What is Mac OS X?

Amit Singh

About

This is a PDF version of "What is Mac OS X?", a document that attempts to give a *hacker over-friendly* answer to the question "What is Mac OS X?". This PDF is provided due to the numerous queries I received for a printable version. Since I did the PDF conversion in a hurry, it is not very nice. Even the chapters do not start on a fresh page. I apologize for such shortcomings.

Please note that I would *not* be updating, correcting, or maintaining this PDF version in *any* way whatsoever, primarily because of lack of time. The online version *will* be maintained. Please do send me your feedback, corrections, and suggestions on the content even if you are reading this PDF version. I would not be able to address problems with the PDF conversion itself, though.

Current Online Version

http://www.kernelthread.com/mac/osx/

Contact

http://www.kernelthread.com/contact

What is Mac OS X?

Introduction

This document attempts to give a *hacker over-friendly* answer to the question "What is Mac OS X?". My original motivation in writing this was that somebody had proxy-volunteered me to give a talk introducing Mac OS X to the Linux Users Group at my work place. While thinking over what to say to those folks, most (if not all) of whom are strangers to Mac OS X, I decided to write a supplementary document that they could refer to at their leisure.

Since I moved to Mac OS X, I have had a few discussions with people who are curious about Apple and Mac OS X, but have not used the platform. Often, what they think is a somewhat distorted, perhaps even incorrect version of the "truth": there are many areas in which they think the Mac is *worse* or *better* than it *really* is. Consider (with colloquialisms preserved):

- Macs are good only for graphics/media work.
- Macs suck for hackers.
- There is very little software for the Mac.
- Macs are overpriced/not worth the price.
- Mac OS X is Unix.
- Mac OS X is not Unix.
- Mac OS X rocks. Linux is crap.
- Mac OS X is crap. Linux rocks.

Some such curiosities and questions are addressed in this document.

Target Audience

This document does not aim to regurgitate Marketing KoolAid, *not that there's anything wrong with it*™, but is intended primarily as an introduction to Mac OS X of those members of the technical community who are not familiar with it. You can think of it as a somewhat low-level *taste* of Apple's operating system. Consequently, some parts are fairly

technical, and the implicit assumption is that you are familiar with fundamental concepts of one or more of BSD, Mach, UNIX, or operating systems in general. In many cases I have made no attempt to provide background details of the concepts referred to in the discussion.

Background

I currently use Mac OS X as my primary operating system. However, I am not a longtime Apple user. I bought my first Apple computer in April 2003. I have used and played with a number of operating systems, and have used Linux, Solaris, *BSD, and the Microsoft Windows family as my primary platforms at different times. The operating system that I have used the most is Linux. I have not been involved in religious riots involving operating systems, and I *really* believe that all these systems (including Mac OS X) have flaws, but they all have good things to offer.

I would like to point out what this document is *not* intended to be. It is:

- ... not a reference or guide to Mac OS X.
- ... not a detailed comparison of Mac OS X with other operating systems. Although I do express some opinions on Mac OS X vs. Linux, etc. in the closing section, the overall discussion is almost exclusively on Mac OS X. The reader could use the information herein to compare Mac OS X with other systems.
- ... not even a reasonably complete discussion of Mac OS X in any way.
- ... not a Mac OS X endorsement. While Mac OS X serves most of *my* needs well, your situation might be different.
- ... not an attempt to get you to switch (sic) to Mac, even though I do sum up *my own* reasons for using Mac OS X in the last section of this document (*Conclusion: Why Mac OS X?*).

As stated earlier, I do not have much experience on the Mac platform. It is very likely that there are errors in my discussion of Mac OS X. I would appreciate if you bring these to my attention.

I hope you would understand that a reasonably complete description of an operating system would take one or more *books*, this document represents a (small) subset of what could be possibly discussed about Mac OS X - essentially what I could come up with during the post-Christmas extended weekend as I was forced to stay indoors by the relentless rain in the Bay Area. I *might* make this a work in progress.

Most importantly, please be aware that I do not work for Apple, and am not associated with them in any way, other than being a customer. Even my customer service experience with Apple has been rather incongruous for that matter.

The opinions herein are solely mine and do not represent anybody else, including my employer in any way.

Contents

A Brief History of Mac OS X

The goal of this document is not to trace the history of Mac OS X in great detail, so this section would be brief.

All of Steve Jobs' operational responsibilities at Apple were "taken away" on May 31, 1985. Soon (within weeks), Jobs had come up with an idea for a startup for which he pulled in five other Apple employees. The idea was to create the *perfect* research computer (for Universities and research labs). Jobs had earlier met up with Nobel laureate biochemist Paul Berg, who had jumped at Jobs' suggestion of using a computer for various simulations. Although Apple *was* interested in investing in Jobs' startup, they were outraged (and *sued* Jobs) when they learnt about the five Apple employees joining Jobs. Apple dropped the suit later after some subsequent mutual agreements. The startup was NeXT Computer, Inc.

Jobs unveiled the first NeXT Computer (running NEXTSTEP 0.8) on October 12, 1988, in San Francisco, although a mature release of the operating system took another year. The name "NEXTSTEP" has gone through a number of capitalization permutations, so we shall simply use "NEXTSTEP". NEXTSTEP 1.0 shipped on September 18, 1989, over two years later than what Jobs had first predicted and hoped for. NEXTSTEP was based on Mach 2.5 and 4.3BSD, and had an advanced GUI system based on Postscript. It used Objective-C as its native programming language, and

included the NeXT Interface Builder.

In the fall of 1990, the first web browser (offering WYSIWYG browsing *and* authoring) was created at CERN by Tim Berners-Lee on a NeXT computer. Tim's collaborator, Robert Cailliau, later went on to say that "*... Tim's prototype implementation on NeXTStep is made in the space of a few months, thanks to the qualities of the NeXTStep software development system ...*"

NEXTSTEP 2.0 was released exactly a year later on September 18, 1990 (with support for CD-ROMs, color monitors, NFS, on-the-fly spell checking, dynamically loadable device drivers, ...). 2.1 followed on March 25, 1991, and 3.0 in September, 1992.

In the 1992 NeXTWORLD Expo, NEXTSTEP 486, a version (costing $995) for the PC was announced. Versions 3.1 and 3.2 were released in May and October, 1993, respectively. The last version of NEXTSTEP, 3.3, was released in February, 1995. A bit earlier, in 1994, NeXT and Sun had jointly released specifications for OPENSTEP, an open platform (comprised of several APIs and frameworks) that anybody could use to create their own implementation of *STEP. NeXT's implementation was named OpenStep, the successor to the NEXTSTEP operating system. Three versions of OpenStep were ever released: 4.0 (July 22, 1996), 4.1 (December, 1996), and 4.2 (January, 1997). SunOS, HP-UX, and even Windows NT had implementations at a point. The GNUstep Project still exists. Even though *STEP ran on many architectures (multi-architecture "fat binaries" were introduced by NeXT), by 1996, things were not looking good for them, and NeXT was giving more importance to *WebObjects*, a development tool for the Web.

Meanwhile, Apple had been desperately seeking to create an operating system that could compete with the onslaught from Microsoft. They actually wanted to beat Windows 95 to market, but failed. Apple suffered a setback when *Pink OS*, a joint venture between IBM and Apple, was killed in 1995. Apple eventually started an advanced operating system codenamed *Copland*, which was first announced to the public in 1994. The first beta of Copland went out in November, 1995, but a 1996 release (as planned and hoped) did not seem feasible. Soon afterwards, Apple announced that they would start shipping "pieces of Copland technology" beginning with System 7.6. Copland turned out to be a damp squib.

At this point Apple became interested in buying Be, a company that was becoming popular as the maker of the BeBox, running the BeOS. The deal between Apple's Gil Amelio and Be's Gassée never materialized - it has been often reported that Apple offered $125 million while Be

wanted an "outrageous" $200 million plus. The total investment in Be at that time was estimated to be only $20 million!

Apple then considered Windows NT, Solaris and even Pink OS. Then, Steve Jobs called Amelio, and advised him that Be was not a good fit for Apple's OS roadmap. NeXT contacted Apple to discuss possibilities of licensing OPENSTEP, which, unlike BeOS, had at least been proven in the market. Jobs pitched NeXT technology very strongly to Apple, and asserted that OPENSTEP was many years ahead of its time. All this worked out, and Apple acquired NeXT in February, 1997, for $427 million. Amelio later quipped that *"We choose Plan A instead of Plan Be."*

Apple named its upcoming NeXT-based system *Rhapsody*, while it continued to improve the existing Mac OS, often with technology that was supposed to go into Copland. Rhapsody saw two developer releases, in September, 1997, and May, 1998.

Jobs became the interim CEO of Apple on September 16, 1997.

Mac OS X was first mentioned in Apple's OS strategy announcement at the 1998 WWDC. Jobs said that OS X would ship in the fall of 1999, and would inherit from both Mac OS and Rhapsody. Moreover, backward compatibility would be maintained to ease customers into the transition.

Mac OS X did come out in 1999, as Mac OS X Server 1.0 (March 16, 1999), a developer preview of the desktop version, and as Darwin 0.1. Mac OS X beta was released on September 13, 2000.

At the time of this writing, Mac OS X has seen four major releases: 10.0 ("**Cheetah**", March 24, 2001), 10.1 ("**Puma**", September 29, 2001), 10.2 ("**Jaguar**", August 13, 2002), and 10.3 ("**Panther**", October 24, 2003).

It would be an understatement to say that OS X is derived from NEXTSTEP and OpenStep. In many respects, it's not just similar, it's the *same*. One can think of it as OpenStep 5 or 6, say. This is not a bad thing at all - rather than create an operating system from scratch, Apple tried to do the *smart* thing, and used what they already had to a great extent. However, the similarities should not mislead you: Mac OS X is evolved enough that what you can do with it is far above and beyond NEXTSTEP/OpenStep.

While unrelated to Mac OS X, Apple came out with their

version of UNIX, called A/UX, in 1988. A/UX was a POSIX compliant system based on AT&T UNIX System V (various releases) and BSD4.2/4.3, with a wide spectrum of features (STREAMS, TCP/IP, FFS, job control, NFS with YP, SCCS, printing, X Window System, compatibility with SYSV and BSD in addition to POSIX, and so on). More importantly, A/UX *combined* various features of the Macintosh with Unix - A/UX 3.x was a combination of the above mentioned Unix features with System 7 for the Macintosh, with the Finder and other Mac applications running under A/UX. The last version of A/UX, 3.1.1, was released in 1995. A/UX was regarded as the *holy grail* of Unices by some people.

Architecture of Mac OS X

This page briefly describes the architecture of Mac OS X and the Mac's firmware. Description of an operating system in reasonable detail would take up one or more books, which this page is not. Consider it as *a high-level description of some low-level details*, if you will.

As mentioned in *A Brief History of Mac OS X*, Mac OS X is a descendent of NEXTSTEP. We shall not point out any further similarities or differences between Mac OS X and these earlier systems.

Darwin

The first version of Darwin, 0.1, was released on March 16, 1999 as a fork of a developer release of Rhapsody. Although Darwin is an operating system in itself, it can be best understood as a collection of technologies that have been integrated by Apple to form a major, central part of Mac OS X. Critical application environments of Mac OS X, such as Cocoa and Carbon, are *not* part of Darwin. So isn't Aqua, the overall GUI of Mac OS X (including the Windowing System), and several other components.

Contents

Darwin 7.0.x (corresponding to Mac OS X 10.3.x) consists of over 250 packages. Many of these are Apple packages (including the Mac OS X kernel and various drivers), while the others originate from *BSD, GNU, etc. Apple has leveraged a lot of existing open source software by integrating it well (usually) with their system: apache, bind, binutils, cvs, gcc, gdb, gimp_print, kerberos, mysql, openssh, openssl, pam, perl, postfix, ppp, python, rsync, samba, and many more BSD/GNU/other packages ... are all part of Darwin. In many cases, Apple has made

important modifications to open source code to optimize/adapt it to their platform (consider gcc and gdb). Moreover, even though one can always configure and control such software "as usual" (editing their configuration file in `vi` or `emacs`, say), Apple provides simplified user interfaces that work well for at least the *not-so-contrived* cases.

An indicator of how well Apple has made use of existing open source software is the large number of sources Darwin draws from: *BSD, GNU, Mach, ... and even Linux. `/usr/sbin/pcsd` is a daemon used to dynamically allocate/deallocate Smart Card reader drivers at runtime and manage connection to the readers. This, and related tools, are taken from the MUSCLE (Movement for the Use of Smart Cards in a Linux Environment) Project. The much advertised "Secure Trash Deletion" is a simple adaptation of the open source `srm` utility (it has been extended to support named forks, for example). Such eclecticism in ingesting technology from various sources and wrapping/integrating it to create a uniform effect is one of the great strengths of Mac OS X.

Portability

Darwin runs on the PowerPC and x86 platforms. What's more, it is even possible to build a "fat" kernel, containing both platform executables in a single file.

Licensing

Darwin packages that are originally from Apple are usually covered under the Apple Public Source License (APSL). The current version (2.0) of APSL *qualifies* as a free software license, although it is *not compatible* with the GPL. The Free Software Foundation offers its views on APSL on their web site.

The GNU-Darwin Project aims to leverage the combination of Darwin and GNU. OpenDarwin is an Apple co-founded effort for fostering cooperative Darwin development. You can download bootable Darwin CD images for both the PowerPC and x86 platforms from these projects' web sites.

Layers

The pieces of software that constitute Mac OS X can be grouped into logical layers as shown below:

Firmware

The Mac firmware and the Mac OS X bootloader are described in the section *Booting Mac OS X*.

XNU: The Kernel

XNU, the Mac OS X kernel, is described in *XNU: The Kernel*. The kernel and its extensions make up the Kernel Environment, the *lowest* (most fundamental) layer of Mac OS X (the firmware and the bootloader are not technically parts of Mac OS X).

Higher Level Services and Environments

Still higher layers of Mac OS X are described in the section *Above the Kernel*.

Higher Level System Startup

Mac OS X System Startup describes the user-level sequence of events that happen when Mac OS X boots.

Booting Mac OS X

This page contains a brief description of the Mac's firmware (analogous to the PC BIOS in many respects), the bootloader, and the typical Mac OS X boot up sequence. There are significant differences between how older (68k, "Old World" PowerMacs) and newer (everything currently, but essentially "New World" machines with Open Firmware 3.x that *load* ROM from a file) boot. The discussion here applies to the newer systems.

The firmware is not part of Mac OS X, but it plays an important role in the operation of the machine, and is useful in debugging. Hence, we discuss it here.

Open Firmware

Background

Open Firmware (*IEEE-1275 Standard for Boot Firmware: Core Requirements and Practices*) is a non-proprietary, platform (CPU and system) independent boot firmware. Similar to a PC's BIOS, Open Firmware is stored in ROM and is the first stored program to be executed upon power-up.

An Open Firmware implementation is based on the Forth programming language, in particular, the FCode dialect (FCode is an ANS Forth compliant dialect that supports compilation of FCode source to bytecode). Apple and Sun are two prominent computer system makers that use implementations of Open Firmware in their systems (Sun's trademark is called *OpenBoot*). The Open Firmware Working Group's home page is hosted at various places, including Apple and Sun.

Thus, the firmware is implemented in Forth, and stored in the ROM as FCode bytecode. Device drivers that are required during system startup are also implemented similarly. Such drivers usually exist in the expansion ROM of expansion cards that are needed before the operating system has loaded.

Interaction

You can enter Open Firmware by pressing the key combination `cmd-opt-O-F` just as you power on a Macintosh. The `cmd` key is the one with the Apple logo, and the `opt` (option) key is the same as the `alt` key. You should see a welcome message and some other verbiage, and should be dropped into a prompt like the following:

```
 ok
0 >
```

You can continue booting the machine by typing `mac-boot`, or shut it down by typing `shut-down`.

Even though this Forth "shell" supports reasonable (for a BIOS) command line editing (you can use `ctrl-a` to go to the beginning of a line, `ctrl-e` to go to the end, `ctrl-u` to erase a line, the up-arrow key for history, etc.), you would find it more convenient (particularly if you are trying to *write* any code in the firmware) to access a Mac's Open Firmware from another (arbitrary) computer, over the network. Here is the command sequence to do this (everything is typed at the Open Firmware prompt, unless stated otherwise):

```
0 > dev /packages/telnet
```

Note that upon success, Open Firmware prints the string "`ok`" on the *same line* as you press <return>. In the examples on this page, if you see `ok`, remember that it is printed by Open Firmware and you are not supposed to type it in (it's not a valid Open Firmware word anyway).

If your Mac's Open Firmware includes the `telnet` package, you would see:

```
0 > dev /packages/telnet  ok
```

If you do get an `ok`, you can run a TELNET server on it:

```
" enet:telnet,10.0.0.1" io
```

This would run a TELNET server on the machine with IP address `10.0.0.1` (you can and should choose any appropriate address). Thereafter, you can connect to Open Firmware on this machine using a TELNET client - say, from a Windows machine. See *The Towers of Hanoi in Open Firmware* for a programming example.

Note that current (at least G4 and above) Apple computers come with Ethernet ports that are auto-sensing and self-configuring, so you do *not* need a cross-over cable to connect it directly to another computer (no hub is required, etc.).

Examples

1. The following command prints the device tree:

```
0 > dev / ls
ff880d90: /cpus
ff881068:   /PowerPC,750@0
ff881488:    /l2-cache
ff882148: /chosen
ff882388: /memory@0
ff882650: /openprom
ff882828:  /client-services
...
More [<space>,<cr>,q,a] ? _
```

2. The following command gives you information about installed RAM:

```
0 > dev /memory .properties ok
name                          memory
device_type                   memory
reg                           00000000  10000000
                              10000000  10000000
```

```
slot-names                      00000003
                                SODIMM0/J25LOWER
                                SODIMM1/J25UPPER
...
dimm-types                      DDR SDRAM
                                DDR SDRAM
dimm-speeds                     PC2700U-25330
                                PC2700U-25330
...
```

The machine in the above command (a PowerBook G4 15, although that is not relevant) has two PC2700 DDR SDRAM chips installed. The two pairs of numbers against `reg` are specify the starting address and size of the chips. Thus, the first RAM chip starts at address `0x0000000` and has a size `0x10000000` (which is 256 MB). The second chip starts at `0x1000000` (256 MB) and has a size 256 MB. The total RAM is thus 512 MB.

If you need to *reduce* the installed RAM size (as seen by Mac OS X) for any reason, without actually having to remove a RAM stick (or you want to simulate an arbitrary size that's less than the total installed RAM), you can actually *delete* the `reg` entry using the `delete-property` command, and specify your own `reg`. Referring to the previous example of the 512 MB PowerBook, the following command essentially *disables* the second RAM stick (note that this change is *not* written to NVRAM - it is transient - once you reboot, the other chip will be detected and used as before):

```
0 > " reg" delete-property  ok
0 > 0 encode-int 10000000 encode-int encode+
" reg" property  ok
```

It must be kept in mind though that the reg properties can change from machine to machine, or more likely with architectural changes (for example, the format changed with the PowerMac G5). A less adventurous and more appropriate way to limit visible RAM is to use the `maxmem` boot argument, such as at the shell prompt:

```
# nvram boot-args="maxmem=128"
```

3. The following command sequence shows you various information on the machine's CPU(s):

```
0 > dev /  ok
0 > dev /cpus  ok
0 > ls
ff886d58:  /PowerPC,G4@0
ff8871f8:    /l2-cache
 ok
0 > dev PowerPC,G4@0  ok
0 > .properties
name                        cpu
reg                         00000000
cpu-version                 80020101
state                       running
clock-frequency             4a817c7b
bus-frequency               09ef21aa
...
```

The rest of the output contains various cache sizes, the processor's graphics capabilities (Altivec, support for certain instructions, ...), and so on. You can think of this as analogous to `/proc/cpuinfo` on Linux.

4. The following command lists files in the root directory of the disk (partition) referred to by the "alias" `hd`

```
0 > dir hd:\
```

```
        Size/
GMT                  File/Dir
        bytes       date
time          Name
         6148     12/25/ 3
4:25:25       .DS_Store
          156      9/12/ 3
20:41:59       .hidden
       589824     12/25/ 3      6:45:
6       .hotfiles.btree
...
```

5. The following command expands the alias `hd`, and gives you the complete path of the device in the tree (type `devalias` by itself to see a list of current aliases, along with what they refer to):

```
0 > devalias hd /pci@f4000000/ata-6@d/disk@0
ok
```

6. You can load a file (kernel) using the `load` command, and boot it using the `boot` command. As stated earlier, `mac-boot` and `shut-down` are predefined to boot the machine normally, or shut it down, respectively. You can get and set variables (options) using the `printenv` and `setenv` commands. These variables are stored in the non-volatile memory (NVRAM) of Open Firmware. For example, if you want your email address to be used as the "OEM banner", you should do the following:

```
0 > setenv oem-banner you@your.email.address
0 > setenv oem-banner? true
```

You do not actually need to drop into Open Firmware to set the NVRAM variables. You can access (get and set) these from within Mac OS X via the `nvram` command line utility.

To sum up, Open Firmware is a powerful tool for controlling, debugging, and exploring the computer.

Operation

When an Open Firmware equipped Macintosh (all current Apple systems at the time of this writing) is powered on, hardware is diagnosed (by some POST code) and initialized. The first entity to control the CPU thereafter is the firmware. Open Firmware (which runs with interrupts disabled) builds a device tree, probes slots for devices, queries PCI devices and assigns them address space appropriately, and then looks for the default boot device (unless one was specified explicitly). The following "snag" keys let the user specify a boot device as the system is powered on:

```
C     device referred to by the 'cd' alias,
a CD-ROM drive
D     device referred to by the 'hd' alias,
a hard disk drive
N     device referred to by the 'enet'
alias, a network card
Z     device referred to by the 'zip' alias,
a ZIP drive
```

It is worth noting that pressing `T` while your Mac powers on would boot it into what's called the *FireWire Target Disk Mode*. Essentially, your Mac becomes a fancy external FireWire disk drive.

You can also specify the complete pathname of a device, or have the machine boot over the network using TFTP:

```
boot enet:<server IP>,<file>,<my IP>;
```

```
<subnet>,<gateway IP>
```

If Open Firmware fails to find a boot device, a *blinking folder* is displayed.

Open Firmware then loads a file of type `tbxi` (ToolBox ROM Image, for historical reasons) from the system partition. Note that this would have been the file called "`Mac OS ROM`" in the System Folder on Mac OS 9, while OS X loads `/System/Library/CoreServices/BootX`, which is the bootloader as well. BootX is then executed and Control is then passed to it.

Note that Open Firmware can directly load ELF, XCOFF and "bootinfo" (any supported format with an XML header) binaries, but not Mach-O, the native executable format on Mac OS X. BootX can load Mach-O binaries.

Bootloader

BootX (`/System/Library/CoreServices/BootX`) is the default bootloader on Mac OS X.

BootX is also the name of an open source bootloader (different from Apple's BootX) that allows dual-booting Mac OS and Linux on "Old World" machines.

BootX can load kernels from various filesystems: HFS+, HFS, UFS, ext2, and TFTP (network, abstracted to look like a filesystem). In addition to Mach-O, BootX can also load ELF kernels, although Mac OS X does not use this feature. To reiterate, BootX can load ELF kernels from an ext2 partition!

The "Old World" Macs had various issues with the implementation of Open Firmware, which in turn caused many booting problems for Apple engineers, and even more problems for the PowerPC Linux port. Now, Apple had access to the firmware's source. They solved most of the problems either via NVRAM patches, or by integrating required changes into BootX itself (in the instances where the changes could not be implemented as patches). As BootX matured, Apple added support for ext2 and ELF with the goal of making the platform more amenable to PowerPC Linux.

The sequence of events when BootX starts executing (after being handed control by Open Firmware) is described below:

- BootX first initializes the Open Firmware client interface (that it would use to talk to the firmware), and retrieves the firmware version.
- It then creates a pseudo-device called `sl_words` ('sl' implies *secondary loader*) in the firmware, and defines various FORTH words in it (it is here that code for the spinning cursor is set up).
- BootX looks up the `options` device in the firmware, which contains various variables (that you can see and set using the `printenv` and `setenv` commands in Open Firmware).

```
0 > dev /options .properties
name                    options
little-endian?          false
real-mode?              false
auto-boot?              true
diag-switch?            false
...
boot-command            mac-boot
...
```

- BootX looks up the `chosen` device, which contains handles for entities such as the boot input/output devices, memory, the MMU, the PMU, the CPU, the PIC, etc. For example, the following command at the Open Firmware prompt shows you the contents of `chosen`:

```
0 > dev /chosen  ok
0 > .properties
name                    chosen
stdin                   ffbc6e40
stdout                  ffbc6600
```

```
memory                  ffbdd600
mmu                     ...
...
```

- BootX initializes handles to the MMU and memory using `chosen`.
- BootX initializes handles to the boot display and the keyboard (if present).
- BootX checks if the "security mode" is "none", or
- BootX checks if the "verbose" (`cmd-v`) or "single user" (`cmd-s`) flags were specified, and sets the "output level" accordingly.
- BootX checks if the system is booting in "Safe Mode".
- BootX claims memory for various purposes.
- BootX finds all displays and sets them up. It does this by searching for nodes of type "display" in the device tree. The primary display is referred to by the `screen` alias. For example, you can try this at the Open Firmware prompt:

```
0 > dev screen  ok
0 > .properties
name            ATY,Bee_A
compatible      ATY,Bee
width           00000400
height          00000300
linebytes       00000400
depth           00000008
display-type    4c434400
device_type     display
character-set   ISO859-1
...
```

- While opening the display(s), BootX also sets the screen color to

the familiar whitish gray.

- BootX looks up the boot device, boot arguments, etc., and determines where to get the kernel from (via a network device, from a block device, etc.), whence the path to the kernel file (`mach_kernel`) is constructed. If booting from a block device (which is the usual case), the path to the kext cache (see `kextcache(8)`) is calculated, along with the extensions directory (usually `/System/Library/Extensions`).

Mac OS X uses a few kinds of "kext" (kernel extension) caches to speed up loading of kexts. Kernel caches are kept in the directory `/System/Library/Caches/com.apple.kernelcaches`. The cache files are named `kernelcache.XXXXXXXX`, where the prefix is a 32-bit `adler` checksum (the same algorithm as used by Gzip).

- At this point, BootX draws the Apple logo splash screen, and starts the spinning cursor. If booting from a network device, a spinning globe is drawn instead.
- Depending on various conditions, BootX tries to retrieve and load the kernel cache file.
- The next step is to "decode" the kernel. If the kernel header indicates a compressed kernel, BootX tries to decompress it (typical LZSS compression is used, as you compress this kind of data once but expand it many times). Since the kernel binary can potentially be a "fat" binary (code for multiple architectures residing in the same binary), BootX checks if it indeed is (fat), and if so, "thins" it (figures out the PowerPC code).
- BootX attempts to decode the file (possibly "thinned") as a Mach-O binary. If this fails, BootX also tries to decode it as ELF.
- If the above fails, BootX gives up, draws the failed boot picture, and goes into an infinite loop.
- If BootX is successful so far, it saves filesystem cache hits, misses and evicts, sets up various boot arguments and values (such as whether this is a graphical or verbose boot, whether there are some flags to be passed to the kernel, the size of installed RAM), and also calls a recursive function to flatten the device tree.

- Finally, BootX "calls" the kernel, immediately before which it "quiesces" Open Firmware, an operation as a result of which any asynchronous tasks in the firmware, timers, or DMA get stopped, etc.

System Startup

Mac OS X user level startup is neither pure BSD style, nor SYSV style, although the presence of `/etc/rc` indicates a BSD heritage. In fact, various things are unsurprisingly similar to NEXTSTEP.

The next section, *XNU: The Kernel*, describes some of the things the kernel does as it comes up. *Mac OS X System Startup* continues with a description of (mostly) user-level startup.

BootCache

Mac OS X uses a boot-time optimization (effectively a smart readahead) called "BootCache" that monitors the pattern of incoming read requests to a block device (the boot disk), and sorts the pattern into a "playlist" (it also measures the cache hit rate and stores the request pattern into a "history list" for being adaptive in future).

The loadable (sorted) read pattern is stored in `/var/db/BootCache.playlist`. Once this is loaded, the cache comes into effect.

Note that this feature requires *at least* 128 MB of physical RAM before it is enabled (automatically).

`/System/Library/Extensions/BootCache.kext` is the location of the kernel extension implementing the cache while `Contents/Resources/BootCacheControl` within that directory is the user-level control utility (it lets you load the playlist, among other things). The effectiveness of BootCache can be gauged from the following: in a recent update to "Panther", a reference to BootCacheControl was broken. BootCache is started (via the control utility) in `/etc/rc`, and a prefetch tag is inserted (unless the system is booting in safe mode). `/etc/rc` looks for `BootCacheControl` in the "kext" directory, as well as in `/usr/sbin`, and finds it in the former (it doesn't exist in the latter). However, another program (possibly `loginwindow.app`) accesses `/usr/sbin/BootCacheControl` directly, and does not find it. For what it's

worth, making `BootCacheControl` available in `/usr/sbin`, say via a symlink, reduces the boot time (measured from clicking on the "Restart" confirmation button to the point where *absolutely* everything has shown up on the system menu) from 135 seconds to 60 seconds on one of my machines!

XNU: The Kernel

The Mac OS X kernel is called `XNU`. It can be viewed as consisting of the following components:

Mach

XNU contains code based on Mach, the legendary architecture that originated as a research project at Carnegie Mellon University in the mid 1980s (Mach itself traces its philosophy to the Accent operating system, also developed at CMU), and has been part of many important systems. Early versions of Mach had monolithic kernels, with much of BSD's code in the kernel. Mach 3.0 was the first microkernel implementation.

XNU's Mach component is based on Mach 3.0, although it's not used as a microkernel. The BSD subsystem *is* part of the kernel and so are various other subsystems that are typically implemented as user-space servers in microkernel systems. XNU's Mach is responsible for various low-level aspects of the system, such as:

- preemptive multitasking, including kernel threads (POSIX threads on Mac OS X are implemented using kernel threads)
- protected memory
- virtual memory management
- inter-process communication
- interrupt management
- real-time support
- kernel debugging support (the built-in low-level kernel debugger, `ddb`, is part of XNU's Mach component, and so is `kdp`, a remote kernel debugging protocol implementation)
- console I/O

The sequence of events prior to the kernel is passed control is described

in *Booting Mac OS X*. The secondary bootloader eventually calls the kernel's "startup" code, forwarding various boot arguments to it. This low-level code is where every processor in the system starts (from the kernel's point of view). Various important variables, like maximum virtual and physical addresses, the threshold temperature for throttling down a CPU's speed, are initialized here, BAT registers are cleared, Altivec (if present) is initialized, caches are initialized, etc. Eventually this code jumps to boot initialization code for the architecture (`ppc_init()` on the PowerPC). Thereafter:

- A template thread is filled in, and an initial thread is created from this template. It is set to be the "current" thread.
- Some CPU housekeeping is done.
- The "Platform Expert" (see below) is initialized (`PE_init_platform()`), with a flag indicating that the VM is not yet initialized. This saves the boot arguments, the device tree and display information in a state variable. Another call to `PE_init_platform()` is made after the VM is initialized.
- Mach VM is initialized.
- The function `machine_startup()` is called. It takes some actions based on the boot arguments, performs some housekeeping, starts thermal monitoring for the CPU, and calls `setup_main()`.
- `setup_main()` performs a lot of work: initializing the scheduler, IPC, kernel extension loading, clock, timers, tasks, threads, etc. and finally creates a kernel thread called `startup_thread` that creates further kernel threads.
- `startup_thread` creates a number of other threads (the idle threads, service threads for clock and device, ...). It also initializes the thread reaper, the stack swapin and the periodic scheduler mechanism. It is here that the BSD subsystem is initialized (via `bsd_init()`). `startup_thread` becomes the pageout daemon once it finishes its work.

At this point, Mach is up and running.

In additions to BSD system calls (the `syscall` API, as well as the `sysctl` and `ioctl` APIs), Mach messaging and IPC can be and is used (as appropriate) to exchange information between the user and

kernel spaces. XNU also provides various ways of memory mapping and block copying. While it may be nice (say, from an academic point of view, if nothing else) to have many APIs in a system, there is always a burden on the programmer for *choosing* wisely what API to use. The situation is similar for user-space APIs on Mac OS X, as we shall see later.

BSD

XNU's BSD component uses FreeBSD as the primary reference codebase (although some code might be traced to other BSDs). Darwin 7.x (Mac OS X 10.3.x) uses FreeBSD 5.x. As mentioned before, BSD runs not as an *external* (or user-level) server, but is part of the kernel itself. Some aspects that BSD is responsible for include:

- process model
- user ids, permissions, basic security policies
- POSIX API, BSD style system calls
- TCP/IP stack, BSD sockets, firewall
- VFS and filesystems (see *Mac OS X Filesystems* for details)
- System V IPC
- crypto framework
- various synchronization mechanisms

Note that XNU has a unified buffer cache but it ties in to Mach's VM.

XNU uses a synchronization abstraction (built on top of Mach mutexes) called *funnels* to serialize access to the BSD portion of the kernel. The kernel variables pointing to these funnels have the `_flock` suffix, such as `kernel_flock` and `network_flock`. When Mach initializes the BSD subsystem via a call to `bsd_init()`, the first operation performed is the allocation of funnels (the kernel funnel's state is set to `TRUE`). Thereafter:

- The kernel memory allocator is initialized.
- The "Platform Expert" (see below) is called upon to see if there are any boot arguments for BSD.
- VFS buffers/hash tables are allocated and initialized.
- Process related structures are allocated/initialized. This includes

the list of all processes, the list of zombie processes, hash tables for process ids and process groups.

- Process 0 is created and initialized (credentials, file descriptor table, audit information, limits, etc.). The variable `kernproc` points to process 0.
- The machine dependent real-time clock's time and date are initialized.
- The Unified Buffer Cache is initialized (via `ubc_init()`, which essentially initializes a Mach VM Zone via `zinit()`, which allocates a region of memory from the page-level allocator).
- Various VFS structures/mechanisms are initialized: the vnode table, the filesystem event mechanism, the vnode name cache, etc. Each present filesystem time is also initialized.
- `mbufs` (memory buffers, used heavily in network memory-management) are initialized via `mbinit()`.
- Facilities/subsystems such as `syslog`, `audit`, `kqueues`, `aio`, and System V IPC are initialized.
- The kernel's generic MIB (management information base) is initialized.
- The data link interface layer is initialized.
- Sockets and protocol families are initialized.

XNU uses a specific type of kernel extensions, NKEs (Network Kernel Extensions), to make the 4.4BSD networking architecture fit in to Mac OS X.

- Kernel profiling is started, and BSD is "published" as a resource in the IOKit.
- Ethernet devices are initialized.
- A Mach Zone is initialized for the vnode pager.
- BSD tries to mount the root filesystem (which could be coming over the network, for example, a Mac OS X disk image (`.dmg`) exported over NFS).
- `devfs` is mounted on `/dev`.

- A new process is created (cloned) from `kernproc` (process 0). This newly created process has `pid` 1, and is set to become `init` (actually `mach_init`, which starts `init`). `mach_init` is loaded and run via `bsdinit_task()`, which is called by the BSD asynchronous trap handler (`bsd_ast()`).

The rest of the user space startup is described in *Mac OS X System Startup*.

I/O Kit

I/O Kit, the object-oriented device driver framework of the XNU kernel is radically different from that on traditional systems.

I/O Kit uses a restricted subset of C++ (based on Embedded C++) as its programming language. This system is implemented by the `libkern` library. Features of C++ that are not allowed in this subset include:

- exceptions
- multiple inheritance
- templates
- RTTI (run-time type information), although I/O Kit has its own run-time typing system

The device driver model provided by the I/O Kit has several useful features (in no particular order):

- numerous device families (ATA/ATAPI, FireWire, Graphics, HID, Network, PCI, USB, HID, ...)
- object oriented abstractions of devices that can be shared
- plug-and-play and hot-plugging
- power management
- preemptive multitasking, threading, symmetric multiprocessing, memory protection and data management
- dynamic matching and loading of drivers (multiple bus types)
- a database for tracking and maintaining detailed information on instantiated objects (the I/O Registry)

- a database of all I/O Kit classes available on a system (the I/O Catalog)
- an extensive API
- mechanisms/interfaces for applications and user-space drivers to communicate with the I/O Kit
- driver stacking

I/O Kit's implementation consists of three C++ libraries that are present in the kernel and available to loadable drivers: `IOKit.framework`, `Kernel/libkern` and `Kernel/IOKit`. The I/O Kit includes a modular, layered run-time architecture that presents an abstraction of the underlying hardware by capturing the dynamic relationships between the various hardware/software components (involved in an I/O connection).

Various tools such as `ioreg`, `ioalloccount`, `ioclasscount`, `iostat`, `kextload`, `kextunload`, `kextstat`, `kextcache`, etc. let you explore and control various aspects of I/O Kit. For example, the following command shows status of dynamically loaded kernel extensions:

```
% kextstat
Index Refs Address Size  Wired Name
(Version) <Linked Against>
    1    1 0x0     0x0   0x0   com.apple.
kernel (7.2)
    2    1 0x0     0x0   0x0   com.apple.kpi.
bsd (7.2)
    3    1 0x0     0x0   0x0   com.apple.kpi.
iokit (7.2)
    4    1 0x0     0x0   0x0   com.apple.kpi.
libkern (7.2)
...
```

The following command lists the details of the I/O Kit registry in excruciating detail:

```
% ioreg -l -w 0
+-o Root  <class IORegistryEntry, retain
count 12>
  | {
  |   "IOKitBuildVersion" = "IOKit Component
Version 7.2:
Thu Dec 11 16:15:20 PST 2003;
root(rcbuilder):RELEASE_PPC/iokit/RELEASE
"
  |   "IONDRVFramebufferGeneration" =
<000000200000002>
...
/* thousands of lines of output */
```

Platform Expert

The Platform Expert is an object (one can think of it as a driver) that knows the type of platform that the system is running on. I/O Kit registers a *nub* (see below) for the Platform Expert. This nub then loads the correct platform specific driver, which further discovers the buses present on the system, registering a nub for each bus found. The I/O Kit loads a matching driver for each bus nub, which discovers the devices connected to the bus, and so on. Thus, the Platform Expert is responsible for actions such as:

- Building the device tree (as described above)
- Parse certain boot arguments
- Identify the machine (including process and bus clock speeds)
- Initialize a "user interface" to be used in case of kernel panics

In the context of the I/O Kit, a "nub" is an object that defines an access point and communication channel for a device (a bus, a disk drive or partition, a graphics card, ...) or logical service (arbitration, driver matching, power management, ...).

libkern and libsa

As described earlier, the I/O Kit uses a restricted subset of C++. This system, implemented by `libkern`, provides features such as:

- Dynamic object allocation, construction, destruction (including data structures such as Arrays, Booleans, Dictionaries, ...)
- Certain atomic operations, miscellaneous functions (`bcmp()`, `memcmp()`, `strlen()`, ...)
- Provisions for tracking the number of current instances for each class
- Ways to avoid the "Fragile Base Class Problem"

`libsa` provides functions for miscellaneous purposes: binary searching, symbol remangling (used for gcc 2.95 to 3.3, for example), dgraphs, catalogs, kernel extension management, sorting, patching vtables, etc.

Above the Kernel

This page discusses the software layers above the kernel in Mac OS X.

Core Services

The Core Services layer can be visualized as sitting atop the kernel. This layer's most important sub-components are `CoreFoundation.`

`framework` and `CoreServices.framework`. It contains various critical *non-GUI* system services (including APIs for managing threads and processes, resource, virtual memory and filesystem interaction):

- **CarbonCore**: Core parts of Carbon, such as various Carbon managers. Carbon has traditionally been a very critical Mac OS API family, and is so in Mac OS X as well.
- **CFNetwork**: An API for user-level networking that includes several protocols such as FTP, HTTP, LDAP, SMTP, ...
- **OSServices**: A framework that includes various system APIs (accessing disk partitions, the system keychain, Open Transport, sound, power, etc.)
- **SearchKit**: A framework for indexing and searching text in multiple languages.
- **WebServicesCore**: APIs for using Web Services via SOAP and XML-RPC.

CoreFoundation also includes a large number of other services. For example, it provides ways so that applications can access URLs, parse XML, maintain property lists, etc. In the directory `/System/Library/Frameworks/`, refer to the directory `CoreFoundation.framework/Headers/` for headers belonging to `CoreFoundation`.

Application Services

This layer can be visualized as being on top of Core Services. It includes services that make up the the graphics and windowing environment of Mac OS X.

The core of the windowing environment is called **Quartz**. Quartz consists of broadly two entities:

- **Quartz Compositor**: consists of the window server (the `WindowServer` program) and some private libraries. Quartz implements a layered compositing engine, in which every pixel on a screen can be shared between different windows in real time.
- **Quartz 2D**: a 2D graphics rendering library.

While NeXT used "Display PostScript" (DPS) for the imaging model in NEXTSTEP, Quartz uses PDF for its drawing model (or as its native format). This makes possible some useful features, such as automatic generation of PDF files (you can save a screenshot "directly" to PDF), import of PDF data into native applications, rasterization of PDF data (including PostScript and EPS conversion), etc. There are Python bindings to the Quartz PDF engine. Note however, that Quartz's PDF support is not a replacement for, say, Adobe's professional level PDF tools.

Quartz also has an integrated hardware acceleration layer called **Quartz Extreme** that automatically becomes active on supported hardware.

The graphics environment also has other rendering libraries, for example, OpenGL (2D and 3D), QuickDraw (2D) and QuickTime.

The Application Services layer also includes various other component frameworks:

- **Apple Events**: a mechanism for inter-application communication.
- **Apple Type Services**: a framework for managing and accessing fonts on Mac OS X.
- **ColorSync**: Apple's color management system that's integrated with Mac OS X.
- **CoreGraphics**: a graphics framework that's used for implementing Quartz.
- **FindByContent**: an API that allows searching specific volumes or folders for files containing the specified content.
- **HIServices**: framework providing human-interface related services, such as for icon management, translation, "pasteboard" (clipboard) etc.
- **LangAnalysis**: an API to the Language Analysis Manager, allows for analyzing *morphemes* (a morpheme is a meaningful linguistic unit, that is, a distinctive collocation of phonemes, having *no smaller meaningful parts*) in text.
- **LaunchServices**: a medium-level interface to starting applications, including opening documents with either a specified or the default application, open URLs, etc.

- **PrintCore**: an API to the printing subsystem.
- **QD**: a 2D graphics engine.
- **SpeechSynthesis**: an API for generating synthesized speech.

QuickTime is both a graphics environment and an application environment. It has excellent features for interactive multimedia that allow for manipulating, streaming, storing and enhancing video, sound, animation, graphics, text, music, and VR.

Application Environments

There are multiple execution environments on Mac OS X within which respective applications execute:

- **BSD**: This application environment is similar to a traditional *BSD system and provides a BSD-based POSIX API. It consists of a BSD runtime and execution environment. Mac OS X uses FreeBSD as a reference code base for its BSD derivations (Panther derives from FreeBSD 5.0). The libraries and headers for this environment reside in their traditional location (`/usr/lib` and `/usr/include`, respectively).
- **Carbon**: This is a set of procedural C-based APIs for Mac OS X that are based on the "old" Mac OS 9 APIs. Note that Carbon does not include *all* the old APIs - a subset of the old APIs has been modified to work with OS X. Some APIs have been dropped as they are not applicable any more because of the radical differences between Mac OS X and Mac OS 9.

The fact that Mac OS X includes APIs and abstractions from so many different systems (Mach, *BSD, Mac OS 9, etc.) makes things rather confusing and messy sometimes. Consider that Mach uses `tasks` (that contain one or more threads), FreeBSD uses processes (with a `proc` structure, `pid`, etc.) while Carbon uses its own notion of processes in the Carbon Process Manager, with process serial numbers (PSNs) which are *not the same* as a BSD pid! If a process is running under the Classic emulator, then multiple Carbon Process Manager processes inside Classic are using *one* BSD process. Consider the following excerpt from the output of the `ps` command:

```
% ps -axw | grep Safari
  345 ?? S 4:18.19 /Applications/
Safari.app/Contents/\
MacOS/Safari -psn_0_917505
```

Safari is linked against both the Carbon and Cocoa frameworks, among others. The above output means that Unix process id `345` maps to Carbon Process Manager PSN `917505`.

You can use the Carbon function `GetProcessForPID (pid_t, ProcessSerialNumber *)` to obtain the PSN for a process given its Unix pid (note that not all processes will have both).

- **Classic**: This is a compatibility environment so that Mac OS 9 applications can be run on Mac OS X. The Classic emulator runs in a protected memory environment, with multiple processes in Mac OS 9 layered on top of one BSD process.
- **Cocoa**: This is an object-oriented API for developing applications written in Objective-C and Java. Cocoa is an important inheritance from NEXTSTEP (a fact testified by the various `NS*` names in its API). It is very well supported by Apple's rapid development tools, and is the preferred way of doing things on Mac OS X if what you want to do can be done through Cocoa. There are many parts of Mac OS X that have not "converted" to Cocoa completely, or at all. A Cocoa application can call the Carbon API. Cocoa is largely based on the OpenStep frameworks, and consists of primarily two parts: the Foundation (fundamental classes) and the Application Kit (classes for GUI elements).
- **Java**: This environment consists of a JDK, both command-line and integrated with Apple's IDE, a runtime (Hotspot VM, JIT), and various Java classes (AWT, Swing, ...).

Cocoa includes Java packages that let you create a Cocoa

application using Java as the programming language. Moreover, Java programs can call Carbon and other frameworks via JNI.

Finally, although Java is considered an Application Environment, the Java subsystem can itself be represented as different layers above the kernel. The Java Virtual Machine along with core JDK packages is analogous to the Core Services layer, and so on (see picture).

Filesystem Hierarchy

Although Mac OS X has many directories similar to a traditional *nix system, such as `/etc` (a symbolic link to `/private/etc`, `/usr`, `/tmp` (a symbolic link to `/private/tmp`, etc., it has many others that are unique to it, for example:

`/`	root, the **Local** domain
`/Applications`	GUI applications
`/Applications/Utilities`	GUI utilities
`/Developer`	Xcode developer tools
`/Library`	User installed frameworks
`/Network`	the **Network** domain
`/System`	the **System** domain
`/System/Library`	System frameworks
`/Users`	Home directories, the **User** domain
`~/Desktop`	Your desktop
`~/Documents`	Your documents
`~/Movies`	Your movies directory (iMovie uses it by default)
`~/Music`	Your music directory (iTunes uses it by default)
`~/Pictures`	Your pictures directory (iPhoto uses it by default)
`~/Public`	For sharing (contents visible to others)
`~/Sites`	Your www (apache) directory

Mac OS X System Startup

This page briefly describes the sequence of events that happen when Mac OS X boots.

Some details of the boot process from power-on until the kernel is up and running are covered in *Booting Mac OS X* and *XNU: The Kernel*. To recapitulate:

- Power is turned on.
- Open Firmware code is executed.
- Hardware information is collected and hardware is initialized.
- Something (usually the OS, but also things like the Apple Hardware Test, etc.) is selected to boot. The user may be prompted to select what to boot.
- Control passes to `/System/Library/CoreServices/BootX`, the boot loader. BootX loads the kernel and also draws the OS badges, if any.
- BootX tries to load a previously cached list of device drivers (created/updated by `/usr/sbin/kextcache`). Such a cache is of the type `mkext` and contains the info dictionaries and binary files for multiple kernel extensions. Note that if the mkext cache is corrupt or missing, BootX would look in `/System/Library/Extensions` for extensions that are needed in the current scenario (as determined by the value of the `OSBundleRequired` property in the `Info.plist` file of the extension's bundle.
- The `init` routine of the kernel is executed. The root device of the booting system is determined. At this point, Open Firmware is not accessible any more.
- Various Mach/BSD data structures are initialized by the kernel.
- The I/O Kit is initialized.
- The kernel starts `/sbin/mach_init`, the Mach service naming (bootstrap) daemon. `mach_init` maintains mappings between service names and the Mach ports that provide access to those services.

From here on, the startup becomes user-level:

- `mach_init` starts `/sbin/init`, the traditional BSD init process. init determines the runlevel, and runs `/etc/rc.boot`, which sets up the machine enough to run single-user.

During its execution, `rc.boot` and the other `rc` scripts source `/etc/rc.common`, a shell script containing utility functions, such as `CheckForNetwork()` (checks if the network is up), `GetPID()`, `purgedir()` (deletes directory contents only, not the structure), etc.

- `rc.boot` figures out the type of boot (Multi-User, Safe, CD-ROM, Network etc.). In case of a network boot (the `sysctl` variable `kern.netboot` will be set to `1` in which case), it runs `/etc/rc.netboot` with a `start` argument.

`/etc/rc.netboot` handles various aspects of network booting. For example, it performs network and (if any) local mounts. It also calls `/usr/bin/nbst` to associate a shadow file with the disk image being used as the root device. The idea is to redirect writes to the shadow file, which hopefully is on local storage.

- `rc.boot` figures out if a file system consistency check is required. Single-user and CD-ROM boots do not run fsck. SafeBoot always runs fsck. `rc.boot` handles the return status of fsck as well.
- If `rc.boot` exits successfully, `/etc/rc`, the multi-user startup script is then run. If booting from a CD-ROM, the script switches over to `/etc/rc.cdrom` (installation).
- `/etc/rc` mounts local file systems (HFS+, HFS, UFS, `/dev/fd`, `/.vol`), ensures that the directory `/private/var/tmp` exists, and runs `/etc/rc.installer_cleanup`, if one exists (left by an installer before reboot).

- `/etc/rc.cleanup` is run. It "cleans" a number of Unix and Mac specific directories/files.
- BootCache is started.
- Various `sysctl` variables are set (such as for maximum number of vnodes, System V IPC, etc.). If `/etc/sysctl.conf` exists (plus `/etc/sysctl-macosxserver.conf` on Mac OS X Server), it is read and `sysctl` variables contained therein are set.
- `syslogd` is started.
- The Mach symbol file is created.
- `/etc/rc` starts `kextd`, the daemon process that loads kernel extension on demand from kernel or client processes.
- `/usr/libexec/register_mach_bootstrap_servers` is run to load various Mach bootstrap based services contained in `/etc/mach_init.d`
- `portmap` and `netinfo` are started.
- If `/System/Library/Extensions.mkext` is older than `/System/Library/Extensions`, `/etc/rc` deletes the existing mkext and creates a new one. It also creates one if one doesn't exist.
- `/etc/rc` starts `/usr/sbin/update`, the daemon that flushes internal file system caches to disk frequently.
- `/etc/rc` starts the virtual memory system. `/private/var/vm` is set up as the swap directory. `/sbin/dynamic_pager` is started with the appropriate arguments (swap filename path template, size of swap files created, high and low water alert triggers specifying when to create additional swap files or delete existing ones).
- `/etc/rc` starts `/usr/libexec/fix_prebinding` to fix incorrectly prebound binaries.
- `/etc/rc` executes `/etc/rc.cleanup` to clean up and reset files and devices.
- `/etc/rc` finally launches `/sbin/SystemStarter` to

handle startup items from locations such as `/System/Library/StartupItems` and `/Library/StartupItems`. A StartupItem is a program, usually a shell script, whose name matches the folder name. The folder contains a property list file containing key-value pairs such as `Description`, `Provides`, `Requires`, `OrderPreference`, start/stop messages etc. You can run `SystemStarter -n -D` as root to have the program print debugging and dependency information (without actually running anything).

- The `CoreGraphics` startup item starts the Apple Type Services daemon (`ATSServer`) as well as the Window Server (`WindowServer`).

By default, the `loginwindow` application (`loginwindow.app` under `/System/Library/CoreServices`) is executed for the console device. You can change this line in `/etc/ttys` if you do not want a graphical login.

Mac OS X Filesystems

Like most modern day operating system implementations, Mac OS X uses an object-oriented *vnode* layer. xnu's VFS layer is based on FreeBSD's, although there are numerous minor differences (for example, while FreeBSD uses mutexes, xnu uses simple locks; XNU's unified buffer cache is integrated with Mach's virtual memory layer, and so on).

Local Filesystems

HFS

HFS (Hierarchical File System) was the primary filesystem format used on the Macintosh Plus and later models, until Mac OS 8.1, when HFS was replaced by HFS Plus.

This section briefly describes the various filesystems supported by "stock" Mac OS X.

HFS+

HFS+ is the preferred filesystem on Mac OS X. It supports journaling, quotas, byte-range locking, Finder information in metadata, multiple encodings, hard and symbolic links, aliases, support for hiding file extensions on a per-file basis, etc. HFS+ uses B-Trees heavily for many of its internals.

> Like most current journaling filesystems, HFS+ only journals meta-data. Journaling support was retrofitted into HFS+ via a simple VFS journaling layer in XNU that's actually filesystem independent. The journal files on an HFS+ volume are called `.journal` and `.journal_info_block` (type `jrnl` and creator code `hfs+`). HFS+, although not a cutting-edge filesystem, supports some unique features and has worked well for Apple.

Similar to HFS

HFS+ is architecturally similar to HFS, with several important improvements such as:

- 32 bits used for allocation blocks (instead of 16). HFS divides the disk space on a partition into equal-sized *allocation-blocks*. Since 16 bits are used to refer to an allocation-block, there can be at most 2^{16} allocation blocks on an HFS filesystem. Thus, using 32 bits for identifying allocation blocks results in much less wasted space (and more files).
- Long file names up to 255 characters
- Unicode based file name encoding
- File/Directory attributes can be extended in future (as opposed to being fixed size)
- In addition to a System Folder ID (for starting Apple operating systems), a dedicated startup file that can easily be found (its location and size are stored in the volume header in a fixed location) during startup, is also supported so that non-Apple systems can boot from a HFS+ filesystem

- Largest file size is 2^{63} bytes

Aliases

Aliases are similar to symbolic links in the sense that it allows multiple references to a file or directory. However, if you move the target (without replacing it), a symlink would break, while an alias would not. This is possible because under HFS+, each file/directory has a unique, persistent identity, which is stored along with the pathname. If one of the two (pathname or unique identity) is wrong (the file cannot be found using it), the alias updates it with the "right" one (using which the file *could* be found). This feature is the reason why you can keep moving applications to different places on your disk without having to worry about breaking their Dock "shortcuts".

In order to make use of aliases, an application must use either Carbon or Cocoa APIs, since this feature is not available through the POSIX API.

Optimizations

HFS+ also has a few specific optimizations. When a file is opened on an HFS+ volume, the following conditions are tested:

- The file is less than 20 MB in size
- The file is not already busy
- The file is not read only
- The file is fragmented (the eighth extent descriptor in its extend record has a non-zero block count)
- The system uptime is at least 3 minutes

If *all* the above are satisfied, the file is *relocated* (de-fragmented) - on-the-fly.

Another optimization is "Hot File Clustering". This is a multi-staged (the stages being `DISABLED`, `IDLE`, `BUSY`, `RECORDING`, `EVALUATION`, `EVICTION` and `ADOPTION`) clustering scheme that records "hot" files (except journal files, and ideally quota files) on a volume, and moves "hot" files to the "hot" space on the disk (0.5% of the total filesystem size located at the end of the default metadata zone - at the start of the volume). The scheme uses an on-disk B-Tree file for tracking (`/.hotfiles.btree` on a volume):

```
# ls -l /.hotfiles.btree
-rw------- 1 root  admin  196608 17 Dec
10:09 /.hotfiles.btree
```

At most 5000 files, and only files less than 10 MB in size are "adopted" under this scheme.

Multiple Forks

HFS+/HFS files have two "forks" - traditionally called the *data* and *resource* forks, either of which may be *empty*. Historically, a resource fork has been used for various things such as custom icons, preferences, license information, etc. As expected, this is incompatible with traditional Unix filesystems, and care must be taken while moving files across filesystems. The article *Command Line Archival in Mac OS X* takes a brief look at the issue and some solutions. From a BSD command line, a file's resource fork can be accessed thus:

```
% ls -l Icon
128 -rwxrwx---  1 amit  amit  0 11 Jun  2003
Icon
# this means the data fork is empty
% ls -l Icon/rsrc
128 -rwxrwx---  1 amit  amit  65535 11 Jun
2003 Icon/rsrc
# the resource fork has is 65535 bytes
```

Thus, unlike aliases, multiple forks *can* be accesses via the POSIX API.

Device Driver Partitions

Although this is not related to HFS+, Mac OS X can load block device drivers from various places: the ROM, a USB or FireWire device and special partitions on a fixed disk. In order to support multiple operating systems or other features, a disk can have more than one device driver

installed, each in its *own partition*. For example, viewing your disk drive's information in `/Applications/Utilities/Disk Utility.app` might tell you that Mac OS 9 drivers are installed on this disk. A "standard" partition scheme on the PowerBook might look like the following:

```
# pdisk /dev/rdisk0 -dump
/dev/rdisk0  map block size=512
#: type name                          length
base      ( size )
1: Apple_partition_map Apple             63 @ 1
2: Apple_Driver43*Macintosh              56 @
64
3: Apple_Driver43*Macintosh              56 @
120
4: Apple_Driver_ATA*Macintosh            56 @
176
5: Apple_Driver_ATA*Macintosh            56 @
232
6: Apple_FWDriver Macintosh             512 @
288
7: Apple_Driver_IOKit Macintosh         512 @
800
8: Apple_Patches Patch Partition        512 @
1312
9: Apple_HFS Untitled           156299656 @
1824    ( 74.5G)
0: Apple_Free                             0+@
156301480
```

In the above dump, the `Apple_partition_map` is a meta-data partition describing the partitions on the disk. The patch partition is a meta-data partition containing patches that must be applied to the system before it can boot. The various "Driver" partitions contain drivers (`Apple_Driver43` contains SCSI Manager 4.3, for

example).

Insensitive!

Note that HFS+ is a *case preserving, case insensitive* filesystem, which can be rather jarring in situations such as the following:

```
# tar -tf freebsd.tar
FreeBSD.txt
freebsd.txt
# The tar file contains two files
# tar -xvf freebsd.tar
FreeBSD.txt
freebsd.txt
# ls *.txt
freebsd.txt
```

The Apple Technical Note titled *HFS Plus Volume Format* describes HFS+ internals in great detail.

ISO9660

`ISO9660` is a system-independent file system for read-only data CDs. Apple has its own set of ISO9660 extensions. Moreover, you would likely run into Mac HFS/ISO9660 hybrid discs that contain both a valid HFS and a valid ISO9660 filesystem. Both filesystems can be read on a Mac, while on "other" systems, you would typically read the ISO9660 data. Note that this doesn't mean there has to be redundant data on the disc: usually the data that needs to be accessed from both Macs and PCs is kept on the ISO9660 volume, and is aliased on the HFS volume.

MSDOS

Mac OS X includes support for MSDOS filesystem (FAT12, FAT16 and FAT32).

NTFS

Mac OS X includes read-only support for NTFS.

UDF

UDF (Universal Disk Format) is the filesystem used by DVD-ROM (including DVD-video and DVD-audio) discs, and by many CD-R/RW packet-writing programs. Note that at the time of this writing, Mac OS X "Panther" only supports UDF 1.5, and *not* UDF 2.0.

UFS

Darwin's implementation of UFS is similar to that on *BSD, as was NEXTSTEP's, but they are not really compatible. Currently, only NetBSD supports it. Apple's UFS is big endian (as was NeXT's) - even on x86 hardware. It includes the *new* Directory Allocation Algorithm for FFS (DirPref). The author of the algorithm offers more details, including some test results, on his site.

Network Filesystems

AFP

The Apple Filing Protocol (AFP) is an Apple proprietary protocol for file sharing over the network. A comparison of AFP and NFS is outside of the scope of this document, but there exists software that enables the two to co-exist (AFP shares can be made to look like NFS shares and vice-versa).

`/usr/sbin/AppleFileServer` is the AFP daemon, which is launched when you select the "Personal File Sharing" checkbox under `System Preferences/Sharing`.

FTP

The `mount_ftp` command mounts locally a directory on an FTP server. Note that this functionality is *read-only* currently. It works transparently through the Finder and the Web Browser.

```
mount_ftp ftp://user:password@hostname/
directory/path node
```

NFS

Mac OS X includes NFS client and server support (version 3) from BSD, including the "NQ" (NFS with leases) extensions. The usual supporting daemons (`rpc.lockd`, `rpc.statd`, `nfsiod`, etc.) are present as well.

SMB/CIFS

Mac OS X "Panther" includes Samba 3.0 to support SMB/CIFS.

WebDAV

A WebDAV enabled directory located at a server specified by an appropriate URL can be mounted as a filesystem via the `mount_webdav` command. Since a .Mac account's iDisk is available through WebDAV, it can be mounted this way.

Note that Mac OS X has a (preliminary) system level event notification framework built around FreeBSD's kqueue/kevent. This can allow graceful mounts and unmounts of network volumes based on changes in network connectivity.

Other/Pseudo Filesystems

cddafs

The `cdda` filesystem is used to make the tracks of an audio CD appear as `aiff` files. Moreover, if the track names can be looked up successfully, the track "files" have corresponding names.

When you insert an audio CD in the drive, Mac OS X "mounts" it using `cddafs` by default, or you can manually do so as follows:

```
# mount_cddafs /dev/disk<N> /tmp/audiocd
```

deadfs

When the underlying filesystem is *disassociated* from a vnode (in the `vclean()` operation), its vnode operations vector is set to that of the *dead filesystem*. All operations in the dead filesystem fail, except for `close()`.

`deadfs` essentially facilitates *revocation* (of access to the controlling terminal, to a forcibly unmounted filesystem, etc.) Consider a situation where you want a backgrounded job of a logged out user to finish what it's doing, and yet have no access to its (erstwhile) controlling terminal. This is achieved by detaching the terminal from the vnode and replacing it with `deadfs`.

devfs

`devfs`, the device filesystem, provides access to the kernel's device namespace in the global filesystem namespace. `devfs` is typically mounted on `/dev` and allows entries in there to be built automatically.

`devfs`, if enabled, is mounted from within the Mac OS X kernel during BSD initialization, although instances of it can be mounted later on using `mount_devfs`.

```
# mount -t devfs devfs /tmp/dev
```

fdesc

The `fdesc` filesystem is typically mounted on `/dev/fd`. It's functionality is similar to `/proc/<pid>/fd` (or simply `/proc/self/fd`) on Linux, that is, it provides a list of all active file descriptors for the currently running process. Note that a typical Linux

system has `/dev/fd` symbolically linked to `/proc/self/fd`.

`/etc/rc` mounts the `fdesc` filesystem during system startup:

```
# mount -t fdesc -o union stdin /dev
```

fifofs

The purpose of `fifofs` is similar to `specfs`

loop*

Functionality similar to Linux's "loop" mounts (or "lofi" on Solaris) is available via the Finder (or simply on the Desktop) - simply double-clicking on a disk image file mounts its filesystem (if supported). The command line utility `hdid` can be used for a finer grained control of this functionality:

```
# hdid floppy.img
/dev/disk3
# hdid http://127.0.0.1/disk.img
/dev/disk4
```

The "disks" `disk3` and `disk4` can be accessed as regular disks. Note that if the disk image to be mounted using HTTP is a dual-fork file, then it is trickier to use it.

Moreover, `hdid` can be directed to use only a subset (a range of sectors) of a disk image. There is also support for encryption, and more importantly, shadowing, wherein a "shadow" file can be used to which all writes can be redirected. When a read occurs in such a case, blocks present in the shadow file have precedence over the ones in the image.

nullfs

The null mount filesystem is a *stackable* filesystem in 4.4BSD. It allows mounting of one part of the filesystem in a different location. This can be used to join together multiple directories into a new directory tree. Thus, filesystem hierarchies on various disks can be presented as one directory tree, subtrees of a writable filesystem can be made read-only, and so on.

Note that this is slightly different (less seamless) from a union mount (see below). While the latter essentially combines seamlessly the filesystems of the mount point and the mounted, nullfs simply intercepts VFS/vnode operations and passes them through (with the exception of `vop_getattr()`, `vop_lock()`, `vop_unlock()`, `vop_inactive()`, `vop_reclaim()`, and `vop_print()` to the original filesystem (of the mount point).

Note that the null filesystem layer also serves as a prototype filesystem, and new layers can be implemented by using the null layer as a template.

Finally, it should be noted that although `nullfs` is present in the `bsd` subtree of Darwin's kernel source, null mounts are not really used by Mac OS X.

ramfs

A ram filesystem can be created under Mac OS X as follows:

```
# hdid -nomount ram://1024
/dev/disk3
```

The above command creates a ram disk with 1024 sectors (sector size being 512), and prints the name of the resultant device on the standard output. Thereafter, a filesystem can be created on this device (the corresponding raw device, technically) as follows:

```
# newfs_msdos /dev/rdisk3
/dev/rdisk3: 985 sectors in 985 FAT12
clusters \
(512 bytes/cluster) \
bps=512 spc=1 res=1 nft=2 rde=512 sec=1024
mid=0xf0 \
spf=3 spt=32 hds=16 hid=0
```

The disk can be mounted as usual:

```
# mount -t msdos /dev/disk3 /tmp/msdos
# mount
/dev/disk3 on /private/tmp/msdos (local)
# df /tmp/msdos
Filesystem 512-blocks Used Avail Capacity
Mounted on
/dev/disk3        987    2   985     0%   /
private/tmp/msdos
```

Finally, you can get rid of the ram disk as follows:

```
# hdiutil detach /dev/disk3
"disk3" unmounted.
"disk3" ejected.
```

specfs

Devices (the so called "special" files) and FIFOs can reside on any arbitrary filesystem (that can house such files). This means their names and attributes are maintained by this "host" filesystem. However, their operations cannot be handled by this filesystem - accesses to such device files need to be mapped to their underlying devices (more

specifically, the respective device drivers). Moreover, device aliases (for example, same major/minor numbers, but different pathnames on a filesystem, or maybe even different filesystems) need to be detected and handled appropriately.

The `specfs` layer facilitates the above. Note that `specfs` is not a user visible filesystem, and it's not "mounted" anywhere.

synthfs

`synthfs` is a pseudo (in-memory) filesystem used to create arbitrary directory trees (if you wanted to "synthesize" mount points for random things, for example). `synthfs` is *not* derived from FreeBSD.

A `synthfs` mount is similar to a typical pseudo filesystem mount:

```
# mount -t synthfs synthfs /tmp/synthfs
```

union

A detailed description of 4.4BSD's "union" mounts, including a short history of similar filesystems, can be found in the USENIX paper titled *Union Mounts in 4.4BSD-Lite*. In the simplest terms, the union mount filesystem extends the null filesystem by *not hiding* the files in the "mounted on" directory. It *merges* the two directories (and their trees) into a single view. Note that duplicate names are suppressed and a lookup locates the *logically topmost* entity with that name.

Consider the following sequence of commands that illustrates the basic concepts of union mounts:

```
# hdiutil create /tmp/msdos1 -volname one \
-megabytes 1 -fs MS-DOS
...
created: /tmp/msdos1.dmg
```

```
# hdiutil create /tmp/msdos2 -volname two \
-megabytes 1 -fs MS-DOS
...
created: /tmp/msdos2.dmg
# hdid -nomount /tmp/msdos1.dmg
/dev/disk3
# hdid -nomount /tmp/msdos2.dmg
/dev/disk4
# mount -t msdos /dev/disk3 /tmp/union
# echo "msdos1: a" > /tmp/union/a.txt
# umount /dev/disk3
# mount -t msdos /dev/disk4 /tmp/union
# echo "msdos2: a" > /tmp/union/a.txt
# echo "msdos2: b" > /tmp/union/b.txt
# umount /dev/disk4
# mount -t msdos -o union /dev/disk3 /tmp/
union
# mount -t msdos -o union /dev/disk4 /tmp/
union
# ls /tmp/union
a.txt b.txt
# cat /tmp/union/a.txt
msdos2: a
# umount /dev/disk4
# ls /tmp/union
a.txt
# cat /tmp/union/a.txt
msdos1: a
# umount /dev/disk3
# mount -t msdos -o union /dev/disk4 /tmp/
union
# mount -t msdos -o union /dev/disk3 /tmp/
union
# cat /tmp/union/a.txt
msdos1: a
```

As a real-life example, `/etc/rc` mounts the "descriptor" filesystem as a union mount:

```
# mount -t fdesc -o union stdin /dev
```

volfs

`volfs`, the "volume" filesystem, is a virtual filesystem that exists over the HFS+ (or a filesystem that supports `volfs`) VFS and serves the needs of two differing APIs (POSIX/Unix pathnames and Mac OS `<Volume ID><Directory><File Name>`). It is there to support the Carbon File Manager APIs on top of the BSD filesystem.

The filesystems that support `volfs` are HFS+, HFS, ISO9660 and UDF.

Consider the following example:

```
# mount
/dev/disk0s9 on / (local, journaled)
devfs on /dev/ (local)
fdesc on /dev (union)
<volfs> on /.vol
...
# ls -l /.vol
total 0
dr--r--r--  2 root  wheel  64 25 Dec 12:45
234881033
```

The entry in `/.vol` is nothing but a representation of `/`. Each mounted volume (a partition, if you will), such as those on external storage devices, would have a representation under `/.vol`. Consider a file, say `/mach_kernel`:

```
# ls -li /mach_kernel
1045670 -rw-r--r--  1 root  wheel  3824080 11 Dec
16:20 /mach_kernel
```

This file would be accessed under `/.volfs` as follows (note that `1045670` is the file's inode number):

```
# ls -li /.vol/234881033/1045670
1045670 -rw-r--r--  1 root  wheel  3824080 11 Dec
16:20 /.vol/234881033/1045670
```

`volfs` is mounted during system startup (in `/etc/rc`):

```
# mkdir -p -m 0555 /.vol && chmod 0555 /.vol &&
mount_volfs /.vol
```

What about XYZ?

While Mac OS X supports many filesystems, you might run into some that are not supported. Linux's `ext2/3` and `Reiser`, for example, are not supported, although you can find an open source implementation of `ext2` for Mac OS X.

Interestingly, BootX, the Mac OS X bootloader, *does* understand the `ext2` filesystem, and can load kernels from it.

An important (though not necessarily critical) omission is that of the `proc` filesystem. This issue, including description of a minimal port

of `/proc` from FreeBSD to Mac OS X, is discussed in *`/proc` on Mac OS X*.

Programming on Mac OS X

Mac OS X is a fertile field for all kinds of programming endeavors, especially if you have a *nix frame of reference. Life is still *much* better for a developer on Windows than on Mac OS X - no matter what one might think of the usability, etc. of Windows. Apple has been trying to improve things for developers lately, which is a good sign.

This page discusses some programming facilities, frameworks and tools available on Mac OS X.

Application Environments

Since Mac OS X is derived from various sources, it has a multitude of Application Environments. We discussed these in *Above the Kernel*. To recapitulate:

BSD

Mac OS X uses FreeBSD as a reference code base for its BSD derivations (Panther derives from FreeBSD 5.0). It includes a BSD-based POSIX API (BSD style system call vector, `uap` based argument passing, etc.). An example of intercepting system calls on Mac OS X is covered in the article *Re-routing System Calls*. OS X also supports System V IPC, asynchronous I/O, `poll()` (emulated over `select()`), etc. Arbitrary C programming is not any different than on any generic Unix platform. Here is an example of re-routing function calls by overwriting and injecting machine instructions.

Carbon

This is a set of procedural C-based APIs for Mac OS X that are based on the old Mac OS 9 API (actually dating back as far back as Mac OS 8.1). Carbon was originally designed to provide an easy development migration path from Mac OS 9 to Mac OS X. The Carbon APIs are all-encompassing (they include legacy interfaces), covering most things that you are likely to do programmatically on Mac OS X. Carbon specific code is not portable to other platforms.

Classic

Mac OS X includes a Classic (Mac OS 9) emulator that executes in a protected memory environment so as to let users run programs written for Mac OS 9. Apple does not encourage you to actually develop for this environment.

Cocoa

This is an object-oriented Objective-C based API that's the preferred way of doing things in Mac OS X (if what you want to do can be done through Cocoa), particularly because of how well it's supported by Apple's Rapid Development tools. However, there are many parts of Mac OS X, and applications from 3rd party vendors, that have not converted to Cocoa completely, or at all. A Cocoa application can call the Carbon API when necessary. Cocoa is largely based on the OpenStep frameworks, and consists of primarily two parts: the Foundation (fundamental classes) and the Application Kit (classes for GUI elements).

> Although Cocoa is not really portable across platforms, you might be able to get your Cocoa programs to work on a number of platforms if you take GNUstep into account. Here's what the GNUstep FAQ has to say about portability between Cocoa and GNUstep (quoted verbatim):
>
> *It's easier from GNUstep to Cocoa than Cocoa to GNUstep. Cocoa is constantly changing, much faster than GNUstep could hope to keep up. They have added extensions and new classes that aren't available in GNUstep yet. Plus there are some other issues. If you start with Cocoa:*
>
> - *Be careful with Cocoa.h includes (Use #ifndef GNUSTEP, for instance)*
> - *Do not use CoreFoundation*
> - *Do not use Objective-C++*
> - *Do not use Quicktime or other proprietary extension*
> - *You need to convert .nib files (see section 1.1.3 Tools for porting)*
> - *Some unfinished classes include NSToolBar and Drawers.*
>
> My definition of "portability" (in the current context) is not about *feasibility*, but the practicality of doing so. Given enough resources, one could port anything to anything - often by

emulating/implementing the "source" API on the "target". The WINE project is a wonderful effort. Microsoft once used help from Mainsoft to get Internet Explorer and Outlook Express to run on Solaris. Still, Win32 code is not portable from a practical viewpoint.

Java

Mac OS X includes a complete J2SE implementation. The Swing implementation generates native OS X GUI elements for a uniform look and feel. JAR files are treated as shared libraries. Note that Cocoa includes Java packages that let you create a Cocoa application using Java as the programming language.

X11

Mac OS X includes (optionally) an X Window System implementation based on XFree86 4.3+. The X server has been optimized for OS X via integration with Quartz and supports OpenGL, rootless and full screen modes, an Aqua-compatible window manager (`quartz-wm`) and a menu in the Dock. The presence of a good quality X server and the X11 SDK is a big win because it makes possible to port (in most cases with no or minor changes) a large number of existing X11 applications to Mac OS X, including use of toolkits such as GTK, KDE, various others.

As mentioned in *Architecture of Mac OS X*, Mac OS X has a number of very different APIs due to the many environments constituting it. The example of BSD and Carbon Process Manager processes was given before. Similarly, what thread API you use on Mac OS X is determined by what environment you are programming in. Mach provides low-level kernel threads. The `pthread` library, `/usr/lib/libpthread.dylib` (actually a symlink to `libSystem.dylib`) provides POSIX threads. Carbon includes a threads package for cooperatively scheduled threads (Thread Manager) and another for preemptively scheduled threads (Multiprocessing Services). Cocoa uses the `NSThread` class, while Java uses `java.lang.Thread`. All these are built using `pthreads`.

Bundles and Frameworks

Mac OS X uses a few concepts not found on many traditional BSD, Linux, Solaris etc. systems.

Bundle

A **Bundle** is a directory that stores executable code and the software resources (icons, splash images, sounds, localized character strings, interface builder "nib" files, `.rsrc` resource files, etc.) related to that code. Although a bundle is a directory containing potentially numerous subdirectories and files, it is treated as a single entity for various purposes.

Mac OS X can have different kinds of bundles:

- An "Application" bundle (such as `Foo.app`) contains everything (except frameworks/libraries coming from elsewhere) needed to run the `Foo` application. It is possible to simply drag `Foo.app` to any location and it will work as expected (you do not even have to do anything to its Dock icon, if any - courtesy the fact that "aliases" on HFS+ do not break if you move a file without replacing it). The `/Applications` directory contains many such bundles.
- A "Framework" (such as `Bar.framework`) is a versioned bundle containing resources such as headers, documentation, etc. The `/System/Library/Frameworks` directory contains numerous frameworks (such as for Kerberos, Python, QuickTime, ScreenSaver, and so on).
- A "Loadable" bundle can be a kernel extension (a `.kext`, similar to a loadable kernel module on Linux, say), many of which exist in `/System/Library/Extensions`, a Plug-in or a Palette.

One of the Finder flags is `kHasBundle`, which, if set, makes the bundle appear as a file package (a single opaque entity), with exceptions and specific behavior for different bundle types.

The various bundle extensions referred to above are only conventions - a bundle can have any extension. For example, instead of a `.app`, you can have a `.debug` or `.profile` to imply debug or profile code, respectively.

Framework

A **Framework**, as stated above, is a type of a bundle that contains shared resources such as dynamic shared libraries, header files, icons

and images, documentation, etc. Moreover, frameworks are *versioned*. Major versions are incompatible while minor versions are compatible. One framework can have multiple major versions.

Consider an example:

```
# ls -lF /System/Library/Frameworks/OpenGL.
framework
total 32
lrwxr-xr-x ... Headers@ -> Versions/Current/
Headers
lrwxr-xr-x ... Libraries@ -> Versions/
Current/Libraries
lrwxr-xr-x ... OpenGL@ -> Versions/Current/
OpenGL
lrwxr-xr-x ... Resources@ -> Versions/
Current/Resources
drwxr-xr-x ... Versions/
```

Except `Versions/`, everything else is a symbolic link (to entities from the current major version). The file `OpenGL` is the dynamic shared library:

```
# file -L OpenGL
OpenGL: Mach-O dynamically linked shared
library ppc
```

The default path for searching frameworks (as used by the dynamic link editor) is:

```
$(HOME)/Library/Frameworks
```

```
/Library/Frameworks
/Network/Library/Frameworks
/System/Library/Frameworks
```

As we have seen, Mac OS X has complex entities (like a `.app` directory tree) exposed as a single, click-able entity through the Finder. The same effect as double-clicking on an entity's icon can be achieved on the command line through the `open` utility. It opens a file, folder, or a URL, in an appropriate manner. For example, opening a `.app` folder would launch that application, opening a URL would launch an instance of the default web browser with that URL, opening an MP3 file would open it in the default MP3 player, etc.

Runtime Environments

Mac OS X has two primary runtime environments: one based on the dynamic link editor, `dyld`, and the other based on Code Fragment Manager (CFM). OS X does *not* support ELF, and there's no `dlopen`, although the `dlcompat` library provides a limited compatibility layer (using native OS X functions) so that common Unix source can be compiled easily.

CFM determines addresses for referenced symbols in executables at *build time* (a static approach). The executable format used is called PEF (Preferred Executable Format). `dyld` resolves undefined symbols at execution time. The executable format is Mach-O (Mach object-file-format).

Mac OS X is natively a `dyld`/Mach-O platform - all system frameworks are built using `dyld`. In fact, the CFM/PEF environment is itself built on top of `dyld`/Mach-O. However, there exist provisions to call `dyld` code from CFM code. Moreover, if you wish to debug or trace a CFM application using GDB, you would need to use a Mach-O program called `LaunchCFMApp`:

```
/System/Library/Frameworks/Carbon.framework/
Versions/\
A/Support/LaunchCFMApp
```

`dyld`/Mach-O is similar in many respects to `ld.so/ELF`, although they differ both conventionally and fundamentally. Some of these are:

- Dynamic shared libraries on Mac OS X have the `.dylib` extension. The functionality of several traditional libraries (such as `libc`, `libdl`, `libinfo`, `libkvm`, `libm`, `libpthread`, `librpcsvc`, etc.) is provided by a single dynamically loadable framework, `libSystem`. `libc.dylib` etc. are simply symbolic links to `libSystem.dylib`.
- Mac OS X builds libraries and applications with a two-level namespace (as compared to a flat namespace in traditional Unix systems). This topic is described in an Apple Developer Note called *Two-Level Namespace Executables*. This also means that `DYLD_INSERT_LIBRARIES`, the `dyld` analog of `ld.so`'s `LD_PRELOAD` will *not work* with two-level namespace libraries. You can force a flat namespace, but that often messes up things enough to stop the application from running at all. Additionally, you *cannot* have "weak" symbols (symbols that can be overridden) in libraries.
- Functionality similar to `ldd` (on Linux, say) is provided by `/usr/bin/otool`.

Mac OS X uses Mach-O and not ELF simply because NEXTSTEP used Mach-O. Apple had a large enough TODO list that moving to ELF for the sake of mainstream conformity was not justified. Like NEXTSTEP, Mac OS X supports "fat" binaries where an executable image contains binaries for more than one platform (such as PowerPC and x86). Apple's port of GNU CC allow for fat binaries to be produced (provided assemblers and libraries are available for each specified architecture).

Prebinding

Mac OS X uses a concept called "prebinding" to optimize Mach-O applications to launch faster. Prebinding is the reason you see the "Optimizing ..." message when you update the system, or install certain software.

The dynamic link editor resolves undefined symbols in an executable (and dynamic libraries) at run time. This activity involves mapping the dynamic code to free address ranges and computing the resultant symbol addresses. If a dynamic library is compiled with prebinding support, it

can be predefined at a given address range. This way, `dyld` can use predefined addresses to reference symbols in such a library. Of course, for this to work, libraries cannot have preferred addresses that overlap. Apple specifies address ranges for 3rd party (including your own) libraries to use to support prebinding.

`update_prebinding` is run to (attempt to) synchronize prebinding information when new files are added to a system. This can be a time consuming process even if you add or change a single file, say, because all libraries and executables that might dynamically load the new file must be found (package information is used to help in this, and the process is further optimized by building a dependency graph), and eventually `redo_prebinding` is run to prebind files appropriately.

`/usr/bin/otool` can be used to determine if a binary is prebound:

```
# otool -hv /usr/lib/libc.dylib
/usr/lib/libc.dylib:
Mach header
magic    cputype cpusubtype filetype ncmds
sizeofcmds flags
MH_MAGIC PPC     ALL         DYLIB    10
1940      \
NOUNDEFS DYLDLINK PREBOUND SPLIT_SEGS
TWOLEVEL
```

Xcode

Xcode is Mac OS X "Panther"'s developer tools package. It includes components typical of a comprehensive IDE:

- A source code editor with code completion
- A file browser/organizer
- Support for version control (CVS and Perforce)
- A documentation viewer that can link symbols in the code to documentation

- A class browser
- Various compilers (the GNU suite, including `gcc 3.3` and integration with `distcc` for distributed builds, `javac`, `jikes`)
- GDB based graphical and command-line debugging
- An Interface Builder application that provides a GUI for laying out interface objects (various GUI elements), customize them (resize, set and modify attributes), connect different objects together, and so on
- Support for packaging

A new Xcode project can be instantiated from a large number of templates. As can be seen, it supports development of various kinds of programs in C, C++, Objective-C, Objective-C++, Java and Assembly. For example, it is almost trivial to create things such as Screen Savers, Preference Panes (the kind you see under System Preferences), etc.

Xcode has some neat *and* useful features: Predictive compilation runs the compiler in the background as you edit the source. Once you are ready to build, the hope is that most of the building would have been done already. "Zero Link" links at runtime instead of compile time, whereby only code needed to run the application is linked in and loaded. A related feature is "Fix and Continue", courtesy which you can make a change to your code and have the code compiled and inserted into a running program. Distributed builds are also supported via integration with `distcc`.

Programming Languages

Compilers and Libraries

Apple provides a customized/optimized GNU CC, with backends for C, C++, Objective-C and Objective-C++. For Java, two compilers are included: `javac` and IBM's `jikes`. Compilers for many other languages are available either precompiled (such as the XL Fortran Advanced Compiler from IBM), or can be compiled from source, which is not any harder in general than compiling the same source on, say, Linux or FreeBSD. The same goes for development libraries - it should be easy to compile many open source platform-independent / multi-platform libraries from source on OS X. Many important libraries and APIs are either included with Mac OS X, or are readily available (Gtk/ Gtk++, Java, OpenGL, Qt, QuickTime, Tcl/Tk, X11R6). The system

comes with several special purpose (and/or optimized) libraries as well, such as for Numerical Computing and Image processing (`BLAS`, `vBigNum`, `vDSP`, `vImage`, `LAPACK`, `vMathLib`, etc.)

Interpreters

A number of scripting languages are included in Mac OS X, such as AppleScript (including the AppleScript Studio IDE), Perl (Mac OS X "Panther" has perl 5.8.1), PHP, Python (Panther has python 2.3, with bindings to CoreGraphics), Tcl, and Ruby. Mac OS X also supports the Open Scripting Architecture (OSA), using which it is possible to get JavaScript (a port of Mozilla JavaScript) in the form of an OSA component. You can get support for more languages (Lisp, Scheme, ...) via Fink, and it should be straightforward to compile most of them from source, if desired. `bash`, `tcsh`, and `zsh` are the *nix shells included.

Scriptability

AppleScript is the preferred scripting system on Mac OS X, providing direct control of many parts of the system as well as applications. In other words, AppleScript lets you write scripts to automate operations, exchange data with and send commands to applications, etc.

While I personally find AppleScript syntax to be exasperating (I never really enjoyed COBOL for that matter), it *is* useful. Consider a contrived example for a taste of AppleScript:

```
tell application "Finder"
    set system_version to (get the version)
as number
    do shell script "say 'This is Mac OSX'"
& system_version
    return system_version
end tell
```

Running the above (either in `ScriptEditor` or through the `osascript` command line utility) should cause the `say` utility (see below) to *speak* the version of Mac OS X (well, assuming you *are* running Mac OS X) to you. By the way, you do *not* need to run a shell

command and use the `say` utility: AppleScript has a `say` command of its own.

Now, whether you are impressed by (or happy with) Mac OS X's *scriptability* depends on which system you are contrasting with. As mentioned at the beginning of this section, if you are coming from a traditional Unix/Unix'ish world (FreeBSD, Linux, Solaris, *nix, ...), you *should* find it impressive, a lot perhaps (barring the miniscule possibility that you are an anti-Mac/pro-XYZ zealot). However, you might not think much of this if you have been using Microsoft's COM/DCOM/.NET.

Along the lines of making a *nix developer/user happy though, Apple has been exposing various aspects of Mac OS X functionality to be driven via a traditional command line. Consider some examples:

`drutil` is a command line tool that interacts with the DiscRecording framework. The command:

```
# drutil getconfig supported
```

will probably tell you more than you want to know about your CD/DVD burner.

`hdiutil` is a powerful utility for manipulating disk images.

`say` is a command line utility to convert input text to speech using the Speech Synthesis manager.

`sips` is a command line interface to the Scriptable Image Processing Server. The graphical abilities of Mac OS X are exposed through this image processing service. The SIPS architecture contains tools for performing basic image alterations and support various image formats. The goal is to provide quick, convenient, desktop automation of common image processing operations.

Tools

The Developer Framework contains a wealth of tools for various programming, debugging, profiling and other developmental tasks. The

CHUD (Computer Hardware Understanding Development) Tools are particularly useful for various kinds of benchmarking and optimization. *Mac OS X Developer Tools Overview* contains detailed information on almost all tools included in Xcode. *Mac OS X Hacking Tools* is a compendium of some useful programs on Mac OS X (mostly outside of Xcode).

Lastly, the OS X `Terminal.app`, is one of the better terminal applications around, which *does* matter to me while programming, since I do not use an IDE for everything. The terminal advertises itself as `xterm-color`, has Unicode support (though broken), drag-and-drop support (dragging an icon to the terminal would drop its pathname or URL string), transparency support (has a minor problem in Panther - there's ghosting of text under certain conditions), etc.

A Sampling of Mac OS X Features

Mac OS X has many "cool", "interesting", and useful features, a number of which directly contribute to the overall usability of the system. This page briefly describes a few features representative of why Mac OS X is a good (Desktop) operating system.

Aqua

The graphical user interface of Mac OS X is called *Aqua*. This includes the look and feel, behavior, and integration of GUI elements. The GUI application environments of Mac OS X, Carbon, Cocoa, and Java, all support Aqua. Classic does not, and Mac OS 9 applications running under Classic look like they did on Mac OS 9. Finally, Mac OS X includes an optimized X Window server, including a native Aqua window manager (`quartz-wm`) that lets you run X11 applications alongside native Aqua programs. `quartz-wm` provides Aqua window controls, drop shadows, etc. However, the X11 application's own look and feel will be the one provided by the particular X11 toolkit being used.

Aqua has numerous distinctive features: Mac OS X uses high-quality photorealistic icons that are rendered at various sizes up to 128x128, allowing for features such as in-place document preview and in-icon status indication.

Mac OS X has a number of functional and unique user interface elements, such as *sheets*, which are document-modal dialogs that attached to and appear to come out of a document's title bar. The Desktop, Dock, and the Finder are also different (in my opinion, mostly

better, from a productivity/usability point of view) from their counterparts on Windows and *nix.

Aqua Human Interface Guidelines is a thorough description (almost 300 pages) of what guidelines to adhere to while creating applications for Mac OS X as well as an overview of various Aqua GUI elements.

While the lucidity and attractiveness of Aqua is visible immediately as you look at a Mac OS X desktop (with the disclaimer that this is a subjective area, so many people may *not* like how it looks), it may take a little while to get used to before you appreciate Aqua's usability.

Data and Information Management

While not really Utopian, Mac OS X makes a very good attempt at keeping various data and information organized by context, rather than having files strewn all over the place. System and Application "preferences" can be global (system-wide) or per-user, and are kept organized as such. The various APIs make sure that (if used properly), all of a user's data is stored deterministically.

One of the most useful features of Mac OS X is its support for *synchronization* of your computer's configuration, or personality, if you will. Currently this data set includes the address book, calendar, and Safari bookmarks, although Apple should add more entities. If you are doing a new installation or clean upgrade of your system, it is a boon to have the following: on the "old" installation, click a button to synchronize the above information to a device, which can be your iPod, or your .Mac account. On the "new" machine, you can reverse synchronize and have this information injected.

Devices

Apple has excellent support for FireWire devices, hardly surprising since they invented FireWire. You can readily boot from external drives, treat a Mac to behave as if it were an external FireWire drive (boot it with the T key pressed, which puts the computer in *Target Disk Mode*), and even connect two computers together using TCP/IP over a FireWire cable. The iSight and the iPod use FireWire connections too.

Apple has also been pushing Bluetooth with their newer computers, although you can get adapters for older models. In addition to using Bluetooth for communicating with phones and PDAs, Apple also uses it for their wireless keyboard and mouse, which are very well designed and work well with Mac OS X (well, the mouse still has *one* button).

Even though Apple computers have custom "chips" (the *KeyLargo* IC, for example, is an I/O controller that provides USB, UDMA, EIDE, sound, communication support, etc. all on a single IC), Apple uses a number of "standard" components (RAM, IDE/SATA disk drives, optical drives, ...) in their machines, things are not always black and white. For example, an arbitrary DVD burner may not work with iDVD, though usually a workaround can be found.

I18N

Mac OS X is localized to a number of regions. It supports Unicode 4.0, various input methods, and multiscript support (a single document can contain multiple scripts). Apple provides tools, including support in Xcode, so that developer can internationalize their applications. Some specific components included are ICU, libiconv and support for `wchar_t`.

I was most impressed by how easy it is to input various Indian languages, including Hindi, on Mac OS X - *out of the box*. I can write an email containing English, Hindi, and other languages, using the QWERTY keyboard to input Hindi *phonetically* - it is very intuitive (assuming you do know Hindi, of course), and far better than my experience on other systems.

iLife

Apple's iLife suite (iDVD, iMovie, iPhoto, iTunes, and GarageBand) are quite possibly the best applications you can get bundled with any operating system. While I do not use iDVD and iMovie myself (I have been experimenting with their professional counterparts, DVD Studio Pro and Final Cut Pro), I find iTunes and iPhoto to be excellent at what they set out to do, although nitpickers can find "issues" no matter what. These two applications manage and organize your assets (music and digital photos), with the option of leaving the "originals" untouched. Asset metadata is stored in a relational database, into which you can have multiple views.

iPhoto

iPhoto can import images from a digital camera. It lets you do some basic operations on images, such as constraining as per various standard sizes, crop, resize, enhance, do red-eye reduction, retouch, convert to black and white, adjust brightness and contrast, etc. You can have a number of output channels for your photos: you can print them on a printer, order prints using Kodak's online print service, email pictures,

create a slideshow, burn them onto optical media, create .Mac slides, export them to a thumbnailed web gallery, and even create a *book* in WYSIWYG fashion that Apple can print and bind for a fee.

iTunes

Unless you have some uniquely specific needs, have a gargantuan gripe against Apple, or are a masochist, you would probably find iTunes on Mac OS X to be the final word in music management. iTunes is a powerful and sophisticated jukebox, an interface to Apple's online music store, and a companion program to the iPod. I believe it represents the best that software can possibly do in making your music experience on a computer pleasant.

Power Management

Mac OS X in conjunction with Macintosh hardware make up for some impressive power and thermal management. The four independently controlled thermal zones and the nine fans of the Power Mac G5 have been discussed aplenty. Mac OS X includes drivers and other logic for the 21 temperature sensors in that machine. The PowerBooks have sophisticated thermal management as well. You can `'grep -i'` for "thermal" and "temperature" in the output of `'ioreg -l'` on Mac OS X for related miscellaneous information.

Based on power/thermal feedback, processor and bus speeds can be reduced to conserve power and control heat. All current Apple notebook batteries have remaining charge indicators. On the PowerBooks, you can change the battery without powering-off courtesy an internal backup battery that holds charge for a few minutes.

Note that by default the system tries to keep network connections alive even if the machine sleeps. For example, if you login (via SSH, say) from one PowerBook to another, and both of them go to sleep, your login will stay alive.

Security

Mac OS X includes various security features, services, and APIs (including what's available on typical *nix systems), such as those for controlling/managing passwords, certificates, public/private keys, application-level privileged operations (capabilities), trust policies, etc. Mac OS X supports Kerberos, OpenSSL, and PAM as well.

Note that many of the above services are exposed through the Keychain

Services API, which any application can use, for example, to "remember" your passwords. It is possible to have a single keychain password instead of multiple passwords across different applications.

root login is disabled by default, and `sudo` is used for administrative access. You can use `/usr/bin/security` from the command line to control the security framework.

Relatively recent security related enhancements include FileVault (encryption of a user's home directory) and Secure File Deletion (see above). As mentioned earlier, Mac OS X is not a trusted system, or as focussed on security as say, OpenBSD, but it should at least be as secure as any modern day *nix system. It does make use of a large amount of open source software, so it would share many of the weaknesses and strengths of those components. Potentially, Apple's integration of such components might create new weaknesses, but I have found their software engineering to be extremely admirable in general.

Speech Interface

Mac OS X includes both speech recognition (part of Carbon) and synthesis frameworks, that are fairly well integrated with the system. Applications can make use of APIs to these frameworks. "Speakable Items", a user customizable interface to the speech recognition engine, is available to arbitrary applications, wherein you can add your own items. You can do the following, for example:

- Open System Preferences, and click on "Speech" under the "System" category. Click on "Speech Recognition".
- Under the "On/Off" tab, select "On".
- Under the "Listening" tab, make sure the appropriate microphone (such as "Internal microphone" is selected. You should check its volume settings so that when you speak, the level indicator remains in the green area.
- Press the `esc` (Escape) key, keep it pressed, and after about a second, say "Get my mail" naturally.
- The mail application should launch (if not already running), or you should be switched to it (if already running).
- Similarly, with `esc` pressed, say "Switch to Safari". You should either be switched to Safari, or it should launch if not already running.

- If you want to add a command called, say, "New window" to Safari (that opens a new browser window, the equivalent of `cmd-N`, you can do so by saying "Define new keyboard command".
- In the "Define keyboard command" sheet that launches, type in `cmd` and `N` (the text field should say `Command+N`). Click on the OK button.
- In the "Define keyboard command" window, type in the phrase "New window" in the phrase text field. You can choose to keep the command limited to Safari only, or be allowed in any application. Click on "Save".
- With `esc` pressed, say "Switch to Safari", followed by "New window". It should open a new Safari window.

As another example, you can play Chess by speaking your moves.

The speech interfaces also add to the accessibility features of Mac OS X. In addition to speech recognition and synthesis, Mac OS X offers visual assistance (zoom features, enhanced contrast, grayscale display), aural assistance (screen flashing), typing assistance (sticky keys, slow keys), and mouse assistance (mouse keys, full keyboard access).

Miscellaneous

Mac OS X 10.3 (Panther) introduced a number of new features. While some of these might be immature and even unstable, they should prove to be useful over time. Some of these are: **iChat AV** (video, audio, and text chat), **FileVault** (encryption of a user's home directory), **Exposé**, Secure Trash Deletion, etc.

> While Panther does securely delete files via a multi-pass overwriting algorithm courtesy the `srm` utility, Mac OS X is not a trusted system (it's not intended to be either). If a "secure" file's contents are in the buffer cache, or an application's address space, and if some of it has been swapped out, the information is vulnerable. You should be able to shut the machine down, connect it to another computer, say using target disk mode, and see "interesting" information through a simple `strings` on the device. OpenBSD, for instance, supports encryption of virtual memory (swap).

Exposé is a feature of the user interface that, among other things, lets you *shrink* each window on the screen *in-place* to a point where no windows overlap on the screen. This is different from automatic window tiling, because the original positions of the windows are *not disturbed*. You activate the in-place shrinking via a key press: F9 arranges all windows on the screen so that they don't overlap. You can "find" the window you are looking for visually - although the windows are shrunk, they are the same as the original windows - movies keep playing, etc. Once you identify the window you are seeking, you can choose it, either through the keyboard or the mouse, and the windows grow back to their original size, in their original positions, with the selected window on top. Exposé has a few other similar features, and Apple might offer a desktop pager based on it soon.

Mac OS X also has an implementation of Zero Configuration Networking, which Apple calls Rendezvous.

Available Software on Mac OS X

One relatively common notion about Mac OS X seems to be that there's not a lot of software for it. While it *is* true that the *quantity* of software available for Mac OS X is not as large as, say, that on Windows or Linux, I believe *the software that does exist for Mac OS X provides you as much functionality, if not more, as software on any other platform.*

Mac OS X has constantly been improving its support of "alien" APIs and frameworks. The introduction of a native X Window System is an example (although there *were* non-Apple ports of XFree86). Projects like Fink (see below) have helped a great deal in facilitating easy availability of open source software on Mac OS X. In my opinion, at this point, the concern that Mac OS X doesn't have 191,700 packages/ libraries available is largely unwarranted, as long as existing software's functionality matches that of other platforms.

A glaring exception is games (see below), where Mac OS X is lacking in comparison to Windows.

Bundled Software

Apple includes a phenomenal amount (and quality) of software with Mac OS X. You can refer to the Mac OS X Technical Specs for details of included software.

It's not just the quantity, or even quality, of software that makes Mac OS

X a highly productive and usable platform: the entire system has a uniform look and feel, applications are very nicely integrated with each other as necessary, and things *mostly* happen as you would expect.

Moreover, Apple has tried to extensively leverage a large amount of existing open source software, as is evidenced by the numerous open source packages that constitute the Darwin technology umbrella.

Open Source Software

Bundled

As stated in *Architecture of Mac OS X*, Darwin critically relies on a lot of open source software for functionality and features. Apache, bind, binutils, cvs, gcc, gdb, pam, perl, postfix, python, rsync, samba, etc. are only *some* of such packages. Moreover, Apple has integrated many of these software pieces with their platform, and has provided user interfaces that try to make configuration and use of such software easy and seamless.

Fink

The Fink project makes available a large set of Unix open source software on Mac OS X. The `fink` tool (written in Perl) uses Debian package management tools (`dpkg`, `dselect`, and `apt-get`). If you need, for example, the text-based web browser `lynx`, you can always download the source and compile it, which may work "out-of-the-box" for many packages, but not for many others. Fink takes care of such porting issues, dependencies, version updates, etc. For example:

```
% sudo apt-get install lynx
Password:
Reading Package Lists... Done
Building Dependency Tree... Done
The following NEW packages will be installed:
  lynx
0 packages upgraded, 1 newly installed, 0 to remove
and 0  not upgraded.
Need to get 1329kB of archives. After unpacking 0B
```

```
will be used.
Get:1 http://us.dl.sourceforge.net 10.3/release/main
lynx 2.8.4-12 [1329kB]
Fetched 1329kB in 5s (221kB/s)
Selecting previously deselected package lynx.
(Reading database ... 7563 files and directories
currently installed.)
Unpacking lynx (from .../lynx_2.8.4-12_darwin-powerpc.
deb) ...
Setting up lynx (2.8.4-12) ...
```

You can use the `fink list` command to see the list of packages currently supported by `fink`. Version 0.6.2 lists over 1100 packages (including various versions of certain packages)! Moreover, you can choose to download and install pre-compiled (binary) packages, or download the source (including Mac OS X specific patches, if any) and compile it. Heck, it's easier to get certain Open Source packages up-and-running via `fink` than, say, on Red Hat Linux (`xv`, for example).

Finally, a very nice approach taken by `fink` is that it does *not* pollute the rest of the system - it installs *everything* under the `/sw` directory.

Other Port Collections

Darwin Ports and GNU-Darwin are also open source port collections for Darwin/Mac OS X.

Random Open Source Software

There is plenty of open source software (possibly arcane or esoteric, but not always) that's not provided by `fink`. It should be easy to compile it from source, provided it's reasonably standard *nix source. There are some provisions to make portability easier: the `dlcompat` library (a non-Apple library providing a subset of the `dlopen/dl*` API), System V IPC support, `<poll.h>`, `<sys/aio.h>`, etc.

OpenOffice

OpenOffice (X11 based, not native) is available for Mac OS X. KOffice is also an option.

Shareware

The Macintosh platform has historically had an active "shareware" community. You can browse a large number of shareware software packages available for Mac OS X on web sites such as VersionTracker, MacUpdate, osx.hyperjeff.net/Apps etc.

> VersionTracker "tracks" not just shareware, but also "free" and commercial software. It also tracks software for other platforms (Mac OS 9, Windows, and Palm OS).

Commercial Software

A lot of commercial software is available for Mac OS X. If you need it, and if you can afford it, you will find plenty of applications to suit a variety of needs. In fact, the kind of commercial software that is available is one key factor that makes Mac OS X *the* most unique operating system currently.

A Sampling of Available Software

This section lists some examples of important, interesting, or commonly used software available for Mac OS X. This is not a complete list by any means, and there are many more "important" applications that I have not listed. I have categorized the software according to its domain of application, rather than as commercial, shareware, freeware, open source, etc.

For a much more comprehensive list of available software for the Mac, please refer to Apple's Software Guide.

Development

Apple has tried hard to make Mac OS X an excellent platform for various kinds of software development. Apple's developer tools are very good, and you also have other options, such as CodeWarrior). What's more, a large number (if not most) of open source compilers, interpreters, tools, and libraries should be usable on Mac OS X.

More details of programming options on Mac OS X were discussed in a previous section (*Programming on Mac OS X*).

Graphics, Media, and Publishing Software

- Adobe After Effects
- Adobe Acrobat Professional
- Adobe GoLive
- Adobe Illustrator
- Adobe InDesign
- Adobe Photoshop
- Adobe Premiere (discontinued)
- Alias Products (ImageStudio, Maya, PortfolioWall, SketchBook Pro, Studio Tools, ...)
- Apple DVD Studio Pro
- Apple Final Cut Express
- Apple Final Cut Pro
- Apple Logic Platinum
- Apple QuickTime Pro
- Apple Shake
- Apple Soundtrack
- NewTek Products (LightWave 3D, VT[3], ...)
- Macromedia Studio MX
- Pixar RenderMan
- QuarkXPress
- Roxio Toast 6 Titanium

Instant Messaging Clients

- Apple iChat AV
- Gaim
- Microsoft MSN Messenger

- Yahoo! Messenger

A number of other open source and/or shareware messenger clients (including multiple-protocol clients and IRC clients) are available. You can visit the VersionTracker site for more details.

Mail Clients

- Apple Mail.app
- IBM Lotus Notes
- Microsoft Entourage X
- Microsoft Outlook Express (OS 9/Classic)
- Mozilla Thunderbird
- Qualcomm Eudora

Media Players

- Apple QuickTime Player
- Microsoft Windows Media Player
- Real Player
- VLC Player

Networking/Connectivity Software

- Apple Remote Desktop
- Microsoft MSN
- Microsoft Remote Desktop Connection
- Nortel Contivity VPN Client

Office, Productivity, and Information Management Software

- Apple iSync
- Apple Keynote
- AppleWorks

- IBM Lotus Notes
- Microsoft Office v.X
- OpenOffice (X11 based, not native)

Sherlock and Watson

Sherlock (or its 3rd party counterpart, Watson) is a remarkably useful tool that you have to use to appreciate. The idea is to present useful information (movies, phone book, TV listings, recipes, dictionary, exchange rates, stock quotes, weather, translation, maps, travel, and many more) without using a web browser, thereby avoiding unnecessary navigation and clutter, while rendering and formatting the information in an appropriate way.

Sherlock is included with Mac OS X, while Watson costs $29.

Tax/Accounting Software

- Intuit QuickBooks Pro
- Intuit Quicken
- Intuit TurboTax

Scientific Software

- Wolfram Mathematica

x86 Emulators

- Bochs
- Microsoft Virtual PC

Bochs is a free, cross-platform, open source x86 emulator. It is extremely flexible and neat, but it is not integrated with or optimized for the Mac, and consequently is rather slow.

Virtual PC is a very useful application that emulates a PC on the Mac. Currently it is the fastest and the most comprehensive x86 emulator for Mac OS X. You can refer to *Many Systems on a PowerBook* for

examples of Virtual PC use.

Web Browsers

Mac OS X includes two web browsers by default: Safari and Internet Explorer (optional in Panther). A number of other web browsers are available for it though (open source and otherwise, free, shareware or commercial, etc.). Here is one list (not necessarily exhaustive):

- Amaya
- Camino
- Firebird
- iCab
- Internet Explorer
- Konqueror
- links (text based)
- lynx (text based)
- Mozilla
- Netscape
- OmniWeb
- Opera
- Safari

If you are feeling nostalgic, you can even get a copy of NCSA Mosaic, both as a native Mac OS 9 application (runs under Classic on OS X), or as an X11-based source archive that you can compile from.

What About Games?

I am very ill qualified to comment in this area, having played computer games only on game consoles all my life. From what I hear though, the Mac is (relatively) a far worse candidate than a PC as a gaming platform - not so much because of technical reasons (anymore), but simply because there are not as many games for the Mac.

This is not to say that there are no games at all! I think there are a few, like Breakout, Super Breakout, ...

Apple's web site has a section on available games for Mac OS X.

Conclusion: Why Mac OS X?

> The previous sections have been a commentary on Mac OS X, mostly in a "statement of fact" fashion. The closing discussion on this page assumes familiarity with the contents of the previous pages.

Bottom line

I have positive feelings about Mac OS X. If I were to choose *a* computing platform today to run my "digital life", it would be Mac OS X, unless my line of work or specific needs require/dictate otherwise.

This doesn't mean what works for me would work for others. This also doesn't mean Mac OS X has no "issues". It has many: it carries a lot of "old" technology and baggage, it is not bleeding edge in many respects, several things are ugly below Apple's pretty interface, it costs money, it needs Apple hardware (although I personally like Apple's machines very much), and it even has kernel panics on its own once in a while. Still, Mac is currently the most worthy client platform in my opinion: the only operating system currently in production that, within reason, *lets you have your cake and eat it too*.

In Search of an Ideal Client Computer

Of the operating systems I have used on my *primary* workstations (for extended periods of time), I believe currently three are "in the running" for client computing: the Linux family, Mac OS X, and the Windows family (in alphabetical order).

> Note that we use the phrase "client computing" or "client platform" to mean the same as, or similar to, "desktop computing", which is somewhat of a misnomer since many people use notebook computers in place of desktops. In any case, the phrase is meant to represent a multi-purpose, multi-faceted, everyday, *personal* machine, that you can use for work *and* play.

*BSD and Solaris

The various BSDs are well suited for servers and embedded applications. They are great if you want to follow a pedantic approach to learning about operating systems, and might even be ideal primary machines for people with either specific or limited needs, but I don't think any of the BSDs are going to be the client computers of tomorrow by themselves (of course, they might very well be the critical *underlying technology* of tomorrow's clients, as evidenced by the presence of FreeBSD in Mac OS X). I have great affinity for Solaris, and I think it still looks very promising (as a server) despite Sun's recent fortunes.

Let us reflect on the three client computing "candidates" briefly.

> Note that these operating systems have server versions too, but we are *only* talking about the client versions.

Windows

The existing "numbers" are ridiculously lopsided in favor of Windows. I have had a more or less neutral attitude towards Windows: I sometimes feel claustrophobic while using Windows, but I do not despise the platform (I think some people *do* have rather strong emotions against Windows). I have usually ascribed my "discomfort" with Windows to my background: my first operating system was SVR4, followed by Linux and *BSD. Windows alienates me from my familiar environments, and has several nuances of its own. I have tried various *nix-like environments for Windows over the years: the MKS toolkit for DOS, Cygwin, MKS for Win32, Microsoft Services for Unix, etc. They all try, admirably, but it's not the same. I admit that the lack of a "true Unix environment" isn't much justification against Windows. In fact, it's a fine platform in many respects - if you have *seriously* developed on multiple platforms, including Windows, you might agree with this.

I think there is too much entropy on Windows. The operating system is too "busy" (in the colloquial sense). Legacy baggage and backward-compatibility contribute to software malice and malfunction, whether it be doctored or accidental. Gigantic user base amplifies flaws (particularly security related) to epidemic proportions. Many experts disagree with numerous aspects of the Windows UI. A number of small "everyday" annoyances cumulatively make life difficult. Consider some examples: the entire system gets unresponsive when you empty a very full recycle bin, or copy large files, IE gets unresponsive when you

empty its cache, even though XP boots very quickly to the point of showing the desktop, it takes a long while before the system is usable - and this is *without* having anything in "Startup". Do you know how many keys there typically are in the Registry - do you know how many Office alone has, and what this does to maintainability? Have you ever installed Windows and spent hours, if not days, downloading and installing drivers specific to your machine (such as for Sony VAIOs, IBM ThinkPads, ...)? Have you ever had stability issues with Windows?

There are even books dedicated to Windows annoyances.

> The default XP theme's color-scheme seems to be Disneyland inspired - not that there's anything wrong with Disneyland. Many people like it, actually.

That said, Windows *can* be a fun platform, even if you are a hacker (in the good connotation of the word). Consider: you can get all the nice developer tools, debuggers like SoftICE, figure out lots of undocumented APIs, overtake some system calls, extend the system, find and fix security flaws, and so on. The fact that it has a huge market share might just make your efforts worthwhile. However, since Windows is largely disjoint with the *nix Universe, many (traditional) hackers steer clear of it for the fear of being "left out". Source code unavailability doesn't help either.

Nevertheless, an overwhelmingly large majority of computer users *are* on Windows. A *lot* of software is available for Windows as a result - it's a feedback loop perhaps, although for the typical end-user, the amount of important *Windows-only* software is probably decreasing. Still, many people don't have a choice, often because they don't *know* that they have a choice. There is a lot of inertia in moving away from Windows, both real (and justifiable) and imaginary. Often, there are existing investments in Windows: monetary, intellectual, legal, political, etc. Most importantly, a *lot* many people simply don't care - they have no motivation, or interest, in "experimentation". They have better things to do in life than evaluate computing platforms and switch. Many people don't even *like* computers - they just *have* to use them.

Microsoft could address some or most of the existing problems with Windows in future versions. They do seem to have the opportunity and the resources. I opine that they did several nice things with Windows 2000. Windows has a lot of things in place (including the user base). Microsoft has a lot of technology (theirs, licensed from 3rd parties, and even open source) to draw from. Thus, even though they probably *will* be the *de facto* client platform of the future, they might actually make

Windows worthy of being so.

Currently Mac OS X is, for lack of a more appropriate word, *better* in my opinion: for those who have a choice, and can make it.

Linux

I have great fondness for Linux. In fact, I feel much indebted to the GNU, Linux, and BSD communities for providing me (and others) with excellent sources of usable, no strings attached knowledge all these years. I have made heavy use (sometimes exclusive use) of Linux in academics and my jobs, including my current job. My extreme interest in Linux is one reason why I am somewhat disappointed in the way Linux as a client platform has evolved.

I think Linux is an excellent, if not the best, operating system for a variety of uses: in server applications, all kinds of embedded applications, in research and academia, for technology companies looking for an operating system as a base to develop their offerings; almost every domain *except* mainstream client computing.

Linux (as a client operating system), like Windows, has a lot of entropy, although of a different kind.

Choice, Choice Everywhere

Linux provides an incredible amount of choices - for everything. You can choose your distribution (and/or flavor) from an insane number of them, although the mainstream ones are not that many (I'm lying: *even they are too many*, and they all differ, sometimes subtly, sometimes vastly). You can run Linux on an even more incredible array of platforms - some of which one would never imagine could run *anything*, let alone an operating system. You can put Linux on watches and PDAs, or make clusters using it cheaply and efficiently. There are all kinds of devices that run Linux. You can install Linux in seemingly infinite number of ways. You can even compile the entire system from source (Gentoo Linux) - optimized for your particular machine to extract the last ounce of speed. You can have more number of *different* filesystems on Linux than you had total *files* on some older operating systems. You can have a similar number of packet classifiers in the kernel, and far too many other entities to be mentioned here. You can have support for devices you never knew existed, for protocols nobody even uses, and of course, for those that everybody uses. You can have almost all the window managers ever invented for X11. You can program in a zillion programming languages on Linux. You can choose from among more than one desktop environments, based on your personal preferences,

taste, working style, religion, or whatever else. You can have APIs from myriad systems and environments on Linux, either natively, or retrofitted/emulated. There's even STREAMS for Linux. Countless organizations, people, software, ... have benefited from Linux (including Mac OS X).

None of the above is exaggeration - I only used some minor figures of speech. Rather than go on and on haphazardly with this, I think that Linux essentially represents the union of a lot of, if not most, operating system (and related) technology ever created. I think it's a marvel that such a thing could be created, *has* been created, and is available *for free!*

> Note that we have used the term "Linux" in the above excerpt in its *misnomered* sense: to mean a Linux kernel based operating system (in other words, a Linux kernel and a lot of accompanying software). There is so much software (applications, libraries, documentation) available for Linux (mostly for free, no strings attached) that it's not even funny.

Even with all these great things, Linux is lacking, perhaps seriously so, when it comes to the specific case of a client platform. It is not so much because of technology deficiencies as it is because of the fact that despite many (some of them very well-meaning and noble) attempts by several entities (people, organizations, businesses, ...) to create operating systems around Linux, a cohesive enough system hasn't resulted.

If you intend to develop a client computing application for Linux, life is *not* easy for you. This sounds counter-intuitive to some. Life is supposed to be *great* if you are a developer on Linux. You have the source code for everything. You have countless eyes looking at all this source, finding problems, improving performance, and so on. There are libraries to do *everything*: the most glorious "parts-bin" that there can be. You can choose from numerous toolkits and even multiple desktop environments, each of which provides good, sometimes excellent, APIs.

Do you see a problem?

All this is great if you are a student, a researcher, a hobbyist, a company trying to use Linux to come up with something they can sell, or anybody trying to morph Linux into a very custom operating system for arbitrary use, etc.

It is *not* great if your domain of development is client computing. Say, you are a developer (individual, or part of a team, etc.) working on, for

example, a professional quality movie editing software for Linux (yes, I *have* looked at Kino - it's a great effort). How long do you think it would take (compared to other platforms)? Which toolkit(s) would you use? Which desktop environment would you integrate it with? Which distribution would you target it for? How do you know what you "like" is "right"? Which all libraries would you leverage so that you don't reinvent the wheel? How would you ensure it works seamlessly on a random installation? Do you think there are system wide, distribution agnostic, desktop environment independent "frameworks" that expose various components of the system's functionality? Is it even clear what "the OS" is? Surely, there are numerous ways to do most things, so which would *you* choose? In this situation, too much choice isn't such a wonderful thing after all. Think about the feasibility and effort requirements of creating something professional level, or even something non-trivial consumer level.

In the context of client computing, Linux's greatest strengths are somewhat offset by this fragmentation and lack of consistent, *overall architecture*. There is no common "glue" tying the system together. Having a bleeding edge kernel with large number of pieces of wide-ranging technology clubbed together, even decently, in an operating system might make a hacker's dream come true, but you need a stronger *fabric* that runs through the system if you want to contend with Windows and Mac OS X.

Linux indeed gives you an incredible number of choices, except the most important choice: suppose I do *not* want to think of, worry about, or handle *any* of this. Suppose I want everything that I need to be there, and to "just work". Suppose I wish to be oblivious of everything that's "under the hood" (maybe I am technically ill-equipped, maybe I do not have the time or inclination, maybe I want to use the computer as a tool so I can create other things, rather than be my own system administrator or software harvester, ...) - would Linux as a client platform "handle it" for me? No. Not today. Not yet.

I think sometimes the "choice" argument is misused. Consider this: how many people write their own disk drivers on Linux because they have the choice to do so? Very few, if any, because it needs uncommon expertise, it's too low-level for most programmers, it's not necessary because the default ones are good enough, etc. Well, for client computing, the same logic applies to a lot more things than disk drivers. A lot of paying customers probably find *everything* to be as low-level as a disk driver. If one wants to sell *them* Linux, one can't tell them to read a recipe of black-magic look alike instructions to do something, go find software on Freshmeat, or read Word documents with `strings` if OpenOffice failed to read them ...

There have been several steps in the right direction, though the efforts are often not collaborative, thereby not synergistic. People have tried to standardize things, from rather lower-level standards such as the LSB, to entire human interface guidelines (GNOME, KDE, ...) A historically popular remedy has been to create a new Linux distribution.

It is extremely encouraging to see the amount of attention Linux based operating systems have gotten, even from a client computing perspective. Governments and businesses seem to be very interested in Linux, for cost reasons, as a reaction against Microsoft's monopoly, and other political reasons. Linux is very popular in emerging countries. While this does give Linux headway, it's no guarantee for success. I believe the "concerns" mentioned above are still valid (and they are not the only obstacles Linux is facing - there are numerous other issues that are beyond the scope of this discussion). Somebody (a company, perhaps) needs to stand behind Linux in a *different* way (rather than *only* coming up with *yet another* "distribution"). I would like to see somebody take a different approach with Linux - maybe even something like what Apple did while coming up with Mac OS X.

I have stopped using Linux on my primary computer (in favor of Mac OS X). I work on Linux at my work, and run it under Virtual PC at home. In general, it is possible to do most things that one does on Linux on Mac OS X, usually with less effort.

As a digression, I can't help but point out that I believe there exists a psychological trap, wherein one spends a lot of time, over and over, tweaking, configuring, making things work, on an installation, while being under the impression that it's a worthwhile activity ... [more] >>>

Mac OS X

It hasn't been a very long time since I got my first Apple computer (April fool's day in the year 2003, complete with drama), but I agree with Apple's general direction with Mac OS X. I have tried to present a view of the current state of Mac OS X in the previous pages, so that you can judge for yourself if it suits your needs.

Mac OS X has managed to become rather Utopian with time. If you care, it doesn't alienate you from *nix. In fact, it gives you a very colorful environment to play with: you have Mach, FreeBSD, a nice driver development environment, and a lot of the system's source code to go with it.

If you are used to *nix (including Linux), you can get a lot of the same, or similar software on Mac OS X. You can compile it yourself, or hope

to find it in one of the port collections.

As discussed in *Available Software on Mac OS X*, there is a lot of professional quality (mostly commercial) software available for the Mac. In its current form, Mac OS X is uniquely positioned, unlike any other system, to offer the benefits of the *nix Universe as well as counterparts (either the same or with similar functionality) of important software from the Windows Universe. Apple have been busy adding to Mac OS X a lot of valuable software they created in-house.

Apple's traditional emphasis on ease-of-use seems to work most of the time, although there are exceptions. In my opinion, Mac OS X is representative of a "best-effort" approach - Apple took technology they had collected over the years, along with technology that had flourished in the open source world, and put together a reasonable system.

Currently there is nothing that I want to be able to run (as per my needs and wants) that does not run on Mac OS X.

Mandatory Apple Hardware

The biggest attached string with Mac OS X is the mandatory use of Apple hardware required. Some people get turned off by this, and *"Apple should release OS X for x86"* is sickeningly clichéd. People even offer "innovative" solutions to Apple's dilemma by saying that Apple could choose to support a fixed set of x86 hardware (so as to maintain the predictability and robustness that OS X currently has). Well, Apple should have no problem running OS X on x86 hardware, if that's what they want. NEXTSTEP and OpenStep ran on x86 hardware, and so does Darwin. It is rumored that Apple even has internal x86 builds of Mac OS X.

Apple apparently doesn't want to do that. This issue has been discussed to death time and again by all kinds of experts and analysts. I'm neither, so I'd just leave it at that.

As an aside, why wouldn't you use Apple hardware? It could be that you cannot pay for Apple hardware because you don't have the means, or you don't think hardware should cost as much. It could be that you hate Apple for making you necessarily buy their machines in order to use Mac OS X. Whatever the reason might be, it's your decision, and your beliefs. If you look at current Apple computers carefully, you'll see that a *lot* of engineering goes into making them. They are not just "pretty"

and stylish (though that might be subjective) - they are functional and well designed. Some people even call them lust worthy.

> If you've never, take a look at the inside of a G5 PowerMac. Do an `ioreg -l -w 0` on that machine. There are nine fans and twenty one temperature sensors on it. It's amazingly quiet.

The bottom line is that Apple wants a "Mac with Mac OS X" to be a package deal - in my opinion it's not such a bad deal. As a consumer, you can always pass it up!

The 1-Button Mouse

This is the *Traveling Salesman Problem* of the Mac Universe. Apparently (as I have learnt over time), all existing "problems" with Apple and Mac OS X are reducible to this. I have had people tell me in all seriousness the following: *"Apple cannot even make a mouse with more than one button. They are losers. Mac OS X sucks."* I don't have a good answer, or any answer for that matter, to this "issue". There are plenty of 3rd party mice available. Since Apple is adopting technologies such as X11 (which make heavy use of multiple buttons), they might as well just offer one themselves.

At the same time, I suspect some of the revulsion against a one-button mouse might just be anticipatory.

A Psychological Trap?

This is a phenomenon I have experienced, and I believe this applies to many others as well. Consider the typical Linux experience of a Linux *aficionado*, although this is not Linux specific. I can speak for myself. Everything on Linux, including the "problems", was a pleasure. *Particularly* the problems. I used to enjoy the installation. I would go through each package, reading its description and dependencies, selecting or de-selecting it, etc. I would carefully partition my disk, arranging things such that a minimal amount of space was "read-write". Although distributions have become much better in supporting various hardware out-of-the-box, there are always things that do

not work without proactive effort on your part, which is fun and feels like an accomplishment. I maintained bleeding edge versions of various packages under `/usr/local`. A few years ago, if somebody complained about lack of software on Linux, my own response would have been: "*If you need something, look if it's already in your distribution; otherwise look for it on the Net; if it doesn't exist, then write it yourself ...*" Well, Richard Stallman did it, but I don't think that can be a general prescription.

The "trap" is to get sucked into this cycle of tweaking, configuration, tinkering, hacking, figuring out how to make things work, which files go with which daemon or application, ... and be happy about it. While such pursuits are noble and academic in their own right, the sense of accomplishment you get from doing this *could* be misleading - even *false*! There is a very thin line between doing something worthwhile (whether it be for yourself or others), or just believe that you are, but it is possible that your time, talent, and creativity could be better used elsewhere.

Consider an example. I was with a friend once who is a Linux convert, but doesn't have much experience on Linux. His attempts to plug-in his USB digital camera to the Linux machine were failing, because the `usb-storage` module did not recognize his camera. Since he asked me for help, I looked at the error messages, at the files under `drivers/usb/storage` in an online Linux kernel tree, and found that `unusual_devs.h` did not have an entry for his camera. He did not have kernel compilation support installed, I didn't want to do more work than necessary, so we just patched `usb-storage.o` and replaced another model's entry with his camera. My friend was impressed beyond belief, not by me, but by Linux! He noted down the steps we took, and is probably patching the module with every kernel update, unless his model is in there already. This is a typical example of the Linux approach, which is why so many people like the platform. My contention is that the sense of machismo you get out of such things can lose its charm after a while, if you can find better things to do with your time.

Finally, I understand that it could be that you *need* to do repeated tweaking and configuration because that's your job - you could be a system administrator. I was one myself for many years while I was an undergraduate student. Even in that case, you would want to optimize your efforts - prevent your work

from becoming dull and dreary - invent ways to automate tasks - in which case it's not a waste of time because you are creating something that's useful. System administration could be a chore or an art - depending on how you do it.

Mac OS X Hacking Tools

Hacking? Tool?

The Jargon File is a popular lexicographic resource amongst hackers (and non-hackers too). Although it might have some subjective definitions I may not agree with, I have conveniently quoted verbatim the definitions of the terms "hacker" and "tool" as a preface to the contents of this page.

hacker

[originally, someone who makes furniture with an axe]

1. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary. RFC1392, the *Internet Users' Glossary*, usefully amplifies this as: A person who delights in having an intimate understanding of the internal workings of a system, computers and computer networks in particular.

more>>>

tool

1. n.A program used primarily to create, manipulate, modify, or analyze other programs, such as a compiler or an editor or a cross-referencing program. Oppose *app*, *operating system*; see also *toolchain*.

more>>>

So?

It is eminently debatable whether one (that means me, you or whosoever else) is a "hacker", but such a debate would probably be fruitless anyway, even meaningless. I do enjoy exploring the details of all sorts of *things*, including operating systems. This page is a compendium of some programs you might come across while tinkering with Mac OS X. Documentation for most of these tools exists, therefore my aim is not to reproduce documentation, but simply to maintain a cache of relevant information. I believe this would be useful to those who are new to Mac OS X, but are interested in exploring the system at a low(er) level. Note that many of the tools listed here are ones that are either new to Mac OS X (as compared to Unix style systems), or are different from their Unix counterparts. In other words, I have avoided listing "standard" Unix/BSD tools. Moreover, do realize that some (like `dynamic_pager` and various daemons) are *not really tools*.

The following list has *not* been fully updated for Panther (10.3.x).

Tools

KernelEventAgent

`/usr/sbin/KernelEventAgent` handles one of the core system services (events such as file systems being mounted and unmounted, low disk space, network connections going down, etc.)

SystemStarter

`/sbin/SystemStarter` is run during system initialization to handle "startup items". See "Mac OS X System Startup" for details.

aexml

`/usr/sbin/aexml` forwards XMLRPC and SOAP requests to the AppleEvent manager for further dispatching. More documentation is available on the Apple Developer Web Site.

appleping

`/usr/bin/appleping` exercises the AppleTalk network by sending packets to a named host.

ardbgd

`/usr/sbin/ardbgd` is the daemon for the Apple Remote Debugging Service.

asr

`/usr/sbin/asr` (Apple Software Restore) efficiently copies disk images and volumes, and can also accurately clone volumes.

bless

`/usr/sbin/bless` is used to set volume bootability characteristics for Macintoshes. The command can be used to select a folder on a mounted volume to act as the *blessed* system folder, and optionally update Open Firmware to boot from that volume. It can also be used to format and setup a volume for the first time. Finally, it can be used to query the folder(s) that are blessed on a volume. Try the following (non-destructive) commands:

```
% sudo bless -verbose -info /
...
% sudo bless -verbose -plist -info /
```

blued

`/usr/sbin/blued` is the Bluetooth daemon.

cac_*

`/usr/sbin/cac_*` are scripts related to CAC (Common Access Card) support. A CAC can be thought of as a SmartCard that combines multiple cards (functions) into one. A CAC can enable physical access to buildings and controlled places, enable computer network and system access and serve as the primary platform for the PKI token.

cmpdylib

`/usr/bin/cmpdylib` compares two dynamic shared libraries for compatibility.

createhomedir

`/usr/sbin/createhomedir` creates and populates local home directories.

ddb

`ddb` is a debugging mechanism that can be compiled into Mac OS X, similar to BSD's `kdb`. While `gdb` can be used over Ethernet (through a kernel stub), `ddb` is compiled into the kernel and is used over a serial line. Most importantly, `ddb` requires an actual *built-in hardware* serial line on the debug target. Fortunately, `gdb` should suffice for almost all debugging needs unless one is trying to debug an Ethernet driver itself, say.

`ddb` is not present by default on Mac OS X. It must be compiled from source (`xnu/osfmk/ddb` in the CVS tree).

defaults

`/usr/bin/defaults` is used to access (read, write and delete) Mac OS X user defaults from the command line. For example, the following will print out Desktop background settings (including the pathname for the desktop background image, if any):

```
% defaults read com.apple.desktop Background
```

dev_mkdb

`/usr/sbin/dev_mkdb` creates a hash access method database (based on Berkeley DB) in `/var/run/dev.db`. This database contains the name of all devices under `/dev`.

diskarbitrationd

`/usr/sbin/diskarbitrationd` is a daemon that listens for connections from clients, notifies clients of the appearance of disks and filesystems, and governs the mounting of filesystems and claiming of disks amongst clients.

disktool

`/usr/sbin/disktool` is a command line utility for disk arbitration. It can be used to rename, eject, mount or unmount disks and volumes.

diskutil

`/usr/sbin/diskutil` is a utility for managing disks and volumes. It can be used to perform operations such as enabling/disabling HFS+ journaling, verifying and repairing permissions, erasing disks (including optical media), partitioning, creating and managing RAID sets etc. You typically need root access to use this utility.

ditto

`/usr/bin/ditto` copies files and directories to a destination directory. `ditto` can be used to "thin" "fat" (multiple-architecture) exectuables. It can also copy files selectively based on the contents of a BOM ("Bill of Materials"). One of the most useful features of `ditto` is that it can preserve resource fork and HFS meta-data information when copying files.

drutil

`/usr/bin/drutil` is a command line tool that uses the DiscRecording framework to interact with attached CD/DVD burning devices.

dscl

`/usr/bin/dscl` is the Directory Service command line utility.

dsperfmonitor

`/usr/bin/dsperfmonitor` is a directory tool for testing plugin performance in Directory Services.

dynamic_pager

`/sbin/dynamic_pager` is started during system initialization to manage swap files. See Mac OS X System Startup for details.

fdisk

`/usr/sbin/fdisk` displays or changes the DOS partition table found in the bootsector of x86 bootable disks.

fixPrecomp

`/usr/bin/fixPrecomp` is a tool for "fixing" precompiled header warnings that occur when headers get out-of-sync with their precompiled versions - after a system update, say.

fixproc

`/usr/bin/fixproc` is a Perl script that "fixes" a named process by performing the specified action (which can be check, kill, restart, exist or fix).

fs_usage

`/usr/bin/fs_usage` presents an ongoing display of system call usage information pertaining to file system activity. By default this includes all system processes except the running `fs_usage` process, `Terminal`, `telnetd`, `sshd`, `rlogind`, `tcsh`, `csh` and `sh`.

fstat

`/usr/bin/fstat` identifies open files (including sockets).

heap

`/usr/bin/heap` lists all the malloc-allocated buffers in the specified process's heap.

hdiutil

`/usr/bin/hdiutil` uses the `DiskImages` framework to manipulate disk image files.

hlfsd

`/usr/sbin/hlfsd` is the home-link file system daemon. It implements a file system containing a symbolic link to a subdirectory within a user's home directory, depending on the user which accessed that link.

installer

`/usr/sbin/installer` is the Mac OS X system software and package installer tool.

install_name_tool

`/usr/bin/install_name_tool` changes the dynamic shared library install names recorded in a Mach-O binary.

ioalloccount

`/usr/sbin/ioalloccount` displays some accounting of memory allocated by `IOKit` allocators, including object instances, in the kernel. This is useful for tracking memory leaks.

ioclasscount

`/usr/sbin/ioclasscount` displays the instance count, offset by the number of direct subclasses that have at least one instance allocated, for the classes specified. This is useful for tracking leaks.

ioreg

`/usr/sbin/ioreg` displays the `IOKit` registry. Try `ioreg -l`, for example, and you can see detailed registry information (including object properties) - such as details of various temperature sensors in the system (on the I2C bus).

iostat

`/usr/sbin/iostat` displays kernel I/O statistics on terminal, disk and cpu operations.

ipconfig

`/usr/sbin/ipconfig` can be used to get the number of network interfaces active (the `ifcount` argument), and also to retrieve various options associated with these interfaces. For example, "`ipconfig getoption en1 lease_time`" prints the DHCP lease time of `en1` if applicable. Finally, `ipconfig` can also be used to set an interface for `BOOTP`, `DHCP` etc.

kdump

`/usr/bin/kdump` displays the kernel trace files produced with `ktrace` in human readable format.

kextcache

`/usr/sbin/kextcache` creates or updates `kext` caches, which are used to speed up kernel extension loading operations and to prepare `kexts` for inclusion in such media as device ROM.

kextload

`/sbin/kextload` can be used to explicitly load kernel extensions, validate them to see that they can be loaded by other mechanisms, such as `kextd`, and to generate symbol files for debugging the `kext` in a running kernel.

kextstat

`/usr/sbin/kextstat` displays the status of any kernel extensions currently loaded in the kernel.

kextunload

`/sbin/kextunload` is used to terminate and unregister `IOKit` objects associated with a kernel extension and to unload the code and personalities for that `kext`.

kgmon

`/usr/sbin/kgmon` generates a dump of the operating system's profile buffers for later analysis by `gprof`.

ktrace

`/usr/bin/ktrace` enables kernel trace logging for the specified processes, causing trace data to be logged to a file. Traced kernel operations include system calls, namei translations, signal processing and I/O.

latency

`/usr/bin/latency` is used for monitoring scheduling and interrupt latency. The tool can also be used to set real time or timeshare scheduling policies.

ld

`/usr/bin/ld` is the (Mach) object file link editor.

leaks

`/usr/bin/leaks` examines a specified process for malloc-allocated buffers which are not referenced by the program.

lipo

`/usr/bin/lipo` creates or operates on multi-architecture ("fat") files. It can list the architecture types in a fat file, create a single fat file from one or more input files, thin out a single fat file to a specified architecture type, and extract, replace and/or remove architecture types from the input file.

lockfile

`/usr/bin/lockfile` can be used to create one or more (conditional) semaphore files, with the provision of waiting for a specified number of seconds and a specified number of retries.

lsbom

`/usr/bin/lsbom` interprets the contents of binary `bom` (bill-of-materials) files. `bom` is a file system used by the Mac OS X installer to determine which files to install, remove, or upgrade.

lsof

`/usr/sbin/lsof` lists information about files opened by processes.

lsvfs

`/usr/bin/lsvfs` lists known (currently loaded) virtual file systems.

mDNSResponder

`/usr/sbin/mDNSResponder` (Multicast DNS Responder) listens

for and responds to DNS-format query packets sent via Multicast to UDP port 5353.

mach_init

`/sbin/mach_init` is a daemon that maintains various mappings between service names and the Mach ports that provide access to those services.

malloc_history

`/usr/bin/malloc_history` inspects a given process and lists the malloc allocations performed by it. It relies on information provided by the standard malloc library when debugging options have been turned on.

mig

`/usr/bin/mig` (Mach Interface Generator) is used to compile procedural interfaces to Mach's message-based APIs, based on descriptions of those APIs.

mkbom

`/usr/bin/mkbom` creates a `bom` (bill-of-materials) given a directory.

mkextunpack

`/usr/sbin/mkextunpack` extracts the contents of a multikext (mkext) archive.

netstat

`/usr/sbin/netstat` symbolically displays the contents of various network-related data structures.

nibindd

`/usr/sbin/nibindd` is a daemon that is responsible for finding, creating and destroying `NetInfo` servers.

nibtool

`/usr/bin/nibtool` is used for printing, verifying and updating `nib` files.

nicl

`/usr/bin/nicl` is a general-purpose utility for operating on `NetInfo` databases. Its commands allow one to create, read and manage `NetInfo` data.

nidomain

`/usr/sbin/nidomain` is an interface to `nibindd` to which it sends all of its requests about the domains served on a given machine. It can also be used to create and destroy `NetInfo` databases.

nifind

`/usr/bin/nifind` finds a directory in the `NetInfo` hierarchy.

nigrep

`/usr/bin/nigrep` searches for a regular expression in the `NetInfo` hierarchy.

niload

`/usr/bin/niload` loads information from standard input into the given `NetInfo` domain.

nireport

`/usr/bin/nireport` prints tables from the `NetInfo` hierarchy.

niutil

`/usr/bin/niutil` is used to do arbitrary reads and writes on the given `NetInfo` domain.

nmedit

`/usr/bin/nmedit` is used to change global symbols to local symbols. It differs from `strip` in that it also changes the symbolic debugging information for the global symbols it changes to static symbols so that the resulting object can still be used with a debugger.

notifyd

`/usr/sbin/notifyd` is a daemon that facilitates processes to exchange stateless notification events.

nvram

`/usr/sbin/nvram` allows manipulation of Open Firmware non-volatile RAM variables.

objcopy

`objcopy` is part of `binutils` that you can download, compile and install. This utility copies the contents of an object file to another, using the GNU BFD (Binary File Descriptor) library to access the object files.

objdump

`objdump` is part of `binutils`. It displays information (including disassembly, if required) about one or more object files.

open

`/usr/bin/open` is a command line utility to open a file (or a directory or URL), just as if you had double-clicked the file's icon.

open-x11

`/usr/bin/open-x11` is a wrapper shell script that provides `open` functionality for X11 applications.

orbd

`/usr/bin/orbd` is the Object Request Broker Daemon. It is a tool to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.

osacompile

`/usr/bin/osacompile` compiles the given files, or standard input if non are listed, into a single output script.

osalang

`/usr/bin/osalang` prints information about installed OSA (Open Script Architecture) languages.

osascript

`/usr/bin/osascript` executes the given script file, or standard input if none is given. Scripts may be plain text or compiled scripts.

otool

`/usr/bin/otool` displays specified parts of object files or libraries (similar to `ldd` on Linux).

pagestuff

`/usr/bin/pagestuff` displays information about the specified logical pages of a file conforming to the Mach-O executable format.

pax

`/bin/pax` is a tool for reading, writing, and listing members of an archive file. It is also used to copy directory hierarchies. `pax` supports various archive formats such as `cpio`, `bcpio`, `sv4cpio`, `sv4crc`, `tar`, and `ustar`.

pbcopy

`/usr/bin/pbcopy` is used to copy standard input to the pasteboard buffer.

pbpaste

`/usr/bin/pbpaste` prints the contents of the pasteboard buffer.

pcscd

`/usr/sbin/pcscd` is a daemon used to dynamically allocate/deallocate Smart Card reader drivers at runtime and manage connections to the readers. Related utilities include `/usr/bin/pcsctest` and `/usr/bin/pcsctool`. These tools are taken from the MUSCLE (Movement for the Use of Smart Cards in a Linux Environment) project, a project to coordinate the development of smart cards and applications under Linux.

pdisk

`/usr/sbin/pdisk` is a menu driven program which partitions disks using the standard Apple disk partitioning scheme.

plutil

`/usr/bin/plutil` can be used to check the syntax of property list files, or convert a plist file from one format to another.

pmset

`/usr/bin/pmset` changes and reads power management settings such as idle sleep timing, wake on administrative access, automatic restart on power loss, etc.

pstat

`/usr/sbin/pstat` displays open file entry, swap space utilization, terminal state, and vnode data structures.

redo_prebinding

`/usr/bin/redo_prebinding` is used to redo the prebinding of an executable or dynamic library when one of the dependent dynamic library changes. The input file, executable or dynamic library, must have initially been prebound for this program to redo the prebinding.

say

`/usr/bin/say` uses the Speech Synthesis manager to convert input text to audible speech and either play it through the sound output device chosen in System Preferences or save it to an AIFF file.

screencapture

`/usr/sbin/screencapture` captures the screen (a window selection or a mouse selection) to the clipboard or a file (as PDF).

scselect

`/usr/sbin/scselect` is used to change current network location, or to list defined locations.

sc_usage

`/usr/bin/sc_usage` displays an ongoing sample of system call and page fault usage statistics for a given process.

scutil

`/usr/sbin/scutil` is a tool to communicate with `configd`, read and write from/to the configuration data store etc.

security

`/usr/bin/security` provides a command line interface to administer Keychains, manipulate keys and certificates, and do most things the Security framework is capable of.

segedit

`/usr/bin/segedit` extracts and/or replaces the named sections from the specified input file and creates an output.

setregion

`/usr/bin/setregion` is the command line utility for setting the DVD drive's "region".

sips

`/usr/bin/sips` is a command line interface to the Scriptable Image Processing Server. The graphical abilities of Mac OS X are exposed through this image processing service. The SIPS architecture contains tools for performing basic image alterations and support various image formats. The goal is to provide quick, convenient, desktop automation of common image processing operations.

slpd

`/usr/sbin/slpd` is the Service Location Protocol daemon that

advertises local services to the network.

slp_reg

`/usr/sbin/slp_reg` is a tool to register URLs via the Service Location Protocol in order for remote machines to discover locally registered services.

softwareupdate

`/usr/sbin/softwareupdate` is a command line utility to perform software updates under Mac OS X.

srm

`/usr/bin/srm` securely (by overwriting, renaming, and truncating before unlinking) removes files or directories.

sw_vers

`/usr/bin/sw_vers` prints the product name (such as Mac OS X), version and build number.

sysctl

`/usr/sbin/sysctl` retrieves kernel state and allows processes with appropriate privilege to set kernel state.

system_profiler

`/usr/sbin/system_profiler` is the command line system profiling utility.

tcpdump

`/usr/sbin/tcpdump` dumps traffic on a network.

top

`/usr/bin/top` displays an ongoing sample of system usage statistics (such as cpu utilization, memory usage etc. for each process).

trpt

`/usr/sbin/trpt` interrogates the buffer of TCP trace records created when a socket is marked for debugging (via `setsockopt()`) and prints a readable description of these records.

update_prebinding

`/usr/bin/update_prebinding` tries to synchronize prebinding information for libraries and executables when new files are added to a system. Prebinding information is pre-calculated address information for libraries used by a given executable or library. By pre-determining where a function in another library is destined to be placed, the dynamic linker does not have to resolve symbols at application startup time.

vm_stat

`/usr/bin/vm_stat` displays Mach virtual memory statistics.

vmmap

`/usr/bin/vmmap` displays the virtual memory regions allocated in a specified process, indicating how memory is being used, and what the purposes of memory at a given address might be.

vpnd

`/usr/sbin/vpnd` is the Mac OS X VPN service daemon.

xcode*

`/usr/bin/xcode*` are Xcode related commands.

xxd

`/usr/bin/xxd` creates a hex dump of a given file or standard input. It can also convert a hex dump back to its original binary form.

Hacker

hacker

[originally, someone who makes furniture with an axe]

1. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary. RFC1392, the *Internet Users' Glossary*, usefully amplifies this as: A person who delights in having an intimate understanding of the internal workings of a system, computers and computer networks in particular.

2. One who programs enthusiastically (even obsessively) or who enjoys programming rather than just theorizing about programming.

3. A person capable of appreciating *hack value*.

4. A person who is good at programming quickly.

5. An expert at a particular program, or one who frequently does work using it or on it; as in ‘a Unix hacker’. (Definitions 1 through 5 are correlated, and people who fit them congregate.)

6. An expert or enthusiast of any kind. One might be an astronomy hacker, for example.

7. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations.

8. [deprecated] A malicious meddler who tries to discover sensitive information by poking around. Hence password hacker, network hacker. The correct term for this sense is *cracker*.

The term ‘hacker’ also tends to connote membership in the global community defined by the net (see *the network*. For discussion of some of the basics of this culture, see the How To Become A Hacker FAQ. It also implies that the person described is seen to subscribe to some version of the hacker ethic (see *hacker ethic*).

It is better to be described as a hacker by others than to describe oneself

that way. Hackers consider themselves something of an elite (a meritocracy based on ability), though one to which new members are gladly welcome. There is thus a certain ego satisfaction to be had in identifying yourself as a hacker (but if you claim to be one and are not, you'll quickly be labeled *bogus*). See also *geek*, *wannabee*.

This term seems to have been first adopted as a badge in the 1960s by the hacker culture surrounding TMRC and the MIT AI Lab. We have a report that it was used in a sense close to this entry's by teenage radio hams and electronics tinkerers in the mid-1950s.

Tool

tool

1. n.A program used primarily to create, manipulate, modify, or analyze other programs, such as a compiler or an editor or a cross-referencing program. Oppose *app*, *operating system*; see also *toolchain*.

2. [Unix] An application program with a simple, ‘transparent’ (typically text-stream) interface designed specifically to be used in programmed combination with other tools (see *filter*, *plumbing*).

3. [MIT: general to students there] vi. To work; to study (connotes tedium). The TMRC Dictionary defined this as “to set one's brain to the grindstone”. See *hack*.

4. n. [MIT] A student who studies too much and hacks too little. (MIT's student humor magazine rejoices in the name *Tool and Die*.)

`/proc` on Mac OS X

The "process" file system (`procfs` for brevity, or simply `/proc`, where it is usually mounted) has become relatively common on Unix and Unix-like systems (Linux, FreeBSD, Solaris ...). `procfs` is meant to be a file system representation of the process table, although some systems put so many features and functionality into `procfs` (Linux, for example) that renders such implementations nothing short of the proverbial kitchen-sink.

In its simplest form `procfs` provides file-system abstraction for processes, presenting information about processes, their execution environment, resource utilization etc. as files. The original design goal of `procfs` was to make debugger implementation easier. Systems like Linux have exposed a lot of other system interfaces and information through `procfs`. There are Linux system utilities that essentially `cat`

files in procfs! There is plenty of documentation for procfs on various systems that you can refer to.

Mac OS X does *not* have procfs. While it would be nice to have, I don't think it is a limitation. Things that could make use of procfs would have to be done differently, that's all: ps would have to be written differently from, say, on Linux. On Mac OS X, ps uses the kvm and sysctl interfaces. gdb, and various other programs also have such differences.

Sometimes it *is* better for the more inquisitive to have procfs. Consider a simple example: you could simply cat a file (/proc/cpuinfo) on Linux and glean a lot of information about the CPU. On Mac OS X, you can look at "About This Mac", or do a "sysctl -a hw" and try to look at relevant information in the output. In general, the Linux output is more detailed. There are instances where it is extremely hard (or even impossible) to look at certain information on Mac OS X: *Suppose you had a "discussion" with a colleague about the size of the disk drive buffer on a particular Mac. You look at the drive's model string in "About This Mac" and look at the specifications on the manufacturer's web site. I found that Hitachi had two different pages with conflicting information regarding the buffer size (2MB and 8MB) for* IC25N080ATMR04. The fastest way I could come up with was to boot Linux from a CD-ROM and look at files under /proc/ide (for each drive, various pieces of information are listed, including the buffer size). This may seem contrived. It is.

Even in light of the previous "example", I don't feel a compelling need for procfs on Mac OS X. It would be instructive to add such support to Mac OS X for "academic" reasons, though. The other night I was fooling around with xnu, and I thought about adding a quick and dirty procfs to it. Darwin 7.0's BSD components come from FreeBSD 5.0. FreeBSD does have procfs, although I realized later that it's not *that* straightforward a port. The rest of this document contains notes on porting procfs from FreeBSD to Darwin.

FreeBSD --> xnu/bsd

Given that code for the vfs layer and various system calls in xnu derived from FreeBSD, it would be reasonable to port procfs from FreeBSD. That said, there are enough differences between the two systems, as we shall see. The following notes describe steps taken to add a very simple (only /proc/<pid> directories visible with support for only cmdline and status).

pseudofs

FreeBSD contains `linprocfs`, a process file system that emulates (a subset of) Linux's `procfs`. `linprocfs` is needed for Linux binary emulation to work completely. This is in addition to FreeBSD's own `procfs`. In order to reduce (eliminate) code duplication between such pseudo file systems, `pseudofs` was added to FreeBSD. `pseudofs` is a framework (rather than a full-fledged file system) for implementing pseudo file systems on top of it. In FreeBSD 5.0, `procfs` is implemented using `pseudofs`, which means that in order to port the former, we need to port the latter first.

As an aside, note that that `xnu/bsd` does have in-memory file systems like `synthfs`, which is used to create arbitrary directory trees (if you wanted to synthesize mount points for random things, say). `synthfs` is *not* in FreeBSD 5.0. Similarly, `volfs`, the "volume file system" on OS X is a virtual file system that exists over the HFS VFS and serves the needs of two differing APIs (Unix pathnames and MacOS <Volume ID><Directory><File Name>). Other than these, FreeBSD and `xnu/bsd` share file systems (to various extents) such as `deadfs`, `devfs`, `fdescfs`, `fifofs`, `nullfs`, `specfs`, `unionfs` etc.

pseudofs.h

`pseudofs.h` needs to be modified with the following points in mind:

- `MFSNAMELEN` (length of file system name type) in `<sys/mount.h>` includes null under `xnu/bsd`, but not under FreeBSD.
- `xnu/bsd` uses "simple locks" in many places where FreeBSD uses mutexes.
- Where FreeBSD passes around "struct thread" pointers, `xnu/bsd` passes around "struct proc" pointers.
- `xnu/bsd` VFS does not have extended attribute support.

- `VFS_SET(...)`, `MODULE_VERSION(...)` and `MODULE_DEPEND(...)` macros are not applicable under `xnu/bsd`. This also means we would have to do something else (as opposed to calling functions when FreeBSD module load/unload events occur) to enable `pseudofs`.

`pseudofs_internal.h`

`pseudofs_internal.h` contains only structure definitions and function prototypes that can be used as is.

The rest of the source files need numerous changes, some of which are outlined below.

`pseudofs.c`

- <sys/module.h>, <sys/mutex.h> and <sys/sbuf.h> do not exist under `xnu/bsd`. We would not need `module.h` as we are not using loadable modules. We would not need `mutex.h` as we are using simple locks. We do need `sbuf.h`. Quoted from the man page: ... *the **sbuf** family of functions allows one to safely allocate, construct and release bounded null-terminated strings in kernel space. Instead of arrays of characters, these functions operate on structures called sbufs, defined in <sys/sbuf.h>*. Since sbufs are used as the basis of `pseudofs`, we simply include them in `xnu/bsd` for our "dirty" implementation.
- We copy over `sbuf.h` and `subr_sbuf.c` from FreeBSD to `xnu/bsd`. Since sbufs use `va_list`, the correct `stdarg.h` needs to be used. Moreover, macros such as `isdigit()` need to be pulled in. There is no memory type `M_SBUF` defined on `xnu/bsd`, of course.
- Functions that take a thread pointer would take a process pointer, as mentioned earlier.
- The `pfs_modevent()` function is not needed.
- Simple locks are used in place of mutexes.

pseudofs_fileno.c

The only changes are adjustments for `M_PFSFILENO` and mutexes.

pseudofs_vncache.c

- `M_PFSVNCACHE` is not defined. Make adjustments.
- Make adjustments for mutexes.
- `vop_t` is not defined. Define it.
- `rm_at_exit()` does not exist. For now, we leave it out! (`/* XXX */`).
- `vget()` and `vn_lock()` need `current_proc()` instead of `curthread`.
- `getnewvnode()` is called slightly differently. Rather than specify a string name for the file system as the first argument, we use a new `enum` (`VT_PROCFS`, say).
- Instead of `(*vpp)->v_vflag = VV_ROOT`, use `(*vpp)->v_flag = VROOT`. Moreover, `VV_PROCDEP` is not defined. Define it.

pseudofs_vnops.c

- Instead of <sys/ctype.h>, simply define the `isdigit()` macro.
- <sys/mutex.h> and <sys/sx.h> do not exist.
- Pull in <vfs/vfs_support.h>
- Functions that take a thread pointer would take a process pointer. Consequently, things like `td->td_proc->p_pid` would become `td->p_pid`, `curthread` would become `current_proc()`, `cn_thread` becomes `cn_proc`, etc.
- We provide a dummy implementation (to begin with) of `p_cansee()`.

- We provide an implementation of `vaccess()`.
- Instead of `vrefcnt(vn)`, we examine `vn->v_usecount` directly.
- Instead of "real" uid/gid, simply use uid/gid.
- We leave out `pfs_getextattr()`.
- We create simple implementations of `vop_defaultop()` and `vop_eopnotsupp()`.
- We declare and populate a `struct vnodeopv_entry_desc` array, with appropriate defaults.
- We leave out `VNODEOP_SET()` - we would perform the equivalent action elsewhere.

We copy over `procfs.c`, `procfs.h`, `procfs_note.c` and `procfs_status.c` from FreeBSD's `procfs/` directory.

`procfs.c`

- We declare `vfs_opv_numops` and `pfs_vnodeop_opv_desc` as externs.
- Instead of `vn_fullpath()`, simply use "unknown" for now.
- Get the pid from the process pointer directly.
- Leave out everything from the constructor except the root, `cmdline` and `curproc`.
- Declare a static pointer to `struct vfsconf`.
- Declare `#define`'s for `procfs_start`, `procfs_quotactl`, `procfs_sync`, `procfs_vget`, `procfs_fhtovp`, `procfs_vptofh` and `procfs_sysctl`, casting `eopnotsupp` appropriately.
- Declare a static `struct pfs_info` for `procfs`.
- Implement `_procfs_mount()` (simply call

`pfs_mount()`), `_procfs_init()` (simply call `pfs_init()`), `_procfs_uninit()` (simply call `pfs_uninit()`.

- Use the following `struct vfsops`:

```
struct vfsops procfs_vfsops = {
    _procfs_mount,
    procfs_start,
    pfs_unmount,
    pfs_root,
    procfs_quotactl,
    pfs_statfs,
    procfs_sync,
    procfs_vget,
    procfs_fhtovp,
    procfs_vptofh,
    _procfs_init,
    procfs_sysctl,
};
```

- Implement `procfs_kernel_mount()` along the lines of `devfs_kernel_mount()`.
- Initialize `pseudofs` at boot time from `bsd/kern/bsd_init.c`.
- Mount `procfs` on `/proc` at boot time from `bsd/kern/bsd_init.c`.

Status

I believe everything is in place, though there *is* a kernel panic soon after `/proc` is mounted :-) Before I could even *begin* over-the-network two-machine kernel debugging, one of my two Macs, an iBook, "died". As if that's not frustrating enough, here's what ensued (it's not a rant, but a mere statement of facts):

> The iBook was about 7 months old when it had its first problem: a *pinched* (by the screen hinge, apparently) display cable that garbled the display for random periods of time. I took the iBook for service (covered by the 1 year warranty).

The iBook was returned to me within a few days with the display fixed, but with a "dead" (not just unbootable, but uninstallable, unusable) disk drive. Needless to say, I had to take it back immediately.

The iBook was returned with a new drive, but random cosmetic damage, which I chose to ignore. Within a few weeks, its display went blank (apparently a "logic board failure"). It is at this point that I needed it for 2-machine debugging to chase the kernel panic.

The iBook was in service for almost three *weeks* this time. It was returned to me with a damaged (scratched) LCD that wasn't put back together properly either, as well as more cosmetic damage.

Exasperated, I took it back, and it was in service for almost two more weeks. They changed the LCD screen, and (I believe) even the casing (to make up for the cosmetic damage).

At this point, the iBook has a new casing, a new logic board, a new hard drive and a new LCD screen. It has been in service for at least 45 days in its first 9 months. However, I have not gotten around to looking at `/proc` since I got the iBook back.

Disclaimer: I do not know how typical an Apple service experience this is. My other Mac, a PowerBook 17, has not had any problems.

Re-routing System Calls

Why?

Re-routing (intercepting, modifying the arguments or results of, blocking, ...) system calls can be a powerful tool for doing "interesting" things with an operating system. It is more challenging (and therefore more fun, perhaps) on systems whose source is not available, such as Solaris, because the published APIs only tell you what goes in and what comes out - what happens within is not always known, or easy to find.

I once created a system call audit mechanism for Linux, and later on used similar techniques while creating a virtualized Solaris operating system. Darwin's source is available for most people to see, so it is

possible to study how its system calls are implemented. Moreover, most of the calls are derived from BSD, so those familiar with FreeBSD, for example, should find it familiar.

Example

The sample code provided below implements a very simple Mac OS X kernel extension that essentially does the following:

- Save a reference to the `open` system call via the `sysent[]` vector.
- Replace the entry for `open` in `sysent[]` to point to our function.
- Undo the above steps when the extension is unloaded.

The extension contains an implementation of `open` that copies the string representing the pathname of the file being opened from user space to kernel space, and prints a message containing the arguments passed to `open`. Thereafter, the function simply calls the original `open` through the saved reference.

Note that the messages printed by the kernel extension are logged in `/var/log/system.log`. All invocations of the `open` system call would result in a message being logged. Thus, you can simply do a `tail -f /var/log/system.log` and run something like `cat /etc/resolv.conf` to see the extension working.

Disclaimer

Please understand that loading unsupported kernel extensions into your operating system kernel is an inherently risky thing to do, and could result in a catastrophic data loss. The following code *is* unsupported. If you wish to download, compile, run or experiment with it, you must do so at **your own risk**.

Instruction Re-writing :: Function Re-routing

Consider the following common (well, maybe *not* so common) problem: *You have written a loadable kernel module (kernel extension, in Mac OS X parlance). There is some function* `foo()` *in the kernel that you would*

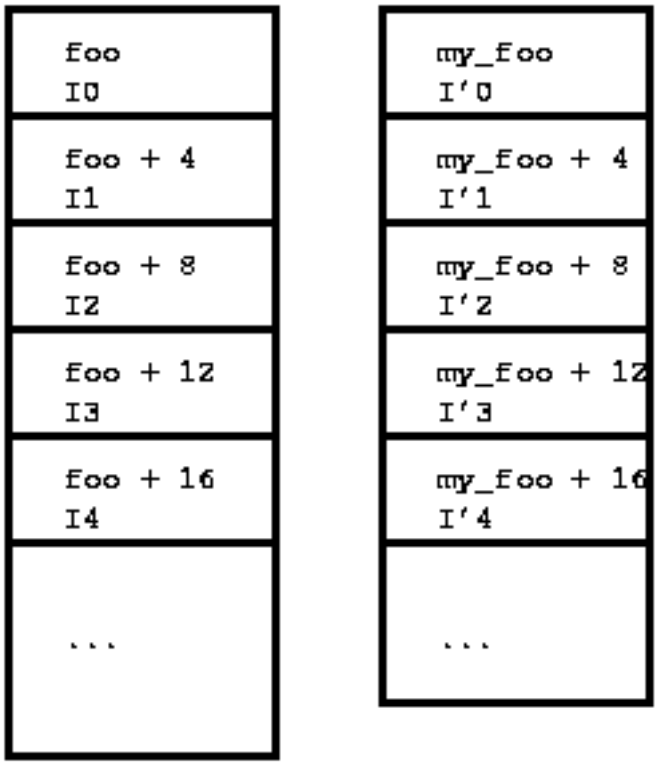
like to re-route, that is, replace with your own version called `my_foo()`. *What's more, you would like to be able to call the original* `foo()` *from* `my_foo()`.

There are several things to be observed here:

- You may not have access to the source code of the kernel.
- Even if you do have access to the source, you may not want to modify the kernel source for various reasons (legal, recompilation, ...).
- You may not know, or want to know every single detail of `foo()` - all you want is to do one or more of the following: *pre-process* the function's arguments, call it and *post-process* the result.

This kind of thing is really easy to do with VFSs. Traditionally, you deal with a collection of function pointers, and it is easy to stack such collections where a specific function `bar()` in each layer do many things: pass through to the layer below, pre- or post- process arguments/ results, block the call from passing through and so on.

Things are a little harder when there are no function pointers. In the situation described earlier, `foo()` is a function at some address in the kernel. There may be an arbitrary number of callers that invoke this function. What follows is a scheme that allows you to *replace* `foo()` with `my_foo()` so that calls to `foo()` result in `my_foo()` being invoked. `my_foo()` can call `foo()` itself though. Consider the picture shown on the right.

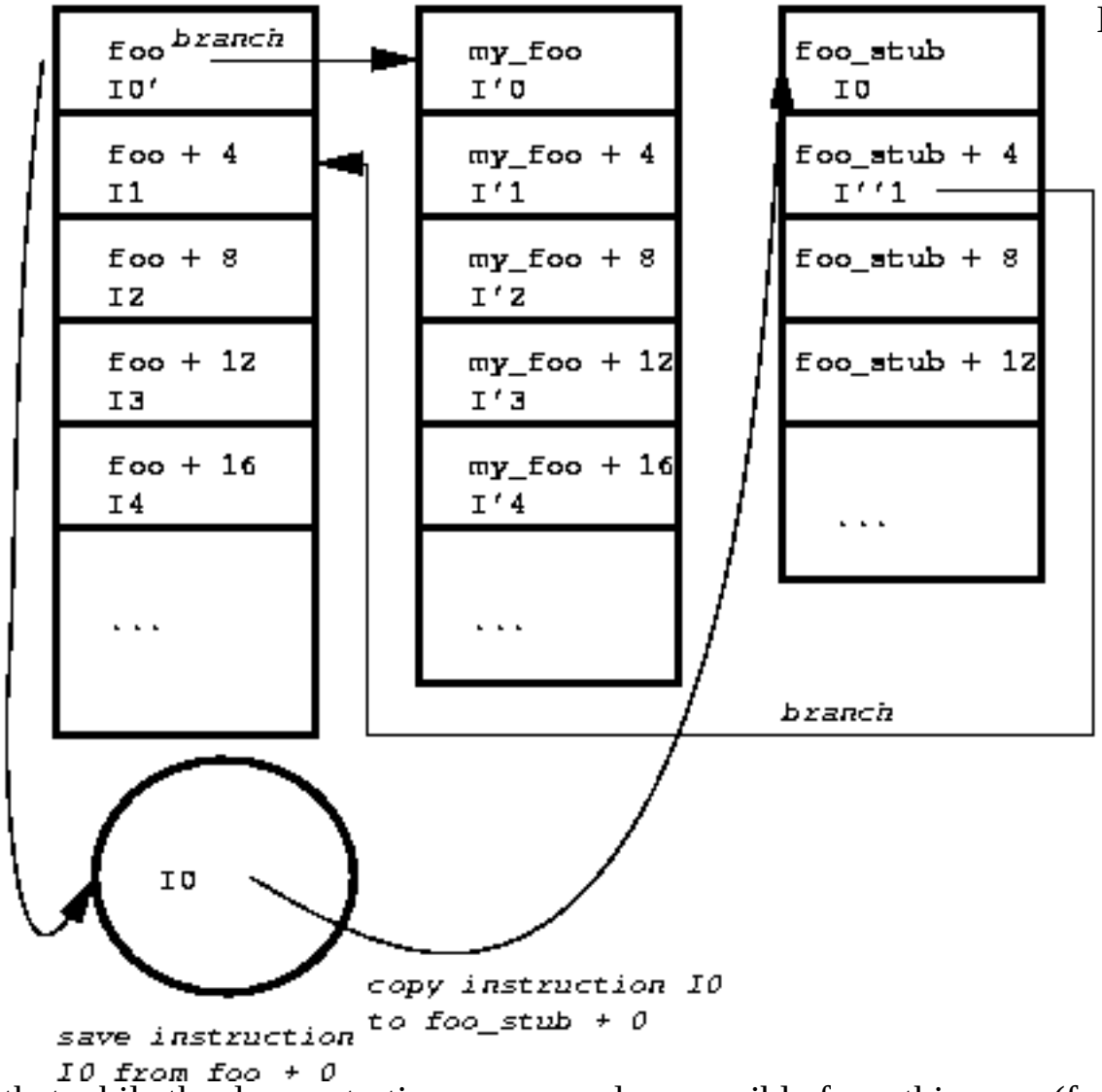


Assume that `foo()` and `my_foo()` are present as shown, with their

respective instructions being `I0 I1 I2 I3 ...` and `I'0 I'1 I'2 I'3 ...` respectively.

A caller of `foo()` eventually executes `I0`. If we were to re-route `foo()` so that `my_foo()` would be called instead, an obvious thing to do would be to replace `I0` with an unconditional branch to `my_foo()`. That would effectively leave `foo()` out of the picture, and all calls to it would implicitly end up invoking `my_foo()`. This is fine if we do not wish to call the real `foo()` at all, but more often than not you do not wish to clobber `foo()`, but want to be able to call it, say from within `my_foo()`, as discussed earlier. A little more work is needed to achieve this.

We save `I0`, the clobbered instruction from `foo()` somewhere. Then we allocate a region of memory large enough to hold a few instructions and mark it as executable (via a call such as `mprotect()`). We call this region `foo_stub`. Note that it is not necessary to dynamically allocate this region of memory - a very convenient way to "pre-allocate" such a region would be to have an actual function (that does nothing) `foo_stub()` alongside `my_foo()`. Finally, we copy `I0` to the beginning of `foo_stub`. The very next instruction would be an unconditional branch to `foo + 4`. Thereafter, the "original" `foo()` can be invoked via `foo_stub()`.



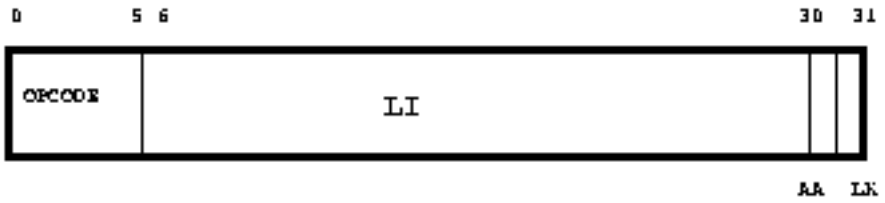
Note that while the demonstration source code accessible from this page (for PowerPC/Mac OS X) does work, it should be considered a prototype. If you were to do such re-routing at production level, you would have to worry about various issues: invalidating the instruction cache, multi-processor complications, the maximum possible distance between `foo()`, `my_foo()` and `foo_stub()` etc. These issues will greatly depend on the specific processor you are working on (SPARC, PowerPC, x86 ...).

The "distance" issue is more severe on certain processors. A jump (unconditional branch) on MIPS uses 6 bits for the operand field and 26 bits for the address field. The effective addressable distance is 28 bits (4 times more) because MIPS refers to the number of words instead of bytes. This works because all instructions are 4 bytes long. 28 bits gives you over 268 MB of leeway.

SPARC uses a 22 bit signed integer for branch address, but has two zero bits appended to it, effectively a 24 bit program counter relative reachability. This gives you 16 MB (+- 8MB) of leeway.

The unconditional branch in our target architecture (PowerPC) has a 24 bit address field, but as in MIPS, all instructions are 32 bits long, so PowerPC refers to words instead of bytes. The effective branch address (`LI || 0b00`) is 26 bits wide, which gives you 64 MB (+- 32 MB) as the maximum distance you can jump. Refer to the picture below for details.

In the




```
b    target_addr (AA = 0, LK = 0)
ba   target_addr (AA = 1, LK = 1)
bl   target_addr (AA = 0, LK = 1)
bla  target_addr (AA = 1, LK = 1)

target_addr specifies the branch target address

If AA = 0, then the branch target address is the sum of
LI || 0b00 sign extended and the address of this
instruction.

If AA = 1, then the branch target address is the value
LI || 0b00 sign extended.

If LK = 1, then the effective address of the instruction
following the branch instruction is placed into the link
register.
```

provided sample code, we assume that our jump will be less than 32 MB in absolute value (no exception handling is attempted if this is not the case!). You are very likely to run into situations when this would *not* be the case: it could be that you are trying to re-route a Mac OS X kernel function that is too far from your version of that function. Though innovative solutions might be possible depending upon the instruction set/processor in question, but a straightforward solution would be to find and use an *unused* region of memory near that function.

Finally, please note that on a production Mac OS X system utilizing this technique, you would use something like `dcbf` and/or `icbf` to flush the cache(s). The sample code does not do that, but simply calls an arbitrary function (say, `sync()`) which effectively has the same result in our case.

Re-routing Dynamic Library Calls

I have run across situations analogous to the following multiple times: *you need to run some program P that you acquired somehow - maybe your company bought it, perhaps you downloaded its binary, maybe it's even open source ... P intends to do something to a network interface,*

but you have multiple interfaces on your system. You want P to use a certain interface, but P does not have provision for specifying that. It simply chooses the first interface on the system as determined by a call to the `SIOCGIFCONF` *ioctl.*

I realize that the above sounded terribly contrived, but what do you do?

A commonly used solution to such "problems" is to "pre-load" a library that "fixes" the issue. In the above case, you would write your own version of `ioctl()` that makes a call to the "original" function/stub in the C library, say, and then post-processes the result to make it look the way you want. You may not even want to call the original function and cook up a result entirely on your own. Traditional ELF systems let you direct the dynamic linker/loader to load specified libraries before all others. This is most commonly done via the `LD_PRELOAD` environment variable, and files such as `/etc/ld.so.preload` for global effect.

Mac OS X does not use ELF. Its runtime model certainly supports dynamic shared libraries, but uses the `Mach-O` and `PEF` binary file formats. `dyld`, the dynamic linker (Mac OS X analog of `ld.so`) and `CFM` (Code Fragment Manager) use these formats, respectively. Just as the operating system kernel in ELF systems executes ELF binaries through `execve()`, Mac OS X kernel does the same with Mach-O. Moreover, the `dl*` API (`dlopen`, `dlsym`, ...) is not supported.

`dyld` does support a number of environment variables that direct its behavior. The closest thing to `LD_PRELOAD` is the variable `DYLD_INSERT_LIBRARIES`, but there is one *big* caveat: *this has no effect on images using two-level namespaces*. The variable `DYLD_FORCE_FLAT_NAMESPACE` needs to be set in order for `DYLD_INSERT_LIBRARIES` to "work". This forces all images in the program to be linked as flat-namespace images (ignoring any two-level bindings). If there *are* multiply-defined symbols because of a forced flat namespace, applications may fail to run properly, or to even run.

Command Line Archival in Mac OS X

Forked Resources

`/bin/pax` on Mac OS X is a command line utility that can read and write file archives and copy directory hierarchies. `Pax` supports various output archive formats:

- cpio
- bcpio
- sv4cpio
- sv4crc
- tar
- ustar

Although `pax` is a useful utility, it does *not* handle "resource forks" on HFS+ file systems. HFS+ files can have two parts (forks): data and resource. The "data fork" contains what a file would usually contain on traditional file systems. The "resource fork" can contain arbitrary information: icons, preview pictures, key-value pairs, program segments etc.

Other command line utilities such as `cp`, `cpio`, `dump` and `tar` also will not handle resource forks. Much of commercial software (most of Adobe's offerings, for example) make use of resource forks, and as such, if any of these utilities are used to handle files with resource forks, things are not going to work properly.

`hfstar` and `hfspax` are modified versions of the respective utilities that handle resource forks.

`MacOSX::File` is a perl module that allows you to get and set HFS+ file attributes. `psync` is an accompanying perl script that uses `MacOSX::File` to implement incremental backup and restore.

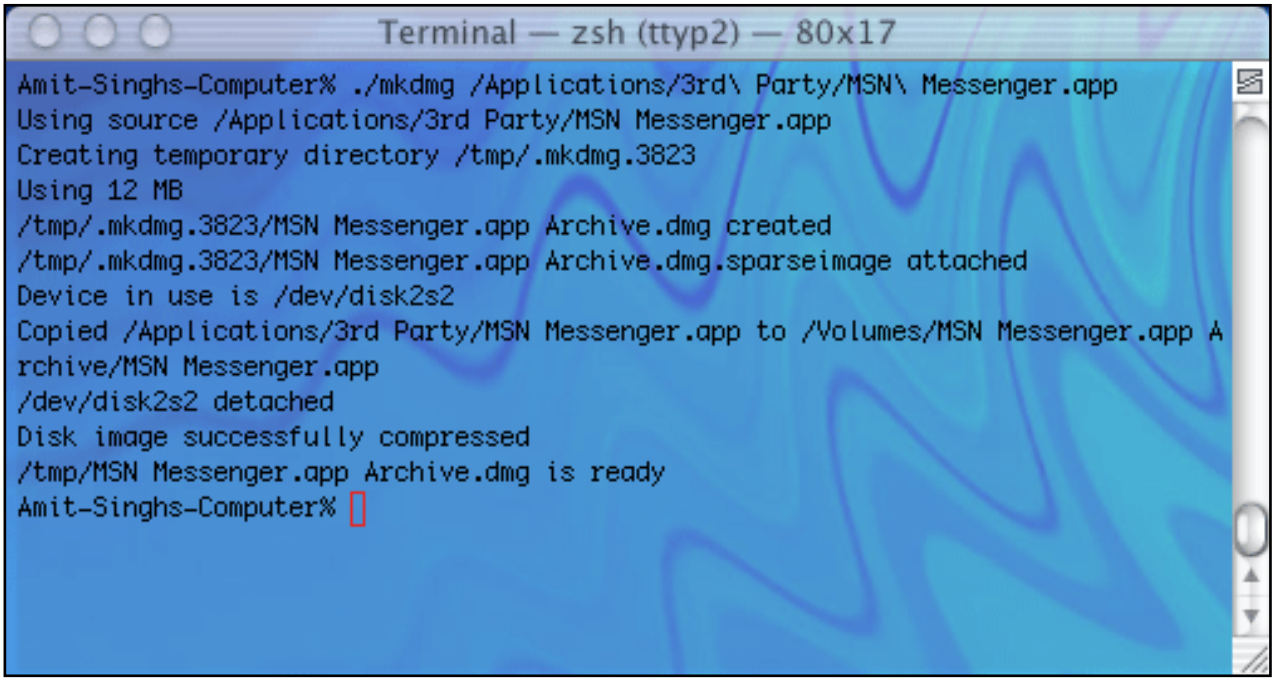
`/usr/bin/ditto` (present by default on Mac OS X) and `/Developer/Tools/CpMac` (present if Developer Tools are installed) can be used to copy files and directories with resource forks preserved.

Finally, there must be (are?) various commercial software packages/ tools that allow full archival of HFS+ file systems.

A Quick n' Dirty Archiver

Even with the complications described above, enough useful utilities are present by default on Mac OS X that a reasonably powerful archiver can be written rather straightforwardly. What follows is an example

depicting use of such utilities. The specific utilities used are `ditto`, `hdid` and `hdiutil`.



We would be developing a small shell script (using `/bin/zsh`, the Z shell) called `mkdmg` that creates a disk image (`dmg`) from files and/or directories. For argument parsing simplicity, the script would only accept *one* file or directory as argument. **Please understand that this script should be considered alpha level software and should be used at your own risk!**

Annotated Source: mkdmg

```
#! /bin/zsh
#
# mkdmg - create a disk image from files/folders
# Copyright (c) 2003-2004 Amit Singh. All Rights Reserved.

# Scratch folder where temporary archives will be created
#
SCRATCH=/tmp/.mkdmg.$$

# Reset path
#
PATH=/bin:/sbin:/usr/bin:/usr/sbin
```

We would be using `/tmp/.mkdmg.$$` as a temporary location where

intermediate archive files are stored. Be warned that if you try this script on folders with excessively large amounts of data in it, you need to have at least twice as much temporary disk space.

```
# Utility function for output
#
croak() { echo -n "\n$1" }

# Utility function for checking return status
#
chkerror() { if; [ $? -ne 0 ]; then; halt; fi }

 Utility function for clean up
#
halt()
{
    rm -rf $SCRATCH
    # defaults write com.apple.finder
ShowRemovableMediaOnDesktop 1
    # chkerror
    # FINDERPID=`ps -auxwww | grep Finder.app | \
    #     grep -v grep | awk '{print $2}'`
    # chkerror
    # kill -HUP $FINDERPID 2>/dev/null >/dev/null
    # chkerror
    exit 1
}
```

One possible (though unnecessary) "tweak" could be to prevent the disk image icon from appearing on the Desktop during the execution of the script. This requires setting the value of the key `ShowRemovableMediaOnDesktop` to `0` in the domain `com.apple.finder` using `defaults`. The `Finder` application should be sent a `HUP` signal after doing so. If this tweak is enabled, the cleanup function should undo the tweak.

```
main()
{

    # Check if exactly one command line argument was specified
    #
    if [ $ARGC -ne 1 ]
    then
```

```
    echo "usage: mkdmg <file|directory>"
    exit 1
fi

# Check if the specified file/directory exists
#
if [ ! -e $1 ]
then
    echo "*** $1 does not exist."
    exit 1
fi

SRC=$1
NAME=`basename $SRC`
NAME="NAME"
ARCH="$NAME Archive"

echo -n "Using source $SRC"
# Change directory to a scratch location
#
cd /tmp

# Create a scratch directory
#
mkdir $SCRATCH
croak "Creating temporary directory $SCRATCH"
```

If "`/foo/bar`" is the complete path of the source file/folder, the output archive will be named "bar.dmg" with the volume name being "bar Archive".

```
# Estimate how much space is needed to archive the file/folder
#
SIZE=`du -s -k $SRC | awk '{print $1}'`
chkerror
SIZE=`expr 5 + $SIZE / 1000`
chkerror
croak "Using $SIZE MB"

# Create a disk image, redirecting all output to /dev/null
#
hdiutil create "$SCRATCH/$ARCH.dmg" -volname "$ARCH"\
    -megabytes $SIZE -type SPARSE -fs HFS+ \
    2>/dev/null >/dev/null
chkerror
```

```
    croak "$SCRATCH/$ARCH.dmg created"
```

The above code estimates the required size of the target volume, and creates a sparse disk image with an HFS+ file system on it.

```
    # Optionally disable display of removable media on Desktop
    #
    # defaults write com.apple.finder
ShowRemovableMediaOnDesktop 0
    # chkerror
    # FINDERPID=`ps -auxwww | grep Finder.app | \
    #     grep -v grep | awk '{print $2}'`
    # chkerror
    # kill -HUP $FINDERPID 2>/dev/null >/dev/null
    # chkerror
    #

    # Mount sparse image
    #
    hdid $SCRATCH/$ARCH.dmg.sparseimage 2>/dev/null >/dev/
null
    chkerror
    croak "$SCRATCH/$ARCH.dmg.sparseimage attached"

    # Find out allocated device
    #
    DEV=`mount | grep "Volumes/$ARCH" | awk '{print $1}'`
    croak "Device in use is $DEV"

    # Use ditto to copy everything to the image
    # Resource forks will be preserved
    #
    ditto -rsrcFork $SRC "/Volumes/$ARCH/$NAME" \
        2>/dev/null >/dev/null
    chkerror
    croak "Copied $SRC to /Volumes/$ARCH/$NAME"

    # Detach the disk image
    hdiutil detach $DEV 2>/dev/null >/dev/null
    chkerror
    croak "$DEV detached"
    # Compress the image (maximum compression)
    hdiutil convert "$SCRATCH/$ARCH.dmg.sparseimage" \
        -format UDZO -o "/tmp/$ARCH.dmg" -imagekey \
        zlib-devel=9 2>/dev/null >/dev/null
```

```
    chkerror
    croak "Disk image successfully compressed"

    croak "/tmp/$ARCH.dmg is ready"

    echo

    halt
}

main $1
```

Xcode Templates

Empty Project

Application

- AppleScript Application
- AppleScript Document-based Application
- AppleScript Droplet
- Carbon Application
- Cocoa Application
- Cocoa Document-based Application
- Cocoa-Java Application
- Cocoa-Java Document-based Application

Bundle

- Carbon Bundle
- CFPlugin Bundle
- Cocoa Bundle

Dynamic Library

- BSD Dynamic Library

- Carbon Dynamic Library
- Cocoa Dynamic Library

Framework

- Carbon Frameowrk
- Cocoa Framework

J2EE

- EJB Module
- Enterprise Application
- Web Module

Java

- Java AWT Applet
- Java AWT Application
- Java JNI Application
- Java Swing Applet
- Java Swing Application
- Java Tool

Kernel Extension

- Generic Kernel Extension
- IOKit Driver

Standard Apple Plug-ins

- AB Action Plug-in for C
- AB Action plug-in for ObjectiveC
- AppleScript Xcode Plugin
- IBPalette
- Preference Pane

- Screen Saver
- Sherlock Channel

Static Library

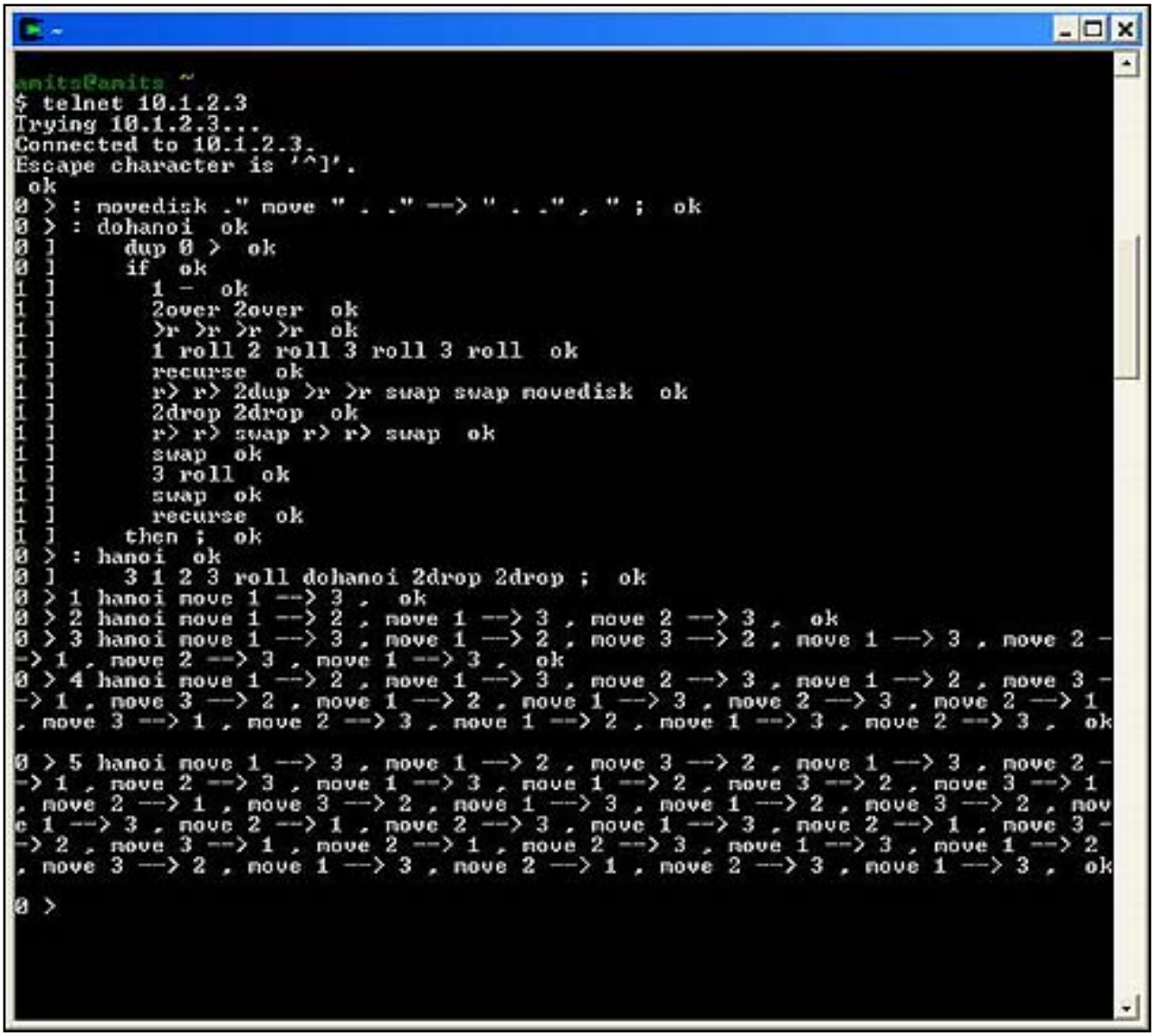
- BSD Static Library
- Carbon Static Library
- Cocoa Static Library

Tool

- C++ Tool
- CoreFoundation Tool
- CoreServices Tool
- Foundation Tool
- Standard Tool

The Towers of Hanoi in Open Firmware

Following is a screen shot of "The Towers of Hanoi" as a FORTH program for the Open Firmware implementation found on "new world" Macintosh systems. This program is essentially the same as the one for Sun OpenBOOT, except that `2r@` and `2r>` are not implemented on Apple's PROM.



Note that rather than type this program at a Macintosh's keyboard, it is easier (particularly for the purpose of taking a screen shot) to use the TELNET protocol to connect a client machine (any platform) to a target machine (Macintosh) running Open Firmware. The procedure is outlined below.

First we need to determine if the Macintosh in question is capable of being a TELNET server while in Open Firmware. Boot the machine into Open Firmware (`Cmd-Option-O-F`) and type the following at the `0>` prompt:

```
dev /packages ls
```

If the output (a listing of Open Firmware packages) contains a node named `/telnet`, then the machine is capable of being debugged over Ethernet.

Connect the Macintosh to another Ethernet capable machine with a TELNET client using a crossover Ethernet cable. Type the following at the Open Firmware prompt of the target machine:

```
" enet:telnet,10.0.0.1" io
```

This would launch a TELNET server with IP address `10.0.0.1` on the target machine. The Ethernet interface of the client machine should be configured with an appropriate address (for example, `10.0.0.2` as the IP address and `255.255.255.0` as the netmask). Thereafter, a TELNET client can be used on the client machine to connect to the Open Firmware of the target Macintosh:

```
telnet 10.0.0.1
```

As the screen shot above might suggest, I used a Windows machine as the client.

The Saga of a PowerBook 17

Background

Sometime in October 2002, I concluded that I need to get a new notebook computer. I am not a "Mac Person" or a "PC person", and I'm not saying this out of embarrassment of the otherwise implied cliché. I have owned machines with many different platforms in the past, and have worked on yet more. Machines that I have owned (not necessarily *purchased* myself) include:

- ARM based development board
- Compaq Notebook
- CRT iMac
- Dell Notebook
- HP PA-RISC based machine
- IBM ThinkPad Notebooks

- Intel based desktop PCs
- MIPS based development board
- Sony VAIOs
- UltraSPARC based Sun machine
- "White" iBook

Nonetheless, no matter how platform agnostic (or "omni-platformic") a person is, one has to choose *some* platform, *some* machine, where one can, say, "run one's life". As would the case be with most "computer people", I like to have a single machine, preferably portable, which can serve as my browsing station, mail receiver/sender, document repository and a reasonable development platform (among other things).

I have owned a few Sony VAIOs in the past, courtesy my work-place. I got the first of these, a Z505, in 1999. It looked very promising indeed - light, sleek (sleekness is subjective), reasonably well featured (FireWire, USB, IR etc. in 1999 *was* good). Unfortunately, my VAIO experience turned out to be rather bad. Within the next three years, I went through no less than three VAIO notebooks, each developing one or more of the following problems rapidly:

- Failing battery (charge goes from 100% to 1% in a couple of minutes).
- Trackpad buttons failing to function.
- PC Card slot going "bad".
- Power socket on the notebook "breaking".
- Ethernet jack malfunctioning.
- Multiple dead pixels.
- Hard disk failing, again, and again!

Now, it may seem that I was exceptionally unlucky with VAIOs, and I do believe I was. Hard disks can fail in *any* computer, but I went through three disks in less than a year, and I must say I didn't enjoy it. Eventually I decided to get a new notebook.

After doing some research on notebooks, I liked the IBM ThinkPad T30. I didn't really consider Apple's Titanium PowerBook because:

- I wanted a notebook that I can also use for my work, which is strictly Linux/Solaris based. I wanted a machine that can run

x86 operating systems natively.

- I regarded the PowerBooks as "pricey".

The ThinkPad T30 was not a cheap machine either (I intended to get the highest end model in that line). I was in a position to get a substantial discount from IBM though, hence it was tempting. There were a few reasons that made me *very* hesitant to go ahead with the purchase:

- Lack of FireWire ports on the T30 (I had a few FireWire peripherals so this was important)
- A really outdated video card with 16 MB of video memory

There were rumors that IBM might be updating the "T" line with much better video cards, processor upgrades etc. In November 2002, I was about to go on a long vacation soon, so I decided to wait. Somehow December passed and I still had not purchased any notebook. In early January, Apple announced the PowerBook G4 17, and I liked the machine so much that I decided to get one. I conveniently "overlooked" some of the reasons I cited earlier: price, the x86 factor, etc. By end of January, I had made a firm decision that I was going to buy the PB17.

The Buying Process

Once I was sure I wanted to buy the PB17, I did some research on the optimal method of buying it. You might want to read my article on *How To Buy An Apple Computer*.

It was very frustrating to follow the PB17 availability saga. Due to various reasons, I ordered rather late, on March 5th, 2003. The procrastination did *not* cause any additional frustration, fortunately.

When I ordered, the estimated shipping time was "3-5 weeks", and my "On or Before" date was April 9th. I was hoping that Apple would start shipping them much sooner. The various Apple rumor sites (click here for a list of some of these) were overflowing constantly with speculations and (mis)information about PB17 (un)availability. This should be well known to those who ordered and waited for their notebooks during this time!

I was pleasantly surprised to learn that my PB17 had been shipped on March 29th - reasonably ahead of the worst case time. As has been the case with many PB17 shipments from Apple, my "2 day shipping" method was upgraded to "overnight priority". Its journey from Taiwan to the United States went smoothly. It was not held up in the US

Customs, and I was all set to receive it by 10:00 am on April Fool's Day.

The Arrival Day

I woke up on April 1st to the noise of my phone ringing. I looked at the time and it was 9:30 am. Shortly afterwards, my wife told me that my friend had called from work to tell me that my PB17 had been delivered. I was, well, *excited* - after all, it *is* a new gadget, to say the least. I got out of bed to go take a shower, and commented on how dark and cloudy it was. My wife told me to go back to sleep. It was 6:30 am. She had set the bedroom clock to 9:30am, and had been successful at her April Fool prank (a few days before I had told her with supreme confidence that I was invulnerable to *any* of her such attempts). I sheepishly told her that PowerBook or no PowerBook, I had been intending to go in early to work *anyway* (yeah, right).

After I reached my work-place, I tracked the package and found that it was on FedEX's vehicle - out for delivery. Right afterwards, I went into a meeting for an hour. When the meeting finished, the package status was "**Delivered**". "*Great!*", I said to myself, and walked to the Receiving area. I was dismayed to find that they had not received *anything*. I asked our receptionist at the front desk, and she had not received anything either. I looked at the tracking information again. Apparently FedEX don't have the entire street address in the "Delivered To" field - just the city. One thing that *was* out-of-place was the name of the person who signed for it: some "*M. Sejdic*". We do not have anybody with that name at work.

I was rather annoyed and concerned by now. The person in charge of Receiving told me that in the past, FedEX *has* delivered packages addressed to us (say, "1234 Foobar Avenue") to a nearby company (say, "1234 Friggin Avenue"). He was nice enough to drive me to "Friggin Avenue" promptly. We asked the front desk at "1234 Friggin Avenue", but they were not helpful, and were actually bordering on unfriendly. We tried to locate their Receiving, and they behaved as if we were the Mafia looking for some payback. I don't have anything to gain by blaming them though.

The next thing we did was what we should have done *first*.

We called FedEX. This is how the conversation went:

FedEX: *Hi, this is Chris at FedEX. How may I help you?*
Me: *Hi Chris, we have package that we would like to track.*
FedEX: *OK. May I have the tracking number please?*
Me: *Sure. It is xxxxxxxxxxx.*

FedEX: *Just a moment please.*
FedEX: *OK, this is a priority overnight package was delivered on or before time at 10:26 am this morning.*
Me: *Really? Could you please check where it was delivered?*
FedEX: *Ummm ... it was addressed to 1234 Foobar Avenue. It was delivered to ...hmmm ... 1077 East Arques Avenue. I think it was mis-delivered.*
Me: *What!? Mis-delivered? The two addresses don't even have a character in common - how could you mis-deliver it?*

I knew I was exaggerating when I said that the two addresses had nothing in common. They both had the word "Avenue" in common, and in all honesty, that is more than enough to throw a FedEX guy off. I *was* livid, but I wanted to get my PowerBook "back", so I continued the conversation:

Me: *Could you please tell me what I should do now?*
FedEX: *Oh well they do mis-deliver sometimes. You can go and pick it up yourself if you want.*
Me: *You think they will just give it to me?*
FedEX: *They should.*
Me: *What if they deny receiving it?*
FedEX: *They can't. They signed for it.*
Me: *Can you tell me the name of the company, or is it a residence?*
FedEX: *It doesn't say that here. I only see the address that I gave you.*
Me: *OK. Thank you.*
FedEX: *Thank you for calling FedEX, and have a nice day.*

I was rather upset at this point. Sure - people make mistakes. Sure, FedEX delivers so many packages every day and some of them might be mis-delivered. However, I was concerned about *my* package and the fact that it might be, well, lost.

The address my package was mis-delivered to is about 7 miles away from my work. It has nothing in common with my work address. I was really perplexed as to how this could happen. My friend drove me to East Arques Avenue, and we tried to find 1077, but couldn't! There seemed to be a hole in the number sequence - they went up to 1059 (or something close), then there was a big parking lot, and then the numbers started at 1099 (or something close). 1077 was nowhere to be seen. The big parking lot belonged to *Fry's Electronics*. My friend joked that it would be so "cool" if they actually gave my PowerBook to Fry's. He thought they might sell it! *Then*, it dawned upon us. My package most probably *was* delivered to Fry's - perhaps with a bunch of other PowerBooks.

We quickly drove around to the back of Fry's, where they have their

Receiving. The receiving area is "protected" by a wall of honeycombed wires. Apparently no "outsider" goes there because some guy who was smoking on the other side of the wall stared at us in blatant awe. We asked if we could speak to somebody who takes care of Receiving. They told us that if we want to buy something from Fry's, the store entrance is on the *other* side! A lady came to talk to us, but she did not speak much English. Nevertheless, I tried to convey to her what the situation was. She laughed at me and told me that what I was saying is impossible. Fry's checks the address labels on every package, and there is no way they could have accepted something with my address on it.

I was exasperated. I was annoyed. I was *very* impatient. At the same time, I do believe I was *very* polite all along (good for me).

I handed her a printout of my tracking information, and requested her to go check if this particular package was received by them. I told her that it was signed for by a certain "M. Sejdic". To this, she said: "Oh, he has gone for lunch already." She did go inside for a full fifteen minutes, and came back to tell me that the said package is indeed with them, but she cannot give it to me. Moreover, her manager has gone out for lunch (it was 11:45 am), and would come back at 1:00 pm. She told me that it is *not* Fry's responsibility, but FedEX's fault. I asked her about Fry's processes that involved checking the address label before accepting, to which she replied monotonically that it was FedEX's fault. I realized that I was wasting my time there, and decided to retreat and come back at 1:00 pm.

While driving back, we remembered that the FedEX distribution center for our locality is nearby, and we could go there and "try". We drove there, and here is how the conversation went, after I explained that my package is at Fry's:

Me: *You do realize it was a priority overnight package, and it has a notebook that cost me over $3,000.*
FedEX: *Oh I understand sir, but this happens. They probably unloaded your package along with the others at Fry's.*
Me: *Well, I went there and they are not giving it to me, and their manager is out for lunch.*
FedEX: *If you want, we can go and get it for you.*

I said to myself: "*What blinding generosity!*"

Me: *Yes, I would like you to get it for me.*
FedEX: *When do you want it? I mean, I know it was priority, but do you want it soon or can you wait?*
Me: *I would like it right away please!*

FedEX: *OK sir, we will have the driver go to Fry's, pick it up and deliver it to you, say, by 2:00 pm (it was 12:30 pm).*
Me: *Yes, I would appreciate that very much.*
FedEX: *All right. I apologize for the inconvenience.*
Me: *Thank you very much.*

I drove back to work, feeling apprehensive and unhappy.

After a little over 30 minutes, a FedEX driver came looking for me at my work. He *had* my PowerBook with him. He mentioned that "*Fry's almost didn't give it back!*". As I signed for it, I asked him if a red flag should be raised while scanning if you deliver a package to the "wrong" address. He said that the scanning only updates the tracking information - it does *not* cross-check the "addressed" to and delivered to "addresses". I also asked him if they count how many boxes they are giving out, to which he said that people do make mistakes.

Whatever.

I was glad that this unnecessary drama had ended, and there would be no further worries. I was wrong.

April 1st Jinx: It's Not Over Yet

I unpacked the machine and was very pleased (yet again - I had played with the PB17 a few times already) with the overall look and feel. The first thing I intended to do was to run the AHT (Apple Hardware Test) disc and verify that the memory is fine and find out (if any - hope not!) dead or stuck pixels.

There is a single DVD containing the AHT, the MacOS X Installer and some other software.

The machine came up and much to my dismay, the hardware test exited immediately with an ominous "memory access" error. I tried to boot the machine (without any hardware test), and it appeared to boot fine. However, this was no guarantee that the memory is "good". The memory (a 512 MB SO-DIMM) that came installed had a Samsung label. I had an additional 512 MB Kingston SO-DIMM that I had purchased through Apple's endorsement by following a link from their Made4Mac section on their web site.

I tried various things, and found out several things:

- Apple's memory causes the AHT to fail immediately if it is

installed in either slot, and if it is the only SO-DIMM installed.

- Kingston's memory works fine in either slot if it is the only SO-DIMM installed. AHT succeeds.
- If I install both SO-DIMMs, AHT still fails, regardless of which stick is in which slot.
- The system boots and appears to work fine in all cases.

I wanted peace of mind, and no further nonsense (my April 1st was not going too well). I called AppleCare and explained the situation to them.

They told me that normally what is done is that they would send out a new SO-DIMM to me, and I can send the defective one back. However, since the PB17s are so new, *"they do not have the part number for memory on their list of things to send out as replacement"*. They told me that the only thing they can do is the following:

- They send me a "box". It will take a couple of days, maybe.
- I put my PB17 in the box and send it to them (the whole thing).
- They will replace the memory, test it, and send it back to me.

I certainly did not want that. I also suffer from the perfectionism disease, so I wanted to be up and running with 1024 MB of RAM (if at all feasible) by the end of the day. Therefore, I did not give up, and asked them if I can walk into an Apple Store and request them to replace my RAM. The AppleCare guy encouraged me to give it a shot, but he warned me that the Apple Stores would not have received any "replacement" stock yet, so chances were next to nil.

I called a nearby Apple Store anyway. I asked them if they had any PB17 memory in stock. They told me that they did! I then reiterated the conversation I had with the AppleCare people. I was told that the Apple Store did have RAM in stock, but for *selling*. If I wanted to *buy* it, I was most welcome and they would hold one for me. If I wanted one replaced, *that* would not be possible.

I told them the FedEX story, and after some persuasion they agreed to help me out. I drove to the Apple Store, and waited at the "Genius Bar". There were around five people in the queue. I overheard one "interesting" conversation between an Apple Genius and a lady with a Pismo which is noteworthy:

Lady: *I was recommended these PC Cards because I needed USB on*

my notebook, but they seem to be very poor quality. They cost $70 each, and they only last one to two weeks, and then you have to buy new ones.
Apple Genius: ...
Lady: *I have another question. I need to install this software, and the instructions say "open your notebook". I am really scared because I think they mean unscrew the screws at the bottom and open it up ... or do you think they mean just lift the screen up?*
Apple Genius: *Ma'am, I think they just mean power it up.*

They took my PowerBook and gave me a claim ticket. I was hoping that this would be quick. After all, they had to run the AHT once, verify that the RAM is "bad", replace the RAM, run AHT again, verify that the RAM is "good", and that's it.

Unfortunately, I had to wait for an hour and a half. They *were* working on it all the while, so there *was* some problem.

The AppleCare guy I had spoken to came to me and told me that they tried 10 different SO-DIMMs, and they *all* caused the AHT to die. I am not sure if they actually spoke to an AHT engineer, but they told me that nothing was wrong with my computer or the memory, it was the *test* that was buggy. There will be an updated hardware test disc soon.

I thanked the Apple Store folks and came back.

I didn't know what to say!

Interesting day, indeed.
