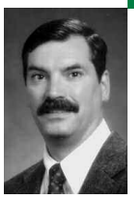
MARSHALL KIRK MCKUSICK

a brief history of the BSD Fast File System



Dr. Marshall Kirk McKusick writes books and articles, teaches classes on UNIX- and BSD-related subjects, and provides expert-witness testimony on software patent, trade secret, and copyright issues, particularly those related to operating systems and file systems. While at the University of California at Berkeley, he implemented the 4.2BSD Fast File System and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD.

mckusick@mckusick.com

I FIRST STARTED WORKING ON THE UNIX file system with Bill Joy in the late 1970s. I wrote the Fast File System, now called UFS, in the early 1980s. In this article, I have written a survey of the work that I and others have done to improve the BSD file systems. Much of this research has been incorporated into other file systems.

1979: Early Filesystem Work

The first work on the UNIX file system at Berkeley attempted to improve both the reliability and the throughput of the file system. The developers improved reliability by staging modifications to critical filesystem information so that the modifications could be either completed or repaired cleanly by a program after a crash [14]. Doubling the block size of the file system improved the performance of the 4.0BSD file system by a factor of more than 2 when compared with the 3BSD file system. This doubling caused each disk transfer to access twice as many data blocks and eliminated the need for indirect blocks for many files.

The performance improvement in the 3BSD file system gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the 3BSD file system was still using only about 4% of the maximum disk throughput. The main problem was that the order of blocks on the free list quickly became scrambled as files were created and removed. Eventually, the free-list order became entirely random, causing files to have their blocks allocated randomly over the disk. This randomness forced a seek before every block access. Although the 3BSD file system provided transfer rates of up to 175 kbytes per second when it was first created, the scrambling of the free list caused this rate to deteriorate to an average of 30 kbytes per second after a few weeks of moderate use. There was no way of restoring the performance of a 3BSD file system except to recreate the system.

1982: Birth of the Fast File System

The first version of the current BSD file system was written in 1982 and became widely distributed in 4.2BSD [13]. This version is still in use today on systems such as Solaris and Darwin. For

large blocks to be used without significant waste, small files must be stored more efficiently. To increase space efficiency, the file system allows the division of a single filesystem block into fragments. The fragment size is specified at the time that the file system is created; each filesystem block optionally can be broken into two, four, or eight fragments, each of which is addressable. The lower bound on the fragment size is constrained by the disk-sector size, which is typically 512 bytes. As disk space in the early 1980s was expensive and limited in size, the file system was initially deployed with a default blocksize of 4 kbytes so that small files could be stored in a single 512-byte sector.

1986: Dropping Disk-Geometry Calculations

The BSD filesystem organization divides a disk partition into one or more areas, each of which is called a cylinder group. Historically, a cylinder group comprised one or more consecutive cylinders on a disk. Although the file system still uses the same data structure to describe cylinder groups, the practical definition of them has changed. When the file system was first designed, it could get an accurate view of the disk geometry, including the cylinder and track boundaries, and could accurately compute the rotational location of every sector. By 1986, disks were hiding this information, providing fictitious numbers of blocks per track, tracks per cylinder, and cylinders per disk. Indeed, in modern RAID arrays, the "disk" that is presented to the file system may really be composed from a collection of disks in the RAID array.

Although some research has been done to figure out the true geometry of a disk [5, 10, 23], the complexity of using such information effectively is high. Modern disks have greater numbers of sectors per track on the outer part of the disk than on the inner part, which makes calculation of the rotational position of any given sector complex to calculate. So in 1986, all the rotational layout code was deprecated in favor of laying out files using numerically close block numbers (sequential being viewed as optimal), with the expectation that this would give the best performance. Although the cylinder group structure is retained, it is used only as a convenient way to manage logically close groups of blocks.

1987: Filesystem Stacking

The early vnode interface was simply an object-oriented interface to an underlying file system. By 1987, demand had grown for new filesystem features. It became desirable to find ways of providing them without having to modify the existing and stable filesystem code. One approach is to provide a mechanism for stacking several file systems on top of one another [22B]. The stacking ideas were refined and implemented in the 4.4BSD system [7]. The bottom of a vnode stack tends to be a disk-based file system, whereas the layers used above it typically transform their arguments and pass on those arguments to a lower layer.

Stacking uses the mount command to create new layers. The mount command pushes a new layer onto a vnode stack; a umount command removes a layer. Like the mounting of a file system, a vnode stack is visible to all processes running on the system. The mount command identifies the underlying layer in the stack, creates the new layer, and attaches that layer into the filesystem name space. The new layer can be attached to the same place as the old layer (covering the old layer) or to a different place in the tree (allowing both layers to be visible).

When a file access (e.g., an open, read, stat, or close) occurs to a vnode in the stack, that vnode has several options:

- Do the requested operations and return a result.
- Pass the operation without change to the next-lower vnode on the stack. When the operation returns from the lower vnode, it may modify the results or simply return them.
- Modify the operands provided with the request and then pass it to the next-lower vnode. When the operation returns from the lower vnode, it may modify the results or simply return them.

If an operation is passed to the bottom of the stack without any layer taking action on it, then the interface will return the error "operation not supported."

The simplest filesystem layer is *nullfs*. It makes no transformations on its arguments, simply passing through all requests that it receives and returning all results that it gets back. Although it provides no useful functionality if it is simply stacked on top of an existing vnode, *nullfs* can provide a loopback file system by mounting the file system rooted at its source vnode at some other location in the filesystem tree. The code for *nullfs* is also an excellent starting point for designers who want to build their own filesystem layers. Examples that could be built include a compression layer or an encryption layer.

The *union* file system is another example of a middle filesystem layer. Like *nullfs*, it does not store data but just provides a name-space transformation. It is loosely modeled on the work on the 3-D file system [9], on the Translucent file system [8], and on the Automounter [19]. The *union* file system takes an existing file system and transparently overlays the latter on another file system. Unlike most other file systems, a union mount does not cover up the directory on which the file system is mounted. Instead, it shows the logical merger of the two directories and allows both directory trees to be accessible simultaneously [18].

1988: Raising the Blocksize

By 1988, disk capacity had risen enough that the default blocksize was raised to 8-kbyte blocks with 1-kbyte fragments. Although this meant that small files used a minimum of two disk sectors, the nearly doubled throughput provided by doubling the blocksize seemed a reasonable tradeoff for the measured 1.4% of additional wasted space.

1990: Dynamic Block Reallocation

Through most of the 1980s, the optimal placement for files was to lay them out using every other block on the disk. By leaving a gap around each allocated block, the disk had time to schedule the next read or write following the completion of the previous operation. With the advent of disk-track caches and the ability to handle multiple outstanding requests (tag queueing) in the late 1980s, it became desirable to begin laying files out contiguously on the disk.

The operating system has no way of knowing how big a file will be when it is first opened for writing. If it assumes that all files will be big and tries to place them in its largest area of available space, it will soon have only small areas of contiguous space available. Conversely, if it assumes that all files will be small and tries to place them in its areas of fragmented space, then the beginning of files that do grow large will be poorly laid out.

To avoid these problems the file system was changed in 1990 to do dynamic block reallocation. The file system initially places the file's blocks in small areas of free space, but then moves them to larger areas of free space as the file grows. With this technique, small files use the small chunks of free space whereas the large ones get laid out contiguously in the large areas of free space. The algorithm does not tend to increase I/O load, because the buffer cache generally holds the file contents long enough that the final block allocation has been determined by the first time that the file data is flushed to disk.

The effect of this algorithm is that the free space remains largely unfragmented even after years of use. A Harvard study found only a 15% degradation in throughput on a three-year-old file system versus a 40% degradation on an identical file system that had had the dynamic reallocation disabled [25].

1996: Soft Updates

In file systems, metadata (e.g., directories, inodes, and free block maps) gives structure to raw storage capacity. Metadata provides pointers and descriptions for linking multiple disk sectors into files and identifying those files. To be useful for persistent storage, a file system must maintain the integrity of its metadata in the face of unpredictable system crashes, such as power interruptions and operating system failures. Because such crashes usually result in the loss of all information in volatile main memory, the information in nonvolatile storage (i.e., disk) must always be consistent enough to deterministically reconstruct a coherent filesystem state. Specifically, the on-disk image of the file system must have no dangling pointers to uninitialized space, no ambiguous resource ownership caused by multiple pointers, and no unreferenced live resources. Maintaining these invariants generally requires sequencing (or atomic grouping) of updates to small on-disk metadata objects.

Traditionally, the file system used synchronous writes to properly sequence stable storage changes. For example, creating a file involves first allocating and initializing a new inode and then filling in a new directory entry to point to it. With the synchronous write approach, the file system forces an application that creates a file to wait for the disk write that initializes the on-disk inode. As a result, filesystem operations such as file creation and deletion proceed at disk speeds rather than processor or memory speeds [15, 17, 24]. Since disk access times are long compared to the speeds of other computer components, synchronous writes reduce system performance.

The metadata update problem can also be addressed with other mechanisms. For example, one can eliminate the need to keep the on-disk state consistent by using NVRAM technologies, such as an uninterruptible power supply or Flash RAM [16, 31]. Filesystem operations can proceed as soon as the block to be written is copied into the stable store, and updates can propagate to disk in any order and whenever it is convenient. If the system fails, unfinished disk operations can be completed from the stable store when the system is rebooted.

Another approach is to group each set of dependent updates as an atomic operation with some form of write-ahead logging [3, 6] or shadow-paging [2, 22A, 26]. These approaches augment the on-disk state with a log of filesystem updates on a separate disk or in stable store. Filesystem operations can then proceed as soon as the operation to be done is written into

the log. If the system fails, unfinished filesystem operations can be completed from the log when the system is rebooted. Many modern file systems successfully use write-ahead logging to improve performance compared to the synchronous write approach.

In Ganger and Patt [4], an alternative approach called soft updates was proposed and evaluated in the context of a research prototype. Following a successful evaluation, a production version of soft updates was written for BSD in 1996. With soft updates, the file system uses delayed writes (i.e., write-back caching) for metadata changes, tracks dependencies between updates, and enforces these dependencies at write-back time. Because most metadata blocks contain many pointers, cyclic dependencies occur frequently when dependencies are recorded only at the block level. Therefore, soft updates track dependencies on a per-pointer basis, which allows blocks to be written in any order. Any still-dependent updates in a metadata block are rolled back before the block is written and rolled forward afterward. Thus, dependency cycles are eliminated as an issue. With soft updates, applications always see the most current copies of metadata blocks, and the disk always sees copies that are consistent with its other contents.

1999: Snapshots

In 1999, the file system added the ability to take snapshots. A filesystem snapshot is a frozen image of a file system at a given instant in time. Snapshots support several important features, including the ability to provide backups of the file system at several times during the day and the ability to do reliable dumps of live file systems.

Snapshots may be taken at any time. When taken every few hours during the day, they allow users to retrieve a file that they wrote several hours earlier and later deleted or overwrote by mistake. Snapshots are much more convenient to use than dump tapes and can be created much more frequently.

To make a snapshot accessible to users through a traditional filesystem interface, the system administrator uses the mount command to place the replica of the frozen file system at whatever location in the namespace is convenient.

Once filesystem snapshots are available, it becomes possible to safely dump live file systems. When dump notices that it is being asked to dump a mounted file system, it can simply take a snapshot of the file system and run over the snapshot instead of on the live file system. When dump completes, it releases the snapshot.

2001: Raising the Blocksize, Again

By 2001 disk capacity had risen enough that the default blocksize was raised to 16-kbyte blocks with 2-kbyte fragments. Although this meant that small files used a minimum of four disk sectors, the nearly doubled throughput provided by doubling the blocksize seemed a reasonable tradeoff for the measured 2.9% of additional wasted space.

2002: Background Fsck

Traditionally, after an unclean system shutdown, the filesystem check program, fsck, has had to be run over all the inodes in a file system to ascer-

tain which inodes and blocks are in use and to correct the bitmaps. This check is a painfully slow process that can delay the restart of a big server for an hour or more. Soft updates guarantee the consistency of all filesystem resources, including the inode and block bitmaps. With soft updates, the only inconsistency that can arise in the file system (barring software bugs and media failures) is that some unreferenced blocks may not appear in the bitmaps and some inodes may have to have overly high link counts reduced. Thus, it is completely safe to begin using the file system after a crash without first running fsck. However, some filesystem space may be lost after each crash. Thus, there is value in having a version of fsck that can run in the background on an active file system to find and recover any lost blocks and adjust inodes with overly high link counts.

With the addition of snapshots, the task becomes simple, requiring only minor modifications to the standard fsck. When run in background cleanup mode, fsck starts by taking a snapshot of the file system to be checked. Fsck then runs over the snapshot filesystem image doing its usual calculations, just as in its normal operation. The only other change comes at the end of its run, when it wants to write out the updated versions of the bitmaps. Here, the modified fsck takes the set of blocks that it finds were in use at the time of the snapshot and removes this set from the set marked as in use at the time of the snapshot—the difference is the set of lost blocks. It also constructs the list of inodes whose counts need to be adjusted, then uses a new system call to notify the file system of the identified lost blocks so that it can replace them in its bitmaps. It also gives the set of inodes whose link counts need to be adjusted; those inodes whose link count is reduced to zero are truncated to zero length and freed. When fsck completes, it releases its snapshot. The complete details of how background fsck is implemented can be found in McKusick [11, 12].

2003: Multi-Terabyte Support

The original BSD fast file system and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. At the time of its design in the early 1980s, the largest disks were 330 Mbytes. There was debate at the time whether it was worth squandering 32 bits per block pointer rather than using the 24-bit block pointers of the file system it replaced. Luckily, the futurist view prevailed, and the design used 32-bit block pointers.

Over the 20 years since it has been deployed, storage systems have grown to hold over a terabyte of data. Depending on the blocksize configuration, the 32-bit block pointers of the original file system run out of space in the 1-to-4-terabyte range. Although some stopgap measures can be used to extend the maximum-size storage systems supported by the original file system, by 2002 it became clear that the only long-term solution was to use 64-bit block pointers. Thus, we decided to build a new file system, one that would use 64-bit block pointers.

We considered the alternatives of trying to make incremental changes to the existing file system versus importing another existing file system such as XFS [27] or ReiserFS [20]. We also considered writing a new file system from scratch so that we could take advantage of recent filesystem research and experience. We chose to extend the original file system, because this approach allowed us to reuse most of its existing code base. The benefits of this decision were that the 64-bit-block–based file system was developed and deployed quickly, it became stable and reliable rapidly, and the same code base could be used to support both 32-bit-block and 64-bit-block

filesystem formats. Over 90% of the code base is shared, so bug fixes and feature or performance enhancements usually apply to both filesystem formats.

At the same time that the file system was updated to use 64-bit block pointers, an addition was made to support extended attributes. Extended attributes are a piece of auxiliary data storage associated with an inode that can be used to store auxiliary data that is separate from the contents of the file. The idea is similar to the concept of data forks used in the Apple file system [1]. By integrating the extended attributes into the inode itself, it is possible to provide the same integrity guarantees as are made for the contents of the file itself. Specifically, the successful completion of an fsync system call ensures that the file data, the extended attributes, and all names and paths leading to the names of the file are in stable store.

2004: Access-Control Lists

Extended attributes were first used to support an access control list, generally referred to as an ACL. An ACL replaces the group permissions for a file with a more specific list of the users who are permitted to access the files. The ACL also includes a list of the permissions each user is granted. These permissions include the traditional read, write, and execute permissions, along with other properties such as the right to rename or delete the file [21].

Earlier implementations of ACLs were done with a single auxiliary file per file system that was indexed by the inode number and had a small fixed-sized area to store the ACL permissions. The small size kept the size of the auxiliary file reasonable, since it had to have space for every possible inode in the file system. There were two problems with this implementation. The fixed size of the space per inode to store the ACL information meant that it was not possible to give access to long lists of users. The second problem was that it was difficult to atomically commit changes to the ACL list for a file, since an update required that both the file inode and the ACL file be written in order to have the update take effect [28].

Both problems with the auxiliary file implementation of ACLs are fixed by storing the ACL information directly in the extended-attribute data area of the inode. Because of the large size of the extended attribute data area (a minimum of 8 kbytes and typically 32 kbytes), long lists of ACL information can be stored easily. Space used to store extended attribute information is proportional to the number of inodes with extended attributes and the size of the ACL lists they use. Atomic updating of the information is much easier, since writing the inode will update the inode attributes and the set of data that it references, including the extended attributes in one disk operation. Although it would be possible to update the old auxiliary file on every fsync system call done on the file system, the cost of doing so would be prohibitive. Here, the kernel knows whether the extended attribute data block for an inode is dirty and can write just that data block during an fsync call on the inode.

2005: Mandatory Access Controls

The second use for extended attributes was for data labeling. Data labels provide permissions for a mandatory access control (MAC) framework enforced by the kernel. The kernel's MAC framework permits dynamically introduced system-security modules to modify system security functional-

ity. This framework can be used to support a variety of new security services, including traditional labeled mandatory access control models. The framework provides a series of entry points that are called by code supporting various kernel services, especially with respect to access control points and object creation. The framework then calls out to security modules to offer them the opportunity to modify security behavior at those MAC entry points. Thus, the file system does not codify how the labels are used or enforced. It simply stores the labels associated with the inode and produces them when a security module needs to query them to do a permission check [29, 30].

2006: Symmetric Multi-Processing

In the late 1990s, the FreeBSD Project began the long hard task of converting their kernel to support symmetric multi-processing. The initial step was to add a giant lock around the entire kernel to ensure that only one processor at a time could be running in the kernel. Each kernel subsystem was brought out from under the giant lock by rewriting it to be able to be executed by more than one processor at a time. The vnode interface was brought out from under the giant lock in 2004. The disk subsystem became multi-processor–safe in 2005. Finally, in 2006, the fast file system was overhauled to support symmetric multi-processing, completing the giant-free path from system call to hardware.

Further Information

For those interested in learning more about the history of BSD, additional information is available from http://www.mckusick.com/history/.

REFERENCES

[1] Apple, "Mac OS X Essentials, Chapter 9 Filesystem, Section 12 Resource Forks" (2003): http://developer.apple.com/techpubs/macosx/Essentials/SystemOverview/FileSystem/chapter_9_section_12.html.

[2] D. Chamberlin and M. Astrahan, "A History and Evaluation of System R," *Communications of the ACM* (24, 10) (1981), pp. 632–646.

[3] S. Chutani, O. Anderson, M. Kazar, W. Mason, and R. Sidebotham, "The Episode File System," *USENIX Winter 1992 Technical Conference Proceedings* (January 1992), pp. 43–59.

[4] G. Ganger and Y. Patt, "Metadata Update Performance in File Systems," *First USENIX Symposium on Operating Systems Design and Implementation* (November 1994), pp. 49–60.

[5] J. L. Griffin, J. Schindler, S.W. Schlosser, J.S. Bucy, and G.R. Ganger, "Timing-accurate Storage Emulation," *Proceedings of the USENIX Conference on File and Storage Technologies* (January 2002), pp. 75–88.

[6] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *ACM Symposium on Operating Systems Principles* (November 1987), pp. 155–162.

[7] J. S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers," *ACM Transactions on Computer Systems* (12, 1) (February 1994), pp. 58–89.

[8] D. Hendricks, "A Filesystem for Software Development," *USENIX Summer 1990 Technical Conference Proceedings* (June 1990), pp. 333–340.

[9] D. Korn and E. Krell, "The 3-D File System," *USENIX Summer 1989 Technical Conference Proceedings* (June 1989), pp. 147–156.

[10] C.R. Lumb, J. Schindler, and G.R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," *Proceedings of the USENIX Conference on File and Storage Technologies* (January 2002), pp. 275–288.

[11] M.K. McKusick, "Running Fsck in the Background," *Proceedings of the BSDCon 2002 Conference* (February 2002), pp. 55–64.

[12] M.K. McKusick, "Enhancements to the Fast Filesystem to Support Multi-Terabyte Storage Systems," *Proceedings of the BSDCon 2003 Conference* (September 2003), pp. 79–90.

[13] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems* (2, 3) (August 1984), pp. 181–197.

[14] M.K. McKusick and T.J. Kowalski, "Fsck: The UNIX File System Check Program," in *4.4BSD System Manager's Manual* (Sebastopol, CA: O'Reilly & Associates, 1994), vol. 3, pp. 1–21.

[15] L. McVoy and S. Kleiman, "Extent-like Performance from a UNIX File System," *USENIX Winter 1991 Technical Conference Proceedings* (January 1991), pp. 33–44.

[16] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon, "Breaking Through the NFS Performance Barrier," *Proceedings of the Spring 1990 European UNIX Users Group Conference* (April 1990), pp. 199–206.

[17] J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *USENIX Summer 1990 Technical Conference* (June 1990), pp. 247–256.

[18] J. Pendry and M.K. McKusick, "Union Mounts in 4.4BSD-Lite," *USENIX 1995 Technical Conference Proceedings* (January 1995), pp. 25–33.

[19] J. Pendry and N. Williams, "AMD: The 4.4BSD Automounter Reference Manual," in *4.4BSD System Manager's Manual* (Sebastopol, CA: O'Reilly & Associates, 1994), vol. 13, pp. 1–57.

[20] H. Reiser, "The Reiser File System" (January 2001): http://www.namesys.com/res_whol.shtml.

[21] T. Rhodes, "FreeBSD Handbook, Chapter 3, Section 3.3 File System Access Control Lists" (2003): http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/fs-acl.html.

[22A] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer System* (10, 1) (February 1992): 26–52.

[22B] D. Rosenthal, "Evolving the Vnode Interface," *USENIX Winter 1990 Technical Conference Proceedings* (June 1990), pp. 107–118.

[23] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger, "Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics," *Proceedings of the USENIX Conference on File and Storage Technologies* (January 2002), pp. 259–274.

[24] M. Seltzer, K. Bostic, M.K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the USENIX Winter 1993 Conference* (January 1993), pp. 307–326.

[25] K. Smith and M. Seltzer, "A Comparison of FFS Disk Allocation Algorithms," *Proceedings of the USENIX 1996 Annual Technical Conference* (January 1996), pp. 15–25.

[26] M. Stonebraker, "The Design of the POSTGRES Storage System," *Very Large DataBase Conference* (1987), pp. 289–300.

[27] A. Sweeney, D. Doucette, C. Anderson, W. Hu, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proceedings of the 1996 USENIX Annual Technical Conference* (January 1996), pp. 1–14.

[28] R. Watson, "Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD," *Proceedings of the BSDCon 2000 Conference* (September 2000).

[29] R. Watson, "TrustedBSD: Adding Trusted Operating System Features to FreeBSD," *Proceedings of the FREENIX Track at the 2001 USENIX Annual Technical Conference* (June 2001), pp. 15–28.

[30] R. Watson, W. Morrison, C. Vance, and B. Feldman, "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0," *Proceedings of the FREENIX Track at the 2003 USENIX Annual Technical Conference* (June 2003), pp. 285–296.

[31] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 1994), pp. 86–97.