SEE
TEXT
ONLY

A Brief History of the BSD FAST FILESYSTEM

by Marshall Kirk McKusick

Following is a taxonomy of filesystem and storage development from 1979 to the present, with the BSD Fast Filesystem as its focus. It describes the early performance work done by increasing the disk blocksize and by being aware of the disk geometry and using that knowledge to optimize rotational layout. With the abstraction of the geometry in the late 1980s and the ability of the hardware to cache and handle multiple requests, filesystems performance ceased trying to track geometry and instead sought to maximize performance by doing contiguous file layout. Small file performance was optimized through the use of techniques such as journaling and soft updates. By the late 1990s, filesystems had to be redesigned to handle the ever-growing disk capacities. The addition of snapshots allowed for faster and more frequent backups. Multiprocessing support got added to utilize all the CPUs found in the increasingly ubiquitous multi-core processors. The increasingly harsh environment of the Internet required greater data protection provided by access-control lists and mandatory-access controls. Ultimately, the addition of metadata optimization brings us to the present and possible future directions.

1979: Early Filesystem Work

The first work on the UNIX filesystem at University of California, Berkeley attempted to improve both the reliability and the throughput of the filesystem. 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 [15]. Doubling the blocksize of the filesystem improved the performance of the 4.0 BSD filesystem by a factor of more than 2 when compared with the 3 BSD filesystem. 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 3 BSD filesystem gave a strong indication that increasing the blocksize was a good method for improving throughput. Although the throughput had doubled, the 3 BSD filesystem 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 3 BSD filesystem provided transfer rates of up to 175 Kbyte per second when it was first created, the scrambling of the free list caused this rate to deteriorate to an average of 30 Kbyte per second after a few weeks of moderate use. There was no way of restoring the performance of a 3 BSD filesystem except to recreate the system.

1982: Birth of the Fast Filesystem

The first version of the current BSD filesystem was written in 1982 and became widely distributed in 4.2 BSD [14]. 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 filesystem allows the division of a single filesystem block into one or more fragments. The fragment size is specified at the time that the filesystem 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 filesystem was initially deployed with a default blocksize of 4,096 so that small files could be stored in a single 512-byte sector.

1986: Dropping the 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 filesystem still uses the same data structure to describe cylinder groups, the practical definition of them has changed. When the filesystem 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 filesystem may really be composed from a collection of disks in the RAID array. While some research has been done to figure out the true geometry of a disk [5, 10, 25], 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 the inner part that 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) believing that 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 filesystem. 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 filesystems on top of one another [24]. The stacking ideas were refined and implemented in the 4.4 BSD system [7]. The bottom of a vnode stack tends to be a disk-based filesystem, 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; an unmount command removes a layer. Like the mounting of a filesystem, 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:

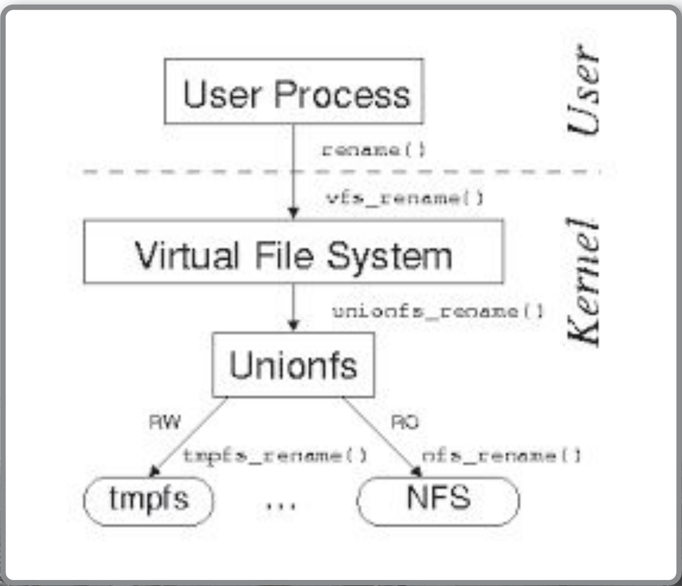
•Do the requested operations and return a result.

•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 filesystem by mounting the filesystem 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 filesystem is another example of a middle filesystem layer. Like the nullfs, it does not store data but just provides a name-space transformation. It is loosely modeled on the work on the 3-D filesystem [9], on the Translucent filesystem [8], and on the Automounter [20]. The union filesystem takes an existing filesystem and transparently overlays the latter on another filesystem. Unlike most other filesystems, a union mount does not cover up the directory on which the filesystem is mounted. Instead, it shows the logical merger of both directories and allows both directory trees to be accessible simultaneously [19].



1988: Raising the Blocksize

By 1988 disk capacity had risen enough that the default blocksize was raised to 8,196-byte blocks with 1,024-byte 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 between 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 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.

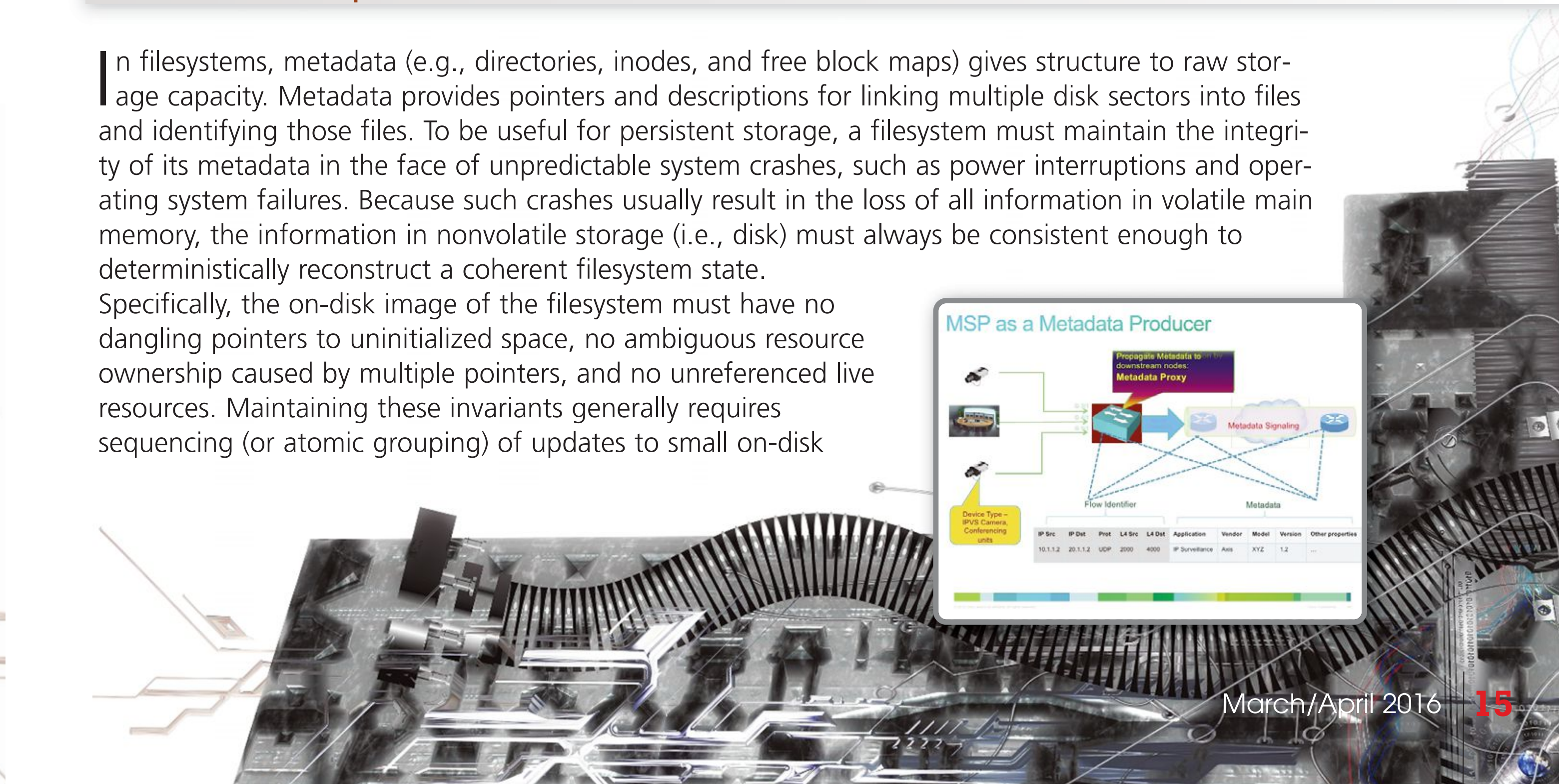
The operating system has no way of knowing how big a file will be when it is first opened for writing. If it assumes that all files will be big and thus tries to place them in its largest area of available space, it will soon have only small areas of contiguous space available. Conversely, if it assumes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

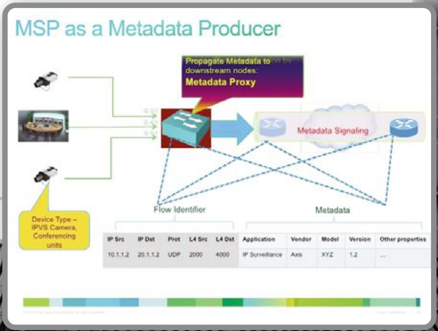
To avoid these problems, the filesystem was changed in 1990 to do dynamic block reallocation. The filesystem initially places the file's blocks in small areas of free space, but then moves them to larger areas of free space as the file grows. Using this technique, small files use the small chunks of free space while the large ones get laid out contiguously in the large areas of free space. The algorithm does not tend to increase I/O load as 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 filesystem versus a 40% degradation on an identical filesystem that had had the dynamic reallocation disabled [27].

1996: Soft Updates

In filesystems, 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 filesystem 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 filesystem 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





IP Src	IP Dst	Prot	L4 Src	L4 Dst	Application	Vendor	Model	Version	Other properties
10.1.1.2	20.1.1.2	UDP	2000	4000	IP Surveillance	Axis	XYZ	1.2	...

metadata objects.

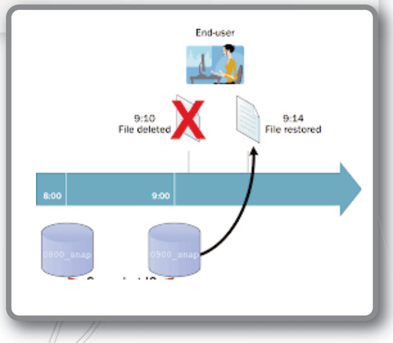
Traditionally, the filesystem used synchronous writes to properly sequence stable storage changes. For example, creating a file involves first allocating and initializing a new inode and then filling in a new directory entry to point to it. With the synchronous write approach, the filesystem forces an application that creates a file to wait for the disk write that initializes the on-disk inode. As a result, filesystem operations like file creation and deletion proceed at disk speeds rather than processor/memory speeds [16,18, 26]. 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 [17, 33]. 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, 28]. 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 filesystems successfully use write-ahead logging to improve performance compared to the synchronous write approach.

An alternative approach called soft updates was evaluated in the context of a research prototype [4]. Following a successful evaluation, a production version of soft updates was written for BSD in 1996. With soft updates, the filesystem 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 afterwards. 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 filesystem added the ability to take snapshots. A filesystem snapshot is a frozen image of a filesystem at a given instant in time. Snapshots support several important features: the ability to provide backups of the filesystem at several times during the day and the ability to do reliable dumps of live filesystems.

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 filesystem at whatever location in the name space that is convenient.

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

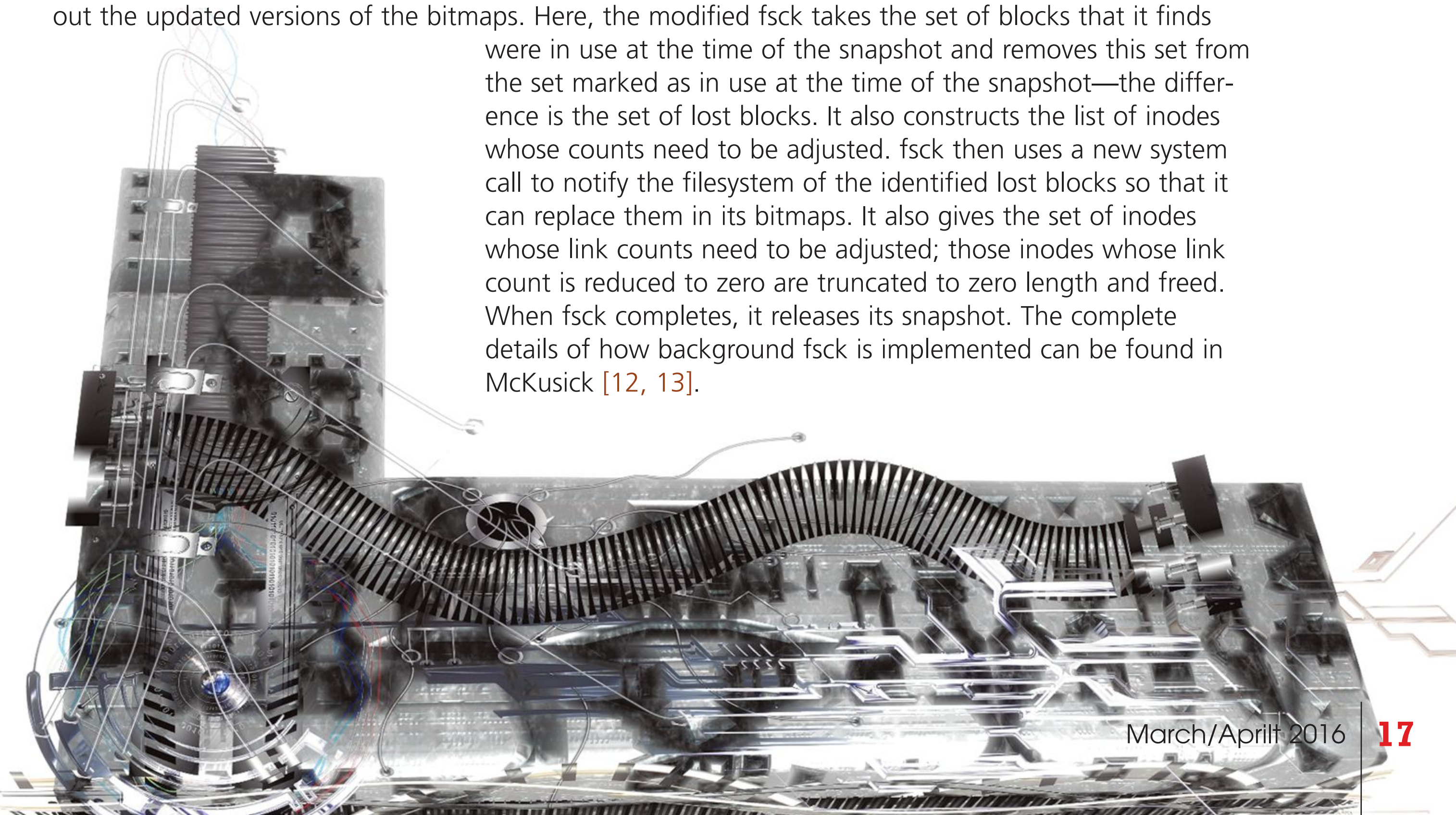
2001: Raising the Blocksize, Again

By 2001 disk capacity had risen enough that the default blocksize was raised to 16,384-byte blocks with 2,048-byte 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 trade-off 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 filesystem to ascertain 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 filesystem (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 filesystem 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 filesystem 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 filesystem 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. fsck then uses a new system call to notify the filesystem 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 [12, 13].



2003: Multi-terabyte Support

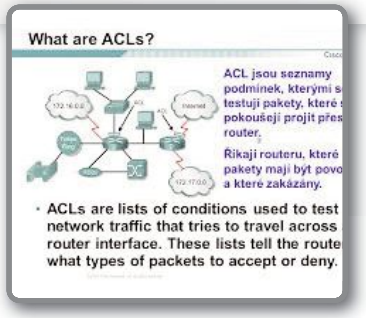
The original BSD fast filesystem and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. At the time of its design in the early 1980s, the largest disks were 330 Mbytes. There was debate at the time whether it was worth squandering 32 bits per block pointer rather than using the 24-bit block pointers of the filesystem that 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 filesystem run out of space in the 1 to 4 terabyte range. While some stopgap measures can be used to extend the maximum-size storage systems supported by the original filesystem, by 2002 it became clear the only long-term solution was to use 64-bit block pointers. Thus, we decided to build a new filesystem, that would use 64-bit block pointers.

We considered the alternatives between trying to make incremental changes to the existing filesystem versus importing another existing filesystem such as XFS [29], or ReiserFS [21]. We also considered writing a new filesystem from scratch so that we could take advantage of recent filesystem research and experience. We chose to extend the original filesystem 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 filesystem 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 filesystem 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 filesystem [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



What are ACLs?

ACL jsou seznamy podmínek, kterými s testují pakety, které s pokoušejí projít přes router.

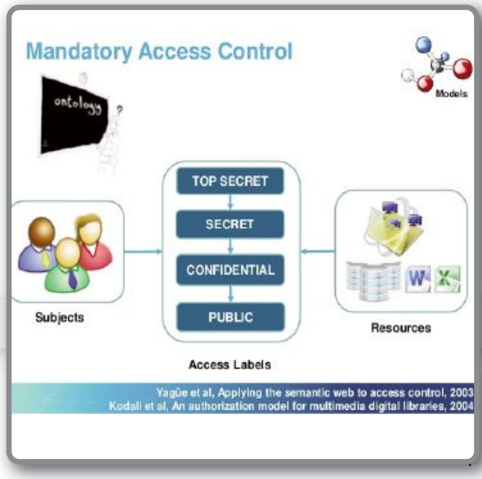
Říkají routeru, které pakety mají být povo a které zakázány.

- ACLs are lists of conditions used to test network traffic that tries to travel across router interface. These lists tell the route what types of packets to accept or deny.

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 that 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 [22].

Earlier implementations of ACLs were done with a single auxiliary file per filesystem that was indexed by the inode number and had a small fixed-sized area to store the ACL permissions. The small size was to keep the size of the auxiliary file reasonable, since it had to have space for every possible inode in the filesystem. 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 to have the update take effect [30].

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 easily stored. Space used to store extended attribute information is proportional to the number of inodes with extended attributes and the size of the ACL lists that they use. Atomic update 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. While it would be possible to update the old auxiliary file on every fsync system call done on the filesystem, the cost of doing so would be prohibitive. Here, the kernel knows if the extended attribute data block for an inode is dirty and can write just that data block during an fsync call on the inode.

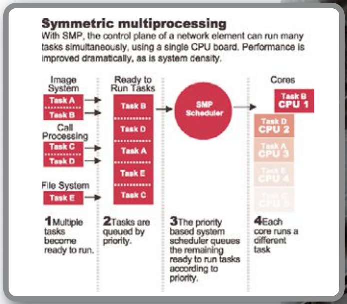

Mandatory Access Control
ontology
Models
TOP SECRET
SECRET
CONFIDENTIAL
PUBLIC
Subjects
Access Labels
Resources
Yagüe et al, Applying the semantic web to access control, 2003
Kodali et al, An authorization model for multimedia digital libraries, 2004

2005: Mandatory Access Controls

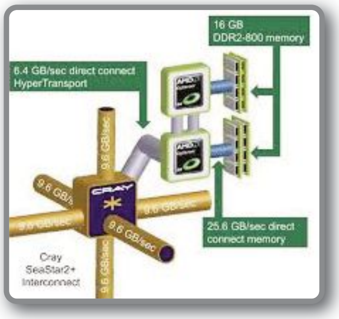
The second use for extended attributes was for data labeling. Data labels provide permissions for a mandatory access control (MAC) framework enforced by the kernel. The kernel's MAC framework permits dynamically introduced system-security modules to modify system security functionality. This framework 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 filesystem 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 [31, 32].

2006: Symmetric Multi-processing

In the late 1990s, the Free BSD 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 filesystem was overhauled to support symmetric multi-processing completing the giant-free path from system-call to hardware.


Symmetric multiprocessing
With SMP, the control plane of a network element can run many tasks simultaneously, using a single CPU board. Performance is improved dramatically, as is system density.
Image System
Task A
Task B
Call Processing
Task C
Task D
File System
Task E
Ready to Run Tasks
Task B
Task D
Task A
Task E
Task C
SMP Scheduler
Cores
Task B CPU 1
Task D CPU 2
Task A CPU 3
Task E CPU 4
1 Multiple tasks become ready to run.
2 Tasks are queued by priority.
3 The priority based system scheduler queues the remaining ready to run tasks according to priority.
4 Each core runs a different task

2009: Journaled Soft Updates

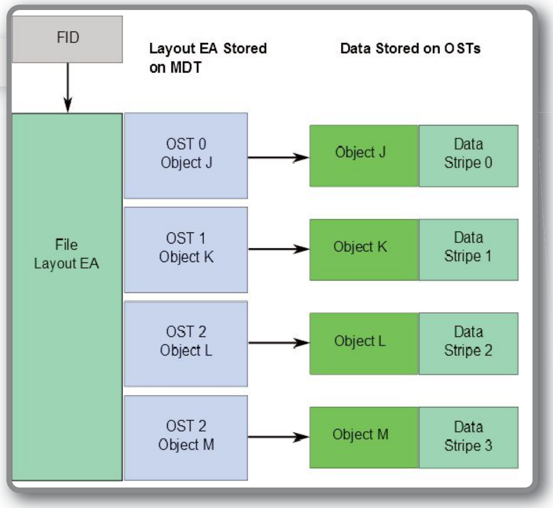


Though soft updates avoided the need to run fsck after a crash, soft updates could still cause blocks and inodes to become lost; they would not be in use by the filesystem but were still claimed as in use in the filesystem's bitmaps. fsck still had to be run periodically to reclaim the lost space. Thus, the idea arose to supplement soft updates with a journal that tracks the freeing of resources so that after a crash the journal can be replayed to recover the lost resources. Specifically, the journal contains the information needed to recover the block and inode resources that have been freed but whose freed status failed to make it to disk before a system failure. After a crash, a variant of the venerable fsck program runs through the journal to identify and free the lost resources. Only if an inconsistency between the log and filesystem is detected is it necessary to run the whole-filesystem fsck. The journal is tiny: 16 Mbyte is usually enough, independent of filesystem size. Although journal processing needs to be done before restarting, the processing time is typically just a few seconds and, in the worst case, a minute. It is not necessary to build a new filesystem to use soft-updates journaling. The addition or deletion of soft-updates journaling to existing Free BSD fast filesystems is done using the tunefs program.

2011: Raising the Blocksize, Yet Again

By 2011 disk capacity had risen enough that the default blocksize was raised to 32,768-byte blocks with 4,096-byte fragments. This increase was also driven by the change of disk technology to 4K sectors. With the increase in sector size, small files once again used a minimum of one disk sector. Thus the filesystem once again doubled throughput with no additional wasted disk space.

2013: Optimized Metadata Layouts



In an effort to speed random access to files and to speed the checking of metadata by fsck, the filesystem holds the first 4% of the data blocks in each cylinder group for the use of metadata [11]. The policy routines preferentially place metadata in the metadata area and everything else in the blocks that follow the metadata area. The size of the metadata area does not need to be precisely calculated as it is used just as a hint of where to place the metadata by the policy routines. If the metadata area fills up, then the metadata can be placed in the regular-blocks area, and if the regular-blocks area fills up, then the regular blocks can be placed in the metadata area. This decision happens on a cylinder group by cylinder group basis, so some cylinder groups can overflow their metadata area while others do not

overflow it. The policy is to place all metadata in the same cylinder group as their inode. Spreading the metadata across cylinder groups generally results in reduced filesystem performance.

The one exception to the metadata placement policy is for the first indirect block of the file. The policy is to place the first (single) indirect block inline with the file data (e.g., it tries to lay out the first 12 direct blocks contiguously, followed immediately by the indirect block, followed immediately by the data blocks referenced from the indirect block). Putting the first indirect block inline with the data rather than in the metadata area is to avoid two extra seeks when reading it. These two extra seeks would noticeably slow down access to a file that uses only the first few blocks referenced from its indirect block.

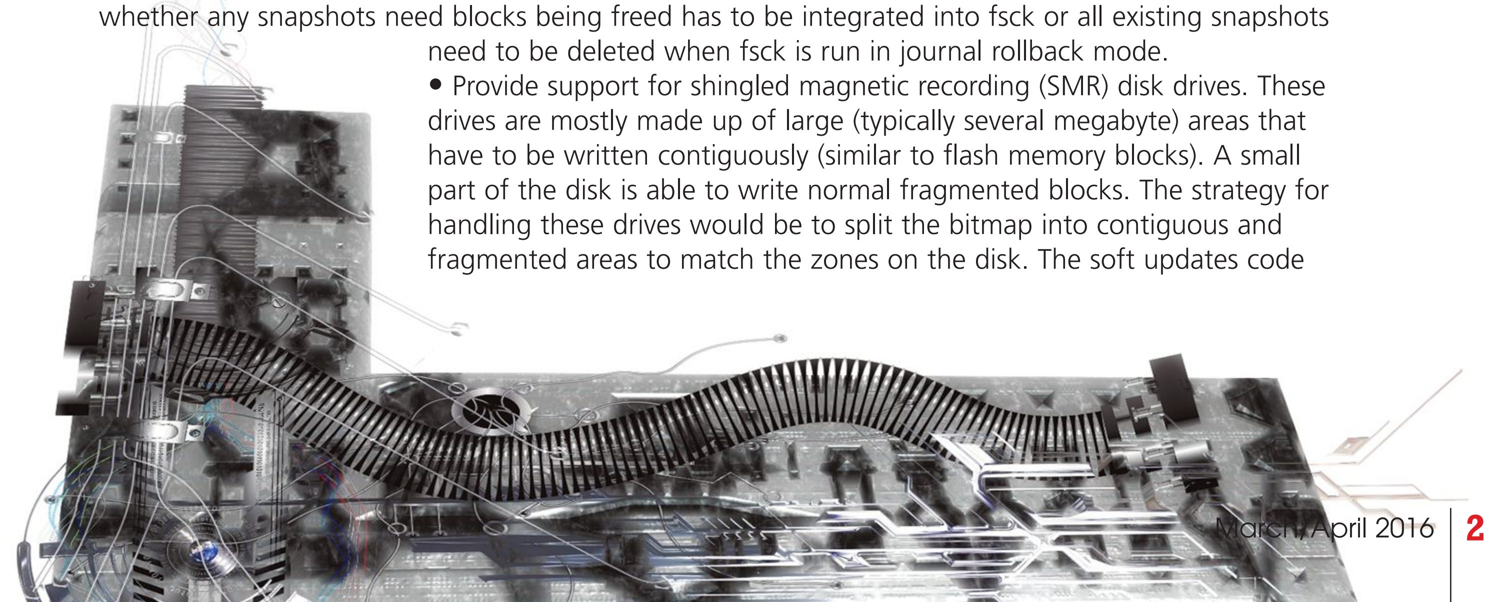
Only the second and third level indirects, along with the indirects that they reference, are allocated in the metadata area. The nearly contiguous allocation of this metadata close to the inode that references them noticeably improves the random access time to the file as well as speeding up the running time of fsck. Also, the disk track cache is often filled with much of a file's metadata when the second-level indirect block is read, thus often speeding up even the sequential reading time for the file.

In addition to putting indirect blocks in the metadata area, it is also helpful to put the blocks holding the contents of directories there, too. Putting the contents of directories in the metadata area gives a speed-up to directory tree traversal since the data is a short seek away from where the directory inode was read and may already be in the disk's track cache from other directory reads done in its cylinder group.

Future Directions

There have been many changes proposed or requested for the filesystem. The first group includes improvements that can be made without the need to change the on-disk format of the filesystem:

- When running on devices such as flash memory that need to bulk-erase blocks before they can be reused, the filesystem notifies the underlying device whenever it is finished using a block. The device is notified using a TRIM request, typically when a block is being freed during a file removal or truncation. Each block that is freed results in a TRIM command being sent through the GEOM layer to the device and a corresponding acknowledgment being returned when it is completed. Often, a file is made up of many consecutive blocks. It would be far more efficient to collect these consecutive blocks together and send a single TRIM request for the entire large block.
- Currently, a filesystem that has enabled journaled soft updates is not able to take snapshots. The restriction applies because the code in fsck that does the journal rollback does not know how to handle snapshot files when it is releasing blocks. Specifically, when a block is released, each of the snapshots needs to be checked to see if the block being released is one that the snapshot wants to claim. Only if none of the snapshots want the block can it be released to the free list. The journal rollback in fsck always releases the blocks to the free list. Thus, if the filesystem contained any snapshots that needed to claim one of the released blocks, it would be corrupted. Either the kernel code that checks whether any snapshots need blocks being freed has to be integrated into fsck or all existing snapshots need to be deleted when fsck is run in journal rollback mode.
- Provide support for shingled magnetic recording (SMR) disk drives. These drives are mostly made up of large (typically several megabyte) areas that have to be written contiguously (similar to flash memory blocks). A small part of the disk is able to write normal fragmented blocks. The strategy for handling these drives would be to split the bitmap into contiguous and fragmented areas to match the zones on the disk. The soft updates code



would then be augmented to collect the soft updates together into batches of blocks that could be written contiguously.

- The current filesystem has a provision to allow individual files to use a larger blocksize. This capability has never been implemented, but would be very beneficial for larger files. Other desired changes to the filesystem would require a new disk format, often referred to as UFS3:

The most important of these changes would be to change the directory format to increase its file number from 32 bits to 64 bits. With increasing disk capacities, the limit of 4 billion files per filesystem has become increasingly problematic. The biggest stumbling block to making this change is the FreeBSD filesystem interface currently only supports 32-bit file numbers. Changing this interface has been discussed ever since the ZFS filesystem with its native 64-bit file numbers was brought into the system. Hopefully the interface change can be realized in time for the FreeBSD 11 release.

- Another limitation of the current filesystem format is that it uses a 16-bit field to record the number of links to a file, thus limiting a file to having 65,535 directory entries referencing it. When changing the on-disk format, this field should be increased to 64 bits to resolve the problem for many years to come.When doing a revision of the on-disk format as part of UFS3, it might be useful to add some ZFS-like features:

Add checksums to improve the robustness and data integrity of the filesystem. The easiest to implement would be to place the checksum in each block of the file. This approach fails to detect blocks that are written to the wrong location. Taking the ZFS approach of storing each checksum with the block pointer avoids this problem but requires a more impactful change to the existing filesystem code.

- Another useful change to improve the robustness and data integrity is to provide redundancy of the filesystem metadata. At a minimum, the filesystem should provide redundant copies of inodes and indirect blocks. If practical, the filesystem should also provide multiple copies of the directory data blocks. And if the implementation was flexible enough, it could provide optional redundancy for the user's data blocks as is available in ZFS.Another technology that is beginning to appear in the marketplace is key/value disks such as those recently released by Seagate. These disks provide objects up to one megabyte in size that are identified using a 64-bit key. The filesystem could be adapted to use these disks by using a one megabyte blocksize, which would dramatically reduce the amount of metadata information that it would need to maintain. The block numbers would be replaced with the 64-bit object keys, and the file content would be stored as the object value. The final fragment of the file could be stored in a smaller object, thus allowing the disk to manage disk fragmentation issues. ●

MARSHALL KIRK MCKUSICK writes books and articles, consults, and teaches classes on Unix- and BSD-related subjects. 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. He has twice been president of the board of the Usenix Association, is currently a member of the FreeBSD Foundation Board of Directors, a member of the editorial board of *ACM Queue* magazine and *FreeBSD Journal*, a senior member of the IEEE, and a member of the Usenix Association, ACM, and AAAS. You can contact him via email at mckusick@mckusick.com.

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

REFERENCES

[1] Apple, *Mac OS X Essentials, Chapter 9 Filesystem, Section 12 Resource Forks*, available from https://en.wikipedia.org/wiki/Resource_fork.

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

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

CONTINUES NEXT PAGE

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

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

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

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

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

[9] D. Korn & E. Krell. “The 3-D File System,” *USENIX Association Conference Proceedings*, pp. 147–156. (June 1989)

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

[11] A. Ma, C. Dragga, A. Arpaci-Dusseau, & R. Arpaci-Dusseau. “ffsck: The Fast File System Checker,” *USENIX FAST ’13 Conference*, available from www.usenix.org/conference/fast13/ffsck-fast-file-system-checker. (February 2013)

[12] M. K. McKusick. “Running fsck in the Background,” *Proceedings of the BSDC on 2002 Conference*, pp. 55–64. (February 2002)

[13] M. K. McKusick. “Enhancements to the Fast Filesystem to Support Multi-terabyte Storage Systems,” *Proceedings of the BSDC on 2003 Conference*, pp. 79–90. (September 2003)

[14] M. K. McKusick, W. N. Joy, S. J. Leffler, & R. S. Fabry. “A Fast File System for UNIX,” *ACM Transactions on Computer Systems* 2(3), pp. 181–197, Association for Computing Machinery. (August 1984)

[15] M. K. McKusick & T. J. Kowalski. “fsck: The UNIX File System Check Program” in *4.4 BSD System Manager’s Manual*, pp. 3:1–21, O’Reilly & Associates, Inc., Sebastopol, California. (1994)

[16] L. McVoy & S. Kleiman. “Extent-Like Performance from a UNIX File System,” *USENIX Association Conference Proceedings*, pp. 33–44. (January 1991)

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

[18] J. Ousterhout. “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?,” *Summer USENIX Conference*, pp. 247–256. (June 1990)

[19] J. Pendry & M. K. McKusick. “Union Mounts in 4.4 BSD -Lite,” *USENIX Association Conference Proceedings*, pp. 25–33. (January 1995)

[20] J. Pendry & N. Williams. “AMD: The 4.4 BSD Automounter Reference Manual” in *4.4 BSD System Manager’s Manual*, pp. 13:1–57, O’Reilly & Associates, Inc., Sebastopol, California. (1994)

[21] H. Reiser. *The Reiser File System*, available from https://en.wikipedia.org/wiki/ReiserFS.

[22] T. Rhodes. *Free BSD Handbook, Chapter 3, Section 3.3 File System Access Control Lists*, available from http://www.FreeBSD.org/doc/en_US.ISO8859-1/books /handbook/fs-acl.html. (March 2014)

[23] M. Rosenblum & J. Ousterhout.“The Design and Implementation of a Log-Structured File System,” *ACM Transactions on Computer Systems* 10(1), pp. 26–52, Association for Computing Machinery. (February 1992)

[24] D. Rosenthal. “Evolving the Vnode Interface,” *USENIX Association Conference Proceedings*, pp. 107–118. (June 1990)

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

[26] M. Seltzer, K. Bostic, M. K. McKusick, & C. Staelin. “An Implementation of a Log-Structured File System for UNIX,” *USENIX Association Conference Proceedings*, pp. 307–326. (January 1993)

[27] M. Seltzer & K. Smith. “A Comparison of FFS Disk Allocation Algorithms,” *Winter USENIX Conference*, pp. 15–25. (January 1996)

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

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, & G. Peck. “Scalability in the XFS File System,” *USENIX Association Conference Proceedings*, pp. 1–14. (January 1996)

[30] R. Watson. “Introducing Supporting Infrastructure for Trusted Operating System Support in Free BSD,” *Proceedings of the BSDC on 2000 Conference*. (September 2000)

[31] R. Watson. “Trusted BSD: Adding Trusted Operating-System Features to Free BSD,” *Proceedings of the Freenix Track at the 2001 USENIX Annual Technical Conference*, pp. 15–28. (June 2001)

[32] R. Watson, W. Morrison, C. Vance, & B. Feldman. “The Trusted BSD MAC Framework: Extensible Kernel Access Control for Free BSD 5.0,” *Proceedings of the Freenix Track at the 2003 USENIX Annual Technical Conference*, pp. 285–296. (June 2003)

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