Hacking the Linux automounter
Current Limitations and Future Directions

Ian M. Kent   Jeffrey E. Moyer

Abstract

The IT industry is experiencing a a considerable shift from proprietary operating systems to Linux. As a result, the features and functionality that customers have come to expect of these systems now must be provided for on Linux.

Many large scale enterprise deployments include an automounter implementation. The automounter provides a mechanism for automatically mounting file systems upon access, and umounting them when they are no longer referenced. It turns out that the Linux automounter is not feature-complete. And there are cases where Linux is just plain incompatible with the implementations from other proprietary vendors.

This paper looks at the common problems in large-scale Linux autofs deployments, and offers solutions to these problems.

In order to solve the current automounter limitations, we must start with an understanding of how things work today. To this end, we will explain some basic information about the automounter, such as how to configure an autofs client machine. We will walk through the code for basic operations, such as the mounting, or lookup, of a directory and the umounting, or expiry, of a directory. We will also look at where autofs fits into the VFS layer.

With a picture of the landscape in place, we will address major issues facing customer deployments. Currently, there are two principle pain points. The first is that the Linux automounter implements direct maps in a way that is incompatible with that of every other implementation. We will discuss the desired behavior and compare it with that of the Linux automounter. We will then look at ways to overcome this incompatibility by extending the autofs kernel interface.

The second major pain point surrounds the use of multi-mount entries for the `/net`, or `-hosts` maps. Because of the nature of multi-mount maps, the Linux implementation mounts and umounts these directory hierarchies as a single unit. This means that clients mounting several big filers can experience resource starvation, causing failed mounts. We will look at this problem from several different levels. We'll start at the root of the problem and show how the kernel, glibc, and automount can be modified to address the issue.

We will conclude with future directions for the automounter.

1 Introduction

With even a modest amount of information, network clients often need many mount entries in their tables to make the organization's information available. To make matters worse, the mount tables often change. The administrative overhead is not workable. This leads to heavy use of an automounter in many enterprises.

An automounter provides the ability to manage mount tables centrally, automatically mounting entries on demand and umounting them after a predefined period of inactivity. In addition to the reduction in administrative overhead, an automounter provides a dramatic reduction in the resources needed to have a significant number of file systems available on demand from an arbitrary number of network servers.

Many enterprises are adopting Linux as client workstations and server platforms, which has considerably increased the use of the Linux automounter in the past 2 years. As a result, bugs are identified and deficiencies are pointed out. Most importantly, places where the Linux implementation differs from that of industry standard implementations have become a significant issue. The most commonly raised discrepancies are:

- The Linux automounter implements direct maps quite differently from the industry standard.
- Multi-mount maps are mounted and umounted as a single unit.
- Browsable maps are not the default.
- The Linux automounter does not support included maps
- The Linux automounter does not support the `-null` map.
- The Linux automounter does not consult `/etc/nsswitch.conf` as it should for determining the source of an automount map.

Each of these issues causes problems in mixed environments, where Linux automount clients share the same maps with other vendor implementations, typically provided by a NIS server. They also cause problems in migrations to Linux from proprietary Unix platforms, where maps must be changed to either do things the Linux way, or work around the limitations of the Linux automounter. We will discuss these issues and others in section 4 .

2 Unix automounter

Every commercial Unix platform has an automounter implementation with a standard set of features. The most well known implementation is the one found in Sun^{TM}Solaris^{TM}. It has set the standard for what to expect in an automounter.

2.1 The master map

An automount configuration consists of a *master map* describing the mount tables it manages. It is generally located in the `/etc` directory and is called either `auto.master` or `auto_master`. It consists of a line for each automount managed mount point, formated as follows:

```
mount-point map-name [mount-options]
```

`mount-point`
: *mount-point* is the full path of the directory of the mount point. If the directory does not exist, it is created. The exception to this convention is that the entry may begin with a plus (+) followed by a map-name, which causes the specified map to be included from its source as if it were itself present in the *master map.*

`map-name`
: *map-name* is the name of the map containing the mount table. If it begins with a slash (/), it is interpreted as a local file name. Otherwise, the *name service switch* configuration is used to locate the source of the map. This can also be one of the special maps: `-hosts` used to mount exports from hosts on the network, or `-null` used to mark a `mount-point` to be excluded when parsing subsequent *master map* entries.

`mount-options`
: *mount-options* is an optional comma-separated list of mount options to be applied to the entries in the map unless entries in the map specify their own options.

Lines beginning with a `#` are comments and are ignored. Long lines may be broken by quoting the new line character with a backslash, as is common practice in configuration files.

The special mount point `/-` is reserved to indicate that the map is a direct mount map and is not associated with any specific top-level directory.

2.2 Mount maps

Mount maps consist of two types – *indirect* and *direct* – and have the following basic format:

```
key [mount-options] location
```

`key`
: *key* is the name used to look up mount table entries in the map. For indirect mount entries, this is the name of the directory upon which the mount will be made. For direct mount entries, this is the full path leading to the directory upon which the mount will be made.

`mount-options`
: *mount-options* is an optional comma-separated list of mount options to be applied to the map entry.

`location`
: *location* specifies the file system that is to be mounted on *key.* It can be a single file system or a number of file systems to select from using availability and proximity metrics. It may also consist of multiple *key [mount-options] location* offsets that each must start with a slash (/). If the first offset is /, then it is optional. These offset mount entries are referred to as multi-mount entries in Linux autofs.

There are a number of standard macro substitutions available for use in *location* specifications. They are commonly used in multiple architecture environments. A description of those normally available can be found in [2] on page 190. For those understood by Linux autofs, see `autofs(5)`.

As in the *master map*, lines beginning with a `#` are comments and are ignored, and long lines may be broken by quoting the new line character with a backslash.

A map *key* of `*` denotes a wild-card entry. This entry is consulted if the specified *key* does not exist in the map. A typical wild-card entry looks like this:

```
* server:/export/home/&
```

The special character `&` will be replaced by the provided key. So, in the example above, a lookup for the *key* `foo` would yield a mount of `server:/export/home/foo`.

The timeout on mounts points defaults to ten minutes and can be changed using a command line option when the service is started.

3 Linux automounter - autofs

The Linux automounter differs in relatively few ways from traditional Unix automounter implementations. In fact, all of the information provided in the last sections regarding configuration data apply to the Linux automounter as well. This section begins with a description of the Linux-specific details of the *master map*, and then moves on to the architecture of the Linux automounter.

3.1 Linux autofs master map

The Linux autofs *master map* syntax is a super set of the standard automount *master map* syntax. This is partly because Linux autofs does not utilise the *name service switch* to locate the source of maps and so must allow it to be specified.

The syntax is:

```
mount-point [maptype:]map-name [mount-options]
```

The fields above are the same as those described in section 2.1 ("The master map"), except for the `maptype`, which can be one of `file`, `program`, `yp`, `nisplus`, `hesoid` or `ldap`. The daemon supports the specification of a map format within the `maptype` parameter. It can be `sun` or `hesoid`, but the init script doesn't cater for it. The default format is `sun`, and it is a subset of the standard sun automount map format. Linux autofs understands much of this map format, and when a full implementation of direct mounts is added, the only things missing will be special maps such as the `-hosts` and `-null`.

3.2 Architecture

The automounter is implemented in two main parts: a user-space daemon, which is responsible for parsing map options and issuing mount and umount commands, and a filesystem, implemented in the kernel. The daemon is further broken up into the daemon proper and a set of loadable modules. To understand how the daemon operates, we will walk through the daemon startup for a minimal setup.

Consider the following `auto.master` map:

```
/net    /etc/auto.net
```

We will not show the contents of the program map, `auto.net`, as it is shipped with autofs. Autofs startup begins with the init script. This script parses the `auto.master` map and spawns one automount daemon for each mount point listed. The example given above will result in an automount command with the following parameters:

```
/usr/sbin/automount /net program /etc/auto.net
```

As shown above, the daemon takes as its options a mount point, the type of the map to be loaded, and the name of the map to be loaded.

Now we will look at the loadable modules. There are three types of modules: lookup, parse, and mount. Lookup modules are used to look up a given *key* in a map. The lookup module has code that understands how to get information from a map source. For example, lookup_file.so is able to read in entries from a file map. Map entries are stored as a key value pair. The key, as noted above, corresponds to a directory. The parse module is then responsible for parsing the value part of the key value pair. Finally, the mount module takes care of doing the actual mounting. This module has to know how to pass arguments on to the mount command. In the case of NFS, this module is also responsible for parsing replicated server entries.

Returning to the example above, the daemon knows that it needs to load the lookup_program module, since the program map type was specified in the command line. It calls the module's lookup_init routine, passing a map format (or none, in this case), and all arguments that the daemon itself did not process. These leftover arguments are considered to be map arguments.

The lookup module will perform its initialization and hand a context pointer back to the caller. Before returning, though, it loads the parse module, calling its parse_init function. It then passes the map format down, as well as any options it did not handle. The parse module will load the mount_nfs module, if it hasn't already been loaded. This module is always loaded, since the primary file system type mounted via autofs has historically been NFS.

3.3 Multi-mounts

Multi-mount entries allow the user to specify a directory hierarchy that will be mounted. For example:

```
mydir   -rw \
    /     server:/export/mydir \
    /src  server2:/export/home/mydir/src \
    /tmp  :/usr/tmp
```

This example demonstrates how to cobble together a single directory structure from multiple servers. One point to note here is that the `mydir` directory contains both an NFS-mounted file system, and mount points beneath it.

Currently, when any directory in this hierarchy is accessed, the automount daemon mounts every entry in the directory hierarchy. The expiry of a multi-mount entry also happens atomically.

This is the mechanism used to implement `-hosts`. The program map `auto.net` generates multi-mount entries on the fly, and the daemon mounts them when `/net/<servername>` is accessed. The `<servername>` is used as the *key*.

3.4 VFS interface

To understand the kernel interface used by autofs, it is necessary to know a little about the Virtual Filesystem Switch (VFS). The VFS is a software layer that handles all system calls related to standard Unix file systems. It does this by defining several data structures that contain information about the file system and objects that provide callback functions. The VFS uses the callback functions to carry out standard file system *operations*. The primary objects are the *superblock*, the *inode*, the *dentry*, and the *file* object. For the interested reader, a description of the VFS, its data structures, and the operations they define can be found in chapter 12 of [7].

The *dentry* object represents a single component of a directory path. One of the main functions of the VFS is to resolve a given file system path to its *dentry* by *walking* each of its path components.

The VFS kernel interface of autofs is conceptually straightforward. The automount functionality is provided largely in the *inode* operation *lookup* to lookup a new dentry, the *dentry* operation *revalidate* to revalidate an existing *dentry*, the *file* operation *readdir* to read a *dentry* directory, and with a file system specific *ioctl* to check for *dentrys* that have not been used for a given timeout.

The bulk of the work done in autofs is the mounting and expiring of file systems.

3.4.1 Mount lookup

Mount requests are triggered when commands or functions such as a `cd`, `ls`, or *open* cause the VFS to walk a directory path within the autofs file system. This in turn calls the autofs4 function *lookup* if the directory doesn't exist, or *revalidate* if it does. Within these functions there are two ways autofs can decide whether a mount needs to be triggered. First, if the directory doesn't exist, then *lookup* creates a negative *dentry* and passes it to the *revalidate* function. *Revalidate* knows that a mount needs to be requested when it sees a negative dentry, so it sends a mount request packet to the automount daemon. The daemon then issues a mount command and returns a status when done. For the second case, when the directory exists, the revalidate function is called and decides whether a mount request needs to be sent by checking whether the *dentry* is an empty directory and not already a mount point. If this is the case, then a mount request packet is sent to the daemon. This process is shown in figure 1 .

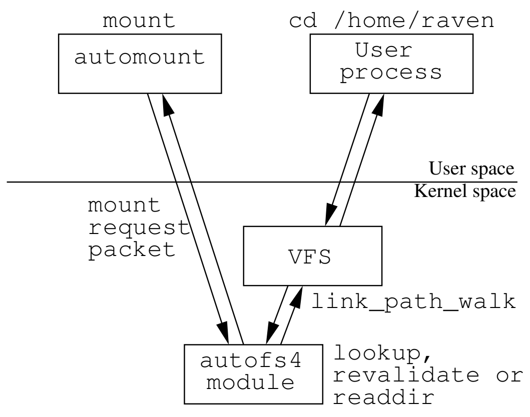

Figure 1: autofs mount lookup

3.4.2 Mount expiry

Expiration of mounts is achieved by calling the autofs expire *ioctl*. The autofs daemon does this when it receives an alarm signal, which has a frequency of one quarter of the mount timeout. The daemon looks for mounted file systems under the path on which it is mounted and asks the autofs kernel module whether it can expire them. If the kernel module decides that the daemon can expire a mounted *dentry*, then it sends an expire request packet to the daemon, which in turn issues an umount command and returns a status when done, as shown in figure 2 .

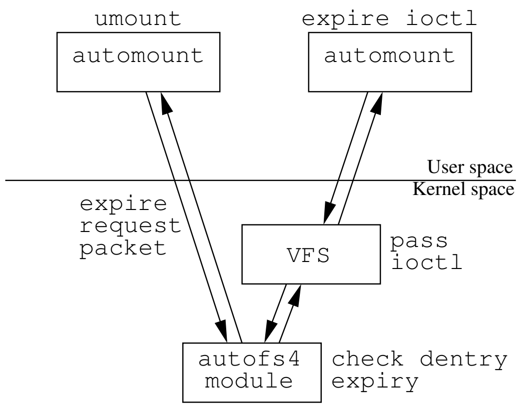

Figure 2: autofs mount expiry

4 Limitations

4.1 Master map semantics

Linux autofs starts instances of automount from its init script by reading a *master map* and parsing its contents. This is not really the right place to perform this task, so it's not surprising that there are a couple of things that the init script doesn't do.

First, if there are multiple instances of a *key*, it is expected that the corresponding maps will be merged. This feature is often used to add local maps to a given *key* on a per client basis.

The other thing that the init script, and hence the *master map* processing doesn not handle is the use of the `-null` map. The `-null` map is used to mark a *master map* `mount-point` as excluded from subsequent parsing. It also umounts these entries during a reload of the *master map*.

4.2 Included Maps

Another feature expected of an automounter is the ability to include a map in-line from within another map using the syntax *+mapname*. This feature is supported in both master maps and mount maps.

Linux autofs does not yet know how to do this. We will briefly discuss this issue in section 6 when we talk about the new version of autofs.

4.3 Large Number of Mounts

There are 2 issues using a large number of NFS (and autofs) mounts. The first is the number of devices available for mounts. The second is reserved port allocation in the RPC layer.

4.3.1 Anonymous devices

NFS and autofs use the anonymous block device major number. In a vanilla 2.4 kernel, this provides a maximum of 255 devices and hence a maximum of 255 mounts[1]. A commonly used patch provides an additional 4 unused major device numbers, which increase the number of devices available for mounts to 1280. The kernel-assigned device numbers provide an additional three major block device numbers for NFS mounts, but they are not yet used. So the number of possible mounts could be 2048. However, the limit on the number of anonymous devices is typically not reached, due to the port allocation limitation in the RPC layer (discussion below).

The maximum number of anonymous devices was substantially increased in the 2.6 kernel[1], and it is questionable whether effort should be spent resolving this same problem in the 2.4 kernel given the port allocation limitation in the RPC layer.

4.3.2 RPC Port Allocation

Many of the RPC based services (mountd, portmap, NFS, etc.) use a reserved port in the range 1-1024 for their operation. This is done to prevent non-privileged users from subverting the services.

When a service requests an RPC connection, binding to a reserved port is the default. The RPC layer scans ports starting from 800 down until it finds one that is unallocated. This method would be fine if RPC were able to multiplex traffic for multiple connections to a server over one or a few sockets. However, it cannot yet do so.

When a source port is not provided during RPC connection establishment, the RPC layer will attempt to allocate a reserved port for both UDP and TCP connections[1]. While this attempt is not so bad for UDP, it's terrible for TCP mount requests because of the lengthy time lag during which the socket is not available for re-use after being closed. Using ports outside the privileged port range is possible only if the exported file system is configured with the “nosecure” option[4]. A code review is needed to establish whether other services, such as mountd and portmap, can be configured to allow insecure ports for their connections. But of course, using insecure ports is generally not a good idea.

Autofs and mount also perform RPC probing to discover whether the target server is available before performing a mount. This process leads to as many as nine ports per mount being used during a mount, which causes rapid exhaustion of reserved port space. The RPC port allocation algorithm allows for a maximum of 800 concurrently mounted file systems when using UDP.

The situation is somewhat different with TCP. For each TCP mount attempt, a client uses multiple reserved ports, and each TCP socket must transition through the TIME_WAIT state to ensure the completion of the TCP three-way handshake. This process ensures that *lost duplicates* don't cause errors on subsequent connections. The TIME_WAIT state is 2*MSL (maximum segment lifetime)[3, Ch 2, Sec 7], which is 60 seconds for the Linux TCP stack. After this timeout, these reserved ports are free for use again.

This leads to a practical limit of around 100 TCP protocol mounts performed in rapid succession. If the mounts are performed much more slowly, as is expected in normal operation, this number is somewhat larger. Nevertheless, it generally falls somewhat short of the theoretical limit of 800 before port allocation problems appear.

4.4 Handling multi-mounts

Multi-mounts were discussed in section 3.3 . These map entries must be handled atomically, mounted and umounted as a single unit. Problems arise when using the `auto.net` program map if the servers have a large number of exports, or if there are a large number of mount point offsets in a multi-mount entry. They must be handled as a single unit due to possible nesting dependencies within the mount hierarchy.

The anonymous device and reserved port exhaustion described in previous sections are the source of the problem. We will present a partial solution to this problem in section 6, where lazy mount/umount of multi-mount map entries is described. Even with the planned improvements, though, there is still a limit on the total number of mounts that can be active at any one time due to resource exhaustion. The only real solution to this problem is multiplexing of RPC connections.

4.5 Parsing nsswitch.conf

Currently, the Linux automounter does only limited parsing of the nsswitch.conf file. It is only referenced when trying to locate the master map during startup. The script just checks what sources are present in the *automount* entry in `nsswitch.conf`, and looks for the *auto.master* map in each location.

There are a couple of reasons for this. First, all other consumers of the `nsswitch.conf` file use the standard glibc interfaces for accessing the `nsswitch.conf` file. This interface is not conducive to the use that automount makes of it. However, writing another parser for the `nsswitch.conf` file format is also not an attractive idea.

The format is described in the `nsswitch.conf(5)` man page. It includes basic usage, such as "subsystem: lookup_list". It also contains some more complex usages, such as "subsystem: lookup_type [*reaction*] lookup_type". The general form of *reaction* is:

```
'[' ( '!'? STATUS '=' ACTION )+ ']'.
```

STATUS can be success, notfound, unavail, or tryagain. *ACTION* is either return or continue. Thus, the following entry will look up a *key* in NIS, and it will fail the lookup if it is not found. However, if the lookup failed because the NIS service was not available, it will try LDAP:

```
automount: nis [NOTFOUND=return] ldap
```

It would be nice to leverage the existing code in glibc for parsing this file. However, if you embed the automounter lookup modules in libc, then it becomes more difficult to update the lookup modules themselves. This practice could also introduce a dependency between the version of the installed automounter and the version of the installed libc package. Such dependencies are not desirable, and could lead to an increased overhead and maintenance burden for the automounter developers. As such, it seems that the right way to address this problem is to parse the nsswitch.conf file from the autofs code itself.

5 Direct mount support

Limited direct mount map support was introduced in autofs version 4.1.

This support is implemented by creating submounts internally for intermediate path components and reduces to indirect automount points for the leaves of the map. If the direct mount map refers to a mount within an existing file system, then the upper levels of that file system will be hidden, because an autofs file system will be mounted over them.

For example, the direct map

```
/nfs/apps/geoframe perseus:/local/apps/geoframe
/nfs/apps/tomcat   perseus:/local/apps/tomcat
```

works fine if the directory tree /nfs is devoted to the direct mount map alone.

But the example

```
/usr/share/man      atlas:/local/${OSNAME}/man
```

will not work, because /usr will be broken out and over mounted.

Another limitation of this implementation is that it can't deal with single directory direct mounts as there is no way to turn them into an equivalent indirect mount. For example, the following will not work:

```
/data               filer:/local/data
```

This is clearly not a good implementation, but because of the severe limitation on the number of anonymous devices in the 2.4 kernel, it was decided to make this compromise to get a limited amount of functionality. Another consideration is that this scheme works with a wide range of older kernel modules and provides adequate functionality for a considerable range of maps found in everyday operation.

The limitations outlined here have all been resolved with the rework of direct mounts described in Section 6 .

6 Autofs Version 5

Work is well underway to resolve a number of the limitations described above. In order to implement the new functionality in a clean and sensible way, it has been necessary to increment the kernel protocol version to 5.00. It seemed sensible then, to avoid confusion, to increment the version of the user space daemon to 5.0.0 as well. Given the decision to increment the major version, it follows that the development priority should be to implement missing functionality rather than attempt to retain compatibility with older versions of autofs. Hence, much of the new functionality will work only with version 5.00 of the kernel module. At this stage, existing indirect mount maps should continue to work as in previous versions.

6.1 Direct mount implementation

The first and most important task is to implement fully functional direct mounts. This is even more important because it will pave the way for lazy mount/umount of multi-mount and host maps.

Two methods are available to do this. The first is to use file system stacking similar to that found in wrapfs from the FiST[5] system. Although using wrapfs from FiST was very compelling, in the end it was thought it would increase the complexity too much when compared to the chosen method.

The method that has been used is to treat each direct mount entry as a distinct mount and take advantage of the VFS *inode* method *follow_link* to trigger mounts. This method is safe because a directory cannot be a symbolic link; therefore the method cannot otherwise be in use. Since the mount point directories are created in the host file system, the VFS doesn't call the autofs *lookup*, *revalidate*, or *readdir* methods, allowing the use of the *follow_link* call (which follows the lookup) to trigger the mount before walking into the next directory. This implementation is surprisingly simple but effective.

The changes needed in the daemon are relatively straightforward as well. A mount option *direct* has been added so the kernel module knows it is a direct mount and can send mount requests at the right time. An additional entry point has been added to each of the lookup modules to enumerate a direct map so that the mount triggers can be set up. This function is also used to enumerate the map entries to perform expiration.

The changes in the communication protocol between the kernel and the daemon also allow a single process to handle an entire direct map.

One difference comes in the expiry of direct mounts. Each direct mount that has had a mount triggered *over* mounts the direct mount point. Because of this it is passed over when the kernel *walks* the directory path. Therefore the business timeout can only be updated during an expire run. As a result, only truly busy mount points (ie. with open files or a processes working directory) will prevent expiry. Hopefully this will not be a problem.

Another issue is that because direct mounts are made on directories within the underlying file system, additions to direct maps cannot be seen until the map is reloaded. Deletions and modifications of map entries are detected as normal.

It is interesting to note that existing industry implementations implement direct mounts in a similar way.

6.2 Lazy mount/umount

Lazy mount/umount of multi-mount map entries has been a difficult problem to solve for some time now. But with the direct mount changes above, we can clearly see how it can be done.

The basic problem to be solved is that of nested mounts. Let's revisit the example of section 3.3 on multi-mounts with a couple of small modifications to demonstrate the problem:

```
mydir   -rw \
    /        server:/export/mydir \
    /src     server2:/export/src \
    /src/f77 server2:/export/src/f77 \
    /src/c   server2:/export/src/c \
    /tmp  :/usr/tmp
```

When `mydir` is accessed, the file system corresponding to the offset / is mounted. But now the file system is not necessarily an autofs file system, so we can never get a callback from the kernel. So autofs never knows another mount is needed. Therefore, we must treat the entry as a single unit and mount everything. Clearly this necessity applies equally when there is nesting at lower levels in the offsets, such as the offsets in the `src` directory.

We can deal with this issue by partitioning the offsets and installing direct mount triggers within each of the file systems. In our example, when `mydir` is accessed we mount the entry corresponding to / and install direct mount triggers for each offset within the list bounded by nesting points. In this case, we install direct mounts for `/src` and `/tmp`. Similarly, when one of these mounts is triggered we mount it and install the corresponding triggers. In the example we mount the entry for `/src` and then install triggers for `/scr/f77` and `/src/c` and so on.

Expiring these is a little trickier, because for multi-mounts like these we need to expire the direct mounts themselves as well as the file systems that may be mounted on them. This is opposite from the way the direct mounts described above behave. To solve this problem, we need a way for the kernel to distinguish multi-mounts from standard direct mounts. The obvious way to do this is to add an additional mount option. It is likely to be *multi*.

The interesting thing about this scenario is that when a file system is mounted on a trigger that is perhaps itself nested, it will always be seen as busy by the expire system because of the file handle for the communication pipe to the daemon. On the other hand, a direct mount trigger without such a mount doesn't hold open a pipe but creates it at mount time. So multi-mounts can expire in a natural way without further complication.

6.3 Host maps

Once the lazy mount/umount has been implemented, this new technique will resolve many of the the resource issues with host maps. It is planned to have a separate module devoted to handling host maps. This should amount to little more than enumerating a local hosts table, enumerating their exports when they are accessed and treating them as multi-mount entries as above.

6.4 Initiator utility

Another important issue is the parsing of *master maps* in the init script. The init script is clearly not the right place for parsing a map specification. As is the case in other industry automount implementations, parsing should be done in a utility designed specifically for that purpose.

Another feature that is expected of an automounter is that when there are multiple entries for a *key* in the master map, these entries should be merged as described in section 2.1 . This can be accomplished by using the multi-map support already present in autofs. Development of this utility will also provide a way to implement `-null` map support in a fairly straightforward way. There will be difficulties with map refresh when part of the multi-map is busy when a changed map requires it to be nulled. But otherwise this utility should work fairly well.

The other requirement of this utility is to use the *name service switch* to look up map sources. The code developed from this should be written so as to be readily usable by the core autofs for the same purpose, thereby solving another of our long-standing limitations.

6.5 Included map support

Included map support, although important for compatibility, has not been worked on yet. Once the work outlined above is further along this problem will be tackled. The design is not yet clear, but it will be along the lines of creating a stack data structure to "push" the context of the map currently being parsed so that parsing of an included map can be done. Once completed, the context of the original map will be "popped", allowing parsing of the original map to continue. We will first see this implemented as part of the initiator utility described in the previous section. The functionality should be able to be applied to mount maps as well as the *master map* in a fairly natural way.

7 Concluding remarks

The astute reader will have noticed that the above implementation of direct mounts and lazy mount/umount of multi-mount maps will use a lot of anonymous devices. This use has become possible since the limit on the number of these devices was greatly increased in the early stable release cycle of the 2.6 kernel. It could be possible for this to function with a 2.4 kernel, but no work has been done to estimate the effort to backport the anonymous device changes. So initially at least, direct mounts will only be available for 2.6 kernels.

This paper has described a good number of the challenges in rounding out the Linux automount implementation. We don't mean to say that these challenges are the only ones we face – just the most difficult to address, as well as those that are fundamental to having a functional automounter on Linux.

The current status of the changes outlined above for autofs version 5 is that the direct mount code is largely done but has seen only limited testing. Development of the lazy mount/umount and the initiator utility has only just begun, so there's a way to go yet. Having said that, the direct mount implementation is crucial in providing a path forward. Hopefully much of the rest of the work needed will follow without too many delays.

References

[1] Linux Kernel source, Versions 2.4 and 2.6, http://www.kernel.org/.

[2] Hal Stern, Mike Eisler and Richardo Labiaga, Managing NFS and NIS, 2nd Edition, O'Reilly, June 2001.

[3] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, UNIX Network Programming, The Sockets Networking API, Volume 1, Third Edition, Addison-Wesley Professional Computing Press, 2004.

[4] Travis Bar, Nicolai Langfeldt, Seth Vidal and Tom McNeal, Linux NFS-HOWTO, http://nfs.sourceforge.net/nfs-howto/, 2002-08-25.

[5] FiST: Stackable File System Language and Templates, Eraz Zadok et al., http://www.filesystems.org/.

[6] Sun^{TM}Microsystems NFS Administration Guide, Chapter 5, http://docs.sun.com/, 1995.

[7] Robert Love, Linux Kernel Development, Second Edition, Novell Press, 2005.