Running “Fsck” in the Background

Marshall Kirk McKusick
Author and Consultant

ABSTRACT

Traditionally, recovery of a BSD fast filesystem after an uncontrolled system crash such as a power failure or a system panic required the use of the filesystem checking program, “fsck”. Because the filesystem cannot be used while it is being checked by “fsck”, a large server may experience unacceptably long periods of unavailability after a crash.

Rather than write a new version of “fsck” that can run on an active filesystem, I have added the ability to take a snapshot of a filesystem partition to create a quiescent filesystem on which a slightly modified version of the traditional “fsck” can run.

A key feature of these snapshots is that they usually require filesystem write activity to be suspended for less than one second. The suspension time is independent of the size of the filesystem. To reduce the number and types of corruption, *soft updates* were added to ensure that the only filesystem inconsistencies are lost resources. With these two additions it is now possible to bring the system up immediately after a crash and then run checks to reclaim the lost resources on the active filesystems.

Background “fsck” runs by taking a snapshot and then running its traditional first four passes to calculate the correct bitmaps for the allocations in the filesystem snapshot. From these bitmaps, “fsck” finds any lost resources and invokes special system calls to reclaim them in the underlying active filesystem.

1. Background and Introduction

Traditionally, recovery of the BSD fast filesystem [McKusick, et al., 1996] after an uncontrolled system crash such as a power failure or a system panic required the use of the “fsck” filesystem checking program [McKusick & Kowalski, 1994]. “Fsck” would inspect all the filesystem metadata (bitmaps, inodes, and directories) and correct any inconsistencies that were found. As the metadata comprises about three to five percent of the disk space, checking a large filesystem can take an hour or longer. Because the filesystem is inaccessible while it is being checked by “fsck”, a large server may experience unacceptably long periods of unavailability after a crash.

Many methods exist for solving the metadata consistency and recovery problem [Ganger et al, 2000; Seltzer et al, 2000]. The solution selected for the fast filesystem is *soft updates*. Soft updates control the ordering of filesystem updates such that the only inconsistencies in the on-disk representation are that free blocks and inodes may be claimed as “in use” in the on-disk bitmaps when they are really unused.

Soft updates eliminates the need to run “fsck” after a system crash except in the rare event of a hardware failure in the disk on which the filesystem resides. Since the only filesystem inconsistency is lost blocks and inodes, the only ill effect from running on the filesystem after a crash is that part of the filesystem space that should have been available will be lost. Any time the lost space from crashes reaches an unacceptable level, the filesystem must be taken offline long enough to run “fsck” to reclaim the lost resources.

Because many server systems need to be available 24x7, there is never an available multi-hour time when a traditional version of “fsck” can be run. Thus, I have been motivated to develop a filesystem check program that can run on an active filesystem to reclaim the lost

This paper was first published in the “Proceedings of the BSDCon 2002 Conference,” San Francisco, California, USA, February 2002, pp 55−64.

resources.

The first approach that I considered was to write a new utility that would operate on an active filesystem to identify the lost resources and reclaim them. Such a utility would fall into the class of real-time garbage collection. Despite much research and literature, garbage collection of actively used resources remains challenging to implement. In addition to the complexity of real-time garbage collection, the new utility would need to recreate much of the functionality of the existing "fsck" utility. Having a second utility would lead to additional maintenance complexity as bugs or feature additions would need to be implemented in two utilities rather than just one.

To avoid the complexity and maintenance problems, I decided to take an approach that would enable me to use most of the existing "fsck" program.

"Fsck" runs on the assumption that the filesystem it is checking is quiescent. To create an apparently quiescent filesystem, I added the ability to take a *snapshot* of a filesystem partition [McKusick & Ganger, 1999]. Using copy-on-write, a snapshot provides a filesystem image frozen at a specific point in time. The next section describes the details on how snapshots are taken and managed.

As lost resources will not be used until they have been found and marked as free in the bitmaps, there are no limits on how long a background reclamation scheme can take to find and recover them. Thus, I can run the traditional "fsck" over a snapshot to find all the missing resources even if it takes several hours and the filesystem is being actively changed.

Snapshots may seem a more complex solution to the problem of running space reclamation on an active filesystem than a more straight forward garbage-collection utility. However, their cost (about 1300 lines of code) is amortized over the other useful functionality: the ability to do reliable dumps of active filesystems and the ability to provide back-ups of the filesystem at several times during the day. This functionality was first popularized by Network Appliance [Hitz et al, 1994; Hutchinson et al, 1999].

Snapshots enable the safe backup of 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.

Periodic snapshots can be made accessible to users. 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, BSD uses the memory-disk driver, *md*. The **mdconfig** command takes a snapshot file as input and produces a **/dev/md0** character-device interface to access it. The **/dev/md0** character device can then be used as the input device for a standard BSD FFS mount command, allowing the snapshot to appear as a replica of the frozen filesystem at whatever location in the namespace that the system administrator chooses to mount it. Thus, the following script could be run at noon to create a mid-day backup of the **/usr** filesystem and make it available at **/backups/usr/noon**:

```
# Take the snapshot
mount -u -o snapshot /usr/snap.noon /usr
# Attach it to /dev/md0
mdconfig -a -t vnode -u 0 -f /usr/snap.noon
# Mount it for user access
mount -r /dev/md0 /backups/usr/noon
```

When no longer needed, it can be removed with:

```
# Unmount snapshot
umount /backups/usr/noon
# Detach it from /dev/md0
mdconfig -d -u 0
# Delete the snapshot
rm -f /usr/snap.noon
```

2. Creating a Filesystem Snapshot

A filesystem *snapshot* is a frozen image of a filesystem at a given instant in time. Implementing snapshots in the BSD fast filesystem has proven to be straightforward. Taking a snapshot entails the following steps:

1) A *snapshot file* is created to track later changes to the filesystem; a snapshot file is shown in Fig. 1. This snapshot file is initialized to the size of the filesystem's partition, and its file block pointers are marked as zero which means "not copied." A few strategic blocks are allocated, such as those holding copies of the superblock and cylinder group maps.
2) A preliminary pass is made over each of the cylinder groups to copy it to its preallocated

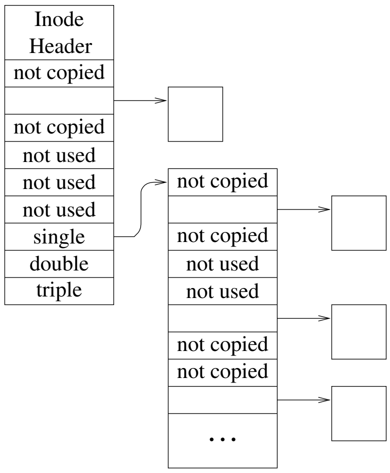

Figure 1: *Structure of a snapshot file*

backing block. Additionally, the block bitmap in each cylinder group is scanned to determine which blocks are free. For each free block that is found, the corresponding location in the snapshot file is marked with a distinguished block number (1) to show that the block is "not used." There is no need to copy those unused blocks if they are later allocated and written.

3) The filesystem is marked as "wanting to suspend." In this state, processes that wish to invoke system calls that will modify the filesystem are blocked from running, while processes that are already in progress on such system calls are permitted to finish them. These actions are enforced by inserting a gate at the top of every system call that can write to a filesystem. The set of gated system calls includes "write", "open" (when creating or truncating), "fhopen" (when creating or truncating), "mknod", "mkfifo", "link", "symlink", "unlink", "chflags", "fchflags", "chmod", "lchmod", "fchmod", "chown", "lchown", "fchown", "utimes", "lutimes", "futimes", "truncate", "ftruncate", "rename", "mkdir", "rmdir", "fsync", "sync", "unmount", "undelete", "quotactl", "revoke", and "extattrctl". In addition gates must be added to "pageout", "ktrace", local domain socket creation, and core dump creation. The gate tracks activity within a system call for each mounted filesystem. A gate has two purposes. The first is to suspend processes that want to enter the gated system call during periods that the filesystem that the process wants to modify is suspended. The second is to keep track of the number of processes that are running inside the gated system call for each mounted filesystem. When a process enters a gated system call, a counter in the mount structure for the filesystem that it wants to modify is incremented. When the process exits a gated system call, the counter is decremented.

4) The filesystem's status is changed from "wanting to suspend" to "fully suspended." This status change is done by allowing all system calls currently writing to the filesystem being suspended to finish. The transition to "fully suspended" is complete when the count of processes within gated system calls drops to zero.

5) The filesystem is synchronized to disk as if it were about to be unmounted.

6) Any cylinder groups that were modified after they were copied in step two are recopied to their preallocated backing block. Additionally, the block bitmap in each recopied cylinder group is rescanned to determine which blocks were changed. Newly allocated blocks are marked as "not copied" and newly freed blocks are marked as "not used." The details on how these modified cylinder groups are identified is described below. The amount of space initially claimed by a snapshot is small, usually less than a tenth of one percent. Actual snapshot file space utilization is given in section 4.

7) With the snapshot file in place, activity on the filesystem resumes. Any processes that were blocked at a gate are awakened and allowed to proceed with their system call.

8) Blocks that had been claimed by any snapshots that existed at the time that the current snapshot was taken are expunged from the new snapshot for reasons described below.

During steps three through six, all write activity on the filesystem is suspended. Steps three and four complete in at most a few milliseconds. The time for step five is a function of the number of dirty pages in the kernel. It is bounded by the amount of memory that is dedicated to storing file pages. It is typically less than a second and is independent of the size of the filesystem. Typically step six needs to recopy only a few cylinder groups, so it also completes in less than a second.

The splitting of the bitmap copies between steps two and six is the way that I avoid having the suspend time be a function of the size of the filesystem. By making our primary copy pass while the filesystem is still active, and then having only a few cylinder groups in need of recopying after it has been suspended, I keep the suspend time down to a small and usually filesystem size independent time.

The details of the two-pass algorithm are as follows. Before starting the copy and scan of all the cylinder groups, the snapshot code allocates a ‘‘progress’’ bitmap whose size is equal to the number of cylinder groups in the filesystem. The purpose of the ‘‘progress’’ bitmap is to keep track of which cylinder groups have been scanned. Initially, all the bits in the ‘‘progress’’ map are cleared. The first pass is completed in step two before the filesystem is suspended. In this first pass, all the cylinder groups are scanned. When the clinder group is read, its corresponding bit is set in the ‘‘progress’’ bitmap. The cylinder group is then copied and its block map is consulted to update the snapshot file as described in step two. Since the filesystem is still active, filesystem blocks may be allocated and freed while the cylinder groups are being scanned. Each time a cylinder group is updated because of a block being allocated or freed, its corresponding bit in the ‘‘progress’’ bitmap is cleared. Once this first pass over the cylinder groups is completed, the filesystem is ‘‘suspended.’’

Step six now becomes the second pass of the algorithm. The second pass need only identify and update the snapshot for any cylinder groups that were modified after it handled them in the first pass. The changed cylinder groups are identified by scanning the ‘‘progress’’ bitmap and rescanning any cylinder groups whose bits are zero. Although every bitmap would have to be reprocessed in the worst case, in practice only a few bitmaps need to be recopied and checked.

3. Maintaining a Filesystem Snapshot

Each time an existing block in the filesystem is modified, the filesystem checks whether that block was in use at the time that the snapshot was taken (i.e., it is not marked ‘‘not used’’). If so, and if it has not already been copied (i.e., it is still marked ‘‘not copied’’), a new block is allocated from among the ‘‘not used’’ blocks and placed in the snapshot file to replace the ‘‘not copied’’ entry. The previous contents of the block are copied to the newly allocated snapshot file block, and the modification to the original is then allowed to proceed. Whenever a file is removed, the snapshot code inspects each of the blocks being freed and claims any that were in use at the time of the snapshot. Those blocks marked ‘‘not used’’ are returned to the free list.

When a snapshot file is read, reads of blocks marked ‘‘not copied’’ return the contents of the corresponding block in the filesystem. Reads of blocks that have been copied return the contents in the copied block (e.g., the contents that were stored at that location in the filesystem at the time that the snapshot was taken). Writes to snapshot files are not permitted. When a snapshot file is no longer needed, it can be removed in the same way as any other file; its blocks are simply returned to the free list and its inode is zeroed and returned to the free inode list.

Snapshots may live across reboots. When a snapshot file is created, the inode number of the snapshot file is recorded in the superblock. When a filesystem is mounted, the snapshot list is traversed and all the listed snapshots are activated. The only limit on the number of snapshots that may exist in a filesystem is the size of the array in the superblock that holds the list of snapshots. Currently, this array can hold up to twenty snapshots.

Multiple snapshot files can exist concurrently. As described above, earlier snapshot files would appear in later snapshots. If an earlier snapshot is removed, a later snapshot would claim its blocks rather than allowing them to be returned to the free list. This semantic means that it would be impossible to free any space on the filesystem except by removing the newest snapshot. To avoid this problem, the snapshot code goes through and expunges all earlier snapshots by changing its view of them to being zero length files. With this technique, the freeing of an earlier snapshot releases the space held by that snapshot.

When a block is overwritten, all snapshots are given an opportunity to copy the block. A copy of the block is made for each snapshot in which the block resides. Overwrites typically occur only for inode and directory blocks. File data is usually not overwritten. Instead, a file will be truncated and then reallocated as it is rewritten. Thus, the slow and I/O intensive block copying is infrequent.

Deleted blocks are handled differently. The list of snapshots is consulted. When a snapshot is found in which the block is active (‘‘not copied''), the deleted block is claimed by that snapshot. The traversal of the snapshot list is then terminated. Other snapshots for which the block are active are left with an entry of ''not copied'' for that block. The result is that when they access that location, they will still reference the deleted block. Since snapshots may not be modified, the block will not change. Since the block is claimed by a snapshot, it will not be allocated to another use. If the snapshot claiming the deleted block is deleted, the remaining snapshots will be given the opportunity to claim the block. Only when none of the remaining snapshots wants to claim the block (i.e., it is marked ''not used'' in all of them) will it be returned to the freelist.

4. Snapshot Performance

The experiments described in this section and the background ''fsck'' performance section used the following hardware/software configuration:

Computer: Dual Processor using two Celeron 350MHz CPUs. The machine has 256Mb of main memory.

O/S: FreeBSD 5.0-current as of December 30, 2001

I/O Controller: Adaptec 2940 Ultra2 SCSI adapter

Disk: Two <IBM DDRS-39130D DC1B> Fixed Direct Access SCSI-2 device, 80MB/s transfers, Tagged Queuing Enabled, 8715MB, 17,850,000 512 byte sectors: 255H 63S/T 1111C

Small Filesystem: 0.5Gb, 8K block, 1K fragment, 90% full, 70874 files, initial snapshot size 0.392Mb (0.08% of filesystem space).

Large Filesystem: 7.7Gb, 16K block, 2K fragment, 90% full, 520715 files, initial snapshot size 2.672Mb (0.03% of filesystem space).

Load: Four continuously running simultaneous Andrew benchmarks that create a moderate amount of filesystem activity intermixed with periods of CPU intensive activity [Howard et al, 1988].

Table 1 shows the time to take a snapshot on an idle filesystem. The elapsed time to take a snapshot is proportional to the size of the filesystem being snapshotted. However, nearly all the time to take a snapshot is spent in steps one, two, and eight. Because the filesystem permits other processes to modify the filesystem during steps one, two, and eight, this part of taking a snapshot does not interfere with normal system operation. The ''suspend time'' column shows the amount of real-time that processes are blocked from executing system calls that modify the filesystem. As Table 1 shows, the period during which write activity is suspended, and thus apparent to processes in the system, is short and does not increase proportionally to filesystem size.

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	0.7 sec	0.1 sec	0.025 sec
7.7Gb	3.5 sec	0.4 sec	0.034 sec

Table 1: *Snapshot times on an idle filesystem*

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	3.7 sec	0.1 sec	0.027 sec
7.7Gb	12.1 sec	0.4 sec	0.036 sec

Table 2: *Snapshot times on an active filesystem*

Table 2 shows the times to snapshot a filesystem that has four active concurrent processes running. The elapsed time rises because the process taking the snapshot has to compete with the other processes for access to the filesystem. Note that the suspend time has risen slightly, but is still insignificant and does not increase in proportion to the size of the filesystem under test. Instead, it is a function of the level of write activity present on the filesystem.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	0.5 sec	0.02 sec
7.7Gb	2.3 sec	0.09 sec

Table 3: *Snapshot removal time on an idle fs*

Table 3 shows the times to remove a snapshot on an idle filesystem. The elapsed time to remove a snapshot is proportional to the size of the filesystem being snapshotted. The filesystem does not need to be suspended to remove a snapshot.

Table 4 shows the times to remove a snapshot on a filesystem that has four active concurrent processes running. The elapsed time rises because the process removing the snapshot has to compete with the other processes for access to the filesystem. The filesystem does not need to be suspended to remove a snapshot.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	1.4 sec	0.03 sec
7.7Gb	4.9 sec	0.10 sec

Table 4: *Snapshot removal time on an active fs*

5. Implementation of Background "Fsck"

Background "fsck" runs by taking a snapshot then running its traditional algorithms over the snapshot. Because the snapshot is taken of a completely quiescent filesystem, all of whose dirty blocks have been written to disk, the snapshot appears to "fsck" to be exactly like an unmounted raw disk partition. "Fsck" runs in five passes that can be summarized as follows:

1) Scan all the allocated inodes to accumulate a list of all their allocated blocks. The fast filesystem preallocates all the inodes that the filesystem will ever be able to use at the time that the filesystem is created. The superblock contains information on where all the preallocated inodes can be found. Thus, "fsck" can find all of them without need of any additional information such as an index-of-inodes file or any of the filesystem directory structure. As part of scanning all the allocated inodes, "fsck" also identifies all the inodes that are allocated as directories for use in the next three passes. When the first pass completes, "fsck" has a list of all the allocated blocks and inodes.
2) Scan all the directories found in pass one. For each entry found in a directory, increment the reference count in the inode that it references. If the referenced inode is a directory, verify that its dot-dot entry points back properly. Also note that an entry to the directory has been found. The one exception to the dot-dot rule is for the root of the filesystem (inode two) whose dot-dot entry should point to itself.
3) Check that every directory (except the root) was found during the second pass. If any directories were not found, they have been somehow lost from the main tree. In traditional "fsck" any lost directories would be placed into the **lost+found** directory. When running with soft updates, unreferenced directories only occur if they were in the process of being deleted. So, if any turn up in pass three, "fsck" marks them for deletion in pass four.
4) If any inodes have a higher reference count than the number of directory entries that reference them, their reference count is adjusted to reflect the correct number of references. Such errors occur when a directory entry has been deleted but the system crashed before updating the on-disk reference count in the inode. The reference count should never need to be increased; if such a condition is found, "fsck" marks the filesystem as needing manual intervention and exits. There is a special case in which no entries for an inode were found. When running with soft updates, an unreferenced inode can only happen if the file was in the process of being deleted. Thus, background "fsck" requests the kernel to decrement the reference count on the inode to zero. The kernel then follows the usual code path for inodes with a zero reference count that releases the inode's claimed blocks and then releases the inode itself. The freed inode and any blocks that it claimed are removed from the list of valid inodes and blocks found in the first pass. In traditional "fsck" running on a filesystem that is not using soft updates, unreferenced inodes are placed in the **lost+found** directory.
5) The list of valid inodes and blocks determined in the first pass and updated in the fourth pass is compared against the bitmaps in the cylinder group maps. If there are any disagreements, the bitmaps in the cylinder groups are updated with the correct entries.

The new background version of "fsck" cannot update the on-disk image of the filesystem as the filesystem state will be different from that of the snapshot. Additionally, a user-level program cannot obtain the kernel-level locks needed to provide consistent updates of filesystem data structures. To ensure that the filesystem state is set using the appropriate locking protocol, a set of system calls was added to enable "fsck" to pass the resource-update requests to the kernel so that they can be made under the appropriate lock.

Five operations were implemented in the kernel:

1) set/clear superblock flags
2) adjust an inode block count
3) adjust an inode reference count
4) free a range of inodes

5) free a range of blocks/fragments

The background version of ‘‘fsck’’ is derived from the traditional disk-based ‘‘fsck’’ by augmenting the traditional ‘‘fsck’’ with calls to the kernel functions in place of writes to the filesystem partition. If at any time, ‘‘fsck’’ finds any inconsistencies other than lost blocks and inodes or high block or reference counts, then either a hardware or software error has occurred and a traditional execution of ‘‘fsck’’ needs to be run. ‘‘Fsck’’ sets a superblock flag (using operation 1) to force this check to be done before the next time that the filesystem is mounted. Optionally, it can forcibly downgrade a corrupted filesystem to read-only. In more detail, the changes to each pass of ‘‘fsck’’ are as follows:

1) After scanning each allocated inode, ‘‘fsck’’ compares the number of blocks that it claims with its count of the number of blocks that it is using. If these values differ, the traditional ‘‘fsck’’ would write back the inode with the updated value. Incorrect block counts typically occur when a partially truncated inode is encountered. The background ‘‘fsck’’ uses operation 2 to have the kernel adjust the count. As the file may be actively growing, the adjustment is done as an increment or decrement to the current value rather than setting an absolute value. No other changes to pass one are required.
2) No changes to pass two are required.
3) If any orphaned directories are found in pass three, they are assumed to have been in the process of being deleted. Thus they are marked for deletion in pass four.
4) In the traditional ‘‘fsck’’, inodes with high but non-zero reference counts need to have their reference counts adjusted. Inodes with zero reference counts need to be zeroed out on disk. With the background ‘‘fsck’’ these two operations can be subsumed into a single system call (operation 3) that adjusts the reference count on an inode. If the count is reduced to zero, the kernel will deallocate and zero out the inode as part of its normal course of operation. So, no additional work is required of ‘‘fsck’’. As with the adjustment of the block count in pass one, the reference count on the inode may have changed since the snapshot because of ongoing filesystem activity. Thus, the adjustment is given as a delta rather than as an absolute value to ensure that the inode retains the correct reference count.
5) The final pass taken by the traditional ‘‘fsck’’ is to rewrite the filesystem bitmaps to reflect the allocations that it has found. As an active filesystem will have continued to allocate and free resources, the state of the bitmaps calculated in the snapshot by the background ‘‘fsck’’ will not be correct, so it cannot write them back in their entirety. However, it can figure out which blocks and inodes are lost by doing an exclusive-or of the bitmaps in the snapshot with the bitmaps that it has calculated. The resulting non-zero bits will be the lost resources. Having determined which resources are lost, ‘‘fsck’’ must cause the live bitmaps to be repaired.

   If an inode has been zeroed on the disk, but has not been marked free in the bitmaps, then it is so marked (using operation 4). If there were any unclaimed blocks that were not released when adjusting the inode reference counts, they are freed (using operation 5). These unclaimed blocks arise from an inode that was zeroed on disk, but whose formerly claimed blocks were not freed before the system crashed.

The final step after a successful background ‘‘fsck’’ run is to update the filesystem status in the superblock. There are two flags in the superblock that track the state of a filesystem. The first is the ‘‘clean’’ flag that is set when a filesystem is unmounted (by the system administrator or at system shutdown) and cleared while it is mounted with writing enabled. The second is the ‘‘unclean-at-mount’’ flag that is described below.

The ‘‘clean’’ flag is used by the traditional ‘‘fsck’’ to decide which filesystems need to be checked. Those filesystems with the flag set are skipped; those filesystems with the flag clear are checked. Following a successful check, the ‘‘clean’’ flag is set. Before soft updates, the kernel did not allow unclean filesystems (e.g., filesystems with the ‘‘clean’’ flag cleared) to be mounted for writing as the corruption could cause the system to panic.

The ‘‘unclean-at-mount’’ flag was added as part of soft updates. Unclean filesystems running with soft updates are safe to mount with writing permitted. However, the system needs to remember that some cleanup may be required. Thus, the ‘‘unclean-at-mount’’ flag gets set when an unclean filesystem is mounted (e.g., mounted with writing enabled without the "clean" flag having been set). The "unclean-at-mount" flag serves two purposes. First, when a filesystem with the "unclean-at-mount" is unmounted, the "clean" flag is not set to show that cleaning is still required. Second it tracks the filesystems that need cleaning. By the time that background "fsck" is run, all the filesystems are mounted so none will have their "clean" flag set. Thus, the "unclean-at-mount" flag is used by the background "fsck" to distinguish which filesystems need to be checked. Those filesystems with the flag set are checked; those filesystems with the flag clear are skipped.

So, the final step after a successful run of a background "fsck" is to clear the "unclean-at-mount" bit in the superblock (using operation 1) so that the filesystem will be marked "clean" when it is unmounted by the system administrator or at system shutdown.

6. Operation of Background "Fsck"

Traditionally, "fsck" is invoked before the filesystems are mounted and all checks are synchronously done to completion at that time. If background checking is available, "fsck" is invoked twice. It is first invoked at the traditional time, before the filesystems are mounted, with the −F flag to do checking on all the filesystems that cannot do background checking. Filesystems that require traditional checking are those that are not running with soft updates and those that will not be mounted at system startup (e.g., those marked "noauto" in **/etc/fstab**). It is then invoked a second time, after the system has completed going multiuser, with the −B flag to do checking on all the filesystems that can do background checking. Unlike the foreground checking, the background checking is started asynchronously so that other system activity can proceed even on the filesystems that are being checked.

The "fsck" program is really just a front end that reads the **/etc/fstab** file and determines which filesystems need to be checked. For each filesystem to be checked, the appropriate back end is invoked. For the fast filesystem, "fsck_ffs" is invoked. If "fsck" is invoked with neither the −F nor the −B flag, it runs in traditional mode and checks every listed filesystem. Otherwise it is invoked with the −F to request that it run in foreground mode. In foreground mode, the check program for each filesystem is invoked with the −F flag to determine whether it wishes to run as part of the boot up sequence, or if it is able to do its job in background after the system is up and running. A non-zero exit code indicates that it wants to run in foreground and it is invoked again with neither the −F nor the −B flag so that it will run in its traditional mode. A zero exit code indicates that it is able to run later in background and just a deferred message is printed. The "fsck" program attempts to run as many checks in parallel as possible. Typically it can run a check on each disk in parallel.

After the system has gone multiuser, "fsck" is invoked with the −B flag to request that it run in background mode. The check program for each filesystem is invoked with the −F flag to determine whether it wishes to run as part of the boot up sequence, or if it is able to do its job in background after the system is up and running. A non-zero exit code indicates that it wanted to run in foreground that is assumed to have been done, so the filesystem is skipped. A zero exit code indicates that it is able to run in background so the check program is invoked with the −B flag to indicate that a check on the active filesystem should be done. When running in background mode, only one filesystem at a time will be checked. To further reduce the load on the system, the background check is typically run at a *nice* value of plus four.

The "fsck_ffs" program does the actual checking of the fast filesystem. When invoked with the −F flag, "fsck_ffs" determines whether the filesystem needs to be checked immediately in foreground or if its checking can be deferred to background. To be eligible for background checking it must have been running with soft updates, not have been marked as needing a foreground check, and be mounted and writable when the background check is to be done. If these conditions are met, then "fsck_ffs" exits with a zero exit status. Otherwise it exits with a non-zero exit status. If the filesystem is clean, it will exit with a non-zero exit status so that the clean status of the filesystem can be verified and reported during the foreground checks. Note that, when invoked with the −F flag, no checking is done. The only thing that "fsck_ffs" does is to determine whether a foreground or background check is needed and exit with an appropriate status code.

When "fsck_ffs" is invoked with the −B flag, a check is done on the specified and possibly active filesystem. The potential set of corrections is limited to those available when running in preen mode (as further detailed in the previous section). If unexpected errors are found, the filesystem is marked as needing a foreground check and ‘‘fsck_ffs’’ exits without attempting any further checking.

7. Background ‘‘Fsck’’ Performance

Over many years of tuning and refinement, ‘‘fsck’’ has been optimized to minimize and cluster its I/O requests. Further, its data structures have been tuned to the point where it consumes little CPU time and thus its running time is totally dominated by the time that it takes to do the needed I/O. The performance of background ‘‘fsck’’ is almost identical to that of the traditional ‘‘fsck’’. Since it is using the same algorithms, the number and pattern of its I/O requests are identical to the traditional program. Though the update requests are done through kernel calls rather than direct writes to the disk, the kernel calls typically execute the same (or through the benefits of soft updates) slightly fewer disk writes. The main added delay of background ‘‘fsck’’ is the time required to take and remove a snapshot of the filesystem. The time for the snapshot operation has already been covered in section 4 above.

Because of ‘‘fsck’’’s I/O clustering, it is capable of using nearly all the bandwidth of a disk. Although background ‘‘fsck’’ only checks one filesystem partition at a time (as compared to traditional ‘‘fsck’’ that checks all separate disks containing filesystems in parallel), even a single instance of ‘‘fsck’’ can cause seriously increased latency to processes trying to access files on the filesystem (or anything else on the same disk) that is being checked.

As there is no urgency in completing the space reclamation, background ‘‘fsck’’ is usually run at lower priority than other processes. The usual way to reduce priority is to *nice* the process to some positive value which results in it getting a lower priority for the CPU. Because ‘‘fsck’’ is nearly completely I/O bound, giving it a lower CPU priority has almost no effect on the time in which it runs and hence in its rate of issuing I/O requests.

As a general solution to reducing the resource usage of I/O bound processes such as background ‘‘fsck’’, a small change has been made to the disk strategy routine. When an I/O request is posted, the disk strategy routine first checks whether the process is running at a positive *nice*. If it is, and there are any other outstanding I/O requests for the disk, the process is put to sleep for *nice* hundredths of a second. Thus, a process running at a *nice* of four will sleep for forty millisecond each time it makes a disk I/O request. Such a process will be able to do at most twenty-five disk I/O requests per second − about a third of the bandwidth of a current technology disk. At the maximum *nice* value of twenty, a process is limited to five I/O requests per second which is low enough to be almost unnoticeable by other processes competing for access to the disk. Because the slow-down is imposed only when there are other outstanding disk requests, I/O bound processes can run at full speed on an otherwise idle system.

Filesystem Size	Elapsed Time	CPU Time
0.5Gb	5.8 sec	1.1 sec
7.7Gb	50.9 sec	8.3 sec

Table 5: *Traditional fsck times on an idle fs*

Table 5 shows the times to run the traditional ‘‘fsck’’ on a filesystem that is otherwise idle. It is running with a *nice* value of zero. It is the only process active on the system, so represents a lower bound on the time that a traditional disk check can be done.

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	8.1 sec	2.5 sec	0.025 sec
7.7Gb	61.5 sec	14.9 sec	0.050 sec

Table 6: *Background fsck times on an idle fs*

Table 6 shows the times to run background ‘‘fsck’’ on a filesystem that is otherwise idle. It is running with a *nice* value of zero. It is the only process active on the system, so represents a lower bound on the time that a background disk check can be done. Note that its running time is only slightly greater than would be expected by adding the time to take and remove a snapshot (see Table 1 and Table 3) to the running time of the traditional ‘‘fsck’’ shown in Table 5.

Table 7 shows the times to run background ‘‘fsck’’ on a filesystem that has four processes concurrently writing to it. It is running with a *nice* value of four. Note that its running time increases by a factor of ten as it is yielding to the other running processes. By contrast, its effect on the other processes is minimal as their aggregate throughput is slowed by less than ten

Filesystem Size	Elapsed Time	CPU Time	Suspend Time
0.5Gb	122 sec	2.9 sec	0.025 sec
7.7Gb	591 sec	16.2 sec	0.050 sec

Table 7: *Background fsck times on an active fs*

percent.

8. Conclusions and Future Work

This paper has described how to take snapshots of the fast filesystem with suspension intervals typically less than one second and independent of the size of the filesystem being snapshotted. When running with soft updates, the only filesystem corruption that occurs is the loss of inodes and data blocks. Using snapshots together with a slightly modified version of the traditional ‘‘fsck’’ it is possible to recover the lost inodes and data blocks while the filesystem is in active use. While a background ‘‘fsck’’ can run in about the same amount of time as a traditional ‘‘fsck’’, it is generally desirable to run it at a lower priority so that it causes less slowdown on other processes on the system.

Snapshots may seem a more complex solution to the problem of running space reclamation on an active filesystem than a more straight forward garbage-collection utility. However, their cost (about 1300 lines of code) is amortized over the other useful functionality: the ability to do reliable dumps of active filesystems and the ability to provide back-ups of the filesystem at several times during the day.

For the future, I need to gain experience with using background ‘‘fsck’’, to gain confidence in its robustness, and to find the optimal priority to minimize its slowdown on the system while still finishing its job in a reasonable amount of time.

9. Current Status

Snapshots and background ‘‘fsck’’ have been running on FreeBSD 5.0 systems since April 2001. All the relevant code including snapshots, the gating functions, the system call additions for the use of ‘‘fsck’’, and the changes to ‘‘fsck’’ itself are available as open-source under a Berkeley-style copyright.

10. Acknowledgments

I thank Rob Kolstad for his helpful comments and review of drafts of this paper. He pointed out several non-obvious gaps in the coverage. Thanks also to Matthew Dillon, Ian Dowse, Peter Wemm, and many other FreeBSD developers that put up with my kernel breakage and helped track down the bugs.

11. References

Ganger et al, 2000.
G. Ganger, M. McKusick, C. Soules, & Y. Patt, “Soft Updates: A Solution to the Metadata Update Problem in Filesystems,” *ACM Transactions on Computer Systems,* 2, p. 127−153 (May 2000).

Hitz et al, 1994.
D. Hitz, J. Lau, & M. Malcolm, “File System Design for an NFS File Server Appliance,” *Proceedings of the San Francisco USENIX Conference,* p. 235−246 (January 1994).

Howard et al, 1988.
J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, & M. West, “Scale and Performance in a Distributed System,” *ACM Transactions on Computer Systems,* 6, 1, p. 51−81 (February 1988).

Hutchinson et al, 1999.
N. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, & S. OMalley, “Logical vs. Physical File System Backup,” *Third USENIX Symposium on Operating Systems Design and Implementation,* p. 239−250 (February 1999).

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,” *Freenix Track at the Annual USENIX Technical Conference,* p. 1−17 (June 1999).

McKusick & Kowalski, 1994.
M. McKusick & T. Kowalski, “FSCK - The UNIX File System Check Program,”

4.4 BSD System Manager's Manual, p. 3:1−21, O'Reilley & Associates, Inc., Sebastopol, CA (April 1994).

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).

12. 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. 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 earned Masters degrees in Computer Science and Business Administration, and a doctorate in Computer Science. He is a past president and present board member 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 20-and-some-odd years. You can contact him via email at `<mckusick@mckusick.com>`.