Mac OS X DP2

John Siracusa Dec 14, 1999 3:00 PM

Tech

Mac OS X DP2 is a window into the future of Mac OS...a dark, scary future ...

Mac OS X DP2 is the second "Developer Preview" release of Mac OS X (pronounced "ten", not "ex").

What is Mac OS X? The most common answer is that it's the "client" or "consumer" release of Mac OS X Server. (What is Mac OS X Server? Find out.) Like its server counterpart, Mac OS X is targeted at any Mac that Apple shipped with a PowerPC G3 processor or better. This means that Macs based on the PowerPC 60x processor are stuck with Mac OS 9.x--even if they've been upgraded with a G3 processor card. The scheduled release date for Mac OS X is (surprise!) a moving target. The current party line has Mac OS X on store shelves some time in 2000. I fearlessly predict that it will not appear until 2001 at the earliest (unless they decide to ship a half-finished product a la Mac OS X Server 1.0), but maybe I'm just a pessimist.

What differentiates a "consumer" or "client" release of an OS from its "server" version? In the case of Mac OS X, the answer is "a lot." Or rather, the answer *had better be* "a lot," because Mac OS X Server in its current state is absolutely unfit as a replacement for Mac OS 8.x/9.x ("classic" Mac OS). Mac OS X Server 1.x is an OS in transition, half way between its NEXTSTEP roots and its Mac OS future. It looks kind of like Mac OS, but it behaves like NEXTSTEP. This will simply not do for an OS that aspires to supplant classic Mac OS. Thus the need for Mac OS X, sometimes referred to as "the real Mac OS X."

Unfortunately, Mac OS X DP2 is not that OS. It's simply an environment that allows developers to build and test their applications on an early version of the core of what will become Mac OS X. This being the case, certain traditional software review metrics make little sense: ease of installation, hardware support, feature count, even performance. This article will not be your typical review, or even your typical preview. There will be no step-by-step installation instructions, no comments on the lack of DHCP support, no guided tour of the networking setup, user account administration systems, and no benchmark graphs.

Mac OS X DP2 itself may not be the future of Mac OS, but it does provide a lot of information regarding the likely future of Mac OS X if you're willing to do a little digging. So let's get started.

The core OS

Mac OS X DP2 is based on a variant of the mach microkernel. Mach was developed in the 1980's and has gone through significant changes since its initial release in 1986 running on a multiprocessor VAX 11/784. It's been used in NEXTSTEP (Mach 2.0 and 2.5), Apple's MkLinux (Mach 3.0), and Mac OS X Server (Mach 2.5 with modifications). Running "`uname -a`" in Mac OS X DP2 reveals this mess:

> Mac OS 10.0 Mac OS Kernel Version 10.0: Mon Oct 25 23:31:09 PDT 1999; root(rcbuilder):xnu/xnu-44:xnu/xnu-44.obj~1/RELEASE_PPC Copyright (c) 1988-1995,1997-1999 Apple Computer, Inc. All Rights Reserved. Power Macintosh powerpc

from which it is not too much of a leap to conclude that Apple plans on starting its own kernel version numbering system at 10.0 and going on from there. Poking around a bit longer confirms this suspicion:

```
/*
 * File: sys/version.h
 *
 * HISTORY
 * 29-Oct-86  Avadis Tevanian (avie) at Carnegie-
 *      Created.
 */

/*
 *      Each kernel has a major and minor version
 *	the major number in general indicate a ch
 *	Changes in minor number usually correspon
 *	changes that the user need not be aware o
 *	values are stored at boot time in the mac
 *	can be obtained by user programs with the
 *	This mechanism is intended to be the form
 *	to provide for backward compatibility in
 *
 *	Following is an informal history of the n
 *
 *	20-Mar-1998	Umesh Vaishampayan
 *		MacOSX DR2
 *
 *	28-Sep-94 David E. Bohman
 *		NEXTSTEP Release 4.0.
 *
 *	03-Sep-91 Doug Mitchell
 *		Major 3 for NeXT release 3.0.
 *
 *	04-Mar-90 Avadis Tevanian, Jr.
 *		Major 2, minor 0 for NeXT release
 *
 *	11-May-89 Avadis Tevanian, Jr.
 *		Advance version to major 1, minor
 *		release 1.0.
 *
 *	05-December-88 Avadis Tevanian, Jr.
 *		Aborted previous numbering, set ma
 *		to conform to NeXT's 0.9 release.
 *
 *	25-March-87  Avadis Tevanian, Jr.
 *             Created version numbering scheme.
 *             minor 0.
 */
[snip]

#define KERNEL_MAJOR_VERSION    10
#define KERNEL_MINOR_VERSION    0
```

So 10.0 it is. Quite a jump from 3.0, but consistency is king, apparently. Also note the presence of Avie Tevanian (Apple's Senior Vice President for Software Engineering since the acquisition of NeXT) throughout the history of Mach. This bodes well for the future of the kernel (if not necessarily the OS), as it could not be in more capable hands.

Peeking into the Mach kernel binary itself reveals some interesting strings like "`Starting CPU %d`" and "`UpdateFolder: found HFS+ folder on HFS volume`", indicating that Mach in DP2 is being extended in both directions: future-proofing for multiple processor Macs, and providing support for legacy Mac OS file systems like HFS.

DP2's ability to "boot and root" from an HFS+ volume (rather than UFS, a traditional Unix-like file system) is further evidence of the advances to the core OS made since the release of Mac OS X Server. This change is transparent: an HFS+ volume with Mac OS X DP2 installed on it does everything that's expected of it (supports Unix-style file permissions, case-sensitive long file names, etc.) and behaves like a normal (if oddly filled) HFS+ volume when viewed from within classic Mac OS--quite a nifty trick.

Application Programming Interfaces

Mach was designed to be modular, making it easy to implement almost any API on top of it. Apple takes advantage of this ability to great effect in DP2. Here are the currently supported APIs:

> ### Classic Mac OS
>
> Classic Mac OS applications will run on DP2--with some major caveats. Classic applications run in a kind of Mac OS 9.x virtual machine implementation known as the "Blue Box", and behave exactly as if they were running in classic Mac OS, warts and all. You can not, however, develop classic Mac OS applications with the development tools included in DP2. Developing classic Mac OS apps using a development tool like CodeWarrior that is itself running inside the Blue Box is theoretically possible, but certainly not something developers are likely to be interested in. It's very clear that the classic Mac OS API is being shown the door, albeit politely.
>
> ### Carbon

Carbon is a revision of the classic Mac OS API that eliminates or changes any functions that do not lend themselves to implementation in a modern, memory-protected, preemptive multitasking environment. Basically, all the dead wood was cleaned out of the classic Mac OS API and replaced with sturdy new oak. Most classic Mac OS applications need only minor revisions to become "Carbonized." Applications written to the Carbon API enjoy all the important benefits of Mac OS X: preemptive multitasking, memory protection, Mach's sophisticated virtual memory implementation, etc.

The genius of Apple's strategy is that Carbon applications *also* run in Mac OS 9.x where they behave just like any other classic Mac OS applications--meaning no memory protection, no preemptive multitasking, and so on. Hence, Carbon is the transitional API for Mac OS developers on both sides of the fence: OS 9.x and OS X. Making the next major revision of your application Carbon- compliant allows you to sell it to both Mac OS 9.x users and Mac OS X users. Expect this to be the most popular development API for Mac OS X in the near future.

Cocoa

Previously known as the "Yellow Box", and as the OpenStep APIs before that, Cocoa is the most modern

API in Mac OS X. The name change from Yellow Box to Cocoa is yet another horrible computer industry pun centered around the Java programming language. It's meant to highlight the fact that all of the Yellow Box APIs are now accessible via Java as well as Objective C.

Cocoa is NEXTSTEP's native API updated for the modern world and made accessible via Java. As any old NEXTSTEP developer will tell you (at length) if given the chance, NEXTSTEP had technology in the 1980's that's just beginning to appear in mainstream computing today: object reuse, sophisticated message passing, network transparency, runtime binding, clean separation of the UI from the "business logic", and platform independence. Expect the Cocoa APIs to be used mostly by ex-NEXTSTEP developers in the short term, with its long-term prospects still up in the air.

Java

DP2 includes JDK 1.2 and the latest revision of Symantec's Just-In-Time compiler.

BSD 4.4

The default installation of DP2 includes the BSD 4.4 environment: everything from the traditional Unix directory structure to the C libraries and command line

> tools. It is unlikely that the full BSD environment will installed by default in the official release of Mac OS X, but the APIs themselves will be present.

For those keeping score, that's five APIs (Classic, Carbon, Cocoa, JDK, BSD) and four languages (C, C++, Objective C, Java). Developers targeting the Mac platform certainly don't lack options. But the array of development choices pale in comparison to current schizophrenic nature of user interface.

User Experience

Getting all of the technologies described earlier to live in harmony in a single operating environment is no easy task. DP2 barely makes an effort in this regard (rumor has it that Apple is keeping the final UI under wraps for now, choosing to expose to developers only the bare essentials of the core OS in DP2), but hints about the future are everywhere. Unfortunately, not all signs point to success...

Mac OS X Server runs classic Mac OS apps via the Blue Box: a virtual machine that runs an actual copy of Mac OS 8.x. The user experience is not unlike connecting two computers to the same monitor. Switching from Mac OS 8.x in Blue Box to Mac OS X Server's UI is a *full-screen affair* in which the current environment takes over the screen completely.

Mac OS X promises a "transparent" Blue Box, but in DP2 it seems that "transparent" is merely a description of the desktop background, not the user experience. DP2 includes the same Mac OS virtual machine application used in Mac OS X Server ("MacOS.app"), but it also includes a newer version ("Classic.app") that is essentially MacOS.app with the desktop background rendered invisible.

What this means is that DP2 has to go through the same rigmarole of installing a disk image containing an actual, complete copy of classic Mac OS (9.0 in DP2), *launching* Mac OS from that image, (complete with startup screen, the marching line of extensions, etc.), and then running classic applications within that environment. The Mac OS 9.0 disk image installation only has to happen once. The classic Mac OS startup process has to happen *once per reboot*. Once Classic.app is running, classic Mac OS applications appear to coexist peacefully on the same screen with native apps in DP2. From a convenience standpoint, you can see why the Blue Box is a stopgap measure, and not something that Apple plans to rely on for long.

A closer look reveals some major quirks in the Blue Box implementation. First and foremost, classic apps use the classic Mac OS menu bar and apple menu when they're in the foreground. All your old friends are there: Calculator, Chooser, the Control Panels folder, and so on. Items installed in the Mac OS 9.0 System Folder modify the menu bar as usual: the Mac OS 9.0 clock is there, "system wide" menus created by third-party extensions appear, all exactly as you'd expect in Mac OS 9.0.

But switch to an application running outside the classic environment and you're in for a shock. Gone are any alterations to the menu bar created by the classic environment. Gone is the familiar Apple menu, replaced by Mac OS X's very different and totally unrelated Apple menu. Even the traditional rounded menu bar corners are gone, replaced by square edges.

The following screenshots illustrate the differences. The applications being run are the AppleScript "Script Editor" application included with Mac OS 9.0, and the native Mac OS X port of "Script Editor" which is included in DP2. I don't think anyone would have trouble differentiating even these two (ostensibly) identical applications running in the two environments.

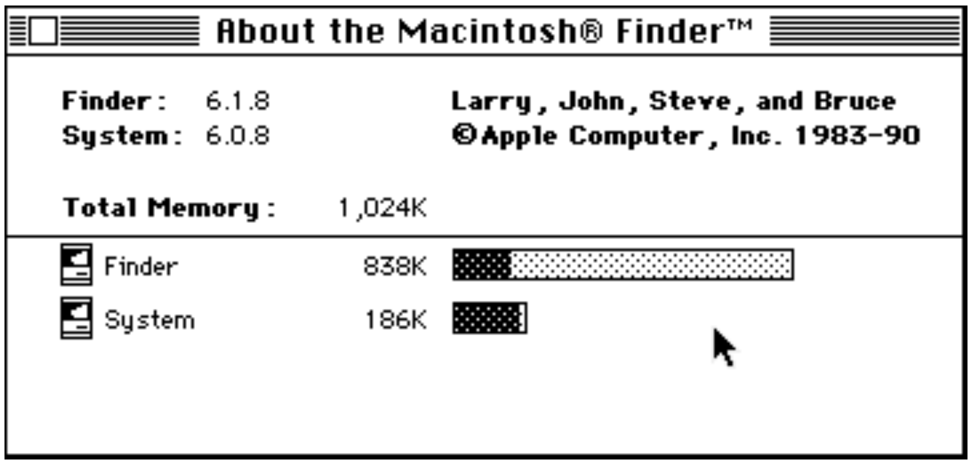
Rounding off menu bar corners is the least of Mac OS X's UI problems. The entire backwards compatibility strategy of running classic apps on an actual installation of classic Mac OS precludes the level of elegance users have come to expect from Mac OS. As jarring as the magically-changing menu bar is, I'm hard-pressed to come up with a more viable alternative that does not break classic apps. Classic applications simply expect to be in a traditional Mac OS environment right down to the smallest detail. MacOS.app and Classic.app provide this environment, but the price for the ultimate in backwards compatibility is a severe fracture in the UI.

In my estimation, this is far and away the most serious problem facing Mac OS X, from both a technological and a marketing standpoint. In the long run, classic applications will give way to Carbonized versions which do not suffer from any of these UI problems. But how long will that take? Will it keep users from upgrading to Mac OS X when its released? Or is the strategy not to release Mac OS X until every application is Carbonized? With Mac OS X's ever-changing release schedule, it often seems that way.

On a brighter note, many things that were once thought to be big stumbling blocks for classic compatibility have turned out better than expected.

Performance for classic apps is completely acceptable, subjectively. Other reviews have benchmarked the classic environment at 80%-95% of the speed of native Mac OS 9.0, and I'm sure it will only get better as development continues. This highlights the fact that the classic environment is not an emulation layer at all in the sense that it does not translate executable code from one instruction set to another. Classic apps run directly on the PowerPC processor very much as they would in native Mac OS 9.0, but with the "dangerous" operations (memory allocation, addressing, etc.) going through another layer of abstraction. Classic.app and MacOS.app trap such calls and service them behind the scenes using the modern Mac OS X core. Classic apps (and Mac OS 9.0 itself) are none the wiser, and every bit of code executing is native PowerPC. The speed hit comes from the overhead of this extra bit of abstraction.

Compatibility is reasonably good at this stage in development, with most applications running without a hitch. Even the "vMac" Mac Plus emulator ran just fine (see screenshot), providing an opportunity to run a Mac OS emulator inside a Mac OS virtual machine inside a Mac OS application. Games, not surprisingly, are the most problematic. Many refused to even launch in DP2, citing errors in initializing everything from Sound Sprockets to OpenGL.


About the Macintosh® Finder™

Finder: 6.1.8
System: 6.0.8

Larry, John, Steve, and Bruce
©Apple Computer, Inc. 1983-90

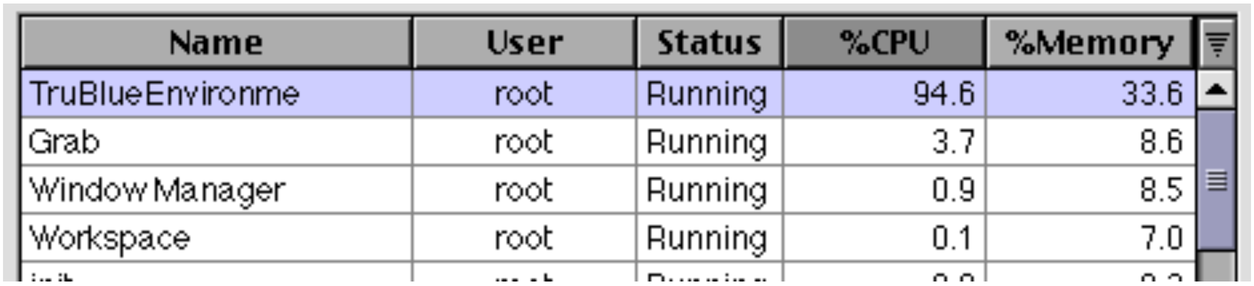
Total Memory: 1,024K

Finder	838K	
System	186K	

System 6.0.8 inside Mac OS 9.0 inside Mac OS X DP2.

Stability in the Blue Box (transparent or otherwise) is exactly what you'd expect from any other installation of Mac OS 9.0. Yes, that means that a single classic application can take down every other classic application and even Classic.app itself, but the core of Mac OS X and any non-classic apps are completely safe. Crashing Classic.app is actually harder than it sounds, but freezing it into an unresponsive state is not difficult if you, say, insist on repeatedly trying to launch games from 1988 that expect to directly access the hardware. But redemption is only a "kill" command away (or available via a few clicks on the GUI process manager, for the less CLI-inclined).

Speaking of process management, have a look at the stats for Classic.app (the process name is "TrueBlueEnvironment"):



Name	User	Status	%CPU	%Memory
TruBlueEnvironme	root	Running	94.6	33.6
Grab	root	Running	3.7	8.6
Window Manager	root	Running	0.9	8.5
Workspace	root	Running	0.1	7.0

Oink! Oink! goes the Blue Box. Click to expand

The Blue Box is, to put it bluntly, a total resource pig. And no, that virtual memory size in the expanded screenshot is not a typo. Classic applications run inside what they think is Mac OS 9.0 computer with a gigabyte of real RAM. Classic.app happily snags a virtual chunk of memory that size and lets Mach worry about the details. Incidentally, the screenshot above is from a Mac G3/400 with 256MB of RAM.

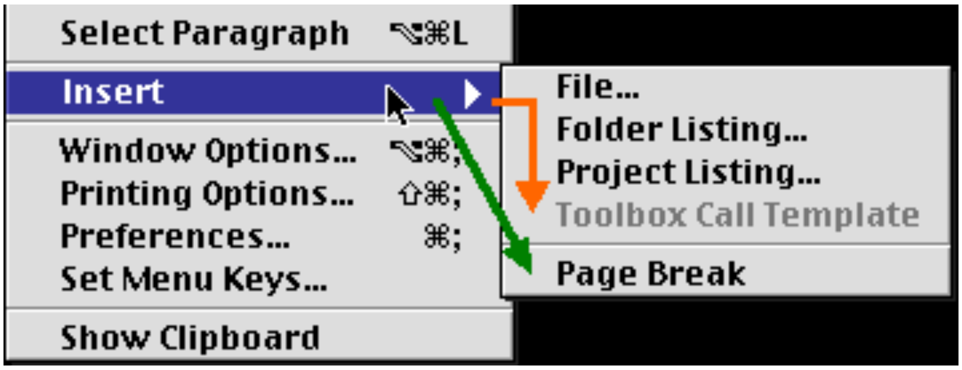
As insane as the resource requirements appear to be, having the classic environment idling in the background does not make a dent in the responsiveness of the OS. Perhaps on a machine with less real RAM it would create more of a problem, but having all of Mac OS 9.0 corralled into a single, well-behaved process within the larger Mac OS X environment really does pay off.

Native applications

Native Mac OS X applications, be they Carbon, Cocoa, Java, or BSD, are nice and snappy. Cosmetic problems abound, but stability is quite good for a developer release. The newer apps are the flakiest. The newly Carbonized Finder-like thingie (it's not the Finder Mac OS users know and love, nor is it the Finder that will ship with Mac OS X which Apple is still keeping close to its vest) is the most crash prone, almost gleefully dying and then restarting in the background while you do other things.

There is very little consistency in the UI widgets at this point. Carbon application widgets (menus, buttons, windows) look and behave exactly like classic Mac OS widgets. Cocoa widgets look and behave like Mac OS X Server widgets, which are basically NEXTSTEP widgets repainted to look sort of like Mac OS. A particularly annoying example of these foreign widgets dressed up in Mac OS clothing is the behavior of hierarchical menus in Cocoa.

In classic Mac OS, the green path in the screenshot below is a perfectly valid way to select "Page Break" from the sub-menu. Experienced Mac users do this quickly, and without thinking. Try that same move in NEXTSTEP, and consequently in Mac OS X Server and Mac OS X DP2, and it won't work. The orange path is the only way. It's this kind of attention to detail that distinguishes the classic Mac OS UI, and its presence in Mac OS X is badly needed if it is to ever "feel" like Mac OS.



Green: Mac OS. Orange: Cocoa.

Cocoa applications also support the (un)usual array of NeXT-like functionality including universal tear-off menus, NeXT-like window resizing widgets, and so on--not particularly offensive, but not very Mac-like either.

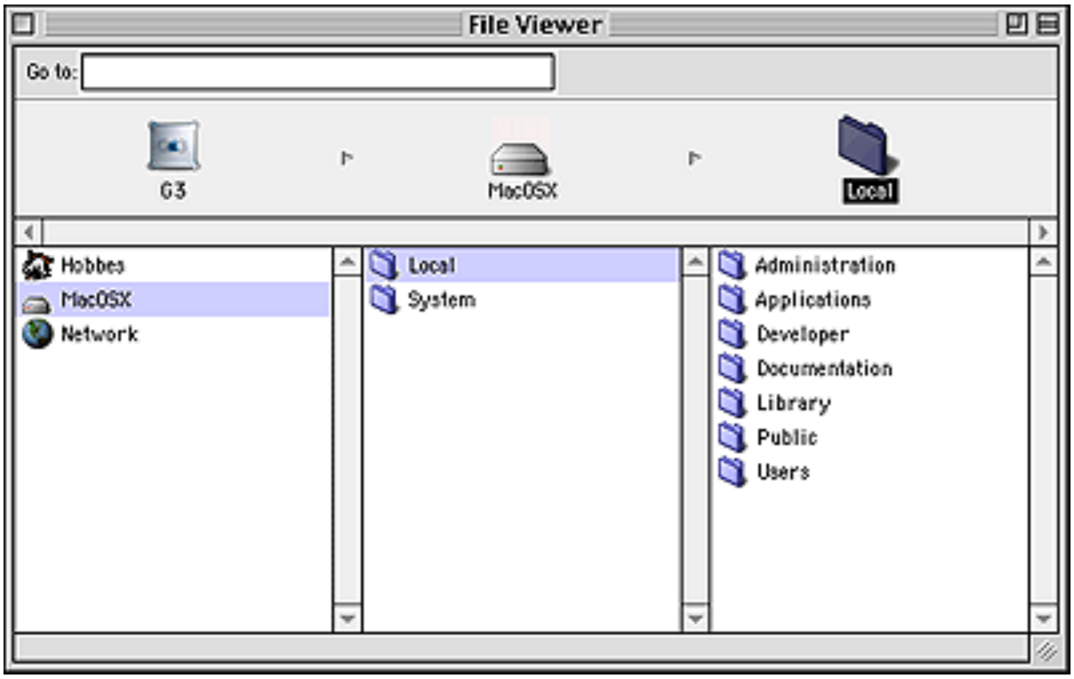
All native application windows enjoy opaque dragging. In informal "shake the window around while compiling and linking an application in ProjectBuilder" testing, screen redraw in DP2 feels a lot more responsive than the same feature in Mac OS X Server. It's apparent that some form of hardware video acceleration has been incorporated into DP2, but the actual presence of "Quartz" (Apple's next-generation low-level graphics library) is harder to pinpoint. Some things suggest its presence, like the translucent drag-select on the desktop (see screenshot) and the strange ability to rotate PDFs in real-time in DP2's "ViewPDF" app. These seem like things that no one would bother implementing by hand in DP2 when they're bound to be trivial to accomplish via Quartz.

Java applications behave, well, like Java applications. It's difficult to gauge Java performance when so much depends on the particulars of the JVM implementation, and DP2 is still using Symantec's venerable JIT. Java applications built on the Cocoa APIs are nearly indistinguishable from other Cocoa applications, but I'm not sure why anyone but a die-hard Java devotee would build an application with a cross-platform language like Java that can only run on a single platform--Mac OS X being the only place where Cocoa is available. (More on the cross-platform puzzle later.)

File system layout

Like Unix, and unlike every version of Mac OS ever made, Mac OS X has a single file system root shared by all mounted volumes. In DP2, it works exactly as you'd expect in Unix. Volumes can be unmounted and remounted at different mount points via the usual "mount" and "umount" commands, the whole nine yards.

By default, all connected volumes are mounted at "/<volume name>" at startup. This lends itself to a close approximation of the traditional Mac OS volume handling through the Finder-like interface. There is a new "root item" that is the conceptual parent for the entire file system: the aptly-named "G3" as seen in the NeXT-style file viewer in the screenshot below. (Expect this icon and name to be easily customizable in Mac OS X. In DP2, it is not.)



File viewer showing the "G3" conceptual file system root.

Beneath "G3" are all the mounted volumes (just "Hobbes" in this case), the disk Mac OS X is installed on ("MacOSX"), and the round globe icon of "Network" (which is actually just a link to "/Network", a directory on the "MacOSX" volume).

The Finder-like interface is very selective about what it shows, and I suspect the actual Mac OS X Finder will be equally selective. For example, the "Network" directory linked to the globe icon under "G3" is not shown at all inside the "MacOSX" volume even though that's where the directory actually is. Similarly, the BSD directories (/usr, /bin, /dev, /etc) on the "MacOSX" volume aren't shown at all.

Directory structure abstraction will be even easier in a Mac OS X Finder that shows mounted volumes on the desktop (or on the long-rumored Mac OS X "tray") like traditional Mac OS. (The Finder-like interface in DP2 does not do this.) In fact, some form of abstraction is necessary because Mac OS X is a multi-user OS. From a normal user's perspective, his world starts in his home directory. He will have access to the entire file tree via the "G3"-rooted abstraction shown above, but the only directory he's likely to have write permissions for is his home directory.

As expected, the whole thorny reality of the file system is open to anyone using the BSD command line interface. Just fire up Terminal.app (which launches the tcsh shell by default) and satisfy your curiosity. It's quite an odd mix of traditional Unix directories (/usr, /tmp, /bin) and the mostly title-cased Mac OS X directories like /Local, /System, and /Library.

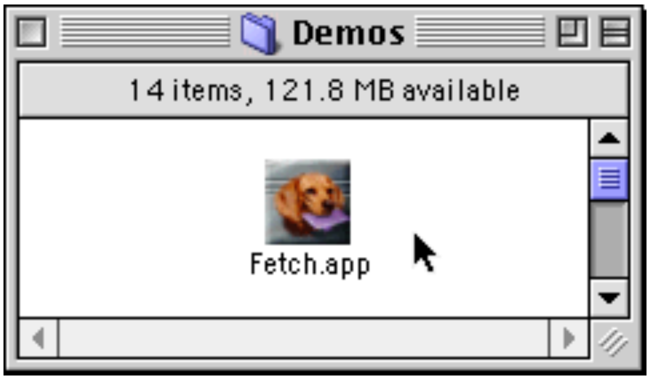
The radical new file system layout (as compared to classic Mac OS) does not present much of a problem, in my opinion. It's trivial to create an environment in which Mac users will feel comfortable through the judicious use of symbolic links and content hiding. Only geeks need be aware of the file system complexities lurking below. That being said, DP2 does not achieve anywhere near this level of comfort for the average user, and any release of Mac OS X that expects to do well with Mac OS customers had better make sure it does.

File system metainformation

Install DP2 on an HFS+ volume and as expected, you get full support for classic Mac OS files and applications with resource forks. Carbon applications also use resource forks, making it clear that the default file system for Mac OS X will end up being HFS+ if only to ensure the promised ability to sell the same Carbon applications to both Mac OS 9.x and Mac OS X users.

But the future of meta-information in Mac OS X is less clear. Resource forks remain for the sake of the transitional API, Carbon. But even Carbon apps are moving away from the traditional Mac OS multi-fork concept and into the realm of NeXT-derived "application wrappers" (also called "bundles" in NEXTSTEP and "packages" in Mac OS 9 and Mac OS X). Put simply, a package is a structured collection of files and directories that is treated specially by the high-level UI. In DP2, packages are identified by a ".app" extension. In Mac OS 9, packages are identified by a particular Finder flag.

Let's look at the Carbonized version of the venerable Mac OS FTP application "Fetch" that's included with DP2. From DP2's Finder-like interface, it looks like this:



Fetch.app launches and runs as expected with a simple double-click. But from the command line, it looks like what it really is: a directory with a bunch of other files and directories inside it. The contents look like this:

```
Fetch.app/
Fetch.app/Support Files/Executables/Mac OS X/Fetc
Fetch.app/Support Files/Info-macosx.plist
Fetch.app/Support Files/PkgInfo
Fetch.app/Support Files/Resources/Non-localized R
```

```
  Fetch.app/Support Files/Resources/Non-localized R
  Fetch.app/Support Files/Resources/Non-localized R
  Fetch.app/Support Files/Resources/Non-localized R
  Fetch.app/Support Files/Resources/Non-localized R
```

The business end of the package is the "Executables" directory, which is further subdivided by operating system (Hmmm...). The file "Fetch" at the tail end of that directory is much like a classic Mac OS application. It has a resource fork filled with pretty much the same stuff as the classic version of Fetch. It has a file type of "APPL". It launches the application when you run it, even from the command line.

So why go through the whole charade of wrapping our little application in this structure? The answer lies in the contents of the other files inside the package, and the future direction of meta-information in Mac OS X. Let's look at the other files in the package more closely.

The file "Info-macosx.plist" is actually an XML property list ("plist") containing a lot of the same meta-information traditionally stored in the classic Mac OS resource fork. Here's a small snippet:

```
  <?xml
  version="1.0" encoding="UTF-8"?>
  <!DOCTYPE plist SYSTEM "file://localhost/System/L
  <plist version="0.9">
```

```
<dict>
        <key>CFBundlePackageType</key>
        <string>APPL</string>
        <key>CFBundleShortVersionString</key>
        <string>Fetch Application
3.0.3, Carbonized</string>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleSignature</key>
        <string>FTCh</string>
        <key>CFBundleGetInfoString</key>
        <string>3.0.3, Copyright 1997
Trustees of Dartmouth College</string>
        <key>CFBundleName</key>
        <string>Fetch</string>
        <key>CFBundleVersion</key>
        <string>3.0.3d0</string>
```

From this we learn a few things. To start, it appears that the XML document type definition ("DTD") is stored locally, allowing for easy changes to the format of this file (unlike changes to the format of, say, Finder meta-information in classic Mac OS. Any changes to that structure require major OS revisions!) But the DTD location is given as a URL, opening the door for it to be located anywhere in the world.

Next, we see the use of the old NeXT term "bundle" in the key names. The "CF" prefix refers to Apple's "Core Foundation" libraries which provide basic resources like string classes, collections, and dictionaries to higher-level Mac OS X APIs like Carbon and Cocoa.

Finally, traditional Mac OS meta-information is easily recognizable in the plist file: type APPL, creator FTCh, version 3.0.3d0, language English, and so on. This file goes on for about 100 lines and includes many more key/value pairs and other deeper structures. It's not clear whether or not Mac OS X is reading the .plist file or the Fetch executable's resource fork. Mac OS 9, at least only needs the Fetch executable itself to run Fetch.app.

Peeking inside the "PkgInfo" file reveals yet another duplication of meta-information, but this time just the essentials: type and creator. It simply contains the text "APPLFTch"

The rest of the files are icons: a NeXT-style TIFF and a bunch of Mac OS 8.5/9.0 style "icns" resources. Again, the icns resources are duplicated within the Fetch executable's resource fork.

Now let's take a look at a Cocoa application package: Chess.app

```
Chess.app/
Chess.app/Support Files/Executables/Mac OS X/Ches
Chess.app/Support Files/Info-macosx.plist
```

```
  Chess.app/Support Files/PkgInfo
  Chess.app/Support Files/Resources/English.lproj/.
  Chess.app/Support Files/Resources/English.lproj/3
  ...bunch of TIFF files snipped...
  Chess.app/Support Files/Resources/English.lproj/C
  Chess.app/Support Files/Resources/English.lproj/C
  Chess.app/Support Files/Resources/English.lproj/C
  Chess.app/Support Files/Resources/English.lproj/L
  Chess.app/Support Files/Resources/Non-localized R
  Chess.app/Support Files/Resources/Non-localized R
  Chess.app/Support Files/Resources/Non-localized R
  Chess.app/Support Files/Resources/Non-localized R
  Chess.app/Support Files/Resources/Non-localized R
  Chess.app/Support Files/Resources/Non-localized R
```

Every file in this package is a simple, flat file, including the "Chess" executable itself (unlike Fetch.app's executable with its resource fork). In place of the Mac OS resource fork, there are various ".nib" files (inside a "Chess.nib" directory). These files (created by Interface Builder, the NeXT-derived GUI layout tool) contain all the interface data usually stored in Mac OS resource forks. "classes.nib" is a plain text file in a proprietary (but very simple to understand) format that defines the attributes for each interface object class. The "obects.nib" file is the binary data for the interface objects themselves. I've heard talk of Apple changing the ".nib" file format to XML, but this change is not present in DP2.

The rest of the files in the package: images, icons (in two formats), sounds, the required GNU license file ("COPYING"), some miscellaneous plain text data files, and even the application credits in the form of an RTF file. The XML "plist" and the simple "PkgInfo" files are also present.

Packaged Future

The future direction of meta-information in Mac OS applications is clear: packages. (...or "bundles", or "application wrappers", depending on who you ask. But "packages" is the official Apple name as of the release of Mac OS 9.0.) Resource forks will stick around for a while, but their long-term future is uncertain.

And yes, for those not following along, the upshot of all this package business is that Mac OS X applications like the Chess.app will travel across any foreign file system with no loss of meta-information or resources. Tar it, zip it, FTP it, it'll always stay perfectly intact. Hurray for technology NeXT had in the 1980's!

Before I leave the topic of meta-information, there's one small wrinkle in this packages-scheme. Earlier I mentioned that the higher-level Finder-like UI in DP2 recognizes packages by the ".app" extension and shows them as normal applications (see the screenshot of Fetch.app above) instead of showing them as what they really are: directories. I also mentioned that packages in Mac OS 9 are identified by a particular Finder flag. But Finder flags are file system dependent. Copy a file from an HFS or HFS+ volume to a FAT or UFS file system and those Finder flags are gone.

This is part of the motivation for the concept of packages, of course, but without the very thing that packages are meant to eliminate (HFS/HFS+ dependent meta-information), how will packages be identified in Mac OS 9 or the release of Mac OS X? Is that ".app" extension really going to stick around? File extensions have never been a part of the Mac OS user experience, so even if they do stick around in Mac OS X, expect them to be hidden by the higher-level UI. In Mac OS 9, I suspect the Finder flag will stick around. This makes sense since Carbon apps have resource forks in their executables--another feature that requires HFS+ or HFS. So the ability to store Finder flags is essentially guaranteed in Mac OS 9.

File name extensions in Mac OS X are a much fuzzier subject. Every file in classic Mac OS has a type and creator, but only packages in Mac OS X have this meta-information. Will every file in Mac OS X be a package? Not likely, but possible. If not, how will Mac OS X identify the file type and creator of "regular" files? By file name extension, that concept so alien to traditional Mac OS? Or will HFS/HFS+-dependent type/creator meta-information solider on into the future? Time will tell.

The big picture

Actually using DP2 is akin to logging into a demented Xterm running a poorly designed window manager theme meant to look something like Mac OS. Launch a Cocoa application and you feel like you've been warped into NEXTSTEP, again running that funny window manager. Run a classic applications and it's like being in a slightly odd version of Mac OS 9, with that alternate NeXT universe still visible in the background. Pull up the command line and you start to think that all of this is one big facade running on top of good old Unix.

The reality, of course, is all of the above and none of the above. But aside from the bugs, I think I'd much rather be stuck using Mac OS X DP2 on a daily basis than Mac OS X Server. They both completely fail the "Mac-like" litmus test, but DP2 is closer to that goal. It also feels faster and is more functional for non-server activities due to its new Carbonized Finder-like interface.

Yes, it really is a modern, buzzword-compliant OS. Yes, things really do continue to happen when you hold down the mouse. No, Mac users won't care about any of this until the user experience doesn't suck.

What is Mac OS X DP2? Revisited...

Mac OS X DP2 is big, complex, and confusing. It's for developers only. Normal users will find its organization Byzantine, and its UI a mess. It screams "work in progress" from every corner. But it's also a very interesting window into the future of Mac OS...a dark, scary future featuring file name extensions, a command line, and hierarchical menus that don't work quite right. Heh, just kidding about that last part. Maybe.

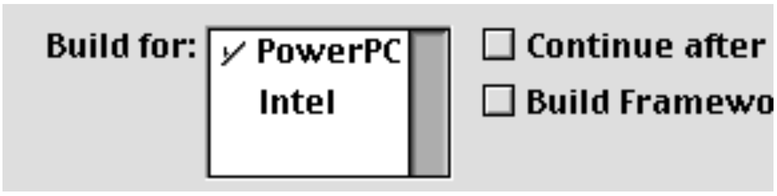
Epilogue: The Cross Platform Wildcard

The OpenStep APIs are cross platform. Mach is cross-platform. WebObjects is cross-platform. x86 builds of Rhapsody, Mac OS X Server, and Mac OS X inside Apple have been all but confirmed. Rumor has it that Apple routinely synchronizes all changes to Mac OS X across both PowerPC and x86 builds of the OS. Clearly, Apple's choice of where to deploy its new operating system is not limited by the technology. If they decided to try releasing a version Mac OS X for x86 processors, it would be technologically within their means. But will they do it? I seriously doubt it. Still, it's fun to look for cross-platform clues within Mac OS X DP2. Here's a sample:

- There's the "i386" directory in the BSD library tree, but that could just be a BSD thing.
- Then there's the fact that the "Executables" directory

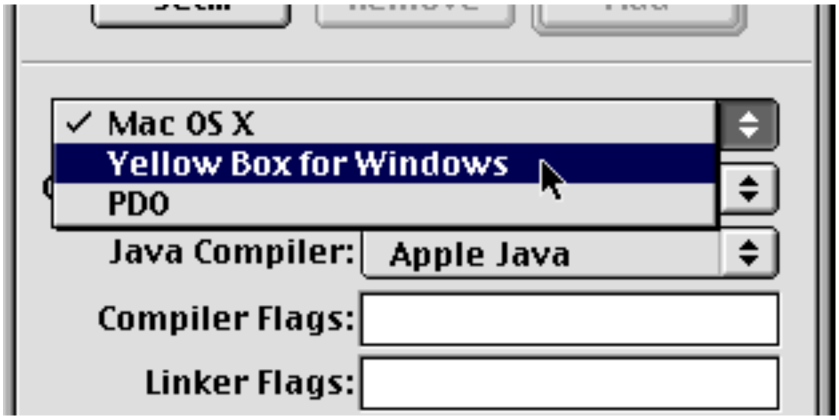
inside Mac OS 9 and Mac OS X "packages" are subdivided by operating system--an organization that doesn't make much sense if packages only need a single PowerPC-native executable.

- Many example applications include ".ico" icon files in addition to TIFFs and Mac OS "icns" resources.
- Here's part of the "build options" window from the ProjectBuilder development tool:



Build for: ✓ PowerPC
Intel

- And finally, here's part of the "project options" window for Cocoa apps in ProjectBuilder:



✓ Mac OS X
Yellow Box for Windows
PDO
Java Compiler: Apple Java
Compiler Flags:
Linker Flags:

This is nothing new to ex-NeXT developers. One of the great strengths of the OpenStep APIs was their ability to target multiple platforms. The same applications built with the OpenStep APIs could run on both NEXTSTEP and Windows (provided the OpenStep libraries were also included with the app). These libraries, also known as the "Yellow Box for

Windows", were to be freely distributable, enabling Yellow Box developers to sell to both Mac OS X and Windows customers. But Yellow Box for Windows is missing in action these days, and Apple either waffles or declines to comment when pressed on the issue.

As Mac OS X marches towards beta and then release, these x86 vestiges will probably start to disappear. The cross-platform card is something to watch for, however. For the first time, the only thing keeping Apple off of the "PC" platform will be its business plan. And hey, with Steve Jobs calling the shots, anything is possible.

John Siracusa *Associate writer*

John Siracusa has a B.S. in Computer Engineering from Boston University. He has been a Mac user since 1984, a Unix geek since 1993, and is a professional web developer and freelance technology writer.

0 Comments