Union Mounts in 4.4BSD-Lite

Jan-Simon Pendry
Sequent UK

Marshall Kirk McKusick
Author and Consultant

ABSTRACT

This paper describes the design and rationale behind union mounts, a new filesystem-namespace management tool available in 4.4BSD-Lite. Unlike a traditional mount that hides the contents of the directory on which it is placed, a union mount presents a view of a merger of the two directories. Although only the filesystem at the top of the union stack can be modified, the union filesystem gives the appearance of allowing anything to be deleted or modified. Files in the lower layer may be deleted with whiteout in the top layer. Files to be modified are automatically copied to the top layer.

This new functionality makes possible several new applications including the ability to apply patches to a CD-ROM and eliminate symbolic links generated by an automounter. Also possible is the provision of per-user views of the filesystem, allowing private views of a shared work area, or local builds from a centrally shared read-only source tree.

1. Background

Sun's Translucent Filesystem (TFS) [Hen90a] provides a method for viewing the contents of a list of directory trees as if they were unioned together. The desire was to provide a filesystem-level interface to the problem of source code control. TFS follows on from work done at Bell Labs such as the 3-D Filesystem [Kor89a] by providing automatic data copying from one directory to another as new source versions are created and edited. TFS was originally implemented entirely outside the kernel but poor performance led to the addition of special support code inside the kernel.

The Plan 9 implementation of *union mounts* is fundamental to the way that system mounts are implemented. A *before*, *after*, or *replace* option can be given to the mount system call. The union operation only takes place at the mount directory itself as compared to the union filesystem that merges the namespace all the way down the directory tree. Similar semantics were chosen for an earlier version of the union filesystem but proved too restrictive for general purpose use in 4.4BSD.

The Spring operating system [Kha93a, Mit94a] makes use of a fully object-oriented implementation technique along with a well-defined set of object methods. Spring supports an implementation of stackable filesystems without requiring special support within the filesystem code itself, purely by taking advantage of the object invocation mechanism and the callbacks that can be used to provide a cache coherency protocol. While these mechanisms appear superior to the vnode stacking architecture in 4.4BSD-Lite, they do not, by themselves, provide any of the namespace and copy-up functionality of the union filesystem.

Other systems such as DOS and VM/CMS provide a way of *appending* a directory to an existing directory such that the contents of both are visible at a single place. Both systems limit appending to a single level of namespace.

2. Introduction

The 4.4BSD kernel vnode architecture is similar in spirit to the original Sun design [Kle86a]. As the demand grew for new filesystem features, it became desirable to find ways of providing them without having to modify the existing and stable filesystem code. One approach is to provide a mechanism for stacking several filesystems on top of each other [Ros90a].

The stacking ideas were refined and implemented in the 4.4BSD system [Hei94a].

The traditional filesystem mount mechanism in 4BSD provides a way to attach complete filesystems to directories in the system namespace. Namespace construction is flexible but the limitation of mounting only complete filesystems can cause some problems in disk space management.

The namespace of a system is constructed from the namespaces within each individual mounted filesystem. Each file system is mounted at a place chosen by the system administrator and many filesystems may be created to lay out the namespace in a controlled fashion. This organization can lead to a conflict between allocating the available disk space to filesystems (wanting to allocate all the disk space to a single filesystem) and the desire to build the most easily used and managed namespace (wanting to create many small filesystems). Earlier releases of BSD did not provide any relief from this conflict. The null and union filesystems allow a new approach to disk and namespace management; they allocate all the disk space to a single filesystem, then attach sub-trees of the filesystem to the namespace as required. This approach allows the disk space to be shared between uses without need of manual intervention by a system administrator. In addition, the namespace in the system can be created and managed as required by users and applications.

4.4BSD provides two filesystem namespace management mechanisms: the null filesystem and the union filesystem. The null filesystem provides a simple flat namespace attachment mechanism, and the union filesystem provides more dynamic hierarchical functionality. These two mechanisms and some possible applications are described below.

3. Null mount filesystem

The *null mount* filesystem is a stackable filesystem in 4.4BSD that allows arbitrary parts of the filesystem tree to be mounted anywhere else in the system namespace. This functionality can be used to glue together several directories to form a new directory tree. One use of this functionality would be to rearrange filesystem trees on multiple disks into one managed tree that is presented to users. Alternatively, the new mount point could be the same as the original directory. This approach allows a sub-tree of a writable filesystem to be made read-only.

The null filesystem implements namespace attachment by attaching the original directory beneath the new mount point. With one exception, all filesystem operations are intercepted and directed to the original filesystem. The exception is the fetch file attribute operation (`VOP_GETATTR`) that modifies the returned data to reflect the namespace of the new mount point (in particular, ensuring that ***getcwd***(3) and ***df***(1) function correctly).

The null filesystem continues to obey the traditional semantics whereby a mounted filesystem hides the files already present in the directory on which it has been mounted.

4. Union mount filesystem

The *union mount* filesystem is a conceptual extension of the null filesystem. The difference is that the union filesystem does not hide the files in the mounted on directory. Instead, it *merges* the two directories (and their trees) into a single view. The union filesystem allows multiple directory trees to be simultaneously accessible from the same mount point. This functionality is similar to that described in [Kor89a]. Sun's TFS uses an entirely different mechanism to provide some of the union filesystem functionality.

At mount time, directories are *layered* above or below the existing view. This structure is called a *union stack*.

4.1. Union mount semantics

A mount stack has a physical and a logical ordering. The physical ordering is the temporal order in which directories were mounted. The last directory mounted in a stack must be the first to be unmounted. Restricting unmounts to the last mounted directory is because of historical Unix semantics that have been carried over into the 4.4BSD-Lite VFS/vnode data structures. If this restriction were to be relaxed then a mechanism for naming lower-layer filesystems would be required. One possibility would be to use an *escape* from the namespace such as the 3-D filesystem's '`...`' which provides access the next lowest layer [Kor89a].

The logical ordering is that seen by filesystem namespace operations. Directories can be mounted either logically at the top or the bottom of the existing view. The logical ordering will be the same as the physical ordering whenever directories are added from above. It will differ from the physical ordering when a directory is logically mounted below the existing view. When accessed through the mount stack, all but the topmost logical layer are read-only.

A union stack where the logical and physical ordering is identical can be thought of as a sequence of lazily evaluated `cp -rp` commands.

4.2. Namespace operations

The union filesystem operates on the filesystem namespace. Special actions are required whenever a request is made to change the namespace of the lower layer, for example using commands like `mv`, `rm` or `chmod`. Since the lower layer is read-only, the union filesystem can only make changes in the upper layer. This restriction leads to several special cases:

Copyup operations

> Whenever file contents or attributes are changed the target file is copied to the upper layer and changes are made there. To maintain the lazy copy semantics, the new copy of the file will be owned by the user who did the original mount, not the user who triggered the *copy-up* operation. In addition, the umask at the time of the mount is applied, not the umask at the time of the copy-up. Operations that cause a copy-up include `link`, `chmod`, `chown` and open for write. See section 4.5.
>
> Copyup operates on only one name at a time. Other links to the same file are not copied and so the link count in the upper layer will be incorrect.

Whiteout operations

> If a name is being removed from the lower layer's namespace a *whiteout* is created in the upper layer. A whiteout (see below) has the effect of masking out the name in the lower layer. Operations that cause a whiteout to be created include `unlink`, `rmdir` and `rename`.

4.3. Union naming

A directory listing of a mount stack shows all the files in all the directories involved in the stack (figure 1).

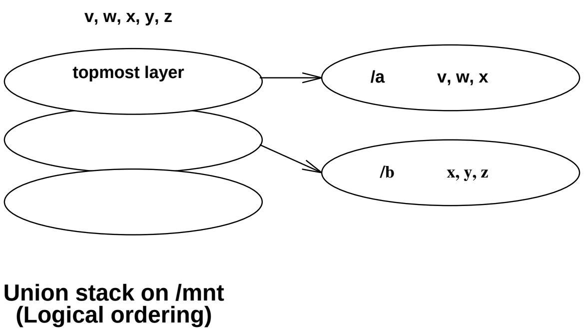

Figure 1: *Stack view*

In this example, `/b` is layered over an empty directory `/mnt`, then `/a` is layered over the new view of `/mnt`. Duplicate names are suppressed so that only one occurrence of '`x`' (and also '`.`' and '`..`') will appear.

A name lookup will locate the logically topmost object with that name. The lookup progresses down the mount stack in the logical stacking order doing a lookup at each layer and generating a list of vnodes corresponding to each layer. In figure 2 the name '`x`' will be found in the layer mounted from `/a` whereas the name '`y`' will be found in the layer mounted from `/b`. The upper or lower layer vnodes could themselves be union vnodes whenever more than two layers were in the stack or if two union stacks were joined.

Two special cases occur during lookup.

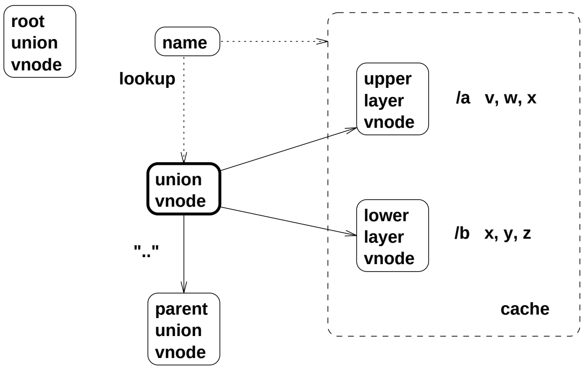

Figure 2: *namespace*

4.3.1. Lookup for `..`

A lookup for '`..`' is treated as a special case. The need for this special case is illustrated in figure 3. The current directory is `/tmp/fred`, a directory that exists in the upper layer, but does not have a corresponding directory in the lower layer. When looking up '`..`', the process wants to reference `/tmp` which does have corresponding directories in both layers. If the union vnode for the `/tmp` directory had fallen out of the cache, then it would need to be reconstructed. To do the reconstruction would require finding the upper and lower level vnodes that the union vnode must reference. The `/tmp` directory for the upper layer could easily be found by looking up the '`..`', entry for `/tmp/fred`, but there is no way to find the `/tmp` directory for the lower layer since there is no directory in which to lookup a '`..`' entry. To avoid this dilemma, each union directory vnode maintains a reference to its parent ('parent union vnode' in figure 3).

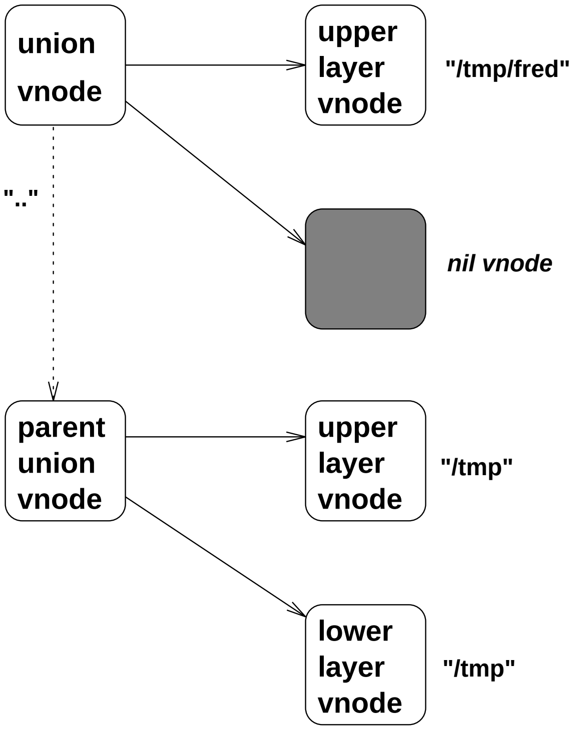

Figure 3: *looking up* `..`

4.3.2. Directory lookup

When a directory is looked up in a lower layer and there is no object with the same name in the upper layer, the union filesystem automatically creates the corresponding upper layer directory, known as a *shadow* directory. These semantics have the side-effect of populating the upper layer directory tree as the union stack is traversed (a `find` will cause it to be fully populated). The upper layer directory is created to ensure that a directory exists should a copy-up operation be required. The copy-up needs a directory in which to create the upper layer file.

The shadow directories could be created at the time of the lookup or at the time of the copy-up. By creating shadow directories aggressively during lookup the union filesystem avoids having to check for and possibly create the chain of directories from the root of the mount to the point of a copy-up. Since the disk space consumed by a directory is negligible, creating directories when they were first traversed seemed like a better alternative. Within the union stack the existence of additional directories is transparent to the user.

4.4. I/O operations

Having located a name, a sequence of filesystem operations is executed. As an example, consider a read I/O request (figure 4). Here, a file with a matching name exists in the lower layer, but not in the upper layer. So, the read I/O operation will be directed to the lower layer vnode.

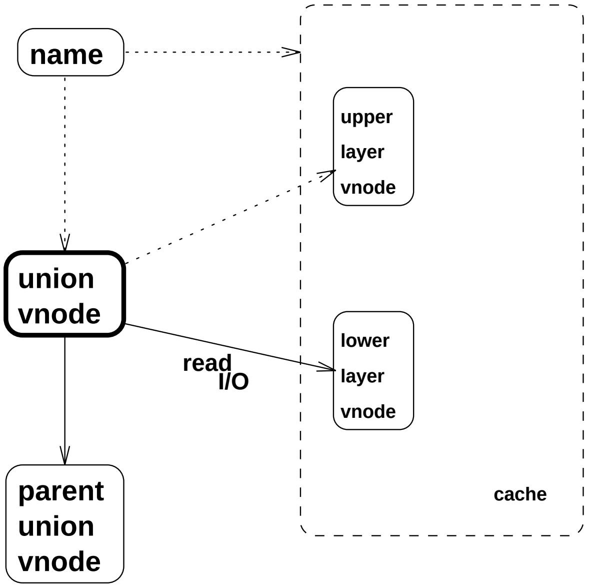

Figure 4: *Read I/O*

Now suppose that the file in the lower layer was opened for writing. Since the file is in the lower layer, it cannot be modified (only files in the topmost layer may be modified). Instead the union filesystem automatically copies the file to the upper layer and continues the open operation on the new copy. This operation is known as a *copy-up* (see figure 5). The underlying copy remains unchanged and is not deleted.

4.4.1. Cache coherence

4.4BSD-Lite does not provide a vnode cache coherence protocol. A cache coherence protocol is needed, at least in principle, to support stackable filesystems such as a compression filesystem. Other research operating systems such as Spring provide a more elegant and complete stacking mechanism that includes provision for full cache coherence mangement [Kha93a, Kha93b].

It was not within the scope of this work to design and implement a suitable cache coherency protocol for 4.4BSD-Lite. So, the current union filesystem implementation ignores the problem of detecting changes in

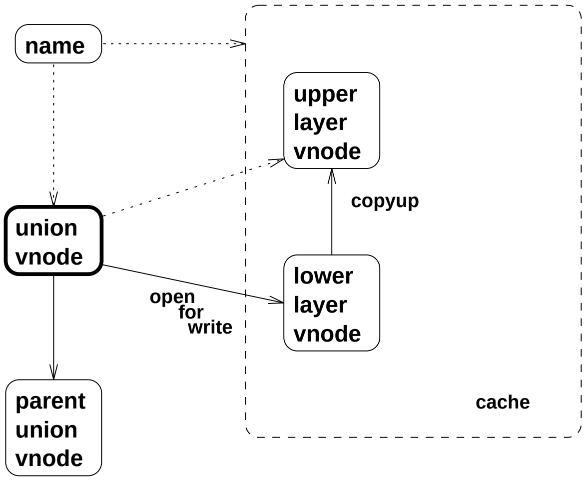

Figure 5: *Copyup operation*

the lower layer. The lack of cache coherency shows up when a lower layer is visible elsewhere in the filesystem and is being modified. Because shadow directories are never deleted by the union filesystem, the upper layer could retain a shadow directory whose corresponding lower layer directory had been deleted.

4.4.2. I/O performance

I/O performance through a union stack is similar to that of the underlying physical filesystem. Since no data or attributes are copied or cached within the union stack, the only performance degradation is that caused by the additional levels of functions calls through the vnode interface – one additional vnode function call for each layer in the union stack.

4.5. Access checks

The file access check made through a union stack is slightly more complex than that made for the file itself. To enforce the ''lazy copy'' semantics the following two tests are made:

1. If a lower layer object exists then the user who did the original mount must have read permission on it.
2. The current user must have the appropriate permission on the topmost object.

Test 1 ensures that access to a file or directory is not given away unwittingly by letting a user create a shadow directory in the top layer and then use the top-layer permissions. The credentials of the user who did the mount are used since the intent is to mimic a lazy copy by that user. Test 2 is the straightforward test that the topmost object can be accessed by the user.

Suppose user `floyd` creates a union stack and there is a file in the lower directory also owned by `floyd` with mode `-rw-r--r--`. As expected, this file would not be writable by a different user, `biema`. Moreover, although the copy-up functionality exists, the new upper layer file would still be owned by `floyd` (as the user who did the mount) and so the permissions would still not be alterable by `biema`.

To be able to modify the file, `biema` would have to copy it to another name, remove the original file (creating a whiteout) and then move the copy back to the original name replacing the whiteout. This sequence of operations are identical to the operations required on existing filesystems and demonstrates how the semantics of file permissions are preserved even through a union stack.

5. Whiteouts

A *whiteout* is a directory entry that hides any identically named objects in the lower layers. Any reference to that name returns the error code `ENOENT`. It is possible to detect the existence of whiteouts in a directory using the '`-W`' option to ***ls***(1).

5.1. Creation and deletion of whiteouts

Whiteouts are created automatically by the union filesystem whenever a namespace modifying operation is executed that cannot be implemented by normal changes solely in the topmost layer. For example, attempting to remove a file in a lower layer results in a new whiteout being created in the upper layer. Whiteouts are removed automatically when a new file or directory with a matching name is created. They can also be explicitly removed using the '`-W`' option to the ***rm***(1) command, causing the lower layer file to reappear.

Suppose an *unlink* is executed on the name '`x`' shown in figure 1. When the name in the `/a` layer is removed the object in the `/b` layer will become visible. To prevent the lower layer object from showing through, a whiteout for the name '`x`' is created in the topmost layer. Similarly, if an unlink is executed on the name '`y`' a corresponding whiteout will be created. However, no whiteout would be created if the file '`w`' was removed since there is no visible object with the same name in any of the lower layers.

Whenever a whiteout is replaced by a directory, the directory inode is automatically tagged with the *opaque* attribute[1]. During a lookup in a union stack, the union filesystem does not continue to the lower layer object if the upper layer object is marked opaque. To understand this requirement, consider the sequence of operations:

```
rm -rf tree
mkdir tree
```

After the `rm` command has completed, there will be a single whiteout in the upper layer covering the name `tree.` When the `mkdir` command is run, the whiteout is replaced with a directory in the upper layer called `tree`. Without the *opaque* attribute, the normal union semantics would make the lower layer tree visible once more. By marking the newly created upper layer directory with the *opaque* attribute the union filesystem ensures that the semantics prevents the lower layer objects in `tree` from becoming visible. Should it be desirable to 'undelete' the underlying object, the *opaque* attribute can be turned off manually using the ***chflags***(1) command.

5.2. Whiteout implementation

Directory entries in 4.4BSD were extended to include a tag field that records the type of the associated file. This tag was added specifically for the benefit of functions such as the file tree walker, ***fts***(3), which can now determine whether a name represents a directory simply by looking at the directory entry. There is no problem keeping the type field in the directory synchronized with the type on the physical filesystem since the type is set permanently at the time of creation.

The semantics of a whiteout for a given name is that a lookup or delete operation on that name should return "No such file or directory" (`ENOENT`).

A natural way to implement whiteouts is to put a small amount of additional logic into the kernel directory scanning function. The current implementation defines a new directory type, `DT_WHT`. A whiteout is created simply by adding a new directory entry with the required name, but with `DT_WHT` as the type. Although no physical inode is allocated, a non-zero file number must be present so that the entry will not be skipped by those programs that ignore directory entries with file number zero. Thus, whiteouts are assigned the file number `WINO`. For UFS filesystems, `WINO` has the numerical value one that is otherwise unused. If a file in the upper layer is being deleted, and hence a whiteout is also being created, the filesystem simply converts the directory entry to the new type and updates the file number.

[1] Files in 4.4BSD-Lite have a set of attribute flags associated with them. These flags can be used to specify that a file is append-only, immutable or not to be dumped during a backup. The union filesystem adds the *opaque* attribute described here.

When a directory entry of this type is discovered during the directory scan, the scan stops (since it has found an entry with the correct name) but returns the `ENOENT` error code described above. The kernel `nameidata` structure, records that a whiteout has been discovered, rather than no entry whatsoever. This information is used by the union filesystem to determine whether to do a lookup in the lower layer or not. If a whiteout has been found then the lookup through the lower layer is skipped.

An alternative would have been to define a new inode and vnode type and allocate an inode for each whiteout object. Had an inode been allocated for each whiteout it would be possible that an over quota error could occur while executing an unlink system call. The directory tag implementation makes running out of resources much less likely. (An out of space error could occur if the top level directory would have to expand to hold the whiteout name, and no disk space was available in the top level filesystem to fulfill the expansion request.)

The ***fts***(3) library identifies whiteout entries and returns information about them. Programs that need to see whiteouts use the `FTS_WHITEOUT` flag when calling `fts_open`. This flag implements the `ls -W` and `find ... -type w` commands.

6. Suppression of duplicate names

Since the union filesystem makes the contents of many directories visible at one place, one issue to consider is that of duplicate names. Duplicate names occur whenever a name appears in more than a single layer in the union stack.

There are two approaches to this issue. The simplest way is to ignore it entirely resulting in the name appearing twice in an `ls` listing. This approach is taken by Plan 9. Multiple identical names cause user confusion, since the user cannot easily discover which one they will really get if they try to access it. It is also impossible to select only one of the names with a globbing expression. For these reasons, 4.4BSD-Lite (and TFS) eliminate the duplicate names.

For the union filesystem, duplicate suppression is implemented in the ***opendir***(3) function. Whenever a union stack is being read, the entire directory is read and duplicates are removed. At the same time, whiteout entries are recognized and cause later entries with the same name to be suppressed. Commands that scan a file tree function correctly and do not have to deal with the case of a name appearing twice.

The decision to implement this functionality in the C library was taken to avoid requiring the kernel to buffer arbitrary amounts of data while executing a sort and unique operation. A new interface, `__opendir2`, that has an extra flags parameter to specify the behavior to be taken with whiteouts is used by the ***fts***(3) library to specify alternative semantics. The standard `opendir` is implemented by a call to `__opendir2` specifying that whiteouts should be suppressed.

7. Union stack management

In the current implementation, it is the responsibility of the user to ensure that filesystems are layered as required. The expectation is that the logical ordering of the union stack will be unchanged for the lifetime of the usage of the filesystems. Changing the ordering of the layers will have the greatest effect on whiteouts. Since the whiteouts themselves are created within the filesystems, if the filesystems are re-ordered, the effect of the whiteouts will vary. At any time, a whiteout will hide names in lower layers, but whiteouts may only be created or deleted in the top layer.

8. NFS as an upper layer

The NFS filesystem does not implement whiteouts, so it is not generally possible to use NFS as an upper layer in a union stack. The exception is that if the union mount is made read-only (using `mount -r ...`) then the mount is allowed since no attempt to create a whiteout will be made. It is always possible to use NFS as a lower layer. The implementation does not specifically know which filesystem will be used as the top layer. Instead, at mount time, the kernel calls the whiteout vnode operation for the upper layer filesystem and asks whether whiteouts are supported. Currently only UFS (fast filesystem, memory-based filesystem and log-structured filesystem) includes support for whiteouts.

9. Applications

9.1. Writable CD-ROMs

A problem with releasing software on a CD-ROM is the difficulty of updating it when patches arrive. The union filesystem can be used to address this problem by mounting an empty UFS directory on top of the CD-ROM mount point.

```
mount -t cd9660 -r /dev/sd1a /usr/src
mkdir /usr/local/src
mount -t union -o above /usr/local/src /usr/src
```

By taking advantage of the union filesystem copy-up and whiteout functions, files on the CD-ROM can now be modified, renamed or removed as needed. Even programs such as ***patch***(1) work as intended.

9.2. Source tree management

The union filesystem can be used to create a private source directory. The user creates a source directory in their own work area, then union mounts the system source directory underneath it.

```
mkdir ~/sys
mount -t union -o below /sys ~/sys
```

The view of the system source in `~/sys` can now be edited without affecting other views of the master source since all changes remain in the private tree. When the changes are completed they can be merged back into the master tree in `/sys`.

9.3. Architecture specific builds

A common difficulty with public domain software is building it for several different architectures from a central (NFS mounted) source area. Usually the software must be reconfigured each time a different architecture is used and the source tree must be cleaned out. Other software uses a combination of different subdirectories or clone trees with symlinks.

The union filesystem provides a clean alternative. Simply attaching the source tree beneath a local build directory will cause all configuration changes and object files to be created locally without modifying the central source tree.

```
mount -t nfs fb:/usr/src /usr/src
mkdir /var/obj
mount -t union -o below /usr/src /var/obj
cd /var/obj/sbin/mount_union
make
```

The union stack limits all modifications to the local system and even hand made changes required for a ‘quick and dirty’ port will not be visible to other machines. The master source tree is still accessible via the path `/usr/src` so changes could be re-integrated to the master source if required.

10. Future work

The existing implementation is functionally complete as originally envisaged. Since starting to make use of the union filesystem some additional enhancements and uses have come to light.

10.1. Cache filesystem support

The existing functionality of the union filesystem is very close to that required by a simple cache filesystem. In particular, the namespace and copy-up operations are already implemented.

The cache protocol would be write-through, rather than copy-back. This approach simplifies cache coherence, especially when using an underlying filesystem such as NFS that does not provide any support for network-wide data coherence. The changes required to implement this idea would be in vnode functions such as `VOP_WRITE` and `VOP_SETATTR` that would need to modify the remote copy first, then apply the change locally. If the remote modification failed, the local copy would be flushed from the cache and a failure returned for the whole operation.

10.2. Automounter

A serious problem with using an automounter [Pen94a] is the way it returns symbolic links pointing from the automount name to the mounted filesystem. One solution to this problem is to add special support to the kernel such as has been done in Solaris 2 [Cal93a]. An alternative is to take advantage of the namespace operations provided by the union filesystem.

As an example, consider an automount point `/home`. This directory will be used as an automount point for users' home directories. The home directories themselves will be mounted under `/automnt` which will be loopback union mounted on `/home`. First the automounter is started and mounts itself on `/home`. Then `/automnt` is union mounted on top of `/home`. These operations give a stack consisting of `/automnt` on top of `/home`, with the automounter beneath it.

When a lookup for user `/home/floyd` arrives, the kernel will do a lookup in the union stack for that name.

- First it will do a lookup in the topmost layer of `/home`, which will search `/automnt` and fail with `ENOENT`,
- Then it will do a lookup in the lower layer of `/home` which has the automounter mounted on it; this lookup will cause the automounter to mount the appropriate filesystem on `/automnt` and then return `ERESTART`:

```
mkdir /automnt/floyd
mount -t nfs host:/wherever /automnt/floyd
[return ERESTART]
```

- The (internal) error code `ERESTART` will cause the system call to be re-executed from the top layer, and this time the lookup will find the name in `/automnt`.

Later lookups will find the name immediately the first time `/automnt` is visited. Periodically the automounter can try to unmount the filesystem:

```
unmount /automnt/floyd
if ok
    rmdir /automnt/floyd
endif
```

The automounter is now only responsible for mounting and unmounting named objects and no longer has to maintain the namespace seen at the automount point itself. This approach makes for a much smaller and simpler program. A simple performance improvement is possible. When the mount point is created by the automounter, it can be marked *opaque* before mounting the home directory. The *opaque* flag will prevent a full union stack traversal on later lookups.

11. Availability

The union filesystem code was first released in 4.4BSD-Lite. The refinement of the interface including the addition of whiteout was added following the initial release of 4.4BSD-Lite. The union filesystem implementation described in this paper did not appear until revision 2 of 4.4BSD-Lite.

The union filesystem does take advantage of the stacking and vnode operation vector expansion features found in the 4.4BSD vnode interface. While the code will not drop into systems lacking these features, it should be possible to port it to other vnode implementations. Several other changes have been made at user level, mostly in the C library. These changes should be straightforward to integrate into the shared C library on most systems.

References

Cal93a.
B. Callaghan and S. Singh, "The Autofs Automounter," *USENIX Association Conference Proceedings,* pp. 59–68 (June 1993).

Hei94a.
J. S. Heidemann and G. J. Popek, "File-system development with Stackable Layers," ACM

Transactions on Computer Systems, pp. 58−89 (February 1994).

Hen90a.
D. Hendricks, “A Filesystem for Software Development,” *USENIX Association Conference Proceedings,* pp. 333−340 (June 1990).

Kha93a.
Y. A. Khalidi and M. N. Nelson, “An Implementation of UNIX on an Object-oriented Operating System,” *USENIX Association Conference Proceedings,* pp. 469−479 (January 1993).

Kha93b.
Y. A. Khalidi and M. N. Nelson, “Extensible File Systems in Spring,” TR-93-18, Sun Microsystems Laboratories, Inc. (September 1993).

Kle86a.
S. R. Kleiman, “Vnodes: An Architecture for Multiple File System Types in Sun UNIX,” *USENIX Association Conference Proceedings,* pp. 238−247 (June 1986).

Kor89a.
D. G. Korn and E. Krell, “The 3-D File System,” *USENIX Association Conference Proceedings,* pp. 147-156 (June 1989).

Mit94a.
James G. Mitchell, et al, “An Overview of the Spring System,” *Proceedings of Compcon* (February 1994).

Pen94a.
J-S. Pendry and N. Williams, “Amd − The 4.4BSD Automounter Reference Manual” in *4.4BSD System Manager’s Manual,* pp. 13:1−57, O’Reilly & Associates, Inc., Sebastopol, CA (1994).

Ros90a.
D. Rosenthal, “Evolving the Vnode Interface,” *USENIX Association Conference Proceedings,* pp. 107−118 (June 1990).

Acknowledgements

We would like to thank Keith Bostic for providing the final impetus necessary to get this work finished and published. We thank Phil Winterbottom and Rob Pike for reading and extensively commenting on a draft of this paper. While we incorporated most of their comments, we ignored their admonition that adding new options to the `ls` command has been banned since 1984.

Biographies

Jan-Simon Pendry is a graduate of Imperial College, London where he did postgraduate research on environments to support logic engineering. He is probably best known as the author of *Amd*, the freely available and widely ported automounter, and also as the author of six of the pseudo-filesystems in 4.4BSD. Jan-Simon has worked for IBM and CNS and is now a consultant with Sequent Corporation in Europe. He has two cats, Floyd and Biema. You can contact him via email at `<jsp@sequent.com>`.

Dr. Marshall Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4BSD fast file system, was involved in implementing the Berkeley Pascal system, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group overseeing the development and release of 4.3BSD and 4.4BSD. He is currently writing a book on 4.4BSD and doing consulting on UNIX related subjects. He is a past president of the Usenix Association, a member of the editorial board of UNIX Review Magazine, and a member of ACM and IEEE.