MWJ The Weekly Journal for Serious Macintosh™ users

may 25, 2003

Signature Not Verified

MWJ

Digitally signed by GCSF Production
Date: 2003.06.02 08:10:48 -05'00'
Location: El Reno, OK

HFS and HFS plus complete

Where have we been? Assembling the longest, most complex **MacCyclopedia™** series ever published. It's high time someone bothered to explain HFS and HFS Plus to people other than file system designers and disk utility authors, and that's what we've set out to do. It's a large job because you have to understand many pieces: trees, special files, binary searches, clumps, extents, and more, so we take them one by one until we can build the big picture of the disk. HFS and HFS Plus are more similar than different, but we show you where the two diverge, and why. We also explain some other file systems, look at what HFS and HFS Plus get for being so complicated, and show you why these file systems were ahead of their times. Unless you owned a Macintosh before 1986, you've always used HFS and HFS Plus, and now you'll learn how they work. **Blocks laid bare, page 1.**

the HFS primer

everything you ever wanted to know, and maybe more

We've been looking at Alsoft's new DiskWarrior 3 release here at **MWJ** World Headquarters lately, and we think it's just nifty. Then something odd happened: we couldn't find the vocabulary to tell you why.

Everyone has the short version of what DiskWarrior does, similar to what Alsoft says on the product's Web page: "DiskWarrior is not a disk repair program in the conventional sense. Instead of patching the original directory, it uses a patent-pending technology to quickly build a new replacement directory using data recovered from the original directory, thereby recovering files and folders that you thought were lost and that no other program could recover."

So – what does that mean? How does one "build a new replacement directory," and why is that better than fixing problems with the existing directory? What does "optimizing" a directory mean on the Macintosh? These should be relatively simple questions, but they're not, because the two main disk formats on the Macintosh – HFS and HFS Plus – are shrouded in myth and intrigue, despite being well documented.

Simply put, HFS and HFS Plus use complicated data structures that puzzle many programmers, much less the average non-programmer. Answers to HFS questions may involve phrases like "thread record" or "extents overflow tree" that have no intuitive meaning. And yet nearly every Macintosh user boots from an HFS or HFS Plus disk. Even the Unix fans that are so fond of UFS in Mac OS X typically boot from HFS Plus because parts of Carbon still require it. HFS and HFS Plus are probably the least understood of the widely-used Macintosh technologies because they defy simple explanation.

We decided that shouldn't stop us.

The more we tried to explain parts of HFS and HFS Plus to put DiskWarrior in its proper context, the more we realized that people other than programmers could benefit from understanding how their disks are organized. These are complicated concepts, and they may take more than one reading to sink in. They certainly did for us. But as usual, when you're not worried about most of the byte-level details and look at the bigger picture, you can understand the basics of HFS and HFS Plus, as well as why they're so intricate.

What we explain in this issue won't teach you how to write a disk repair program, or how to fix damaged disks in a block editor. It should be enough for you to understand what disk repair programs do, how they do it, and how a disk is supposed to function. HFS and HFS Plus are

2 ▶

◀ 1

powerful file systems, and like every other file system, they're all about organizing data on disk, so our story begins with the disks themselves.

block party

Today's disks are designed to deliver lots of data fast. Need 30MB transferred every second to keep up with streaming video and audio? No problem: today's hard drives communicate through ATA and the PCI bus, in conjunction with the host computer, to deliver data via direct memory access, or *DMA*. Through this significant bit of serious voodoo magic, the hard drive winds up putting data into the RAM chips where it's wanted – no reading it into the kernel and copying it to an application address space or any of that garbage.

Unfortunately, sometimes it's about very little data. For example, since Apple changed applications to be packages instead of single files, there's no easy way to store the application's file type and creator type. Directories don't have such metadata, because every directory is the same "kind" – a directory – and was created by the operating system to hold other files. It's not like some directories really hold something else, like pixels, or maybe pudding. Apple's first recommendation was for each application package to store the information in a "`Contents/PkgInfo`" file that's exactly eight bytes long: the four-byte file type followed by the four-byte creator type. (Apple has since deprecated the `PkgInfo` file – it's still allowed but is no longer required, as file type and creator type information for applications must now be found in the package's `Info.plist` file.)

DMA is overkill to transfer eight bytes of data. In fact, that's one of the problems of disk storage: transfer rates aren't so bad once everything's going, but getting started is painful. The operating system has to figure out exactly where on disk the information is located, spin the drive up to speed if it was at rest, move the mechanical reading head to the appropriate position, wait for the right spot to come around, and read the data. Writing is just as bad, except slower.

counting blocks

It's plainly unacceptable to go through all this work just to pull eight bytes from disk, or worse, to pull four bytes from disk and then have to do it again for the *next* four bytes. If you're going to go to the disk for a few bytes, you might as well get a few more just in case you're going to need them, because the penalty for not doing so is just too great. In the days of 5.25-inch disks, the quantum was one *sector* on the disk – 256 bytes. The 3.5-inch disk raised that to 512 bytes, and thanks to both compatibility concerns and natural technical inertia, hard disk makers adopted the same quantum, typically called a *block*.

You have to work hard to find a Mac-compatible disk drive that can read less than 512 bytes at a time. If you ask the operating system for four bytes from a file, the device driver reads the entire 512-byte block containing those four bytes, but the OS returns only the four bytes you wanted. If you then ask for the next four bytes, the OS already has them buffered if they were in the same block. If not (perhaps the first four bytes were at the end of a block), the OS has to go back to the disk to get the next block of the file. That's why many of today's hard disks actually read more than the operating system requests, caching it on the disk controller in a megabyte or two of fast cache RAM – even if the OS only requests one block, the disk might read several blocks at once, just in case the OS isn't smart enough to make the best use of the drive's capabilities.

Even so, the disk quantum is the block – you can't read or write less than one block. Bytes 0 through 511 on the disk are in block 0, bytes 512 through 1023 are in block 1, and so on. Bytes 2 through 513 are in two blocks, even though that's only 512 bytes total: blocks start and end on multiples of 512 bytes on the disk. No matter how an operating system organizes a disk into files or partitions, the drivers that actually talk to the disk read and write exclusively in block multiples.

Most drivers today are pretty smart. If they know they want 10MB of contiguous data from the disk, they'll make one request for 20,480 blocks and let the drive and controller work DMA magic. The stupid way is to make 20,480 requests for one block each, introducing lots of unnecessary overhead. It's one of the rules of modern computer programming: ask for what you want, not what you think you can get. Let the OS surprise you if it can. You can always fall back to a more modest request, but most of the time, the OS does that for you.

partition erudition

As far as the disk itself and the ATA controller are concerned, a disk is nothing but one big pile of blocks. All modern Macintosh operating systems, however, treat a disk as more than that. The first several blocks of any modern Macintosh hard disk contain a *partition map* dividing the disk into discrete partitions. One such partition contains the partition map itself; others usually contain Mac OS 9 drivers for the hard disk (perhaps an ATA driver, perhaps a SCSI driver) that the operating system is free to use or ignore. One or more of the partitions contain what regular, non-nerd people consider to be a "disk" – a volume that shows up on the desktop or in Mac OS X's "`/Volumes`" directory.

You might partition an "80GB" hard disk (more like 74GB when formatted) into two 36GB volumes. In that case, you'd probably have partitions for the partition map, a few drivers, both 36GB volumes, and an "Apple Free" partition with the remaining 1GB to 2GB of space you didn't use anywhere else. The disk's drivers take care of these details, telling higher-level portions of the operating system that your particular piece of hardware is in fact, two "disks" of 36GB each.

In fact, if you're a total Unix nerd, you probably already know that under Mac OS X, "`/dev/disk0`" is your entire first hard disk (including the partition map), and "`/dev/disk0s1`" and other devices starting with "`disk0`" are the partitions on that disk, excluding the partition map itself. In our example, only two of those will show up as disks with mountable file systems. Rainer Brockerhoff's US$10 XRay can show you the Unix device for any mounted volume, if you're interested.

As far as anything higher-level than a device driver is concerned, each of these partitions is really a separate device, because the device driver said it's so. The first block of your primary HFS Plus partition might be block 16,384 on the disk, but it's "block 0" of the partition. When we refer to blocks on a "disk," we mean on a volume – a single partition of any hard disk that has a partition map, and nearly every hard disk does.

a simple file system

Once all the driver and volume mess is resolved, there remains the question of how data is organized on the disk. If the OS can't keep track of the blocks that a file occupies on the disk, it can't ever find the file again, and most people don't want write-only disks. The data of each file – or of each file's fork – is typically stored as the entire contents of a series of blocks on disk. That is, a 2048-byte file takes the entire space of four blocks on disk, with no extra structure or file system information stored within those four blocks.

Some file systems, including the hot new Linux-friendly ReiserFS, try to take advantage of unused space in a block, such as by storing a 400-byte file and an 8-byte file in the same 512-byte block. So far, Apple's file systems are not among these. In today's economy, a few hundred bytes of hard disk space are much cheaper than the thousands of hours of writing and debugging such byte-wringing file systems require, not to mention the bills for headache pills and padded rooms for overworked programmers. The ReiserFS folks deserve credit for getting it done, because it's no small task.

Some of Apple's file systems have been elegant in their simplicity and limitations. The ProDOS file system was a stalwart of the Apple II line from 1984 onward. It originally appeared as the SOS file system for the Apple III in 1981, but the Apple II version had a lot more users, so more people know them as "ProDOS disks" than as "SOS disks."

ProDOS's organization is deceptively simple. Each file's directory entry contained a *storage type* and a pointer to the file's *key block*. A file less than 512 bytes long needs only one disk block, so the key block for such files points to the one and only data block. Most files need more than one block, and in those cases, the key block points to an *index block* that, in turn, contains the block numbers of the file's data blocks. If a program needs data from the tenth block of the file, the operating system goes to the index block and finds the tenth entry in it – the block number of the tenth block of the file. The algorithm is so easy you could duplicate it in a few lines of AppleScript.

ProDOS used two-byte block numbers, so each 512-byte index block can point to 256 data blocks. That's enough to accommodate a file of 128KB, which wasn't bad in days of 140KB floppy disks. If you needed a larger file, ProDOS added one more level of indirection: the key block became a *master key block* that pointed to up to 256 index blocks, each of which pointed to up to 256 data blocks. That was enough for a 32MB file, but since ProDOS elsewhere imposed a three-byte limit on file lengths, no file could be larger than 16MB anyway. The storage type field in each directory entry told ProDOS which of these three formats the file used, from a small file with no index block to a huge file with a master index block.

4 ▶

◀ 3

Without that hint, the operating system would have no idea what the key block meant.

Directories used a linked list instead of the tree-like structure employed for data files. Each directory block contained exactly thirteen directory entries, with the very first entry holding information about the directory itself. Each directory block also held the block numbers for the next and previous blocks in that directory. The volume's root directory *always* occupied blocks 2 through 6, so any code reading a ProDOS disk always knew how to find any file: start at block 2, find each subdirectory entry, follow it to that subdirectory, and repeat until you find the entry for the file itself. (Useless trivia: block 0 on a ProDOS disk contains Apple II boot code; block 1 typically contained Apple III boot code.)

The ProDOS file system is simple enough for non-programmers to understand, though not necessarily on first reading, but it's also limited. Individual files are capped at 16MB, as mentioned, and ProDOS disks can be no larger than 32MB. That's because every block number had to fit in two bytes, allowing a maximum of 65,536 blocks at 512 bytes (0.5KB) each, a total of 32MB. Directory entries were also fairly small, permitting only fifteen character file names with a one-byte file type and two-byte "auxiliary type" that Apple later had to use as an extension of the file type. There was no room for non-ASCII names, creator types, creation or modification times more specific than the nearest minute, or any of those finer points of modern disk life.

Also, since the volume directory was limited to four blocks of 13 entries each (minus one to describe the volume itself), the root directory was strictly limited to 51 entries. The root directory could not grow because block 6, the next block, was reserved for the start of the *volume bitmap* – a series of up to sixteen blocks (65,536 bits) with one bit for each block on the disk. If the bit corresponding to a given block is 1, the block is in use; if the bit is 0, the block is available for other use. It's how ProDOS knows where to find free blocks for use in new and growing files, but its absolute position hamstrings the volume directory at 51 entries. Imagine trying to live with that restriction today.

enter HFS

The engineers who designed the file system for Apple's next big computer, the Macintosh, were determined to use their larger 3.5-inch disks to make a bigger and better file system. Longer file names, more metadata for the system's use, more flexibility, the whole nine yards. Their first attempt was not entirely sufficient: MFS, the Macintosh Filing System, was a flat-file system with significant limitations. Although you could create "folders" in the Finder, they weren't really folders: every file on an MFS disk was really stored at the root level. The Finder and the "Open…" and "Save As…" dialog boxes faked the presentation of folders. It didn't take two years before Apple realized that MFS would be way too slow. It's best not to discuss MFS too much; it's a touchy subject in some quarters.

Such lessons did inform engineering decisions for the next Macintosh file system, the *Hierarchical File System*, better known as *HFS*.

Take that bit about directories, for example. ProDOS directories are simple lists of linked blocks, easy to understand but hard to use. If you want to open a file in a directory containing 4000 entries, a ProDOS-compatible operating system would have to look at every directory entry and see if it was for the file in question. If it finds the file, it opens it. If not, it keeps going until it runs out of directory entries. Again, that's quite simple, but examining an average of 2000 entries before finding any file, or examining 4000 entries before returning "file not found," is way too painful for everyday use.

It doesn't have to be that way. Suppose you're asked to guess a pre-determined number between 1 and 4000. As long as the other person agrees to tell you if your guess is correct, too high, or too low, you can find the correct number in no more than twelve guesses – every time. You may already know how, too. Make your first guess exactly halfway in the available range, and each time you're wrong, discard the half of the range you know does not contain the answer. If you guess 2000 and you're told it's too high, you know the answer isn't between 2001 and 4000, so throw that half away. Then guess halfway in the new range – at 1000 – and repeat until you get it right.

Because each guess discards half of the available choices, finding the right answer from *N* options takes, at most, $\log_2 N$ attempts, rounded up to the nearest integer. For a sample of 4000 options, $\log_2 4000 = 11.97$, so you can always get the number in 12 attempts. Without the "higher" or "lower" hints, finding a random number out of 4000 would, when averaged over time, take about 2000 guesses. The difference between 2000 tries and 12 tries to find a file in a directory is a huge win if it can be done.

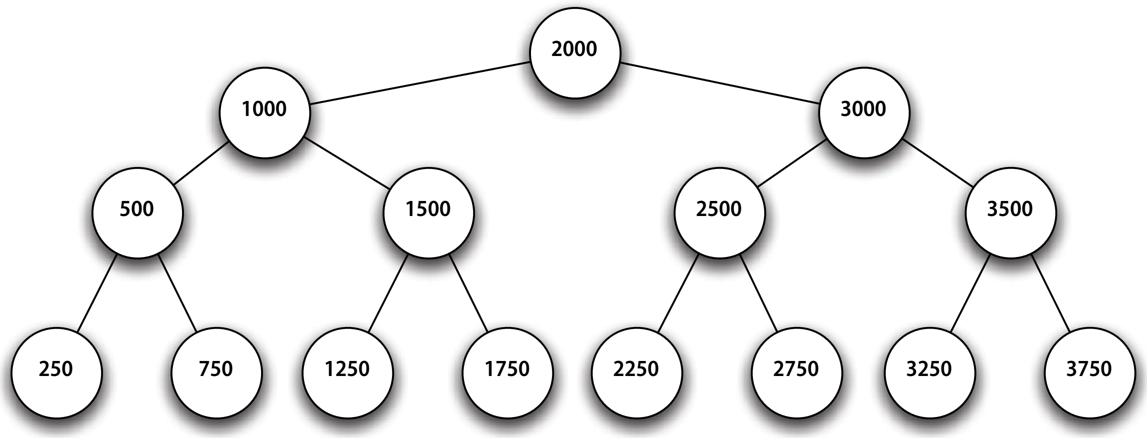

FIGURE 1 – THE "GUESS THE NUMBER" TREE

This algorithm is called a *binary search*, and to make it work, you have to know if any directory entry you look at is "higher" or "lower" than the one you want. That's why it can't work with ProDOS, where the directory entries are in no specific order. The operating system has to know the order of files in a directory or it can't judge whether the sought-after file should come "before" or "after" the entry it's examining, the logical equivalent to "higher" or "lower." Basically, the directory must be sorted, at least enough to be able to read it in order.

Sorting a ProDOS-style directory is out of the question: adding a file whose name starts with "M" to the middle of a 4000-entry directory would mean moving 2000 entries "down" by one to make room for the new file, and that's *worse* than 2000 searches. Apple's HFS designers solved this problem by storing the catalog in a data structure specifically designed for binary searches.

the joy of trees

A classic *binary tree* is, in effect, a pre-completed binary search. The root node of the tree contains the middle value of the data set. If the value you're looking for is less than that, you branch left; otherwise, you branch to the right. When you get to the next node, you again compare its value to the value you seek, branching left or right and continuing until you either find the value you want or you run out of nodes. It works because the nodes are sorted: if you read the nodes in a tree diagram from left to right (using the left edges) regardless of the tree level, you get all the values in order. Figure 1 shows the first three levels of a standard balanced binary tree for our "guess the number between 1 and 4000" game. Note that by following the tree to the left or right at each node, you don't need any computation at all to find the result: just keep going until you reach the end or find your value.

You may be thinking, "What a huge waste of time! Why spend effort to build a data structure that essentially saves you one division-by-two operation?" It turns out that our guess-the-number game is a bit of a convoluted example. If you already know the data is 4000 consecutive numbers in order, you wouldn't have to search hard at all to find one of them. But what if you only have fifteen items with values somewhere between, say, 1 and 500,000,000? If you start guessing at 250,000,000, a binary search guarantees you'll find any value in 29 or fewer steps, but if you only have 15 values in the first place, *that's* a huge waste of time.

Figure 2 shows a balanced binary tree with 15 random values between 1 and 500,000,000. As you can see, since there are only 15 entries, it takes no more than $\log_2 15$ (rounded up, that's 4) attempts to either find any entry in the tree or know that it does not exist. For any random value, it may take all four attempts to discover it's not there, because only four out of five hundred million possible keys exist in the tree. Even so, four is a lot better than fifteen or twenty-nine comparisons for the same results.

That's why binary tree variants work well for HFS and HFS Plus. The set of all possible HFS Plus file names is the set of all possible Unicode text values between 1 and 255 characters (up to 512 bytes), and there may not be enough disk space on the planet to hold that huge set of possibilities. We already saw how simple data structures like linked lists aren't suitable for large directories, but raw binary searches aren't suitable when the data could have a huge number of values. Binary trees strike the necessary com-

6 ▶

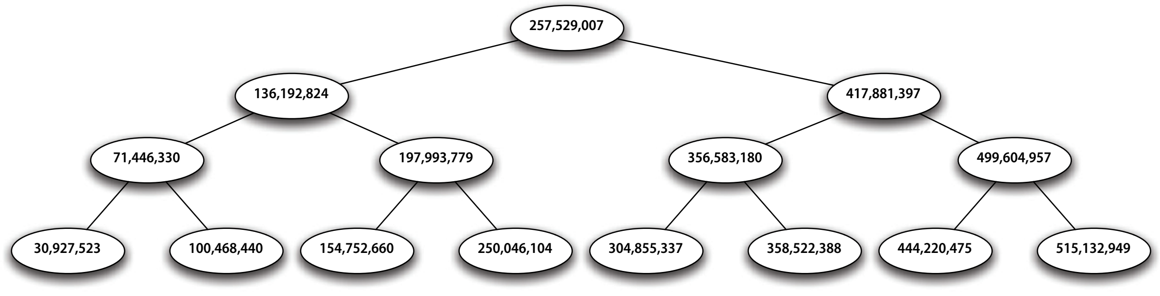

FIGURE 2 – A SMALLER TREE WITH A LARGER RANGE

◀ 5

promise, providing fast searches of large or small sets of data with huge ranges.

keeping your balance

The trade-off? Higher maintenance. The structures in our examples are actually *balanced binary trees*: every node has zero or two children that have their own children all the way to the bottom of the tree, and the middle value is in the root node of the tree. If a binary tree isn't balanced, there's no guarantee that half of the possible child values are on either side of a particular node. For example, if file names could start with the letters "A" through "Z," the root node of an unbalanced tree could be a file starting with "A." Most other files would fall on the right side of such a tree. In the worst case scenario, searching an unbalanced binary tree could require examining every single node, though it's *usually* faster than that.

Only a balanced tree guarantees finding a result out of *N* entries in $\log_2 N$ or fewer tries, and that's why Apple's engineers chose these balanced trees to store the catalog on an HFS disk. The problem is that the tree has to stay balanced, so adding any new node may force relinking much of the tree. Figure 3 shows what happened to the tree in Figure 2 when we replaced two numbers with others that fell in a different place in the sort order. Although only two numbers changed, eleven of the fifteen nodes in the tree "moved" to different positions because the sort order changed, including the entire left half of the tree because one of the new numbers was less than the previous smallest value in the tree. Moving a few thousand entries in a large tree can be some serious work, and that's the trade-off for fast searches of large data sets.

It's full of stars!

HFS and HFS Plus use a slightly different kind of tree, called a *B*-Tree*, that has three more important properties. In our sample diagrams, each node has only one *record* (a number), but in B*-trees, each node has multiple records. If a node had two records, like the numbers 15 and 29, the search algorithm would go "left" if the sought value was less than 15, "right" if it was greater than 29, and "down" if it was between 15 and 29. That's right– a node with *N* records has *N*+1 children, so a two-record node points to three children. Storing more records in one node makes it easier to rebalance the tree, as much of the time you can simply add another record to an existing node to avoid much of the nasty work.

Just in case that was too easy to understand, HFS and HFS Plus make it more difficult: an HFS or HFS Plus B*-tree node with *N* records has *N* children, not *N*+1. Why? Because in a move everyone finds odd, HFS-style B*-trees *never point left*. Or, more technically, "in a given subtree, there are no keys less than the first key of that subtree's root node." If the search algorithm gets to a node with two records, like the numbers 15 and 29, it knows that the value it seeks is

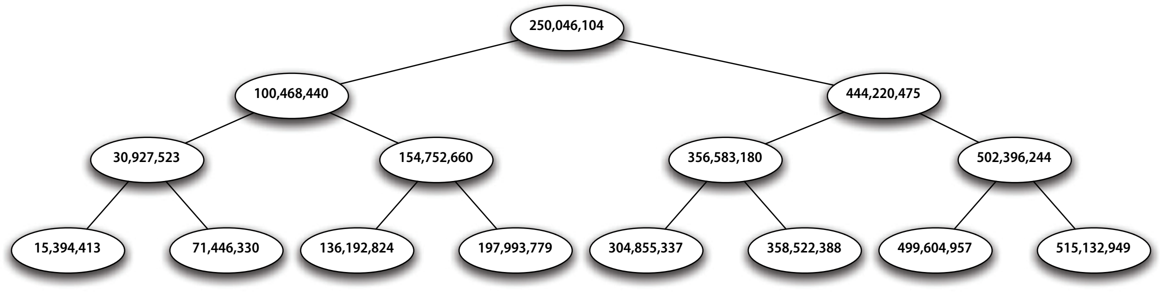

FIGURE 3 – THE TREE, REBALANCED

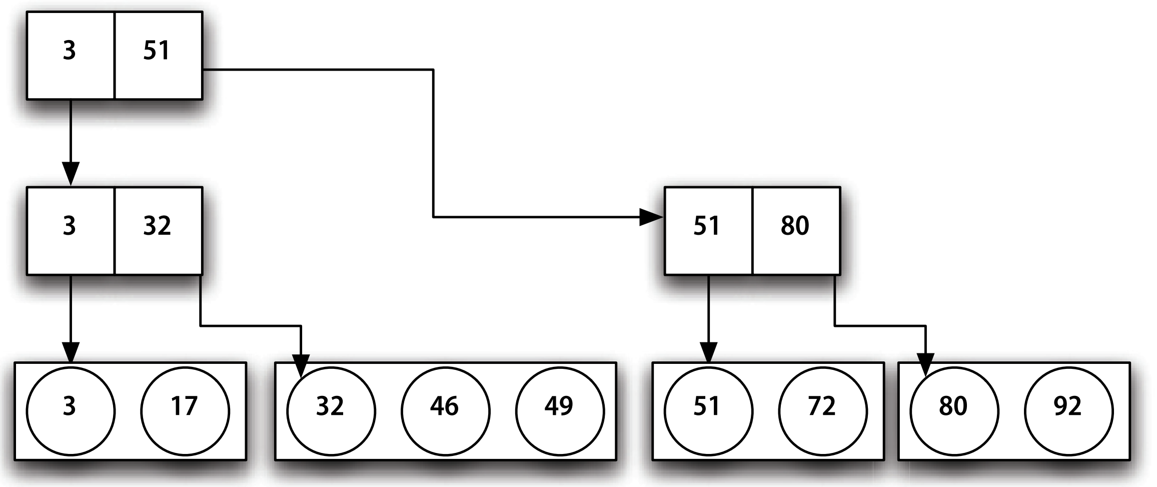

FIGURE 4 – AN HFS-STYLE B-TREE*

either between 15 and 29 or greater than 29. If it were less than 15, the algorithm wouldn't have arrived at this node.

That variant is unique to HFS-style B-trees, but all B*-trees share the final characteristic: only the *leaf* nodes at the very bottom of a B*-tree contain the actual data you want. All the other nodes higher in the tree contain only *index* nodes that eventually point to a leaf node. For example, a file's directory entry contains more than the file name: it has the file type, creator type, modification dates and times, and the information pointing to the file's data blocks. In an HFS-style B*-tree, all that information is found only in the leaf nodes at the end of the tree. Every other node just has a filename and a pointer to either another index node for that file or to the file's real leaf record.

Figure 4 shows a B*-tree much more like what HFS and HFS Plus actually use. There are multiple records in each rectangular node, and only the leaf nodes at the bottom of the tree contain actual data records (still indicated by circles). The root node and second level are just index nodes – their only purpose is to point to other nodes and further the search. Since HFS-style B*-trees never point "left," we arranged Figure 4 to emphasize that each node contains no children with values smaller than its own smallest value. That requires every index level to repeat the lowest-numbered value that pointed to it, and requires the root node to start with the lowest-numbered value in the entire tree. Otherwise, at some point, the algorithm would have to branch "left."

It sounds confusing but makes searching simpler. In each node you find, starting with the root node, you find the record with the greatest value that's still less than or equal to the value you're seeking. You then go where that record points and repeat the process until you reach the leaf nodes at the bottom of the tree. At that point, you either find the value you want or know that it's not in the tree at all.

Note that one of the leaf nodes in Figure 4 contains three records even though the others contain only two. That's fine – Apple requires every node in an HFS-style B*-tree to be large enough to hold at least two records, but it can contain more. HFS Plus defaults to 4KB per node, easily enough to hold two records of full 512-byte Unicode file names, usually many more with typical names. (On a test iMac, we found about fifty records in most index nodes.) You might get the tree in Figure 4 if "49" was the last leaf record added to the tree: it would be far easier to just add it to the node where it's shown than to create a new node and rebalance the tree. Had the algorithm added "49" to the third leaf node instead of the second, it would have required changing the index nodes valued "51" at both higher levels of the tree to instead point to "49," and that's more work than necessary.

At some point, however, nodes fill up. When that happens, the file system implementation must create a new node and move some of the existing data into it, careful to keep everything sorted from "left to right" in each node and, in fact, in each level of the tree. That's one reason Apple uses B*-trees: moving and splitting a bunch of index nodes is easier than moving and splitting a bunch of leaf nodes. Every index node contains its index value (like the numbers 49 and 51 in our example) and a pointer to its child node that's no more than the number of the child node. It's a lot easier to rearrange a bunch of values and node numbers than to rearrange all the other

8 ▶

◀7

data that might be associated with an actual value, like a file's full directory entry.

Some people say that a B*-tree is not a variant of a *binary* tree because each B*-tree node can have more than two children, and a true binary tree has only either zero or two children per node. Other people disagree, but since the National Institute of Standards and Technology sides with the zero-or-two definition, we'll accept it. Most of the diagrams in Part 1 had two children per node, but that doesn't mean the B*-trees are *binary* trees. NIST calls them *k-ary trees*, a term we find needlessly confusing when "trees" works quite nicely.

the catalog file

The difficulty of moving nodes and values around depends in large part on how the information is stored on disk. Tree diagrams demonstrate the concepts, but it's not like your hard disk has separate levels of blocks for parts of a tree – everything is just a block. Actually, to HFS and HFS Plus, storage is parceled out in chunks called *allocation blocks*, a multiple of 512-byte disk blocks. That's one reason HFS Plus became necessary. Just as ProDOS volumes are limited to 64K of 512-byte blocks (that's 65,536 of them), an HFS disk is limited to 64K allocation blocks.

To make volumes larger than 32MB, HFS has to increase the allocation block size – a 64MB volume, for example, has allocation blocks of 1KB each (two 512-byte blocks). Alas, by the time you get up to a 74GB volume on one of today's 80GB hard disks, HFS would require an allocation block size of over 1MB. That means you take a full 1MB of disk space for each of those eight-byte `PkgInfo` files, because HFS can't allocate less than one allocation block of disk space. HFS Plus allows for 4MB worth of allocation blocks, or 4,294,967,296 of them. By default, HFS Plus uses 4KB allocation blocks, even on volumes that could use smaller ones, because Mac OS X uses 4KB buffered disk input and output, and transfers a 4KB chunk of data more efficiently than any smaller size.

But every allocation block is created equal – none is any better than any other, and the main purpose of allocation blocks is to combine to make files. In fact, all the nodes in the tree for an HFS or HFS Plus disk's catalog are stored in one big file called – wait for it – the *catalog file*.

The location of the catalog file itself is hard-coded so you don't have to find the catalog file to find the catalog file. Both HFS and HFS Plus disks contain a structure in block 2 of the volume that defines the volume's characteristics; we'll talk more about it later, but one of the things it tells you is where to find important HFS and HFS Plus data like the catalog file.

what is a node?

The format of a text file is pretty easy: every byte is text. Binary files have more complicated structures, and the B*-tree in the catalog file certainly qualifies as "binary" and "complicated." However, given what you've already learned, it's not so bad. The catalog file is grouped into nodes, just like the tree. The size of an HFS Plus node is determined when the disk is initialized, and the length must be a number of bytes that's a power of two with an exponent between 9 and 32, inclusive (that's a power-of-two block size between 512 bytes and 32,768 bytes, or 0.5KB to 32KB, inclusive). HFS nodes are always 512 bytes, but HFS filenames are never longer than 63 bytes (31 characters, possibly two bytes each in non-Roman writing systems), so a 512-byte node holds more HFS file names.

Each node begins with a 14-byte *node descriptor* that, in another display of plain language, describes the node. The file system implementation needs to know several things about each node, including what kind of node it is (such as an index node or a leaf node – each node contains only one kind of record, so the kind of records defines the kind of node), how many records it contains, what depth the node is in the B*-tree, and where to find the next and previous nodes of the same kind.

This information is how the file system implementation finds other index nodes when it's time to rebalance the tree. Individual records only point to child nodes, but each node's own structure contains forwards and backwards pointers to other nodes of the same kind, such as other index nodes or other leaf nodes. If a node needs to be split, the algorithm doesn't have to walk the whole tree to find the node with the next highest values at the same level. Each node already has that information, as well as the next lowest values at the same level. In Figure 4, for example, that means each of the third-level leaf nodes knows the node number of the node to its left and right. If the algorithm needed to move the leaf record with value "49" from the second node to the third, it knows the number of the third leaf node without searching for it.

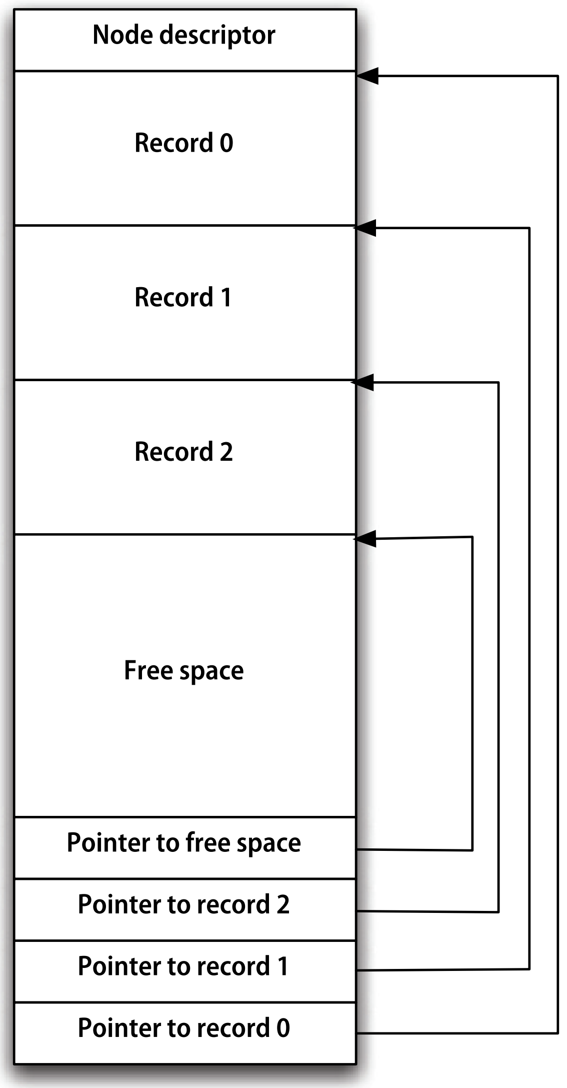

FIGURE 5 – NODE STRUCTURE

After the node descriptor, each node contains as many records as fit. In a characteristic HFS-style fashion, the two bytes at the very end of the node point back to the location of the first record in the node, always at byte 14. The two bytes just before that at the end of the node point to the second record in the node, and so on, with records building from the front to the middle of the node, and pointers to the records within the node building from the end back towards the middle. That leaves the biggest set of free space in the middle of the node in one chunk. If a record gets deleted from a node, the implementation must compact the other records to maximize the free space.

Figure 5 shows this arrangement in a diagram freely extrapolated from Apple's developer documentation. It also shows that every node has *N* records and *N*+1 pointers, because the *N*+1st pointer points to the beginning of the free space in the node, preventing an implementation from having to calculate where it is.

storing the nodes

The HFS or HFS Plus catalog file is just a bunch of these nodes. Just as each node contains a header explaining what's in the node, every B*-tree file – including the catalog file – contains a *header node* that describes the entire B*-tree structure, including information like the size of each node and what nodes in the tree are currently in use, just as each node knows what records within it are available or occupied. The header node knows how deep the entire tree is, the number of the root node where searches start, and everything else the algorithm needs to search and find any record in the tree.

An empty catalog file starts with a slew of empty nodes waiting to be filled, pre-allocated to help reduce fragmentation. As you create files, these nodes fill up. When every node in the catalog file is full, the HFS or HFS Plus code grows the file and allocates more nodes. Each node's number is determined by its position within the catalog file, so once a node is created, it doesn't move on disk. It shifts positions in the tree depending on the values in other nodes, and it may be full of records or have nothing but free space, but node #1 stays node #1 forever, no matter where it is in the tree, what records it contains, or even if it's in use at all.

If a node's number changed, the implementation would have to walk the entire tree and change every reference to that node. That's not how B*-trees work. Each node knows its own children, but does not know what other nodes point *to* it. Since any node can have any logical position in the tree at any time, the implementation doesn't have to do the nasty work of relocating nodes, just relinking them. Any node can be the root node, for example: whatever node is listed in the header node as the root node is the root node. If rebalancing the tree needs to change that, the implementation doesn't empty out the old root node and move in new records – it takes the node number for the new root node and stashes it in the header node.

When the catalog tree gets very large, logically consecutive nodes could be spread out over several blocks on the disk. Both Mac OS 9 and Mac OS X try to cache as much of the catalog file in RAM as possible, but a hard disk's catalog file could take 20MB or more of disk space, and that's too much RAM for the

10 ▶

OS to spend on cache on most systems. The operating systems also work to minimize fragmentation by allocating space for the catalog file in clumps large enough to hold hundreds of nodes and thousands of files at once.

Still, it's possible that a deep tree search could wind up jumping between multiple catalog file fragments on the disk. The good news is that even on a hard disk with a huge number of files, like 500,000, the B*-tree guarantees the search algorithm will find what you're looking for in 19 or fewer hops between nodes. Experts say that most HFS Plus disks with regular use rarely have a catalog file with more than three or four fragments, though the repeated relinking of nodes in the tree could make the disk head jump back and forth several times in a single fragment.

the vermicious CNID

A record in a B*-tree contains a key and data, or in terms you may recognize from Mac OS X property lists, a key and a value. You use the key to find the record so you can access the data you want. Our diagrams have used integers for keys, and we've implied that the real keys in HFS-style B*-trees are the filenames. That's not quite true: the keys are CNIDs and filenames concatenated together.

CNID stands for *catalog node ID*, but everyone uses the abbreviation because the number is not the ID of a catalog node, though once upon a time it might have been. The CNID is a unique four-byte number between 16 and 4,294,967,295 (or, in hexadecimal, between `0x10` and `0xFFFFFFFF`). Each CNID has historically been unique per disk: if you delete the file or folder with CNID 1575, no other file or folder on that disk can ever have CNID 1575. The next available CNID is tracked in the volume information in block 2 in a field named `nextCatalogID`. CNID values are typically assigned sequentially to avoid gaps since they can't be reused.

Until Mac OS 9.1, if you created and deleted so many files that you reached CNID 4,294,967,295, you'd exhausted the disk and could never create another file or folder on it. You'd have to back it up, erase it, and restore it. Erasing the disk resets the next file to use CNID 16, and each file you restore gets a new, low CNID as it's created. Fortunately, the largest CNID is so big that you could create and delete 100,000 files on an HFS Plus disk every day for over seven years before reaching the limit.

Starting in Mac OS 9.1, though, Apple says that's no longer true for HFS Plus. At that time, Apple defined a previously-reserved bit (bit 12) in the HFS Plus volume header, saying that if it was set, the volume might have recycled CNIDs on it. (This is the same way Apple indicates that Journaled HFS Plus is active in Mac OS X 10.2.2 and later – journaling is active when bit 13 of the volume attributes is set, and CNIDs may be reused if bit 12 is set.)

The documentation was a warning for disk utility developers. Since CNID numbers are assigned sequentially, `nextCatalogID` should be greater than the value of any CNID already on the volume. With bit 12 set, that may not be true. The next CNID value may have wrapped around to low numbers when lots of big-value CNIDs are already assigned to files. Apple writes, "A disk repair utility that sees [bit 12] set should not complain if it finds a CNID larger than `nextCatalogID` already in use."

We've considered this carefully after failing to mention it in **MDJ** when this series first ran, and firmly believe that this feature, as implemented in Mac OS 9.1, is actually a bug. None of this is mentioned in the official HFS Plus Volume Format documentation, but then again, neither is journaling – *six months* after users got their hands on it. That's unimportant compared to the documented rules for changing anything important in HFS Plus. Another field in block 2 of HFS Plus disks, `lastMountedVersion`, is a four-byte value that tells anyone reading an HFS Plus disk that the rules may have changed.

To quote the HFS Plus documentation, "It is very important for implementations (and utilities that directly modify the volume!) to set the `lastMountedVersion`. It is also important to choose different values when non-trivial changes are made to an implementation or utility. If a bug is found in an implementation or utility, and it sets the `lastMountedVersion` correctly, it will be much easier for other implementations and utilities to detect and correct any problems." The initial value for `lastMountedVersion` was '`8.10`' (for Mac OS 8.1). It changed to '`10.0`' under Mac OS X, and to '`HFSJ`' when journaling is active. In fact, `lastMountedVersion` is how Journaled HFS Plus knows if some non-journaled version of HFS Plus mounted a disk that had a valid journal on it. If Mac OS X 10.2.2 or later mounts a disk that has a journal but `lastMountedVersion` is not '`HFSJ`', it knows that some other code has been using the disk, and it ignores the journal (**MWJ** 2002.11.18).

The problem is that `lastMountedVersion` did *not* change when Apple started reusing CNID values in Mac OS 9.1. Such reuse had always been illegal in both HFS and HFS Plus, and any properly-written disk utility would have seen a reused CNID as a proper sign of a damaged volume. But if that's the case, the `lastMountedVersion` should not have been '`8.10`', because that tells all implementations and utilities that the rules for the disk are the same as they were under Mac OS 8.1, as documented in Technical Note #1150. Under those rules, any reused CNID is a problem and should be fixed, not tolerated.

Apple has the right to change HFS Plus as it sees fit, but under the rules Apple still maintains, it is "very important" that any "non-trivial changes" be accompanied by a new value for `lastMountedVersion` so all HFS Plus code knows what's going on. Had Apple done that, perhaps with a value like '`9.10`' for Mac OS 9.1, everything would be fine. Everything *is* fine if `lastMountedVersion` is '`10.0`' or '`HFSJ`' or anything past '`8.10`', because any other value signals to any interested code that *something* has changed. But to say that values that have always been illegal are now allowed without changing the very field designed to pass that information along? That's a bug.

Nonetheless, the HFS Plus implementations in Mac OS 9.1 and later can and do reuse CNIDs if bit 12 of the volume attributes is set, even if `lastMountedVersion` hides that with an incorrect value of '`8.10`'. With this bit set, an HFS Plus disk is not full until it either runs out of storage space or until it holds more than 4,294,967,280 files, give or take sixteen or so.

The value of IDs

You may have heard about file IDs or folder IDs on HFS and HFS Plus volumes. On the disk, those are the CNID values. They're why Mac OS aliases have always worked so much better than Unix symbolic links or Windows shortcuts. Those non-Macintosh files refer to their targets solely by full pathname, so if you move or rename the target file, a symbolic link or shortcut breaks. CNIDs are permanent for each file and folder leaf record, so even if you rename the file or change any other information about it, the CNID stays the same. That's why as long as you leave a file on the same volume under Mac OS 9, an alias pointing to it always finds it.

By the way, in Mac OS X 10.2 and later, aliases resolve by full pathname first and only second by CNID. That's probably an improvement. If you throw a file away and replace it with a new one in the same place with the same name, Mac OS X 10.2 resolves an alias to the original file by finding its replacement. Mac OS 9 finds the original file in the Trash.

The CNID is important in the catalog file because the key for finding any given file or folder is its parent folder's CNID concatenated together with the file or folder's name. So, for example, if an HFS Plus file system implementation searched for a folder named "`Library`" whose parent folder had CNID of 65,536, the search key in the catalog B*-tree would be the four byte value 65,536 (`0x00010000`) followed immediately by the Unicode text "`Library`". (In an HFS implementation, the name "`Library`" would be in one of the Macintosh's eight-bit text encodings, not in Unicode.)

An HFS or HFS Plus disk only has one catalog file, not a separate catalog file for each subdirectory on the disk. As such, it might be hard to find all the files in a given folder, but it's not. Because every file's search key starts with its parent folder ID, all files in any folder wind up sorted together in filename order in the B*-tree. It's the binary equivalent of beginning all your filenames with "`AA`" to make sure they rise to the top of a list window, or "`ZZ`" to make them sink to the bottom. These search keys effectively organize the giant catalog B*-tree by subdirectories, so listing all files in a directory can be as easy as walking the forward and backward links between index nodes at the right level.

These search keys are why Mac OS programs traditionally specify files not by pathname, but instead by parent folder ID and filename. All the system must do is concatenate the two together and search the tree to find the file. Traditional Unix programs look for files by full pathname, though. That requires two searches through the tree: one to find the CNID of the parent folder, and a second to find the file once the OS can combine its name with its parent folder ID to form the right search key. There are *thread records* in the catalog B*-tree, another kind of record that helps the OS find a file or folder by its CNID. Thread records are mandatory for all folders and files in HFS Plus, and mandatory for folders but optional for files in HFS. In practice, HFS files don't

12 ▶

◀ 11

have thread records (or accessible CNIDs) until code requests one, usually by creating an alias to the file.

The double-search required to find a file by full pathname may undergird the animosity that Apple's NeXT-heritage management feels towards HFS Plus. It does not explain why Apple management continues to encourage developers to locate files by pathname instead of directory ID and filename – and, in fact, has worked to make directory ID values less useful in Mac OS X. Perhaps Apple engineering management thinks that if the company eliminates HFS Plus advantages one by one, people will eventually be glad to use the less-capable UFS volume format that the NeXT systems always used. So far, it hasn't been working, and the truth is probably more benign, as we'll explore later.

the catalog summary

When you assemble all these pieces in the proper order, you get an HFS-style catalog file. The volume's static information in block 2 points to the beginning of the catalog file. The catalog file itself starts with a header node that describes the B*-tree structure, and that's followed by at least one and usually many more nodes. If the records for every file on the volume can fit in a single leaf node, then that's all the catalog file may contain (other than the header node). It's entirely possible for the root node – the one where B*-tree searches start – to be a leaf node. However, as soon as there are enough files to need more than one leaf node, you need index nodes, and the normal tree structure starts building.

Nodes are numbered by their position in the catalog file. Once they're created, the nodes themselves never move even though their contents may completely change depending on what the B*-tree needs. The file system implementation rebalances and relinks the B*-tree by making nodes point to different nodes, not by moving the node structure to a different place within the catalog file. A new catalog file on a new disk initially has room for hundreds of nodes and thousands of file and folder records, but it can grow to take up more disk space if necessary, though that does fragment the file. A fragmented catalog file is better than a ProDOS-style "catalog full" error.

To be sure, these data structures are complicated to implement, and even experts have difficulty using them correctly at times. Textbooks on the subject describe these kinds of binary trees as requiring lots of debugging and careful attention to edge cases, insertion, and deletion methods.

They're also hard to examine. It's fairly easy to look at ProDOS directories using a block editor and understand what's going on, but that's because ProDOS directory blocks are simple: all but five out of 512 bytes in each block are file entries, so the format is pretty easy to grok. Any given 512-byte block in an HFS-style catalog file, however, could be just one-eighth of a B*-tree node. Records in that node (if there are any at all) could be index records that point to other nodes, leaf records with real file or folder information, or thread records that help resolve aliases.

The node you find may be anywhere at all in the logical organization of the B*-tree, or it may be empty and waiting re-use. To find out any of this, you have to know what block of the node you're looking at, interpret all the data structures, and follow them to other blocks and nodes. There are probably fewer than two dozen people on the planet who can look at HFS Plus catalog blocks and know what they mean without constantly referring to the documentation, using massive amounts of psychotropic drugs, or using a program that interprets the information for you, like Norton Disk Editor+ in Symantec's Norton Utilities. The HFS documentation is slightly easier, but not much.

It's worth the pain to various versions of the Mac OS because HFS and HFS Plus provide exactly what Macintosh users and programmers need: a file system that handles huge numbers of files and very large disks without bogging down. The B*-tree in the catalog file lets traditional Macintosh programs find files very quickly. Pathname-based access takes more searches through the tree, but other file systems also have to find each directory in a pathname to reach a file, and the B*-trees make sure each of those searches is fast, even in really big directories. That's one heck of a win for giving up a format that's easy to see in a block editor, even if making it work did give some of Apple's best programmers extra gray hairs.

damage control

HFS and HFS Plus were designed for those benefits, but the catalog tree described here has another relevant benefit: it's hard to kill. Sure, you can write a buggy program that damages a catalog file beyond repair, but it's hard to do.

There are more ways to damage a file than you can count, particularly if you're devious about it, but the most common damage to the catalog file is when some code writes the wrong data to it. It could be as small as a few errant bytes (say, an HTML tag that got put in the wrong place in memory), or as nasty as an entire allocation block from the wrong file. Under Mac OS X, protected memory is supposed to stop most of this: all of the code that writes blocks directly, or writes to the catalog file at all, should live only in the kernel's address space. Nonetheless, lots of people run Mac OS 9, and accidents do still happen even under Mac OS X. Maybe some data in the cache wasn't flushed to disk before a system crash or kernel panic, or maybe an allocation block number got corrupted so the wrong block got written to the catalog file.

That's going to cause problems, but probably not as many as it could.

Again, look back at ProDOS directories. 507 out of 512 bytes in each ProDOS directory block belong to some file's directory entry. If even one random byte in such a block gets trashed, there's more than an 80% chance it will damage something. (Not every byte in a directory entry is used or important, otherwise it would be a 99.8% chance of damage.)

If it damages the wrong byte of a directory entry, you could lose access to that file. If it damages the pointer to the next or previous directory block, you could lose access to many other directory blocks because the links are broken. On the other hand, if the damage is changing an "A" in a file name to a "Z", or damages an unused byte, or changes the creation time to an invalid date and time, you lucked out. It's still damage, but it doesn't stop you from using the directory or any of its files, though you might need to rename any files with name damage.

The more bytes damaged, the greater the chance you'll lose something important. If some bad code replaces a ProDOS directory block with a completely different block (for example, a segment of a text file), the files that directory block referenced may be lost forever. If you lose the number of a ProDOS file's key block (whether it's the only block in the file or a master index block), you can't reliably recover it. Nothing else on the disk identifies that block as some file's key block.

Now think about a random byte of damage in an HFS Plus catalog B*-tree node (by default, the same as the default allocation block for HFS Plus). First, you have to consider if the node is even in use in the B*-tree. If not, then there's not much chance of any significant damage. If the node is in use, it may not be full, so a big part of it may be empty and therefore impervious to damage as well.

If the damage hits a record in a node, what kind of node was it? If it was an index record, the damage doesn't affect the file's real information in a leaf node somewhere, just how the B*-tree search finds it, and a disk repair program can find and fix the problem. If it was a leaf record, then like ProDOS, some random bytes could damage the entire entry and some might just mess with the filename.

If a pointer to another node gets trashed, the node is still in the catalog file, and almost any utility can find it and figure out something's wrong. If an entire node gets wiped out, it only affects file information if it was a leaf node. If it was an index or thread node, a disk repair program will eventually figure out that the catalog file contains leaf nodes that aren't properly linked in the tree and fix it. And even if a leaf node gets stomped on, other HFS structures can provide disk repair programs with hints that something is wrong – like index and thread nodes that appear to point nowhere.

The very complexity and redundancy that makes HFS-style catalogs difficult to understand provides a significant level of protection against corruption. Again, you can't fix every problem without losing some data, but it takes a particularly unlucky bit of damage to lose a file – and especially a disk – beyond recovery.

HFS and HFS Plus have always treated the catalog as just another file. Unlike file systems like ProDOS, the HFS catalog doesn't occupy a specific set of blocks on disk and therefore isn't limited to a certain size or number of entries. What's more, any HFS or HFS Plus implementation can read the catalog file just like it reads any other file, so there's less special-case code. It's a good idea, but to make it work, HFS and HFS Plus store what you might call the "directory entry" for the catalog file in their static volume information, in the same place on every HFS or HFS Plus disk.

The catalog file is one of five "special" files on HFS Plus disks whose locations are kept that way. HFS has only three special files per disk, as you'll see. To explain that, we must move beyond how HFS and HFS Plus find a file's directory entry and learn how they find the file itself – the allocation blocks that hold the file's data.

14 ▶

the nature of files

Not to resurrect an old debate, but there are two kinds of information stored for every file: the information *in* the file, and the information *about* he file. The former is the file's *data*; the latter is its *metadata*. On HFS and HFS Plus disks, the difference is one of storage. A file's data is stored in a series of allocation blocks assigned to the file, and its metadata is stored in the file's catalog leaf record.

HFS Plus catalog records for files and folders are slightly different. Both contain the CNID of the file or folder's parent folder, as well as Unix-style permissions, a creation date, modification date (for the file or folder's contents), backup date, and date of last access. (In HFS and HFS Plus, a *date* is a number of seconds since a fixed reference point, and therefore really means a date and time, not just a day. In HFS, these date-time values are in your local time; HFS Plus stores all dates in Greenwich Mean Time except for the volume's own creation date. That's still in local time.

Each HFS Plus file and folder catalog record also contains a text encoding "hint" as to what language or script was in effect when the file or folder got its name. Mac OS 8.1 through 9.2.2 lack extensive Unicode support, so the operating system uses this hint to help convert file or folder names to and from Unicode in ways that don't surprise and annoy you.

It's entirely possible for one Unicode filename to contain Chinese, Icelandic, English, and Greek characters, among others. The Mac OS 9 Finder cannot represent such a filename in one script system, so the Mac OS uses this hint about the primary script system at the time the file was named to help translate the filename into something that makes as much sense as possible, and vice-versa. It's most effective when you create a file under Mac OS 8.1 through 9.2.2, since Mac OS X has no trouble with full Unicode filenames. Only Mac OS 9 and later allow any programs to use full Unicode filenames, so some programs still don't support them (Microsoft Excel v.X comes to mind). Even so, the hint is primarily for older versions of Mac OS, not for application programs.

Both HFS and HFS Plus reserve thirty-six bytes for "Finder info," and as Apple emphasizes, "its format is not part of the HFS Plus [or HFS] specification." These fields are the source of most of the Great Metadata Arguments in Mac OS X, for they contain the "Macintosh-specific" metadata: file type, creator type, label, ID of a file's comment in the desktop database, and the ID of the folder where this file or folder lived before you moved it to the desktop so the "Put Away" command can work. You'll also find the Finder flags here, identifying whether a file is an alias, or invisible, or has a custom icon, or is a stationery pad, among other things. You can read about these 20-byte `FInfo` and 16-byte `FXInfo` records in Apple's developer documentation, if you wish.

Folder catalog records contain an additional value called the *valence* – the number of files contained within that folder. Technically, HFS Plus is almost a "flat-file" system because there is only one catalog file per volume. Most people, however, agree that HFS and HFS Plus are hierarchical because the catalog is sorted by directory IDs first and filenames second. Yet, since each folder is not cataloged in a separate file, the file system needs some easy way to keep track of how many files are supposed to be in each folder so implementations don't have to search the entire tree to find out. That's the valence. Before Mac OS X, by the way, the system dropped to its knees and cried if you tried to put more than 32,768 items in any folder. If you ever want to use an HFS Plus disk under Mac OS 9, don't test this rule.

File catalog records have no valence, since files don't contain other files on disk, but they have all the rest of the information described for folder catalog records. On the other hand, folders have no file data – the catalog records for all files and folders are in the catalog file, so a folder catalog record doesn't point to any allocation blocks on the volume. Files, of course, have data, as that's pretty much the point of a file system. Each file catalog record points to both a data fork and a resource fork for that file. Each fork of an HFS or HFS Plus file is what other file systems consider to be an entire file – one stream of bytes. The reference to each fork contains the fork's size in bytes, the number of allocation blocks it uses, the fork's clump size, and its list of extents.

files and their blocks

Clumps? Extents? Ah, now we're to the heart of the matter. Designing a disk's catalog structure is important, but so is designing how to find the data blocks for the files themselves. Some files systems are simpler than others, but that leads to those pesky tradeoffs. Let's observe.

As noted earlier, ProDOS tracks files very simply. Each file's directory entry points to a key block that holds the entire file if the file is smaller than 512 bytes. If not, the key block is either an index block that points to as many as 256 data blocks for the file, or a master index block that points to up to 256 index blocks for really large files. (The three kinds of files are called *seedling*, *sapling*, and *tree* files, in case people in masks wielding copies of Beneath Apple ProDOS kidnap you and demand that you join their cult.)

When a ProDOS implementation allocates files sequentially on a mostly-empty disk, these index blocks and master index blocks wind up close to the data blocks, so the drive head doesn't have to move back and forth much to read the file sequentially. In fact, even on the old 64K Apple II, ProDOS tried to keep a file's current index block in memory for even faster access and fewer disk head movements. ProDOS uses a volume bitmap at the beginning of the volume to know what blocks are available, and it allocates free blocks sequentially where possible to avoid wasting disk space. If a ProDOS volume has lots of free space fragments when you create a big file, the new file fills them in and becomes fragmented itself.

ProDOS's fragmentation doesn't come close to the level of FAT, though. The MS-DOS file system is named for its *file allocation table*, the method it uses both for tracking free blocks and for assigning them to files. We'll use HFS-style terms to explain this, so be warned that Microsoft's documentation may use other terms, like "clusters" instead of "allocation blocks."

In the first two versions of FAT, named FAT12 and FAT16 for the number of bits used to count allocation blocks (allowing, respectively, 4096 or 65,536 allocation blocks on a disk), you'll find the file allocation table near the start of a volume. Each file or folder's directory entry points to one allocation block on the disk, the first allocation block of the file. That block's entry in the file allocation table points to the next allocation block for the same file, chaining through the table until the pointer has a special value indicating the end of the file.

For example, imagine a text file that occupies four sequential allocation blocks, with the first one numbered 1000. The file's directory entry contains the number 1000 for the file's first block. Entry #1000 in the FAT contains the number of the file's second allocation block, #1001. The FAT entry for block #1001 points to allocation block #1002, and the entry for #1002 points to #1003. The entry for #1003 contains a special marker meaning "end of file," a number between 65,519 and 65,535 (`0xFFF8` through `0xFFFF`) in FAT16 with similar top-of-range values for FAT12 and FAT32. A FAT value of zero means the block is available and not part of any file or folder.

Bill Gates cooked up this scheme in the 1970s in a five-day programming marathon long before Microsoft made a disk operating system, but it's stuck around because it's clever and easy to implement. The same file allocation table tells you not only if a block is in use but to what file it belongs, or at least to what chain of blocks. Disk repair programs can find orphaned chains of data easily if you lose part or all of a file. There's no sapling or tree files that keep track of one file's information separate from the catalog entry or the file itself. It's quick and efficient.

It's also hard on drive mechanisms. FAT nearly always stores the table near the front of the volume, so every time an implementation needs to find the next block in a file, it has to go back to the beginning of the partition to get the information, then back to the data area to read the block, then back to the beginning for the next block number, and so on. Caching helps tremendously, but it takes a lot of RAM to cache an entire FAT structure, and even a caching implementation has to write changes to disk as they happen or you'll have damaged disks in case of a power outage or flip of the wrong switch. Some implementations keep multiple copies of the FAT structure at different points on the disk, but that's just trading off faster seek time in reading for more complicated writes to update all the copies of the FAT.

FAT also fragments files rather badly. Every time a file or folder needs a new block, it gets the first free block in the file allocation table. Once you start deleting files, the next ones you add fill in all the gaps. Unlike some other disk formats, FAT and ProDOS allocate space for files one block at a time, actively encouraging fragmentation rather than wasting any disk space.

This, by the way, is why PC experts are so very keen on disk defragmentation – under FAT12 and FAT16 (but less so under NTFS and FAT32), it can produce stunning disk performance improvements. If some key Windows file expands by one allocation block five or six separate times, it's going to wind up in five or six separate parts of the disk, even if Windows reads it hundreds of times per day. Optimizing or defragmenting the drive puts all those blocks next to each other. If you're not at the computer so it's not wasting your time, the process can

16 ▶

◀ 15

drastically reduce stress on the drive and noticeably improve performance.

That's not so much the case for HFS and HFS Plus.

the extent of the matter

HFS has always approached things slightly differently. The very concept of using a B*-tree instead of a more space-efficient linked list shows that HFS's designers valued performance more than squeezing every single byte off a disk, a strategy that has survived the test of time. You can tell because FAT32 and NTFS, Microsoft's biggest new file systems, "borrow" several concepts from HFS (to put it politely), including using a catalog file stored as a tree structure.

Performance and efficiency aren't always opposites. For example, most files, once created, never grow. Sure, lots of files you'll create definitely grow afterwards – your documents, logs, mailboxes, databases, and all those files under your direct control. Yet look at how many files are on your hard drive. We have a standard flat-panel iMac that hasn't seen much added to it, and its HFS Plus volume header reports over 170,000 files and nearly 52,000 folders. We certainly did not create that many files or folders using the machine. Most of them came from installing the operating system and several programs. In today's Mac OS X-friendly package format, each application contains several folders and usually dozens of files.

Those files won't grow. They're fixed upon installation and will sit there until they're updated, replaced, or deleted. It's best for both performance and efficiency to keep them as intact as possible. That way the file system can record fragments instead of individual allocation blocks. There is absolutely no need for HFS or HFS Plus to keep track of a few hundred separate allocation blocks for one file if those allocation blocks are consecutive. It need only track the number of the first block and the number of blocks in a row that belong to the file.

HFS and HFS Plus call that concept an *extent*. It's defined just that way: a list of contiguous allocation blocks identified by the first allocation block in the extent and the number of allocation blocks in the extent. It therefore only takes two numbers to describe a contiguous 2000-block fork starting at allocation block 4256: the numbers 4256 and 2000. Were it not this way, HFS would have to track 2000 separate two-byte block numbers for the fork. HFS Plus would have to monitor 2000 four-byte block numbers, a waste of two full allocation blocks to replace two numbers.

Other file systems didn't follow this route back in the 1980s because, frankly, it wastes disk space. HFS and HFS Plus need contiguous fragments or this strategy becomes worse than ProDOS or FAT. If one file takes four non-contiguous allocation blocks, that means four separate extents for HFS and HFS Plus, and that's bad for both performance and efficiency.

To get around that, both HFS and HFS Plus allocate storage in multiples of allocation blocks called *clumps*. The default clump for an HFS Plus volume may be something like sixteen allocation blocks, or 64KB of disk space, but each individual file or folder may override that, using a larger clump or no clump at all. Also, the clump is just a hint, not a requirement: implementations do not have to use clumps at all, but they make for fewer extents. Each file's catalog record contains a clump size for both the data and resource forks, and the volume header contains a default clump size for each kind of fork. Traditionally, an implementation that uses clumps uses each fork's own clump size, and if it's empty, it uses the volume's default clump size for that kind of fork instead.

How does it work? An implementation that uses clumps allocates at least one clump's worth of allocation blocks any time it allocates any space for a file at all. On the disks we examined, most files had no clump size set, and in fact, didn't appear to be using clumps: each file used only as many allocation blocks as necessary. In fact, Apple's implementations do allocate by clumps, but when you close a file, any blocks in the clump that the file didn't used are returned to the volume as free space.

Why not go clump crazy and leave the space allocated just in case the file needs it later? The default clump is too much for most files. 64KB is sixteen allocation blocks on an HFS Plus disk, and if the file never grows to use it, those blocks are wasted until you optimize or defragment the disk. The extreme example is still the `PkgInfo` file, deprecated though it is. Spending 64KB of disk space on an eight-byte file would waste almost as much space as using HFS instead of HFS Plus.

Instead, Apple's implementations of HFS and HFS Plus keep track of where the last extent was allocated and try to allocate from that point forward to reduce fragmentation and, therefore, the number of extents. When done writing to a file, any allocation blocks left in the last allocated clump are freed again, eliminating the possibility of small fragments between every extent.

thy extents runneth over

HFS and HFS Plus do not attach a separate list of extents to every file the way ProDOS does for data block numbers. The good news is that if a contiguous file never grows, it only needs one extent, period. The bad news is that it could need as many as one extent per allocation block, and that's a lot to track.

Apple's classic HFS implementation "prefers" to track extents in groups of three, so each file's catalog record has room to track three of the file's extents. As long as the file is in three or fewer fragments, that's all an implementation needs to access the entire file. If it needs more, HFS allocates more extent records for the file in groups of three. Note that those are extents records – places to track extents – and not ranges of allocation blocks themselves. HFS stores these extent-tracking placeholders in the second of its three special files, the *extents overflow file*, sometimes (but less correctly) called just the *extents file*. Like the catalog file, the extents overflow file is stored as a B*-tree, and its location is specified in block 2, not in the catalog file.

The extent overflow tree's records are of only one type: extent data records. Those are far simpler than catalog nodes. An extent data record contains the CNID of the file that owns the extents, a number indicating whether the extents are part of the data or resource fork, and the first allocation block number within the file that these extents store. For example, if the first three extents of an HFS file hold the first 108 blocks of the file, the first allocation block number of the first extent overflow record is 109 – not allocation block #109 on the disk, but the 109th allocation block of the file. After that are the extents themselves: the allocation block number on disk and the length of the extent in blocks.

The key for an extent record in the extents overflow tree is the fork type (0 for data forks and 255 for resource forks) concatenated with the file's CNID and the starting allocation block of the extent. When HFS rebalances the extents overflow tree, these keys naturally group all data forks together on the "left" side and all resource forks on the "right" side, helping the search algorithm choose the right part of the tree from the beginning. All extents for a given file then sort together, followed by the extents within the file sorted by position. It's quite efficient.

Three extents isn't much, though, so HFS Plus increased its "preference" to eight extent records at a time. Each HFS Plus file's catalog record has room for eight data fork extents and eight resource fork extents. Any extents beyond that per fork have to go in the extents overflow file, still found by looking for its information in block 2. The keys are the same as for HFS, though for HFS Plus, block numbers are four bytes each instead of two bytes. The tree sorts the same way, retaining the efficiency in finding extents by fork, file ID, and position within the file.

The default clump size for the extents overflow file itself on an HFS Plus disk depends on the program that initialized it. On a 40GB iMac hard disk, the clump size for the extents overflow file is 3MB, enough room for 768 nodes with around 50 records in each clump. On an original 5GB iPod, though, the extents overflow file clump size is 4MB, enough for 1024 nodes per clump. The problem is fragmentation: the volume header only has room for eight extents per special file, and you start getting into weird territory if you try to find extra extents that belong to the extents overflow file by looking in the extents overflow file. Operating systems therefore try pretty hard to keep those files from fragmenting too much by using big clump sizes. The iPod's clump size is probably larger, even though it's a smaller volume, because the folks formatting the iPod expected it to contain more fragmented files.

Incidentally, defragmenting a volume eliminates this problem: programs like PlusOptimizer and Norton Speed Disk rearrange the blocks on disk so that every file has one and only one extent. A freshly-optimized volume therefore has a nearly-empty extents overflow file, because no file on the disk needs more than one extent to hold all its blocks. Optimizing also defragments the special files, making the disk ready for tens of thousands more files and fragments before any of the special files need new extents of their own.

Most files need only a few extents anyway, so you may not notice any performance difference after defragmenting or optimizing an HFS or HFS Plus disk. Even if a file has eight or more extents, the disk head moves far less than it does under FAT, where the drive has to seek back to the beginning of the disk to find the next block of every file, at least when the entire file allocation table is not cached in RAM. ProDOS also requires lots of disk head movement unless a file's current index block, and perhaps the master index block for tree file, are cached in RAM.

18 ▶

◀17

HFS and HFS Plus generally defeat these problems with clump and extent-oriented disk allocation. Even a system log file that expanded regularly had only three extents on our test system. As long as the file's catalog record is cached, the disk head never has to seek more than twice to find every block in that log file. This approach fragments free space instead of data. It may therefore bog down on a nearly-full volume because every new file would then have to fit in the gaps, fragmenting them far worse than the other files on the disk.

However, most people will trade slower performance on the last files added to a full volume to get fewer fragments and better performance on the other 99% of that volume. Earlier file systems were designed for much smaller and slower disks, and therefore placed a premium on using every possible block of disk space. HFS and HFS Plus are willing to leave a few blocks unused, not only on the disk but also in the catalog and extents overflow files (in the form of unused records) to gain performance.

where to start?

The catalog and extent overflow trees are well-designed structures for fast searching to find the information and location of any file on the disk. So how do you find the catalog and extents overflow files? As noted, you can't look in the catalog file unless you know where it is, so you obviously can't find it by looking in the catalog file. You could find the extents overflow file in the catalog file, if Apple had wanted it that way, but if it were a regular file instead of a "special" file, it might show up in some dialog boxes, or worse, you might find ways to try to delete it. Deleting the extents overflow file on an unoptimized disk would be unspeakably bad.

That's why those two files are "special" files, stored in a specific, unchanging location on each HFS or HFS Plus volume. The details are a bit different between the two file systems, as HFS Plus rectifies a few irregularities that made HFS a bit more complicated than necessary. Consider an 800KB floppy disk, from back in HFS's heyday. Those disks are easily small enough to use 512-byte allocation blocks, so you'd think an 800KB floppy disk would have 1600 allocation blocks. (By the way, we mean "disk" here and not "volume" – floppy disks are too small to have partition maps. And floppy disks use only HFS: the smallest possible HFS Plus volume is 32MB, the point where HFS allocation blocks must grow beyond 512 bytes.)

In fact, floppy disks have 1594 allocation blocks – six blocks on the disk are not part of the HFS allocation block scheme. So where did they go?

HFS volume structures

The first two 512-byte blocks (block 0 and block 1) of every bootable Macintosh volume are Mac OS *boot blocks*. They contain information about the boot volume, and in some cases, machine-language 68K or PowerPC code that can help load the Mac OS. Macintosh hardware rarely uses the boot blocks, and "New World" machines – the original iMac and later machines that don't keep large parts of the Mac OS in ROM – don't use the boot blocks at all, to our knowledge. We'll cover what those machines use instead of boot blocks later.

We've noted that the third 512-byte block of every HFS or HFS Plus volume, block 2, holds information about the volume itself. Like ProDOS, it's always in block 2 so file system implementations always know where to start. For HFS, block 2 is the *master directory block*, and it includes data such as the volume's creation and modification dates, the number of allocation blocks on the volume, how big each allocation block is, the volume's name, the clump sizes for special files and the default clump sizes for other files, the count of files and folders on the disk, and the sizes and extents of the two HFS special files – the catalog file and the extents overflow file. The first two bytes of block two are the HFS signature: "`BD`", as in "big disk." (The original MFS block 2 held the MFS signature "`rw`", for designer Randy Wigginton.) The master directory block also contains the *volume attributes*, a set of up to sixteen flags that tell implementations important things about the volume, such as whether or not it's locked in software, was unmounted properly, if its blocks shouldn't be cached in RAM, and so on.

Immediately after the master directory block in block 3, you'll find a map of the volume that uses a single bit for each allocation block. Just as in ProDOS, if a single bit has value "0," the corresponding allocation block is free; if the value is "1," the corresponding allocation block is in use somewhere. This map that uses bits to describe the volume is, unsurprisingly, the *volume bitmap*. Each 512-byte block contains 4096 bits, far more than the 1600 physical blocks on an 800KB floppy disk, so one block is enough for the floppy's volume bitmap. Larger disks need more blocks, up to a maximum of 16 blocks for an HFS volume with 65,536 allocation blocks. The size of an HFS volume bitmap is determined during initialization and cannot be changed. In theory, the volume bitmap can be anywhere on the volume, since the master directory block contains the volume bitmap's first block number, but all of Apple's HFS implementations put it in block 3.

That's four of the six missing blocks. The other two are at the end of the disk. The last 512-byte block of any HFS volume is reserved because, according to Apple, it's used during the CPU manufacturing process. The next-to-last 512-byte block contains the *alternate master directory block* – a faint copy of the master directory block stored exclusively to help disk repair programs. It's not a true copy of the master directory block: it's only updated when one of HFS's two special files – the catalog file or extents overflow file – grows larger. It doesn't contain an accurate count of files or folders or modification times, but since any repair program *must* find the special files, the alternate master directory block caches their location and size.

Those are the last two, accounting for all six missing 512-byte blocks, and explaining why most 800K floppy disks have 1594 allocation blocks. Some, however, have 1593, in another bit of useless trivia we'll explore shortly.

HFS plus volume structures

The larger HFS Plus file system improves this design but doesn't reinvent it. For starters, every 512-byte block on an HFS Plus volume is part of an allocation block. If you could have an HFS Plus 800KB floppy disk, it would have 1600 512-byte allocation blocks, but the ones containing 512-byte blocks used by HFS Plus itself are marked "used" in the volume bitmap.

HFS Plus, however, calls its volume structure the *volume header* instead of the master directory block. It includes most of the same information, but with some refinements based on a decade of seeing HFS used and abused in real situations. As noted previously, the HFS Plus volume header contains the `lastMountedVersion` field to let code know if a newer or older version of HFS Plus has been writing to the disk. There's space for the last date and time a disk repair program checked the volume, a list of text encodings used on the volume to help Mac OS 8 and Mac OS 9 implementations that don't use Unicode directly, and more volume attributes, such as whether the volume reuses CNID values or whether journaling is enabled.

As with HFS, a faint copy of the volume header is stored in the next to last 512-byte block, updated only when the size of one of the "special" files changes. The volume header contains the fork information – size and first eight extents – for five special files. We've already discussed the catalog and extent overflow files, the only two special files HFS Plus shares with HFS. Now it's time to look at the other three, but we'll start with one that's not.

finding free space

Remember how HFS and HFS Plus keep track of where the last extent was allocated as a hint for where to allocate the next extent? When a disk is new, the hint points to the huge area right after the catalog and extents overflow files. As you allocate new extents, the hint keeps moving forward through the free space. When files get deleted, however, the hint does not move backward. Trying to fill in all the free space from deleted files would promote fragmentation, and both HFS and HFS Plus prefer a volume with some wasted space to one that's packed full but slow to use.

Once the hint goes so far as to point to the end of the volume, HFS and HFS Plus can either report the disk as full, or scavenge for the free space left by deleted files. That raises a different problem: how does HFS or HFS Plus know what allocation blocks are free? The catalog record for each file contains the file's first extents – three of them for an HFS file, eight for an HFS Plus file – and the extents overflow file contains records of all extents that didn't fit in catalog records. Walking the entire catalog and extents overflow trees therefore describes every allocation block on the volume that's in use. All the other blocks must therefore be available.

That's way too much work to find a 4KB allocation block for a new alias or URL clipping, and that's why there's a volume bitmap. The file system implementation powers through at least 32 bits of the map at a time, scanning for any zeroes. As soon as it's found a single zero, it's found a free allocation block.

HFS and HFS Plus differ on where to find the volume bitmap, though. You already know that HFS's volume bitmap almost always starts in 512-byte block #3 and takes a maximum of sixteen blocks. HFS Plus, on the other hand, uses 32 bits for block

20 ▶

◀ 19

numbers instead of the 16-bit numbers HFS allows, so a single HFS Plus disk may have almost 4.3 billion allocation blocks. Even at one bit per block, that's 512MB of disk space just to store the volume bitmap – if you have a 2TB disk. You probably have a smaller disk and don't need a volume bitmap that large. Also, the design decision to leave the volume bitmap out of any HFS allocation block complicates the code in ways Apple didn't want to carry over to HFS Plus.

Therefore, in HFS Plus, the "volume bitmap" is actually the third special file, called the *allocation file*. Its directory record is in the volume header with the directory entries for the catalog and extents overflow files. It's stored in allocation blocks, just like every other file. An implementation or an optimizer may deliberately fragment the allocation file, so that (for example) the bits describing the second half of a huge volume are stored in the second half of the volume, perhaps keeping the drive head from moving so much.

Similarly, since the volume bitmap is in a file and not of fixed size, it's theoretically possible to grow or shrink an HFS Plus volume. If your disk has a free partition after an HFS partition you want to grow, utility software can (at least theoretically) combine the two partitions and grow the allocation file to represent all the new space. HFS disks have a fixed-size volume bitmap that's almost always at the beginning of the disk, immediately followed by the catalog file, preventing the volume bitmap from any chance of growing.

In essence, putting the volume bitmap in a file instead of in a fixed area of the disk makes such utilities somewhat straightforward, even if few of them exist today. The same features under HFS would require rewriting the entire disk and would drive most programmers mad, or close enough that you couldn't tell the difference.

culling the block herd

These behind-the-scenes HFS and HFS Plus structures have logical names: the catalog file, the extents overflow file, the allocation file, the volume header. It makes the complicated structure a little easier to follow.

You knew *that* wouldn't last. After all, this is technology.

Any bad blocks on a disk should be *spared*, or excluded from general use. (A *bad* block, by the way, is any block the disk can't read or write reliably, indicated when the drive returns the dreaded "I/O error" for that block. It's not like "bad" blocks hang around the beginning of the catalog file smoking and drinking vodka.) As late as System 6.0.8, though, the Mac OS refused to use any volume if it found *any* bad blocks on it during initialization. Although the code involved mostly affected floppy drives, a single bad block ruled out the entire disk (and again, we mean disk – floppies don't have partitions or partition maps.)

System 7 was a bit more lenient with questionable disks. If initialization turns up a bad block on a disk, the Disk Initialization Manager goes back through the disk, writing a test bit pattern to every block and reading it back. If any of the blocks return errors or don't give back the same information they got, the system spares all the blocks on the bad block's entire physical track. You know this is happening because the traditional "Erasing…" dialog box is followed by one that says "Re-verifying disk…", but that's all the feedback you get. The system then marks all the allocation blocks containing these bad blocks as "used" in the volume bitmap.

Any disk repair program – even the sanity check the Mac OS performs on volumes that weren't properly unmounted – may see such blocks as a mistake and try to mark them as free. To prevent that, spared blocks must be marked as if they were allocated. They're said to belong to the *bad block file*, but that's wrong: they're not part of any file. "Bad block file" is a colloquial term to refer to the concept of bad blocks marked as used on the disk. It's not a real file, it's not a special file – in fact, it's not a file at all.

A true file of bad blocks would be easier to track, but it would tempt too many programs to pretend it was a real file. A bad block "file" couldn't be defragmented, of course, because it's the physical media that's bad. Since HFS had no room for more "special" files, a bad block file would have to look like a regular file, with all the same potential for disaster – seeing it, trying to rename it, trying to delete it, and so on.

On the other hand, it's not enough just to mark bad blocks as "used" in the volume bitmap (or the allocation file). Any disk repair program, all the way from Mac OS 9's mount-time check through Mac OS X's fsck and bigger fish like Disk First Aid and Norton Disk Doctor, would rightly treat blocks marked as "used" but not part of any file as an error, and mark them as free.

In fact, the HFS Plus specification requires this kind of check when mounting any volume that was not unmounted cleanly. Preferably starting with an empty volume bitmap, any HFS Plus implementation must first mark as used every allocation block used by the volume structures and the "special" files. It then has to walk the catalog tree and mark every block in use by any file's catalog extents, and then do the same for all extents in the extents overflow file, followed by more of the same for one more special file if it exists. When done with all that, any blocks still marked as "free" probably are. This is the chatter your disk makes for a few minutes when you reboot after a crash or kernel panic. Apple says you can't skip this step unless the volume was unmounted cleanly.

This free space reclamation for blocks that are marked "used' but don't belong to any file would defeat the purpose of sparing if implemented that way. Clearly, bad blocks must be part of some HFS or HFS Plus structure. So, as usual, the engineers invented a workaround. Each bad block is not only marked "used" in the volume bitmap, but also added to an extent in the extents overflow file. Disk repair programs see that the block belongs to an extent and is marked as used in the volume bitmap or allocation file, so they leave it alone.

The big utilities, though, would treat an extent not assigned to any file as a sign of damage. The CNID number ties extents to files, you may recall. CNID numbers less than 16 are reserved for HFS itself, so all extents for bad blocks look like they belong to a file with a CNID of 5. Existing disk utilities should have recognized that CNID as belonging to the system and left its extents alone. Just in case, though, Apple documented this use, and also set a previously-reserved bit in the volume attributes to signal that the volume contains spared blocks.

HFS Plus could have changed this system, adding a "special" file for bad blocks like the allocation file that holds the volume bitmap, but in the end, it wouldn't have provided any advantages. Disk utility programs already knew and supported the HFS scheme, so the code was already written. Since the HFS disk organization requires using one extent for every bad block that's not contiguous with another bad block, any more than nine such extents would still require using the extents overflow file to track bad blocks. Switching to a special file would have added more code to disk utilities for no appreciable benefits, so Apple didn't do it.

As for that useless trivia we promised? When sparing blocks on an 800K floppy disk, System 7 and later versions also subtract one allocation block from the disk's total. That's because the System 6 Finder and earlier versions copy disks block-by-block, not file-by-file, if the two disks both contain exactly 1594 allocation blocks – the size of an HFS 800K floppy disk, as described earlier. If there's even one bad block on a disk of 1594 allocation blocks, the system changes count of allocation blocks in the master directory block to 1593 – leaving 512 bytes of disk space unaccounted for – to prevent the Finder from doing a block-by-block disk copy.

the remaining special files

The HFS Plus volume header does point to the first few extents of a few more "special" files, but you've probably never heard of them. One has never been used, and one is only for operating systems other than Mac OS 9.

the startup file

The "New World" machines may not use the traditional boot blocks, but they still need to find information on how to load the operating system. For Mac OS 9, it's stored in the "`Mac OS ROM`" file in the blessed System folder. That file could be anywhere on the disk, but the Open Firmware in all "New World" machines contains enough of an HFS and HFS Plus implementation to find it and read it.

In Mac OS X, the equivalent file is nominally "`/System/Library/CoreServices/BootX`", but that's not quite where the information comes from. Mac OS X is built on top of Darwin, an open-source operating system that may or may not have access HFS or HFS Plus disks. Darwin certainly can boot from other file systems. No one wants to force Darwin to include a full HFS Plus implementation in every kernel on every hardware platform just to find the `BootX` file in case it's on an HFS Plus disk.

The design engineers knew that finding a boot file on an HFS Plus disk is far more difficult than for simple file systems like FAT or ProDOS – finding the catalog tree, searching it, finding the file, and then reading it, including reading and searching the extents overflow tree if the file is fragmented. That's a big pile of code to find a single file.

That's why HFS Plus includes the option for a *startup file* as a "special" file, with its information

22 ▶

◀ 21

stored in the volume header next to the catalog file and extents overflow file. As with the other "special" files, the volume header has room for up to eight extents of a startup file. Finding it is therefore just as simple as with ProDOS or FAT.

An implementation finds the volume header in the block 2 of the partition, and then finds the information for the startup file starting in the 432nd byte of that block. That provides the startup file's size in both bytes and allocation blocks, as well as its first eight extents. All code has to do to find the file is go find the first extent and read it, followed by any other extents. Apple notes that a startup file may have more than eight extents, but since that would require finding the others by searching the B*-tree in the extents overflow file, "doing so defeats the purpose of the startup file."

the attributes file

The final HFS Plus "special" file doesn't exist. It could exist, mind you, and implementations must be prepared to deal with it if it shows up one day, but in the five years since HFS Plus debuted in Mac OS 8.1, there hasn't been a single Macintosh disk with an *attributes file* unless someone made it by hand to test some code.

HFS (and, before it, MFS) had always been loners in the world of file systems thanks to their support of forks. The concept of bundling multiple streams of data as a single file was so unusual that for many years, it earned the ire of programmers on other platforms who didn't want the concept of a *file* expanded beyond opening, reading, writing, and closing. The journals and message boards were full of complaints that standard code for copying files would leave out resource forks, that resource forks didn't transfer "properly" because they were two data streams instead of one, and other such slurs.

These complaints were absurd, as even a moment's thought would show. Complaining that multiple forks "breaks" existing code for handling files is like complaining that JPEG2000 "breaks" existing JPEG code, or that the AppleWorks 6 file format "breaks" AppleWorks 5 code. For some reason, these self-appointed guardians of technical consistency decided that files, alone among all computer constructs, could never evolve beyond the 1960s. It was mainly Unix protectionism. Unix had evolved around treating everything as a single-forked file – devices, files, printers, *everything* – and its partisans didn't want to have to rewrite thousands of programs to deal with a more advanced concept of "file." But since Unix was considered the standard for operating systems, Unix's refusal to allow forked files kept most other operating systems from evolving, too. The Mac OS was not among the laggards, and it was reviled for it.

By the mid-1990s, that was starting to change. Microsoft's NTFS, the file system introduced with Windows NT 3.5 in 1993 or so, supports multiple streams of data per file, but Microsoft calls them *streams* instead of *forks*. NTFS allows for any number of forks per file, and Windows NT uses one of those streams to store what would normally be file metadata – security information like owners and permissions, for example. It's a pretty good idea: by not limiting the size of metadata to fixed-size directory structures (or catalog leaf records as in HFS), the operating system can add more information about each file with any revision. Older versions of the OS just ignore the new information. With both HFS and NTFS supporting forked files, even the maintainers of FreeBSD and Linux were considering the issue by 1997.

Apple's engineers certainly had this broader acceptance and flexible metadata management in mind when designing HFS Plus. By default, HFS Plus is like HFS: each file's catalog entry describes a data fork and a resource fork. However, just as the extents overflow file contains extents that don't fit in the catalog entry, the *attributes file* is supposed to contain forks that don't fit in the catalog entry.

The attributes file contains a B*-tree, just like the catalog and extent overflow files. There are only two defined kinds of records in the tree. The first kind is a fork record that functions like the one for each special file or for each fork in a catalog leaf record – the size of the fork in bytes and allocation blocks, and the first eight extents for the fork. The second kind of record is an extents record, with up to eight more extents for any fork that doesn't fit in the first eight extents. In essence, the attributes file is supposed to be a combination catalog and extents overflow file for any extra fork for any file on the disk.

It's not.

Apple never defined the search keys for records in the attributes file, so it's impossible to actually implement the attributes file right now. Apple only published enough information so that, if the attributes file does exist on some disk, disk utilities can scan it and know what allocation blocks belong to extents described within it. That way, such blocks aren't accidentally marked "free" when repairing a volume or scanning one that was unmounted badly. It's not enough information to find a fork because there's no search key. HFS Plus intends to implement extra forks by naming them, but even if you have the name of the fork, you don't know how to search for it in the tree.

HFS Plus was being implemented for Mac OS 8.1 during the same time Apple was reorganizing its engineering management to include engineers and management from the recently-purchased NeXT Software, Inc. at the same time other Apple engineers were writing the first HFS Plus code. At that time, thanks to interest in NTFS and even Linux developers, multiple forks seemed an important future direction for file systems. In the years since, the disdain that the former NeXT managers carry for forked files (and for most file system concepts that were designed after 1976) has become much clearer than it was at the time.

Apple's current engineering direction pushes both internal and third-party developers to avoid resource forks at all, preferring simple, Unix-endorsed single-stream files. HFS Plus's capability to store multiple forks in any file is entirely unimportant to current Apple management, and not likely to be implemented in the foreseeable future.

the HFS plus future

For a time and half a time, it looked like Apple's current engineering management acutely wanted to kill HFS Plus. Technical Note #2034, *Mac OS X Programming Guidelines*, was based on the internal "ten commandments" that a senior Apple executive imposed on the company's own software developers. Two of those guidelines were to prefer filename extensions to real file types and creator types, and to avoid using resource forks. Those two pieces of metadata are the main differences most users would see if they were using Unix File System (UFS) partitions instead of HFS Plus partitions. The understandable conclusion is that Apple wants everything to run on UFS, so it's discouraging anything HFS Plus has that UFS does not.

That still may be a long-term goal, but it looks less likely with the passing years. At first, the turned-up nose crowd inside and outside of Apple painted HFS Plus as a nasty compromise that carried through all of the odious features of HFS while tacking on security features, Unicode names that can't be typed from the divinely-inspired command line, and more forks rather than fewer. They said it was just an interim file system to satisfy all those ridiculous Mac OS programs until they could be rewritten to use a proper Unix file system.

It hasn't turned out that way. Apple's urges to prefer filename extensions have been resoundingly rejected by the majority of Macintosh developers, and HFS Plus continues to succeed for the same reasons did: it's very fast at finding files and keeps files in as few extents as possible. Arguably, the best file systems let the drive transfer data as quickly as it can, and that means large contiguous blocks, not tiny fragments spread over the disk. HFS Plus's extents handle that well.

Unfortunately, it makes tests like the recent "Mac OS X Filesystem Performance Comparison" by Jason Titus misleading at best. Titus used the open-source IOZone "filesystem benchmark" to test Mac OS X on HFS Plus, Journaled HFS Plus, regular HFS, UFS, and an alpha version of the Linux ext2 file system. He noted such results as "64KB block writes are 10%-20% faster" on Journaled HFS Plus disks compared to HFS Plus disks, or that HFS is "up to 10% faster (usually for 64KB block writes – seems to be a weakness of HFS Plus)." He's also impressed with the alpha-level ext2 filesystem, noting it "has not yet been optimized" and "could greatly improve after a 1.0 release."

But what Titus is actually testing is the *implementation* of these file systems, not the design of the data structures themselves. You now know that the performance of any disk depends on fragmentation, as well as the size of the transfer requests and the buffer sizes that IOZone tests. Was the HFS Plus volume optimized at the time of testing, or at least freshly initialized and empty? Were the other volumes configured the same way? A new file on an empty disk of almost any file system gets stored in what HFS would call a single extent.

IOZone, however, tests with multiple processes running at once, and with different transfer sizes. If two programs simultaneously tried to create 400KB files by writing 4KB at a time to an HFS Plus disk, the first free extent would go to the first program, the second free extent to the second program, and so on, immediately fragmenting both files. That's why programs are supposed to write the entire 400KB in one chunk, so the operating system can do its best work. That's not always practical, as with database opera-

24 ▶

◀ 23

tions, but to think that IOZone's test is completely typical is misleading.

What's more, if all the files are unfragmented, IOZone's test essentially boils down to disk performance. The same drive should read the same 400KB from disk at the same speed, no matter what file system implementation makes the request. The only difference is whether the 400KB is fragmented or not, and that's not something IOZone either controls or tries to test. IOZone tests parameters like the disk cache that a file system really doesn't control. Titus's tests show how the disks he used react under IOZone's type of testing, not any general truths about HFS, HFS Plus, ext2, or UFS.

We know from its design that HFS and HFS Plus are very fast at finding file information so the disk can read or write the information, at the expense of extra disk activity when it has to rebalance one of its B*-trees. We know that other file systems might be a little faster when finding files by pathname, but the Mac OS never did that until Mac OS X, so it's no wonder HFS and HFS Plus don't focus on pathnames. We know that HFS and HFS Plus files are typically less fragmented than in some other file systems, and that usually boosts performance.

And we know that other file systems have borrowed the same concepts HFS introduced in 1986: FAT32 and NTFS both use trees instead of linked lists for directories. The most buzzworthy new file system of the past several years is ReiserFS, a fast Linux file system with journaling and extreme space efficiencies from storing small parts of multiple files in single physical blocks. Those are two of ReiserFS's three big-name features. The third? It uses B*-trees, just like HFS did over fifteen years ago.

ReiserFS's developers say, "Balanced trees are more robust in their performance, and are a more sophisticated algorithmic foundation for a file system. When we started our project, there was a consensus in the industry that balanced trees were too slow for file system usage patterns. We proved that if you just do them right they are better. We have fewer worst-case performance scenarios than other file systems and generally better overall performance. If you put 100,000 files in one directory, we think its fine; many other file systems try to tell you that you are wrong to want to do it."

HFS Plus may not be as hot as ReiserFS, and it's not surprising that the command-line crowd refuses to acknowledge the millions of HFS disks using B*-trees years before anyone had written a single line of ReiserFS code. Even so, Apple now finds itself the owner of a file system that not only uses the hottest file system storage techniques with full support for both Unix-style permissions and Mac OS metadata, but also finds that it's been tested in the field for over five years with solid results, is extensively documented, and rarely causes problems for anyone.

Not even engineering dogma is enough to surrender those advantages. Apple has already added journaling to HFS Plus, and more improvements or features may appear in the next major version of Mac OS X, code-named "Panther," to be shown at WWDC 2003. Implementing new file systems with B*-trees and little fragmentation is complicated; Apple not only has it done, but has had it in the field for over five years.

The company may continue to push programs to not rely on HFS Plus features, since the ability to work with lots of disks makes an OS stronger, not weaker. Also, since Mac OS X tracks Finder information like file types and creator types on non-HFS volumes, and that information must be stored in extra files on those file systems and not in catalog entries, using HFS Plus features slows performance on non-HFS Plus disks. There's something to be said for using the least-common denominator in disk-intensive applications.

Most applications aren't opening and closing files every second, though. Most people want to find files fast, read them fast, and write them reasonably fast. They also don't want to have to use a disk optimizer every week to avoid noticeable performance degradation. HFS Plus delivers all these components in spades. It works well, people who initially rejected its concepts have copied it, and we think it's here to stay.

Staff: **Matt Deatherage, Justin Seal, Nathaniel Irons, Jerry Kindall, John C. Welch, John Gruber**