfully implemented in Copland.

After Copland was cancelled in 1996, Apple was in even more desperate need for a new operating system. System 8 was nothing but an embellished version of System 7.7, with the legendary instability as a result. Apple started to consider other operating systems among which Windows, Solaris, Linux[4] and BeOS. The latter proved to be the most attractive: it was geared toward the end-user, it offered impressive performance (which was especially true in the field of multimedia, one of the strong points of BeOS) and it had almost everything that Apple ever dreamt of. It had memory protection, preemtive symmetric multiprocessing, it ran on both the PowerPC and x86 architecture, it had a file system called BeFS that handled metadata on a file system level and more. Some key features were missing however. This included for example support for file and printer sharing. As such, Apple was not willing to pay too much and Be asked too much. This resulted in Apple eventually breaking off the negotiations and turning their attention to another worthwhile OS.

The other contender who had caught the eye of Apple, NeXT, was a company founded by Steve Jobs after he left Apple in 1986 and it so happened to be that NeXT offered a very attractive NeXTStep operating system. So what was NeXT? Well, NeXT was an entirely new personal computer designed for universities, colleges and research labs, very much based on the ideas of the original Macintosh. It was based on Mach 2 and 4.3BSD and it had a window server that was based on Display PostScript.[5] It offered a UNIX command line and a great GUI. There was a Dock from which programs could be launched, the

[4] MkLinux was a serious attempt to port Linux to the Power Macintosh.

[5] Essentially, the screen is rendered as one big PDF, hence explaining why OS X has such well-integrated support for PDF.

concept of hiding an application instead of quitting an application was present and it allowed to drag-and-drop complex objects such as images or movies to other programs. Also, it had an object-oriented device driver framework which eventually resulted in I/O-Kit [10]. Additionally, it used Objective-C (which is a superset of C that is simpler than C++) and it ran on multiple architectures. Similar to System 7, it was possible for a single binary to run on all these different architectures (*i.e.* a fat binary).

So what is this Mach we have been talking about in both Copland and now in NeXTStep? Mach [1] is a new kernel foundation that was developed at the Carnegie Mellon University as a possible replacement for the kernel of UNIX. In many ways, Mach was and is special. For starters, it is a clean microkernel where only the most basic of kernel services run in a privileged kernel mode. This is in sharp contrast to the monolithic kernel found in UNIX. Yet even though it uses a microkernel instead of a monolithic kernel, it retains binary compatibility with 4.3BSD. As it is a research kernel, it has all sorts of nice and brand-new features. It has full support for preemptive multiprocessing on multiprocessor systems, protected virtual memory, extensive support for IPC and a real-time scheduling framework to offer (soft) real-time support.

Apple decides to go with NeXTStep and Apple takes Steve Jobs back on board, where he shall eventually regain his position as CEO. In the years to come, two projects coexist. On one hand, the development of System 8 is continued to provide a temporary System 9. On the other hand, Apple teams up with Sun to form OpenStep and starts the development of Rhapsody. This operating system is to incorporate all the interesting features found in both the Red and the Pink project, as well as the features from Copland, worthwhile technology of System 8 and the technology of the brand new NextStep acquisition.

The first team, the team that tries to keep System 8 alive and extend it into System 9, does not just sit around, but acts as a playground for some of the original Copland ideas. Most new maintenance versions of System 8 see the adoption of some of the features of Copland. Some of the things that get incorporated include the HFS+ file system and the support for protective virtual memory. Even though they did improve System 8 and afterwards System 9, all the improvements could not hide that internally System 8 and 9 were old beasts and nothing made this more obvious than the cooperative multitasking that kept causing the system to be somewhat unreliable.

The second team, the one working on Rhapsody, quite quickly comes forth with the Darwin XNU[6] kernel. This kernel represents the culmination of all the research that Apple and others around the globe had spent onto developing a potent kernel. Influences and ideas that have finally made it into the Darwin XNU kernel include, but are certainly not limited to, the Mach kernel, work on FreeBSD, NetBSD and OpenBSD and research and work on respectively Copland and System 8.

[6] XNU stands for X is not Unix.

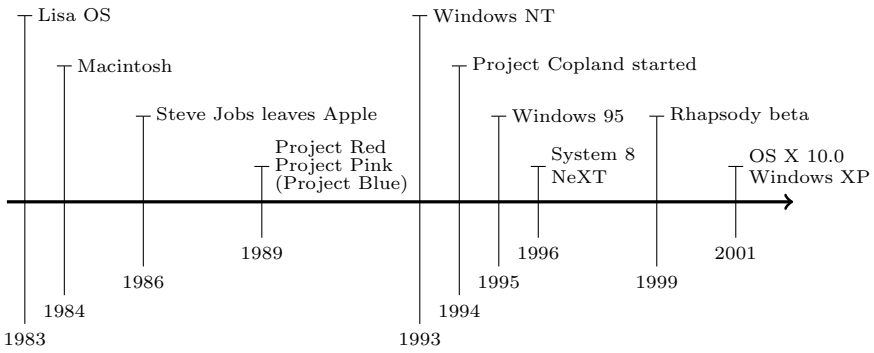


Figure 2: Timeline with noteworthy events that led to OS X.

This time around, thanks to a solid research base and a dedicated development team, things worked out as planned and in 2001 a brand new operating system is launched: OS X (pronounced OS Ten). However, saying that it is a successor of System 9 does not give it the credit that it deserves. Every nook and cranny has been looked at and redeveloped from scratch.[7] Finally, Apple had a new operating system that could easily face the fierce competition from Windows XP and that had a new code base that is nowadays proving itself to be very expendable and easily maintainable.

[7]The system wasn't even binary compatible with System 9, meaning that developers had to redevelop all their programs for OS X. A challenging approach, but one that proved to result in high quality programs developed with high quality programming standards.

3 Darwin XNU Kernel

Simply put, the Darwin XNU kernel[8] is a monolithic kernel with the associated performance, but with the maintainability of a microkernel. The kernel consists out of *(1)* the Mach kernel along with the drivers who are built on top of I/O kit and *(2)* out of a number of BSD functionality such as the virtual file system and the networking, which sit on top of the Mach kernel. This composition can be seen in Figure 3, which is a slightly adapted version of the image found in [11]. This strict separation allows for easy exchange of BSD features to newer versions and has in general the same benefits as a layered software design.

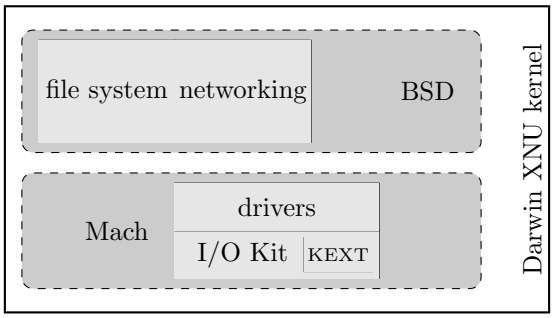
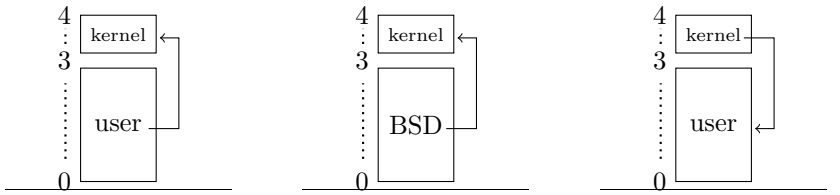


Figure 3: The composition of the Darwin XNU Kernel

However, performance-wise, keeping such a strict separation would be a true penalisation. Consider the case where we have a strict separation and the BSD features are to be found in user space. Furthermore, assume that the kernel is permanently loaded in the upper half of the memory.[9] Thus, in our assumption, only the Mach kernel is loaded at the top of the memory and both the user process and the BSD features are in user space. Whenever a user needed access to either the file system or the networking, this would mean that the user sends a call to the BSD feature – Figure 4(a). The kernel sees this call and swaps the user process with the BSD process – Figure 4(b). This BSD process performs a call to the Mach kernel and may return a result according to the function that was called. The kernel notices the result and returns the control to the user process by swapping the BSD process with the user process – Figure 4(c). As such, a simple call to the file system would result in two swaps. Otherwise, if the BSD subsystem were part of the kernel, the user process would not need to be swapped as the kernel could process the request immediately.

[8] Also often called the Darwin kernel or the XNU kernel; we will use them interchangeably.

[9] This is not the case with the Darwin XNU kernel. We shall later see that OS X actually performs a 4/4 switch instead of the 3/1 switch found in Linux, Windows, ... Basically, this means that the kernel is swapped for a user process and does not remain loaded at the top of the memory. Taking this into account, the problem we present would only be aggravated. However, for educational purposes and for generality in the discussion of monolithic kernel versus microkernel, we assume a kernel that is loaded at the top of the memory.



(a) user calls some BSD function, which is passed through the kernel

(b) user process is swapped with the BSD subsystem which performs the action via the kernel

(c) user process is swapped back (and may receive a response)

Figure 4: Performance problem of a true microkernel: two swaps

Such problems are inherent to microkernels and it is for this reason, the development team of the Darwin XNU kernel decided to not limit themselves to a true microkernel but add the BSD features to the kernel. This improves performance, without sacrificing the maintainability since, at a code level, the Darwin XNU kernel can be seen as a microkernel architecture. Let us now take a deeper look at the individual parts of the Darwin kernel.

3.1 Mach

3.1.1 Mach Basics

Mach [1] was a project that ran at the Carnegie Mellon University from 1985 to 1994. It was one of the first microkernels and was designed from the bottom-up to be fairly small and highly extensible. It was seen not simply as an extension to the existing UNIX kernel, but as an actual replacement for the kernel in the 4.3BSD version of UNIX. The Mach microkernel introduced quite a number of features that were unique in 1986, the year when the first version of the Mach kernel was introduced.

For starters, Mach introduced support for multiprocessors by replacing the idea of a single process by the idea of a single task with one or more threads. Though at present this idea is widely applied, in 1986 it was common practice to use coroutine packages to manage multiple contexts within a single process [21]. Yet, since the kernel itself is not aware of these coroutines, they cannot be scheduled on multiple processors. This limits the use of such coroutines to *(1)* simplifying the code by splitting the task up in multiple coroutines and *(2)* overcome the need to implement such a coroutine as a process, which would incur a significant overhead. As Mach knows the concept of a thread, it is possible to obtain an additional benefit from splitting a single task (or process) into multiple threads; *(3)* each thread can be separately scheduled, allowing multiple threads of a single task to be executed in parallel.

A task is rather similar to a process; it is a collection of system resources. These include a virtual address space and a set of port rights.[10] By nature, a task is a high overhead object, similar to a process. A task has no life of its own, since only the threads of a task execute instructions. A thread is by design a low overhead object since it does not need to keep track of most of the system resources (in most cases it only needs to keep track of its register state). All threads share the system resources of their task, whereas different tasks do not share any resources without an explicit act of sharing. Actions performed on a single thread only affect the thread (*e.g.* suspending), whereas actions performed on a task affect all the threads of that task. Regardless of the difference between a process and a task, tasks share the concept of UNIX processes to create new tasks by forking, ensuring that tasks are related to each other in a tree structure [1].

Another improvement in Mach was a new virtual memory design. This virtual memory design incorporated, among others, a copy-on-write virtual copy operation. Copy-on-write is a way of promising that data will be copied as soon as the need arises, *i.e.* as soon as writes occur to either the source or the destination. If no writes occur, the data is simply read from the original location. Most of the copy operations between tasks are used to share data between tasks, thus changes to the copied data rarely occur. Using a copy-on-write prevents an expensive physical copy and significantly improves performance.

Sharing memory between tasks by using copy-on-write and read-write techniques can be done in a controlled fashion in Mach. This is achieved via an inheritance mechanism that can be set to shared, copy-on-write (default) or none. This setting will be used by the children address maps of the parent. For example, if a parent does not allow sharing then neither will its children.

Mach also offers memory protection. Like inheritance, protection may be specified on a per-page basis, where each page has two protection values: the current and maximum protection. Each protection level is a combination of read, write, and execute permissions. The maximum protection provides an upper bound for the value that the current protection may assume. If ever the maximum protection drops below the current protection, then also the current protection is lowered. The use of two protection values allows for some flexibility: as long as a page obeys the maximum level, it is free to set a more tight current protection without limiting its children to this tighter protection.

All the information about the memory is stored inside a memory map, where we have a single memory map per task. These maps can be recursive, so one map may be part of a bigger map. These maps govern information such as the level of protection on the memory pages and information on how inheritance should occur. Additional information is also kept in these maps such as what pagein behaviour is desirable (*e.g.* sequential or random caching of related pages). Thanks to these maps, it is possible to offer a fine-grained level of control.

[10] For example, a task A has receive rights on his own port and may have send rights on the port of some other task B. This grants task A the right to contact task B.

Yet another improvement was the introduction of a capability-based interprocess communication facility [1]. That is, there can be communication between processes, but only if these processes have the permission (*i.e.* capability) to do so. Mach interprocess communication facility is defined in terms of ports and messages and provides both security and location independence. In essence, this means that different processes may be located on physically different processors or machines without the processes ever being aware of this fact.

The port is the basic transport abstraction provided by Mach. A port is a protected kernel object into which messages may be placed by tasks and from which messages may be removed [1]. A port can be seen as a big mailbox. Every task has a mailbox in which other tasks may drop new messages.[11] Depending on the number of messages that are already in the mailbox of task A, it may take some time before task A reads the newly sent message from task B, but given enough time, every message in the mailbox will be read. All threads of a task may be attempting to receive messages from a given port, but only one of those threads can receive any given message. That is, as soon as the message is read by any thread, the message is removed from the mailbox. Sending messages to another task can be done in a synchronous or asynchronous fashion.

Task B can also indicate what should happen if the mailbox of task A is full. Task B can either *(1)* block until it can deliver the message, *(2)* have an error code returned or *(3)* it can ask the kernel to accept the message and to forward it to task A as soon as possible. If task B chooses the latter option, it cannot send another message to task A until it receives confirmation from the kernel that the previous message has been successfully posted to task A.

This particular implementation of interprocess communication, or IPC for short, is capable of transparently extending across network boundaries, which gives Mach a definite edge when it comes to distributed computing where tasks may be found on physically separated machines. The IPC facility further integrates with the virtual memory design to allow transferring large amounts of data via copy-on-write techniques.[12] Originally, Mach only provided message queues for IPC. In later versions this has been extended to include semaphores, notifications, lock sets and remote procedure calls.

A final improvement is directly based on how UNIX was implemented in those days. UNIX had seen a proliferation of different mechanisms for managing objects and resources which caused access to different resources to be quite inconsistent. Whereas the original implementation of UNIX only had the concept of pipes as a composition tool [19], a stunning array of additional facilities had been introduced including, but certainly not limited to, sockets, streams and various forms of semaphores. The Mach kernel tried to return UNIX to

[11] As said, this is not true in Mach, or Darwin for that matter. If task B wants to send a message to task A, it first needs to have permission to do so. Once task B has this permission, it can send any number of messages to task A. Hence the term capability-based.

[12] The benefit of such copy-on-write techniques only increases when looking at distributed computing, where the communication between processes on different machines might be slow.

the original model of consistent interfaces by introducing an object-oriented approach. This approach provided the basic abstractions for the main concepts which could then easily be extended. Thanks to the use of inheritance for these extensions, inconsistencies can be avoided. The four basic concepts for which Mach provides an abstraction are:

- *task* (with a UNIX process represented as a task with a single thread);
- *thread* (with all threads of a task sharing access to the task resources);
- *port*: a communication channel that is logically a queue of messages protected by the kernel and that provides two operations: *send* and *receive*;
- *message*: a typed collection of objects used by threads to communicate.

More of these abstractions have been identified by others. The Darwin XNU kernel assumes that the Mach kernel has three additional abstractions [11]:

- *time*: objects such as clocks, timers and waiting behaviour;
- *address space*: the range of memory addressable by a task;
- *memory object*: the internal units of memory management, these include name entries and regions and these are representations of potentially persistent data that may be mapped into address spaces.

3.1.2 Mach in Darwin XNU

Now that we have seen a rough sketch of the inside of Mach, let us look at how everything is retrofitted in Darwin. As indicated at the beginning of this section, the Darwin kernel does not use Mach as a microkernel but links it with other kernel components into a single kernel address space for performance reasons. Furthermore, a large number of modifications have been made to the original Mach kernel to support the requirements of the Darwin kernel. This is self-explanatory; a long time has passed since the inception of Mach. During that time, UNIX has evolved considerably. For example, it would be foolish to still represent a process of UNIX as a task with a single thread, since nowadays a process can have multiple threads. Above all, Mach is valued for its abstractions, its extensibility and its flexibility in its implementation as part of Darwin.

The fundamental services and primitives of the Darwin kernel are based on Mach 3.0 [11]. Among others, Mach provides

- highly parallel execution, including preemptively scheduled threads and support for SMP[13];
- a flexible scheduling framework, with support for (soft) real-time usage;

[13]Where SMP stands for *S*ymmetric *M*ulti*P*rocessing: it allows two or more identical processors to connect to a single shared main memory.

- a complete set of IPC primitives, including messaging, RPC, synchronisation and notification;
- security and resource management as a fundamental principle of design; this is enforced by virtualising all resources.

As Mac OS X incorporates quite a number of BSD features, it can run normal BSD processes and the associated POSIX threads. In Mac OS X processes and POSIX threads, are implemented on top of Mach tasks and threads, respectively.

For scheduling, Darwin uses the flexible framework for thread-scheduling policies that is part of Mach. To be precise, the scheduler is based on the scheduler found in OSF MK 7.3 [6] – a version of Linux built on top of OSF[14] Mach 3 – which is an extension of the scheduler found in Mach 3. Following policies are found in OS X. The time-sharing policy constantly changes the priority of a thread to balance its resource consumption against other time-sharing threads. Fixed priority threads execute for a certain quantum of time or time slice and are then put at the end of the queue of threads of equal priority. The time slices of a thread can be set to infinity to ensure that the thread runs until it blocks or until it is preempted by a thread of higher priority.

OS X also provides time constraint scheduling for real-time performance.[15] This policy allows to specify that a thread must get a certain time quantum within a certain time period. Since all policies can be used in Darwin, the different priorities can be divided into four bands: normal, high, kernel mode and real-time. As an academic example, assume that the band of high priorities ranges from 30 to 45. If a task has priority 36, we say that is is in the high band of priorities. Tasks can migrate between priority levels in a given band (mostly as a result of the time-sharing policy) but cannot migrate to another band under normal conditions.

The virtual memory implementation of Mach has largely remained unaltered in OS X. The idea is to have a virtual memory that does not take different kinds of storage into account. This allows for a uniform control of all the memory, whether this memory resides in the main memory or is swapped to secondary storage (or, even remote memory for that matter).

The Darwin XNU kernel makes extensive use of IPC. This should not come as a surprise, as an important element of the Mach philosophy is to have communication between tasks. This is implemented in a peer-to-peer fashion where tasks may take on the role of both client and server and can thus, respectively,

[14] Open Software Foundation

[15] Once again, this is soft real-time; the kernel will make every effort to honour the request, but it cannot be guaranteed. More specifically, if the thread requests reasonable resources it will most likely be granted the request, but if it behaves in a compute-bound fashion it may be demoted to a normal thread. This behaviour is a deliberate design decision since every thread is allowed to register as a real-time thread. The behaviour acts as a fail-safe to avoid such blatant behaviour and ensures that a thread cannot *set* its own priority but only *influence* the choices that the kernel will make.

both request and provide services for other tasks. Darwin not only uses messages, but also uses (counted) semaphores, notifications (similar to semaphores), lock sets (where a lock is obtained by a thread for the duration of a transaction) and RPC. These were not part of the original version of Mach, but they were present in the latest version of Mach, Mach 3, on which Darwin is based. An additional difference with the original version of Mach is that in Darwin, all messages are delivered in an asynchronous fashion.

Clearly, a lot of the low-level capabilities of the Darwin XNU kernel come directly from the Mach microkernel. Even though a number of (significant) modifications have been made, Darwin holds tight to the philosophy of the Mach kernel. Darwin benefits from the rich features that the Mach microkernel provides as well as from the high extensibility and maintainability of Mach. Nevertheless, Darwin clearly differentiates itself from the original Mach kernel by adding new features to the kernel space. Some of these are discussed next.

3.2 I/O Kit

The I/O Kit is a collection of system frameworks, libraries, tools, and other resources for creating device drivers in Mac OS X. The I/O Kit builds heavily on the work done by NeXT to develop a new driver framework that improves maintainability by allowing drivers to be constructed in an object-oriented fashion. Furthermore, a great deal of attention was paid to providing driver developers with a large set of libraries that can easily be accessed in order to reduce the driver development time. I/O Kit is based on an object-oriented programming model implemented in a restricted form of C++ that omits features unsuitable for use within a multithreaded kernel [10]. Some excluded features are: exceptions, multiple inheritance, templates and run-time type information. This version of C++, based on embedded C++, was chosen over Objective-C to give driver developers access to a more commonly used language without sacrificing object-oriented development.

A special part of the I/O Kit is the KEXT (*K*ernel *EXT*ension) manager. This part of the I/O Kit allows Darwin to dynamically match and load drivers for hardware that has just been plugged in. This not only makes the deployment of drivers a lot easier, but it also improves the stability of the system. Since drivers can be dynamically loaded, they can also be dynamically unloaded. In the unfortunate event that a driver crashes, the driver can simply be reloaded without affecting the system. A very desirable function, indeed.

Normally, most hardware should not need custom drivers based on I/O Kit. Most standard devices that conform to well-defined and well-supported specifications are already supported. Even if a device requires a custom driver, it may only need a user-space driver, provided it uses a FireWire or USB connection to the computer [22]. This is in part due to the extensive support for generic devices that is already present in the I/O Kit. Most of the time, the I/O kit already takes care of a considerable part of the communication without the need

for an explicit driver. Furthermore, on top of the kernel lies a considerable layer of so-called core services. These core services provide even more functionality and better abstractions. As such, a considerable amount of drivers can be developed without the explicit need for I/O kit or kernel space accessibility.

3.3 BSD

The BSD portion of the Mac OS X kernel is derived primarily from FreeBSD and it takes care of a large number of features that are not provided by the Mach kernel. Even though a lot of the code needs to be adapted to plug parts of BSD into the Darwin kernel, it provides massive benefits. The Darwin kernel profits from all the work that has gone into developing BSD. As such, it obtains a very stable set of features with high performance behaviour. Furthermore, it allows OS X to remain up-to-date with only a small set of engineers working on the kernel. As new versions of BSD are released, they can be put in place of the old version to obtain the latest functionality and performance improvements. Also, when certain parts of the BSD code needs to be rewritten to better serve the goal of OS X, the Apple engineers can rely on a large and skilled user base to obtain the necessary information to get the work done. From a user's point of view, the BSD application environment in OS X is similar to, but not the same as, a traditional user-level environment on a BSD-based Unix system [22].

Let us browse through a number of the features provided by the BSD portion of the kernel. An important part of the BSD facilities are the POSIX APIs and the POSIX IPC mechanisms. Thanks to this code base and a considerable amount of work by Apple, the latest version of OS X, Leopard, is 100% POSIX compliant. As such (and thanks to quite a few other modifications), OS X Leopard has received the UNIX 03 certification [25], which makes it the first BSD-based OS to receive such a certification and one of the few operating systems in the world to have this certification. This is especially important for enterprise customers, where the conformation to standards is an important part of the selection criteria.

Another feature provided by the BSD subsystem is the TCP/IP stack with the associated BSD sockets for RPC and firewalling[16] support. Once again, the code in use on OS X is not a direct port, but is a tailored version to the specific needs of OS X. One such extension is the Network Kernel Extensions (NKE) mechanism. This mechanism allows to extend the networking architecture of the system by means of loadable kernel modules that interact with the networking stack [22]. This can be done without the need to reboot, modify, or recompile the kernel. Through the use of NKE, it is possible to monitor and modify network traffic [13]. Obviously, the NKE is a specialised version of KEXT, specifically tailored to the networking features provided by the Darwin kernel.

[16] In Mac OS X 10.4 Tiger, the firewall is a direct application of the BSD IP firewall or "ipfw". Such support is still present in the current version of OS X alongside another and more user-friendly firewall based on application rights. The availability of "ipfw" gives advanced users easy access to a custom firewall with support for bandwidth throttling.

Yet another part of OS X that is obtained through the BSD subsystem is the virtual file system (VFS) and numerous file system implementations. One of the file systems that is inherited from BSD is the file-system-independent VFS-level journaling mechanism, which is being used to provide journaling to the HFS+ file system of OS X. VFS was first pioneered for the UNIX platform by Sun Microsystems in 1985 [20]. Since then, it has been successfully applied in a great number of operating systems, among which Linux and BSD. The power of VFS lies in the possibility to abstract the file system from the upper part of the kernel. As such, each and every file system can be accessed and manipulated in a uniform, file system independent fashion, allowing multiple file systems to coexist in a clean and maintainable manner.

Even though a considerable amount of features are found in the BSD subsystem, it cannot be said that the Darwin kernel runs a well-defined BSD kernel (*e.g.* as a single Mach task or otherwise). This is even more strange if one sees that the Darwin kernel can execute BSD processes. Strictly speaking, a BSD process does not execute: it corresponds to exactly one Mach task, which contains one or more Mach threads and it is these threads that execute. As such, even the BSD-style *fork()* operation uses Mach calls to create a task and a thread [22]. From a user's point of view, Darwin offers (almost) everything a normal BSD kernel would. From a technical point of view, BSD is a subsystem that runs on top of the Mach kernel.

3.4 Additions

Some important features of the Darwin XNU kernel were left out in the above overview. This subsection looks at these features and details why they are so important for the Darwin kernel, and OS X in general. For starters, we will look at Mach-O files and Universal Binaries. The first is used by OS X to implement several types of system components whereas the second provides an archive of binaries for multiple architectures. It will be important to look at how they interact and especially where they do not interact. Historically, a lot of incorrect or skewed information has found its way onto the world wide web on how these universal binaries (or fat binaries) actually work.

Another feature that is often mentioned in the same breath, but one that is technically very different, is how the Darwin kernel is capable of running 32-bit and 64-bit binaries intertwined. At the surface, this problem seems to coincide with the problem that is solved by universal binaries. However, universal binaries only solve the problem of choosing the correct architecture to run on a given processor. They do not solve the problem of how one can run different kinds of binaries for the same architecture at the same time.

3.4.1 Mach-O Files and Universal Binaries

The Mach Object Format (Mach-O) is the Mach counterpart of the Executable and Linkable Format (ELF) as found in UNIX and of the Portable Executable (PE) as found in Windows (*i.e.* .exe, .sys, .dll). Mach-O and ELF are both standard file formats for executables[17], shared libraries, object files and others. Mach-O shares the same benefits that ELF enjoys; it is a highly flexible and extensible format and above all it is not bound to any particular architecture. This independence on the architecture is one of the major benefits of Mach-O and this enables Universal Binary archives to work their magic.

The file layout of Mach-O and ELF share some similarities. Both start with a header that contains basic information. In the case of Mach-O, the header contains information on the particular architecture the file is made for, what kind of file it is (*e.g.* an executable, a library ...), how many load commands there are and a magic number to identify the format. The load commands of the Mach-O format corresponds with the section header table of ELF and links to the different sections that can be found in the segments that follow the load commands. A similar subdivision into segments and sections can be found in ELF. A visualisation of the Mach-O format can be found in Figure 5.[18]

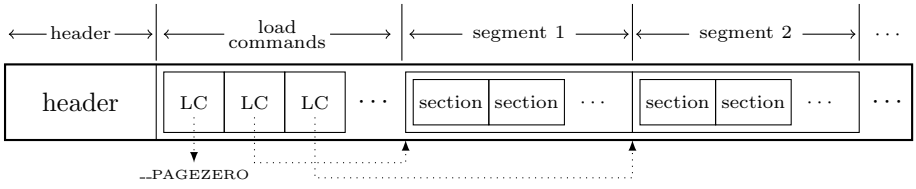


Figure 5: A typical Mach-O file.

Fat binaries or Universal Binaries provide an archive for a number of Mach-O files. This is necessary, since any Mach-O file can only support a single architecture [12] at a time. A fat binary prepends a number of concatenated Mach-O files with a fat header and a corresponding number of fat architecture specifiers. The fat header contains a new magic number to differentiate it from an ordinary Mach-O file and contains information on the number of different architectures it supports. Each fat architecture specifier provides a description of the destination architecture along with an offset of where to find the respective Mach-O file and a size to indicate the length of the Mach-O file. By definition, the fat files are always big endian for compatibility reasons. The specific Mach-O files are big or little endian depending on the specific architecture they are for. A visualisation of the resulting archive can be found in Figure 6.

[17] In OS X these are bundles, which allow grouping all resources needed by the executable.

[18] In the picture, the first load command links to an unreadable segment starting at address zero and named __PAGEZERO. Its existence is to cause a bus error if a NULL pointer is dereferenced. More information can be found in the man pages of the GNU Linker `ld`.

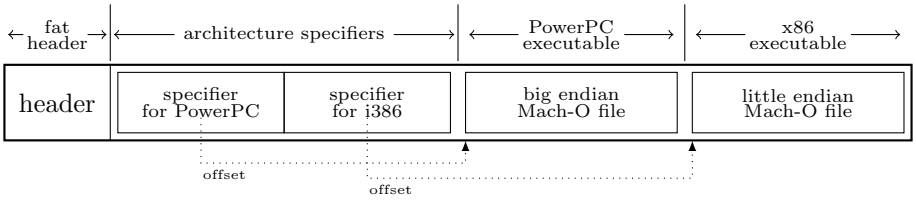


Figure 6: A Universal Binary for PowerPC and x86.

A common misconception is that Mach-O is essential for Universal Binaries. Yet anyone who concludes that Apple might as well implement ELF instead of Mach-O and still retain the benefit of Universal Binaries would be absolutely correct.[19] There is no need for Apple to stick to Mach-O for some magic universal feature this format is providing, since there simply is no such feature. However, moving to ELF would mean rewriting the kernel to support ELF. This is most likely one of the main reasons for Apple to stick to Mach-O.

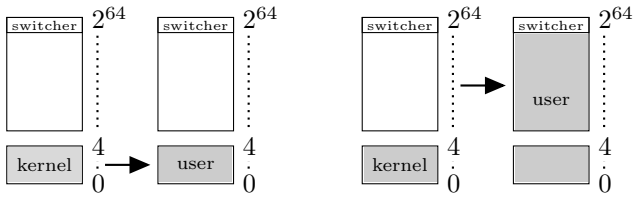
3.4.2 64-bit programs on 32-bit kernel

The Darwin kernel has been in a constant transition to a 64-bit kernel. Apple already faced such a transition when moving from 32-bit PowerPC to 64-bit PowerPC. That transition was never finalised and now a similar transition from 32-bit x86 to 64-bit x86_64 awaits. Every such transition faces two big problems. On one hand, user tasks need to be ported from 32-bit to 64-bit. On the other hand, drivers need to be ported from 32-bit to 64-bit. Apple chose to move the user space to 64-bit, with backward compatibility for 32-bit user tasks while retaining the driver space in 32-bit with 64-bit support for drivers who need it.

Supporting 32-bit drivers requires a 32-bit kernel, since drivers are implemented as kernel extensions. But how does such a kernel support 64-bit user tasks? Actually, the trick is quite simple and is an added benefit of the 4/4 swap that the Darwin kernel employs. Since the kernel is not loaded at the top of the memory, but is swapped in its entirety for the user process, a 32-bit kernel requires only a small extension to be able to swap itself for a 64-bit user task. After all, the difference between 64-bit and 32-bit is mostly in the memory range that can be addressed. As such, the kernel has a small 64-bit extensions or switcher that resides at the top of the memory and is always present. It is this extensions that manages interrupts and switches between tasks and threads in the case when a 64-bit process is run. This can be seen in Figure 7. This simple trick is thus nothing more than an exploit of an inherent performance weakness of the Darwin kernel: the swapping of the kernel for a user task.[20]

[19] The only requirement is an executable format that provides architectural independence.

[20] Do not confuse this with the two swaps needed by a microkernel. The two swaps we are talking about here are a deliberate design decision and allow a normal 32-bit user task access to the full 4GB range of memory since the kernel is not occupying the upper region.



(a) swapping the 32-bit kernel for a 32-bit user task

(b) swapping the 32-bit kernel for a 64-bit user task

Figure 7: Swapping the 32-bit kernel for a 32-bit user task.

There are some caveats. A process is either 32-bit or 64-bit. As such, a 64-bit program cannot load a 32-bit plug-in. For some application developers this poses serious problems in transitioning to 64-bit as they rely heavily on plug-ins, both from in-house as well as third parties [8]. Nevertheless, this simple implementation allows 32-bit drivers to access 64-bit hardware through special constructs and allows a more easy transition from 32-bit to 64-bit tasks.

3.5 Exploring the Kernel With DTrace

DTrace is a dynamic tracing framework developed by Sun Microsystems [17]. It can be used for tracing, observing and debugging the kernel and applications. What makes DTrace different is the ability to dynamically hook itself to any possible function call and/or return. This can be done with very little performance penalty; the overhead of tracing is often negligible and deactivated probes have no performance impact at all. This makes DTrace applicable on production systems, where tracing facilities can simply be turned on when the need arises. Though a number of ports are underway for different platforms, until recently DTrace was only available under Solaris. Porting DTrace to another platform is quite a challenge, as it tightly integrates with the kernel. This tight integration is required to enable the benefits we listed previously.

As of Leopard (OS X 10.5), Apple ported a significant subset of DTrace to OS X. Apple did extend DTrace with a number of mechanisms to limit the applications that can be traced. Such applications, or more specifically their tasks, may set the `P_LNOATTACH` flag to ensure that the task it not tractable. One notable example of such a program is iTunes[21] and any traces of iTunes will simply be ignored.

[21] It may seem like a strange choice to exclude a simple multimedia program from tracing, but still allow more fundamental parts of the system such as file system interactions and kernel operations to be traced. For starters, any lower level traces of iTunes will also fail (*i.e.* it behaves as a 'stealth' application: only certain quantitative information can be obtained). On the other hand, the choice to exclude iTunes is not so strange after all. A considerable part of today's multimedia formats carry DRM protection. Allowing these DRM tasks to be traced would allow for an all too easy hacking attempt.

DTrace allows to write tracing scripts in a language that is a subset of C, called D (not to be confused with D, see [3]). D exhibits only a small number of functions and variable types that are found in C. This subset is extended with tracing-specific functionality. These D programs resemble `awk` programs in that they consist of a set of actions, *i.e.* they are action-based programs and not the more traditional top-down structured programs.

We will not be giving a thorough overview of DTrace since this falls beyond the scope of this paper. The purpose of this section is to ensure that the reader is aware of the presence of DTrace and has a notion of its capabilities. Later on in this paper, we will use DTrace to observe key elements of the inner workings of OS X. We will only consider so-called one-liners; programs consisting of a single line. This will give us all the information we need and suffices for most of the simpler tasks.

One example of such a usage would be to demonstrate that the BSD subsystem is an essential part of the Darwin kernel. By using DTrace, we can intercept all the system calls to the kernel. Moreover, we can get the exact name of the system call that is being made, as well as how often that system call is performed in a set timespan. To do this, we need the one-liner

```
dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'
```

Dissecting this one-liner is not so hard. The '`-n`' argument allows us to specify the probe(s) we want to trace. We want to trace all the probes that are connected to a system call entry (`syscall::entry`). All we want to do with this information is count how often this system call is made. Running this command for a second may give us the following result:

```
dtrace: description 'syscall:::entry' matched 427 probes
^C
1   fstat64
1   sendto
1   seteuid
2   geteuid
2   recvfrom
3   socket
3   semwait_signal
5   munmap
6   mmap
7   ioctl
9   getuid
11  sigaltstack
11  sigprocmask
12  read
33  select
39  __sysctl
```

This very small trace offers a wealth of information. For starters, DTrace reports that 427 probes are activated. This means that a program has access to 427 different system calls to access kernel functionality, since each probe will have attached itself to a single system call. This information already confirms that OS X has about a hundred more system calls than BSD[22], *i.e.* BSD (and the associated system calls) are only part of the Darwin kernel. When browsing through the list of system calls returned by DTrace, it is striking that only one system call, `semwait_signal`, is not part of the BSD subsystem. Clearly, programs running on the Darwin kernel extensively use the BSD subsystem. This is only obvious, as some major parts of OS X that are built on top of the Darwin kernel also have their roots in the open source community.

The trace also tells something about what was going on in the system at the time of the trace. Since most of these system calls are BSD system calls, the explanation of these calls is readily available from [24]. The status of some file was checked (`fstat64`), the user permissions were read and set (`seteuid, geteuid, getuid`) and a mapping was established – and removed – between some address space and a file or shared memory object (`munmap, mmap`). Furthermore, a socket was established (`socket`) and messages were sent and received from a socket (`sendto, recvfrom`). All this information was obtained during a single trace that took no more than a second. Clearly, DTrace is a very potent tool for observing and debugging any user program and/or kernel.

Let us look at another example. The one-liner

```
dtrace -F -n 'pid$target:libSystem*::entry,
              pid$target:libSystem*::return
              /ustackdepth < 6/ {}'
          -c "ln example linked"
```

will allow us to trace the flow (`-F`) of the creation of a hard link up to a stack depth of 6. The result[23] of this trace is

```
CPU FUNCTION
  1  -> _exit
  1    -> _sysenter_trap
  0  <- __cxa_atexit
  0  -> getopt$UNIX2003
  0  <- getopt$UNIX2003
  0  -> stat
  0    -> _sysenter_trap
  0      -> lstat
  0        -> _sysenter_trap
  0          -> cerror
```

[22] FreeBSD has slightly less than 330 different system calls [24].

[23] Note that this result has been truncated for visualisation purposes and running this trace on another system will result in a more elaborate trace.

```
0          <- cthread_set_errno_self
0        <- cerror
0        -> stat
0          -> _sysenter_trap
0            -> cerror
0            <- cthread_set_errno_self
0          <- cerror
0          -> lstat
0            -> _sysenter_trap
0              -> cerror
0              <- cthread_set_errno_self
0            <- cerror
0            -> link
0              -> _sysenter_trap
```

We can immediately deduce that this trace was run on a multi-core system, indicated by the presence of multiple CPUs (*i.e.* we have CPU 0 and 1). One of the first steps is to parse the command line (`getopt$UNIX2003`). After determining the status of the file(s) (`lstat, stat`), the hard link is eventually established (`link`). Similarly, if we were to establish a symbolic link, this would eventually call `symlink`. This examples serves as a demonstration for the ability to trace a single command and as yet another example of how Darwin behaves in a very BSD-like manner.

4 HFS Plus File System

The HFS Plus file system is the preferred file system in OS X. HFS stands for Hierarchical File System, a naming that dates back to System 2. With the release of System 2.1, the old Macintosh File system (MFS) was replaced by HFS. HFS was a new file system designed for use on hard drives and shares some interesting ideas with MFS as well as several welcome additions. Like MFS, files on HFS can have multiple forks (normally a data and a resource fork[24]) and files are referenced by IDs rather than by their names (which can be up to 255 characters long). Welcome additions included the hierarchical aspect, which allowed directories to be nested, while the use of B*-trees instead of a flat file to store all the directory and file listings allowed for significantly better performance. Most of the HFS implementation is 32-bit, with the exception of the file directory. As a result, the number of files that can be stored on a drive with HFS is limited to 65 535.

In 1998, HFS Plus was introduced along the introduction of Mac OS 8.1. HFS Plus is also known as HFS+ or Mac OS X Extended. Though architecturally similar to HFS, HFS Plus provides several important benefits. Noteworthy features include a metadata journaling mechanism, hot file clustering, on-the-fly defragmentation and intermingled access control lists (ACLs) and UNIX permissions. Not all of these features were present in the original implementation of HFS+ since HFS+ has continued to evolve over the last decade. We shall consider the state of HFS+ as of OS X 10.5 Leopard and we will as such not detail when each and every addition was made.[25] Regardless, it is important for the reader to realise that HFS+ is somewhat of an old file system with a lot of features retrofitted onto the original implementation of HFS+.

In the remainder of this chapter, we will skim through the implementation details of HFS+. As soon as the most fundamental concepts have been presented, we will move our focus to those features that set HFS+ apart. As we shall see, most of these features are geared towards the GUI, ease of use and performance related aspects of the system.

4.1 HFS+ Overview

HFS+ was introduced to provide better support for larger drives, better internationalisation, better security and even more features that provide additional benefits to the user. These benefits will be highlighted as we come across them. To be able to better understand the following paragraphs, you might take a look at Figure 8(a), which details the concepts that are in use by HFS+.

[24] The resource fork is mostly in place to support the GUI by storing icons and other auxiliary data separate from the main data.

[25] To be more precise, we shall look at HFSX (introduced in 10.3) with additional support for ACL (introduced in 10.4).

Like any other file system, HFS+ considers a volume as an instance of a file system. Such an instance may span a single hard disk, part of a hard disk or multiple hard disk (*e.g.* through the usage of RAID). Space on an HFS+ volume is allocated to files in fundamental units called allocation blocks. All these blocks are (conceptually) numbered sequentually. (For an overview of the concepts used in a HFS+ volume, refer to Figure 8(a)).

This area has seen major improvements compared to HFS. HFS is limited to 2^{16} blocks instead of the 2^{32} allocation blocks provided by HFS+. This value is immediately linked with the minimum block size and hence with the minimal space that each file consumes. For example, assume a hard drive with a capacity of 2^{37} bytes or 128 GB. When the file system can only keep track of 2^{16} blocks, it follows that each block has size $2^{37}/2^{16} = 2^{16}$ bytes = 4 MB, hence each file takes up a multiple of 4MB. Given a file of 5MB, this file would span 2 blocks and consume 8MB. Similarly, a file of 256 bytes would need a single block and would consume 4MB. With 2^{32} allocation blocks, the minimal block size drops dramatically to $2^{37}/2^{32} = 2^5$ bytes = 32 bytes,[26] making HFS+ far better suited for large hard drives (*i.e.* $>$ 1GB).[27]

HFS+ also uses the concept of an extent [14]. An extent is a range of continuous blocks. Each such an extent is described by an extent descriptor which is a pair of start location and a number of allocation blocks. For example, the extent $\langle 50, 100 \rangle$ describes 100 blocks starting at block number 50 on the volume. An eight-element array of HFS+ extent descriptors constitutes an extent record. HFS+ can use such a single extent record, *i.e.* up to the first eight extents of a file, as part of the file's basic metadata. For a file with more than a single extent record, HFS+ maintains additional extent records but these are not part of the metadata. Later we will see how this first extent record can be used to speed up access to files.

Yet another concept is the concept of clumps. Whereas a block is a fixed-size number of contiguous sectors, a clump is a fixed-size number of contiguous blocks. Though this concept is related to extents, it is not completely equal since an extent may be a variable-size number of contiguous blocks. During allocation of space to a fork (remember that a file consists out of at least two forks, a data fork and a resource fork), HFS+ may decide to do this allocation in terms of clumps instead of in terms of blocks to avoid external fragmentation,

[26] Using such a small block size would not be wise. A small file of 32MB would already require a million blocks. Keeping track of all these blocks is not at all easy and all these blocks may be found scattered over the hard drive surface, making I/O performance terrible. Then again, most hard drives use sector sizes of 512 bytes, hence using a block size of 32 bytes would still consume 512 bytes since a single sector may only contain a single block. It suffices to say that a lower bound on the block size of 32 bytes serves no practical usability, yet shows how the system has still plenty of potential for even larger hard drives. In practice, the block size employed by HFS+ is a multiple of the sector size of the medium and defaults to 4KB.

[27] Internal fragmentation remains a problem and HFS+ does not have techniques such as tailing in ReiserFS or fragments such as in UFS to alleviate the problem. However, since HFS+ is mostly targeted at a user workflow, files tend to be big and internal fragmentation can be experimentally verified to be less than 5% in most cases and on average around 3%.

i.e. it provides a means to artificially augment the block size to better support larger files and distribute them over fewer separate parts.[28] To fully understand the benefit of clumps, it is important to recall that a file is fragmented when the blocks of the file are scattered across the volume. If we split a file in small blocks, we get a lot of blocks that we need to allocate. If we split the same file in larger blocks (*i.e.* clumps) we need far less clumps and as such the file cannot become as fragmented.

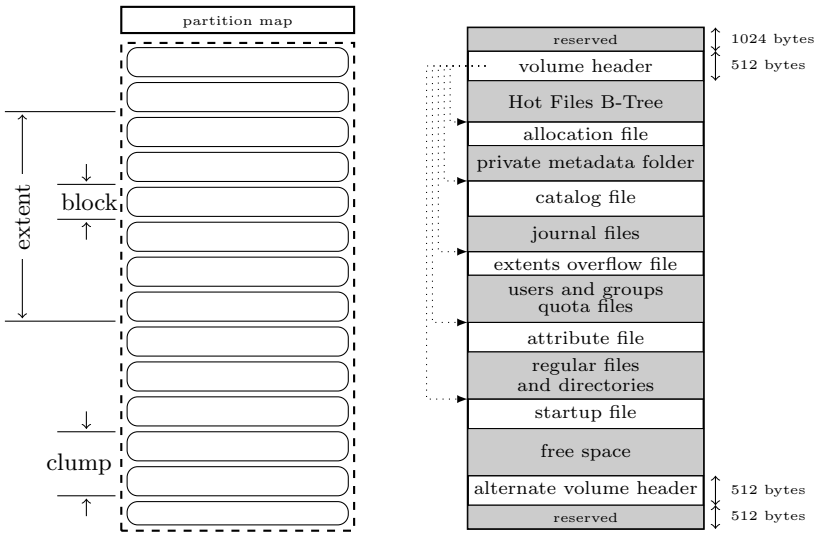
Conceptually, this rounds up how HFS+ sees a volume and a summary can be found in Figure 8(a) [22]. Now the attention will move to the structure of a HFS+ volume, *i.e.* the logical ordering. Like any other modern file system, HFS+ will extensively use B-Trees for quite a few parts of the implementation. We will not discuss such B-trees and will simply assume that the reader is familiar with them. In the implementation, HFS+ uses a variant of B+ Trees, which themselves are a variant of B-Trees. The biggest difference in the variant of B+ Trees that is used is that N keys point to N children instead of the conventionally used $N + 1$ children; the first child is dropped. We will use the name B-Tree and B+ Tree interchangeably, as only their implementation differs and not their purpose. Most B+ Trees used in HFS+ are implemented using a single fork and are not user accessible.[29]

Figure 8(b) [22] gives an overview of the structure of a HFS+ volume. Unless otherwise specified, the location and size of each block is not representative. However, for spacing, colouring and consistency (with [22]), the proposed layout was used. All white areas and the reserved upper and lower areas are special areas. Apart from the volume headers (and the reserved areas), the system has no idea where to look for these special areas and will rely on the volume header to link to these structures. As such, the loss of the volume header would make the system irrecoverably damaged, which is why an identical alternate volume header is stored at the end of the volume as a back-up.

Let us take a closer look at the special areas that the volume header links to and what they are used for. The allocation file tracks whether an allocation block is in use on a volume. This file is implemented as a bitmap where a simple bit provides the necessary information (used or free) for each block. Since this is an actual file, it can be distributed over several blocks [14]. This makes it a lot easier for the allocation bitmap to grow and shrink as needed. Having the ability to grow and shrink is a requirement to allow the HFS+ volume to dynamically shrink or expand in both volume size and block size. In order to speed up the allocation of new space, each HFS+ volume keeps track of a roving pointer which serves as a hint to where to start looking for free bits [22].

[28] Keep in mind that though clumps are used to prevent fragmentation, extents are used to determine the degree of fragmentation. This follows from the fact that every clump is an extent, but not vice versa. Regardless of the number of contiguous blocks, we still see them as part of a contiguous whole and as such we cannot speak of external fragmentation.

[29] The exception being the B-Tree used for Hot File Clustering, which has two forks (but still, the resource fork is simply empty) and which is user accessible.



(a) concepts used in HFS+ (b) structure of a HFS+ volume

Figure 8: Concepts and structure of a HFS+ volume.

The catalog file[30] is used to maintain information about the hierarchy of files and folders on a volume [14]. This file is implemented as a B-Tree and it stores information on the name of each file/directory along with their unique catalog node ID or CNID (similar to an inode in UNIX). Each catalog file leaf node keeps track of two records for every file/directory:

- The *file/folder record* contains the standard file/folder metadata such as:
  - the CNID;
  - UNIX permissions;
  - the first extent record (*i.e.* the first eight extents) of the data and resource forks (in case of a file);
  - the number of children (in case of a folder).
- The *file/folder thread record* is used to keep a link between a file/folder and the parent directory and further allows to look up a file/directory through its CNID.

The extents overflow file keeps track of which blocks belong to a file's forks by maintaining a list of extents [14]. This is only done when a file has more than

[30] Once again, we explicitly use the term "file", suggesting all the benefits a normal file offers such as easy distribution over several blocks and a homogeneous file system that only needs to consider blocks, extents and clumps.

eight extents since these first eight are maintained by the catalog file. Similar to the catalog file, a B-tree is used for the extents overflow file. Since HFS+ employs proactive and retroactive means to limit fragmentation, most files fit into the extent record that is maintained by the catalog file. One additional use of the extents overflow file is to store information on the bad block file. This bad block file is used to keep track of bad sectors (and thus blocks) on the disk. Being a file, it has a CNID and the same benefits as any other file. However, the bad block file is not like any ordinary file since it does not appear in the catalog file nor is it a special file as it does not appear in the volume header. This is done on purpose to prevent any kind of user interference. The bad block file links all the bad blocks together and these bad blocks are respectively marked as used in the allocation file, to provide a simple, but effective means of ensuring that the file system does not use these blocks.[31]

The startup file is a special file intended to hold information needed when booting a system that does not have built-in (ROM) support for HFS Plus. Nowadays, this file serves almost no purpose as it dates back to the transition period from HFS to HFS+ and it will not be further discussed.

The last special file, the attributes file, is used for implementing named forks. A named fork is simply another byte-stream to extend or support the data and resource forks. Since OS X 10.4, these named forks have seen their first application with the implementation of access control lists (ACLs), which are designed to be fully compatible with the file permissions as used in Windows XP and upwards. Theoretically, future named forks may have their own extents but for now support in OS X 10.4 and upwards is limited to inline attributes which can fit within a single B-Tree node, limiting their size to 3802 bytes [22].

4.2 On-The-Fly Defragmentation

Modern file systems are far more robust to fragmentation than their predecessors. Most of them accomplish this through proactive measures. HFS+ is one of those file systems. We have already discussed one of the proactive measures, namely the usage of clumps, or artificially bigger blocks, that better support large files and limit the number of extents that such files need. Another such improvement in HFS+ is delayed allocation. Though delayed allocation is used primarily to reduce I/O [9], a side-effect is that interwoven write operations of two different data streams are untangled and written as single extents, as seen in Figure 9.

[31] Simply marking bad blocks as used would not work. One of the steps in a consistency check is to find blocks that are marked as used but that do not appear in any file extent and marking them as free again. This would involuntary re-enable these bad blocks for usage.

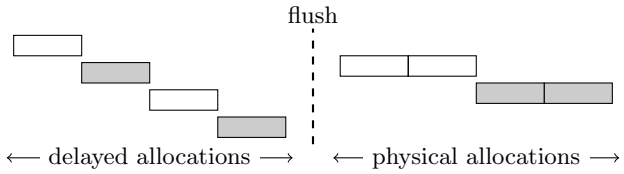


Figure 9: Preventing fragmentation through delayed allocations.

A more recent addition to HFS+ extends the file system with a retroactive tool to combat fragmentation. This feature is called on-the-fly defragmentation (or incorrectly, adaptive hot file clustering). When a user file is opened, on-the-fly defragmentation will check the file for a number of conditions:

- the file system must not be read-only;
- the file system must be journaled;
- the file is a regular file;
- the file is not already open;
- the file fork being accessed is nonzero and no more than 20MB;
- the fork is fragmented into eight or more extents;
- the system has been up for more than three minutes.[32]

If all these conditions are met, on-the-fly defragmentation will try to relocate the file to contiguous blocks. Because on-the-fly defragmentation is only considered when a file is opened and because it only works on smaller files with noticeable fragmentation, the performance penalty is kept to a bare minimum, ensuring that the user does not notice it at all.

The combination of proactive and retroactive techniques to tackle fragmentation ensure that fragmentation is a non-issue with HFS+. Test results indicate that on any given volume using HFS+ the number of unfragmented files is roughly above 99.5%. Whereas file systems that only use proactive measures are bound to become fragmented over time, HFS+ proves to be unaffected by file system aging.

4.3 Adaptive Hot File Clustering

Hot File Clustering is an adaptive, multistage clustering scheme based on the idea that frequently accessed files are both small in size and in number [22]. To improve performance of these small, frequently accessed files, they are moved near the volume's metadata, into the metadata zone. This reduces the seek time for most accesses. As files are moved into the metadata zone, they are

[32]This last condition ensures that bootstrapping has finished.

also defragmented (allocated in a single extent), which further improves performance [14]. Like many special files, a B-Tree is used by the hot file clustering to keep track of which files are to be found in the hot file area, a part of the metadata area.

These so-called hot files are tracked by their temperature, which is an indication of their relative importance. The temperature of a file is given by

$$temperature = \frac{\text{number of bytes read during recording phase}}{\text{file size in bytes}}.$$

The migration of these hot files in or out of the hot files area of the metadata zone is a gradual process, consisting of several phases. During the *(1)* recording phase, the system keeps track of the block read from file forks to identify candidate hot files. The recording phase takes a predetermined amount of time. During the *(2)* evaluation phase, the candidate hot files are analysed to determine which new hot files will be merged with the current set of hot files. In the *(3)* eviction phase, older and less frequently used hot files are moved out of the hot file area to make room for new hot files. In the last phase, the *(4)* adoption phase, the actual move occurs where the newest and hottest files are moved to the hot file area by allocating contiguous space for each such file.

The confusion with on-the-fly defragmentation often arises because hot file clustering also defragments the files. However, only a select number of files (*i.e.* hot files) get this special treatment. On-the-fly defragmentation will defragment all the files, given that they satisfy the conditions that we previously listed (see page 30).

4.4 Journaling

Every HFSX volume may have a journal. This journal can be used to speed up recovery after the volume has not been safely unmounted. In such a case, the problem may be that certain transactions to the file system did not complete. If that is the case, a number of files may already have been altered (*e.g.* a node of a B-Tree may have been changed) while other essential parts (such as rebalancing the tree) may have been interrupted. The problem is thus one where the system sees an action as atomic, but the file system needs to make numerous changes to perform this action. In the case where a number of changes need to be undertaken to complete a single action, the journal makes it quick and easy to restore the volume to a consistent state without the need to scan all the structures. Inconsistencies may arise from *e.g.* a power failure.

A journal works by grouping modifications into transactions that are recorded in a journal file. For every transaction, the modifications to the special files are first copied to a special journal area. Once the journal copy of the changes has been completed, the changes are written to their final destination, a process that is often referred to as a replay. After the replay, the journal marks

the transaction as committed and it is removed from the journal. In the event of an interruption, the journal will either have a completed journal copy and continues the replay or it was in the middle of making the journal copy and it simply ignores the transaction. The journaling applied in HFS+ is an example of write-ahead journaling, where changes are first written to the journal before they are moved to their final destination. Important to note is that journaling in HFS+ is only used for the volume structures and the metadata and not for contents of a fork. As a result, these last are not protected by the journal.[33]

Journaling was retrofitted into HFS+ by introducing a VFS-level journaling layer in the kernel [22]. This layer exports an interface that can be used by any file system to incorporate a journal. The log file used by journaling in HFS+ is a circular on-disk buffer. The journal header keeps track of which parts of the buffer is active and contain pending transactions. Since only a single part of the buffer can be active, this can easily be done by tracking the start and the end of the active part. This does imply that the buffer may never be allowed to be completely full, in order to prevent ambiguity when start = end.

The journal info block (`/.journal_info_block`) keeps track of the journal buffer and the journal header location and size (which are stored together in the `/.journal` file). Because both are files, they can easily be tracked by the allocation bitmap and the catalog file. Under normal circumstances, the journal files are not looked up through the catalog file but through a special reference found in the volume header that links directly to the journal info block.

A single transaction stored in the buffer consists of several blocks and the location where these blocks need to be written. This is represented by a block list header, which describes the number and sizes of the blocks. Immediately after the block list header follow the contents of these blocks.[34] An overview of the linking and structures that are used can be found in Figure 10 [14].

4.5 In Retrospect: Spotlight

Spotlight is a search engine that is an integrated part of OS X. It allows desktop searching that works almost instantly from two different points of view. For starters, *(1)* search results are delivered almost instantly, which is accomplished by indexing metadata for future queries. Also, *(2)* newly created files are indexed almost instantly by relying on the virtual file system to report any changes. To accomplish the task, Spotlight is a collection of both user-level and kernel-level mechanisms. In essence, Spotlight can be seen as consisting out of four *major* parts [22]:

[33] In another footnote we already briefly mentioned that one of the tasks of a consistency check is to see which blocks on the volume are marked as used but are not referenced by a file extent. Now that we know how journaling works, it is fairly easy to see how this strategy goes hand-in-hand with journaling these critical files.

[34] Since the block list header can only be a limited size, it is possible to group multiple block list headers and associated blocks for a single transaction. This can be indicated in the successive headers by marking one of the fields stored in these headers.

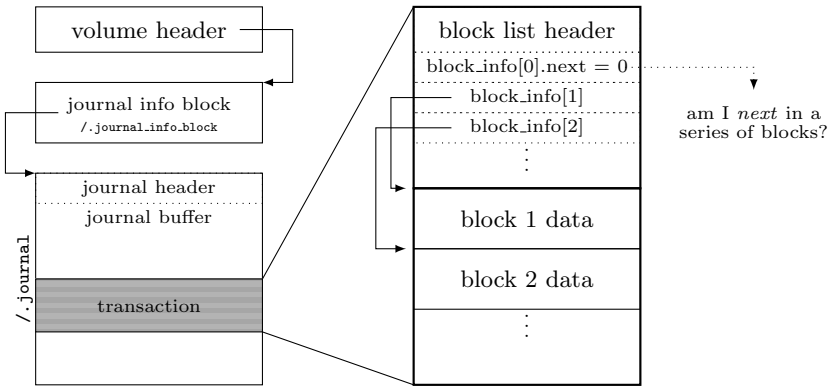


Figure 10: A simple transaction consisting out of a single block.

- the `fsevents` change notification mechanism (file system changes);
- a metadata store and content index for each volume;
- metadata importer plug-ins (to allow 3rd party format support);
- the Spotlight server task, `mds`.

Whenever VFS detects a change to a filesystem (*i.e.* creation, deletion or modification) it triggers `fsevents` to report this change to any subscribers, among which is the Spotlight server. The Spotlight server then relies on the metadata importers to obtain useful metadata on the file that was changed. The use of these importers is important for a number of reasons. To start, *(1)* it allows third party vendors with proprietary file formats to be integrated in Spotlight, without the need for Apple to know the specifics of the file format. Equally important is that, in general, *(2)* files contain very limited metadata since most of the metadata is provided by the user (who is lazy). Since each plug-in is targeted at a single file or a few file formats, it is possible to obtain quite a lot of relevant metadata in an automated fashion.

After obtaining the required metadata, this information is stored in a content index (*i.e.* to allow searching based on the content of a file) and metadata store (*i.e.* to allow searching based on the metadata associated with a file). This database of metadata allows for quick lookup of data based on all kinds of provided metadata such as the author of the document, the name of the file and possibly even (parts of) the contents of the file. Now that Spotlight has a database filled with metadata, other tasks can query the Spotlight server task to obtain results for their queries. Since OS X integrates Spotlight into the menu bar, it is easy for the user to query for files. There is also a programming interface that allows programs to look up information through the Spotlight server. As such, third party programs can programatically parse the results and proceed with these results as they please.

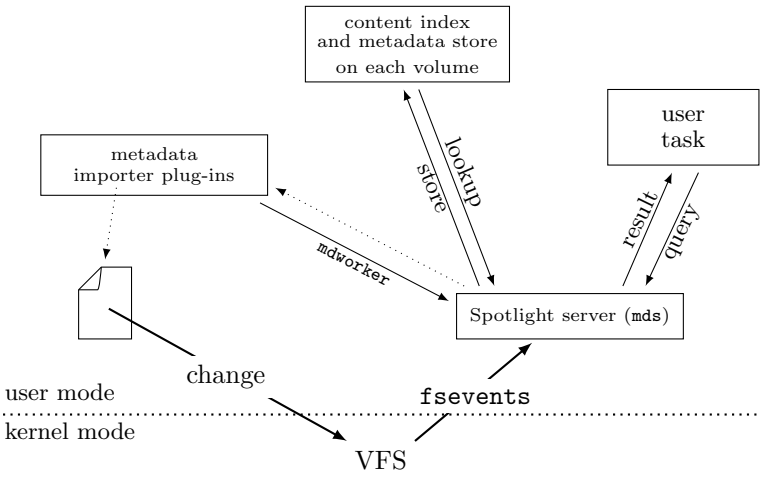


Figure 11: Inner workings of Spotlight.

A drawback of Spotlight is that it works on a file level of granularity. As such, it can only link back to these files. This poses a problem, since certain things, such as a calendar, are often stored in a single file. Nevertheless, users want to be able to search through all their calendar events and preferably want to have immediate access to these events through the search results proposed by Spotlight. The workaround used in the default applications of OS X is simply to store individual files for these entities and let Spotlight index these files instead of the single huge file.

A place where Spotlight excels is by relying on VFS to be notified of changes. By doing so, Spotlight is capable of indexing any and all file systems, whether this is FAT32, HFS+ or even AFP, which is a network file system protocol. This has the very desirable effect of being able to search all file systems, instead of only those file systems that support metadata indexing, such as BFS.[35]

Using DTrace, we can verify that Spotlight indexes files as soon as they are written/changed/deleted. for this, we use the D script[36]:

```
dtrace -n 'syscall::open*:entry
          { printf("%s %s",execname,copyinstr(arg0)); }'
```

[35] BFS or BeFS was the native file system of BeOS. Though the metadata indexing was very similar in features, it was limited to the BFS file system. Spotlight is often seen as to improve on the BFS metadata indexing. This statement contains more truth than some believe, since it was the same engineer who played key roles in the design implementation of both BFS and Spotlight [22].

[36] In the script, we need `copyinstr(..)` to copy the result from kernel space into user space since even the root user does not have access to kernel space.

The resulting (truncated) trace is:

```
CPU     ID  FUNCTION:NAME
  0  17606  open:entry TextEdit /../oss.txt
  0  17606  open:entry mds /../oss.txt
  0  17606  open:entry mds .
  0  17606  open:entry mdworker /../oss.txt
```

The above trace, where a file named 'oss.txt' is written to disk by the program 'TextEdit', clearly shows how mds and mdworker immediately kick into action as soon as the file system event is triggered.

4.6 Exploring HFS+ Using `hfsdebug`

The tool `hfsdebug` is written by Amit Singh, the author of [22]. The naming suggests that the main purpose of this tool is for debugging, but this is not the case. First and foremost, the tool is an excellent companion when trying to explore and observe the HFS+ file system. Because the tool uses the internal structures of HFS+ to obtain its information, it can offer a plethora of information that would not be accessible through more conventional means.

For example, the tool can be used to explore the records in the B-Trees of the special files we discussed earlier. As such, the tool can be used to obtain information about the extents used in the file system and the degree of fragmentation (this functionality of `hfsdebug` is used in the appendix, where we examine the fragmentation level of HFS+ on a number of different volumes). Another use for the tool is to find out how many and which files are marked as hot files. For example, running '`hfsdebug -H -t 3`' results in the following information (truncated for visualisation purposes):

```
# Top 3 Hottest Files on the Volume
rank  temperature       cnid  path
1           16409       6745  .../Generic RGB Profile.icc
2            4097    1546502  .../WebKit.app/.../Current
3            3770     546506  .../private/etc/paths.d/TeX

A total of 3699 Hot Files are being tracked.
```

This is run on a machine with a calibrated monitor and which is most often used to browse the internet and typeset texts in LaTeX. Clearly, the choices that were made are sound ones. At the top of the list is the calibration profile. This profile has not changed in months, yet is read whenever the user logs in. The second hot file links to one of the files found in the application bundle of WebKit/Safari (*i.e.* the files needed to launch the browser WebKit). The third file contains the location of the current TeX binary on the system.

If desired, the tool can also be used to obtain extensive information on a single file. For example, running '`hfsdebug /bin/ls`' will return information on the '`ls`' file such as

```
# Catalog File Record
accessDate          = Sun Nov 16 11:49:46 2008
```

the last time the file was accessed (found in the catalog file record),

```
#BSD Info
linkCount           = 1
```

the number of links to the file (part of the BSD information),

```
# Data Fork
logicalSize         = 73696 bytes
totalBlocks         = 18
fork temperature    = no HFC record in B-Tree
clumpSize           = 52
extents             =  startBlock   blockCount      % of file
                          0x3a8a32         0x12       100.00 %
                      18 allocation blocks in 1 extents total.
```

and information on the actual size, fragmentation, clump size and temperature of the file (information that is linked to the data fork of the file). It is this last information block that is the most useful for our purpose. The first two are handy additions, but we need not rely on `hfsdebug` to obtain the information (*i.e.* one could use the `stat` command to obtain the latest access date).

Appendix

A. Trying Out System 1 through Mac OS 9

First-hand experience is the most valuable experience of them all. Luckily, System 1 through System 7 can be run without any problem on any ordinary computer, whether this computer has an OS X, Windows or Linux operating system. This is made possible thanks to the program Mini vMac [18] which is a miniature Macintosh emulator. Current day performance greatly excels compared to the Macintosh computers of those days, so the emulation should run at near-instant speed on any modern computer. Another great resource for running predecessors of Mac OS X is the System 1 Headquarters [26] which not only provides instruction and a rough overview of the system, but also links to the required Mac Plus ROM file and the System 1 image.

As of Mac OS 8, things are a bit trickier. If you happen to have an old Macintosh with a PowerPC G3 or G4 lying around, it suffices to install Mac OS X 10.4 or earlier and run the Classic environment. In this environment, Mac OS 8 and Mac OS 9 can run as native applications. You may be uncomfortable with running OS X 10.x. In such a case, it is possible to install Mac OS 8 or Mac OS 9 as the default operating system on a Macintosh PowerPC G3 or G4.

If you do not have an old Macintosh with a PowerPC G3 or G4 or you do not have a Macintosh altogether, you can use SheepShaver [2] to emulate a PowerPC. Once again, this program runs on any ordinary computer, whether this computer is powered by OS X, Windows or a Linux operating system. Note that performance can be quite slow, especially when running Mac OS 9, since SheepShaver reports itself as a PowerPC G4 100Mhz processor. Nevertheless, SheepShaver allows to experience these operating systems first-hand.

B. HFS+ Fragmentation Results

Fragmentation on HFS+ volumes was tested using the `hfsdebug` program. The tool has been used to gather information on a number of different HFS+ volumes with different users, different usage patterns and different file system ages. Keep in mind that file system testing is quite tricky. Different users exhibit different usage patterns and comparing multiple filesystems would require some form of artificial aging, hence it is almost impossible to compare different filesystem in a truly objective way. These results should therefore not be seen as a comparison in any way, but as random samples of HFS+ volumes.

Volume Size	39.06 GB	111.47 GB	232.44 GB
Volume Type	boot/root	boot/root	boot/root
Volume Age	347 days	137 days	835 days
Objects			
number of files	372396	499742	690014
number of folders	80295	90232	141082
Data Forks			
non-zero	369587	493252	685285
fragmented	671	841	876
allocated space	26.449 GB	30.540 GB	115.429 GB
actually used	25.535 GB	29.387 GB	113.848 GB
difference	0.914 GB	1.152 GB	1.581 GB
average file size	72.45 KB	62.47 KB	174.20 KB
Resource Forks			
non-zero	1311	447	1200
fragmented	2	2	24
allocated space	72.77 MB	24.36 MB	112.15 MB
actually used	69.65 MB	23.28 MB	109.19 MB
difference	3.12 MB	1.08 MB	2.96 MB
External Fragmentation			
data forks	3.46 %	3.77 %	1.37 %
resource forks	4.29 %	4.45 %	2.64 %
weighted average	3.46 %	3.77 %	1.37 %
Internal Fragmentation			
data forks	0.182 %	0.171 %	0.128 %
resource forks	0.153 %	0.447 %	2.000 %
weighted average	0.181 %	0.171 %	0.131 %
% unfragmented	99.819 %	99.829 %	99.869 %

Table 1: Overview of HFS+ fragmentation results.

The fragmentation statistics were obtained by running '`hfsdebug -s -t 5`'. The volume age and size is determined by running '`hfsdebug -v`', which returns detailed header information on the HFS+ volume.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, 1986. Usenix Association.

[2] Gwenole Beauchesne. Sheepshaver. `http://gwenole.beauchesne.info/en/projects/sheepshaver`. *As retrieved on December 16, 2008*.

[3] Kris Bell, Lars Ivar Igesund, Sean Kelly, and Michael Parker. *Learn to Tango with D*. Apress, 2008.

[4] Jim Carlton. *Apple:: The Inside Story of Intrigue, Egomania, and Business Blunders*. Crown Business, 1997.

[5] Sean Cotter and Michael J. Potel. *Inside Taligent Technology*. Addison Wesley Longman, 1995.

[6] Barbou des Places F. B., Stephen N., and Reynolds F. D. Linux on the OSF mach3 micro-kernel. In *Proceedings of the First Conference on Freely Redistributable Software*, pages 33–46, 1996.

[7] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

[8] Apple Inc. *64-Bit Transition Guide*. `http://developer.apple.com/documentation/Darwin/Conceptual/64bitPorting/index.html`. *As retrieved on December 16, 2008*.

[9] Apple Inc. *File-System Performance Guidelines*. `http://developer.apple.com/documentation/Performance/Conceptual/FileSystem/Articles/FilePerformance.html`. *As retrieved on December 16, 2008*.

[10] Apple Inc. *I/O Kit Fundamentals*. `http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/chapter_1_section_1.html`. *As retrieved on December 16, 2008*.

[11] Apple Inc. *Kernel Programming Guide*. `http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/About/chapter_1_section_1.html`. *As retrieved on December 16, 2008*.

[12] Apple Inc. *Mac OS X ABI Mach-O File Format Reference*. `http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html`. *As retrieved on December 16, 2008*.

[13] Apple Inc. *Network Kernel Extensions Programming Guide*. `http://developer.apple.com/documentation/Darwin/Conceptual/NKEConceptual/index.html`. *As retrieved on December 16, 2008*.

[14] Apple Inc. *Technical Note TN1150: HFS Plus Volume Format*. `http://developer.apple.com/technotes/tn/tn1150.html`. *As retrieved on December 16, 2008*.

[15] Apple Computer, Inc. *Creating Fat Binary Programs*. `http://developer.apple.com/documentation/mac/runtimehtml/RTArch-87.html`. *As retrieved on December 16, 2008*.

[16] Owner W. Linzmayer. *Apple Confidential: The Real Story of Apple Computer, Inc.* No Starch Press, 1999.

[17] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall, 2006.

[18] Paul C. Pratt. Mini vmac. `http://minivmac.sourceforge.net/`. *As retrieved on December 16, 2008*.

[19] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17(365-375), 1974.

[20] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings Summer 1985 USENIX Conference*, pages 119–130, Portland OR, USA, 1985.

[21] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The itc distributed file system: principles and design. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 35–50, New York, NY, USA, 1985. ACM.

[22] Amit Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006.

[23] C.P. Thacker, E M McCreight, B W Lampson, R F Sproull, and D R Boggs. *Alto: A Personal Computer*. McGraw- Hill, 1982.

[24] The FreeBSD Documentation Project. *FreeBSD Developers' Handbook*. `http://www.freebsd.org/doc/en/books/developers-handbook/`, 2008. *As retrieved on December 16, 2008*.

[25] The Open Group. *Single UNIX Specification version 3 certificate Grant for OS X Leopard 10.5*. `http://www.opengroup.org/openbrand/certificates/1190p.pdf`. *As retrieved on December 16, 2008*.

[26] Dan Vanderkam. System 1.0 headquarters. `http://www.nd.edu/~jvanderk/sysone/`. *As retrieved on December 16, 2008*.