PTFS: A Non-Journalling Filesystem for Fast and Consistent Recovery

Rachel Greenstadt, Chris Laas, Jacob Strauss

Abstract— **We present Phase Tree File System (PTFS), a filesystem designed for quick and consistent recovery. Its "phase-tree" approach provides atomic changes between consistent states of the filesystem without the attendant complexity of journalling. We implement a simple version of the system and integrate it with SFS. This implementation is not optimized for performance, but instead demonstrates the recovery properties of the system. We conclude that this approach would be viable if its performance could be improved, and suggest ways of doing so.**

I. Introduction and Motivation

The problem of maintaining filesystem consistency across system crashes has long been important to systems designers. Filesystem operations must appear atomic to users, but many involve several steps internally. Should the machine crash in between any of these internal operations, the machine must be able to return filesystem data to a consistent state in a reasonable amount of time. For example, in a `rename` operation, two directories must be altered: a name is removed from one, and added to another. If the computer is interrupted during this operation, a file could be rendered either nameless or doubly named. Caching, an essential optimization, compounds the consistency problem: a disk salvager program often has difficulty determining what data has actually been written to disk and in what order.

Traditionally, this problem has been solved by disk utilities such as `fsck`, which search the disk in order to restore the filesystem to a consistent state. However, as filesystems grow larger, the amount of time spent by these startup utilities increases at an intolerable rate. Journalling mechanisms have been a common approach to improving startup and recovery time: a journalling filesystem periodically writes checkpoints to disk when a consistent representation is reached, and when recovering from a crash, it simply disregards everything after the last checkpoint on disk. An alternate approach that we have decided to explore, the approach of "phase-tree" filesystems[6], implements atomic commits of changes to the metadata by making the changes in a "working copy" of the data, and periodically replacing the root of the entire filesystem with a pointer to this working set. This avoids the extra storage of the log, and, more importantly, the complexity and overhead it entails. We have built a filesystem, PTFS, based on these concepts.

The time available to us significantly constrained the implementation of our design. We did not write code for a metadata cache, background defragmenter, or asynchronous commit, although these features would significant improve performance. Furthermore, working with SFS as a front end confined us to Solaris and C++, and limited some of the features we were able to provide; however, it provided a reasonable filesystem framework, taking some of the load of implementation off our backs.

II. Goals

The goal of our project was to create an on-disk filesystem and implement it with an SFS[2] front end. The system would be represented simply but exhibit the properties of fast and consistent recovery. This was to be accomplished by making filesystem transactions atomic, so that the on-disk representation would always be in a consistent state. Once committed, operations

Thanks to Joe Foley, Malima Wolf, and Jessica Fry for putting up with us.

would be permanent and could only be undone by a second transaction. We wanted our system to work well with SFS, as this would provide a convenient user level interface to the filesystem.

In order to obtain these properties, we used the phase-tree approach described in the following section. The design of PTFS allows a high-performance implementation which would offer low latency, low CPU overhead, and reasonable space and copying efficiencies. The system which we implemented demonstrates the validity of the approach, but we did not strive for performance, instead focusing on clarity and simplicity.

III. The Phase Tree Approach to Filesystem Recovery

In the phase tree approach, modification to the filesystem are performed in a transient "working area" of the disk, until the changes are committed. At that point, the root of the filesystem is rewritten, swapping the pointers to the working area and the committed area; in effect, all changes to the on-disk filesystem occur atomically when the root is overwritten. A very simplistic and inefficient implementation might work as follows. The disk has three sections:

1. a single sector, the "current root" pointer
2. roughly half the disk, the committed set
3. roughly half the disk, the working set

All updates are made to the working set, which contains a fairly conventional Unix filesystem. Periodically, at a time when the working set is in a consistent state, a single write to the current root pointer toggles to the two sets. As a result, all the intermediate updates are made, in effect, atomically. If the system crashes in the middle of a critical section, the write is rolled back, as the committed set is the one that is loaded upon the next system startup.

Unfortunately, this system would result in unacceptable amounts of data copying and wasted space. When an update occurs, half of the disk must be copied into the other half in order to provide an up-to-date working set. However, because only the system metadata is required to recover from a crash, we can structure our disk so as to make this approach much less costly. Our system is more efficient in this regard. Our goal in the implementation was to write a proof-of-concept system which would always be consistent after a crash, and wouldn't take significant extra time to recover.

It is important to note that this sort of system does not completely replace a filesystem checker and fixer such as `fsck`, and that this is not the aim of the system. Such a utility would still be necessary in cases of hardware failure, which, since it is arbitrary, could make the filesystem unavailable or insufficient for recovery. The goal is to provide faster recovery in the case of a machine crash.

IV. Related Work

The inspiration for our project came from the phase-tree algorithm used by the Tux2[6] project, which is a patch to the `ext2` filesystem which turns it into a phase-tree system by structuring the filesystem data as a tree. The design of the Tux2 project is strongly driven by the desire to achieve compatibility with `ext2`. Due to this, it is limited in the ways in which it can structure the filesystem: for instance, there is a predetermined number of inodes. We did not desire compatibility with ext2, and so did not need to implement several of Tux2's more esoteric aspects (such as the tree structure).

Our system is also related to journalling approaches[5][4][7], and it uses some concepts associated with them, such as block indirection. Journalling systems are designed to provide consistency across crashes, and also to allow transactions to be recorded and replayed if necessary. Generally, all file system structure and metadata changes are written to the log in a synchronous manner; the filesystem periodically flushes these changes out to the primary data area. This type of logging is referred to as write-ahead logging, since the log is written before the data. This second property allows changes to be rolled forward in the event of a crash; however, it adds quite a bit of overhead to the system. A log requires a large amount of free space to be available at all times; it also tends to fragment the disk, requiring a periodic cleaning pass. Finally, while write performace on journalling filesystems is good, read performance suffers, due to poor locality.

Another way to deal with filesystem consistency is to use synchronous writes to sequence dependent metadata updates. FFS and BSD systems work this way. The primary problem with this type of approach is performance degredation. Metadata updates in these systems happen at disk speeds, which are considerably slower than processor and memory speeds. This can cause some filesystems to not bother with dependencies at all and aim to reconstruct whatever is possible after the fact.

A possible way around the need for synchronous writes or logging can be found in soft updates[1]. The idea behind soft updates is to delay writes for metadata changes. Dependencies between updates are tracked and then enforced during write-back. Updates can be rolled back before blocks are written and rolled forward afterwards. This approach looks very interesting, though decidedly complex. The filesystem must maintain very detailed dependency information to get the performance benefits.

V. System Design

A. Disk Layout

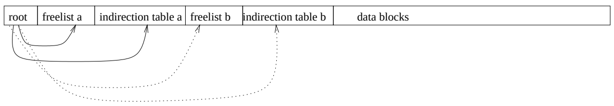

Fig. 1. On-Disk Structure

Our primary concern in implementing the on-disk structure was simplicity. Most of the advantage of our system lies in the fact that it can provide many of the benefits of journalling systems without the attendant complexity. It is therefore important to make sure that the system structure does not grow to be overly complex; there are only a few functionally distinct areas of the disk. The *metaroot block*, at physical position zero, contains global filesystem information (such as total number of blocks) and a flag indicating the current committed set; there are two identical metadata sets (one working and one committed), each containing a *block indirection table* and a *free-list* (Figure 1). The block indirection table is a mapping from "virtual" block numbers to "physical" block numbers; it contains one 32-bit value per block on disk. The free-list is a bitmap indexed by physical block numbers. Whenever a commit occurs the root is overwritten to toggle the committed and working sets; at this point, any changes to the new committed set must be copied into the new working set. Much of this work can be avoided simply by caching a small amount of metadata, however.

The block indirection table works very similarly to a page table in a virtual memory system. Above a certain level of abstraction, the filesystem uses only virtual block numbers, without knowing their physical location on disk; this makes switching the indirection tables upon a commit much simpler, and allows the lower levels of the filesystem to move around the physical location of data at will. Virtual block numbers also subsume the function of inode numbers, since there does not need to be any separate inode table in this model.

A different disk structure might be useful in minimizing the amount of copies. Tux2[6] structures its filesystem data and metadata as a tree, whose nodes are a metaroot and metadata index blocks, and whose leaves are block bitmaps, inode bitmaps, inode blocks, group descriptor blocks and spare metadata files. The argument is that this design can be restructured easily, and the entire tree does not have to be rewritten during an update. However, maintaining the freelist in this type of structure is difficult, because in order to work well it has to be a separate branch of the tree; this prevents the most important efficiency benefit from being realized, because all nodes up to the root (instead of simply a subtree) must be copied on every write. Tux2 solves this problem by deferring access to the freelist for most transactions; however, how and if this works is not well specified. The system as a whole seems very complex. Perhaps it is structured in this way because it is designed to be built on top of the ext2 system, which does not use the sort of block indirection that we have proposed. Nevertheless, there might be some performance gain to be had by structuring the filesystem as a tree, although the freelist problem would have to be solved to achieve this.

It might be possible to disperse the freelist over the tree. This would allow an important optimization to the system: on a write transaction affecting more than one node in the tree, only the least common subtree would have to be copied (as opposed to the entire path from each up to the root). The problem with doing this is that it is extremely difficult, if not impossible, to guarantee that a free block will be available anywhere near the area which you are modifying. In this case, an allocator might have search the whole tree in order to find a free block. The approach would have to be complex and is not yet guaranteed to provide much benefit; however, alternate ways of structuring the disk are important to consider when thinking of ways to optimize the system.

B. Data Structures

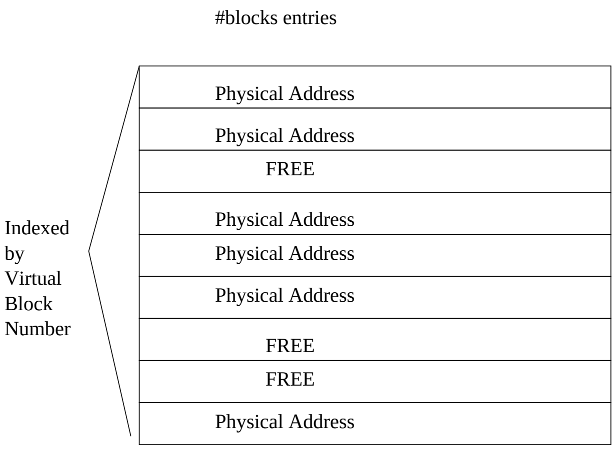

Fig. 2. Block Indirection Table

The data structures used in the filesystem are fairly simple. The freelist is a bit map which contains a bit for every physical block in the system. Allocated blocks are represented by zeroes and free blocks are represented by ones. The block indirection table, on the other hand, is indexed by virtual block number and contains within it either a pointer to the physical block or an indication that the address is unallocated (Figure 2). The length of the pointer is dependent on the number of blocks in the filesystem.

There is no notion of inode numbers distinct from block numbers in our system. An inode number is the same as a virtual block number. All inodes take up one block; if this is deemed inefficient, it is simple to pack inodes to use than a full block of space on disk by indexing inodes by virtual block number and some additional subblock index bits.

Inodes contain the following information:

- an *inode number*: this is the same as the virtual block number
- *mode*: specifies permission bits on the file
- *nlink*: a refcount of the number of links into a file; used for linking and deallocation purposes
- creation, access, and modification *times*
- owner and group *id numbers*
- *size*: the size of the file represented by this inode
- *blocks*: virtual block numbers of the data blocks containing the actual data for this file; 3 of these are pointers to indirect blocks rather than data blocks
- file *type*
- Other entries found in inodes from other filesystems; these are not currently supported by SFS.

The metaroot block contains a “pointer” to the committed indirection table, a checksum, and global filesystem information. The committed indirection table “pointer” is a single bit which indicates which of the two indirection tables (and correspondingly which freelist) was last committed. (The other set of tables is then the working set.)

Indirect block tables are handled as specified in “The Unix Time-Sharing System[3].” There are thirteen data block pointers, ten of which point to data blocks and three of which point to deeper levels of indirection of the system.

Directories are handled in the standard way[3]. Their data blocks contain entries which consist of a filename and an inode (virtual block) number. They are indicated by the type value in the inode.

C. Filesystem Operations

The higher level filesystem operations use four primary operations in the virtual layer. (The fifth, `virtual_commit`, is considered separately in the next section.) These are:

1. `virtual_alloc_block`
2. `virtual_free_block`
3. `virtual_read_block`
4. `virtual_write_block`

All of these operations mutate the working set of the metadata, leaving the committed set unchanged.

C.1 `virtual_alloc_block`

The virtual block allocation function returns a free virtual address to be used by the caller (which usually immediately writes to the block). It sequentially reads blocks of the indirection table and scans them for a free virtual address. If it finds one, it returns it; otherwise, if there is no free addresses (which can only occur if there is no disk space left), it returns an error.

C.2 `virtual_free_block`

This function marks virtual blocks as free. It takes a virtual address and calculates the proper offset into the block indirection table. It then reads in the correct block of the block indirection table, finds the physical block number corresponding to the block, and marks the physical block free in the working free list. The function then marks the corresponding virtual address as unallocated.

C.3 `virtual_read_block`

Reading blocks is done by calculating an offset into the block indirection table, looking up the physical address of the block, and then reading in that physical block. (In our implementation, reading from disk is simulated by a call into the underlying filesystem.)

C.4 `virtual_write_block`

The `virtual_write_block` function takes a virtual block number and a buffer containing the data to be written. It calculates the offset into the indirection table, reads that indirection block, and determines the physical block number currently corresponding to that virtual address. Since this physical block's data is conceptually in the committed set, we don't want to overwrite it. As a result, we instead mark the block free in the working set's indirection table and freelist. We then allocate a new physical block into which we will write the data; it is important to note that this block must be marked as free in **both** the committed and working sets' free lists, so that we don't overwrite data in the committed set. This requires us to read out blocks from both free tables and scan them for a block which is free in both. We then mark the block "not free" in the current *working* set. Finally, we write the data to the physical block on disk (or outsource this to the underlying filesystem as in the read case).

D. Higher Level Filesystem Operations

All of the filesystem operations (`read, write, symlink, mkdir, rename`, etc.) are implemented conventionally. It is important to note that they use exclusively virtual blocks numbers to read, write, free and allocate blocks; this preserves the integrity of the two sets of metadata. Operations with critical sections perform a commit when they are finished: these operations are `write, create, link, symlink, remove, rename, mkdir, rmdir,` and `setattr`.

E. Commit Algorithm

Changes to the system are committed as an atomic write(Figure 3); we do this simply by toggling the `committed_indirection_table` bit in the root block, recalculating the checksum, and then writing it back to disk. If the system were to crash before or after this point, we would still have a consistent filesystem because we would only load the committed set of data and build the working set from there.

After a commit, the rest of the system must be updated to make the system ready for the next commit. The new root must be copied into memory and the internal pointers to the indi-

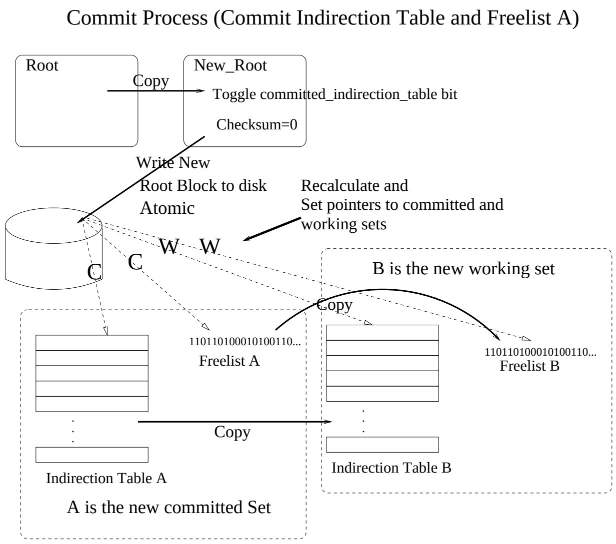

Fig. 3. Committing metadata set A

rection tables and free lists recalculated. Also, the newly committed free and indirection tables must be copied to the new working tables before the next commit happens; if this data were not up to date before the next commit, our filesystem would be left inconsistent. Currently, our update mechanism, simply copying over the entire set of tables, is quite inefficient. This is an area where our system would benefit by caching written data and only copying modified portions of the tables.

Currently our system performs commits whenever it is required to preserve filesystem integrity; this is after every filesystem operation which modifies the block indirection table or the freelist. An alternate scheme would be to commit at periodic intervals, perhaps once per second. This would improve the performance of writes by reducing the amount of data copying required.

F. Recovery: Initializing the System

When the system comes back up after a crash we check that something is not terribly awry by making sure the checksum in the root block is valid and that all parameters are within reasonable ranges. We then recalculate the checksum and proceed as though we had just done a commit (Figure 4).

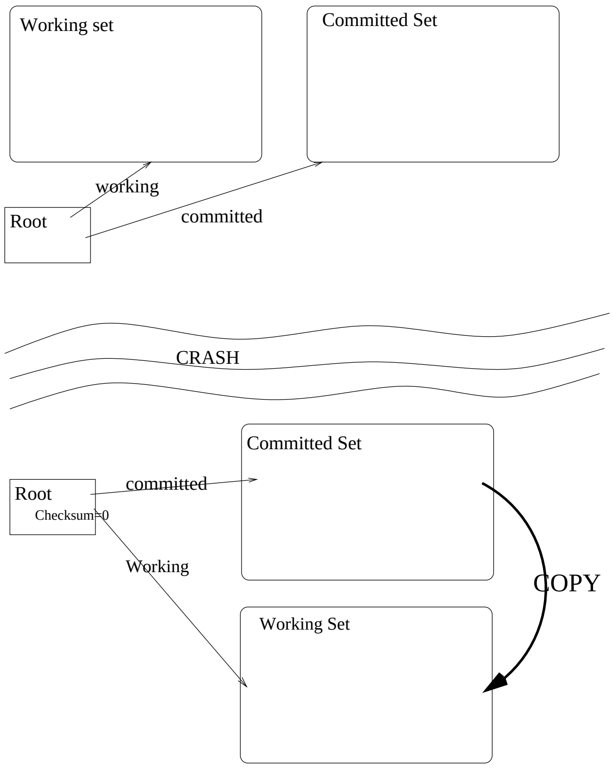

Fig. 4. System Recovery

It is unfortunate that we still have to recopy the entire filesystem indirection table on initialization; however, this is still considerably more efficient than reading through the whole disk and reconstructing the metadata, as a disk checker does. And, unlike these checkers, it is possible to allow normal filesystem operations to occur immediately while this copying is proceeding. The only cost might be somewhat slower performance and delaying any commits until the recover is completed.

G. Using Caching to Provide Better Performance

Although we have not implemented one, our system is designed to work with a write-back cache; this would greatly improve the efficiency of the system. In this filesystem, a cache would exhibit a good deal of location locality for reads and writes; this is different than a log-structured filesystem, which only exhibits temporal locality, and only on writes[4]. Caching the block indirection table, much like caching of a virtual memory page table, would reduce much of the inefficiency associated with the extra level of indirection. In addition, because we expect *all* of the updated working set to be in the cache (having just been written back) after a commit, the inefficiency incurred by the copying of data from the committed to working set can be minimized: we would only have to write back blocks which have changed. In many cases, there would be no copying required at all; for example, in the common case of a single data file or data region being repeatedly written, the same block will be changed repeatedly between commits.

VI. Implementation

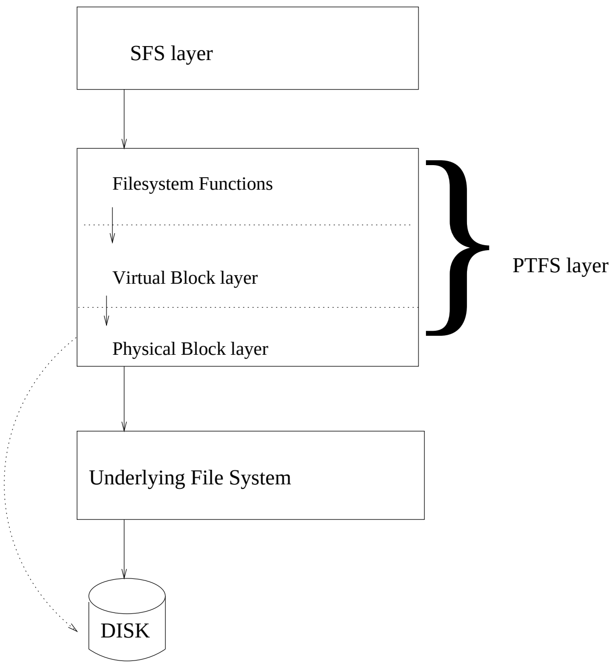

Fig. 5. Structure of the Implementation

The system we built is divided into several abstraction layers (Figure 5). The top layer basically takes the SFS methods, which were passed into the SFS filesystem by the client, and forwards them to our filesystem functions instead. The filesystem calls, then, use only the virtual block numbers, never seeing a physical block number at all. The virtual layer then calls `physical_read_block` and `physical_write_block` wrapper functions, which basically pass the requests along to our "disk", which is a file in the server's host filesystem.

A. Integration with SFS

We modified the SFS server to call into ptfs, but did not make any modifications to the SFS client. As a result, our system lacks certain features that the SFS client does not currently support. For example, SFS does not provide user and group IDs on file creation. As a result, all files created by the current SFS client default to ownership by root.

B. Project Status

PTFS works correctly in simple tests, but the code is by no means robust. There are several known issues that have not yet been resolved, such as handling "disk full" errors correctly. The system does function well enough that ptfs source code can be compiled on a ptfs filesystem, and the resulting binaries subsequently executed correctly.

VII. System Evaluation

There are several areas where the design concept for PTFS is different from the simpler version of PTFS actually implemented. The differences in the commit algorithm and how often it is invoked have substantial performance implications.

A. Performance

No attempt has been made in the current implementation to make write performance efficient; unsurprisingly, as a result, it is rather slow. The problem is that commits currently force the entire metadata structure to be copied: every write involves a copy of the entire block indirection table and freelist. This is hardly required by our design, however, and caching should help a great deal, reducing the number of extra writes to nearly zero. Implementing a cache is the logical next step to improve the system, because without this, it really is impractical.

That said, the majority of the writes done by the system should be done in continguous blocks, which ought to help performance. In addition, it would not be difficult to run a defragmenter in the background, due to the decoupling between virtual block addresses and physical block locations. It is important to note that commits, the metadata copy on initialization, and any defragmentation routines can run asynchronously. It should therefore be possible to achieve the same sort of fast write performance as in a logging filesystem. The extra lookup due to the block indirection table should make reads only slightly slower than in a conventional unix filesystem, if caching reduces indirection table misses to nearly zero.

Due to the mandatory recopying of the working set of data, recovery is not instantaneous. However, the set of metadata is much smaller than the entire disk, so our system has the potential to be much faster than a system using only a disk checker for recovery. Only about half a percent of the disk is required for the metadata which must copied on startup in our system, and some simple clever checkpointing techniques might reduce this further.

Assuming a 4 gigabyte filesystem with 512 byte disk blocks, there are 8,388,608 blocks. 32-bit pointers are used to address these blocks, so 65,536 blocks are required for each block indirection table and 2,048 blocks are required for each freelist. This results in 32 megabytes being copied on initialization. Since the data in these portions of the disk is stored contiguously, it should be possible to do this copying in a few seconds; this is perfectly reasonable for initialization after a crash, though unreasonable for routine writes.

In conclusion, while our rough implementation is slow, several improvements to the current implementation are possible and easy.

VIII. Conclusion

Perhaps the most important thing to be learned from this project is that keeping a log is not the only way to produce fast filesystem recovery and consistency across crashes. The phase-tree approach works and is an interesting solution to the problem. It merits further exploration. Our work in researching this problem suggests that implementing quick recovery procedures will inevitably cause some overall performance degradation to the system; the degree to which this can be minimized is uncertain, although it seems likely that performance can be made acceptable. What is certain is that adding the feature of fast crash recovery will increase the complexity of the system; the phase-tree approach feature to a system in as simple way as seems possible.

References

[1] Marshall Kirk McKusick et al., Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. USENIX 1999

[2] Mazieres et al., Separating key management from file system security, SOSP 1999

[3] Dennis M. Ritchie and Ken L. Thompson. The UNIX time-sharing system. *Bell System Technical Journal 57,6,* part 2 (1978) pages 1905-1930.

[4] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, California (October 13-16, 1991) pages 1-15. Published as *Operating Systems Review 25,5*(1991). Also appeared in *ACM Transactions on Computer Systems 10*, 1 (February, 1992) pages 26-52

[5] Sweeney, Scalability in the XFS File System, USENIX Winter 1996

[6] Tux2, http://innominate.org/ phillips/tux2/(2000)

[7] VERITAS Software Corporation: File System White Paper, (November 1996)