A Cached WORM File System

Sean Quinlan†

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a general-purpose file system that uses a write-once-read-many (WORM) optical disk accessed via a magnetic disk cache. The cache enables blocks to be modified multiple times before they are written to the WORM and increases performance. Snapshots of the file system can be made at any time without limiting the users' access to files. These snapshots reside entirely on the WORM, are accessible to the user via a second read-only file system, do not contain multiple copies of unchanged data, and can be used to rebuild the file system in the event that the disk cache is destroyed. The file system has been implemented as part of Plan 9, an experimental operating system under development at AT&T Bell Laboratories.

Keywords

File systems   Caches   WORM   Backup

Introduction

Write-once optical disk technology is an excellent medium for data backup and archival. Write-once optical disks have a price per bit similar to magnetic tapes, allow efficient random access to large volumes of data, and are more compact. The advent of jukeboxes, such as the fifty-disk Sony Writable Disk Auto Changer, now makes it feasible to hold hundreds of gigabytes on-line.

Write-once optical disks belong to the class of write-once-read-many (WORM) devices. A particular block on a WORM can be written only once but can be read many times. Conventional file systems are designed under the assumption that blocks on storage devices can be written more than once. This difference poses the question of how a WORM device can be used to advantage in a file system.

The File Motel [1] uses a WORM to store backup copies of the files of conventional file systems. A separate database, which resides on magnetic disk, is used to find the files on the WORM. The integrity of the WORM data is ensured by the ability to reconstruct this database if it is corrupted. This approach works well, but a file system is not

†Current address is Computer Science Department, Stanford University (sean@neon.stanford.edu).

backed-up in an atomic operation. Copying all files from the disk file system to the WORM requires a substantial period of time and unless the disk file system is unmounted while this occurs, one can still modify files. The result is a backup that is "smeared" over time.

An alternate approach is to place a general purpose file system directly on a WORM, taking into account the restriction that blocks can be written only once. One example is the Optical File Cabinet [2,3] which uses a block replacement strategy. The file system sees a logical address space that is much smaller than the size of the WORM device. Logical block addresses are mapped to physical WORM addresses; each time a logical block is written, the mapping is changed so that an unwritten physical block is used, i.e., a logical block can be written multiple times.

The Optical File Cabinet has the advantage that the file system is unchanged and requires only a different device driver. However, a WORM block is allocated for every write operation and so the logical address space must be much smaller than the physical address space. This allocation can be reduced by the use of a cache, but the consumption of WORM blocks is still high. Optical disks are still considerably slower than magnetic disks so the performance of the Optical File Cabinet compares poorly with a conventional file system.

This paper describes a novel approach for integrating a WORM device into a file system. As with the Optical File Cabinet, the file system resides on the WORM. However, the file system uses the address space of the WORM, carrying out reads and writes to WORM block addresses. A general purpose file system is presented to the user; the write-once nature of the main storage medium is hidden by using a copy-on-write scheme.

When a block is written, the data is not transferred directly to the WORM; instead, it is cached on a magnetic disk, the WORM cache, with the result that certain blocks can be written multiple times and the consumption of WORM blocks is reduced. The cache also masks the relatively slow performance of the optical disk.

In addition, the backup system is unusual. At the user's request, the file system freezes activity and flushes the WORM cache, thus ensuring that all data is on the WORM. A consistent snapshot of the file system appears within seconds in a parallel read-only file system, which can be accessed using standard file utilites. From the user's perspective, the entire backup procedure takes a matter of seconds.

Block States

The WORM cache contains a subset of blocks in the file system address space. Each block in the cache has an associated state, one of *free*, *write*, *dump*, or *read*. There is a pseudo state *notfound* for all blocks that are not in the cache but they are conceptually *read* blocks. The states have the following properties:

free: The block is only in the cache and has not been written to the WORM. The block contains no data and is not in use in the file system.

write: Only in the cache. The block can be read and/or written: read-write.

dump: Only in the cache. The block is read-only.

read: On the WORM and in the cache. The block is read-only.

Blocks are created in the *free* state. The transition to the *write* state occurs when a block is incorporated into the file system tree and used to store data. A block can return to the *free* state if the block is subsequently not needed, for example when a file is deleted.

The dump command flushes the cache and results in a consistent and complete snapshot of the file system residing on the WORM. The transition from *write* to *dump* occurs when the dump command is executed, representing the semantic transition from read-write to read-only. The *dump* state is only transient; the transition from *dump* to *read* occurs when the block is written to the WORM by a background daemon.

Figure 1. gives the state transition diagram for a block.

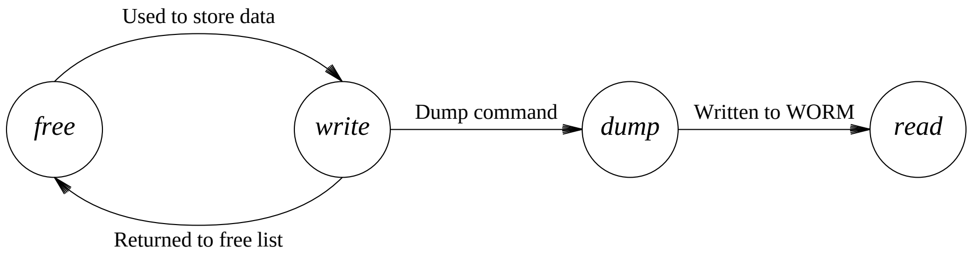


Figure 1. The state transition diagram for a block.

The File System

The file system provides a structure similar to the UNIX® file system. There is a hierarchy of directories and ordinary files which are accessed by a set of system calls resembling open, close, read, write, etc. Like the UNIX file system, blocks are accessed via a pool of buffers that reside in memory and have the effect of caching both read and write operations. When a block is referenced that does not reside in the buffer pool then I/O operations are required.

I/O operations do not occur directly to the WORM, but instead take place via the WORM cache which resides on a magnetic disk. The WORM cache consists of blocks that are read-write or read-only depending on a block's current state. The file system consists of a mixture of both read-write and read-only blocks. In contrast, a conventional file systems assumes that all blocks are read-write.

The implementation of the file system hides the fact that certain blocks are read-only. The user sees a general purpose file system that has no restrictions, other than the usual security restrictions, on which files may be modified. The WORM and the cache are treated as a special device, a cache WORM device, that contains both read-write and read-only blocks. A conventional file system was modified to handle the presence of read-only blocks by using a copy-on-write scheme. This scheme is described in the remainder of this section.

The file system consists of two elements: a set of files that are either directories or ordinary files and a list of *free* blocks. The free list is discussed in the next section; for now, let us assume a *free* block can be obtained as needed.

Directories are files of directory entries. A directory entry contains information about a file such as its name, owner, permissions, etc, and a list of block addresses containing data for the file. In our implementation, this list has a more complicated structure for random access to large files but, for the purpose of this discussion, we will consider the list to consist of a simple array of addresses. The extension to more complex structures is not difficult.

The directory hierarchy is a true tree, i.e. there are no hard links and no blocks are shared between files. A root block contains the directory entry for the root of the tree and all blocks can trace a path from this root. The address for a block is stored in exactly one location, namely the directory entry of the file in which the block is contained. If a block is to be replaced by a different block then only this one address need be changed.

When a file is opened, the system walks the path from the root to the file. At each step in the walk, a current directory entry in either a *dump* or *read* block is moved to a *write* block. This move is achieved as follows:

- A *free* block is marked as *write*.
- The data in the old block is copied to the *write* block.
- The pointer to the old block, which resides in the parent directory, is changed to point to the *write* block. The pointer can be changed as the root is always maintained in a *write* block so, by induction, the parent directory is in a *write* block. The result is that the directory entry of all open files is contained in a *write* block and can thus be modified.

After a file is opened, a copy-on-write scheme is used to enable the file to be modified. If an attempt is made to write to either a *dump* or *read* block in the file then a *write* block is allocated and substituted using the same procedure as when the file was opened. This substitution is possible since the directory entry for the file is contained in a *write* block. Figure 2. gives an example of opening then modifying the file `/etc/passwd`.

A *write* block is allocated when an attempt is made to write to a *dump* or *read* block. The structure of the file system is used to determine the location of the pointer which needs to be changed to reflect the substitution of the *write* block. The pointer itself is guaranteed to reside in a *write* block.

The Dump Command

One of the main goals of this file system is to provide a convenient and efficient means of recording snapshots of the entire file system. These snapshots reside entirely on the WORM, which is both reliable and write-once; there is no need to backup the file system to an external medium. Moreover, as the snapshot resides on the same device as the file system, blocks can be shared between both, and even with previous snapshots. This sharing reduces the storage requirements for snapshots of the entire file system to the equivalent of incremental backups.

To create a snapshot, the dump command is executed. The snapshot of the file system is accessed as a directory in a second read-only file system called the dump. The user can mount this second file system and access files in the standard manner, although write operations are not permitted. Utilities such as the UNIX commands `cp`, `grep`, `diff`, `ls`, etc, can be used to examine and restore information contained in the snapshots. The root of the dump file system contains a directory entry for each snapshot taken and thus the user has a complete history of backups, available online, and accessible in a familiar fashion. It should be noted that the original file permissions apply, so no

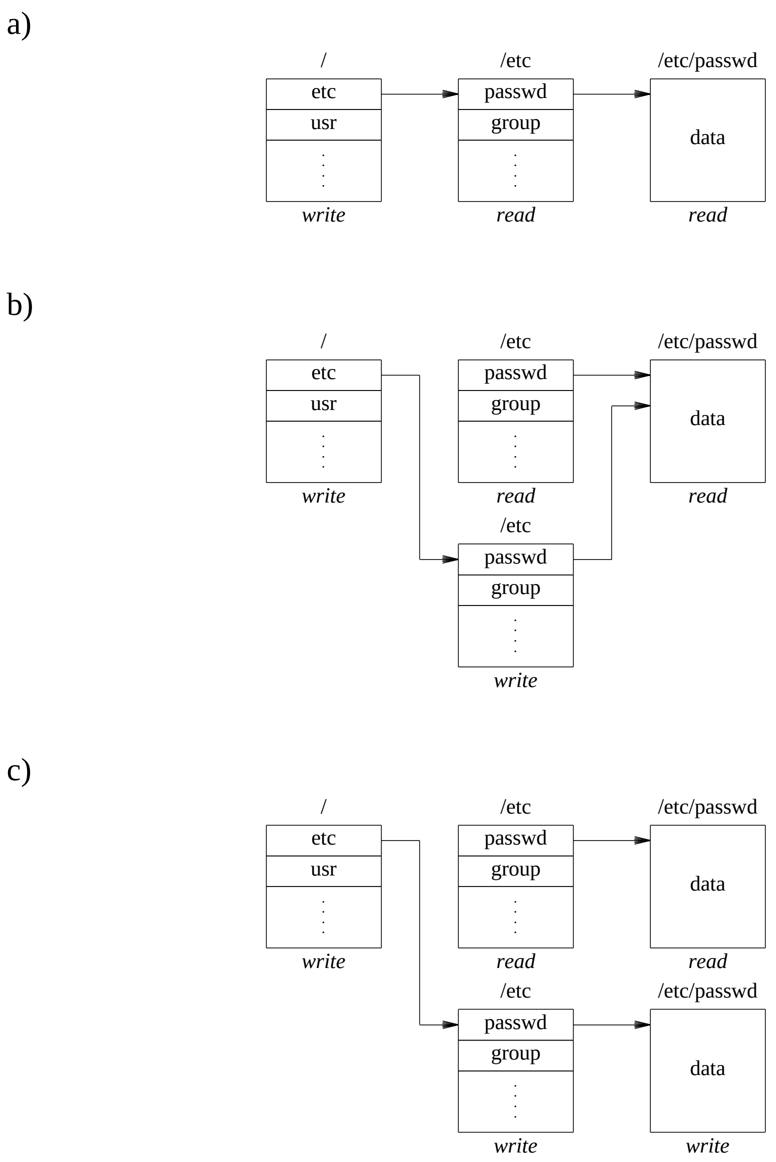


Figure 2. Opening then modifying `/etc/passwd`.

special arrangements need to be made for access to the dump; one cannot use the dump to subvert security. However, the dump cannot be changed; if a snapshot is made at a time when a sensitive file is temporarily readable then the file will remain accessible forever.

The dump file system is read-only; the copy-on-write scheme used in the main file system is not needed as there are no write operations. It follows that the invariants used to implement copy-on-write can be relaxed. In particular, the directory entry for an open file is not moved to a *write* block and the file system structure is a graph rather than a tree. A block that remains unchanged between dumps will appear more than once in the dump file system even through it appears only once on the WORM.

The dump occurs in three stages: First, user activity is frozen and the WORM cache is flushed. Second, the dump file system is modified to include a snapshot of the file system. Third, the file system is returned to a state from which normal operation can continue.

The file system is placed in a consistent state by completing all current file operations and freezing all further activity. Any dirty memory I/O buffers are written to the WORM cache and then the WORM cache is flushed. To flush the WORM cache requires all *write* blocks to be written to the WORM, a time consuming process. To hide this time from the user, *write* blocks are converted to the transient *dump* state; no WORM I/O is performed. This change of state represents the semantic transition from read-write to read-only. A background daemon, described below, writes *dump* blocks to the WORM, completing the final transition to the *read* state.

Flushing the WORM cache has the effect of making all blocks in the file system read-only, i.e., a snapshot has been taken. The root of the file system provides an access point to the snapshot.

A pointer to the root block is stored as a new entry in the root directory of the dump file system. The new entry has a unique name generated from the time that the dump command was issued. To create an entry, the dump file system must be modified, requiring the root of the dump file system to be moved to a *write* block. This modification is a special case where the dump file system is not read-only and results in a new root address for the dump file system.

The final step is to return the file system to a state from which normal operation can continue. The root of the file system is moved to a *write* block. Also, the path from the root to each open file is walked and the associated directory entries are moved to *write* blocks. Modifications to the file system can now be handled as discussed in the previous section. Note that changes to the file system are only visible through the new root block; the file system remains unchanged when viewed through the old root. In effect, the snapshot and the file system will split apart as the file system is modified. Figure 3. illustrates what the file system might look like before and after a dump. The file `/etc/passwd` is assumed to be open.

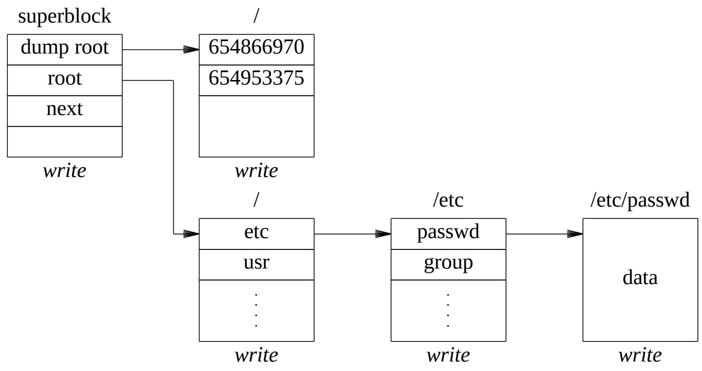
The dump is performed as an atomic operation; no changes to the file system are permitted during the procedure. In our current configuration, the dump command takes about ten seconds. The file system is inaccessible for these ten seconds but this is acceptable for us, especially as dumps are generally performed only at night.

The majority of time spent executing the dump command involves marking the *write* blocks as *dump*. The search of the WORM cache takes constant time determined by the size of the cache, currently 122 megabytes.

The other substantial task of the dump command is rewalking the path for each open file. The rewalk is necessary to ensure that the minimum number of directory entries are in *write* blocks. A large number of open files could increase the period in which the file system is inaccessible.

After the dump command, the WORM cache contains *dump* blocks that have not been written to the WORM. A background daemon copies *dump* blocks to the WORM and marks them as *read*. This copying is entirely transparent to the user. As the daemon simply converts blocks from the *dump* to *read* state the dump command can be executed again while there are still *dump* blocks in the WORM cache.

a)



b)

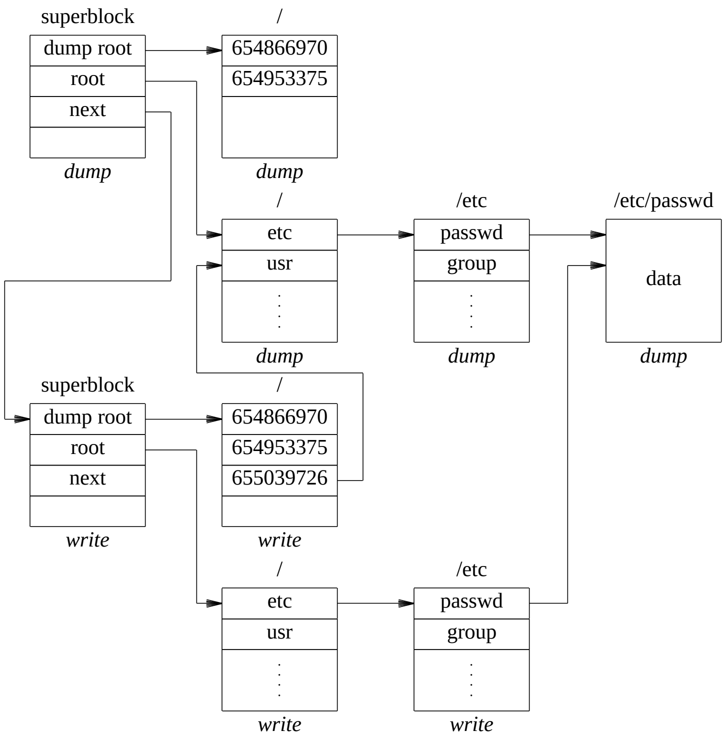


Figure 3. Before and after the dump command, assuming `/etc/passwd` is open.

The time to write all *dump* blocks to the WORM is expected to be similar to an incremental backup of a regular file system to a similar performance media. First, the number of blocks to write is minimal as any block that has not changed since the previous dump will be shared in both snapshots and need not be written. Second, the background daemon competes for access to the WORM only when a miss occurs in the WORM cache, a rare event, and this conflict is resolved in a fair fashion to avoid starvation.

Recovery

The reliability of the WORM would be negated if the file system could not be recovered in the event that the WORM cache is destroyed. This system can recover from such a failure. If the WORM cache is destroyed then only modifications made after the most recently completed dump are lost.

When the dump command is executed, the address of the root of the file system and the address of the root of the dump file system changes. The root addresses are stored with other information in the superblock. Before all *write* blocks are marked *dump*, a superblock is allocated. The address of the new superblock is stored in the old superblock; the superblocks are thus linked together.

The first superblock on the WORM is at a known address. The list of superblocks can be traversed until the last superblock is reached, detected by the presence of a link to an unwritten block. To restore the system, the WORM cache is initialized and both the superblock and the root of the file system are moved to *write* blocks. There is one small complication for the case in which the WORM cache is destroyed while there are blocks in the *dump* state. In this case, the most recent dump has not completely been written to the WORM and a previous dump must be used instead.

The Free List

The free list is used to maintain a record of *free* blocks. The head of this free list is contained in the superblock. When a file is deleted, all *write* blocks in the file are marked *free* and added to the free list, requiring examination of the state of each block. This examination imposes a small but noticeable overhead for large files.

The superblock contains the current size of the file system, i.e., the maximum block address which represents the high water mark on the WORM. When the free list becomes empty the high water mark is increased; in effect a number of blocks are created. These new blocks are placed in the cache as *free* blocks and added to the free list. Increasing the size of the file system is limited by two factors. First, the file system can not be larger than the total size of the WORM device; when this limit is reached the file system is full. Second, the WORM cache can only contain a certain number of *free*, *write*, and *dump* blocks; when this limit is reached the WORM cache must be flushed, using the dump command,

The WORM Cache

All WORM I/O is performed through the WORM cache which resides on a magnetic disk. The cache has two purposes: First, write operations are only performed on blocks that reside in the cache and have not been written to the WORM; multiple writes to a fixed address are thus possible. Second, the WORM device that we use, a Sony WDD-2000, is considerably slower than magnetic disk; caching WORM reads increases performance.

I/O operations are preceded by the translation of the WORM block address to a cache address, using the cache map, and the cache address is then used as the location for the I/O operation. In the case of a read operation, the block may not reside in the cache causing a cache miss and a read from the WORM device.

The cache map consists of a table of cache entries which each contain a WORM address and a block state. A cache entry with a non-zero WORM address implies that the block resides in the cache and is in the given state. The data for the block is in a cache data block which is associated with the cache entry. The cache map resides on the disk together with the actual data blocks and each block of the cache map contains an array of d cache entries where d is constant. If there are w blocks in the cache map then there are wd cache entries, which correspond to wd cache data blocks, and $w + wd$ blocks on the disk. For our implementation we use a 122 Megabyte disk made up of 2K blocks for which $d = 255$ and $w = 243$ are suitable values.

The cache is d-way set associative: to translate a WORM address x, the block (x mod w) of the cache map is searched. A block in the *free*, *write* or *dump* state is always maintained in the cache, hence the cache-entries for such blocks are *fixed*. As the cache is set associative, the situation can arise in which all entries in a cache map block are fixed and no new blocks can be added. However, blocks are placed in the cache in only two situations: when creating a *free* block and when reading from the WORM. In the first case, if the *free* block cannot be placed in the cache then it is ignored and as a result a WORM block is wasted. In the second case, if the block is not placed in the cache then a WORM read will be performed every time the block is accessed and the performance of the system will be reduced.

We would like a high probability that no cache map block will contain only fixed entries. To achieve this, the number of fixed entries is limited to some fraction α of the size of the cache. If we assume that such entries have unique addresses with a uniform probability distribution, then the probability that none of the cache map blocks is full of fixed entries is given by the function $P(w, d, n)$ where:

w is the number of blocks in the cache map.

d is the number of entries in a block.

n is the number of fixed entries in the cache, i.e. $n = \alpha wd$.

$$P(w, d, n) = \begin{cases} 1 & \text{if } n < d \\ 0 & \text{if } n \geq d \end{cases} \quad \text{for } w = 1$$

and

$$P(w, d, n) = \sum_{i=0}^{d-1} \binom{n}{i} \left(\frac{1}{w}\right)^{i} \left(1 - \frac{1}{w}\right)^{n-i} P(w-1, d, n-i) \quad \text{for } w > 1$$

For our implementation with $d = 255$, $w = 243$, and $\alpha = 0.75$, we get a value for $P(w, d, \alpha wd)$ of about 99.9%. Thus, if a dump is not preformed until the fraction of fixed entries reach α, we can still expect, with 99.9% probability, that no cache map block will contain only fixed entries.

In the rare case that a cache map block does become full of fixed entries, the creation of *free* blocks may fail. Every failure causes one block on the WORM to be wasted since it is never added to the free list and hence never used. The expected number of allocation failures, E, is given by the formula

$$E = \frac{\displaystyle\sum_{i=d+1}^{n} (i-d) \binom{n}{i} \left(\frac{1}{w}\right)^{i} \left(1 - \frac{1}{w}\right)^{n-i}}{\displaystyle\sum_{i=d+1}^{n} \binom{n}{i} \left(\frac{1}{w}\right)^{i} \left(1 - \frac{1}{w}\right)^{n-i}}$$

In our implementation E is about 4 thus we expect to waste about 4 blocks once every thousand dumps. We consider this an acceptable rate given that it allows a simple solution for the hash collision problem.

The round-robin algorithm is used to determine which of the possible entries in a cache map block is used when adding a block to the cache. The extra complexity and storage requirements to implement other algorithms such as LRU did not appear justified, however, future performance analysis of the file system may prove this incorrect.

Finally, the overhead in accessing WORM blocks via the WORM cache is minor. The majority of accesses are expected to be satisfied by the I/O buffers, which act as a first level cache in memory. For a WORM block, the WORM address is used to check these buffers and no examination of the WORM cache is needed. If a block is not in a buffer then the WORM cache is checked by searching one block of the cache map; a linear search of d entries. The cache map blocks are also accessed via the buffers and will generally reside in memory. A cache hit results in I/O to the WORM cache disk, while a cache miss requires a read from the WORM. In either case, the cost of the I/O should outweigh the cost of searching the WORM cache. Note that as the size of the WORM cache is increased, the value of d can be held constant, although α may need to be adjusted.

Implementation

The file system has been implemented on a VAX-750 used as a dedicated file server for Plan 9, an experimental operating system under development at AT&T Bell Laboratories. The implementation consists of about 7000 lines of C. The system is configured with 6 megabytes of memory for I/O buffers, 122 megabytes of disk for the WORM cache, and a single 1.5 gigabyte write-once optical disk (Sony WDD).

In the near future the system will be moved to a MIPS 6280 with 100 megabytes of I/O buffers, 1 gigabyte of WORM cache, and a 300 gigabyte Sony Writable Disk Auto Changer (a jukebox). The jukebox will be regarded as a single device and accessed via the one file system.

This file system was not designed for the UNIX operating system; however, the structure is similar to the UNIX file system with the exception that hard links are not possible. This restriction is not severe as soft links are generally an alternative and Plan 9 uses a different approach for file aliasing. The current file system can be accessed transparently as a remote file sever from machines in our laboratory that run the UNIX operating system.

Future Work

One of the major issues that has not been addressed is how to deal with the finite size of the WORM device. Each time the dump command is executed, some WORM blocks are irrevocably written. A problem arises when the WORM becomes full.

Our current solution is to assume the WORM has infinite size; we have yet to fill one of our optical disk jukeboxes. Over the three years that we have used optical disk jukeboxes, the density of the disks has quadrupled. Although we consume WORM disks at an exponential rate, the technology is improving even faster.

Even if the WORM does become full, it is expected that the size of the file system will still be considerably smaller that the size of the WORM. The WORM will contain many blocks that are not accessible via the file system, e.g., blocks from files that can only be accessed via the dump file system. It may be possible to move the file system to a second WORM media, enabling the inaccessible blocks to be reused.

Conclusion

The described method of utilizing a WORM device is both general and efficient. Users are provided with a standard file system which can be larger than the available magnetic disk resources. Snapshots of the file system can be taken at regular intervals with almost no inconvenience to the user. The snapshots reside entirely on the WORM and are accessed as a regular, albeit read-only, file system. In the event that the WORM cache is destroyed, the file system can be restored to the state of a previous snapshot. Finally, if the working set of the file system is smaller than the WORM cache, then little or no performance degradation is noticed by the user.

Acknowledgements

Ken Thompson suggested the idea of using the WORM address space for the file system and the algorithm for the dump command. He also provided much help with the implementation. This paper was improved by comments from Steve Bellovin, Tom London, Ross Quinlan, Ken Thompson, and two anonymous reviewers.

References

[1] Hume, A., *The File Motel - An Incremental Backup System for Unix,* Summer Usenix Conference Proceedings, 1988, pp 61-70.

[2] Gait, J., *The Optical File Cabinet: A Random Access File System for Write Once Optical Disk,* IEEE Computer, May 1988.

[3] Laskodi, T., Eifrig, B., Gait, J., *A UNIX File System for a Write-Once Optical Disk,* Summer Usenix Conference Proceedings, 1988, pp 51-60.