***** Computer Library Periodicals, Jan 1990 : Doc #14753 *****

Journal: Microsoft Systems Journal Sept 1989 v4 n5 p1(13)

Title: Design goals and implementation of the new High Performance File System. (includes related article on B Trees and B+ Trees)

Author: Duncan, Roy.

Summary: The High Performance File System (HPFS) enhancement to OS/2 Version 1.2 solves all the problems of the File Allocation Table (FAT) file system and is designed to meet the demands expected into the next few decades. HPFS not only serves as a way to organize data on random access block storage devices, but is also a software module that translates file-oriented requests from applications programs to device drivers. HPFS is also an example of an installable file system which makes it possible to access several incompatible volume structures on the same OS/2 system simultaneously. Excellent throughput is achieved by the use of advanced data structures such as intelligent caching, read-ahead and write-behind. Disk space is managed more economically by the use of sectoring. HPFS also includes greatly improved fault tolerance. Applications programs need only simple modifications to make use of extended attributes and long filenames.

Descriptors..
Product: OS-2 Extended Edition 1.2 (Operating system) (product enhancement).
Topic: OS-2
File Management
Enhancements
Data Structures
Disk Space Allocation
Sectoring.
Feature: illustration
table
chart.
Caption: (Comparison of FAT and High Performance File System.)
(Overall structure of an HPFS volume.)
(Overall structure of an Fnode.)

Record#: 07 585 454.

Full Text:

THE HPFS IS A WAY OF ORGANIZING DATA ON A RANDOM ACCESS BLOCK STORAGE DEVICE. IT IS ALSO A SOFTWARE MODULE THAT TRANSLATES FILE-ORIENTED REQUESTS FROM AN APPLICATION PROGRAM INTO MORE PRIMITIVE REQUESTS THAT A DEVICE DRIVER CAN UNDERSTAND.

The High Performance File System (hereafter HPFS), which is making its first appearance in the OS/2 operating system Version 1.2, had its genesis in the network division of Microsoft and was designed by Gordon Letwin, the chief architect of the OS/2 operating system. The HPFS has been designed to meet the demands of increasingly powerful PCs, fixed disks, and networks for many years to come and to serve as a suitable platform for object-oriented languages, applications, and user interfaces.

The HPFS is a complex topic because it incorporates three distinct yet interrelated file system issues.

First, the HPFS is a way of organizing data on a random access block storage device. Second, it is a software module that translates file-oriented requests from an application program into more primitive requests that a device driver can understand, using a variety of creative techniques to maximize performance. Third, the HPFS is a practical illustration of an important new OS/2 feature known as installable file systems.

This article introduces the three aspects of the HPFS. But first, it puts the HPFS in perspective by reviewing some of the problems that led to the system's existence.

1 FAT File System

The so-called FAT file system, which is the file system used in all versions of the MS-DOS operating system to date and in the first two releases of OS/2 (Versions 1.0 and 1.1), has a dual heritage in Microsoft's earliest programming language products and the Digital Research CP/M operating system software originally written for 8080-based and Z-80-based microcomputers. It inherited characteristics from both ancestors that have progressively turned into handicaps in this new era of multitasking, protected mode, virtual memory, and huge fixed disks.

The FAT file system revolves around the File Allocation Table for which it is named. Each logical volume has its own FAT, which serves two important functions: it contains the allocation information for each file on the volume in the form of linked lists of allocation units (clusters, which are power-of-2 multiples of sectors) and it indicates which allocation units are free for assignment to a file that is being created or extended.

The FAT was invented by Bill Gates and Marc McDonald in 1977 as a method of managing disk space in the NCR version of standalone Microsoft Disk BASIC. Tim Paterson, at that time an employee of Seattle Computer Products (SCP), was introduced to the FAT concept when his company shared a booth with Microsoft at the National Computer Conference in 1979. Paterson subsequently incorporated FATs into the file system of 86-DOS, an operating system for SCP's S-100 bus 8086 CPU boards. 86-DOS was eventually purchased by Microsoft and became the starting point for MS-DOS Version 1.0, which was released for the original IBM PC in August 1981.

When the FAT was conceived, it was an excellent solution to disk management, mainly because the floppy disks on which it was used were rarely larger than 1 Mb. On such disks, the FAT was small enough to be held in memory at all times, allowing very fast random access to any part of any file. This proved far superior to the CP/M method of tracking disk space, in which the information about the sectors assigned to a file might be spread across many directory entries, which were in turn scattered randomly throughout the disk directory.

When applied to fixed disks, however, the FAT began to look more like a bug than a feature. It became too large to be held entirely resident and had to be paged into memory in pieces; this paging resulted in many superfluous disk head movements as a program was reading through a file and degraded system throughput. In addition, because the information about free disk space was dispersed across many sectors of FAT, it was impractical to allocate file space contiguously, and file fragmentation became another obstacle to good performance. Moreover, the use of relatively large clusters on fixed disks resulted in a lot of dead space, since an average of one-half cluster was wasted for each file. (Some network servers use clusters as large as 64Kb.)

The FAT file system's restrictions on naming files and directories are inherited from CP/M. When Paterson was writing 86-DOS, one of his primary objectives was to make programs easy to port from CP/M to his new operating system. He therefore adopted CP/M's limits on filenames and extensions so the critical fields of 86-DOS File Control Blocks (FCBs) would look almost exactly like those of CP/M. The sizes of the FCB filename and extension fields were also propagated into the structure of disk directory entries. In due time, 86-DOS became MS-DOS and application programs for MS-DOS proliferated beyond anyone's wildest dreams. Since most of the early programs depended on the structure of FCBs, the 8.3 format for filenames became irrevocably locked into the system.

During the last couple of years, Microsoft and IBM have made valiant attempts to prolong the useful life of the FAT file system by lifting the restrictions on volume sizes, improving allocation strategies, caching pathnames, and moving tables and buffers into expanded memory. But these can only be regarded as temporizing measures, because the fundamental data structures used by the FAT file system are simply not well suited to large random access devices.

The HPFS solves the FAT file system problems mentioned here and many others, but it is not derived in any way from the FAT file system. The architect of the HPFS started with a clean sheet of paper and designed a file system that can take full advantage of a multitasking environment, and that will be able to cope with any sort of disk device likely to arrive on microcomputers during the next decade.

2 HPFS Volume Structure

HPFS volumes are a new partition type—type 7—and can exist on a fixed disk alongside of the several previously defined FAT partition types. IBM-compatible HPFS volumes use a sector size of 512 bytes and have a maximum size of 2199Gb (2^{21} sectors). Although there is no particular reason why floppy disks can't be formatted as HPFS volumes, Microsoft plans to stick with FAT file systems on floppy disks for the foreseeable future. (This ensures that users will be able to transport files easily between MS-DOS and OS/2 systems.)

An HPFS volume has very few fixed structures (Figure 1). Sectors 0-15 of a volume (8Kb) are the BootBlock and contain a volume name, 32-bit volume ID, and a disk bootstrap program. The bootstrap is relatively sophisticated (by MS-DOS standards) and can use the HPFS in a restricted mode to locate and read the operating system files wherever they might be found.

Sectors 16 and 17 are known as the SuperBlock and the SpareBlock respectively. The SuperBlock is only modified by disk maintenance utilities. It contains pointers to the free space bitmaps, the bad block list, the directory block band, and the root directory. It also contains the date that the volume was last checked out and repaired with CHKDSK/F. The SpareBlock contains various flags and pointers that will be discussed later; it is modified, although infrequently, as the system executes.

The remainder of the disk is divided into 8Mb bands. Each band has its own free space bitmap in which a bit represents each sector. A bit is 0 if the sector is in use and 1 if the sector is available. The bitmaps are located at the head or tail of a band so that two bitmaps are adjacent between alternate bands. This allows the maximum contiguous free space that can be allocated to a file to be 16Mb. One band, located at or toward the seek center of the disk, is called the directory block band and receives special treatment (more about this later). Note that the band size is a characteristic of the current implementation and may be changed in later versions of the file system.

3 Files and Fnodes

Every file or directory on an HPFS volume is anchored on a fundamental file system object called an Fnode (pronounced "eff node"). Each Fnode occupies a single sector and contains control and access history information used internally by the file system, extended attributes and access control lists (more about this later), the length and the first 15 characters of the name of the associated file or directory, and an allocation structure (Figure 2). An Fnode is always stored near the file or directory that it represents.

The allocation structure in the Fnode can take several forms, depending on the size and degree of contiguity of the file or directory. The HPFS views a file as a collection of one or more runs or extents of one or more contiguous sectors. Each run is symbolized by a pair of doublewords—a 32-bit starting sector number and a 32-bit length in sectors (this is referred to as run-length encoding). From an application program's point of view, the extents are invisible; the file appears as a seamless stream of bytes.

The space reserved for allocation information in an Fnode can hold pointers to as many as eight runs of sectors of up to 16Mb each. (This maximum run size is a result of the band size and free space bitmap placement only; it is not an inherent limitation of the file system.) Reasonably small files or highly contiguous files can therefore be described completely within the Fnode (Figure 3).

HPFS uses a new method to represent the location of files that are too large or too fragmented for the Fnode and consist of more than eight runs. The Fnode's allocation structure becomes the root for a B+ Tree of allocation sectors, which in turn contain the actual pointers to the file's sector runs (see Figure 4 and the sidebar, "B Trees and B+ Trees"). The Fnode's root has room for 12 elements. Each allocation sector can contain, in addition to various control information, as many as 40 pointers to sector runs. Therefore, a two-level allocation B+ Tree can describe a file of 480 ($12 * 40$) sector runs with a theoretical maximum size of 7.68Gb ($12 * 40 * 16$Mb) in the current implementation (although the 32-bit signed offset parameter for DosChgFilePtr effectively limits file sizes to 2Gb).

In the unlikely event that a two-level B+ Tree is not sufficient to describe a highly fragmented file, the file system will introduce additional levels in the tree as needed. Allocation sectors in the intermediate levels can hold as many as 60 internal (nonterminal) B+ Tree nodes, which means that the descriptive ability of this structure rapidly grows to numbers that are nearly beyond comprehension. For example, a three-level allocation B+ Tree can describe a file with as many as 28,800 ($12 * 60 * 40$) sector runs.

Run-length encoding and B+ Trees of allocation sectors are a memory-efficient way to specify a file's size and location, but they have other significant advantages. Translating a logical file offset into a sector number is extremely fast: the file system just needs to traverse the list (or B+ Tree of lists) of run pointers until it finds the correct range. It can then identify the sector within the run with a simple calculation. Run-length encoding also makes it trivial to extend the file logically if the newly assigned sector is contiguous with the file's previous last sector; the file system merely needs to increment the size doubleword of the file's last run pointer and clear the sector's bit in the appropriate freespace bitmap.

4 Directories

Directories, like files, are anchored on Fnodes. A pointer to the Fnode for the root directory is found in the SuperBlock. The Fnodes for directories other than the root are reached through subdirectory entries in their parent directories.

Directories can grow to any size and are built up from 2Kb directory blocks, which are allocated as four consecutive sectors on the disk. The file system attempts to allocate directory blocks in the directory band, which is located at or near the seek center of the disk. Once the directory band is full, the directory blocks are allocated wherever space is available.

Each 2Kb directory block contains from one to many directory entries. A directory entry contains several fields, including time and date stamps, an Fnode pointer, a usage count for use by disk maintenance programs, the length of the file or directory name, the name itself, and a B Tree pointer. Each entry begins with a word that contains the length of the entry. This provides for a variable amount of flex space at the end of each entry, which can be used by special versions of the file system and allows the directory block to be traversed very quickly (Figure 5).

The number of entries in a directory block varies with the length of names. If the average filename length is 13 characters, an average directory block will hold about 40 entries. The entries in a directory block are sorted by the binary lexical order of their name fields (this happens to put them in alphabetical order for the U.S. alphabet). The last entry in a directory block is a dummy record that marks the end of the block.

When a directory gets too large to be stored in one block, it increases in size by the addition of 2Kb blocks that are organized as a B Tree ("B Trees and B+ Trees"). When searching for a specific name, the file system traverses a directory block until it either finds a match or finds a name that is lexically greater than the target. In the latter case, the file system extracts the B Tree pointer from the entry. If there is no pointer, the search failed; otherwise the file system follows the pointer to the next directory block in the tree and continues the search.

A little back-of-the-envelope arithmetic yields some impressive statistics. Assuming 40 entries per block, a two-level tree of directory blocks can hold 1640 directory entries and a three-level tree can hold an astonishing 65,640 entries. In other words, a particular file can be found (or shown not to exist) in a typical directory of 65,640 files with a maximum of three disk hits—the actual number of disk accesses depending on cache contents and the location of the file's name in the directory block B Tree. That's quite a contrast to the FAT file system, where in the worst case more than 4000 sectors would have to be read to establish that a file was or was not present in a directory containing the same number of files.

The B Tree directory structure has interesting implications beyond its effect on open and find operations. A file creation, renaming, or deletion may result in a cascade of complex operations, as directory blocks are added or freed or names are moved from one block to the other to keep the tree balanced. In fact, a rename operation could theoretically fail for lack of disk space even though the file itself is not growing. In order to avoid this sort of disaster, the HPFS maintains a small pool of free blocks that can be drawn from in a directory emergency; a pointer to this pool of free blocks is stored in the SpareBlock.

5 Extended Attributes

File attributes are information about a file that is maintained by the operating system outside the file's overt storage area. The FAT file system supports only a few simple attributes (read only, system, hidden, and archive) that are actually stored as bit flags in the file's directory entry; these attributes are inspected or modified by special function calls and are not accessible through the normal file open, read, and write calls.

The HPFS supports the same attributes as the FAT file system for historical reasons, but it also supports a new form of file-associated, highly generalized information called Extended Attributes (EAs). Each EA is conceptually similar to an environment variable, taking the form

name–value

except that the value portion can be either a null-terminated (ASCIIZ) string or binary data. In OS/2 1.2, each file or directory can have a maximum of 64Kb of EAs attached to it. This limit may be lifted in a later release of OS/2.

The storage method for EAs can vary. If the EAs associated with a given file or directory are small enough, they will be stored right in the Fnode. If the total size of the EAs is too large, they are stored outside the Fnode in sector runs, and a B+ Tree of allocation sectors can be created to describe the runs. If a single EA gets too large, it can be pushed outside the Fnode into a B+ Tree of its own.

The kernel API functions DosQFileInfo and DosSetFileInfo have been expanded with new information levels that allow application programs to manipulate extended attributes for files. The new functions DosQPathInfo and DosSetPathInfo are used to read or write the EAs associated with arbitrary pathnames. An application program can either ask for the value of a specific EA (supplying a name to be matched) or can obtain all of the EAs for the file or directory at once.

Although application programs can begin to take advantage of EAs as soon as the HPFS is released, support for EAs is an essential component in Microsoft's long-range plans for object-oriented file systems. Information of almost any type can be stored in EAs, ranging from the name of the application that owns the file to names of dependent files to icons to executable code. As the HPFS evolves, its facilities for manipulating EAs are likely to become much more sophisticated. It's easy to imagine, for example, that in future versions the API might be extended with EA functions that are analogous to DosFindFirst and DosFindNext and EA data might get organized into B Trees.

I should note here that in addition to EAs, the LAN Manager version of HPFS will support another class of file-associated information called Access Control Lists (ACLs). ACLs have the same general appearance as EAs and are manipulated in a similar manner, but they are used to store access rights, passwords, and other information of interest in a networking multiuser environment.

6 Installable File Systems

Support for installable file systems has been one of the most eagerly anticipated features of OS/2 Version 1.2. It will make it possible to access multiple incompatible volume structures—FAT, HPFS, CD ROM, and perhaps even UNIX—on the same OS/2 system at the same time, will simplify the life of network implementors, and will open the door to rapid file system evolution and innovation. Installable file systems are, however, only relevant to the HPFS insofar as they make use of the HPFS optional. The FAT file system is still embedded in the OS/2 kernel, as it was in OS/2 1.0 and 1.1, and will remain there as the compatibility file system for some time to come.

An installable file system driver (FSD) is analogous in many ways to a device driver. An FSD resides on the disk in a file that is structured like a dynamic-link library (DLL), typically with a SYS or IFS extension, and is loaded during system initialization by IFS= statements in the CONFIG.SYS file. IFS= directives are processed in the order they are encountered and are also sensitive to the order of DEVICE= statements for device drivers. This lets you load a device driver for a nonstandard device, load a file system driver from a volume on that device, and so on.

Once an FSD is installed and initialized, the kernel communicates with it in terms of logical requests for file opens, reads, writes, seeks, closes, and so on. The FSD translates these requests—using control structures and tables found on the volume itself—into requests for sector reads and writes for which it can call special kernel entry points called File System Helpers (FsHlps). The kernel passes the demands for sector I/O to the appropriate device driver and returns the results to the FSD (Figure 6).

The procedure used by the operating system to associate volumes with FSDs is called dynamic mounting and works as follows. Whenever a volume is first accessed, or after it has been locked for direct access and then unlocked (for example, by a FORMAT operation), OS/2 presents identifying information from the volume to each of the FSDs in turn until one of them recognizes the information. When an FSD claims the volume, the volume is mounted and all subsequent file I/O requests for the volume are routed to that FSD.

7 Performance Issues

The HPFS attacks potential bottlenecks in disk throughput at multiple levels. It uses advanced data structures, contiguous sector allocation, intelligent caching, read-ahead, and deferred writes in order to boost performance.

First, the HPFS matches its data structures to the task at hand: sophisticated data structures (B Trees and B+ Trees) for fast random access to filenames, directory names, and lists of sectors allocated to files or directories, and simple compact data structures (bitmaps) for locating chunks of free space of the appropriate size. The routines that manipulate these data structures are written in assembly language and have been painstakingly tuned, with special focus on the routines that search the freespace bitmaps for patterns of set bits (unused sectors).

Next, the HPFS's main goal—its prime directive, if you will—is to assign consecutive sectors to files whenever possible. The time required to move the disk's read/write head from one track to another far outweighs the other possible delays, so the HPFS works hard to avoid or minimize such head movements by allocating file space contiguously and by keeping control structures such as Fnodes and freespace bitmaps near the things they control. Highly contiguous files also help the file system make fewer requests of the disk driver for more sectors at a time, allow the disk driver to exploit the multisector transfer capabilities of the disk controller, and reduce the number of disk completion interrupts that must be serviced.

Of course, trying to keep files from becoming fragmented in a multitasking system in which many files are being updated concurrently is no easy chore. One strategy the HPFS uses is to scatter newly created files across the disk—in separate bands, if possible—so that the sectors allocated to the files as they are extended will not be interleaved. Another strategy is to preallocate approximately 4Kb of contiguous space to the file each time it must be extended and give back any excess when the file is closed.

If an application knows the ultimate size of a new file in advance, it can assist the file system by specifying an initial file allocation when it creates the file. The system will then search all the free space bitmaps to find a run of consecutive sectors large enough to hold the file. That failing, it will search for two runs that are half the size of the file, and so on.

The HPFS relies on several different kinds of caching to minimize the number of physical disk transfers it must request. Naturally, it caches sectors, as did the FAT file system. But unlike the FAT file system, the HPFS can manage very large caches efficiently and adjusts sector caching on a per-handle basis to the manner in which a file is used. The HPFS also caches pathnames and directories, transforming disk directory entries into an even more compact and efficient in-memory representation.

Another technique that the HPFS uses to improve performance is to preread data it believes the program is likely to need. For example, when a file is opened, the file system will preread and cache the Fnode and the first few sectors of the file's contents. If the file is an executable program or the history information in the file's Fnode shows that an open operation has typically been followed by an immediate sequential read of the entire file, the file system will preread and cache much more of the file's contents. When a program issues relatively small read requests, the file system always fetches data from the file in 2Kb chunks and caches the excess, allowing most read operations to be satisfied from the cache.

Finally, the OS/2 operating system's support for multitasking makes it possible for the HPFS to rely heavily on lazy writes (sometimes called deferred writes or write-behind) to improve performance. When a program requests a disk write, the data is placed in the cache and the cache buffer is flagged as dirty (that is, inconsistent with the state of the data on disk). When the disk becomes idle or the cache becomes saturated with dirty buffers, the file system uses a captive thread from a daemon process to write the buffers to disk, starting with the oldest data.

In general, lazy writes mean that programs run faster because their read requests will almost never be stalled waiting for a write request to complete. For programs that repeatedly read, modify, and write a small working set of records, it also means that many unnecessary or redundant physical disk writes may be avoided. Lazy writes have their dangers, of course, so a program can defeat them on a per-handle basis by setting the write-through flag in the OpenMode parameter for DosOpen, or it can commit data to disk on a per-handle basis with the DosBufReset function.

8 Fault Tolerance

The HPFS's extensive use of lazy writes makes it imperative for the HPFS to be able to recover gracefully from write errors under any but the most dire circumstances. After all, by the time a write is known to have failed, the application has long since gone on its way under the illusion that it has safely shipped the data into disk storage. The errors may be detected by the hardware (such as a "sector not found" error returned by the disk adapter), or they may be detected by the disk driver in spite of the hardware during a read-after-write verification of the data.

The primary mechanism for handling write errors is called a hotfix. When an error is detected, the file system takes a free block out of a reserved hotfix pool, writes the data to that block, and updates the hotfix map. (The hotfix map is simply a series of pairs of doublewords, with each pair containing the number of a bad sector associated with the number of its hotfix replacement. A pointer to the hotfix map is maintained in the SpareBlock.) A copy of the hotfix map is then written to disk, and a warning message is displayed to let the user know that all is not well with the disk device.

Each time the file system requests a sector read or write from the disk driver, it scans the hotfix map and replaces any bad sector numbers with the corresponding good sector holding the actual data. This look-aside translation of sector numbers is not as expensive as it sounds, since the hotfix list need only be scanned when a sector is physically read or written, not each time it is accessed in the cache.

One of CHKDSK's duties is to empty the hotfix map. For each replacement block on the hotfix map, it allocates a new sector that is in a favorable location for the file that owns the data, moves the data from the hotfix block to the newly allocated sector, and updates the file's allocation information (which may involve rebalancing allocation trees and other elaborate operations). It then adds the bad sector to the bad block list, releases the replacement sector back to the hotfix pool, deletes the hotfix entry from the hotfix map, and writes the updated hotfix map to disk.

Of course, write errors that can be detected and fixed on the fly are not the only calamity that can befall a file system. The HPFS designers also had to consider the inevitable damage to be wreaked by power failures, program crashes, malicious viruses and Trojan horses, and those users who turn off the machine without selecting Shutdown in the Presentation Manager Shell. (Shutdown notifies the file system to flush the disk cache, update directories, and do whatever else is necessary to bring the disk to a consistent state.)

The HPFS defends itself against the user who is too abrupt with the Big Red Switch by maintaining a DirtyFS flag in the SpareBlock of each HPFS volume. The flag is only cleared when all files on the volume have been closed and all dirty buffers in the cache have been written out or, in the case of the boot volume (since OS2.INI and the swap file are never closed), when Shutdown has been selected and has completed its work.

During the OS/2 boot sequence, the file system inspects the DirtyFS flag on each HPFS volume and, if the flag is set, will not allow further access to that volume until CHKDSK has been run. If the DirtyFS flag is set on the boot volume, the system will refuse to boot; the user must boot OS/2 in maintenance mode from a diskette and run CHKDSK to check and possibly repair the boot volume.

In the event of a truly major catastrophe, such as loss of the SuperBlock or the root directory, the HPFS is designed to give data recovery the best possible chance of success. Every type of crucial file object—including Fnodes, allocation sectors, and directory blocks—is doubly linked to both its parent and its children and contains a unique 32-bit signature. Fnodes also contain the initial portion of the name of their file or directory. Consequently, CHKDSK can rebuild an entire volume by methodically scanning the disk for Fnodes, allocation sectors, and directory blocks, using them to reconstruct the files and directories and finally regenerating the freespace bitmaps.

9 Application Programs and the HPFS

Each of the OS/2 releases thus far have carried with them a major discontinuity for application programmers who learned their trade in the MS-DOS environment. In OS/2 1.0, such programmers were faced for the first time with virtual memory, multitasking, interprocess communications, and the protected mode restrictions on addressing and direct control of the hardware and were challenged to master powerful new concepts such as threading and dynamic linking. In OS/2 Version 1.1, the stakes were raised even further. Programmers were offered a powerful hardware-independent graphical user interface but had to restructure their applications drastically for an event-driven environment based on objects and message passing.

In OS/2 Version 1.2, it is time for many of the file-oriented programming habits and assumptions carried forward from the MS-DOS environment to fall by the wayside. An application that wishes to take full advantage of the HPFS must allow for long, free-form, mixed-case filenames and paths with few restrictions on punctuation and must be sensitive to the presence of EAs and ACLs. After all, if EAs are to be of any use, it won't suffice for applications to update a file by renaming the old file and creating a new one without also copying the EAs.

But the necessary changes for OS/2 Version 1.2 are not tricky to make. A new API function, DosCopy, helps applications create backups—it essentially duplicates an existing file together with its EAs. EAs can also be manipulated explicitly with DosQFileInfo, DosSetFileInfo, DosQPathInfo, and DosSetPathInfo. A program should call DosQSysInfo at run time to find the maximum possible path length for the system, and ensure that all buffers used by DosChDir, DosQCurDir, and related functions are sufficiently large. Similarly, the buffers used by DosOpen, DosMove, DosGetModName, DosFindFirst, DosFindNext, and like functions must allow for longer filenames. Any logic that folds cases in filenames or tests for the occurrence of only one dot delimiter in a filename must be rethought or eliminated

The other changes in the API will not affect the average application. The functions DosQFileInfo, DosFindFirst, and DosFindNext now return all three sets of times and dates (created, last accessed, last modified) for a file on an HPFS volume, but few programs are concerned with time and date stamps anyway. DosQFsInfo is used to obtain volume labels or disk characteristics just as before, and the use of DosSetFsInfo for volume labels is unchanged. There are a few totally new API functions such as DosFsCtl (analogous to DosDevIOCtl but used for communication between an application and an FSD), DosFsAttach (a sort of explicit mount call), and DosQFsAttach (determines which FSD owns a volume); these are intended mainly for use by disk utility programs.

In order to prevent old OS/2 applications and MS-DOS applications running in the DOS box from inadvertently damaging HPFS files, a new flag bit has been defined in the EXE file header that indicates whether an application is HPFS-aware. If this bit is not set, the application will only be able to search for, open, or create files on HPFS volumes that are compatible with the FAT file system's 8.3 naming conventions. If the bit is set, OS/2 allows access to all files on an HPFS volume, because it assumes that the program knows how to handle long, free-form filenames and will take the responsibility of conserving a file's EAs and ACLs.

10 Summary

The HPFS solves all of the historical problems of the FAT file system. It achieves excellent throughput even in extreme cases—many very small files or a few very large files—by means of advanced data structures and techniques such as intelligent caching, read-ahead, and write-behind. Disk space is used economically because it is managed on a sector basis. Existing application programs will need modification to take advantage of the HPFS's support for extended attributes and long filenames, but these changes will not be difficult. All application programs will benefit from the HPFS's improved performance and decreased CPU use whether they are modified or not.

This article is based on a prerelease version of the HPFS that was still undergoing modification and tuning. Therefore, the final release of the HPFS may differ in some details from the description given here—Ed.

Sidebar: B Trees and B+ Trees

Most programmers are at least passingly familiar with the data structure known as a binary tree. Binary trees are a technique for imposing a logical ordering on a collection of data items by means of pointers, without regard to the physical order of the data.

In a simple binary tree, each node contains some data, including a key value that determines the node's logical position in the tree, as well as pointers to the node's left and right subtrees. The node that begins the tree is known as the root; the nodes that sit at the ends of the tree's branches are sometimes called the leaves.

To find a particular piece of data, the binary tree is traversed from the root, At each node, the desired key is compared with the node's key; if they don't match, one branch of the node's subtree or another is selected based on whether the desired key is less than or greater than the node's key. This process continues until a match is found or an empty subtree is encountered (see Figure A).

Such simple binary trees, although easy to understand and implement, have disadvantages in practice. If keys are not well distributed or are added to the tree in a non-random fashion, the tree can become quite asymmetric, leading to wide variations in tree traversal times.

In order to make access times uniform, many programmers prefer a particular type of balanced tree known as a B Tree. For the purposes of this discussion, the important points about a B Tree are that data is stored in all nodes, more than one data item might be stored in a node, and all of the branches of the tree are of identical length (see Figure B).

The worst-case behavior of a B Tree is predictable and much better than that of a simple binary tree, but the maintenance of a B Tree is correspondingly more complex. Adding a new data item, changing a key value, or deleting a data item may result in the splitting or merging of a node, which in turn forces a cascade of other operations on the tree to rebalance it.

A B+ Tree is a specialized form of B Tree that has two types of nodes: internal which only point to other nodes, and external, which contain the actual data (see Figure C).

The advantage of a B+ Tree over a B Tree is that the internal nodes of the B+ Tree can hold many more decision values than the intermediate-level nodes of a B Tree, so the fan out of the tree is faster and the average length of a branch is shorter. This makes up for the fact that you must always follow a B+ Tree branch to its end to get the data for which you are looking, whereas in a B Tree you may discover the data at an intermediate node or even at the root.