Soft Updates: A Solution to the Metadata Update Problem in File Systems

Gregory R. Ganger, Yale N. Patt
Department of EECS, University of Michigan
ganger,patt@eecs.umich.edu

Abstract

Structural changes, such as file creation and block allocation, have consistently been identified as a source of problems (performance, integrity, security and availability) for file systems. This report describes ***soft updates****, an implementation technique that allows a file system to safely use delayed writes for metadata updates. We show that a file system using soft updates asymptotically approaches memory-based file system performance while providing stronger integrity and security guarantees than most UNIX file systems. For metadata update intensive benchmarks, this improves performance by more than a factor of two when compared to the conventional synchronous write approach. In addition, soft updates can improve file system availability by relegating crash-recovery assistance (e.g., the* fsck *utility) to an optional and/or background role, reducing file system recovery time to a few seconds.*

1 Introduction

In a file system, **metadata** (e.g., directories, inodes and free block/inode maps) gives structure to the raw storage capacity. It provides for linking multiple disk sectors into files and identifying those files (e.g., via names). To be useful for non-temporary storage, a file system must maintain the integrity of its metadata in the face of unpredictable system failures (e.g., power loss).[1]

Traditionally, file systems protect metadata integrity by carefully sequencing certain disk writes.[2] For example, when creating a new file, the file system allocates an inode, initializes it and constructs a directory entry that points to it. If the system goes down after the new directory entry has been written to disk but before the initialized inode is written, integrity may be compromised since the contents of the on-disk inode are unknown. To protect metadata consistency, the initialized inode must reach stable storage before the new directory entry. We refer to this requirement as an update **dependency**, as writing the directory entry **depends** on first writing the inode. The sequencing constraints essentially map onto three simple rules: (1) Never point to a structure before it has been initialized, (2) Never re-use a resource before nullifying all previous pointers to it, and (3) Never reset the old pointer to a resource before the new pointer has been set (when renaming files).

Synchronous[3] writes are used for metadata update sequencing by many variants of both the original UNIX^{TM} file system [Ritchie78] and the Berkeley fast file system (FFS) [McKusick84]. As a result, metadata updates in these file systems proceed at disk speeds rather than processor/memory speeds [Ousterhout90, McVoy91, Seltzer93]. The performance degradation can be so dramatic that many implementations choose to ignore certain update dependencies, reducing integrity, security and/or availability. For example, many file systems do not initialize newly allocated blocks before attaching them to files, which can reduce both integrity and security (the uninitialized block often contains previously deleted file data). Also, many file systems do not guarantee the state of the on-disk free block/inode maps, electing to reconstruct them after a system failure (e.g., with the *fsck* utility [McKusick94]).

[Ganger94] evaluates an alternative approach to update sequencing, called scheduler-enforced ordering, in which the file system uses asynchronous writes for metadata and passes any sequencing restrictions to the disk scheduler with each request. This approach was shown to outperform

[1]This paper assumes that main memory is volatile (i.e., its state is lost during power outages). The issues addressed become much simpler when main memory is non-volatile.

[2]With this approach, each individual on-disk metadata update must also be atomic. This can be achieved by forcing each individual metadata structure (e.g., a directory entry or an inode) to be fully contained by a single disk sector. Each disk sector is protected by error correcting codes that will almost always flag a partially written sector as unrecoverable. This may result in loss of structures, but not loss of integrity. In addition, many disks will not start laying down a sector unless there is sufficient power to finish it.

[3]There are three types of UNIX file system writes: synchronous, asynchronous and delayed. A write is **synchronous** if the process issues it (i.e., sends it to the device driver) immediately and waits for it to complete. A write is **asynchronous** if the process issues it immediately but does not wait for it to complete. A **delayed** write is not issued immediately; the relevant buffer cache blocks are marked dirty and flushed later by a background process (unless the cache runs out of clean blocks).

the conventional implementation by more than 30 percent for metadata update intensive benchmarks. However, with scheduler-enforced ordering, delayed writes cannot safely be used when sequencing is required. Also, all disk schedulers (generally located in storage device drivers) must support the modified interface and the corresponding sequencing rules.

Most other approaches to maintaining metadata integrity entail some form of logging (e.g., [Hagmann87, Chutani92, Journal92]) or shadow-paging (e.g., [Chamberlin81, Ston87, Chao92, Seltzer93]). Generally speaking, these approaches augment the on-disk state with additional information that can be used to reconstruct the metadata after most system failures.[4] While these approaches have been successfully applied, there is value in exploring implementations that do not require changes to the on-disk structures (which may have a large installed base) and may offer higher performance with lower complexity.

This report describes **soft updates**, an implementation technique that allows a file system to safely use delayed writes for metadata updates [Ganger94]. With soft updates, the cost of maintaining integrity is low and performance asymptotically approaches that of a memory-based file system (within a few percent). For metadata update intensive benchmarks, this improves performance by more than a factor of two when compared to the conventional approach. Also, additional update sequencing can be realized with little performance loss. So, integrity and security can be improved relative to many current implementations. Further, the on-disk state can be maintained such that the file system can be safely mounted and used immediately after a system failure,[4] reducing file system recovery times by more than two orders of magnitude.

The remainder of this report is organized as follows. Section 2 describes soft updates in general. Section 3 describes our re-implementation of a UNIX file system, which uses soft updates to improve performance, integrity, security and availability. Section 4 evaluates the performance of our soft updates implementation. Section 5 discusses important non-performance characteristics, including file system recovery, user-interface semantics and implementation complexity. Section 6 draws conclusions and discusses avenues for future research. The appendix provides the low-level details and non-proprietary portions of our soft updates implementation.

2 Soft Updates: General

Using delayed writes for metadata can substantially improve performance by combining multiple updates into a much smaller quantity of background disk writes. The savings come from two sources: (1) multiple updates to a single metadata component (e.g., removal of a recently added directory entry), and more significantly, (2) multiple independent updates to a block of metadata (e.g., several entries added to a directory block).

To maintain integrity, sequencing constraints must be upheld as dirty blocks are written to stable storage. To meet this requirement, dependency information is associated with the in-

[4]The exception to this is a media failure. While rare, a media failure can result in complete loss of some metadata. In this case, off-line assistance is necessary. Generally, the lost metadata is replaced with default values and the remaining metadata is updated accordingly.

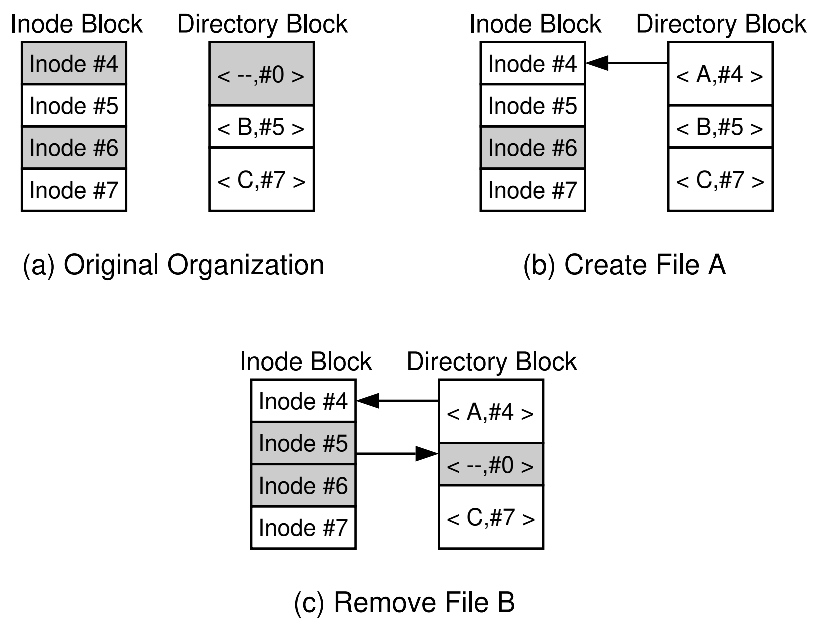


Figure 1: A Cyclic Dependency. (b) and (c) each show a pair of in-memory metadata blocks after a structural change. The shaded metadata structures are de-allocated and ready for reuse. The arrows indicate dependencies. When creating a new file, the newly initialized inode must be written to disk before the new directory entry. When removing a file, the reset directory entry must be written before the re-initialized inode. Viewed at a block level, the two metadata blocks depend on each other. Viewed at a finer granularity, there are two independent update sequences.

memory metadata. When making a structural change, the file system modifies the in-memory copies of the relevant metadata (via delayed writes) and updates the corresponding dependency information. The dependency information is then used when dirty blocks are flushed to disk.

When we began this work, we envisioned a dynamically managed DAG (Directed, Acyclic Graph) of dirty blocks for which disk writes are issued only after all writes on which they depend complete. In practice, we found this to be a very difficult model to maintain, being susceptible to cyclic dependencies and aging problems (blocks could consistently have dependencies and never be written to stable storage). Like false sharing in multiprocessor caches, these difficulties relate to the granularity of the dependency information. The blocks that are read from and written to disk often contain multiple metadata structures (e.g., inodes or directory fragments), each of which generally contains multiple dependency causing components (e.g., block pointers and directory entries). As a result, originally independent metadata changes can easily cause dependency cycles (see figure 1) and excessive aging. Detecting and handling these problems increases implementation complexity and reduces performance.

With soft updates, the file system maintains dependency information at a very fine granularity. Before and after copies are kept for each individual metadata update as well as a list of update(s) on which it depends. A block containing dirty metadata can be written at any time, so long as any updates within that block that have pending dependencies are first temporarily "undone" (rolled back). Thus, every block, as written to disk, is consistent with respect to the current on-disk state. When a disk write completes, any undone updates in the source memory block are restored before it can be accessed. So, for example, the two metadata blocks in figure 1c can be safely transferred to disk with three writes (see figure 2). With this approach, aging problems do not occur because new dependencies are not added to existing update sequences. Dependency cycles do not occur because independent sequences of dependent updates remain independent and no single sequence is cyclic.

3 Soft Updates: Implementation

3.1 System Software Base

We have implemented soft updates in UNIX SVR4 MP, AT&T/GIS's production operating system for symmetric multiprocessing. We use the *ufs* file system for our experiments, which is based on the Berkeley fast file system [McKusick84]. File system caching is well integrated with the virtual memory system, which is similar to that of SunOS [Gingell87, Moran87].

One important aspect of the file system's reliability and performance is the **syncer daemon**. This background process executes at regular intervals, writing out dirty buffer cache blocks. The syncer daemon in UNIX SVR4 MP operates differently from the conventional "30 second *sync*"; it awakens once each second and sweeps through a fraction of the buffer cache, initiating an asynchronous write for each dirty block encountered. This approach tends to reduce the burstiness associated with the conventional approach.

3.2 Basic Changes

In our implementation, almost all of the synchronous and asynchronous metadata updates have been replaced with delayed writes.[5] Prior to each change, code has been added to set up dependency information. Dependency-setup code has also been added before metadata updates whose integrity is not protected in the conventional implementation (e.g., block allocation). Additional procedures are invoked to enforce, update and remove dependency structures as dirty blocks are written to disk. Some dependencies are handled by the undo/redo approach described in the previous section. Others are handled by postponing in-memory updates until after the updates on which they depend have been written to stable storage. In a sense, these deferred updates are undone until the disk writes on which they depend complete.

When a disk write completes, there is often some processing needed to update/remove dependency information, restore undone changes, and deal with deferred work. An implementation of soft updates requires some method of performing these tasks in the background. Very simple

[5]Some exceptions are: (1) when the user explicitly requests synchronous updates, as with the *fsync()* system call or the O_SYNC modifier to the *open()* system call, (2) when unmounting a file system, and (3) when updating the superblock.

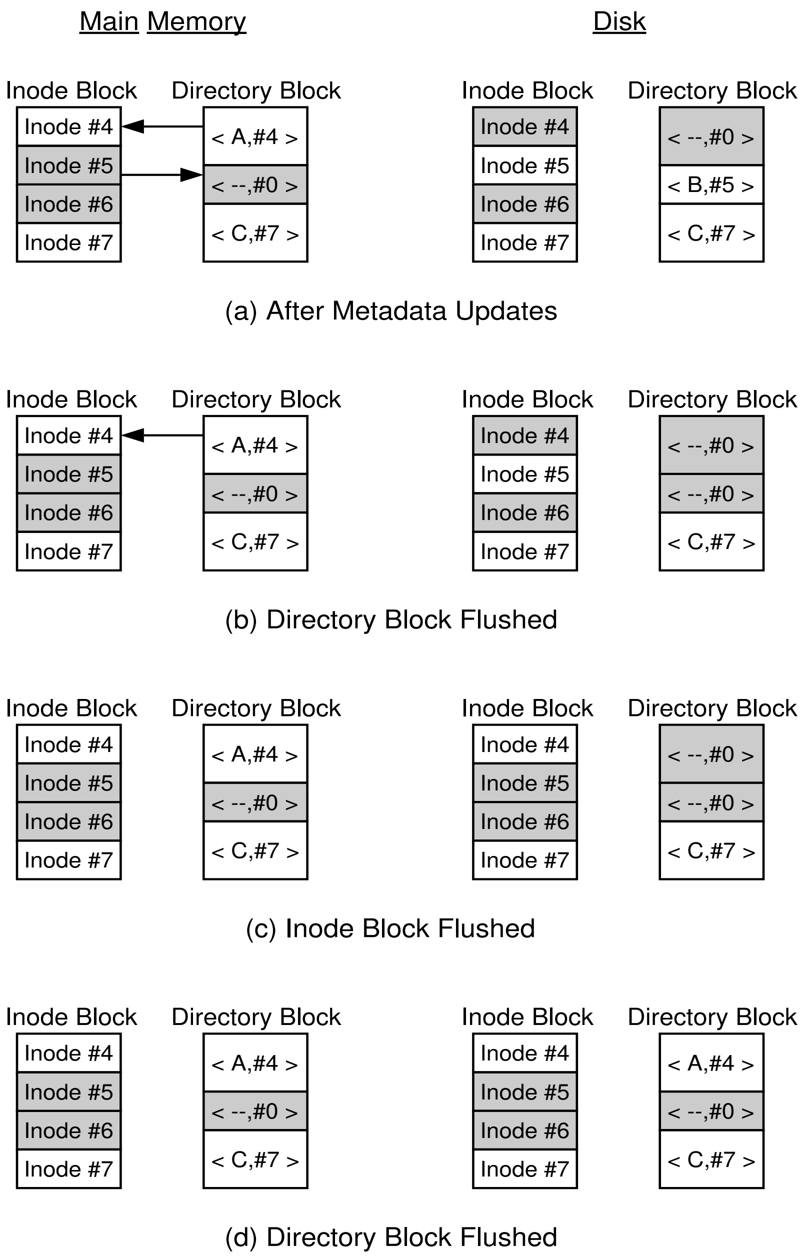


Figure 2: Undo/Redo Operations in Soft Updates. (a) shows the in-memory and on-disk copies of the two modified metadata blocks from figure 1c. (b), (c) and (d) show the same blocks after each of three disk writes. For the duration of each disk write, the in-memory copy matches the resulting on-disk copy.

changes can be made during the disk I/O completion interrupt service routine (ISR), which calls a pre-defined procedure in the higher-level module that issued the request. However, any task that can block and wait for a resource (e.g., a lock or, worse yet, an uncached disk block) cannot be handled in this way. Such a task must be handled outside of the ISR, preferably by a background process. We use the syncer daemon for this purpose. Any tasks that require non-trivial processing are appended to a single **workitem** queue. When the syncer daemon next awakens (within one second), it services the workitem queue before its normal activities.

3.3 Dependency Structures

We use special-purpose structures for storing dependency information. Each dependency structure contains a type (currently, 1 of 16) and several type-specific values that identify the associated metadata, hold before/after copies of updated metadata values, and cross-reference with related dependency structures. We found it useful to have "organizational" dependency structures for inodes and file blocks and to index them separately from the actual in-memory copies. This simplifies dependency structure management when file cache pages and in-core[6] inode structures are reused. Whenever a directory block or inode is accessed, the code checks for an associated dependency structure and makes certain that any undone updates are reflected in the copy visible to users. Two sets of dependency structures are attached to each organizational structure: those that are simply waiting for the associated metadata to be written to disk and those that support undo/redo on portions of the "owner" that depend on other updates.

3.4 Dependencies

There are four main structural changes requiring sequenced metadata updates: (1) block allocation, (2) block de-allocation, (3) link addition (e.g., file creation), and (4) link removal.[7] The dependencies associated with each are described below together with how we deal with them in our implementation.

Block Allocation

When a new block/fragment is allocated for a file, the new block pointer, whether in the inode or an indirect block,[8] should not be written to stable storage until after the block has been initialized. Also, if the on-disk free space maps are being protected, the free space map from which the block/fragment is allocated must be written to disk before the new pointer is. These

[6]The file system always copies an inode's contents from the inode block (kept in the buffer cache) into an **in-core** (or internal) inode structure before accessing them.

[7]Adding (or removing) links for directories is basically the same as for normal files. The only difference is that, for directories, there are two link additions (or removals) involved – one for the entry in the parent directory and one for the ".." entry in the added/deleted directory.

[8]Large files use blocks of pointers, called **indirect blocks**, to identify their file blocks. This approach is used with one or more levels of indirection, as necessary.

two dependencies are independent of each other and apply to both file blocks and indirect blocks. The block/fragment allocation code remains unchanged, except that delayed writes are utilized and three dependency structures are constructed. The main structure, associated with the inode or indirect block containing the new pointer, holds before and after values for the pointer and the file size (for pointers in the inode only). The other two structures are associated with the new block and the relevant cylinder group block[9], respectively. Each of these structures contains pointers to the other two in order to simplify processing after associated blocks are written to disk.

If an inode or indirect block containing an unsafe pointer (i.e., a pointer with pending dependencies) is written to disk, the in-memory copy (i.e., the source memory for the disk write) of the metadata pointer is overwritten with its old value. The new value is still present in the dependency structure. Also, for file blocks pointed to by the inode, the file size (also present in the inode) is manipulated to protect fragments that have been extended. Because the inode block's contents are out-of-date, all accesses (reads and writes) are inhibited during the disk write. When the disk write completes, the most up-to-date values for the block pointers and file size are restored in the in-memory copy.

When a disk write for the new block or the cylinder group block completes, the associated dependency structure is de-allocated and the cross-reference pointers in the other structures are nullified. After both blocks have been written, the allocation becomes safe and the main structure can be de-allocated.

When extending a file causes a fragment to be upgraded, either to a larger fragment or to a full block, the on-disk location changes if the previous fragment can not simply be extended. In this case, the old fragment must be de-allocated, but not until after the on-disk pointer has been updated (which involves the dependencies above). If the old fragment is newly allocated and has pending dependencies, it and the associated dependency structures can be de-allocated immediately. Otherwise, the old fragment is freed in the background after the new pointer reaches disk (or becomes obsolete, due to de-allocation).

Block De-allocation

De-allocated blocks must not be reused before previous on-disk pointers to them have been reset. We achieve this by not de-allocating the block (i.e., setting the bits in the in-memory free space map) until the reset block pointer reaches stable storage. When block de-allocation is required, the appropriate pointers are nullified (via a delayed write) and the old file size, new file size and old block pointer values are placed in a dependency structure associated with the inode or indirect block. Now-obsolete dependency structures for recent allocations are de-allocated at this point.[10] The block pointers are then reset in the in-memory metadata. After the modified metadata has been written to disk, the dependency structure is added to the workitem queue.

[9]Each cylinder group block contains the portions of the free block and free inode maps corresponding to the elements of its cylinder group.

[10]We do not currently exploit the option of immediately de-allocating blocks whose pointers have not yet been written to disk.

The blocks are freed by the syncer daemon using the same code paths as the original file system. Any dependency structures associated with the de-allocated blocks are considered complete at this point and handled accordingly — this applies only to directory blocks (see below).

Link Addition

When adding a directory entry, the (possibly new) inode, with its incremented link count[11], must be written to disk before the directory entry's pointer to it.[12] Also, if the inode is new and the on-disk free inode maps are being protected, the free inode map from which the inode is allocated must be written to disk before the new pointer. These two dependencies are independent of each other. The link addition code remains unchanged, except that delayed writes are utilized and three (or two) dependency structures are constructed. The main structure, associated with the directory block containing the new entry, holds the offset of the new entry and the new inode pointer value (the old value is NULL). The other two structures are associated with the inode and the relevant cylinder group block (if the inode is new), respectively. Each of these structures contains pointers to the other two in order to simplify processing after associated blocks are written to disk.

If a directory block with an unsafe entry is written to disk, the in-memory copy of the entry's inode pointer is nullified to indicate that the entry is invalid. The new value is still present in the dependency structure. Because the directory block's contents are out-of-date, all accesses (reads and writes) are inhibited during the disk write. After the disk write completes, the correct inode number is restored. However, the directory entry is not updated and marked as dirty immediately after the disk write completes. Rather, we allow the system to reuse the cache block (i.e., VM page), if necessary. When the directory block is next accessed, the directory entry is brought up-to-date. If the block is not accessed within 15 seconds, the syncer daemon updates it and initiates an asynchronous write.

When a disk write for the inode or the cylinder group block completes, the associated dependency structure is de-allocated and the cross-reference pointers in the other structures are nullified. After both blocks have been written (or just the inode, if it was not new), the directory entry is safe and the main structure can be de-allocated.

Link Removal

When removing a directory entry, the on-disk entry's inode pointer must be nullified before the corresponding inode's link count is decremented (possibly freeing the inode for reuse).[13] We

[11] The **link count** indicates the number of directory entries pointing to an inode.

[12] This is not really required for additional links to existing files if the *fsck* utility is run on the file system before using it after a system failure. However, safely using the file system before running the *fsck* utility requires that each on-disk inode's link count be greater than or equal to the number of on-disk directory entries pointing to it.

[13] The actual requirement is that the on-disk inode should not be re-initialized or pointed to by a new on-disk directory entry or free inode map before all previous on-disk directory entry pointers to it have been nullified. Our more stringent requirement simplifies the implementation and protects on-disk link counts for safe post-crash file system use.

achieve this by not decrementing the in-memory inode's link count until after the reset pointer reaches stable storage. So, to remove a link, the in-memory directory entry is nullified via a delayed write. If the directory entry was recently added and is not yet safe, the associated dependency structures are removed and the inode's link count is decremented (the link addition and removal have been serviced with no disk writes!). Otherwise, the inode number is placed in a dependency structure associated with the directory block. After the directory block has been written to disk (or becomes obsolete, due to de-allocation), the dependency structure is added to the workitem queue. The link count is decremented by the syncer daemon and, if it becomes zero, the inode and its blocks are de-allocated using the same code as the original file system (modified as described above).

Rename

File renaming is usually performed in two steps by first adding the new link and then removing the old link. To prevent loss of file identification, the new link must be written to disk before the old link is removed. Our implementation does not currently provide this guarantee, protecting the link addition and removal as described above but not the sequencing between them. We believe that most rename operations can be protected with soft updates by using redo/undo on the removed directory entry and preventing the file system from reusing it. However, it may be necessary to revert to synchronous writes in some cases, such as when the old directory is removed immediately after renaming the file. Detecting and handling all such possibilities may be complex. It is worth noting that the current *ufs* implementation does not provide guarantees for the rename operation.

3.5 Crash Recovery

The modifications described above guarantee that the on-disk copies of inodes, directories, indirect blocks and free space/inode bitmaps are always safe for immediate use after a system failure (other than a media failure). However, the *ufs* file system maintains a number of free block/inode counts (e.g., per cylinder group, per cylinder and per <cylinder, rotational position> pair) in addition to the bitmaps. These counts are used to improve efficiency during allocation and therefore must be consistent with the bitmaps for safe operation. Unfortunately, there is no convenient way to guarantee post-crash consistency of these counts via simple update sequencing, for two main reasons:

1. The counts and bitmaps for a single cylinder group block are not in the same disk sector. If a disk write is interrupted (e.g., by a power failure), one may be written and the other not.

2. Some of the counts are located in the superblock rather than the cylinder group block.

So, when mounting a file system that would require an *fsck* without soft updates, we recompute the auxiliary counts from the bitmaps.

4 Performance comparison

In this section, we compare the performance of three file system implementations. As a baseline (and a goal), we ignore ordering constraints (*No Order*) and use delayed writes for all metadata updates. This baseline has the same performance and lack of reliability as the delayed mount option described in [Ohta90]. It is also very similar to the memory-based file system described in [McKusick90]. The *Conventional* implementation uses synchronous writes to sequence metadata updates. The *Soft Updates* measurements are from our current implementation.

4.1 Experimental Apparatus

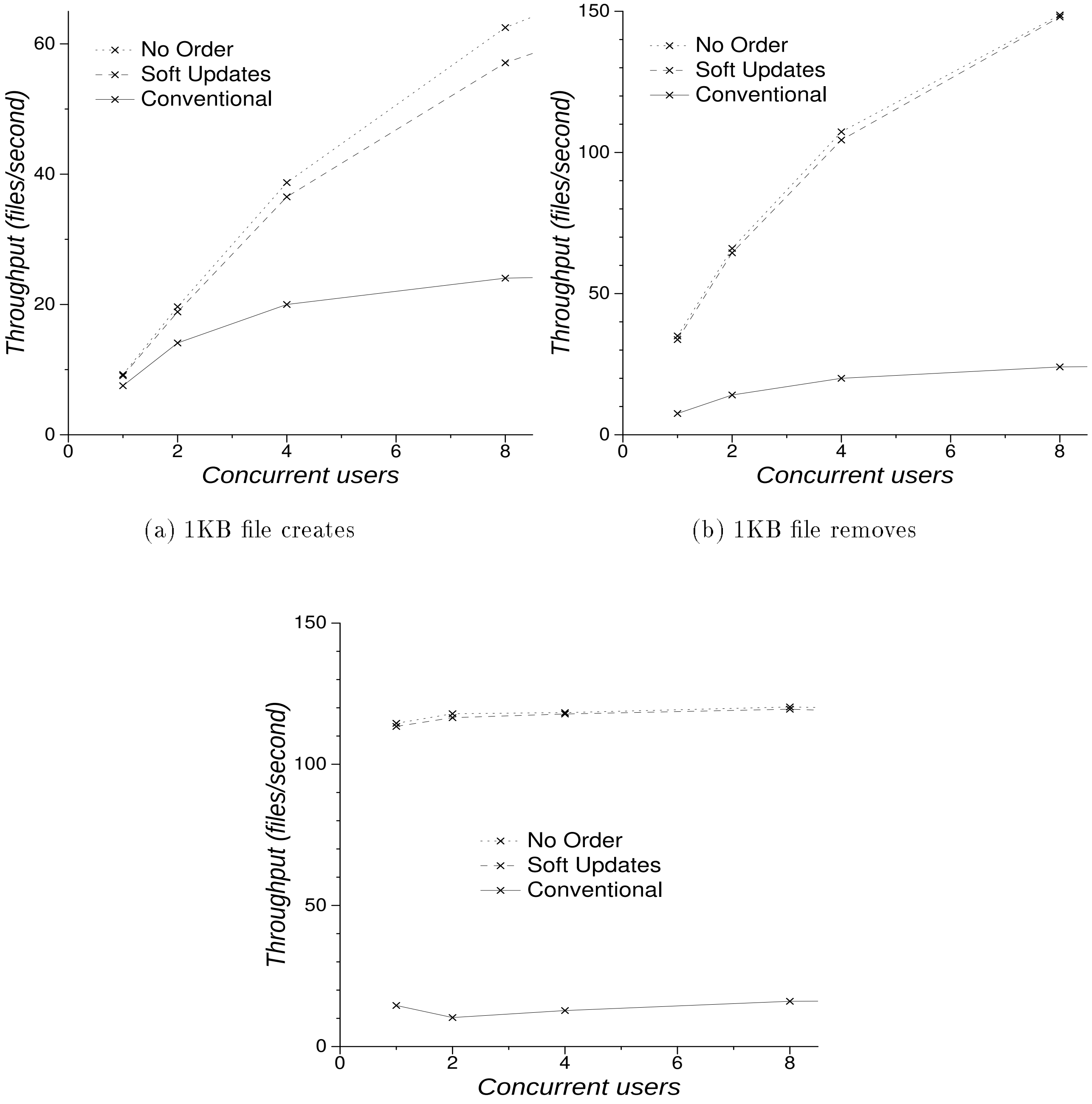
All experiments were performed on an NCR 3433, having a 33MHz Intel 80486 processor and 48 MB of main memory (44 MB for system use and 4 MB for a trace buffer). The HP C2447 disk drive used in the experiments is a high performance, 3.5-inch, 1 GB SCSI storage device [HP92]. The scheduling code in the device driver combines sequential requests, and the disk prefetches sequentially into its on-board cache. Command queueing at the disk is not utilized.

We run all experiments with the network disconnected and with no other non-essential activity. We obtain our measurement data from two sources. The UNIX *time* utility provides total execution times and CPU times. We have also instrumented the device driver to collect I/O traces, including per-request queue and service delays. The traces are collected in the 4 MB trace buffer mentioned above and copied to a separate disk after each experiment. The timing resolution is approximately 840 nanoseconds, and the tracing alters performance by less than 0.01 percent (assuming that the trace buffer could not be otherwise used).

4.2 Metadata Throughput

Figure 3 compares the metadata update throughput supported by the three implementations as a function of the number of concurrent "users." Each "user" works in a separate directory. As a result, create throughput improves with the number of "users" because less CPU time is spent checking the directory contents for name conflicts. Similar behavior is observed for remove throughput. *No Order* and *Soft Updates* both outperform the conventional implementation, and the differences increase with the level of concurrency as performance becomes less CPU bound and more disk bound. The power of using delayed writes for metadata can be seen in figure 3c, where each created file is immediately removed. *No Order* and *Soft Updates* proceed at processor/memory speeds, achieving over 6 times the throughput of *Conventional*, which proceeds at disk speeds.

In most cases, *Soft Updates* performance is within 2 percent of *No Order*. For high create throughput, however, *Soft Updates* performance is up to 10 percent lower. Most of this performance degradation is due to writing modified metadata blocks more than once because of undo/redo dependency handling. Much of this is unnecessary and could be avoided by modifying the syncer daemon to explicitly flush blocks with no dependencies before blocks that depend



(a) 1KB file creates

(b) 1KB file removes

(c) 1KB file create/removes

Figure 3: Metadata update throughput. Each data point (10,000 files split among the "users") is an average of several independent executions. All coefficients of variation are less than 0.05.

Ordering Scheme	Alloc. Init.	Elapsed Time (seconds)	Percent of No Order	CPU Time (seconds)	Disk Requests	I/O Response Time Avg (ms)
Conventional	N	393.4	123.3	70.7	36007	261.2
	Y	720.9	225.9	90.0	51615	127.0
Soft Updates	N	324.3	101.6	68.4	32058	322.0
	Y	314.2	98.5	78.7	31799	232.6
No Order	N	319.1	100.0	67.2	31595	318.1

Table 1: Scheme comparison using 4-user copy. Each datum is an average of several independent executions. The elapsed times are averages among the "users", with coefficients of variation less than 0.05. The CPU times are sums among the "users", with coefficients of variation less than 0.02. The disk system statistics are system-wide, with coefficients of variation less than 0.1.

on them (e.g., newly allocated file pages before inode blocks). With this change, we expect *Soft Updates* create throughput to be much closer to that of *No Order*.

4.3 Metadata Intensive Benchmarks

To illustrate particular performance characteristics of the three implementations, we use two metadata update intensive benchmarks. In the N-user copy benchmark, each "user" concurrently performs a recursive copy of a separate directory tree (535 files totaling 14.3 MB of storage taken from the first author's home directory). In the N-user remove benchmark, each "user" deletes one newly copied directory tree.

Table 1 shows performance data for the 4-user copy benchmark. For *Soft Updates* and *Conventional*, results are shown both for enforcing allocation initialization[14] (i.e., guaranteeing that newly allocated blocks are initialized on disk before on-disk pointers to them are set) and for ignoring it. For *Conventional*, enforcing allocation initialization increases the elapsed time by

[14]For all other measurements, allocation initialization is not enforced for *Conventional*.

Ordering Scheme	Elapsed Time (seconds)	Percent of No Order	CPU Time (seconds)	Disk Requests	I/O Response Time Avg (ms)
Conventional	85.0	1072	12.8	4725	49.7
Soft Updates	6.67	84.1	5.73	436	53.1
No Order	7.93	100.0	7.33	473	60.9

Table 2: Scheme comparison using 4-user remove. Each datum is an average of several independent executions. The elapsed times are averages among the "users", with coefficients of variation less than 0.05. The CPU times are sums among the "users", with coefficients of variation less than 0.02. The disk system statistics are system-wide, with coefficients of variation less than 0.2.

more than 80 percent. For *Soft Updates*, the performance difference is inconsequential.[15] So, while a conventional implementation must trade performance for integrity/security, soft updates can be used to achieve both.

No Order decreases the elapsed time by 23 percent and the number of disk requests by 12 percent (by combining multiple updates to individual file system blocks) when compared to *Conventional* with no allocation initialization. Unlike with the create throughput experiments, which saturate the system with newly created files, *Soft Updates* does not significantly hurt performance relative to *No Order*. In fact, both the elapsed times and the numbers of disk requests differ by less than 2 percent.

The performance differences are more extreme for file removal (table 2), which consists almost entirely of metadata updates. The elapsed times for *No Order* and *Soft Updates* are more than an order of magnitude lower than that of *Conventional*. Note that *Soft Updates* elapsed time is 15 percent lower than that of *No Order* for this benchmark. This is due to the deferred removal approach used by *Soft Updates*, which postpones resource de-allocation activity until after certain disk writes complete. Most of this postponed work occurs after the "user" process completes. The order of magnitude decrease in disk activity (e.g., *Soft Updates* verses *Conventional*) demonstrates the value of using delayed writes for metadata.

[15]Our measurements consistently indicate that enforcing allocation initialization improves performance slightly (approximately 3 percent) with *Soft Updates*, by reducing the number of disk reads. This suggests that the cache hit rate increases. Without additional evidence, we believe that the additional hits are for inode blocks that remain in the cache until allocation dependencies are satisfied.

Ordering Scheme	(1) Create Directories	(2) Copy Files	(3) Read Inodes	(4) Read Files	(5) Compile	Total
Conventional	2.34 (0.48)	3.99 (0.42)	4.13 (0.42)	5.94 (0.28)	285 (0.94)	304 (1.3)
Soft Updates	0.40 (0.49)	2.80 (0.40)	4.11 (0.31)	5.80 (0.40)	273 (0.98)	286 (1.3)
No Order	0.40 (0.49)	2.73 (0.50)	4.12 (0.31)	5.83 (0.39)	271 (0.69)	284 (1.1)

Table 3: Scheme comparison using Andrew benchmark. Each value (in seconds) represents an average of 100 independent executions. The values in parentheses are the standard deviations.

4.4 Andrew Benchmark

Table 3 compares the three implementations using the original Andrew file system benchmark [Howard88], which consists of five phases: (1) create a directory tree, (2) copy the data files, (3) examine the status of every file, (4) read every byte of each file, (5) compile several of the files. As expected, the most significant differences are in the metadata update intensive phases (1 and 2). The read-only phases (3 and 4) are practically indistinguishable. The compute-intensive compile phase improves by about 5 percent with *Soft Updates* and *No Order*. The compile phase dominates the total benchmark time because of aggressive compilation techniques and a slow CPU (by 1995 standards).

4.5 Sdet

Figure 4 compares the three implementations using Sdet, from the SPEC SDM suite of benchmarks. This benchmark [Gaede81, Gaede82] concurrently executes one or more **scripts** of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, file creation and various UNIX utilities). The scripts are generated randomly from a predetermined mix of functions. The reported metric is scripts/hour as a function of the script concurrency. *No Order* outperforms *Conventional* by 20–60 percent, and *Soft Updates* throughput is within 2 percent of *No Order*.

5 Non-performance comparisons

5.1 File System Recovery

With the conventional implementation, the *fsck* utility must be run on a file system before it can be mounted after any system failure. By guaranteeing that the on-disk metadata can always be used safely (except when media corruption destroys live metadata), our soft updates implementation lifts this requirement. A file system can be safely mounted and used immediately after most system failures, but may contain several inconsistencies:

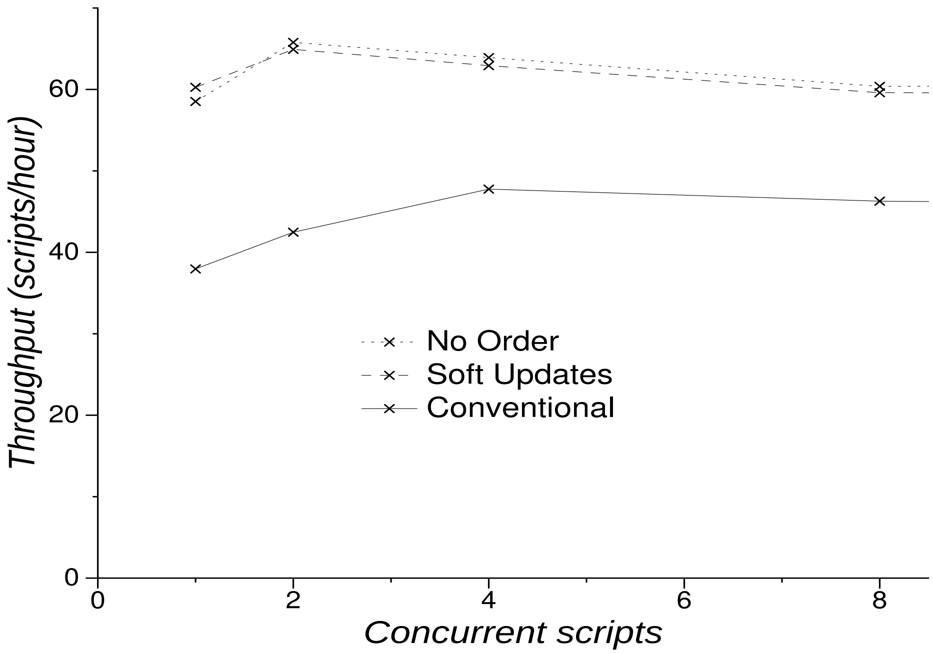


Figure 4: Performance comparison using Sdet. Each data point is an average of several independent executions and all coefficients of variation are less than 0.02.

- Unused blocks may not appear in the free space maps.
- Inode link counts may exceed the number of associated directory entries.
- Unreferenced inodes may not appear in the free inode maps.

One can run the *fsck* utility on the file system, when convenient, to reclaim unreferenced resources and correct link counts.

For the file system used in the performance measurements of section 4, in which about 70 percent of the 850 MB are allocated to files, the *fsck* utility executes in 5–7 minutes. With our soft updates implementation, on the other hand, the file system can be mounted after a system failure in 3–5 seconds. This includes the time necessary to read and modify the superblock, read all cylinder group blocks, recompute the auxiliary free space/inode counts (approximately 1 second of pure compute time) and initialize the various in-memory structures. Most of this time is independent of soft updates.

5.2 File system semantics

The use of synchronous writes to sequence metadata updates does not imply synchronous file system semantics. In general, the last write in a sequence of metadata updates is asynchronous or delayed. In many cases, when a file system call returns control to the caller, there is no guarantee that the change is permanent. For link addition and block allocation, the last update adds the pointer to the directory block, inode or indirect block. So, the requested change is not permanent when the system call returns.[16] For link removal and block de-allocation, however, the last update modifies the free space/inode maps. When the system call returns, the link is permanently removed and/or the blocks have been de-allocated and are available for reuse. With soft updates, neither is true. In particular, de-allocated resources do not become available for reuse until after the re-initialized inode (or indirect block) reaches stable storage.

Some system calls have a flag telling the file system to guarantee that changes are permanent before returning. It may be useful to augment additional file system calls (e.g., link addition) with such a flag in order to support certain applications (e.g., those that require lock files).

5.3 Implementation complexity

Our implementation of soft updates consists of 1800 lines of C code and is restricted to the file system and buffer cache modules. No changes to the on-disk metadata structures are required. Having learned key lessons from an initial implementation, we completed a partial soft updates implementation (described in [Ganger94]) in three weeks. With an additional two weeks, update sequencing for free space/inode maps and fragment extension was added and debugged.

[16] Software locking schemes that use lock files may encounter surprises because of this.

6 Conclusions and Future Work

The use of synchronous writes and the *fsck* utility to protect metadata integrity has been identified as a file system performance and availability problem [Ousterhout90, McVoy91, Seltzer93]. We have described a new mechanism, soft updates, that can be used to achieve memory-based file system performance while providing stronger integrity and security guarantees (e.g., allocation initialization) and higher availability (via shorter recovery times) than most UNIX file systems. This translates into a performance improvement of more than a factor of 2 in many cases (up to a maximum observed difference of a factor of 15).

While our experiments were performed on a UNIX system, the results are applicable to a much wider range of operating environments. Every file system, regardless of the operating system, must address the issue of integrity maintenance. Some (e.g., MPE-XL^{TM}, CMS^{TM}, Windows NT^{TM}) use database techniques such as logging or shadow-paging. Others (e.g., OS/2^{TM}, VMS^{TM}) rely on carefully ordered synchronous writes and could directly use our results.

With soft updates, the *fsck* utility becomes a secondary crash-recovery tool to be used when it is convenient. However, while *fsck* executes, the file system must be made read-only to prevent users from changing the metadata. We believe that the remaining post-crash inconsistencies (mainly unclaimed resources) could be fixed by a background file system process without requiring that the file system be read-only. Such a process's task would be very similar to garbage collection in object-oriented systems, except that it is needed only after a system failure.

Most UNIX file system caches do a poor job of exploiting disk idle time. Workload studies (e.g., [Ousterhout85, McNutt86, Baker91, Ramakrishnan92]) have consistently demonstrated that file system activity is very bursty, consisting of interspersed periods of intense and near-zero usage. UNIX disk activity is also very bursty [Ruemmler93]. Disk idle time could be used by the file system cache to flush dirty blocks to disk. Such an approach is much less likely to suffer from the collisions between user-awaited disk activity and background disk activity that plague most UNIX systems [Carson92]. To provide data hardness guarantees, idle time flushing could be combined with a more conventional time-based approach, such as a syncer daemon or per-block timers [Mogul94].

Because the soft updates mechanism appears so promising, we plan to compare it with other successful methods of protecting metadata integrity, such as non-volatile RAM (NVRAM), logging and shadow-paging. NVRAM can increase data persistence and provide slight performance improvements as compared to soft updates (by reducing syncer daemon activity), but is expensive. Write-ahead logging provides the same protection as soft updates, but must use delayed group commit to achieve the same performance levels, increasing implementation complexity. Using shadow-paging to maintain integrity is difficult to do with delayed writes. Combined with soft updates, however, late binding of disk addresses to logical blocks [Chao92] could provide very high performance. The log-structured file system [Seltzer93] is a special case of shadow-paging that protects integrity by grouping many writes atomically (with a checksum to enforce atomicity). The large writes resulting from log-structuring can better utilize disk bandwidth, but the required cleaning activity reduces performance significantly.

7 Acknowledgements

We thank Jay Lepreau, Kirk McKusick, John Wilkes, and Bruce Worthington for directly helping to improve the quality of this report. Our research group is very fortunate to have the financial and technical support of several industrial partners, including AT&T/GIS, DEC, HaL, Hewlett-Packard, Intel, Motorola, MTI and SES. In particular, AT&T/GIS enabled this research with their extremely generous equipment gifts and by allowing us to generate experimental kernels with their source code. The high performance disk drives used in the experiments were donated by Hewlett-Packard.

References

[Baker91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed File System", *ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.

[Carson92] S. Carson, S. Setia, "Analysis of the Periodic Update Write Policy for Disk Cache", *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, January 1992, pp. 44–54.

[Chamberlin81] D. Chamberlin, M. Astrahan, et. al., "A History and Evaluation of System R", *Communications of the ACM*, Vol. 24, No. 10, 1981, pp. 632–646.

[Chao92] C. Chao, R. English, D. Jacobson, A. Stepanov, J. Wilkes, "Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees", Hewlett-Packard Laboratories Report, HPL-CSP-92-9 rev 1, November 1992.

[Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, "The Episode File System", *Winter USENIX Conference*, January 1992, pp. 43–60.

[Gaede81] S. Gaede, "Tools for Research in Computer Workload Characterization", *Experimental Computer Performance and Evaluation*, 1981, ed. by D. Ferrari and M. Spadoni.

[Gaede82] S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities", *13th International Conference on Management and Performance Evaluation of Computer Systems*, 1982, pp. 62–67.

[Ganger94] G. Ganger, Y. Patt, "Metadata Update Performance in File Systems", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 49–60.

[Gingell87] R. Gingell, J. Moran, W. Shannon, "Virtual Memory Architecture in SunOS", *Summer USENIX Conference*, June 1987, pp. 81–94.

[Hagmann87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *ACM Symposium on Operating Systems Principles*, November 1987, pp. 155–162.

[Howard88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, "Scale and Performance in a Distributed File System", *IEEE Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51–81.

[HP92] Hewlett-Packard Company, "HP C2244/45/46/47 3.5-inch SCSI-2 Disk Drive Technical Reference Manual", Part Number 5960-8346, Edition 3, September 1992.

[Journal92] NCR Corporation, "Journaling File System Administrator Guide, Release 2.00", NCR Document D1-2724-A, April 1992.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.

[McKusick90] M. McKusick, M. Karels, K. Bostic, "A pageable memory based filesystem", *United Kingdom UNIX systems User Group (UKUUG) Summer Conference*, July 1990, pp. 9–13.

[McKusick94] M. McKusick, T.J. Kowalski, "Fsck – The UNIX File System Check Program", *4.4 BSD System Manager's Manual*, O'Reilley & Associates, Inc., Sebastopol, CA, 1994, pp. 3:1–21.

[McNutt86] B. McNutt, "An Empirical Study of Variations in DASD Volume Activity", *Computer Measurement Group (CMG) Conference*, 1986, pp. 274–283.

[McVoy91] L. McVoy, S. Kleiman, "Extent-like Performance from a UNIX File System", *Winter USENIX Conference*, January 1991, pp. 1–11.

[Mogul94] J. Mogul, "A Better Update Policy", *Summer USENIX Conference*, 1994, pp. 99–111.

[Moran87] J. Moran, "SunOS Virtual Memory Implementation", *European UNIX Users Group (EUUG) Conference*, Spring 1988, pp. 285–300.

[Ohta90] M. Ohta, H. Tezuka, "A fast /tmp file system by delay mount option", *Summer USENIX Conference*, June 1990, pp. 145–150.

[Ousterhout85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *ACM Symposium on Operating System Principles*, 1985, pp. 15–24.

[Ousterhout90] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Summer USENIX Conference*, June 1990, pp. 247–256.

[Ramakrishnan92] K. Ramakrishnan, P. Biswas, R. Karelda, "Analysis of File I/O Traces in Commercial Computing Environments", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1992, pp. 78–90.

[Ritchie78] D. Ritchie, K. Thompson, "The UNIX Time-Sharing System", *Bell System Technical Journal*, Vol. 57, No. 6, July/August 1978, pp. 1905–1930.

[Ruemmler93] C. Ruemmler, J. Wilkes, "UNIX Disk Access Patterns", *Winter USENIX Conference*, January 1993, pp. 405–420.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *Winter USENIX Conference*, January 1993, pp. 201–220.

[Ston87] M. Stonebraker, "The Design of the POSTGRES Storage System", *Very Large DataBase Conference*, September 1987, pp. 289–300.

This appendix decribes our experimental implementation of soft updates, assuming basic familiarity with both the report and some instance of the Berkeley Fast File System.

Due to proprietariness constraints with the system software base for our experiments (AT&T/GIS's UNIX SVR4 MP), we are unable to distribute source code directly. We can, however, provide the main sections of our original code and descriptions of what the surrounding code does. Even with this approach, we must (unfortunately) be vague in certain areas to prevent undesired dissemination of AT&T/GIS's trade secrets.

Everything herein is distributed without guarantees of any kind. We also point out that our soft updates implementation was constructed for research purposes and was not originally intended for public consumption.

Questions can be directed to Greg Ganger (ganger@eecs.umich.edu).

1. In A Nutshell

Our soft updates implementation changes almost all of the synchronous and asynchronous metadata updates to delayed writes. Some exceptions are: (1) when the user explicitly requests synchronous updates, as with the fsync() system call or the O_SYNC modifier to the open() system call, (2) when unmounting a file system, and (3) when updating the superblock. One of the procedures described below is called prior to each of the modified updates, and others are used to support these setup procedures. We modified only one of the existing file system structures, appending a field to the "buf" structure:

```
        softdep *b_dep;         /* List of dependencies for this metadata block. */
                                /* The "softdep" structure is defined below. */
```

A production-quality soft updates implementation may require changes to the FSCK utility. For example, our soft updates implementation can result in a zero-length directory (e.g., after a power failure), and the default FSCK utility on our research platform does not fix this problem. The proper repair sequence for a zero-length directory is straight-forward. Other examples may exist.

2. Integration of Virtual Memory and File Caching

The operating system that we work with partially integrates file system caching with the virtual memory system. All file data (including directory blocks and soft links) are cached together with virtual memory pages in a common pool of physical page frames. Other file system metadata (e.g., inode blocks, cylinder group blocks and the superblock) is cached in a separate pool of page frames, commonly referred to as the "buffer cache". Certain aspects of our soft updates implementation exist only because of this distinction. For example, dependency structures for file pages are indexed independent of cache status. For buffer cache blocks, on the other hand, corresponding dependency structures are attached directly to the "buf" structure. Also, file cache pages and buffer cache blocks are indexed/addressed differently. Adding soft updates to a file system implementation that uses only a "buffer cache" may involve different issues.

3. Abstracting the Proprietary System Software

To obscure any proprietary operating system code, we have replaced (in this description) all internal procedure call and structure definitions with generic names. Some common structures (e.g., inode, direct, buf, and fbuf) that are defined in unprotected header files are referred to by name, but the definitions and uses of fields are not provided.

VFS_ID(ip): represents some form of unique VFS identification for the file system to which the in-core inode, "ip", belongs.

INODE_NUMBER(ip): represents the inode number for "ip".

DEVICE_NUMBER(ip): represents the logical device number for "ip".

FILE_SIZE(ip): represents the file size for "ip".

FILE_TYPE(ip): represents the file type for "ip", such as "FILETYPE_DIRECTORY" or "FILETYPE_REGULAR".

FS_ID(ip): represent the address of the file system structure for "ip"'s file system.

DIRECT_BLOCK_POINTER(ip, i): represents the pointer for block #"i" in in-core inode, "ip". There are "NDADDR" direct block pointers in each inode. The same notation is used when "ip" corresponds to an inode block entry.

INDIRECT_BLOCK_POINTER(ip, i): represents the "i"th indirect pointer in in-core inode, "ip". There are "NIADDR" indirect block pointers in each inode. The same notation is used when "ip" corresponds to an inode block entry.

LINK_COUNT(ip): represents the link count for "ip".

INODE_FLAGS(ip): represents the flags for in-core inode, "ip". The one flag manipulated by the soft updates code identifies whether or not the in-core inode has been modified since it was last copied to the in-memory inode block. We will refer to this flag as "INODE_MODIFIED".

BLOCK_SIZE(ip, i): determines the size of the block/fragment pointed to by direct block pointer #"i" (there are no indirect fragments) for "ip".

BLOCK_NUMBER(ip, off): determines the block pointer number containing the byte at offset, "off".

FS_INDIRPTRCNT(fsid); represents the number of pointers in an indirect block for file system, "fsid".

FS_BLOCK_SIZE(fsid): represents the block size for file system, "fsid".

FS_FRAG_SIZE(fsid): represents the fragment size for file system, "fsid".

FS_FRAGSTOBLKS(fsid, fragno); determines the block number containing fragment "fragno", dividing it by the number of fragments per block (actually done with a right shift).

FS_BLKSTOFRAGS(fsid, blkno); determines the fragment number corresponding to block "blkno", multiplying it by the number of fragments per block (actually done with a left shift).

FS_FRAGROUNDUP(fsid, size); rounds "size" upwards to the nearest multiple of the fragment size.

FS_FRAGCNT(fsid, size); determines the number of fragments needed to hold "size" bytes.

FS_SIZEOFBLOCK(fsid, filesize, lbn); determines the size of block #"lbn" in a file of size, "filesize".

FS_INODE_OFFSET(fsid, inum): determines the offset of inode "inum" in the inode block that contains it.

BUF_DEVICE_NUMBER(bp): represents the device for buffer cache block, "bp".

BUF_ADDR_IN_DEVICE(bp): represents the block number for "bp".

BUF_MEMORY_BLOCK(bp): represents the address of the main memory for "bp".

BUF_BLOCK_SIZE(bp): represents the size of the main memory for "bp".

BUF_FLAGS(bp): represents the control flags for "bp". One flag manipulated by the soft updates code marks a block as dirty. We will refer to this flag as "BUFFER_DIRTY". Another flag indicates that the upcoming disk I/O should be asynchronous with respect to the issuing process. We will refer to this flag as "BUFFER_ASYNCIO".

FBUF_MEMORY_BLOCK(fbp): represents the main memory address for the file page managed by fbuf, "fbp".

DIRENTRY_INODE_NUMBER(entryptr): represents the inode number for directory entry, "entryptr".

ptr *KERNEL_MEMORY_ALLOCATE(size): allocate "size" bytes of non-pageable memory. The virtual memory page size is "PAGESIZE".

KERNEL_MEMORY_DEALLOCATE(ptr, size): de-allocate the "size" bytes of non-pageable memory starting at "ptr".

BLOCK_ZERO(ptr, size): initialize "size" bytes, starting at "ptr", with zeros.

BLOCK_COPY(fromptr, toptr, size): copy "size" bytes from "fromptr" to "toptr".

struct inode *UFS_GET_INCORE_INODE(parms): get the inode defined by the parms in an in-core inode structure.

UFS_RETURN_INCORE_INODE(ip): return the in-core inode "ip", allowing others to use it or the in-core slot.

UFS_BLOCK_DEALLOCATE(ip, blkno, size): de-allocate file system block "blkno" of "size" bytes, setting the appropriate freemap entries so it can be reused.

UFS_INODE_DEALLOCATE(ip, inum): de-allocate inode "inum", setting the appropriate free inode bitmap entries so that it can be reused.

struct buf * BREAD(dev, addr, size): get an in-memory copy of block #"addr" of "size" bytes from device "dev".

BWRITE(bp): initiate a disk write for buf, "bp". The write is asynchronous if the BUFFER_ASYNCIO flag is set.

BRELEASE(bp): release "bp" back into the buffer cache.

FBREAD(parms, &fbp): get an in-memory copy of the file page described by the parms. "fbp" should point to the new fbuf.

FBAWRITE(fbp): initiate an asynchronous write for fbuf, "fbp".

LOG_ERROR(errmsg): indicate that an error has occurred. We do this by printing a message on the console.

INDUCE_FAULT(): A significant error has occurred -- drop into the kernel debugger to preserve the state when the error was identified. We usually accomplish this by de-referencing a NULL pointer.

4. General Dependency Structure

We found it convenient to utilize a single general structure to maintain the dependency information that forms the foundation of soft updates. While less efficient in terms of memory utilization, this approach simplifies memory management and structure indexing/cross-referencing. A production-quality implementation should probably use several differently-sized structures to reduce memory utilization. The structure we use contains the equivalent of 16 integers:

```
typedef struct sftdp {
   struct sftdp *hash_next;
   struct sftdp *hash_prev;
   struct sftdp **listtail;
   struct sftdp *next;
   struct sftdp *prev;
   int id;
   int type;
   int arg[9];
} softdep;
```

The first two pointers are used for hash indexing. The next three allow the structure to live on another list and be removed easily at any time. The ‘‘id’’ is used to identify the relative age of two structures -- the one with the lower ‘‘id’’ is older. A single 32-bit integer may be insufficient for active systems that are rebooted infrequently. However, the ‘‘id’’ field is actively used in only one procedure, so combining two integers is a reasonable option. The ‘‘type’’ field identifies the actions to be taken and the meanings of the ‘‘arg’’ array values.

The set of types and the corresponding arg array usages are (note that much of this will not be clear until the remainder of the description is examined):

```
#define SOFTDEP_PAGEDEP    1

        arg[0] = VFS identification for the file system
        arg[1] = inode number for the file
        arg[2] = block pointer number within file
        arg[4] = list of dependencies waiting for file page
        arg[5] = list #0 of update dependencies for file page
        arg[6] = list #1 of update dependencies for file page
        arg[7] = list #2 of update dependencies for file page
        arg[8] = list #3 of update dependencies for file page
```

The "PAGEDEP" structure is used only for organization purposes. It links together the various dependencies related to a particular file page. Update dependencies are hashed (on bits 3 and 4 of the offset into the file page) to determine which of the four lists they are kept in.


```
#define SOFTDEP_INODEDEP    2

        arg[0] = VFS identification for the file system
        arg[1] = inode number
        arg[2] = -1 (indicates inode; see below)
        arg[4] = list of dependencies waiting for "in-core" inode
        arg[5] = list of update dependencies for "in-core" inode
        arg[6] = list of update dependencies for inode block
```

The "INODEDEP" structure is used only for organization purposes. It links together the various dependencies related to a particular inode.

```
#define SOFTDEP_INDIRDEP    3

        arg[0] = logical device number for the file system
        arg[1] = file system block number for indirect block
        arg[2] = memory address for buffer cache block
        arg[3] = list of ALLOCINDIRECT dependencies for indirect block
        arg[4] = memory address for safe copy of indirect block
```

A single "INDIRDEP" structure manages all allocation dependencies for pointers in an indirect block.

```
#define SOFTDEP_ALLOCDIRECT 4

        arg[0] = VFS identification for the file system
        arg[1] = inode number for the file
        arg[2] = block pointer number added
        arg[3] = new value of block pointer
        arg[4] = old value of block pointer
        arg[5] = file size after block allocation
        arg[6] = file size before block allocation
        arg[7] = state of the block pointer (see below)
        arg[8] = address of the corresponding "ALLOCSAFE"
                 (or "BITMAPSAFE_BLOCK") structure
```

An "ALLOCDIRECT" structure is attached to an "INODEDEP" when a new block or fragment is allocated and pointed to by the corresponding inode.

```
#define SOFTDEP_FREEFRAG    5

        arg[0] = logical device number for the file system
        arg[1] = address of the file system structure
        arg[2] = file system block number for the fragment
        arg[3] = size of the file before fragment was replaced
        arg[5] = lbn of the replaced fragment within the file
```

A "FREEFRAG" structure is attached to an "INODEDEP" when a previously allocated fragment is replaced with a larger fragment, rather than extended. The "FREEFRAG" structure is constructed and attached when the allocation becomes safe.

#define SOFTDEP_ALLOCINDIRECT 6

arg[0] = logical device number for the file system
arg[1] = file system block number for the indirect block
arg[2] = offset of pointer in the indirect block
arg[3] = new value of the block pointer (old value is 0)
arg[4] = memory address of the safe copy (see "INDIRDEP")
arg[5] = size of the allocated block
arg[7] = state of the block pointer (see below)
arg[8] = address of the corresponding "ALLOCSAFE"
(or "BITMAPSAFE_BLOCK") structure

An "ALLOCINDIRECT" structure is attached to an "INDIRDEP" when a new block is allocated and pointed to by the corresponding indirect block.

#define SOFTDEP_ALLOCSAFE 7

arg[0] = logical device number for the file system
arg[1] = address of the file system structure
arg[2] = address of the corresponding allocation structure
arg[4] = address of the corresponding "INODEDEP"'s arg[4]
arg[6] = address of the related "BITMAPSAFE_BLOCK" (if pending)

An "ALLOCSAFE" structure is attached to a newly allocated file page or indirect block.

#define SOFTDEP_BITMAPSAFE_BLOCK 8

arg[0] = logical device number for the file system
arg[1] = address of file system structure
arg[2] = address of the corresponding allocation structure
arg[4] = address of the corresponding "INODEDEP"'s arg[4]
arg[6] = address of the related "ALLOCSAFE" (if pending)

A "BITMAPSAFE_BLOCK" structure is attached to a cylinder group block when a block or fragment is allocated from it.

#define SOFTDEP_FREEBLOCKS 9

arg[0] = inode number for the previous owner of the blocks
arg[1] = logical device number for file system
arg[2] = address of the file system structure
arg[3] = previous file size
arg[4] = new file size
arg[5] = used for double-checking number of blocks released
arg[6] = address of the block pointers to be de-allocated (see below)
arg[7] = VFS identification for the file system

A "FREEBLOCKS" structure is attached to an "INODEDEP" when the corresponding file's length is reduced.

#define SOFTDEP_ADD 10

arg[0] = inode number for the directory
arg[1] = state of the new directory entry (see below)
arg[2] = offset of the new directory entry in the directory block
arg[3] = inode number for the new directory entry
arg[4] = block number within directory containing new entry
arg[5] = address of the corresponding "ADDSAFE" (if pending)
arg[6] = address of the corresponding "BITMAPSAFE_INODE" (if pending)
arg[7] = address of the next "ADD" waiting for the "BITMAPSAFE_INODE"

An "ADD" structure is attached to a directory file page when a new directory entry is added.

#define SOFTDEP_ADDSAFE 11

arg[1] = pointer to the corresponding "ADD" structure

An "ADDSAFE" structure is attached to an inode when a new directory entry pointing to it is added.

#define SOFTDEP_BITMAPSAFE_INODE 12

arg[0] = VFS identification for the file system
arg[1] = inode number of the removed directory entry
arg[2] = -2 (indicates BITMAPSAFE_INODE)
arg[7] = pointer to the first "ADD" waiting for this "BITMAPSAFE_INODE"

A "BITMAPSAFE_INODE" structure is attached to a cylinder group block when an inode is allocated from it.

#define SOFTDEP_REMOVE 13

arg[0] = VFS identification for the file system
arg[1] = inode number of the removed directory entry

A "REMOVE" structure is attached to a directory file page when a directory entry is removed.

#define SOFTDEP_FILEFREE 14

arg[0] = VFS identification for the file system
arg[1] = inode number of the unlinked file

A "FILEFREE" structure is attached to an inode when its link count is reduced to zero.

#define SOFTDEP_CYLGROUP 15

arg[0] = logical device number for the file system

arg[1] = file system block number for cylinder group block
arg[2] = -3 (indicates cylinder group block)
arg[4] = cylinder group block size

A "CYLGROUP" structure is attached to the delayed workitem list (see below) when a block or an inode is allocated from the corresponding cylinder group block.

```
#define SOFTDEP_INODEBLOCK      16
```

arg[0] = list of INODEDEPs for contents of the inode block
arg[1] = address of the file system structure
arg[2] = file system block size

An "INODEBLOCK" structure is attached to a cached inode block whenever any of the corresponding inodes have pending allocation dependencies.

5. Locking

There are several critical sections of soft updates code that must occur atomically with respect to other critical section instances. Some form of locking must be used to protect these code regions. On a uniprocessor, temporarily blocking interrupts is generally sufficient. On a multiprocessor, a spin-lock or some similar construct may also be needed. In the code sections below, we use ‘‘ACQUIRE_LOCK’’ and ‘‘FREE_LOCK’’ to represent this requirement.

6. Structure Allocation/De-allocation

For efficiency purposes, memory management for dependency structures is maintained by the soft updates code. As needed, pages of kernel memory are allocated and broken into lists of 16-integer dependency structures. Allocating a new structure consists of taking the first list entry. De-allocation consists of adding a structure to this list.

```
softdep *softdep_free_list = NULL;
int softdep_id_counter = 0;
int softdep_numfreed = 0;  /* for debugging purposes only */

softdep *softdep_allocate()
{
   softdep *new;
   int id;

   ACQUIRE_LOCK;
   new = softdep_free_list;
   if (new == NULL) {
      softdep *temp;
      int i;
      FREE_LOCK;
      temp = KERNEL_MEMORY_ALLCOATE(PAGESIZE);
      for (i=0; i<=((PAGESIZE/sizeof(softdep))-2); i++) {
         temp[i].next = &temp[(i+1)];
      }
      temp[((PAGESIZE/sizeof(softdep))-1)].next = NULL;
      ACQUIRE_LOCK;
      new = temp;
   }
   softdep_free_list = new->next;
   new->id = softdep_id_counter;
   softdep_id_counter++;
   FREE_LOCK;
   return(new);
}

/* the soft updates lock should be held upon entry */
void softdep_deallocate(old)
softdep *old;
{
   old->next = softdep_free_list;
   softdep_free_list = old;
   softdep_numfreed++;
}
```

7. List Management

Active list management is performed by two general functions, using the "next", "prev" and "listtail" values. The "listtail" value must be set appropriately before adding a structure to the list. Each list is doubly-linked and circular with "listtail" pointing to the tail. The two functions append and remove a structure to/from a list, respectively.

```
/* the soft updates lock should be held upon entry */
void softdep_appendtolist(dep)
softdep *dep;
{
   softdep *tail = *dep->listtail;
   if (tail) {
      dep->next = tail->next;
      tail->next = dep;
      dep->prev = tail;
      dep->next->prev = dep;
   } else {
      dep->next = dep;
      dep->prev = dep;
   }
   *dep->listtail = dep;
}
```

```
/* the soft updates lock should be held upon entry */
void softdep_removefromlist(dep)
softdep *dep;
{
   if (dep->listtail) {
      dep->next->prev = dep->prev;
      dep->prev->next = dep->next;
      if (*dep->listtail == dep) {
         *dep->listtail = (dep->prev == dep) ? NULL : dep->prev;
      }
   }
   dep->next = NULL;
   dep->prev = NULL;
   dep->listtail = NULL;
}
```

8. Structure Indexing

We found it very useful to index "PAGEDEP" and "INODEDEP" dependency structures separately from the file blocks and inodes that they correspond to. This helps when the in-memory copy of an inode or file block must be replaced. It also obviates the need to access an inode or file page when simply updating (or de-allocating) dependency structures. "PAGEDEP" and "INODEDEP" dependency structures are identified by three fields: a virtual file system identifier (one of those mounted), an inode number and a block number within the file (-1 for the inode itself). For these structures, the first three "arg" array entries are used for identification. We use a hashing mechanism for indexing, with circular, doubly-linked lists for overflow. The hash value consists of the low bits of the inode number. A better approach might combine bits from both the inode number and the block number.

We use the same mechanism for "BITMAPSAFE_INODE" dependency structures, setting the block number to -2. Indexing of these structures is needed to support protection of newly added links to not yet safe inodes.

```
#define SOFTDEP_HASHBUCKETS   128
#define SOFTDEP_HASHMASK         0x0000007F

softdep *softdep_hash[SOFTDEP_HASHBUCKETS];

/* Executed during system init. before mounting any file systems */
void softdep_initialize()
{
   int i;
   for (i=0; i<SOFTDEP_HASHSIZE; i++) {
      softdep_hash[i] = NULL;
   }
}
```

```
/* the soft updates lock should be held upon entry */
void softdep_appendtohash(dep)
softdep *dep;
{
   dep *tail = softdep_hash[(dep->arg[1] & SOFTDEP_HASHMASK)];
   if (tail) {
      dep->hash_next = tail->hash_next;
      tail->hash_next = softdep;
      dep->hash_prev = tail;
      dep->hash_next->hash_prev = dep;
   } else {
      dep->hash_next = dep;
      dep->hash_prev = dep;
   }
   softdep_hash[(dep->arg[1] & SOFTDEP_HASHMASK)] = dep;
}

/* the soft updates lock should be held upon entry */
void softdep_removefromhash(dep)
softdep *dep;
{
   if ((int) dep->hash_next | (int) dep->hash_prev) {
      dep->hash_next->hash_prev = dep->hash_prev;
      dep->hash_prev->hash_next = dep->hash_next;
      if (softdep_hash[(dep->arg[1] & SOFTDEP_HASHMASK)] == dep) {
         softdep_hash[(dep->arg[1] & SOFTDEP_HASHMASK)] = (dep->hash_prev == dep) ?
NULL : dep->hash_prev;
      }
      dep->hash_next = NULL;
      dep->hash_prev = NULL;
   }
}

/* Return the appropriate softdep or an initialized new one.              */
/* A new structure is passed as a parameter because the soft              */
/* updates lock is held on entry and is not to be released.               */
softdep * softdep_locked_get_softdep(vfsid, inum, lbn, new)
int vfsid;
int inum;
int lbn;
softdep *new;
{
   softdep *tail = softdep_hash[(inum & SOFTDEP_HASHMASK)];
   softdep *tmp = tail;
   if (tmp) {
      do {
         if ((tmp->arg[2] == lbn) && (tmp->arg[1] == inum) && (tmp->arg[0] == vfsid)) {
            softdep_deallocate(new);
            new = NULL;
            break;
         }
```

```
      tmp = tmp->hash_next;
    } while (tmp != tail);
  }
  if (new) {
    tmp = new;
    tmp->type = (lbn == -1) ? SOFTDEP_INODEDEP : SOFTDEP_PAGEDEP;
    tmp->arg[0] = vfsid;
    tmp->arg[1] = inum;
    tmp->arg[2] = lbn;
    softdep_appendtohash(tmp);
  }
  return(tmp);
}

/* Return the softdep for the described page/inode or allocate       */
/* new one.  The soft updates lock should be free on entry and        */
/* is held on exit.                                                    */
softdep * softdep_get_softdep(vfsid, inum, lbn)
int vfsid;
int inum;
int lbn;
{
  softdep *new = softdep_allocate();
  ACQUIRE_LOCK;
  return(softdep_locked_get_softdep(vfsid, inum, lbn, new));
}

/* Return the softdep for the described page if one exists,           */
/* else NULL.  The soft updates lock should be held on entry.          */
softdep * softdep_locked_checkfor_softdep(vfsid, inum, lbn)
int vfsid;
int inum;
int lbn;
{
  softdep *tail = softdep_hash[(inum & SOFTDEP_HASHMASK)];
  softdep *tmp = tail;
  if (tmp) {
    do {
      if ((tmp->arg[2] == lbn) && (tmp->arg[1] == inum) && (tmp->arg[0] == vfsid)) {
        return(tmp);
      }
      tmp = tmp->hash_next;
    } while (tmp != tail);
  }
  return(NULL);
}

/* Return the softdep for the described page/inode if one             */
/* exists, else NULL.  The soft updates lock should be free on        */
/* entry and is held on exit.                                          */
softdep * softdep_checkfor_softdep(vfsid, inum, lbn)
int vfsid;
int inum;
int lbn;
{
  ACQUIRE_LOCK;
  return(softdep_locked_checkfor_softdep(vfsid, inum, lbn));
}
```

9. The workitem Queue

It is sometimes useful and/or necessary to clean up certain dependencies in the background rather than during execution of an application process or interrupt service routine. To realize this, we append dependency structures corresponding to such tasks to a "workitem" queue. In a soft updates implementation, pending workitems should not wait for more than a couple of seconds. In our implementation, pending workitems are serviced by the syncer daemon, which awakens once each second. An alternative would be to have a dedicated file system process that awakens once each second (to amortize the cost) if such activity is needed.

```
softdep *softdep_workitem_pending = NULL;
```

To add a workitem:

```
dep->listtail = &softdep_workitem_pending;
softdep_appendtolist(dep);
```

In the background process:

```
while (softdep_workitem_pending) {
  softdep_handle_workitem();
}

void softdep_handle_workitem()
{
  softdep *work;
  int type, arg0, arg1, arg2;
  ACQUIRE_LOCK;
  if (softdep_workitem_pending == NULL) {
    FREE_LOCK;
    return;
  }
  work = softdep_workitem_pending->next;
  softdep_removefromlist(work);
  type = work->type;
  arg0 = work->arg[0];
  arg1 = work->arg[1];
  arg2 = work->arg[2];
  FREE_LOCK;
  if (type == SOFTDEP_REMOVE) {
```

```
        softdep_handle_workitem_remove(work);
      } else if (type == SOFTDEP_FREEBLOCKS) {
        softdep_handle_workitem_freeblocks(work);
      } else if (type == SOFTDEP_FREEFRAG) {
        softdep_handle_workitem_freefrag(work);
      } else if (type == SOFTDEP_FILEFREE) {
        softdep_handle_workitem_filefree(work);
      } else if (type == SOFTDEP_CYLGROUP) {
        softdep_handle_workitem_cylgroup(work);
      } else if (type == SOFTDEP_PAGEDEP) {
        softdep_handle_workitem_pagedep(arg0, arg1, arg2);
        return;
      } else {
        LOG_ERROR("Unknown softdep_workitem type %d\n", type);
        INDUCE_FAULT();
      }
      ACQUIRE_LOCK;
      softdep_deallocate(work);
      FREE_LOCK;
    }
```

The various "softdep_handle_workitem_*" procedures are described below.

10. Delayed workitems

It is sometimes useful to delay a particular workitem for some period of time, as it might be dealt with during the normal flow of subsequent file accesses. We realize this with a second workitem queue that is manipulated once each second. In our implementation, the syncer daemon executes this code, but the secondary clock interrupt service routine code represents another option.

```
    #define SOFTDEP_MAXDELAY         32
    #define SOFTDEP_MAXDELAYMASK  0x0000001F
    softdep *softdep_workitem_delayed[SOFTDEP_MAXDELAY];
    int softdep_delayno = 0;
```

Executed during system initialization:

```
    for (i=0; i<SOFTDEP_MAXDELAY; i++) {
      softdep_workitem_delayed[i] = NULL;
    }
```

To add a delayed workitem:

```
    dep->listtail = &softdep_workitem_delayed[((softdep_delayno + 15) & SOFTDEP_MAXDE
LAYMASK)];  /* to delay approx. 15 seconds */
    softdep_appendtolist(dep);
```

Executed once each second before normal workitem handling:

```
    if (softdep_workitem_delayed) {
      softdep_handle_workitem_delayed();
    }

    void softdep_handle_workitem_delayed()
    {
      softdep *work;
      ACQUIRE_LOCK;
      softdep_delayno = (softdep_delayno + 1) & SOFTDEP_MAXDELAYMASK;
      while (work = softdep_workitem_delayed[softdep_delayno]) {
        softdep_removefromlist(work);
        work->listtail = &softdep_workitem_pending;
        softdep_appendtolist(work);
      }
      FREE_LOCK;
    }
```

11. Protecting the freemaps (or bitmaps)

In order to eliminate the need to execute fsck before mounting a file system after a power failure, one must (conservatively) guarantee that the on-disk copy of the bitmaps never indicate that a live inode or block is free. So, when a block or inode is allocated, the bitmap should be updated (on disk) before any new pointers. When a block or inode is freed, the bitmap should not be updated until all pointers have been reset. The latter dependency is handled by the delayed de-allocation approach described below for block and inode de-allocation. The former dependency is handled by calling the following procedure when a block or inode is allocated. The resulting "BITMAPDEP" is passed to the routines (described below) that set up related dependencies. Each "BITMAPDEP_INODE" is also inserted into the hash indexing structure so that any additional link additions can be made dependent on the inode allocation.

The ufs file system maintains a number of free block counts (e.g., per cylinder group, per cylinder and per <cylinder, rotational position> pair) in addition to the bitmaps. These counts are used to improve efficiency during allocation and therefore must be consistent with the bitmaps. There is no convenient way to guarantee post-crash consistency of these counts with simple update ordering, for two main reasons: (1) The counts and bitmaps for a single cylinder group block are not in the same disk sector. If a disk write is interrupted (e.g., by power failure), one may be written and the other not. (2) Some of the counts are located in the superblock rather than the cylinder group block. So, we focus our soft updates implementation on protecting the bitmaps. When mounting a file system, we recompute the auxiliary counts from the bitmaps. Our code is not provided in this description, because it is so specific to the base system software. However, it is straight-forward and consists of less than 100 lines of C code.

```
    softdep *softdep_completed_bitmapdeps = NULL;
```

```
/* Called just after updating the cylinder group block to          */
/* allocate an inode or block (or frag).                            */
softdep * softdep_setup_bitmapdep(bp, ip, allocval)
struct buf *bp;        /* bp for the freemap information (cylinder group block) */
struct inode *ip;      /* inode related to allocation, if inode alloc */
                       /* otherwise, NULL */
int allocval;          /* the inode number allocated, or 0 if block alloc */
{
  softdep *bitmapdep = softdep_allocate();
  softdep *cylgroupdep = softdep_get_softdep((int) BUF_DEVICE_NUMBER(bp), (int) BU
F_ADDR_IN_DEVICE(bp), -3);
  /* If a "CYLGROUP" structure for this cylinder group does not yet */
  /* exist, construct one and add it to the delayed workitem list. */
  /* The goal here is to clear bitmap dependencies before the inode */
  /* is written to disk, to avoid writing it a second time. */
  if (cylgroupdep->type != SOFTDEP_CYLGROUP) {
    cylgroupdep->type = SOFTDEP_CYLGROUP;
    cylgroupdep->arg[4] = BUF_BLOCK_SIZE(bp);
    cylgroupdep->listtail = &softdep_workitem_delayed[((softdep_delayno + 10) & SOFTD
EP_MAXDELAYMASK)];
    softdep_appendtolist(cylgroupdep);
  }
  if (ip) {
    bitmapdep->type = SOFTDEP_BITMAPSAFE_INODE;
    bitmapdep->arg[0] = VFS_ID(ip);
    bitmapdep->arg[1] = allocval;
    bitmapdep->arg[2] = -2;      /* indicates BITMAPSAFE_INODE */
    bitmapdep->arg[7] = 0;
    softdep_appendtohash(bitmapdep);
  } else {
    bitmapdep->type = SOFTDEP_BITMAPSAFE_BLOCK;
    bitmapdep->arg[2] = 0;
  }
  bitmapdep->listtail = &bp->b_dep;
  softdep_appendtolist(bitmapdep);
  FREE_LOCK
  return(bitmapdep);
}

/* Called when, for whatever reason (usually a spurious error),      */
/* an allocated inode or block is freed rather than used.           */
void softdep_toss_bitmapdep(bitmapdep)
softdep *bitmapdep;  /* bitmapdep related to the resource allocation */
{
  ACQUIRE_LOCK
  softdep_removefromlist(bitmapdep);
  if (bitmapdep->type == SOFTDEP_BITMAPSAFE_INODE) {
    softdep_removefromhash(bitmapdep);
  }
  softdep_deallocate(bitmapdep);
  FREE_LOCK
}

/* this workitem routine simply initiates an asynchronous write     */
/* for the corresponding cylinder group block, if it is dirty.      */
void softdep_handle_workitem_cylgroup(work)
softdep *work;
{
  struct buf *bp;
  ACQUIRE_LOCK
  softdep_removefromhash(work);
  FREE_LOCK
  bp = BREAD(work->arg[0], work->arg[1], work->arg[4]);
  if (BUF_FLAGS(bp) & BUFFER_DIRTY) {
    BUF_FLAGS(bp) |= BUFFER_ASYNCIO;
    BWRITE(bp);
  } else {
    BRELEASE(bp);
  }
}
```

12. Direct block allocation dependencies

When a new block is allocated, the corresponding disk locations must be initialized (with zeros or new data) before the on-disk inode points to them. Also, the freemap from which the block was allocated must be updated (on disk) before the inode’s pointer. These two dependencies are independent of each other and are needed for all file blocks and indirect blocks that are pointed to directly by the inode. Just before the "in-core" version of the inode is updated with a newly allocated block number, a procedure (below) is called to setup allocation dependency structures. These structures are removed when the corresponding dependencies are satisfied or when the block allocation becomes obsolete (i.e., the file is deleted, the block is de-allocated, or the block is a fragment that gets upgraded). All of these cases are handled in procedures described later.

When a file extension causes a fragment to be upgraded, either to a larger fragment or to a full block, the on-disk location may change (if the previous fragment could not simply be extended). In this case, the old fragment must be de-allocated, but not until after the inode’s pointer has been reset. In most cases, this is handled by later procedures, which will construct a "FREEFRAG" structure to be added to the workitem queue when the inode update is complete (or obsolete). The main exception to this is when an allocation occurs while a pending allocation dependency (for the same block pointer) remains. This case is handled in the main allocation dependency setup procedure by immediately freeing the unreferenced fragments.

Block allocation dependencies are handled with undo/redo on the in-memory copy of the inode block. A particular block pointer dependency can be in any of three states:

```
#define SOFTDEP_STATE_ATTACHED        1
#define SOFTDEP_STATE_UNDONE          2
#define SOFTDEP_STATE_COMPLETE        3
```

"ATTACHED" indicates that the in-memory copy is up-to-date and the on-disk copy is out-of-date. "UNDONE" indicates that both copies are out-of-date (the dependency structure still has the necessary information). For direct block allocation, "in-memory copy" refers to the cached inode block rather than the "in-core" inode structure. "COMPLETE" indicates that the in-memory copy may be out-of-date, but that the on-disk copy is up-to-date.

```
void softdep_setup_allocdirect(ip, lbn, newblkno, oldblkno, newsize, oldsize, bp)
struct inode *ip;            /* inode to which block is being added */
int lbn;                     /* block pointer within inode */
int newblkno;                /* disk block number being added */
int oldblkno;                /* previous value of pointer, 0 unless fragment */
int newsize;                 /* new size of file */
int oldsize;                 /* old size of file */
struct buf *bp;              /* bp for indirect block, if not a file page */
softdep *bitmapdep;          /* from corresponding freemap update */
{
   softdep *pdep = softdep_allocate();
   softdep *idep = softdep_allocate();
   softdep *pagedep;
   softdep *inodedep;
   softdep *extra;
   softdep *tmp;
   int freeblkno = -1;
   int freesize;
   softdep *tail = NULL;

   if (lbn < NDADDR) {      /* NDADDR = # of direct block pointers in inode */
      extra = softdep_allocate();
   }
   idep->type = SOFTDEP_ALLOCDIRECT;
   idep->arg[0] = VFS_ID(ip);
   idep->arg[1] = INODE_NUMBER(ip);
   idep->arg[2] = lbn;
   idep->arg[3] = newblkno;
   idep->arg[4] = oldblkno;
   idep->arg[5] = newsize;
   idep->arg[6] = oldsize;
   idep->arg[7] = SOFTDEP_STATE_UNDONE;
   idep->arg[8] = (int) pdep;     /* cross-reference to corresponding softdep */
   pdep->type = SOFTDEP_ALLOCSAFE;
   pdep->arg[0] = DEVICE_NUMBER(ip);
   pdep->arg[1] = FS_ID(ip);
   pdep->arg[2] = (int) idep;     /* cross-reference to corresponding softdep */
   bitmapdep->arg[0] = DEVICE_NUMBER(ip);
   bitmapdep->arg[1] = FS_ID(ip);
   inodedep = softdep_get_softdep(VFS_ID(ip), INODE_NUMBER(ip), -1);
   idep->listtail = (softdep **) &inodedep->arg[5];
   pdep->arg[4] = (int) &inodedep->arg[4];
   if (bitmapdep->listtail == &softdep_completed_bitmapdeps) {
      softdep_removefromlist(bitmapdep);
      softdep_deallocate(bitmapdep);
      pdep->arg[6] = 0;
   } else {
      bitmapdep->arg[2] = (int) idep;
      bitmapdep->arg[4] = (int) &inodedep->arg[4];
      bitmapdep->arg[6] = (int) pdep;  /* ALLOCSAFE and BITMAPDEP */
      pdep->arg[6] = (int) bitmapdep;  /* are cross-referenced and */
                                       /* treated identically */
   }
   if (lbn >= NDADDR) {                /* If allocating an indirect block */
      pdep->listtail = &bp->b_dep;
   } else {
      pagedep = softdep_locked_get_softdep(VFS_ID(ip), INODE_NUMBER(ip), lbn, extra);
      pdep->listtail = (softdep **) &pagedep->arg[4];
      if (tmp = (softdep *) inodedep->arg[5]) {
allocdirect_check:
         tail = tmp;
         do {
            if (tmp->arg[2] == lbn) {
                        /* if pending fragment allocation for same lbn */
               softdep_removefromlist(tmp);
               if ((idep->arg[4] != tmp->arg[3]) || (lbn >= NDADDR)) {
                  LOG_ERROR("Non-matching new and old blknos in softdep_setup_allocdirect: 
%d != %d\n", idep->arg[4], tmp->arg[3]);
                  INDUCE_FAULT();
               }
               if (tmp->arg[8]) {
                  softdep_removefromlist(tmp->arg[8]);
                  if (((softdep *)tmp->arg[8])->arg[6]) {
                     softdep_removefromlist(((softdep *)tmp->arg[8])->arg[6]);
                     softdep_deallocate(((softdep *)tmp->arg[8])->arg[6]);
                  }
                  softdep_deallocate(tmp->arg[8]);
               }
               if (tmp->arg[7] != SOFTDEP_STATE_COMPLETE) {
                  idep->arg[4] = tmp->arg[4];
                  idep->arg[6] = tmp->arg[6];
                  if (newblkno != tmp->arg[3]) {
                     int tmpsize = FS_BLOCK_SIZE(FS_ID(ip)) - ((tmp->arg[3] - FS_BLKSTOFR
AGS(FS_ID(ip), FS_FRAGSTOBLKS(FS_ID(ip), tmp->arg[3]))) * FS_FRAG_SIZE(FS_ID(ip)));
                     freeblkno = tmp->arg[3];
                     freesize = FS_FRAGROUNDUP(FS_ID(ip), FS_SIZEOFBLOCK(FS_ID(ip), t
mp->arg[5], lbn));
                     freesize = (freesize > tmpsize) ? tmpsize : freesize;
                     if (tmp->arg[3] == tmp->arg[4]) {
                        /* remove only the extended part of the previous frag */
```

```
                        tmpsize = FS_FRAGROUNDUP(FS_ID(ip), FS_SIZEOFBLOCK(FS_ID(ip),
tmp->arg[6], lbn));
                        freeblkno += FS_FRAGCNT(FS_ID(ip), tmpsize);
                        freesize -= tmpsize;
                     }
                  }
               }
               softdep_deallocate(tmp);
               tail = (softdep *) -1;
               break;
            }
            tmp = tmp->next;
         } while (tmp != tail);
      }
      if ((tail != (softdep *) -1) && (inodedep->arg[6]) && (inodedep->arg[6] != (int) tail)) {
         tmp = (softdep *) inodedep->arg[6];
         goto allocdirect_check;
      }
   }
   softdep_appendtolist(idep);
   softdep_appendtolist(pdep);
   FREE_LOCK;
   if (freeblkno != -1) {
      UFS_BLOCK_DEALLOCATE(ip, freeblkno, freesize);
   }
}

/* this workitem, which is constructed in later procedures,          */
/* de-allocates fragments that were replaced during file block       */
/* allocation.                                                       */
void softdep_handle_workitem_freefrag(work)
softdep *work;
{
   struct inode tip;
   int freesize = FS_FRAGROUNDUP(work->arg[1], FS_SIZEOFBLOCK(work->arg[1], wo
rk->arg[3], work->arg[5]));
   DEVICE_NUMBER(&tip) = work->arg[0];
   FS_ID(&tip) = work->arg[1];
   UFS_BLOCK_DEALLOCATE(&tip, work->arg[2], freesize);
}
```

13. Indirect block allocation dependencies

The same dependencies exist when a new block is allocated and pointed to by an entry in a block of indirect pointers. The undo/redo states described above (#12) are also used here. Because an indirect block contains many pointers that may have dependencies, a second copy of the entire in-memory indirect block is kept. The buffer cache copy is always completely up-to-date. The second copy, which is used only as a source for disk writes, contains only the safe pointers (i.e., those that have no remaining update dependencies). The second copy is freed when all pointers are safe. Unlike with file pages, we do not allow the cache to replace indirect blocks with pending update dependencies. Three procedures are used when setting up indirect block pointer dependencies. The division is necessary because of the organization of the "bmap" routine and because of the distinction between file pages and other file system blocks.

```
/* Called just before setting an indirect block pointer to a        */
/* newly allocated file page.                                        */
softdep * softdep_setup_allocindir_phase1(ip, lbn)
struct inode *ip;           /* inode for file being extended */
int lbn;                    /* allocated block number within file */
{
   softdep *bdep = softdep_allocate();
   softdep *pdep = softdep_allocate();
   softdep *pagedep;

   pdep->type = SOFTDEP_ALLOCSAFE;
   pdep->arg[2] = (int) bdep;
   pdep->arg[6] = 0;
   bdep->type = SOFTDEP_ALLOCINDIRECT;
   bdep->arg[7] = SOFTDEP_STATE_UNDONE;
   bdep->arg[8] = (int) pdep;
   pagedep = softdep_get_softdep(VFS_ID(ip), INODE_NUMBER(ip), lbn);
   pdep->listtail = (softdep **) &pagedep->arg[4];
   softdep_appendtolist(pdep);
   FREE_LOCK;
   return(bdep);
}

/* Called just before setting an indirect block pointer to a        */
/* newly allocated indirect block.                                   */
softdep * softdep_setup_allocindir_phase1a(bp)
struct buf *bp;             /* newly allocated indirect block */
{
   softdep *bdep = softdep_allocate();
   softdep *pdep = softdep_allocate();
   softdep *pagedep;

   pdep->type = SOFTDEP_ALLOCSAFE;
   pdep->arg[2] = (int) bdep;
   pdep->arg[6] = 0;
   bdep->type = SOFTDEP_ALLOCINDIRECT;
   bdep->arg[7] = SOFTDEP_STATE_UNDONE;
   bdep->arg[8] = (int) pdep;
   pdep->listtail = &bp->b_dep;
   ACQUIRE_LOCK;
   softdep_appendtolist(pdep);
   FREE_LOCK;
   return(bdep);
}
```

```
/* called after setting new indirect block pointer but before          */
/* freeing the bp representing the indirect block itself (same         */
/* procedure for both new file pages and new indirect blocks).         */
/* "dep" should be the value returned by the phase1[a] routine         */
/* called previously.                                                  */
void softdep_setup_allocindir_phase2(bp, lbn, blkno, bsize, dep)
struct buf *bp;         /* bp for the in-memory copy of the indirect block */
int ptrno;              /* offset of pointer in indirect block */
int blkno;              /* new value for pointer (old value is 0) */
int bsize;              /* new size of pointed to block (old value is 0) */
softdep *dep;           /* the dep structure returned by a phase1[a] routine */
softdep *bitmapdep; /* from the corresponding bitmap update */
{
  softdep *new = NULL;
  softdep *tmp;
  softdep *indirdep = NULL;

  dep->arg[0] = BUF_DEVICE_NUMBER(bp);   /* These two values uniquely identify  */
  dep->arg[1] = BUF_ADDR_IN_DEVICE(bp); /* a device block as addressed by a bp */
  dep->arg[2] = ptrno;
  dep->arg[3] = blkno;
  dep->arg[5] = bsize;
  while (1) {
    ACQUIRE_LOCK;
    if (tmp = bp->b_dep) {
      do {
        if (tmp->type == SOFTDEP_INDIRDEP) {
          indirdep = tmp;
          break;
        }
        tmp = tmp->next;
      } while (tmp != bp->b_dep);
    }
    if ((indirdep == NULL) && (new)) {
      indirdep = new;
      softdep_appendtolist(indirdep);
      new = NULL;
    }
    if (indirdep) {
      if (bitmapdep->lasttail == &softdep_completed_bitmapdeps) {
        softdep_removefromlist(bitmapdep);
        softdep_deallocate(bitmapdep);
        if (dep->arg[7] == SOFTDEP_STATE_COMPLETE) {
          softdep_deallocate(dep);
          break;
        }
      } else {
        bitmapdep->arg[2] = (int) dep;
        bitmapdep->arg[6] = dep->arg[8];
        if (bitmapdep->arg[6]) {
          ((softdep *)bitmapdep->arg[6])->arg[6] = (int) bitmapdep;
        } else {
          dep->arg[8] = (int) bitmapdep;
          dep->arg[7] = SOFTDEP_STATE_UNDONE;
        }
      }
      dep->arg[4] = indirdep->arg[4];
      dep->listtail = (softdep **) &indirdep->arg[3];
      softdep_appendtolist(dep);
      ((int *) indirdep->arg[4])[(dep->arg[2])] = 0;
    }
    if (new) {
      tmp = (softdep *) new->arg[4];
      softdep_deallocate(new);
      new = tmp;
    }
    FREE_LOCK;
    if (indirdep) {
      break;
    }
    new = softdep_allocate();
    new->type = SOFTDEP_INDIRDEP;
    new->arg[0] = BUF_DEVICE_NUMBER(bp);
    new->arg[1] = softdep->arg[1];   /* address  on device */
    new->arg[2] = BUF_MEMORY_BLOCK(bp);
    new->arg[3] = NULL;
    new->arg[4] = KERNEL_MEMORY_ALLOCATE(BUF_BLOCK_SIZE(bp));
    new->listtail = &bp->b_dep;
    BLOCKCOPY(/* from */ new->arg[2], /* to */ new->arg[4], BUF_BLOCK_SIZE(bp));
  }
  if (new) {
    KERNEL_MEMORY_DEALLOCATE(new, BUF_BLOCK_SIZE(bp));
  }
}
```

14. Block de-allocation dependencies

When blocks are de-allocated, the on-disk pointers must be nullified before the blocks are made available for use by other files. (The true requirement is that old pointers must be nullified before new on-disk pointers are set. We chose this slightly more stringent requirement to reduce complexity.) Our implementation handles this dependency by updating the inode (or indirect block) appropriately but delaying the actual block de-allocation (i.e., freemap and free space count manipulation) until after the updated versions reach stable storage. After the disk is updated, the blocks can be safely de-allocated whenever it is convenient. Our experimental soft updates implementation (and therefore the code below) handles only the common case of reducing a file’s length to zero. Other cases are handled by the conventional synchronous write approach. The extension to partial file truncation should be straight-forward.

An alternative version of the dependency structure is used to maintain copies of the block numbers to be freed:

```
typedef struct {
   int extend[16];
} softdep_extension;

/* This routine should be called from the routine that shortens        */
/* a file's length, before the inode's size or block pointers          */
/* are modified.                                                       */
int * softdep_setup_freeblocks_phase1(ip, length)
struct inode *ip;          /* The inode whose length is to be reduced */
int length;                /* The new length for the file */
{
   softdep *new = softdep_allocate();
   softdep_extension *freedep = (softdep_extension *) softdep_allocate();
   int i;

   new->type = GANGER_SOFTDEP_FREEBLOCKS;
   new->listtail = NULL;
   new->arg[0] = INODE_NUMBER(ip);
   new->arg[1] = DEVICE_NUMBER(ip);
   new->arg[2] = FS_ID(ip);
   new->arg[3] = FILE_SIZE(ip);
   new->arg[4] = length;
   new->arg[7] = VFS_ID(ip);
   for (i=0; i<NDADDR; i++) {        /* copy the direct block pointers */
      freedep->extend[i] = DIRECT_BLOCK_POINTER(ip, i);
   }
   for (i=0; i<NIADDR; i++) {        /* copy the indirect block pointers */
      freedep->extend[(i + NDADDR)] = INDIRECT_BLOCK_POINTER(ip, i);
   }
   new->arg[6] = (int) freedep;
   return((int *) new);
}

/* This routine should be called immediately after the in-memory            */
/* inode block has been updated, but before it is released (this            */
/* required modification of the inode block update routine) and        */
/* before any file blocks are actually de-allocated back to the        */
/* file system.  After this procedure the file length reduction        */
/* routine should release the in-memory inode block and return         */
/* to its caller.  The blocks will be de-allocated back to the         */
/* file system by subsequent soft updates code.  Getting the           */
/* third parameter here required adding a parameter to the file        */
/* truncation routine.                                                 */
void softdep_setup_freeblocks_phase2(ip, dep, del)
struct inode *ip;          /* The inode whose length has been reduced */
softdep *dep;              /* The value returned by phase1 */
int del;                   /* Is the inode being cleaned for de-allocation? */
{
   softdep *inodedep;
   softdep *idep;
   softdep *tmp;

   /* The ufs implementation with which we worked double-checks */
   /* the state of the block pointers and file size as it reduces */
   /* a file's length.  Some of this code is replicated here in */
   /* our soft updates implementation.  "arg[5]" of "dep" is also */
   /* used to transfer a part of this information to the procedure */
   /* that eventually de-allocates the blocks. */

   if (del) {
      /* This sets up the inode de-allocation dependency described */
      /* below (#17). */
      softdep *tmp = softdep_allocate();
      tmp->type = SOFTDEP_FILEFREE;
      tmp->arg[0] = VFS_ID(ip);
      tmp->arg[1] = INODE_NUMBER(ip);
      tmp->arg[2] = DEVICE_NUMBER(ip);
      tmp->arg[3] = FS_ID(ip);
   }
   inodedep = softdep_get_softdep(VFS_ID(ip), INODE_NUMBER(ip), -1);
   while ((idep = (softdep *) inodedep->arg[5]) || (idep = (softdep *) inodedep->arg[6])) {
      /* because the file length has been truncated to zero, any */
      /* pending block allocation dependency structures associated */
      /* with this inode are obsolete and can simply be de-allocated. */
      /* The corresponding "ALLOCSAFE" and "BITMAPDEP" structures */
      /* are also de-allocated. */
      softdep_removefromlist(idep);
      if (idep->type == SOFTDEP_ALLOCDIRECT) {
         if (idep->arg[8]) {
            softdep_removefromlist((softdep *) idep->arg[8]);
            if (((softdep *)idep->arg[8])->arg[6]) {
               softdep_removefromlist(((softdep *)idep->arg[8])->arg[6]);
               softdep_deallocate(((softdep *)idep->arg[8])->arg[6]);
            }
            softdep_deallocate((softdep *) idep->arg[8]);
         }
         if ((idep->arg[4] != 0) && (idep->arg[3] != idep->arg[4]) && (idep->arg[2] < NDADD
R) && (idep->arg[7] != SOFTDEP_STATE_COMPLETE)) {
            /* set up a FREEFRAG structure for the fragment that was */
            /* clobbered by the corresponding allocation */
            idep->type = SOFTDEP_FREEFRAG;
            idep->arg[0] = DEVICE_NUMBER(ip);
            idep->arg[1] = FS_ID(ip);
            idep->arg[5] = idep->arg[2];
            idep->arg[2] = idep->arg[4];
            idep->arg[3] = idep->arg[6];
            idep->listtail = (softdep **) &inodedep->arg[4];
            softdep_appendtolist(idep);
```

```
        } else {
          softdep_deallocate(idep);
        }
      } else {
        LOG_ERROR("Unknown inodedep arg5 type in softdep_setup_freeblocks: %d\n", idep
->type);
        INDUCE_FAULT();
      }
    }
    dep->listtail = (softdep **) &inodedep->arg[4];
    softdep_appendtolist(dep);
    if (del) {
      softdep *pagedep;
      int i;
      tmp->listtail = (softdep **) &inodedep->arg[4];
      softdep_appendtolist(tmp);
      for (i=0; i<=min(NDADDR, BLOCK_NUMBER(ip, FILE_SIZE(ip))); i++) {
        pagedep = softdep_locked_checkfor_softdep(inodedep->arg[0], INODE_NUMBER(ip),
 i);
        if (pagedep) {
          /* Skip any forthcoming workitem processing of soon-to-be */
          /* de-allocated pagedeps.  This is an ugly hack and should */
          /* be replaced with a flag that says "soon-to-be-de-allocated" */
          /* and an indication of the "FREEBLOCK"s ‘‘id’’. */
          softdep_removefromlist(pagedep);
          if ((pagedep->arg[4] | pagedep->arg[5] | pagedep->arg[6] | pagedep->arg[7] | pagedep-
>arg[8]) == 0) {
            /* De-allocate the no-longer useful pagedep.  This cuts */
            /* down significantly on the number of live pagedeps when */
            /* there are many short-lived files. */
            softdep_removefromhash(pagedep);
            softdep_deallocate(pagedep);
          }
        }
      }
    }
    FREE_LOCK;
  }

  /* This workitem routine performs the block de-allocation.            */
  /* The workitem is added to the pending list after the updated        */
  /* inode block has been written to disk.  As mentioned above,         */
  /* checks regarding the number of blocks de-allocated (compared       */
  /* to the number of blocks allocated for the file) are also           */
  /* performed in this function.  The specific code has been            */
  /* removed from below.                                                */
  void softdep_handle_workitem_freeblocks(work)
  softdep *work;
  {
    softdep_extension *freedep;
    struct inode tip;
    daddr_t bn;
    int i;
    int baselbns[NIADDR];
    int tmpval = 1;

    baselbns[0] = NDADDR;
    for (i=1; i<NIADDR; i++) {
      tmpval *= FS_INDIRPTRCNT(work->arg[2]);
      baselbns[i] = baselbns[i-1] + tmpval;
    }
    freedep = (softdep_extension *) work->arg[6];
    INODE_NUMBER(&tip) = work->arg[0];
    DEVICE_NUMBER(&tip) = work->arg[1];
    FS_ID(&tip) = work->arg[2];
    VFS_ID(&tip) = work->arg[7];
    FILE_SIZE(&tip) = work->arg[3];
    for (i = (NIADDR-1); i >= 0; i--) {
      bn = freedep->extend[(i + NDADDR)];
      if (bn) {
        UFS_INDIRTRUNC(&tip, bn, i, baselbns[i], work->arg[7], work->id);
        /* UFS_INDIRTRUNC is also modified (see below).  The last */
        /* three parameters were added to support the changes. */
        softdep_deallocate_indirectblock(bp);
        UFS_BLOCK_DEALLOCATE(&tip, bn, FS_BLOCK_SIZE(work->arg[2]));
      }
    }
    for (i = (NDADDR-1); i >= 0; i--) {
      bn = freedep->extend[i];
      if (bn) {
        softdep_deallocate_filepage(&tip, VFS_ID(&tip), INODE_NUMBER(&tip), i, work->i
d);
        UFS_BLOCK_DEALLOCATE(&tip, bn, BLOCK_SIZE(&tip, i));
      }
    }
    softdep_deallocate(freedep);
  }

  /* while we cannot provide the code present in this function,         */
  /* we can indicate the changes that we made.                          */
  UFS_INDIRTRUNC(ip, bn, level, baselbn, vfsp, id)
  {
    ...
    int lbnadd = FS_INDIRPTRCNT(FS_ID(ip));

    for (i=(level-1); i>0; i--) {
      lbnadd *= FS_INDIRPTRCNT(FS_ID(ip));
    }

    /* setup and scanning through indirect block, "i" is the counting
    /* variable for looping through the pointers. */
```

```
    /* each recursive call to UFS_INDIRTRUNC should look something like */
    /* UFS_INDIRTRUNC(ip, BLKNO, (level-1), (baselbn + (i*lbnadd)), vfsp, id); */

    /* Before each call to UFS_BLOCK_DEALLOCATE, the appropriate one */
    /* of the following procedures should be called: */
    /* softdep_deallocate_indirectblock(bp);  */
    /* softdep_deallocate_filepage(ip, vfsp, INODE_NUMBER(ip), (baselbn + i), id); */
}

/* This routine is called before an indirect block is de-          */
/* allocated, but after all of the pointed to blocks have          */
/* been de-allocated.  It simply de-allocates any associated       */
/* dependency structures.                                          */
void softdep_deallocate_indirectblock(bp)
struct buf *bp;             /* The indirect block to be freed */
{
    softdep *bdep;
    softdep *indirdep;
    int *indirdepmem = NULL;

    ACQUIRE_LOCK;
    while (bdep = bp->b_dep) {
        softdep_removefromlist(bdep);
        if (bdep->type == SOFTDEP_INDIRDEP) {
            indirdepmem = (int *) bdep->arg[4];
            indirdep = bdep;
            while (bdep = (softdep *) indirdep->arg[3]) {
                if (bdep->arg[8]) {
                    softdep_removefromlist((softdep *) bdep->arg[8]);
                    if (((softdep *)bdep->arg[8])->arg[6]) {
                        softdep_removefromlist(((softdep *)bdep->arg[8])->arg[6]);
                        softdep_deallocate(((softdep *)bdep->arg[8])->arg[6]);
                    }
                    softdep_deallocate((softdep *) bdep->arg[8]);
                }
                softdep_removefromlist(bdep);
                softdep_deallocate(bdep);
            }
            softdep_deallocate(indirdep);
        } else if (bdep->type == SOFTDEP_ALLOCSAFE) {
            softdep_removefromlist((softdep *) bdep->arg[2]);
            softdep_deallocate((softdep *) bdep->arg[2]);
            if (bdep->arg[6]) {
                softdep_removefromlist(bdep->arg[6]);
                softdep_deallocate(bdep->arg[6]);
            }
            softdep_removefromlist(bdep);
            softdep_deallocate(bdep);
        } else {
            LOG_ERROR("Unknown softdep type at deallocate_indirectblock: %d\n", bdep->type)
;
                INDUCE_FAULT();
            }
        }
        FREE_LOCK;
        if (indirdepmem) {
            KERNEL_MEMORY_DEALLOCATE(indirdepmem, BUF_BLOCK_SIZE(bp));
        }
}

/* This routine is called just before a file block is de-          */
/* allocated.  It simply de-allocates any associated dependency    */
/* structures that precede "id".  The "id" checking is required    */
/* because <vfsid, inum, lbn> triples can be reused after a        */
/* file length reduction and before the de-allocation              */
/* dependencies are satisfied.                                     */
void softdep_deallocate_filepage(ip, vfsid, inum, lbn, id)
struct inode *ip;           /* inode that owned the file page */
int vfsid;                  /* vfs identification for the file */
int inum;                   /* inode number of the file */
int lbn;                    /* lbn within the file for the file page */
int id;                     /* ‘‘id’’ number of the "FREEBLOCKS" structure */
{
    int i;
    softdep *pdep;
    softdep *keep = NULL;
    softdep *freefrags = NULL;
    softdep *tmp;
    softdep *pagedep = softdep_checkfor_softdep(vfsid, inum, lbn);

    if (pagedep == NULL) {
        FREE_LOCK;
        return();
    }
    while (pagedep->arg[4]) {
        pdep = ((softdep *) pagedep->arg[4])->next;
        softdep_removefromlist(pdep);
        if (pdep->id > id) {
            pdep->listtail = &keep;
            softdep_appendtolist(pdep);
        } else if (pdep->type == SOFTDEP_ALLOCSAFE) {
            if (pdep->arg[6]) {
                softdep_removefromlist((softdep *) pdep->arg[6]);
                softdep_deallocate((softdep *) pdep->arg[6]);
            }
            tmp = (softdep *) pdep->arg[2];
            softdep_removefromlist(tmp);
            if ((tmp->type == SOFTDEP_ALLOCDIRECT) && (tmp->arg[4] != 0) && (tmp->arg[
3]) != tmp->arg[4]) && (tmp->arg[7] != SOFTDEP_STATE_COMPLETE)) {
                tmp->next = freefrags;
                freefrags = tmp;
            } else {
```

```
          softdep_deallocate(tmp);
        }
        softdep_deallocate(pdep);
      } else if (pdep->type == SOFTDEP_REMOVE) {
        pdep->listtail = &softdep_workitem_pending;
        softdep_appendtolist(pdep);
      } else {
        LOG_ERROR("Unknown arg4 softdep type %d at deallocate_filepage\n", pdep->type);
        INDUCE_FAULT();
      }
    }
    while (keep) {
      pdep = keep->next;
      softdep_removefromlist(pdep);
      pdep->listtail = (softdep **) &pagedep->arg[4];
      softdep_appendtolist(pdep);
    }
    for (i=5; i<=8; i++) {
      while (pdep = (softdep *) pagedep->arg[i]) {
        softdep_removefromlist(pdep);
        if (pdep->id > id) {
          pdep->listtail = &keep;
          softdep_appendtolist(pdep);
        } else if (pdep->type == SOFTDEP_ADD) {
          if (pdep->arg[5]) {
            softdep_removefromlist(pdep->arg[5]);
            softdep_deallocate(pdep->arg[5]);
          }
          if (tmp = (softdep *) pdep->arg[6]) {
            while (tmp->arg[7] != (int) pdep) {
              tmp = (softdep *) tmp->arg[7];
            }
            tmp->arg[7] = pdep->arg[7];
            if ((tmp == pdep->arg[6]) && (tmp->arg[7] == 0)) {
              softdep_removefromlist(tmp);
              softdep_removefromhash(tmp);
              softdep_deallocate(tmp);
            }
          }
          softdep_deallocate(pdep);
        } else {
          LOG_ERROR("Unknown argI softdep type %d at deallocate_filepage\n", pdep->type
);

          INDUCE_FAULT();
        }
      }
      while (keep) {
        pdep = keep->next;
        softdep_removefromlist(pdep);
        pdep->listtail = (softdep **) &pagedep->arg[i];
        softdep_appendtolist(pdep);
        }
      }
      if ((pagedep->arg[4] | pagedep->arg[5] | pagedep->arg[6] | pagedep->arg[7] | pagedep->arg[
8]) == 0) {
        softdep_removefromlist(pagedep);
        softdep_removefromhash(pagedep);
        softdep_deallocate(pagedep);
      }
      FREE_LOCK;
      while (pdep = freefrags) {
        int freesize = FS_FRAGROUNDUP(FS_ID(ip), FS_SIZEOFBLOCK(FS_ID(ip), pdep->a
rg[6], pdep->arg[2]));
        UFS_BLOCK_DEALLOCATE(ip, pdep->arg[4], freesize);
        freefrags = pdep->next;
        ACQUIRE_LOCK;
        softdep_deallocate(pdep);
        FREE_LOCK;
      }
    }
```

15. Directory entry addition dependencies

When adding a new directory entry, the inode (with its incremented link count) must be written to disk before the directory entry's pointer to it. Also, if the inode is newly allocated, the corresponding freemap must be updated (on disk) before the directory entry's pointer. These requirements are met via undo/redo on the directory entry's pointer, which consists simply of the inode number. The ufs implementation with which we work adds directory entries in multiple routines. A procedure (below) is called by each to set up the dependency structures.

As directory entries are added and deleted, the free space within a directory block can become fragmented. The ufs file system will compact a fragmented directory block to make space for a new entry. When this occurs, the offsets of previously added entries change. Any "ADD" dependency structures corresponding to these entries must be updated with the new offsets. Two procedures (below) are used to accomplish this task.

```
/* This routine is called after the in-memory inode's link              */
/* count has been incremented, but before the directory entry's         */
/* pointer to the inode has been set.                                   */
void softdep_setup_add(dp, lbn, ip, offset, bitmapdep)
struct inode *dp;           /* inode for directory */
int lbn;                    /* block within directory containing new entry */
struct inode *ip;           /* inode pointed to by new directory entry */
int offset;                 /* offset of new inode pointer within directory block */
                            /* For convenience, we use use the offset of the directory */
                            /* entry structure and index into the structure as needed. */
softdep *bitmapdep;         /* from the corresponding inode allocation, if any */
```

```
{
  softdep *new = softdep_allocate();
  softdep *idep = softdep_allocate();
  softdep *extra = softdep_allocate();
  softdep *pagedep;
  softdep *inodedep;

  new->type = SOFTDEP_ADD;
  new->arg[0] = INODE_NUMBER(dp);
  new->arg[1] = SOFTDEP_STATE_ATTACHED;
  new->arg[2] = offset;
  new->arg[3] = INODE_NUMBER(ip);
  new->arg[4] = lbn;
  idep->type = SOFTDEP_ADDSAFE;
  new->arg[5] = (int) idep;
  idep->arg[1] = (int) new;
  pagedep = softdep_get_softdep(VFS_ID(dp), INODE_NUMBER(dp), lbn);
  inodedep = softdep_locked_get_softdep(VFS_ID(ip), INODE_NUMBER(ip), -1, extra);
  new->listtail = (softdep **) &pagedep->arg[(5 + ((offset >> 2) & 0x00000003))];
  idep->listtail = (softdep **) &inodedep->arg[4];
  softdep_appendtolist(new);
  softdep_appendtolist(idep);
  if (bitmapdep == NULL) {
    bitmapdep = softdep_locked_checkfor_softdep(VFS_ID(ip), INODE_NUMBER(ip), -2);
  }
  if (bitmapdep) {
    if (bitmapdep->listtail == &softdep_completed_bitmapdeps) {
      softdep_removefromlist(bitmapdep);
      softdep_removefromhash(bitmapdep);
      softdep_deallocate(bitmapdep);
    } else {
      new->arg[6] = (int) bitmapdep;
      new->arg[7] = bitmapdep->arg[7];
      bitmapdep->arg[7] = (int) new;
    }
  } else {
    new->arg[6] = 0;
    new->arg[7] = 0;
  }
  FREE_LOCK;
}

/* This procedure is called just before beginning to compact a          */
/* directory block (which must be owned exclusively by the               */
/* caller) to determine whether or not pending dependency                */
/* structures may need to be updated and, if so, return the              */
/* corresponding pagedep (so that it is not necessary to search          */
/* for it every time).              */
softdep *softdep_directoryentry_change_pagedep(dp, lbn)
struct inode *dp;                /* inode for directory */
int lbn;                         /* block number within directory */
{
  softdep *pagedep = softdep_checkfor_softdep(VFS_ID(dp)), dp->i_number, lbn);
  if ((pagedep) && ((pagedep->arg[5] | pagedep->arg[6] | pagedep->arg[7] | pagedep->arg[8])
== 0)) {
    /* if no pending "ADD" dependencies for the directory block */
    pagedep = NULL;
  }
  FREE_LOCK;
  return(pagedep);
}

/* This procedure is called to change the offset of a directory          */
/* entry.  Note that the actual entry movement is performed in          */
/* this procedure.  The pagedep returned from the call above is */
/* passed as a parameter.  This procedure is not called if the          */
/* pagedep is NULL.                                                     */
void softdep_change_directoryentry_offset(pagedep, dp, lbn, fbp, oldoffset, newoffset, entrysi
ze)
softdep *pagedep;             /* returned from previous procedure */
struct inode *dp;             /* inode for directory */
int lbn;                      /* block number within directory */
struct fbuf *fbp;             /* fbuf for directory block */
int oldoffset;                /* current offset of directory entry */
int newoffset;                /* new offset of directory entry */
int entrysize;                /* size of directory entry */
{
  softdep *pdep;
  int fbpaddr = (int) FBUF_MEMORY_BLOCK(fbp);

  ACQUIRE_LOCK;
  if ((pagedep->hash_next == NULL) || (pagedep->arg[0] != VFS_ID(dp)) || (pagedep->arg[1]
!= INODE_NUMBER(ip)) || (pagedep->arg[2] != lbn)) {
    LOG_ERROR("invalid pagedep state at change_directoryentry_offset\n");
    INDUCE_FAULT();
  }
  if ((pagedep) && (pdep = (softdep *) pagedep->arg[(5 + ((oldoffset >> 2) & 0x00000003))]
)) {
    softdep *tail = pdep;
    do {
      if ((pdep->type == SOFTDEP_ADD) && (pdep->arg[2] == oldoffset)) {
        pdep->arg[2] = newoffset;
        if ((newoffset & 0x0000000C) != (oldoffset & 0x0000000C)) {
          softdep_removefromlist(pdep);
          pdep->listtail = (softdep **) &pagedep->arg[(5 + ((newoffset >> 2) & 0x00000003)
)];
          softdep_appendtolist(pdep);
        }
        break;
      }
      pdep = pdep->next;
    } while (pdep != tail);
```

```
        }
        BLOCK_COPY((fbpaddr + oldoffset), (fbpaddr + newoffset), entrysize);
        FREE_LOCK;
    }
```

16. Directory entry removal dependencies

When removing a directory entry, the entry's inode pointer must be nullified on disk before the corresponding inode's link count is decremented (possibly freeing the inode for re-use). This dependency is handled by updating the directory entry but delaying the inode count reduction until after the directory block has been written to disk. After this point, the inode count can be decremented whenever it is convenient. The ufs implementation with which we work removes directory entries in several routines. In each, a procedure (below) is called to set up the dependency structures.

```
/* This routine should be called immediately after removing            */
/* a directory entry.  The inode's link count should not be            */
/* decremented by the calling procedure -- the soft updates            */
/* code will perform this task when it is safe.                        */
void softdep_setup_remove(dp, lbn, ip, offset)
struct inode *dp;          /* inode for the directory being modified */
int lbn;                   /* directory block number within file */
struct inode *ip;          /* inode for directory entry being removed */
int offset;                /* offset of directory entry in block */
{
    softdep *new = softdep_allocate();
    softdep *tmp;
    softdep *pagedep;

    new->type = SOFTDEP_REMOVE;
    new->arg[0] = VFS_ID(ip);
    new->arg[1] = INODE_NUMBER(ip);
    pagedep = softdep_get_softdep(VFS_ID(dp), INODE_NUMBER(dp), lbn);
    new->listtail = (softdep **) &pagedep->arg[4];
    if (tmp = (softdep *) pagedep->arg[(5 + ((offset >> 2) & 0x00000003))]) {
        softdep *tail = tmp;
        do {
            /* Check for an "ADD dependency for the same directory entry. */
            /* If present, then both dependencies become obsolete and can */
            /*  be de-allocated. */
            if (tmp->arg[2] == offset) {
                /* Must be ATTACHED at this point, so just delete it */
                if (tmp->arg[3] != INODE_NUMBER(ip)) {
                    LOG_ERROR("Non-matching inum %d being removed at softdep_setup_remove, s
hould be %d\n", INODE_NUMBER(ip), tmp->arg[3]);
                    INDUCE_FAULT();
                }
                if (tmp->arg[5]) {
                    softdep_removefromlist((softdep *) tmp->arg[5]);
                    softdep_deallocate((softdep *) tmp->arg[5]);
                }
                if (tmp->arg[6]) {
                    softdep *tmp2 = (softdep *) tmp->arg[6];
                    while (tmp2->arg[7] != (int) tmp) {
                        tmp2 = (softdep *) tmp2->arg[7];
                    }
                    tmp2->arg[7] = tmp->arg[7];
                    if ((tmp2 == tmp->arg[6]) && (tmp2->arg[7] == 0)) {
                        softdep_removefromlist(tmp2);
                        softdep_removefromhash(tmp2);
                        softdep_deallocate(tmp2);
                    }
                }
                softdep_removefromlist(tmp);
                softdep_deallocate(tmp);
                pagedep = NULL;
                break;
            }
            tmp = tmp->next;
        } while (tmp != tail);
    }
    if (pagedep) {
        softdep_appendtolist(new);
    } else {
        softdep_deallocate(new);
        LINK_COUNT(ip) -= 1;
        INODE_FLAGS(ip) |= INODE_MODIFIED;
    }
    FREE_LOCK;
}

/* This workitem decrements the inode's link count.  If the            */
/* link count reaches zero, the file is removed.  For the "GET"        */
/* function call below, "arg[0]" contains the appropriate              */
/* "VFS_ID(ip)" value and "arg[1]" contains the inode number.          */
/* If values other than these (or those that can be computed           */
/* directly from these) are needed, they must be added in              */
/* "softdep_setup_remove".                                             */
void softdep_handle_workitem_remove(work)
softdep *work;
{
    struct inode *ip = UFS_GET_INCORE_INODE(/*  ...  */);
    LINK_COUNT(ip) -= 1;
    INODE_FLAGS(ip) |= INODE_MODIFIED;
    UFS_RETURN_INCORE_INODE(ip);
}
```

17. Inode de-allocation dependencies

When an inode's link count is reduced to zero, it can be de-allocated. We found it convenient to postpone de-allocation until after the inode is written to disk with its new link count (zero). At this point, all of the on-disk inode's block pointers are nullified and, with careful dependency list ordering, all dependencies related to the inode will be satisfied and the corresponding dependency structures de-allocated. So, if/when the inode is reused, there will be no mixing of old dependencies with new ones. This artificial dependency is set up by the block de-allocation procedure above (#14) and completed by the following procedure.

```
void softdep_handle_workitem_filefree(work)
softdep *work;
{
  struct inode tip;

  if (softdep_checkfor_softdep(work->arg[0], work->arg[1], -1)) {
    LOG_ERROR("inodedep survived file removal at softdep_handle_workitem_filefree\n");
    INDUCE_FAULT();
  }
  FREE_LOCK;
  INODE_NUMBER(&tip) = work->arg[1];
  DEVICE_NUMBER(&tip) = work->arg[2];
  FS_ID(&tip) = work->arg[3];
  UFS_INODE_DEALLOCATE(&tip, INODE_NUMBER(&tip));
}
```

18. Disk writes

The dependency structures constructed above are most actively used when file system blocks are written to disk. No constraints are placed on when a block can be written, but unsatisfied update dependencies are made safe by modifying (or replacing) the source memory for the duration of the disk write. When the disk write completes, the memory block is again brought up-to-date.

```
/* Called just before initiating a disk write for a file page.          */
/* This is needed because of the VM/FS integration.                     */
void softdep_initiate_filepage_write(bp, ip, lbn)
struct buf *bp;          /* the bp for the write request */
struct inode *ip;        /* the file's inode */
int lbn;                 /* block number within file to be written */
{
  softdep *tmp = softdep_checkfor_softdep(VFS_ID(ip), INODE_NUMBER(ip), lbn);
  if (tmp) {
    tmp->listtail = &bp->b_dep;
    softdep_appendtolist(tmp);
  }
  FREE_LOCK;
}
```

```
/* Called just before entering the device driver to initiate           */
/* a new disk write.  The OS with which we work initiates an           */
/* I/O request by passing a "struct buf *" to the appropriate          */
/* device driver.                                                       */
void softdep_handle_disk_write_initiation(bp)
struct buf *bp;          /* structure describing disk write to occur */
{
  softdep *dep;
  softdep *pdep;
  struct direct *entryptr;
  int i;

  ACQUIRE_LOCK;
  if (dep = bp->b_dep) {
checknext:
      if (dep->type == SOFTDEP_PAGEDEP) {
        for (i=5; i<=8; i++) {
          pdep = (softdep *) dep->arg[i];
          if (pdep == NULL) {
            continue;
          }
          do {
            entryptr = (struct direct *)(BUF_MEMORY_BLOCK(bp) + pdep->arg[2]);
            if (DIRENTRY_INODE_NUMBER(entryptr) != pdep->arg[3]) {
              LOG_ERROR("Non-matching inum at softdep_handle_disk_write_initiation, %d
!= %d\n", DIRENTRY_INODE_NUMBER(entryptr), pdep->arg[3]);
              INDUCE_FAULT();
            }
            DIRENTRY_INODE_NUMBER(entryptr) = 0;
            pdep->arg[1] = SOFTDEP_STATE_UNDONE;
            pdep = pdep->next;
          } while (pdep != dep->arg[i]);
        }
      } else if (dep->next->type == SOFTDEP_INODEBLOCK) {
        softdep_initiate_write_inodeblock(dep->next, bp);
      } else if (dep->type == SOFTDEP_INDIRDEP) {
                /* replace up-to-date version with safe version */
        BUF_MEMORY_BLOCK(bp) = dep->arg[4];
      } else if ((dep->type == SOFTDEP_ALLOCSAFE) && ((dep = dep->next) != bp->b_dep
)) {

        goto checknext;
      }
  }
  FREE_LOCK;
}

/* Called from within the procedure above to deal with                 */
/* unsatisfied allocation dependencies in an inodeblock.  Note          */
/* that the soft updates lock is held upon entry and never              */
/* released.                                                            */
```

```
void softdep_initiate_write_inodeblock(inodeblock, bp)
softdep *inodeblock;              /* The "INODEBLOCK" dependency structure */
struct buf *bp;                   /* The inode block */
{
  softdep *inodedep;
  softdep *idep;
  struct dinode *dp;
  int oldblksize;
  if (inodedep = (softdep *) inodeblock->arg[0]) {
    do {
      dp = (struct dinode *)BUF_MEMORY_BLOCK(bp) + FS_INODE_OFFSET(inodebloc
k->arg[1], inodedep->arg[1]);
      if (idep = (softdep *) inodedep->arg[6]) {
        do {
          if (idep->type != SOFTDEP_ALLOCDIRECT) {
            LOG_ERROR("Unknown arg5 softdep type %d in update_inodeblock\n", idep->t
ype);
            INDUCE_FAULT();
          }
          if (idep->arg[7] == SOFTDEP_STATE_ATTACHED) {
            if (idep->arg[2] < NDADDR) {
              if (DIRECT_BLOCK_POINTER(dp, idep->arg[2]) != idep->arg[3]) {
                LOG_ERROR("Direct pointer #%d does not match at softdep_update_inodebl
ock: %d != %d\n", idep->arg[2], DIRECT_BLOCK_POINTER(dp, idep->arg[2]), idep->arg[3]);
                INDUCE_FAULT();
              }
              DIRECT_BLOCK_POINTER(dp, idep->arg[2]) = idep->arg[4];
              /* Manipulate the file size field(s).  If this is the */
              /* current outermost block number, then replace the */
              /* current size with the old size (arg[6]). Note that */
              /* the dependency list is traversed in reverse order, */
              /* so multiple unsafe extensions to a file will properly */
              /* modify the file's length.  I suggest adding this */
              /* part of the undo/redo for block allocation only */
              /* after debugging the remainder...  FSCK should properly */
              /* adjust the file length if the system crashes. */
              oldblksize = FS_FRAGROUNDUP(inodeblock->arg[1], FS_SIZEOFBLOCK(in
odeblock->arg[1], idep->arg[6], idep->arg[2]);
              if (idep->arg[5] > FILE_SIZE(dp)) {
                LOG_ERROR("newsize is greater than filesize in softdep_update_inodeblock"
);
                INDUCE_FAULT();
              }
              if ((idep->arg[5] == FILE_SIZE(dp)) || ((idep->arg[6] < FILE_SIZE(dp)) && (o
ldblksize > 0) && (oldblksize < FS_BLOCK_SIZE(inodeblock->arg[1])))) {
                idep->arg[5] = FILE_SIZE(dp);
                FILE_SIZE(dp) = idep->arg[6];
              }
            } else {
              if (INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] - NDADDR)) != idep->ar
g[3]) {
                LOG_ERROR("Indirect pointer #%d does not match at softdep_update_inode
block: %d != %d\n", (idep->arg[2] - NDADDR), INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] -
NDADDR)), idep->arg[3]);
                INDUCE_FAULT();
              }
              INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] - NDADDR)) = 0;
              /* The file size for indirect block additions is not */
              /* guaranteed.  Such a guarantee would be non-trivial */
              /* to achieve.  The conventional synch. write impl. */
              /* also does not make this guarantee.  Again, FSCK should */
              /* catch and fix discrepancies. Arguably, the file */
              /* size can be over-estimated without destroying integrity */
              /* when the file moves into the indirect blocks (i.e., is */
              /* large).  If we want to postpone FSCK, we are stuck with */
              /* this argument. */
            }
          } else {
            LOG_ERROR("Unknown alloc softdep state at update_inodeblock: %d\n", idep->
arg[7]);
            INDUCE_FAULT();
          }
          idep = idep->prev;   /* traverse list from tail to head */
        } while (idep != (softdep *) inodedep->arg[6]);
      }
      inodedep = inodedep->next;
    } while (inodedep != (softdep *) inodeblock->arg[0]);
  }
}

/* This routine is called during the completion interrupt            */
/* service routine for a disk write (from the procedure called       */
/* by the device driver to inform the file system caches of          */
/* a request completion).  It should be called early in this         */
/* procedure, before the block is made available to other            */
/* processes or other routines are called.                           */
void ufs_handle_disk_write_complete(bp)
struct buf *bp;                 /* describes the completed disk write */
{
  softdep *bdep;
  softdep *pdep;
  softdep *reattach = NULL;
  int *indirdepdel = NULL;

  ACQUIRE_LOCK;
  while (bp->b_dep) {
    bdep = bp->b_dep->next;
    softdep_removefromlist(bdep);
    if ((bdep->type == SOFTDEP_FREEBLOCKS) ||
        (bdep->type == SOFTDEP_FREEFRAG) ||
        (bdep->type == SOFTDEP_FILEFREE)) {
      bdep->listtail = &softdep_workitem_pending;
```

```
    softdep_appendtolist(bdep);
  } else if (bdep->type == SOFTDEP_PAGEDEP) {
    softdep_handle_written_filepage(bdep, bp);
  } else if (bdep->type == SOFTDEP_INODEBLOCK) {
    softdep_handle_written_inodeblock(bdep, bp);
    if (bdep->arg[0]) {
      reattach = bdep;
    } else {
      softdep_deallocate(bdep);
    }
  } else if (bdep->type == SOFTDEP_ADDSAFE) {
    pdep = (softdep *) bdep->arg[1];
    pdep->arg[5] = 0;
    if (pdep->arg[6]) {
    } else if (pdep->arg[1] == SOFTDEP_STATE_ATTACHED) {
      softdep_removefromlist(pdep);
      softdep_deallocate(pdep);
    } else {
      pdep->arg[1] = SOFTDEP_STATE_COMPLETE;
    }
    softdep_deallocate(bdep);
  } else if (bdep->type == SOFTDEP_ALLOCSAFE) {
    softdep_handle_blockallocdep_partdone(bdep);
  } else if (bdep->type == SOFTDEP_BITMAPSAFE_BLOCK) {
    if (bdep->arg[2]) {
      softdep_handle_blockallocdep_partdone(bdep);
    } else {
      bdep->listtail = &softdep_completed_bitmapdeps;
      softdep_appendtolist(bdep);
    }
  } else if (bdep->type == SOFTDEP_BITMAPSAFE_INODE) {
    if (bdep->arg[7]) {
      while (pdep = (softdep *) bdep->arg[7]) {
        bdep->arg[7] = pdep->arg[7];
        pdep->arg[6] = 0;
        if (pdep->arg[5]) {
        } else if (pdep->arg[1] == SOFTDEP_STATE_ATTACHED) {
          softdep_removefromlist(pdep);
          softdep_deallocate(pdep);
        } else {
          pdep->arg[1] = SOFTDEP_STATE_COMPLETE;
        }
      }
      softdep_removefromhash(bdep);
      softdep_deallocate(bdep);
    } else {
      bdep->listtail = &softdep_completed_bitmapdeps;
      softdep_appendtolist(bdep);
    }
  } else if (bdep->type == SOFTDEP_INDIRDEP) {
    BUF_FLAGS(bp) |= BUF_DIRTY;
```

41

```
        BUF_MEMORY_BLOCK(bp) = bdep->arg[2];
        if (bdep->arg[3]) {
          reattach = bdep;
        } else {
          indirdepdel = (int *) bdep->arg[4];
          softdep_deallocate(bdep);
        }
      } else {
        LOG_ERROR("Unknown softdep type %d at handle_disk_write_complete\n", bdep->ty
pe);

        INDUCE_FAULT();
      }
    }
    if (reattach) {
      reattach->listtail = &bp->b_dep;
      softdep_appendtolist(reattach);
    }
    FREE_LOCK;
    if (indirdepdel) {
      KERNEL_MEMORY_DEALLOCATE(indirdepdel, BUF_BLOCK_SIZE(bp));
    }
  }

  /* Called from within the procedure above.  Note that the soft          */
  /* updates lock is held upon entry and exit.                            */
  void softdep_handle_blockallocdep_partdone(done)
  softdep *done;    /* the completed dependency structure */
  {
    softdep *idep = (softdep *) done->arg[2];
    int freedone = 1;
    idep->arg[8] = done->arg[6];
    if (done->arg[6]) {
      ((softdep *)done->arg[6])->arg[6] = 0;
    } else if (idep->type == SOFTDEP_ALLOCDIRECT) {
      if ((idep->arg[4] != 0) && (idep->arg[3] != idep->arg[4]) && (idep->arg[2] < NDADDR)
) {

        done->type = SOFTDEP_FREEFRAG;
        done->arg[2] = idep->arg[4];  /* first frag number to free */
        done->arg[3] = idep->arg[6];  /* file size in bytes */
        done->arg[5] = idep->arg[2];  /* lbn in file */
        done->listtail = (softdep **) done->arg[4];
        softdep_appendtolist(done);
        freedone = 0;
      }
      if (idep->arg[7] == SOFTDEP_STATE_ATTACHED) {
        softdep_removefromlist(idep);
        softdep_deallocate(idep);
      } else {
        idep->arg[7] = SOFTDEP_STATE_COMPLETE;
      }
    } else if (idep->type == SOFTDEP_ALLOCINDIRECT) {
```

```
      int *indirdepmem = (int *) idep->arg[4];
      indirdepmem[(idep->arg[2])] = idep->arg[3];
      softdep_removefromlist(idep);
      softdep_deallocate(idep);
    } else {
      LOG_ERROR("Unknown alloc softdep type at handle_disk_write_complete: %d\n", idep
->type);
      INDUCE_FAULT();
    }
    if (freedone) {
      softdep_deallocate(done);
    }
  }

/* Called from within the procedure above to restore in-memory           */
/* inode block contents to their most up-to-date state. Note             */
/* that the soft updates lock is held upon entry and never               */
/* released.                                                             */
void softdep_handle_written_inodeblock(inodeblock, bp)
softdep *inodeblock;          /* the "INODEBLOCK" dependency structure */
struct buf *bp;               /* the inode block */
{
  softdep *inodedep;
  softdep *idep;
  softdep *inodedeps = NULL;
  softdep *doneallocs = NULL;
  struct dinode *dp;

  while (inodedep = (softdep *) inodeblock->arg[0]) {
    softdep_removefromlist(inodedep);
    dp = (struct dinode *)BUF_MEMORY_BLOCK(bp) + FS_INODE_OFFSET(inodeblock
->arg[1], inodedep->arg[1]);
    if (idep = (softdep *) inodedep->arg[6]) {
      idep = idep->next;  /* traverse list from head to tail */
      /* Keep the block dirty so that it will not be reclaimed */
      /* until all associated dependencies have been cleared and */
      /* the corresponding updates written to disk. */
      BUF_FLAGS(bp) |= BUFFER_DIRTY;
      do {
        if (idep->type != SOFTDEP_ALLOCDIRECT) {
          LOG_ERROR("Unknown arg5 softdep type %d in reattach_inodedep\n", idep->typ
e);
          INDUCE_FAULT();
        }
        if (idep->arg[2] < NDADDR) {
          if (DIRECT_BLOCK_POINTER(dp, idep->arg[2]) != idep->arg[4]) {
            LOG_ERROR("Direct pointer #%d does not match at softdep_reattach_inodedep:
%d != %d\n", idep->arg[2], DIRECT_BLOCK_POINTER(dp, idep->arg[2]), idep->arg[4]);
            INDUCE_FAULT();
          }
          DIRECT_BLOCK_POINTER(dp, idep->arg[2]) = idep->arg[3];
          /* Reset the file size to its most up-to-date value. */
          if ((FILE_SIZE(dp) == idep->arg[6]) || (idep->arg[5] > FILE_SIZE(dp))) {
            FILE_SIZE(dp) = idep->arg[5];
          }
        } else {
          if (INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] - NDADDR)) != 0) {
            LOG_ERROR("Indirect pointer #%d does not match at softdep_reattach_inodedep
: %d\n", (idep->arg[2] - NDADDR), INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] - NDADDR)));
            INDUCE_FAULT();
          }
          INDIRECT_BLOCK_POINTER(dp, (idep->arg[2] - NDADDR)) = idep->arg[3];
        }
        if (idep->arg[7] == SOFTDEP_STATE_COMPLETE) {
          idep->hash_next = doneallocs;
          doneallocs = idep;
        } else if (idep->arg[7] == SOFTDEP_STATE_UNDONE) {
          idep->arg[7] = SOFTDEP_STATE_ATTACHED;
        } else {
          LOG_ERROR("Unknown alloc softdep state %d at reattach_inodedep\n", idep->arg
[7]);
          INDUCE_FAULT();
        }
        idep = idep->next;
      } while (idep != ((softdep *)inodedep->arg[6])->next);
    }
    while (idep = doneallocs) {
      doneallocs = idep->hash_next;
      softdep_removefromlist(idep);
      softdep_deallocate(idep);
    }
    if ((inodedep->arg[4] | inodedep->arg[5] | inodedep->arg[6]) == 0) {
      softdep_removefromhash(inodedep);
      softdep_deallocate(inodedep);
    } else {
      inodedep->next = inodedeps;
      inodedeps = inodedep;
    }
  }
  while (inodedep = inodedeps) {
    inodedeps = inodedep->next;
    inodedep->listtail = (softdep **) &inodeblock->arg[0];
    softdep_appendtolist(inodedep);
  }
}

/* Called from within the procedure above.  Note that the soft           */
/* updates lock is held upon entry and exit.                             */
void softdep_handle_written_filepage(pagedep, bp)
softdep *pagedep;             /* "PAGEDEP" associated with the file page */
struct buf *bp;               /* describes the completed disk write */
{
```

```
    while (pagedep->arg[4]) {
      softdep *pdep = ((softdep *)pagedep->arg[4])->next;
      softdep_removefromlist(pdep);
      if (pdep->type == SOFTDEP_REMOVE) {
        pdep->listtail = &softdep_workitem_pending;
        softdep_appendtolist(pdep);
      } else if (pdep->type == SOFTDEP_ALLOCSAFE) {
        softdep_handle_blockallocdep_partdone(pdep);
      } else {
        LOG_ERROR("Unknown softdep type %d at handle_written_filepage\n", pdep->type);
        INDUCE_FAULT();
      }
    }
    softdep_removefromlist(pagedep);
    if ((pagedep->arg[5] | pagedep->arg[6] | pagedep->arg[7] | pagedep->arg[8]) == 0) {
      softdep_removefromhash(pagedep);
      softdep_deallocate(pagedep);
    } else {
      /* This occurs only for directory blocks with pending entry */
      /* addition dependencies.  This attempts to ensure that the */
      /* most recent directory additions reach stable storage within */
      /* a reasonable period of time.  If users access the directory */
      /* block before the 15 seconds expire, the reattach routine */
      /* (below) removes the structure from the delayed workitem queue. */
      pagedep->listtail = &softdep_workitem_delayed[((softdep_delayno + 15) & SOFTDEP_
MAXDELAYMASK)];
      softdep_appendtolist(pagedep);
    }
  }

  /* This workitem routine simply accesses the corresponding file        */
  /* page, reattaches the pagedep and initiates an asynchronous          */
  /* write.                                                              */
  void softdep_handle_workitem_pagedep(vfsid, inum, lbn)
  int vfsid;          /* VFS identification for the file */
  int inum;           /* inode number for the file */
  int lbn;            /* block number within file of the file page */
  {
    struct inode *ip;
    struct fbuf *fbp;

    ip = UFS_GET_INCORE_INODE(/*  ...  */);
    if (FBREAD(/*  ...  */, &fbp)) {
      LOG_ERROR("fbread failed in softdep_handle_workitem_pagedep\n");
      INDUCE_FAULT();
    }
    softdep_reattach_pagedep(ip, lbn, fbp);
    FBAWRITE(fbp);
    UFS_RETURN_INCORE_INODE(ip);
  }
```

19. Writing back in-core inode structures

The file system only accesses an inode’s contents when it occupies an "in-core" inode structure. These "in-core" structures are separate from the page frames used to cache inode blocks. Only the latter are transferred to/from the disk. So, when the updated contents of the "in-core" inode structure are copied to the corresponding in-memory inode block, the dependencies are also transferred. The following procedure is called when copying a dirty "in-core" inode to a cached inode block.

```
  /* This routine is called just after the "in-core" inode             */
  /* information has been copied to the in-memory inode block.         */
  /* Recall that an inode block contains several inodes.               */
  void softdep_update_inodeblock(ip, dp, bp)
  struct inode *ip;         /* the "in_core" copy of the inode */
  struct dinode *dp;        /* the inode block copy of the inode */
  struct buf *bp;           /* the inode block */
  {
    softdep *idep;
    int numcomplete = 0;
    int oldblksize;
    softdep *inodedep = softdep_checkfor_softdep(VFS_ID(ip), INODE_NUMBER(ip), -1);
    if (inodedep == NULL) {
      FREE_LOCK;
      return;
    }
    while (idep = (softdep *) inodedep->arg[4]) {
      idep = idep->next;
      softdep_removefromlist(idep);
      idep->listtail = &bp->b_dep;
      softdep_appendtolist(idep);
    }
    while (idep = (softdep *) inodedep->arg[5]) {
      idep = idep->next;
      softdep_removefromlist(idep);
      idep->listtail = (softdep **) &inodedep->arg[6];
      softdep_appendtolist(idep);
    }
    if (inodedep->arg[6] == 0) {
      softdep_removefromlist(inodedep);
      softdep_removefromhash(inodedep);
      softdep_deallocate(inodedep);
    } else {
      if ((bp->b_dep == NULL) || (bp->b_dep->next->type != SOFTDEP_INODEBLOCK)) {
        inodeblock->type = SOFTDEP_INODEBLOCK;
        inodeblock->listtail = &bp->b_dep;
        softdep_appendtolist(inodeblock);
        /* Make certain that "INODEBLOCK" is at head of list */
        bp->b_dep = inodeblock->prev;
        inodeblock->arg[0] = 0;
```

```
            inodeblock->arg[1] = FS_ID(ip);
            inodeblock->arg[2] = FS_BLOCK_SIZE(FS_ID(ip));
            inodeblock = NULL;
          }
          if (inodedep->listtail == NULL) {
            inodedep->listtail = (softdep **) &bp->b_dep->next->arg[0];
            softdep_appendtolist(inodedep);
          }
        }
        if (inodeblock) {
          softdep_deallocate(inodeblock);
        }
      }
```

20. In-core inode structure reclamation

Because there are a finite number of "in-core" inode structures, they are reused regularly. By transferring all inode-related dependencies to the in-memory inode block and indexing them separately (via "INODEDEP"s), we can allow "in-core" inode structures to be reused at any time and avoid any increase in contention.

21. Page frame reclamation

As with "in-core" inode structures, the set of page frames available for staging/caching file data is finite. Because of the integration of virtual memory and file caching in our base operating system, cached file pages can be reclaimed by the virtual memory system WITHOUT informing the file system. Therefore, our implementation must be able to match non-satisfied dependencies to file pages independent of their cache status.

```
      /* This routine is called when the file system code accesses          */
      /* any file page.  It should be called to reset any undone            */
      /* updates before the page is returned.  Note that such work is       */
      /* necessary only with directory blocks.  In our implementation,      */
      /* it is necessary to touch the file page before calling this         */
      /* routine, to make certain that it is resident in main memory        */
      /* (so that the soft updates lock is not held while faulting it in).  */
      void softdep_reattach_pagedep(ip, lbn, fbp)
      struct inode *ip;          /* the inode for the accessed file */
      int lbn;                   /* the block number within the file */
      struct fbuf *fbp;          /* the file page */
      {
        softdep *pdep;
        struct direct *entryptr;
        int numcomplete = 0;
                                 /* get the starting address of the page */
        unsigned int fbpaddr = FS_BLKSTOFRAGS(FS_ID(ip), FS_FRAGSTOBLKS(FS_ID(ip),
(unsigned int) FBUF_MEMORY_BLOCK(fbp)));

        softdep *pagedep = softdep_checkfor_softdep(VFS_ID(ip), INODE_NUMBER(ip), lbn);
        if (pagedep) {
          for (i=5; i<=8; i++) {
            pdep = (softdep *) pagedep->arg[i];
            if (pdep = NULL) {
              continue;
            }
            do {
              if (pdep->arg[1] == SOFTDEP_STATE_ATTACHED) {
                /* Skip traversing the full list if its already been */
                /* attached since the last disk write.  Note that this */
                /* would be better accomplished with a flag in the */
                /* pagedep. */
                goto reattach_pagedep_done;
              }
              entryptr = (struct direct *)(fbpaddr + pdep->arg[2]);
              DIRENTRY_INODE_NUMBER(entryptr) = pdep->arg[3];
              if (pdep->arg[1] == SOFTDEP_STATE_UNDONE) {
                pdep->arg[1] = SOFTDEP_STATE_ATTACHED;
              } else if (pdep->arg[1] == SOFTDEP_STATE_COMPLETE) {
                pdep->hash_next = dellist;
                dellist = pdep;
              } else {
                LOG_ERROR("Unknown add softdep state in reattach_pagedep: %d\n", pdep->arg[
1]);

                INDUCE_FAULT();
              }
              pdep = pdep->next;
            } while (pdep != pagedep->arg[i]);
          }
          while (pdep = dellist) {
            dellist = pdep->hash_next;
            softdep_removefromlist(pdep);
            softdep_deallocate(pdep);
          }
          softdep_removefromlist(pagedep);
          if ((pagedep->arg[4] | pagedep->arg[5] | pagedep->arg[6] | pagedep->arg[7] | pagedep->ar
g[8]) == 0) {
            softdep_removefromhash(pagedep);
            softdep_deallocate(pagedep);
          }
        }
reattach_pagedep_done:
        FREE_LOCK;
      }
```