Implementing Union Filesystem as a 9P2000 File Server

Latchesar Ionkov
*Los Alamos National Laboratory**
lionkov@lanl.gov

ABSTRACT

This paper describes the design and implementation of a 9P2000 file server that allows mounting multiple 9P2000 file servers on the same mount point. The goal was to explore the challenges of such implementation.

1. Introduction

The Unix operating system represents all storage devices connected to a computer as a single file hierarchy. It allows a file tree to be attached to an existing directory, replacing the current content with the one attached. Historically Unix grouped similar files in a single directory – executables in /bin, libraries in /lib, etc. Using multiple storage devices somehow complicates the picture. Executables shared from a remote system need to be mounted in a separate mounting point, and although they still reside in a bin directory, it is hard for the user (and to the system) to know where all executables are located. With the years, few mounting points are standardized (/usr, /usr/local, /opt), and kludges are added (PATH, LD_LIBRARY_PATH environment variables) to help dealing with the problem.

Plan9 [8] operating system tries to solve the problem, allowing multiple file trees to be attached simultaneously to a mounting point, presenting the user a union of the content of the trees. The merging occurs only on the mounting point level. If more than one file tree contains a directory with the same name, traversing that directory will return only the content of the directory from the first tree. That restriction doesn't prevent the usage of union directories in Plan9. A standard installation has at least three union directories (/bin, /dev, and /net), with more than 15 file trees attached to /dev.

Plan9 represents all services the system, or applications provide as file trees. The user-space programs use 9P2000 [1] protocol to export their synthetic file system. It defines 13 operations that can be used for directory traversal and reading and writing to the exported files.

This paper describes an attempt to explore the challenges of implementing a solution that does deep merging of the file trees mounted on the same place. We decided to implement it as a 9P2000 file server for the following reasons: It is easier to write and debug user-space programs, than to change a kernel; a file server can be used both in Plan9 and in Linux (via v9fs [4][3] file system).

2. Design

Unfs exports two file trees. The main one represents a single file hierarchy that has external 9P2000 file servers attached to it. The second one controls the view of the first one. It is used to connect and disconnect the external file servers as well as to bind parts of the namespace to other places. When mounting a file tree, the user can specify whether to replace the content on the mount point, to show it before, or after the current content.

The controlling file server exports a single file named ctl. The user can write commands to it in order to change Unfs namespace. Reading from it returns the description of the current namespace. The format of the commands ctl accepts is:

*LANL publication: LA-UR-06-7888

```
command = mount-command | bind-command
mount-command = ‘‘mount’’ flag mount-point server aname
bind-command = ‘‘bind’’ flag mount-point path
flag = ‘‘-r’’ | ‘‘-a’’ | ‘‘-b’’
```

9P2000 protocol defines 13 operations. They can be roughly grouped in three categories: session management (version, auth, attach, flush and error), file operations (walk, open, create, read, write, clunk), and metadata operations (stat, wstat). The file and metadata operations use a 32-bit number called fid to identify the file that they are working on. The fids pool is managed by the file server clients. A fid to the root of the file tree is identified by the attach operation, new fids can be created using the walk operation. Every file on the file server has a 80-bit number, called qid, associated with it. The qid defines the file type and its version. It also contains a 64-bit unique sub-identifier of the file (path). If two files on the same server have qid path, they are assumed to be the same. Deleting and recreating a file with the same name should have different qid path.

Unfs uses server fid (SFid) structure for to reference files on the external file servers that it connects to. For every fid referencing a file on unfs, there is associated client fid (CFid) structure. Not all CFids are associated with a client fid. The file server also keeps a mount table (Mntable) that lists the mounted file trees as well as the places they are mounted on.

It is obvious that in order to support unification on every directory level, a CFid needs to be associated with multiple SFids. If a CFid is not a mount point, walking a name from it is translated to walks to that name from the associated SFids. If all server walks fail, the unfs walk fails too. If a walk succeeds the unfs walk succeeds.

There is a question what to do with SFids that failed to walk to a name. One approach is to clunk them. We don't need them anymore if we want to walk forward, or to open a fid. But keeping only the SFids that successfully walked to the current CFid path causes problems if we want to walk the CFid back (i.e. walk to "..."). Let's say that we have two trees (Figure 1), T1 and T2 attached to /src, and a CFid C, pointing to /src. The CFid will contain two SFids, S1 and S2, referencing the root directories of T1 and T2. Walking C to "libspfs" tries to walk S1 and S2 to "libspfs". The walks with both S1 and S2 succeed. Next we try to walk to srv.o. The S1 walk succeeds, the S2 fails, because there is no srv.o in T2. C now contains only S1. If we try to walk back to the directory containing srv.o, we can go back only in T1.

```
T1:
        unfs/
        libspfs/
                conn.c
                conn.o
                error.c
                error.o
                np.c
                np.o
                srv.c
                srv.o


T2:
        unfs/
        libspfs/
                conn.c
                error.c
                np.c
                srv.c
```

Figure 1: T1 and T2

In order to support walking back, we need to keep all SFids, even the one that failed, associated with the CFid. We need to distinguish the active SFids from the inactive ones, so we don't try to walk SFids that don't reflect the current CFid path. We also need to know how many times we have to walk back before a SFids becomes an active one again. For that purpose we introduce a new field in the CFid structure called level. The level shows the distance of the CFid path from the root of the file system. For example, a CFid pointing to "/src/libspfs/srv.o" has level 3. For every SFid associated with the CFid we keep the level of the path it successfully walked to. In the example above, when CFid points to /src/libspfs/srv.o, C's level is 3, S1 level is 3, and S2 level is 2. In that case only S1 is active, and walking forward will attempt to walk a name only in T1. If we walk back, both S1 and S2 become active again, and walking to "srv.c" will walk to the file both in T1 and T2.

An entry in the mount table contains the path that a file tree is attached to and a SFid (in case of mounting an external file server) or a CFid (in case of binding part of the current namespace to another place). Every entry is part of two lists: a list of all entries in the table in the order they were attached, and a list of all entries attached to the same mount point, up until the first entry that replaces the mount point contents. Every entry has a level. The level is a negative number if the tree is added after the current content, zero if the tree has to replace the current content, or positive number if the tree is added before the current content. When a tree is attached, depending on its flags, its entry is added in front, at the end of the list of entries, or in case of replacing the content, it creates a new list of entries for that path.

The unopened CFids point to the file before the mount table is consulted. Before an operation is performed on a CFid, the system looks up the mount table for file trees attached to the current CFid path. If there are entries for the path, the list of the SFids is modified to reflect the view with the additional file trees. After the operation is finished, and is different than open, the list of the SFids associated with the CFid is changed to point back to the state before the mount entries for the current (possibly new) path are consulted. We need to know which SFids are added because of the mount entries and to distinguish them from the ones possibly coming from the original tree. For every SFids associated with a CFid, we keep a "startlevel" property that shows on what level the SFid was added to the CFid. If the level of the CFid reaches that startlevel of an SFid, the SFid is clunked and removed from the list.

If there are file trees associated with the current CFid path, the list of SFids is modified as follows. The level of the CFid is increased by one. The entries for the path are traversed adding the clones of the SFids associated with them, setting their startlevel and level to the current CFid level. The SFids for the mount entries with negative number are added before the current SFids, the non-negative ones are added after them. If there is not a mount entry with level zero (replace), the level of the current SFids is increased by one, otherwise they are left as they were.

```
From        To               Flags       Level
/           localhost:666    Replace       0
/bin        /usr/bin         After        -1
/bin        /usr/local/bin   After        -2
/lib        /usr/lib         Replace       0
/lib        /usr/local/lib   After        -1
```

Figure 2: Sample Mount Table

As seen on Figure 4, the CFid B still keeps the SFid S2 that points to the original directory /lib, but it is not used for further walking. It is going to be used when the mount entries are "unevaluated". The unevaluation will remove S6 and S7 and will leave B to point again only to S2.

Examining only the mount entries attached to a path doesn't always give us the correct list of SFids. Let's look at this example:

```
mount -r / tcp!localhost!6666 ‘’
bind -r /bin /Users/lucho/bin
mount -a / tcp!localhost!6666 ‘’
```

```
Before evaluation of the mount entries:
  CFid A:
      level 1
      name /bin
      SFids S1(start 0, level 1, /bin)

  CFid B:
      level 1
      name /lib
      SFids S2(start 0, level 1, /lib)

  CFid C:
      level 1
      name /usr
      SFids S3(start 0, level 1, /usr)
```

Figure 3: CFids before mount point evaluation

```
After the evaluation:
  CFid A:
      level 2
      name /bin
      SFids S1(start 0, level 2, /bin),
          S4(start 2, level 2, /usr/bin),
          S5(start 2, level 2, /usr/local/bin)

  CFid B:
      level 2
      name /lib
      SFids S2(start 0, level 1, /lib),
          S6(start 2, level 2, /usr/lib),
          S7(start 2, level 2, /usr/local/lib)

  CFid C:
      level 1
      name /usr
      SFids S3(start 0, level 1, /usr)
```

Figure 4: CFids after mount point evaluation

After the bind operation, the content of /bin is going to be equivalent to the content of /Users/lucho/bin. If a CFid C that points to / tries to walk to /bin, after mount table consultation, the SFid associated with C will be "replaced" by a SFid pointing to /Users/lucho/bin and the walk will work correctly. But after the second mount, the things don't look that well:

```
Before evaluation
  CFid C:
      level 1
      name /bin
      SFids S1(/bin), S2(/bin)

After evaluation
  CFid C:
      level 1
      name /bin
      SFids S3(/Users/lucho/bin)
```

There is a single mount entry for /bin, its flags say that it should replace the current content, so listing the files in /bin will give us only the content of /Users/lucho/bin. That is obviously wrong, mounting tcp!localhost!6666 after all trees at / should also show the content of its /bin directory.

To solve the problem we change the implementation of the mount and bind commands. In addition to their own entry in the mount table, they can add more, one for each existing entry whose attaching place lays within the added tree. So in the example, instead of having a mount table with three entries:

```
/       tcp!localhost!6666       Replace
/bin    /Users/lucho/bin         Replace
/       tcp!localhost!6666       After
```

we are going to have four entries:

```
/       tcp!localhost!6666      Replace
/bin    /Users/lucho/bin        Replace
/       tcp!localhost!6666      After
/bin    tcp!localhost!6666/bin  After
```

With this addition, checking only the mount entries pointing the a CFids current path is going to give us the correct list of SFids.

Another problem arises on what qid value to return for a given CFid. There is no guarantee that the qid paths that the connected file servers return are unique across all servers. There should also be a way to combine the qids for all SFids associated with a CFid and create the appropriate Qid with a unique qid path and appropriate file type and version number.

Unfs usesthe following approach, mentioned in [5]: every server has assigned a unique value from 1 to 255. If the most significant byte in the qid path is zero, it is replaced by the server id, and the resulting qid path is unique across all servers unfs is connected to. If the most significant byte is not zero, a hash table is checked if it contains an entry for the (server, qid path) pair. If an entry exist, the unique path associated with that entry is returned, otherwise, a new entry is added and new unique path (with most significant byte zero) is generated and returned. The qid for a CFid is generated as follows: if there is a single SFid associated with a CFid, the qid of that SFid is returned. Otherwise the same hash table is used to check and generate a unique value for an ordered list of (srv, qid path) pairs.

3. Unfs Operations

In this section, we describe the individual Unfs operations. First two are done via the control file, and modify the namespace. The rest correspond to some of the 9P2000 operations that Unfs implements.

When Unfs is started, it serves only a single empty directory. Mount and bind commands have to be used to build the Unfs namespace.

Mounting a file server

When a file server is mounted to the namespace, Unfs first checks if the mount point exists in the current namespace. Then it creates a connection and establishes 9P2000 session. It adds an entry in the mount table, containing the mount point and the SFid to the root file of the file server. It may add more than one entry as described in the previous section.

Binding a file

When a bind command is executed, Unfs first checks if the mount point exists in the current namespace. Then it creates a CFid pointing to the root of the namespace and walks it to the specified path. It adds an entry to the mount table, containing the mount point and the CFid to that path. It may add more than one entry as described in the previous section.

Unmounting a mount or bind

The unmount command removes an entry from the mount table. If there were other entries that were added because of the entry, they are removed too.

Cloning a CFid

When a CFid is cloned, the result is a new CFid that contains clones of all SFids associated with the original CFid.

Walking from CFid

If the name is other than "..", before a CFid is walked, all mount entries associated with the current path are followed. The SFids associated with the SFid that have the same level as the CFid are walked to the specified name. The ones that succeed have their level increased.

If the name is "..", all SFids that have the same level as CFid are walked back, and their level is decreased. The CFid's level is also decreased. Then all mount entries that are associated with the new path are "unfollowed", i.e. all SFids with level equal to their startlevel are clunked.

If the first successful SFid points to a regular file, the Qid of the resulting file is the same as the SFid's Qid. Otherwise, the Qid is calculated from the Qids of all SFids that point to directories. The CFid's level is increased. The type of the CFid is the same as the type of the first SFid.

Opening a CFid

First, all mount entries associated with the current path are followed. If the type of the CFid is "regular file", only the first SFid on the same level as the CFid is opened, otherwise all SFids that point to directories are opened. The result (success or failure) is returned to the client.

After a CFid is opened, it cannot be walked anymore, so it is safe to clunk all unopen SFids.

If one of the SFid operation fails, the CFid will be in invalid state – some of the SFids are going to be opened, some not. There is no way to "unopen" a fid. Normally if an open fails, the fid is clunked, so we are not going to handle that case in other way than just returning an error.

Reading from a CFid

If the CFid point to a regular file, the operation is translated to a read to the only open SFid.

If the CFid points to a directory, the offset of the read cannot be an arbitrary number. It is either 0, or the offset of the previous read plus the number of the bytes the previous read returned. If the offset is zero, the read is translated to a read with offset zero from the first open SFid. The CFid saves the last SFid a read operation was performed on, and the next permitted offset. If a read from the current SFid doesn't return any data (count zero means end of file), the read operation tries reading from the next open SFid adjusting the offset of the read.

Writing to a CFid

Writing to directories is not allowed, a write operation is translated to a write to the first open SFid.

Clunking a CFid

Clunking a CFid clunks all SFids associated with it.

Creating a file

First, all mount entries associated with the current path are followed. The implementation sequentially send create commands to all SFids with the same level as the CFid, until one of the creates succeeds. The Qid of the CFid is changed to the Qid of the successful SFid.

If a directory is created, the rest of the SFids should be walked to that name, so the Qid of the file is calculated correctly.

Removing a file

Removing a file pointed to a SFid is translated to remove operation on all SFids on the same level. If any of the removes fails, the result of the operation is an error.

Reading file's metadata

Reading the metadata of a file pointed to a CFid is translated to a read of the metadata on the first SFid to that level.

Modifying file's metadata

Modifying the metadata of a file pointed by a CFid modifies the metadata of the file, pointed by the first SFid on that level.

4. Related Work

Early attempts to provide union mounting of file systems are the 3-D File System [6] and TFS [2]. TFS implements whiteouts (if there is an attempt to delete a file that exists on a read-only tree, a special file is created on the topmost tree that "hides" the file) and copy-ups (if there is an attempt to modify a file on a read-only tree, its content is copied to the topmost tree and modified there).

Plan9 [8] is one of the first operating systems that have support of union directories implemented in the kernel. It doesn't merge the content of the duplicate directories recursively and doesn't eliminate the duplicate file names.

The implementation of union mounts in 4.4BSD [7] allows adding trees to the top, or the bottom of the current view. Only the topmost directory is writable. Lookups in a lower level create corresponding shadow directories on the top level. It supports white-outs and copy-ups. Reading from a union directory eliminates the duplicate file names.

There is also an implementation of union mounts for Linux [**?**]. It allows multiple writable file trees, white-outs and copy-ups and duplicate filename elimination.

5. Conclusion

We created and tested a prototype that implements the design described in that paper. It can correctly mount and bind multiple directories on the same mount point. We still need to perform proper performance tests. There are some ideas how to improve the performance further that we need to explore. For example v9fs doesn't use the fact that the Qid path is a unique number, and we can skip the generation of unique path. We can also delay the cloning of SFids that are not on the top level when a CFid is cloned. Another issue that needs further work is adding authentication to the file server.

References

[1] Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[2] David Hendricks. A filesystem for software development. In *USENIX Summer*, pages 333–340, 1990.

[3] Eric Van Hensbergen and Latchesar Ionkov. The v9fs project. http://v9fs.sourceforge.net.

[4] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9p2000 under linux. In *Freenix Annual Conference*, pages 83–94, 2005.

[5] Jason Hickey. Providing asynchronous file i/o for the plan 9 operating system. Master's thesis, Massachusetts Institute of Technology, May 2004.

[6] D. G. Korn and E. Krell. A new dimension for the unix file system. *Softw. Pract. Exper.*, 20(S1):19–34, 1990.

[7] Jan-Simon Pendry and Marshall K. McKusick. Union mounts in 4.4bsd-lite. In *USENIX Winter*, pages 25–33, 1995.

[8] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.