USENIX Association

Proceedings of BSDCon ’03

San Mateo, CA, USA
September 8–12, 2003

Enhancements to the Fast Filesystem To Support Multi-Terabyte Storage Systems

Marshall Kirk McKusick
Author and Consultant

Abstract

This paper describes a new version of the fast filesystem, UFS2, designed to run on multi-terabyte storage systems. It gives the motivation behind coming up with a new on-disk format rather than trying to continue enhancing the existing fast-filesystem format. It describes the new features and capabilities in UFS2 including extended attributes, new and higher resolution time stamps, dynamically allocated inodes, and an expanded boot block area. It also describes the features and capabilities that were considered but rejected giving the reasons for their rejection. Next it covers changes that were made to the soft update code to support the new capabilities and to enable it to work more smoothly with existing filesystems. The paper covers enhancements made to support live dumps and changes made to filesystem snapshots needed to avoid deadlocks and to enable them to work efficiently with multi-terabyte filesystems. Similarly, it describes changes that needed to be made to the filesystem check program to work with large filesystems. The paper gives some comments about performance, and decribes areas for future work including an extent-based allocation mechanism and indexed directory structures. The paper concludes with current status and availability of UFS2.

1. Background and Introduction

Traditionally, the BSD fast filesystem (which we shall refer to in this paper as UFS1) [McKusick et al, 1996; McKusick, Joy et al, 1984] and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. The UFS1 filesystem was designed in the early 1980's when the largest disks were 330 megabytes. 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 twenty years that it has been deployed, storage systems have grown to hold over a terabyte of data. Depending on the block size configuration, the 32-bit block pointers of UFS1 run out of space in the 1 to 4 terabyte range. While some stop-gap measures can be used to extend the maximum size storage systems supported by UFS1, 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 filesystem, UFS2, that would use 64-bit block pointers.

We considered the alternatives between trying to make incremental changes to the existing UFS1 filesystem versus importing another existing filesystem such as XFS [Sweeney et al, 1996], or ReiserFS [Reiser, 2001]. 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 UFS1 filesystem as this approach allowed us to reuse most of the existing UFS1 code base. The benefits of this decision were that UFS2 was developed and deployed quickly, it became stable and reliable rapidly, and the same code base could be used to support both UFS1 and UFS2 filesystem formats. Over 90 percent of the code base is shared, thus bug fixes and feature or performance enhancements usually apply to both filesystem formats.

Sections 2, 3, and 4 discuss the UFS2 filesystem itself. Sections 5 and 6 discuss enhancements that were made during the development of UFS2 but which transfer over to UFS1 as well. Sections 7 and 8 describe how we overcame problems of scale brought on by the enormous size of filesystems supported by UFS2. The last three sections conclude with discussions of performance, future work, and current status.

2. The UFS2 Filesystem

The on-disk inodes used by UFS1 are 128-bytes in size and have only two unused 32-bit fields. It would not be possible to convert to 64-bit block pointers without reducing the number of direct block pointers from twelve to five. Doing so would dramatically increase the amount of wasted space as only direct block pointers can reference fragments. So, the only viable alternative is to increase the size of the on-disk inode to 256 bytes.

Once one is committed to changing to a new on-disk format for the inodes, it is possible to include other inode-related changes that were not possible within the constraints of the old inodes. While it is tempting to throw in everything that has ever been suggested over the last twenty years, we feel that it is best to limit the addition of new capabilities to those that are likely to have a clear benefit. Every new addition adds complexity which has a cost both in maintainability and performance. Obscure or little used features may add conditional checks in frequently executed code paths such as read and write slowing down the overall performance of the filesystem even if they are not used.

Although we decided to come up with a new on-disk inode format, we chose not to change the format of the superblock, the cylinder group maps, or the directories. Additional information needed for the UFS2 superblock and cylinder groups is stored in spare fields of the UFS1 superblock and cylinder groups. Maintaining the same format for these data structures allows a single code base to be used for both UFS1 and UFS2. Because the only difference between the two filesystems is in the format of their inodes, code can dereference pointers to superblocks, cylinder groups, and directory entries without need of checking what type of filesystem is being accessed. To minimize conditional checking of code that references inodes, the on-disk inode is converted to a common in-core format when the inode is first read in from the disk, and converted back to its on-disk format when it is written back. The effect of this decision is that there are only nine out of several hundred routines that are specific to UFS1 versus UFS2. The benefit of having a single code base for both filesystems is that it dramatically reduces the maintenance cost. Outside of the nine filesystem format specific functions, fixing a bug in the code fixes it for both filesystem types. A common code base also means that as the symmetric multiprocessing support gets added, it only needs to be done once for the UFS family of filesystems.

Although we still use the same data structure to describe cylinder groups, the practical definition of them has changed. In the era of UFS1, the filesystem 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. Modern disks hide 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 [Griffin et al, 2002; Lumb et al, 2002; Schindler et al, 2002], the complexity of using such information effectively is quite high. Modern disks have greater numbers of sectors per track on the outer part of the disk than the inner part which makes calculation of the rotational position of any given sector quite complex to calculate. So, for UFS2, we decided to get rid of all the rotational layout code found in UFS1 and simply assume that laying out files with numerically close block numbers (sequential being viewed as optimal) would give the best performance. Thus, the cylinder group structure is retained in UFS2, but it is used only as a convenient way to manage logically close groups of blocks. The rotational layout code had been disabled in UFS1 since the late 1980s, so as part of the code base cleanup it was removed entirely.

The UFS1 filesystem uses 32-bit inode numbers. While it is very tempting to increase these inode numbers to 64 bits in UFS2, doing so would require that the directory format be changed. There is a lot of code that works directly on directory entries. Changing directory formats would entail creating many more filesystem specific functions which would increase the complexity and maintainability issues with the code. Furthermore, the current APIs for referencing directory entries use 32-bit inode numbers. So, even if the underlying filesystem supported 64-bit inode numbers, they could not currently be made visible to user applications. In the short term, applications are not running into the four billion files-per-filesystem limit that 32-bit inode numbers impose. If we assume that the growth rate in the number of files per filesystem over the last twenty years will continue at the same rate, we estimate that the 32-bit inode number should be sufficient for another ten to twenty years. However, the limit will be reached before the 64-bit block limit of UFS2 is reached. So, the UFS2 filesystem has reserved a flag in the superblock to indicate that it is a filesystem with 64-bit inode numbers. When the time comes to begin using 64-bit inode numbers, the flag can be turned on and the new directory format can be used. Kernels that predate the introduction of 64-bit inode numbers check this flag and will know that they cannot mount such filesystems.

Another change that was contemplated was changing to a more complex directory structure such as one that uses B-trees to speed up access for large directories. This technique is used in many modern filesystems such as XFS [Sweeney et al, 1996], JFS [Best & Kleikamp, 2003], ReiserFS [Reiser, 2001], and in later versions of Ext2 [Phillips, 2001]. We decided not to make the change at this time for several reasons. First, we had limited time and resources and we wanted to get something working and stable that could be used in the time frame of FreeBSD 5.0. By keeping the same directory format, we were able to reuse all the directory code from UFS1, did not have to change numerous filesystem utilities to understand and maintain a new directory format, and were able to produce a stable and reliable filesystem in the time frame available to us. The other reason that we felt that we could retain the existing directory structure is because of the dynamic directory hashing that was added to FreeBSD [Dowse & Malone, 2002]. The dynamic directory hashing retrofits a directory indexing system to UFS. To avoid repeated linear searches of large directories, the dynamic directory hashing builds a hash table of directory entries on the fly when the directory is first accessed. This table avoids directory scans on subsequent lookups, creates, and deletes. Unlike filesystems originally designed with large directories in mind, these indices are not saved on disk and so the system is backwards compatible. The effect of the dynamic directory hashing is that large directories in UFS cause minimal performance problems.

Borrowing the technique used by the Ext2 filesystem a flag was also added to indicate that an on-disk indexing structure is supported for directories [Phillips, 2001]. This flag is unconditionally turned off by the existing implementation of UFS. In the future, if an implementation of an on-disk directory-indexing structure is added, the implementations that support it will not turn the flag off. Index-supporting kernels will maintain the indices and leave the flag on. If an old non-index-supporting kernel is run, it will turn off the flag so that when the filesystem is once again run under a new kernel, the new kernel will discover that the indexing flag has been turned off and will know that the indices may be out date and have to be rebuilt before being used. The only constraint on an implementation of the indices is that they have to be an auxiliary data structure that references the old linear directory format.

3. Extended Attributes

A major addition in UFS2 is support for 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 [Apple, 2003]. 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.

The current implementation has space in the inode to store up to two blocks of extended attributes. The new UFS2 inode format had room for up to five additional 64-bit pointers. Thus, the number of extended attribute blocks could have been in the range of one to five blocks. We chose to allocate two blocks to the extended attributes and to leave the other three as spares for future use. By having two, all the code had to be prepared to deal with an array of pointers, thus if the number got expanded into the remaining spares in the future the existing implementation will work without change. By saving three spares, we provided a reasonable amount of space for future needs. And, if the decision to allow only two blocks proves to be too little space, one or more of the spares can be used to expand the size of the extended attributes in the future. If vastly more extended attribute space is needed, one of the spares could be used as an indirect pointer to extended attribute data blocks.

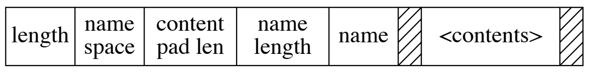

Figure 1: *Format of Extended Attributes*

Figure 1 shows the format used for the extended attributes. The header of each attribute has a 4-byte length, 1-byte name space class, 1-byte content pad length, 1-byte name length, and name. The name is padded so that the contents start on an 8-byte boundary. The contents are padded to the size shown by the “content pad length” field. Applications that do not understand the name space or name can simply skip over the unknown attribute by adding the length to their current position to get to the next attribute. Thus, many different applications can share the usage of the extended attribute space, even if they do not understand each other's data types.

The first of two initial uses for extended attributes is to support access control lists, generally referred to as ACLs. An ACL replaces the group permissions for a file with a more specific list of the users that are permitted to access the files along with a list of the permissions that they are 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 [Rhodes, 2003].

Earlier implementations of ACLs were done with a single auxiliary file per filesystem that was indexed by the inode number and had a small and fixed sized area to store the ACL permissions. The size was small 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 requires that both the file inode and the ACL file be written to have the update take effect [Watson, 2000].

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 kilobytes and typically 32 kilobytes), 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.

The second use for extended attributes is for data labeling. Data labels are used to provide permissions for mandatory access controls (MACs). The kernel provides a MAC framework that 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 which is called by code supporting various kernel services, especially with respects 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 modules needs to query them to make a permission check [Watson, 2001; Watson et al, 2003].

We considered storing symbolic links in the extended attribute area. We chose not to do this for three reasons. First, the time to access an extended storage block is the same as the time to access a regular data block. Second, since symbolic links rarely have any extended attributes, there would be no savings in storage since a filesystem fragment would be needed whether it was stored in a regular data block or in an extended storage block. Third, if it were stored in the extended storage area, it would take more time to traverse down the attribute list to find it.

4. New Filesystem Capabilities

Several other improvements were made when the enlarged inode format was created. We decided to get an early jump on the year 2038 problem (specifically, Tue Jan 19 03:14:08 2038 GMT which could be a really ugly way to usher in my 84th birthday). We expanded the time fields (which hold seconds-since-1970) for access, modification, and inode-modification times from 32-bits to 64-bits. At plus or minus 136 billion years that should carry us from well before the universe was created until long after our Sun has burned itself out. We left the nanoseconds fields for these times at 32-bits as we did not feel that added resolution was going to be useful in the foreseeable future. We considered expanding the time to only 48-bits. We chose to go to 64-bits as 64-bits is a native size that can be easily manipulated with existing and likely future architectures. Using 48-bits would have required an extra unpacking or packing step each time the field was read or written. Also, going to 64-bits ensures enough bits for all likely measured time so will not have to be enlarged.

At the same time we also added a new time field (also 64-bit) to hold the birth time (also commonly called the creation time) of the file. The birth time is set when the inode is first allocated and is not changed thereafter. It has been added to the structure returned by the ‘‘stat’’ system call so that applications can determine its value and so that archiving programs such as **dump**, **tar**, and **pax** can save this value along with the other file times. The birth time was added to a previously spare field in the ‘‘stat’’ system call structure so that the size of the structure did not change. Thus, old versions of programs that use the ‘‘stat’’ call continue to work.

To date, only the **dump** program has been changed to save the birth time value. This new version of **dump** which can dump both UFS1 and UFS2 filesystems, creates a new dump format which is not readable by older versions of **restore**. The updated version of **restore** can identify and restore from both old and new dump formats. The birth times are only available and setable from the new dump format.

The ‘‘utimes’’ system call sets the access and modification times of a file to a specified set of values. It is used primarily by archive retrieval programs to set newly extracted files times back to those associated with the file in the archive. With the addition of birth time, we added a new system call that allows the setting of access, modification, and birth times. However, we realized that many existing applications will not be changed to use the new ‘‘utimes’’ system call. The result will be that the files that they retrieved from archives will have a newer birth time than access or modification times.

To provide a sensible birth time for applications that are unaware of the birth time attribute, we changed the semantics of the ‘‘utimes’’ system call so that if the birth time was newer than the value of the modification time that it was setting, it sets the birth time to the same time as the modification time. An application that is aware of the birth time attribute can set both the birth time and the modification time by doing two calls to ‘‘utimes’’. First it calls ‘‘utimes’’ with a modification time equal to the saved birth time, then it calls ‘‘utimes’’ a second time with a modification time equal to the (presumably newer) saved modification time. For filesystems that do not store birth times, the second call will overwrite the first call resulting in the same values for access and modification times as they would have previously gotten. For filesystems that support birth time, it will be properly set. And most happily for the application writers, they will not have to conditionally compile the name of ‘‘utimes’’ for BSD and non-BSD systems. They just write their applications to call the standard interface twice knowing that the right thing will happen on all systems and filesystems. For those applications that value speed of execution over portability can use the new version of the ‘‘utimes’’ system call that allows all time values to be set with one call.

Another incremental change to the inode format was to split the flags field into two separate 32-bit fields, one for flags that can be set by applications (as in UFS1) and a new field for flags maintained strictly by the kernel. An example of a kernel flag is the SNAPSHOT flag used to label a file as being a snapshot. Another kernel-only flag is OPAQUE which is used by the union filesystem to mark a directory which should not make the layers below it visible. By moving these kernel flags into a separate field, they will not be accidentally set or cleared by a naive or malicious application.

4.1. Dynamic Inodes

One of the common complaints about the UFS1 filesystem is that it preallocates all its inodes at the time that the filesystem is created. For filesystems with millions of files, the initialization of the filesystem can take several hours. Additionally, the filesystem creation program, **newfs**, had to assume that every filesystem would be filled with many small files and allocate a lot more inodes than were likely to ever be used. If a UFS1 filesystem uses up all its inodes, the only way to get more is to dump, rebuild, and restore the filesystem. The UFS2 filesystem resolves these problems by dynamically allocating its inodes. The usual implementation of dynamically allocated inodes requires a separate filesystem data structure (typically referred to as the inode file) that tracks the current set of inodes. The management and maintenance of this extra data structure adds overhead and complexity and often degrades performance.

To avoid these costs, UFS2 preallocates a range of inode numbers and a set of blocks for each cylinder group. Initially each cylinder group has a single block of inodes allocated (a typical block holds 32 or 64 inodes). When the block fills up, the next block of inodes in the set is allocated and initialized. The set of blocks that may be allocated to inodes is held as part of the free-space reserve until all other space in the filesystem is allocated. Only then can it be used for file data.

In theory a filesystem could fill using up all the blocks set aside for inodes. Later after large files had been removed and many small files created to replace them, the filesystem might find itself unable to allocated the needed inodes because all the space set aside for inodes was still in use. Here, it would be necessary to reallocate existing files to move them to new locations outside of the inode area. Such code has not been written as we do not anticipate that this condition will arise in practice as the free space reserve used on most filesystems (8%) exceeds the amount of space needed for inodes (typically 2-6%). On these systems only a process running with root privileges would ever be able to allocate the inode blocks. Should the code prove necessary in actual use, it can be written at that time. Until it is written, filesystems hitting this condition will return an ‘‘out of inodes’’ error on attempts to create new files.

One of the side benefits of dynamically allocating inodes is that the time to create a new filesystem in UFS2 is about 1 percent of the time that it takes in UFS1. A filesystem that would take one hour to build in a UFS1 format can be built in under a minute in the UFS2 format. While filesystem creations are not a common operation, having them build quickly does matter to the system administrators that have to do such tasks with some regularity.

The cost of dynamically allocating inodes is one extra disk write for every 64 new inodes that are created. Although this cost is quite low compared to the other costs of creating 64 new files, some systems administrators might want to preallocate more than the minimal number of inodes. If such a demand arises, it would be trivial to add a flag to the **newfs** program to preallocate additional inodes at the time that the filesystem is created.

4.2. Boot Blocks

The UFS1 filesystem reserved an 8 kilobyte space at the beginning of the filesystem in which to put a boot block. While this space seemed huge compared to the 1 kilobyte book block that it replaced, over time it has gotten increasingly difficult to cram the needed boot code into this space. Consequently we decided to revisit the boot block size in UFS2.

The boot code has a list of locations to check for boot blocks. A boot block can be defined to start at any 8 kilobyte boundary. We set up an initial list with four possible boot block sizes: none, 8 kilobytes, 64 kilobytes, and 256 kilobytes. Each of these locations was selected for a particular purpose. Filesystems other than the root filesystem do not need to be bootable, so can use a boot block size of zero. Also, filesystems on tiny media that need every block that they can get such as floppy disks can use a zero size boot block. For architectures with simple boot blocks, the traditional UFS1 8 kilobyte boot block can be used. More typically the 64 kilobyte boot block is used (for example on the PC architecture with its need to support booting from a myriad of busses and disk drivers).

We added the 256 kilobyte boot block in case some architecture or application needs to set aside a particularly large boot area. While this was not strictly necessary as new sizes can be added to the list at any time, it can take a long time before the updated list gets propagated to all the boot programs and loaders out on the existing systems. By adding the option for a huge boot area now, we can ensure it will be readily available should it be needed on short notice in the future.

One of the unexpected side effects of using a 64 kilobyte boot block for UFS2 is that if the partition had previously had a UFS1 filesystem on it, the superblock for the former UFS1 filesystem may not be overwritten. If an old version of **fsck** that does not first look for a UFS2 filesystem is run and finds the UFS1 superblock, it can incorrectly try to rebuild the UFS1 filesystem destroying the UFS2 filesystem in the process. So, when building UFS2 filesystems, the **newfs** utility looks for old UFS1 superblocks and zeros them out.

5. Changes and Enhancements to Soft Updates

Traditionally, filesystem consistency has been maintained across system failures either by using synchronous writes to sequence dependent metadata updates or by using write-ahead logging to atomically group them [Seltzer et al, 2000]. Soft updates, an alternative to these approaches, is an implementation mechanism that tracks and enforces metadata update dependencies to ensure that the disk image is always kept consistent. The use of soft updates obviates the need for a separate log or for most synchronous writes [McKusick & Ganger, 1999].

The addition of extended attribute data to the inode required that the soft updates code be extended so that it could ensure the integrity of these new data blocks. As with the file data blocks, it ensures that the extended data blocks and the bitmaps that show that they are in use are written to disk before they are claimed by the inode. Soft updates also ensure that any updated extended attribute data is committed to disk as part of an ‘‘fsync’’ of the file.

Two important enhancements were made to the existing soft updates implementation. These enhancements were initially made for UFS2 but because of the shared code base with UFS1 were trivially integrated to work with UFS1 filesystems as well.

When a file is removed on a filesystem running with soft updates, the removal appears to happen very quickly, but the process of removing the file and returning its blocks to the free list may take up to several minutes. Prior to UFS2, the space held by the file did not show up in the filesystem statistics until the removal of the file had been completed. Thus, applications that clean up disk space such as the news expiration program would often vastly overshoot their goal. They work by removing files and then checking to see if enough free space has showed up. Because of the time lag in having the free space recorded, they would remove far too many files. To resolve problems of this sort, the soft updates code now maintains a counter that keeps track of the amount of space that is held by the files that it is in the process of removing. This counter of pending space is added to the actual amount of free space as reported by the kernel (and thus by utilities like **df**). The result of this change is that free space appears immediately after the ‘‘unlink’’ system call returns or the **rm** utility finishes.

The second and related change to soft updates has to do with avoiding false out-of-space errors. When running with soft updates on a nearly full filesystem with high turnover rate (for example when installing a whole new set of binaries on a root partition), the filesystem can return a filesystem full error even though it reports that it has plenty of free space. The filesystem full message happens because soft updates has not managed to free the space from the old binaries in time for it to be available for the new binaries.

The initial attempt to correct this problem was to simply have the process that wished to allocate space wait for the free space to show up. The problem with this approach is that it often had to wait for up to a minute. In addition to making the application seem intolerably slow, it usually held a locked vnode which could cause other applications to get blocked waiting for it to become available (often referred to as a lock race to the root of the filesystem). Although the condition would clear in a minute or two, users often assumed that their system had hung and would reboot.

To remedy this problem, the solution devised for UFS2 is to co-opt the process that would otherwise be blocked and put it to work helping soft updates process the files to be freed. The more processes trying to allocate space, the more help that is available to soft updates and the faster free blocks begin to appear. Usually in under one second enough space shows up that the processes can return to their original task and proceed to completion. The effect of this change is that soft updates can now be used on small nearly full filesystems with high turnover.

6. Enhancements for Live Dumps

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 back-ups of the filesystem at several times during the day, the ability to do reliable dumps of live filesystems, and the ability to run a filesystem check program on a active system to reclaim lost blocks and inodes [McKusick, 2002].

With the advent of filesystem snapshots, the **dump** program has been enhanced to safely dump live filesystems. When given the -L flag, **dump** verifies that it is being asked to dump a mounted filesystem, then takes a snapshot of the filesystem and dumps the snapshot instead of on the live filesystem. When **dump** completes, it releases the snapshot.

The initial implementation of live dumps had the **dump** program do the ‘‘mount’’ system call itself to take the snapshot. However, most systems require root privilege to use the ‘‘mount’’ system call. Since dumps are often done by the *operator* user rather than *root*, an attempt to take a snapshot will fail.

To get around this problem, a new set-user-identifier *root* program was written called **mksnap_ffs**. The **mksnap_ffs** command creates a snapshot with a given name on a specified filesystem. The snapshot file must be contained within the filesystem being snapshotted. The group ownership of the file is set to *operator*; the owner of the file remains *root*. The mode of the snapshot is set to be readable by the owner or members of the *operator* group.

The **dump** program now invokes **mksnap_ffs** to create the snapshot rather than trying to create it directly. The result is that anyone with *operator* privileges can now reliably take live dumps. Allowing *operator* group access to the snapshot does not open any new security holes since the raw disk is also readable by members of the *operator* group (for the benefit of traditional **dump**). Thus, the information that is available in the snapshot can also be accessed directly through the disk device.

7. Large Filesystem Snapshots

Creating and using a snapshot requires random access to the snapshot file. The creation of a snapshot requires the inspection and copying of all the cylinder group maps. Once in operation, every write operation to the filesystem must check whether the block being written needs to be copied. The information on whether a blocks needs to be copied is contained in the snapshot file metadata (its indirect blocks).

Ideally, this metadata would be resident in the kernel memory throughout the lifetime of the snapshot. In FreeBSD, the entire physical memory on the machine can be used to cache file data pages if the memory is not needed for other purposes. Unfortunately, data pages associated with disks can only be cached in pages mapped into the kernel physical memory. Only about 10 megabytes of kernel memory is dedicated to such purposes. Assuming that we allow up to half of this space to be used for any single snapshot, the largest snapshot whose metadata that we can hold in memory is 11 megabytes. Without help, such a tiny cache would be hopeless in trying to support a multi-terabyte snapshot.

In an effort to support multi-terabyte snapshots with the tiny metadata cache available, it is necessary to observe the access patterns on typical filesystems. The snapshot is only consulted for files that are being written. The filesystem is organized around cylinder groups which maps small contiguous areas of the disk. Within a directory, the filesystem tries to allocate all the inodes and files in the same cylinder group. When moving between directories different cylinder groups are usually inspected. Thus, the widely random behavior occurs from movement between cylinder groups. Once file writing activity settles down into a cylinder group, only a small amount of snapshot metadata needs to be consulted. That metadata will easily fit in even the tiny kernel metadata cache. So, the need is to find a way to avoid thrashing the cache when moving between cylinder groups.

The technique used to avoid thrashing when moving between cylinder groups is to build a look aside table of all the blocks that were copied during the time that the snapshot was made. This table lists the blocks associated with all the snapshot metadata blocks, the cylinder groups maps, the super block, and blocks that contain active inodes. When a copy-on-write fault occurs for a block, the first step is to consult this table. If the block is found in the table, then no further searching needs to be done in any of the snapshots. If the block is not found, then the metadata of each active snapshot on the filesystem must be consulted to see if a copy is needed. This table lookup saves time as it not only avoids faulting in metadata for widely scattered blocks, but it also avoids the need to consult potentially many snapshots.

Another problem with snapshots on large filesystems is that they aggravated existing deadlock problems. When there are multiple snapshots associated with a filesystem, they are kept in a list ordered from oldest to youngest. When a copy-on-write fault

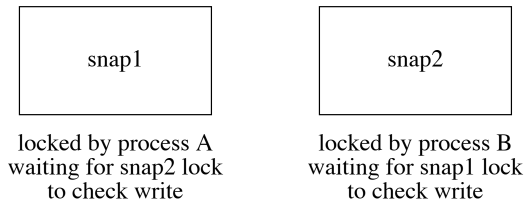
snap1
snap2
locked by process A waiting for snap2 lock to check write
locked by process B waiting for snap1 lock to check write

Figure 2: *Snapshot deadlock scenario*

occurs, the list is traversed letting each snapshot decide if it needs to make a copy of the block that is about to be written. Originally, each snapshot inode had its own lock. A deadlock could occur between two processes each trying to do a write. Consider the example in Fig. 2. It shows a filesystem with two snapshots, snap1 and snap2. Process A holds snapshot 1 locked and process B holds snapshot 2 locked. Both snap1 and snap2 have decided that they need to allocate a new block in which to hold a copy of the block being written by the process that holds them locked. The writing of the new block in snapshot 1 will cause the kernel running in the context of process A to scan the list of snapshots which will get blocked at snapshot 2 because it is held locked by process B. Meanwhile, the writing of the new block in snapshot 2 will cause the kernel running in the context of process B to scan the list of snapshots which will get blocked at snapshot 1 because it is held locked by process A.

The resolution to the deadlock problem is to allocate a single lock that is used for all the snapshots on a filesystem. When a new snapshot is created, the kernel checks whether there are any other snapshots on the filesystem. If there are, the per-file lock associated with the new snapshot inode is released and replaced with the lock used for the other snapshots. With only a single lock, the access to the snapshots as a whole are serialized. Thus, in Fig. 2, process B will hold the lock for all the snapshots and will be able to make the necessary checks and updates while process A will be held waiting. Once process B completes its scan, process A will be able to get access to all the snapshots and will be able to run successfully to completion. Because of the added serialization of the snapshot lookups, the look-aside table described earlier is important to ensure reasonable performance of snapshots. In gathering statistics on our running systems, we found that the look-aside table resolves nearly half of the snapshot copy-on-write lookups. Thus, we found that the look-aside table keeps the contention for the snapshot lock to a reasonable level.

8. Running Fsck on Large Filesystems

Traditionally, after an unclean system shutdown, the filesystem check program, **fsck**, has had to be run over all inodes in a filesystem to ascertain which inodes and blocks are in use and to correct the bitmaps. The current implementation of soft updates guarantees 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 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. A special case of the overly high link count is one that should be zero. Such an inode will be freed as part of reducing its link count to zero. This garbage collection task is less difficult than it might at first appear, since this version of **fsck** only needs to identify resources that are not in use and cannot be allocated or accessed by the running system [McKusick & Ganger, 1999].

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 [McKusick, 2002].

As filesystems have gotten bigger the time to run either a foreground or a background **fsck** has increased to multiple hours. Being able to run **fsck** in background has largely mitigated the running time issue because it allows normal system operation to proceed in parallel.

Another problem with running **fsck** on large filesystems is that the memory that it consumes grows in proportional to the size of the filesystem being checked. The main consumption of memory is four bytes per regular inode, 40 to 50 bytes per directory inode, and one bit per filesystem data block. On a typical UFS2 filesystem with 16 kilobyte blocks and 2 kilobyte fragments, the data-block map requires 64 megabytes of memory per terabyte of filesystem. Because UFS2 does not preallocate inodes, but rather allocates the inodes as they are needed, the memory required is dependent on the number of files that are created in the filesystem.

Filesystem	Files per Tb	Dirs per Tb	**fsck** memory per Tb	maximum checkable filesystem
/usr	93M	15M	1200K	3Tb
/jukebox	243K	18K	66K	60Tb

Table 1: *Maximum filesystem sizes checkable by* **fsck** *on a 32-bit architecture*

The number of files and directories in a filesystem make a huge difference in the amount of memory required by **fsck**. Table 1 shows the two ends of the spectrum. At one end is a typical FreeBSD **/usr** filesystem assuming that it grew at its current file and directory mix to fill a 1 terabyte disk. The memory footprint of **fsck** is dominated by the memory to manage the inodes and would require the entire address space of a 32-bit processor for a filesystem of about 3 terabytes. At the other extreme is the author's **/jukebox** filesystem assuming that it grew at its current file and directory mix to fill a 1 terabyte disk. The memory footprint of **fsck** is dominated by the memory to manage the data blocks and would require the entire address space of a 32-bit processor for a filesystem of about 60 terabytes. My expectation is that as disks get larger, they will tend to be filled with larger files of audio and video. Thus, in practice **fsck** will run out of space on 32-bit architectures at about 30 terabyte filesystems. Hopefully by the time that such filesystems are common, they will be running on 64-bit architectures.

In the event that **fsck** hits the memory limit of 32-bit architectures, Julian Elischer has suggested that one solution is to implement an "offline, non-in-place" version of **fsck** using all those techniques we learned in CS101 relating to mag-tape merge sorts. **Fsck** would have to have a small (20 gigabyte) disk partition set aside to hold working files, to which it would write files of records detailing block numbers, etc. Then it would do merge or block sorts on those files to get them in various orders (depending on fields in the records). **Fsck** would then recombine them to find such things as multiple referenced blocks and other file inconsistencies. It would be slow, but at least it could be used to check a 100 terabyte array, where the in-memory version would need a process VM space of 13 Gigabytes which is clearly impossible on the 32-bit PC.

Journalling filesystems provide a much faster state recovery than **fsck**. For this reason, there is ongoing work to provide a journalling option for UFS2. However, even journalling filesystems need to have a filesystem recovery program such as **fsck**. In the event of media or software failure, the filesystem can be damaged in ways that the journal cannot fix. Thus, the size of the recovery program is an issue for all filesystems. Indeed, the fact that UFS needs to use **fsck** in its general operation ensures that **fsck** is kept in good working order and is known to work even on very large filesystems.

9. Performance

The performance of UFS2 is nearly identical to that of UFS1. This similarity in performance is hardly surprising since the two filesystem share most of the same code base and use the same allocation algorithms. The purpose of UFS2 was not to try and improve on the performance of UFS1 which is already within 80-95% of the bandwidth of the disk. Rather it was to support multi-terabyte filesystems and to provide new capabilities such as extended attributes without losing performance. It has been successful in that goal.

10. Future Work

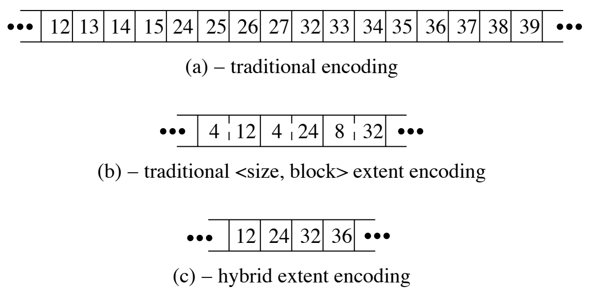
••• 12 13 14 15 24 25 26 27 32 33 34 35 36 37 38 39 •••
(a) – traditional encoding
••• 4 12 4 24 8 32 •••
(b) – traditional <size, block> extent encoding
••• 12 24 32 36 •••
(c) – hybrid extent encoding

Figure 3: *Alternative file metadata representations*

With the addition of dynamic block reallocation in the early 1990s [Seltzer & Smith, 1996], the UFS1 filesystem has had the ability to allocate most files contiguously on the disk. The metadata describing a large file consists of indirect blocks with long runs of sequential block numbers, see Fig. 3-(a). For quick access while a file is active, the kernel tries to keep all of a file's metadata in memory. With UFS2 the space required to hold the metadata for a file is doubled as every block pointer grows from 32-bits to 64-bits. To provide a more compact representation, many filesystems use an extent-based representation. A typical extent-based representation uses pairs of block numbers and lengths. Figure 3-(b) represents the same set of block number as Fig. 3-(a) in an extent-based format. Provided that the file can be laid out nearly contiguously, this representation provides a very compact description. However, randomly or slowly written files can end up with many non-contiguous block allocations which will produce a representation that requires more space than the one used by UFS1. This representation also has the drawback that it can require a lot of computation to do random access to the file since the block number needs to be computed by adding up the sizes starting from the beginning of the file until the desired seek offset is reached.

To gain most of the efficiencies of extends without the random access inefficiencies, UFS2 has added a field to the inode that will allow that inode to use a larger block size. Small, slowly growing, or sparse files set this value to the regular filesystem block size and represent their data in the traditional way show in Fig. 3-(a). However, when the filesystem detects a large dense file, it can set this inode-block-size field to a value two to sixteen times the filesystem block size. Figure 3-(c) represents the same set of block number as Fig. 3-(a) with the inode-block-size field set to four times the filesystem block size. Each block pointer references a piece of disk storage that is four times larger which reduces the metadata storage requirement by 75 percent. Since every block pointer other than possibly the last one references an equal sized block, computation of random access offsets is just as fast as in the traditional metadata representation. It also cannot degrade to a larger representation than the traditional metadata representation.

The drawback to this approach is that once a file has committed to using a larger block size, it can only utilize blocks of that size. If the filesystem runs out of big blocks then the file can no longer grow and either the application will get an ‘‘out-of-space’’ error, or the filesystem has to recreate the metadata with the standard filesystem block size. My current plan is to write the code to recreate the metadata. While recreating the metadata usually will cause a long pause, We expect that condition to be quite rare and not a noticeable problem in actual use.

11. Current Status

The UFS2 filesystem was developed for the FreeBSD Project by the author under contract to Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program. Under the terms of that contract, the software must be released under a Berkeley-style copyright. The UFS2 filesystem was written in 2002 and first released in FreeBSD 5.0. Extensive user feedback in that release has been helpful in shaking out latent short-comings particularly in the ability of UFS2 to smoothly handle the really big filesystems for which it was designed. The biggest current limitation is that the disk labels used in FreeBSD 5.0 can only describe 2 terabyte disks. We are hoping that the new larger disk labels will be available by the time FreeBSD 5.1 is released.

12. References

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

Best & Kleikamp, 2003.
S. Best & D. Kleikamp, "How the Journaled File System handles the on-disk layout," *http://www-106.ibm.com/developerworks/linux/library/l-jfslayout/* (2003).

Dowse & Malone, 2002.
I. Dowse & D. Malone, "Recent Filesystem Optimizations on FreeBSD," *Proceedings of the Freenix Track at the 2002 Usenix Annual Technical Conference,* p. 245−258 (June 2002).

Griffin et al, 2002.
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,* p. 75−88 (January 2002).

Lumb et al, 2002.
C. R. Lumb, J. Schindler, & G. R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," *Proceedings of the Usenix Conference on File and Storage Technologies,* p. 275−288 (January 2002).

McKusick, 2002.
M. McKusick, "Running Fsck in the Background," *Proceedings of the BSDCon 2002 Conference,* p. 55−64 (February 2002).

McKusick et al, 1996.
M. McKusick, K. Bostic, M. Karels, & J. Quarterman,, *The Design and Implementation of the 4.4BSD Operating System,* p. 269−271, Addison Wesley Publishing Company, Reading, MA (1996).

McKusick & Ganger, 1999.
M. McKusick & G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of the Freenix Track at the 1999 Usenix Annual Technical Conference,* p. 1−17 (June 1999).

McKusick et al, 1984.
M. McKusick, W. Joy, S. Leffler, & R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* 2, 3, p. 181−197 (August 1984).

Phillips, 2001.
D. Phillips, "A Directory Index for Ext2," *Proceedings of the Usenix Fifth Annual Linux Showcase and Conference* (November 2001).

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

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

Schindler et al, 2002.
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,* p. 259−274 (January 2002).

Seltzer et al, 2000.
M. Seltzer, G. Ganger, M. McKusick, K. Smith, C. Soules, & C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," *Proceedings of the San Diego Usenix Conference,* p. 71−84 (June 2000).

Seltzer & Smith, 1996.
M. Seltzer & K. Smith, "A Comparison of FFS Disk Allocation Algorithms," *Winter USENIX Conference,* p. 15−25 (January 1996).

Sweeney et al, 1996.
A. Sweeney, D. Doucette, C. Anderson, W. Hu, M. Nishimoto, & G. Peck, "Scalability in the XFS File System," *Proceedings of the 1996 Usenix Annual Technical Conference,* p. 1–14 (January 1996).

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

Watson, 2001.
R. Watson, "TrustedBSD: Adding Trusted Operating System Features to FreeBSD," *Proceedings of the Freenix Track at the 2001 Usenix Annual Technical Conference* (June 2001).

Watson et al, 2003.
R. Watson, W. Morrison, C. Vance, & 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).

13. Biography

Dr. 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 filesystem, 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. His particular areas of interest are the virtual-memory system and the filesystem. One day, he hopes to see them merged seamlessly. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a doctoral degree in Computer Science. He is president of the Usenix Association, and is a member of ACM and IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at http://www.mckusick.com/˜mckusick/) in the basement of the house that he shares with Eric Allman, his domestic partner of 24-and-some-odd years. You can contact him via email at <mckusick@mckusick.com>.