Linux Software Engineering
-
Towards a Modern AutoFS

By: Mike Waychison <michael.waychison@sun.com>
Edited By: Tim Hockin <thockin@sun.com>

Table of Contents

1 Abstract

Automounting is a system that allows local and network filesystems to be mounted as needed. The automount configuration for a system is distributed via flat files or via network based lookups. This can become a difficult system to get right given the large set of features required and some new features available in Linux today. By breaking out the task of automounting from a daemon into a usermode helper application, we are able to simplify the architecture in both userspace and kernelspace. This enables us to solve some existing problems, deal with Linux filesystem namespaces, and gives us an architecture to provide per-namespace automount configurations. This document describes such a system and details what infrastructure needs to be added to the Linux kernel before such a system can be implemented.

2 Introduction

Traditionally, automounting has been implemented in one of two ways. The earlier implementations usually handled the entire problem of automounting by creating a userspace NFS server. More modern implementations have added kernel support directly by either modifying the VFS layer or by creating filesystems that cope with the problem. Both of these architectures have traditionally relied on one or more daemons that handled all policy in userspace and were responsible for performing the actual mounting of filesystems.

The earlier implementations that used a userspace NFS server worked by mounting NFS shares at appropriate locations in the filesystem tree. The daemon that served those shares would then be able to trap all directory traversals and perform mount actions as required. Some systems have also used similar techniques to catch triggering actions and have the desired filesystem mounted elsewhere, using symbolic links that point back to them. These systems lent themselves to difficult administration and often lead to hung filesystems and cruft mounts when the daemon was unexpectedly killed.

Later implementations placed traps directly into the kernel by creating a new filesystem called ‘autofs’. This filesystem would be responsible for triggering on directory traversals and would pass mount requests up to userspace. The kernel infrastructure became necessary as it became more and more evident that implementing everything in userspace became extremely tedious and difficult to properly manage.

Different architectural models exist for daemon implementations. Solaris currently uses a single daemon approach that handles all requests coming from kernelspace. This allows for easy management of changing maps and for dealing with the expiry of nested maps. The single daemon approach was also preferred because it consolidated any process overhead for the entire system into one process.

One of the difficulties in managing a single daemon automount system is that the entire system must be tooled to work asynchronously. This includes all components from performing NIS lookups to performing NFS mounts. Although this is an achievable goal, it

requires a lot of work. It is much simpler to have a single process for each automount trigger that uses the existing synchronous facilities.

Unlike Solaris, Linux uses a multi-process daemon approach. This system works adequately given the level of functionality it aims to support, but it is not without flaws. Linux currently only supports the use of indirect maps and the nesting of maps via the fstype=autofs mount option. Each indirect mountpoint has exactly one daemon and one map associated with it. All map lookups are performed synchronously within the daemon. This means that a lookup for an entry within a given mount may block and may cause a second lookup within the same indirect mount to block unnecessarily. This is a fair design decision as both entries are determined to be coming from the same source and will equally block[1]. Mounts are handled in an asynchronous fashion by forking and executing the mount(1) command. Linux's implementation is multi-process in the sense that each indirect mountpoint has an associated daemon process. Nesting of maps is handled by maintaining parent-child relationships between daemon processes. A parent process manages the parent map. When an entry in this map is accessed which has the 'fstype=autofs' option specified, the daemon forks and executes a new copy of itself. This child daemon is responsible for the nested map. The two processes communicate with each other using signal IPC so that they may synchronize expiry of the nested map.

One problem with Linux's multi-process approach is that it does not handle lazy mounts in multimount entries. This is evident in the implementation of automount 3.9.99-4.0.0-pre10 (the most current release at the time of this writing). In this daemon implementation, multimounts are not lazy mounted and will, by default, attempt to mount all entries in the multimount immediately. It will also, by default, fail if any of the filesystems fail to mount properly. This latter problem has been quick-fixed by the addition of a 'nostrict' option for multimounts. This leads to large numbers of potentially unneeded filesystems being mounted and causes unnecessary latency. A multimount entry may contain multiple shares from different hosts and mounting them all can cause a noticeable lag to a user application. Mounting unneeded remote filesystems also increases the likelihood that one of the filesystems will go stale and hang processes which attempt to access them. A filesystem can go stale when the system serving it crashes, is rebooted or when network connectivity is lost. For example, depending on the configuration given to an NFS client, a crashed server will cause all processes accessing the NFS filesystem to hang indefinitely.

Another flaw that both Solaris' and Linux's automount models have is that neither lends itself to dealing nicely with filesystem namespaces, a new feature in Linux as of kernels 2.4.19 and 2.5.22. Namespaces allow a system to have multiple distinct mount hierarchies. Namespaces are created by a call to the clone(2) system call with the CLONE_NEWNS flag. This call creates a new process as it usually would, but the new process receives an entirely new copy of the parent process's namespace. This is done by creating a new mount table for the given process, and essentially re-executing all previous

[1] The exception to this is of course any networked map that is being served from a slave server. Any cached entries may be returned at different speeds and blocking times may actually vary. A second example of where this rule doesn't apply is when a local file-backed map is referenced, which in turn includes another map from the network.

mounts in the new mount table. This creates a completely distinct mount table, and allows any changes such as bind mounts, moved mounts, new mounts, and unmounts to only be reflected within that namespace. This idea was partially borrowed from distributed operating systems such as Plan 9[i] and Spring[ii], which allows users to create their own mount hierarchies, independent from any other user.

The key as to why these automounting implementations cause problems when using namespaces is that the previous implementations rely on a daemon process that inherently resides within a single namespace. Whenever an autofs mount is triggered, the kernel communicates with the daemon, which in turn mounts the filesystem onto the given path. Unfortunately, this breaks cross namespace functionality because the mounted filesystem is grafted into the daemon's namespace, which may or may not be the same namespace as used by the triggering application. Namespaces are designed such that cross-namespace facilities are deliberately absent. The easiest method for performing any cross-namespace functions is to execute within the alternative namespace.

Determining whether a process is a user application causing a trigger or an automount daemon performing a mount has also traditionally been difficult and required special casing. We can avoid any such special casing by providing a file descriptor that describes the target directory to the automounter, which would in turn fchdir(2) to the target location.

With the current state of automount understood, we can explore the problems that exist today and look at new approaches to automounting.

3 Requirements

A new automount system involves several new requirements in order to work gracefully with new Linux facilities. To enumerate these requirements we must start by examining the current implementations and determining where things begin to break. Specifically, we will look at the current modes of userspace / kernelspace communication used by both the current Linux autofs3 and autofs4 implementations.

Traditionally the autofs filesystem has needed a way to distinguish whether an application that traverses into an autofs mount is a regular user process, or the daemon coming in to perform a mount. This has previously been handled in Linux by identifying the daemon using its process group, as registered at mount time. The use of the daemon's process group not only abuses Unix semantics, but also makes handling complex automount hierarchies very difficult. It forces the implementation to handle nested mounts using distinct processes in order to traverse the outer directories as if it were not the automounter.

Another big caveat to the current approach is the system's reliance on the automount daemon registering an open pipe with the kernel. This registration is made at mount time using a mount option to pass the pipe's file descriptor. This kind of communication channel registration makes for a system that is incapable of self-healing. It is impossible

in this form of communication for the daemon to disconnect from the kernel and reconnect. A daemon that dies (accidentally or forcefully) will leave the system with autofs filesystems mounted yet stuck in what is called a 'catatonic' state. The autofs filesystems will give up trying to communicate with their respective daemons and will not process any new triggers. On top of that, any expiry runs that should be occurring will cease to run as they are invoked by the daemon itself[2]. This forces an administrator to either manually unmount all the filesystems left behind, or more often than not, simply restart the daemon, causing more filesystem to be mounted over the existing stale ones in wait for the next reboot.

Another reason one would want to move away from a single daemon approach is because automounting semantics are not very clear when namespaces are used. One of the driving forces behind implementing distinct namespaces in Linux is to allow the root user to create distinct mount environments for differing services and users. This is different from chrooting because processes outside of the chroot environment can still navigate any new mountpoints within the chroot. When using namespaces, processes cannot navigate mounts that are not within their own namespace.

One particularly useful advantage to namespaces is that a user may mount a privileged filesystem such as a Samba share, without allowing any other users to see the mount in question. Not even the root user himself would be able to gain access to the contents of the filesystem. Another possibility of namespaces is that a system may be configured such that upon login, the login process could create a new namespace for that user and bind mount $HOME/tmp over /tmp. In effect, the user has a private /tmp directory that no other user is capable of accessing.

Namespaces are currently implemented such that root can create a new namespace by deriving from an existing namespace. From this derivation, the namespace may be completely customized by adding and removing mounts in the system. Given the current Linux autofs implementation, any derived namespace will inherit autofs filesystems, but they do not work as expected, as the persistent daemon has no access to this namespace and cannot thus mount new filesystems upon trigger. Instead of the mount occurring in the derived namespace, where it was triggered, the mount will occur in the original namespace in which the daemon is running and not be visible from the triggering namespace. Further, any filesystems that were automounted in the original namespace will persist in the new namespace and will never expire. The original mount will expire in its own namespace but the cloned copy of it will not be visible to the daemon. Even if the kernel (which can see all namespaces) told the daemon that the mount needed to be expired, the daemon itself has no way to unmount the filesystem in a namespace other than its own. This is clearly not the desired functionality.

Ideally, one would like to properly be able to inherit automount triggers when creating a new namespace. Automount triggers would ideally work as configured in the parent

[2] Expiry is triggered from userspace via an ioctl on the root directory of the autofs filesystem. The filesystem will in turn check to see if any of the current sub-mounts have been inactive for some period of time and will return the path(!) of the entry to expire back to userspace. Userspace will then attempt to unmount the path using umount(8).

namespace, but also be removable and installable using a different automount configuration. It is also desirable to have a system that is not reliant on a persistent daemon and which is capable of healing any stale triggers. The most obvious approach to handle these kinds of problems is to remove any persistent namespace context – namely the kernel's reliance on a single daemon, while providing more namespace context during the mounting process.

The following new set of architectural requirements become necessary:

- Automount triggers should continue to operate properly within a cloned namespace. We want to be sure that an automount trigger that exists in both the parent and child namespaces will cause a mount to occur in the appropriate namespace only.
- Automount triggers that are inherited from a parent namespace should remain distinct from their parent counterpart. We cannot allow a user in one namespace to alter the automount configuration across multiple namespaces.
- Filesystems that have been automounted and duplicated into a cloned namespace should continue to expire.
- The addition or removal of an automount trigger should only affect the namespace in which the change applies.

In addition, the following functions are required above and beyond the existing Linux automount implementation in order to be in line with the functionality provided by other Unix implementations:

- Both direct and indirect maps should work as expected.
- The system should expire and unmount any unused automounted filesystems.
- Lazy mounting should occur wherever possible.
- The system must be able to scale to thousands of mounts.
- The browsing of indirect maps should be supported.
- The system should be able to handle changing maps and update the current configuration as required.

4 Analyzing the Alternatives

Working with these requirements in mind, different types of architectures can be considered. Several facets of each potential architecture need to be examined.

1) Are any of the required facilities to implement this architecture already in place?
2) How much state is duplicated between userspace and in kernelspace?
3) How well can automount triggers be handled in a multi-namespace environment?
4) How simple is the implementation and how prone is it to error?

With these questions, we can evaluate different architectures for our new system. The following are a couple differing ways a new automounting system can be architected.

1) Perform everything in kernelspace. There is no need for a daemon. A utility will communicate with the kernel to install all the triggers. It is the kernel's responsibility to catch all directory traversals that require a new mount to occur. The kernel also handles name-service lookups, map entry parsing and performing the actual mounts.

   Pros:
   - Makes handling cross-namespace triggers a lot easier as full access to kernel data-structures is available.
   - Managing atomicity when handling a trigger is greatly simplified.
   - Full access to map resources is available.

   Cons:
   - Lookups being performed in the kernel places an enormous amount of logic in the kernel that is probably better left in userspace.
   - Does not leverage the benefit of using the mount(8) utility which already handles mounting different filesystems very well. Many filesystems, notably NFS and SMB, have differing APIs for handling mounts and require packed structures to be passed to the kernel.
   - Requires new APIs to be put in place that will allow userspace applications to remove triggers from their mount table.
   - Canceling a trigger action (e.g.: via a SIGINT) becomes much more difficult to handle properly.

2) Continue using a multiprocess daemon using file descriptors to describe the target mountpoints. Use a daemon similar to that used in the current Linux automount package. Augment the kernelspace/userspace communication protocol so that we can have the daemon mount and unmount on file descriptors (which are namespace aware) instead of by pathnames (which are namespace dependent).

   Pros:
   - Automounting continues to work across a cloned namespaces.

   Cons:
   - Requires new API that allows the passed back file descriptor to be re-associated with a map and key.
   - Would require one persistent process per direct/indirect mountpoint.
   - Difficult to handle lazy mounting of multimounts.
   - Difficult to manage a large hierarchy of processes that is continuously in flux.
   - Duplicates structure information found in the kernel.
   - Doesn't allow for clean administration of differing automount schemas across different namespaces.
   - Requires new system calls to natively support mounting and unmounting on a file descriptor.
   - Cloned namespaces are left with automount triggers that do not have a daemon running in the new namespace.

3) Create a single process daemon that is capable of handling all trigger requests across the system. Again, uses file descriptors passed back from kernelspace to describe mountpoint targets.

   Pros:
   - Consolidated process and memory overhead.
   - Can be done without maintaining too much state in the userspace daemon.
   - Continues to work as desired across cloned namespaces.

   Cons:
   - Requires new API for grabbing the file descriptor on which to mount, and associate the proper map sources.
   - Access to map information across namespaces is difficult to access. Files may differ, as may network service client configurations.
   - Requires new system calls to natively support mounting and unmounting on a file descriptor.
   - Requires asynchronous infrastructure to handle synchronous name service APIs.
   - Managing differing automount configurations becomes difficult.

4) Use a usermode helper application that handles the trigger requests. Contextual information is passed to the kernel when installing the automount trigger. This information is then passed back to a usermode helper application that is invoked on each triggering action. The usermode helper is invoked within the triggering action's namespace. All lookup logic and mounting is handled by the usermode helper which then mounts the desired filesystem on a given file descriptor which describes the target directory.

   Pros:
   - API for passing file descriptor and associated map information is already in place. All information can be passed in to the helper application via command line arguments, environment variables and through open file descriptors.
   - No daemon means state is only maintained in kernelspace.
   - Allows in place replacement of the userspace infrastructure.
   - No need to worry about a daemon dying and leaving the system with stale automount triggers.
   - Easy access to local namespace configuration for both file maps and network services.

   Cons:
   - A lot of triggers occurring simultaneously would invoke many processes.
   - A new facility that allows mounting operations using file descriptors of directories is needed.

The alternative approach of using a usermode helper application to handle the mount requests using a usermode helper application quickly becomes a viable option when one realizes the benefits in both cross-namespace use and reliability. By moving any logic that

was previously in the daemon out into a usermode application, we can enrich the userspace/kernelspace protocol by giving the process context about where the triggering action occurred. The use of the hotplug system is preferred in this implementation because it is already a well-defined and accepted form of kernelspace to userspace communication, though a separate but similar system could be used instead. /sbin/hotplug is currently invoked with any number of arguments and any number of environment variables. The goal is to have all trigger events be performed by the userspace agent. Unfortunately, as we will discover, implementing expiry is a more difficult task and must be done completely in the kernel.

Implementing automounting without having a single persistent daemon does also have its own problems. It assumes that the system upon which the automounting is occurring will have enough system resources to be able to handle a high automounting load. By invoking a single process per automount action, we are consuming more resources than a more traditional automount system would otherwise consume, and doing so in bursts. It is the belief of the author that these extra resources are reasonable and will not grossly affect the performance of the system. These assumptions should however be properly qualified by performing relevant benchmarks and stress tests on a prototype implementation.

The rest of this document describes a way to implement an automount system that uses a usermode helper application to perform automount requests.

5 Proposed implementation

By removing the need for a persistent daemon and by adding mountpoint navigation facilities we are able to address all of the shortcomings of the current Linux automount system and fulfill all of the new requirements introduced by namespaces. The preferred approach is to use a userspace helper application similar in nature to that used by the hotplug subsystem. /sbin/hotplug already provides userspace defined agents for a variety of systems and adding an automount agent is as simple as dropping a file in the /etc/hotplug directory.

It must be noted that the hotplug action will run outside of any chroot(2) environments. The current Linux automount implementations do not enforce any such restriction and mixing automounting with chroot(2) leads to undefined behavior. Chroots are different from namespaces because they share portions of the mount-table while differing namespaces do not. Forcing the hotplug invocation to occur at the root of a namespace enforces a single automount configuration per namespace. These semantics are similar to those on other operating systems when automounting and chroots are used in conjunction.

Registering an automount in a namespace will still be handled as a filesystem that will be responsible for catching any triggering actions. In the current Linux autofs implementations, the file descriptor for the writing end of an open pipe is passed as a mount option and used for kernelspace to userspace communication. This makes the kernel dependent on the pipe being open for communication with userspace. This causes

an automount trigger to become catatonic when the reading end of the pipe is closed. This communication artifact will be completely removed as part of the new protocol.

The daemon's process group is used in the existing automounter implementation to let the filesystem determine if the process causing a trigger was a user process accessing automounted resources or an automount daemon satisfying a prior request. In the design outlined in this document, we avoid this issue altogether by allowing the servicing process to bypass pathname walks. This is done by using file descriptors to describe target locations of mounts.

In addition to describing target directories as file descriptors, mount operations that are be capable of dealing directly with file descriptors are needed. Assuming new mount facilities are in place, mount operations throughout this document are done in terms of directory file descriptors. Rudimentary requirements are summarized in section 6.1.

Installing automount triggers in a system will be handled by mounting 'autofs' filesystems at the appropriate locations. Mount options will be used to pass all the context information needed later by the helper application when responding to triggering actions. Most of these mount options will not be interpreted by the kernel itself. They solely serve to pass contextual information to the helper application upon invocation. All mount options that are interpreted by the kernel are noted as such.

5.1 Indirect Maps

The implementation of indirect maps will be done using an autofs filesystem similar to that found in the current implementation. The main difference being that it will take a list of mount options indicating that it is an indirect map as well as where the indirect map entries can be found. For example, if the directory /home is to be an indirect mountpoint using the map auto_home, the following mount command would be used:

```
mount -o maptype=indirect,mapname=auto_home \
      -t autofs autofs /home
```

This would mount a filesystem of type autofs on the /home directory in the current namespace. The 'maptype' mount option is used by the filesystem code and tells it to use indirect map semantics[3].

A simple example indirect map might have a single entry as follows:

```
mikew       host:/export/home/mikew
```

Later on, if user mikew were to access his home directory /home/mikew, the system hotplug handler would be invoked as root in the same namespace as the triggering process:

```
/sbin/hotplug autofs mount
```

[3] The difference between direct and indirect semantics is that a direct map requires a trigger to occur on traversal into the autofs filesystem while an indirect map requires a trigger to occur traversal into each subdirectory. Direct maps are described in more detail in the next section.

This process is invoked in the same namespace as the triggering process because in order for the triggering process to see the mounts, we require that all mounts occur in the namespace of the triggering application. Also, the hotplug helper needs to access the configuration of the triggering application's namespace. This configuration may include the /etc directory, as well as any NIS and/or LDAP settings. Execution of the hotplug system is currently hard-coded to run in init's context. Running /sbin/hotplug in an arbitrary namespace differs from the existing hotplug functionality and should be documented as such.[4]

When invoked, the following environment variables[5] would be set:

```
MOUNTFD=0
MAPNAME=auto_home
MAPKEY=mikew
```

The hotplug agent would be responsible for performing the keyed lookup of $MAPKEY in the map named $MAPNAME. It would then use the information in the entry to perform the mount directly on the $MOUNTFD specified before returning a successful exit code. For the simple indirect mount case, these three environment variables comprise all the information that is required to properly perform the userspace actions. The $MOUNTFD environment variable refers to the number of an open file descriptor of the directory upon which to mount. The new mount system call will be used to allow for file descriptor based mount operations. A file descriptor is preferred because it allows any mount-related system calls to completely bypass any pathname resolution, thus allowing the automounter to bypass any triggers directly. This simplifies any blocking logic when a mount is occurring and eliminates the need for identifying the helper application as performing the mount. This allows us to have automount triggers handled by individual processes without any special reliance on their process group. It also alleviates the need for persistence (again, due to the process group dependency).

Once an autofs filesystem is mounted, we no longer rely on its absolute path for automount functionality. We effectively disassociate any map context information from the actual location of the mount. This allows autofs mounts to be moved (mount(8) –move option) or bound (mount(8) –bind option) without affecting automount functionality. It also allows an administrator to install automount triggers without modifying the /etc/auto_master file. For example, a map auto_ws could be manually installed on directory /ws using a command such as:

```
mkdir /ws
mount -o maptype=indirect,mapname=auto_ws -t autofs autofs /ws
```

---

[4] This semantic difference may justify using a different executable rather than /sbin/hotplug. Either way, hotplug is used for the sake of discussion.

[5] This document uses environment variables to pass values to the hotplug agent because it is easier to convey their relations in pseudo-code terms. An actual implementation may choose to use command line arguments instead of environment variables because '/sbin/hotplug autofs mount auto_home mikew 0' appears clearer. This is an implementation detail and of little importance to the discussion at hand.

This can be done without affecting any currently configured automount triggers.

5.1.1 Browsing

When an indirect map is installed on a directory, the resulting filesystem has no files or directories within it. Subdirectories are created upon lookup. For instance, the indirect mount on /home mentioned above would have no contents (other than the usual '.' and '..' entries) until access to some subdirectory is performed.

The exception to this rule is when the map entry for /home contains the option 'browse':

```
/home       auto_home   -browse
```

In this case, a directory listing of /home should return a directory entry for each valid key in the associated map. None of the entries should be automounted when this is performed. Such actions are delayed until the directories are traversed. This is useful from a user perspective, allowing a user to enumerate all entries that are available without requiring any mounts to occur.

In order to implement this functionality we begin by adding a 'browse' mount option to the autofs filesystem. This option switches behavior such that an indirect mount filesystem will call the usermode helper with the following information upon the first directory listing request (called by the ->readdir file operation on the root directory of the filesystem). The usermode helper will be called with the 'browse' action and will receive the following information on invocation:

```
MAPNAME=auto_home
OUTPUTFD=0
```

It is then the helper application's responsibility to retrieve the map and validate the entries. It will then pass the keys of the map back to kernelspace by printing them out to the file descriptor described by $OUTPUTFD. The kernel will take the values written to $OUTPUTFD and will later used them to fill in requests to readdir. It will need to create dummy directory entries so that lookups caused by calling stat(2) will return valid results. Once again, the usermode helper application will run within the same namespace as the triggering application so that namespace-local configuration is used.

In order to maintain some form of coherency between changing maps, these dummy directory entries will remain in place within the dcache so that the kernel doesn't need to query the usermode helper as often. These entries will periodically timeout and will be unhashed from the dcache. Any subsequent directory listing requires the kernel refresh these entries with a new call to the usermode helper. The timeout will be specified as another mount option ('browsetimeout=<seconds>') to the autofs filesystem. The value will be passed back to the usermode helper when mounting as the environment variable $BROWSETIMEOUT, so that the usermode helper may

inherit these values for any nested maps. This environment variable will be specified for all automount types, however, the browsetimeout mount option will only be used by autofs mounts that have maptype=indirect and the browse options set. Other configurations will silently ignore this value. A default value of 10 minutes (600 seconds) will be assumed.

Executing the usermode helper within the namespace of the triggering application does have a problem when browsing is used. We are caching map keys in kernelspace and can run into coherency problems when an autofs super_block is associated with multiple namespaces which have differing automount maps in /etc. This kind of situation may occur if a namespace is cloned and a new /etc directory with a different auto_home map is mounted. The results from a readdir within the first namespace may differ than the expected results from a readdir in the derived namespace. In order to handle this, facilities need to be added that allow autofs super_blocks to be cloned when cloning namespaces. Doing so ensures that an autofs super_block is local to its namespace and the namespace-local configuration. Cloning of super_blocks is described in section 6.3.

5.2 Direct Maps

Direct maps will be handled in a similar fashion to indirect maps. The main differences are outlined as follows:

1. The mount option 'maptype' is now 'direct'. This tells the filesystem code to have direct map semantics.
2. The map key for the direct mount entry is now passed as a new mount option called 'mapkey'. It will be the key to use when looking up the entry in the direct map. For direct map entries, this will always be the same as the path upon which the trigger is mounted; however, handling lazy mounts will also use this value as they will use the same kind of automount trigger.

This is different from indirect maps where the map key is produced by a directory lookup. Direct automounts have no such directory lookup and this contextual information must be explicitly specified at mount time. The value of this mount option is used as the $MAPKEY environment variable when the hotplug agent is invoked.

When a user process traverses into the root of an autofs filesystem that has maptype=direct, a mount needs to be performed. The triggering process will block while the hotplug userspace helper application is again invoked in the triggering process's namespace. For example, assume that the auto_master file has the following entry:

```
/-    /etc/auto_direct
```

This tells the installing application (see below: The Userspace Utility) to iterate over the /etc/auto_direct map and install a direct automount trigger for each of the entries in the map. Assume the auto_direct file contains one entry:

```
/usr/share  hostname:/export/share
```

To install this entry, the following mount command would be used:

```
mount -o maptype=direct,mapname=/etc/auto_direct,mapkey=/usr/share \
  -t autofs autofs /usr/share
```

This hands the kernel all the information it needs to pass back to the hotplug agent in order to let it perform the mount when necessary. When the agent is invoked, it is again called with the 'mount' action and it is passed the same environment variables as in the case of an indirect mount. In our example these are:

```
MOUNTFD=0
MAPNAME=/etc/auto_direct
MAPKEY=/usr/share
BROWSETIMEOUT=600
```

The helper application will need to go through and lookup the key[6] '/usr/share' in the map '/etc/auto_direct', parse the entry and finally mount the relevant filesystem on the directory specified by the given file descriptor. This is exactly the same logic as required for handling indirect maps.

5.3 Multimounts and Offsets

5.3.1 Explanation

A multimount is a map entry with an extended syntax that allows for a potentially complex hierarchy of filesystems to be mounted on a given directory. Multimounts may occur in both direct and indirect maps. They are most often used to enable the automounting of one NFS share nested within another. For example, if we want to automount hosta:/export/src on /usr/src and hostb:/export/linuxsrc on /usr/src/linux, we would need to use a multimount. In this case the multimount entry would be placed in a direct map and would look like the following:

```
/usr/src                hosta:/export/src \
            /linux      hostb:/export/linuxsrc
```

In this example, the hosta:/export/src is to be mounted directly on the /usr/src directory, and hostb:/export/linuxsrc. The mount information for /usr/src could have also been written as:

```
/usr/src    /           hosta:/export/src \
            /linux      hostb:/export/linuxsrc
```

In this example, the '/' of the multimount is explicit whereas in the first example it was implied. Both path components '/' and '/linux' are called offsets. A multimount is comprised of a set of offsets, each of which has a set of sources. In all the examples

[6] Even though the value of the key looks like an absolute path, it should not be interpreted as such. Its sole purpose is to index into the given map.

in this document, only one source (such as an NFS share) is given for each offset. There can very well be more than one source per offset. This technique of listing multiple sources is used to specify fail-over redundancy. Handling NFS fail-over redundancy is better implemented within the NFS subsystem and is not described in this document.

By design, the multimount syntax is really just a superset of the regular map entry syntax. For example, the following two map entries are equivalent:

```
Entry 1:
      mikew             hostc:/export/home/mikew

Entry 2:
      mikew       /     hostc:/export/home/mikew
```

In the first entry, the '/' offset is implied. So by design, all map entries may be treated as a multimount. Most of which simply only have the 'root offset' defined.

One of the interesting aspects of multimounts is that entries do not have to have a 'root offset' defined at all. For instance, consider the situation where three users exist on the system and their home directories all come from NFS servers. The indirect map for /home may look something like this:

```
userA       host:/export/home/userA
userB       host:/export/home/userB
userC       host:/export/home/userC
```

A new user is then added to the system who needs /home/userD/server1 to come from one server, while /home/userD/server2 to be mounted from a second server. There is no need to mount anything directly on /home/userD. This can be quickly added to the above map as the following entry:

```
userD       /server1    host1:/export/share1          \
            /server2    host2:/export/share2
```

In this entry, there are two different offsets defined, namely '/server1' and '/server2' but there is no 'root offset' defined.

To complicate matters even more, offsets can also nest within each other:

```
/usr        /           hosta:/export/share/usr \
            /src        hostb:/export/src       \
            /src/linux  hostc:/linuxsrc
```

The desired behavior is to 'lazy-mount' all these mounts. This means that only those directories that are accessed are ever mounted. So, if only /usr is being accessed, then only the share from hosta is mounted. Only when /usr/src is first accessed will

the share from hostb be mounted. The same ‘laziness’ holds for /usr/src/linux from hostc.

5.3.2 Implementation

An interesting aspect of implementing lazy mounts is that a multimount entry can be broken down into several direct mounts. This is done by associating an offset value with each direct mount trigger. This offset value is used at trigger time to identify which portion of the mount has just triggered and which subsequent triggers need to be installed. This offset value will be specified at autofs mount-time using a new mount option, 'mapoffset', and will be passed down to the hotplug agent as a new environment variable: $MAPOFFSET. The 'mapoffset' mount option will default to '/' if it is not explicitly specified. This builds on the definitions explained above for both direct and indirect maps.

With this in mind, we provide an example using the following direct multimount entry from map auto_direct:

```
/usr        /           hosta:/export/share/usr \
            /src        hostb:/export/src       \
            /src/linux  hostc:/linuxsrc
```

The mount command used to install the trigger would now look as follow (with additions in bold):

```
mount -o maptype=direct,mapname=auto_direct,mapkey=/usr\
  ,mapoffset=/ -t autofs autofs /usr
```

Once this automount trigger has been installed, a first access to the directory /usr will cause /sbin/hotplug to be invoked with the following environment variables:

```
MOUNTFD=0
MAPNAME=auto_direct
MAPKEY=/usr
BROWSETIMEOUT=600
MAPOFFSET=/
```

$MOUNTFD, $MAPNAME, $MAPKEY are still defined as in the explanations of both direct and indirect map handling. The agent is to retrieve the entry with key ‘/usr’ from the map ‘auto_direct’ and parse it. The key addition is that it now uses the $MAPOFFSET to figure out which part of the entry is being mounted. Once the filesystem is mounted, the agent then mounts any other required child offsets on top of the filesystem before exiting. So, in the case of traversing into the /usr directory, the following actions are performed:

- lookup key ‘/usr’ in map ‘auto_direct’
- parse entry
- lookup offset ‘/’ in entry
- mkdir('/tmp/<unique_dir>')

- mount 'hosta:/export/share/usr' '/tmp/<unique_dir>'
- mkdir(‘/tmp/<unique_dir>/src’)
- mount –o maptype=direct,mapname=auto_direct,mapkey=/usr\
  ,mapoffset=/src –t autofs 'autofs' './tmp/<unique_dir>/src'
- fchdir($MOUNTFD)
- mount –move '/tmp/<unique_dir>' '.'
- rmdir /tmp/<unique_dir>
- exit(EXIT_SUCCESS)

In this and following examples, we choose to use a temporary directory '/tmp/<unique_dir>' as an intermediate root of our mount because we need to be able reach into the newly mounted filesystem to install the child offsets. If we had directly mounted the share from hosta on $MOUNTFD, we would not be able to change the current working directory into the newly mounted filesystem without first traversing back into the parent directory and then walking back across the trigger. Using this intermediate directory allows us to bypass this completely. Once we have finished performing all of the nested mounts we complete the transaction by moving tree of mounts directly onto the target directory and returning a successful exit code.[7]

Comparing the initial autofs mount and the nested autofs mount, we notice that the only difference between the trigger on /usr and the trigger on /usr/src is the mapoffset mount option. This differentiator is enough to distinguish the two automount triggers.

If a user were then to traverse into /usr/src, similar actions are performed by the agent:

- lookup key ‘/usr’ in map ‘auto_direct’
- parse entry
- lookup offset ‘/src’ in entry
- mkdir('/tmp/<unique_dir>')
- mount 'hostb:/export/src' '/tmp/<unique_dir>'
- mkdir(‘/tmp/<unique_dir>/linux’)
- mount –o maptype=direct,mapname=auto_direct,mapkey=/usr\
  ,mapoffset=/src/linux –t autofs 'autofs' '/tmp/<unique_dir>/linux'
- fchdir($MOUNTFD)
- mount –move '/tmp/<unique_dir>' '.'
- rmdir('/tmp/<unique_dir>')
- exit(EXIT_SUCCESS)

Finally, if one walks into the /usr/src/linux directory:

- lookup ‘/usr’ in map ‘auto_direct’

[7] A final implementation would preferably use what we refer to as 'floating mountpoints' as described in section 6.1, 'Mountpoint file descriptors' to achieve the same desired effect without requiring the building of mountpoints in a temporary directory.

- parse entry
- lookup offset ‘/src/linux’ in entry
- mkdir('/tmp/<unique_dir>')
- mount 'hostc:/linuxsrc' '/tmp/<unique_dir>'
- fchdir($MOUNTFD)
- mount –move '/tmp/<unique_dir>' '.'
- rmdir('/tmp/<unique_dir>')
- exit(EXIT_SUCCESS)

5.3.3 Multimounts without root offsets

The only remaining problem to be dealt with is multimounts that have no ‘root offset’. These are a special case of regular multimounts and can be handled by still installing the direct mount trigger on the root of the multimount. However, instead of mounting a real filesystem upon trigger, a tmpfs filesystem is mounted before the agent proceeds to install child trigger mounts. Following is the auto_home map bound to /home from a previous example:

```
userA       host:/export/home/userA
userB       host:/export/home/userB
userC       host:/export/home/userC
userD       /server1    host1:/export/share1          \
            /server2    host2:/export/share2
```

We still install the indirect trigger on /home as before:

```
mount -o maptype=indirect,mapname=auto_home -t autofs autofs /home
```

When a process traverses into the /home/userD directory, the following environment variables are passed to the /sbin/hotplug agent:

```
MOUNTFD=0
MAPNAME=auto_home
MAPKEY=userD
MAPOFFSET=/
```

The agent takes this information and performs the following actions:

- lookup ‘userD’ in map ‘auto_home’
- parse entry
- lookup offset ‘/’ in entry
- mkdir('/tmp/<unique_dir>')
- // no root offset found! Install dummy filesystem:
  - mount –t tmpfs 'tmpfs' '/tmp/<unique_dir>'
- // handle child offsets
  - mkdir(/tmp/<unique_dir>/server1)
  - mount –o maptype=indirect,mapname=auto_home,\
    mapkey=userD,mapoffset=/server1 –t autofs 'autofs'
    '/tmp/<unique_dir>/server1'

    - mkdir(/tmp/<unique_dir>/server2)
    - mount –o maptype=indirect,mapname=auto_home,\
      mapkey=userD,mapoffset=/server2 –t autofs 'autofs'
      '/tmp/<unique_dir>/server2'
- // remount the tmpfs filesystem read-only because it is just a dummy filesystem.
  - mount -o remount,ro '/tmp/<unique_dir>'
- // move the tree of mounts onto the target directory
  - fchdir($MOUNTFD)
  - mount –move '/tmp/<unique_dir>' '.'
  - rmdir('/tmp/<unique_dir>')
  - exit(EXIT_SUCCESS)

We use a tmpfs filesystem on /home/userD because we need to be able to create directories and we would like to have these directories exist on a filesystem that is expirable. Traditionally, the directory of the root offset for entries with no defined root offset is immutable. It may not be changed by any userspace program. We use the simple approach of remounting the filesystem read-only once we have created the directories to simulate this effect.

The two nested direct mount triggers act as they normally would.

5.4 Expiry

Handling expiry of mounts is difficult to get right. Several different aspects need to be considered before being able to properly perform expiry.

In the existing Linux autofs implementations, the system works such that the userspace daemon will ask the autofs filesystem code to check to see if any of the automounted filesystems can expire (this is done by calling an ioctl on the base directory of the autofs filesystem). The autofs filesystem will then acquire the necessary locks and walk each of the currently mounted filesystems to see if anybody is using them. If the kernel code determines that a mount is ready to be expired, it sends the path back to the daemon. The daemon in turn unmounts it from userspace. This method of expiry has several problems:

- The autofs filesystem really should know as little about VFS internal structures as possible. In this case, the filesystem code is charged with walking across mountpoints and manually counting reference counts. This task is much better left to the VFS internals.
- Unmounting the filesystem from userspace is racy, as any program can begin using a mount between the time the daemon has received a path to expire and the time it actually makes the umount(2) system call. This sequence of events would make the expiry fail. Even worse, manually unmounting several mounts in a multimount can possibly lead to an expiry that fails to unmount after some of the mounts have already been unmounted, leaving the multimount in an inconsistent state.
- Having userspace initiate mount expiry requires a userspace application to periodically make the query the kernel. This is done using a daemon, but as we have

already discovered, automounting with a daemon does not work well when you are working in a multi-namespace environment.

These points suggest that the kernel's VFS sub-system should be charged with handling expiry. Some of the benefits of having it perform this functionality over other ad-hoc solutions are:

- All data structure specifics (like navigation and lock semantics) are maintained within the same component of the kernel. This improves maintainability and sustainability of the kernel proper and of individual filesystem implementations.
- Other filesystems would like to have expiry functionality in the VFS sub-system. Providing this service at the VFS layer would reduce duplicated efforts between filesystems to support this functionality. Similar to this is the way the VFS layer provides read-only functionality for all filesystems from a higher level of abstraction.

The following questions must be answered before a complete expiry solution is designed:

- How will the kernel determine the expiry timeout value? In other words, how does it know how much time must pass for an unused mountpoint before it expires?

  We will need to pass timeout values in from userspace. The simplest method to pass this information to the kernel is to pass it to the VFS layer as a mount option. This option is tentatively named 'vfsexpire' and will accept a timeout value given in seconds[8].

  As described above, we may be installing multiple mounts upon each trigger. This tree of mounts will need to expire together as an atomic unit. We will need to register this block of mounts to some expiry system. This will be done by performing a remount on the base automounted filesystem after any nested offset mounts have been installed

- How will the VFS layer verify that a filesystem is inactive?

  The VFS layer can atomically peek into the mountpoint structures (struct vfsmount) and look at the given reference counts to determine whether a filesystem is currently active or not.

Reference counting alone does not solve the issue of having to be able to atomically unmount several mountpoints. This is evident when lazy-mounting is considered. We would like to expire a base mountpoint that may optionally have nested autofs mounts ready to catch a trigger. These nested mounts increase the reference count on the base mount, and thus need to be considered as counting towards the total reference count.

[8] Unfortunately, the current mount system calls do not allow arbitrary information to be passed directly to the VFS layer if they cannot be represented as a boolean flag. A new set of system calls and interface semantics will need to be thought about and implemented for this mount option to be available.

These nested mounts in turn must recursively also be inactive for the base mount to expire.

The proposed semantics are as follows:

- A mount may be made without the vfsexpire mount option. In this case, the value defaults to 0, specifying that this mountpoint will never expire.
- A mount may be made with the vfsexpire=n mount option. This specifies that the kernel may detach this mount at some time after at least n seconds have passed with the mount inactive.
- An existing mountpoint may be remounted with vfsexpire=0. This signifies that if this mountpoint was to previously set to expire, it no longer will.
- An existing mountpoint may be remounted with vfsexpire=n, where n is non-zero. This signifies that this mountpoint together with any mountpoints currently underneath it will expire atomically. That is to say, if all of the said mounts are inactive (no one is using any of them, and nothing else is later mounted within them), only then will the entire tree of mounts expire together. This is an all or nothing expiry, where a hierarchy of mountpoints expires as a single unit.

We require that a tree of mounts be able to expire atomically together to ensure that we do not wind up with a partial expiry. A partial expiry would break our ability to lazy mount as some of the nested autofs filesystems would no longer be mounted. Such an arrangement would remain inconsistent until the root of the expiry is unmounted.

The unmount itself will be performed within the kernel. Doing so assures that the unmount occurred while nobody was accessing the filesystem. Further details on how native expiry support may be implemented are described below in section 6.2.

5.5 Handling Changing Maps

In a network that uses automounting in abundance, it is expected that maps will change fairly often. It is desirable that systems using the new automounting architecture will stay coherent with the maps provided by the nameservices on the network.

Before designing a strategy to handle changing maps, it is important to first understand what types of changes can occur. Table 1 describes a cross-section of map entry types and of the types of changes that may occur. This cross-section view allows us to identify how map changes are propagated to a running configuration given the automount system described thus far.

	Entry Modified	Entry Removed	Entry Added
Direct Entry (in direct map included from auto_master)	Updated on Expiry	Requires Removal	Requires Addition
Indirect Map (as listed in auto_master)	Requires updating associated context	Requires Removal	Requires Addition
Indirect Entry	Updated on Expiry	Updated on Expiry	Works

Table 1Strategies for Changing Maps

Most of the changes that may occur get propagated to a running system the next time a trigger is performed. This means that any updates to maps for an already mounted system becomes active after an expiry occurs. Each triggering action causes a new map lookup to occur. These map lookups will cause the trigger to receive any new modified entries.

5.5.1 Base Triggers

There are however certain conditions where a running system will not be completely in sync with changing maps. These changes involve the modification of the master map as well as any direct maps. Entries in these maps will need to be reflected on the running system by running a utility program that will synchronize the map contents against the filesystem layout on a running machine. This will involve adding or removing direct and indirect mountpoints as well as refreshing the context associated with each indirect mountpoint.

A utility program will need to create a delta between the running system and the master map and direct maps involved. This information is available from the proc filesystem (/proc/self/mounts). The program will then be able to identify any entries that would have come from the master or direct maps (by finding the autofs filesystem that are mounted on unique paths prefixes) and add and remove filesystems from the running namespace to bring the mount table in line with the maps.

We must also consider the case where indirect entries from the master map and direct entries from direct maps are installed and the maps subsequently change. In order to handle updating the context associated with the indirect trigger filesystem atomically, a remount is performed on the autofs filesystem with the new context passed as mount options. A simple approach would allow the remount to happen on a pathname because the following assumptions hold:

- The filesystem is an indirect filesystem, which will never be covered by another filesystem. If it is, then it is not updated.

- Because it is an indirect filesystem, remounting it will not cause any other filesystems to be incorrectly triggered (because the base directory of the filesystem is immediately available).

However, there remains the issue of a direct map entry that changes from one map to another, or is removed from the direct map set. Access to a direct map mount is not available when it is covered by another filesystem, and accessing it directly by pathname would in turn cause the direct mount to trigger and mount a different filesystem. Because of these problems, we need to define some method that allows a direct mount to be accessible in a manner that would not trigger a new mount, nor follow into any overlaying mounts. The proposed solution is to adopt a new interface that allows user space to navigate mountpoints on a given system. The goal is to use this navigation in conjunction with mount operations (such as unmounting and re-mounting with new options) to reconfigure an automount system and bring it up-to-date with all of the changing maps. Such a system for navigating mountpoints is described below in section 6.1.

5.5.2 Forcing Expiry to Occur

Given a new interface that allows the navigation of mountpoints within a namespace, we now have the ability to force expiry completely from userspace. Forcing expiry to occur becomes as trivial as writing a simple utility that gets the mountpoint file descriptor for the root filesystem and traverses across all mountpoints. Whenever this utility would see a mountpoint of type 'autofs', we would walk amongst its immediate child mountpoints and performing a lazy unmount[9] on each child mountpoint. Similarly, we can also remove all autofs filesystems from a given namespace by lazy unmounting them as well.

5.6 The Userspace Utility

The userspace utility program to be used in administrating an automounted system would preferably be called 'automount'. It would fulfill the following functions:

```
automount install [mastermapname]
```

This action would go through the master map (overridden by the mastermapname) and would install triggers within the running namespace.[10]

```
automount refresh
```

This action would go through the current namespace and update the base autofs filesystems as described in the section titled "Base Triggers". It would not perform a lazy unmount of all the mounted filesystems.

[9] See umount(8), 'Lazy unmount'.

[10] The master map (with default value '/etc/auto_master') will need to be accessible from the calling namespace, as would any other file map references.

```
automount detachall
```

This action would perform a lazy unmount on all the automounted filesystems.

```
automount uninstall
```

This action would remove all autofs triggers from the current namespace.

6 New Facilities

The following sub-sections describe in high-level detail the new facilities that are needed in order to fully support a robust automount system. The descriptions that follow are in places deliberately over-simplified as several of their design aspects are open for much discussion and debate.

It is hoped that the ideas below are well entertained. It is the intent of the author to further investigate details for each concept introduced and to propose more elaborate requests for comments to the community. Suggestions and comments are most welcomed for the sections that follow.

6.1 Mountpoint file descriptors

Mountpoint file descriptors are intended to describe mountpoints as first-class citizens within the Linux environment. By being able to describe mountpoints using file descriptors, we allow programmers and system administrators to continue using the tools they are used to, while at the same time enriching the semantics allowed for mountpoints. Some of the desired benefits of describing mountpoints as file descriptors are as follows:

- We wish to be able to use common APIs such as read(2) and write(2) to communicate with a mountpoint. This would be useful for communicating mount options specific to the filesystem, as well as with the VFS layer directly.
- We wish to be able to enumerate mountpoints somehow such that they may be modified without causing any path traversals to occur. This has the added benefit that we may access mountpoint configurations for mountpoints that are covered by other filesystems.

These mountpoint descriptors will most likely be accessible via a new mount system call, mount2. Mount2 will multiplex the following actions:

- 'Mount'. Take a mountpoint file descriptor and mount it on a directory, specified by a second file descriptor.
- 'Unmount'. Given a mountpoint file descriptor, attempt to unmount the filesystem if it isn't busy.

- 'LazyUnmount'. Given a mountpoint file descriptor, detach the filesystem from its namespace. Perform a lazy cleanup of resources when the filesystem is no longer in use.
- 'ForcedUnmount'. Given a mountpoint file descriptor, force an unmount to occur. Forcing unmounts is useful for filesystems such as hung NFS shares.
- 'Bind'. Given a source directory file descriptor, create a new mountpoint file descriptor that can later be mounted on any given directory file descriptor using the Mount sub-command.
- 'GetMfd'. Given a directory file descriptor, this command will return the directory's associated mountpoint file descriptor if the directory is the base of a mountpoint.
- 'GetDirFd'. Given a mountpoint file descriptor, this command will return an open directory as a file descriptor. This directory file descriptor will represent the base of the mountpoint as described by the mountpoint file descriptor.
- 'GetFirstChild/GetNextChild'. Facilities will also be put in place to navigate the children mountpoint file descriptors of a given mountpoint file descriptor.

Reading from a mountpoint file descriptor will result in a summary of the underlying filesystem, such as its type, the options it is using and its absolute path within the current namespace.

When a mountpoint file descriptor is unmounted using either the Unmount or LazyUnmount commands, the mountpoint it represents would remain valid. Instead of being directly associated within a namespace, the mountpoint is considered 'floating'. A floating mountpoint can be re-associated with a namespace by performing the Mount command. One of the benefits of floating mountpoints is that one can mount a filesystem without associating it with a namespace. The floating mountpoint can then be navigated by first acquiring the base directory of the mountpoint using the GetDirFd command and then changing the current working directory to it using fchdir(2).

Because of the way support for forcing unmounts is implemented, the ForcedUnmount command will invalidate the given mountpoint file descriptor upon successful completion. Any attempts to access the base directory on a forcefully unmounted filesystem will result in an error.

Together, these commands allow one to implement all of the mount operations with which we are familiar. For example, assuming a filesystem is mounted at /from, a move operation can be achieved in the following steps:

- sourcefd = open("/from")
- targetfd = open("/to")
- mfd = mount2(GetMfd, sourcefd)
- mount2(LazyUnmount, mfd)
- mount2(Mount, mfd, targetfd)

This example takes advantage of the fact that the underlying filesystem is still valid when it is lazily unmounted. We effectively disassociate the filesystem with the current

namespace (using LazyUnmount) and then re-associate it back with the namespace by calling Mount. Similarly, a recursive bind operation may be done by recursively visiting each mountpoint and creating new floating mountpoints using the Bind operation. These new mountpoints may be stitched together in userspace using the Mount operation along with directory file descriptors obtained using the GetDirfd operation before finally associating the new tree of mountpoints in the namespace using the Mount operation.

6.2 Native Expiry Support

David Howell from Red Hat has already implemented an expiry system that may eventually make it into the mainline kernel. His implementation is used to add automount functionality to the AFS filesystem. Specifically, the AFS filesystem implementation catches dangling symlinks whose symlink target is formatted to contain all the information needed in order to mount an AFS cell. His expiry implementation extends the VFS API such that one can construct a mountpoint and have it grafted into the current namespace's tree, while simultaneously linking the mountpoint into an expiry run list. This list is provided by the filesystem implementation. Linking into an expiry run list is handled by the VFS layer so that the filesystem itself need not worry about the locking semantics involved.

The experimental AFS automount patch periodically calls a new VFS function, mark_mounts_for_expiry. This function will traverse a list of vfsmounts and determine which are not in use and marks them appropriately. These markings state that the mountpoint has been inactive since that last mark_mounts_for_expiry run. If a later mark_mounts_for_expiry run comes across a vfsmount that already has a marking and is still inactive, the mountpoint is scheduled to be detached from the namespace. These markings are cleared on all calls to mntput, so any user which uses the mount between calls to mark_mounts_for_expiry will either put the mountpoint in an active state, or transition back to an inactive state but also clear the marking.

The mark_mounts_for_expiry patch has a few limitations that will need to be dealt with in order to completely integrate it with the VFS sub-system:

- The VFS layer currently delegates the run of mark_mounts_for_expiry to each individual filesystem. The delegation forces duplicate code between filesystems that wish to support mountpoint expiry. It also keeps a user from marking arbitrary mounts as being expirable. Each filesystem type must hold onto a list_head for their own expiry list, of which the filesystem code is not allowed to traverse without acquiring VFS-owned locks. These lists should be consolidated into the VFS layer directly. The VFS layer would in turn periodically call mark_mounts_for_expiry.
- Using a boolean marking forces the expiry timeout to be the within one and two times the period between calls to mark_mounts_for_expiry. This is fine, however it neglects the possibility of having per-mountpoint configurable timeouts. Greater configurability and granularity can be achieved by having each vfsmount store a

timeout period value. Instead of using a boolean marking, a counter would be used that would count up to the timeout value before expiring.

- In the mark_mounts_for_expiry patch, expiry is specified by a call to do_add_mount. This call now takes an additional argument, a list_head used to enumerate all mountpoints that should expire. By having the VFS layer handle expiry natively, we would no longer need to have this API addition. Instead, the VFS layer would intercept the vfsexpire mount option and will update its mount table and internal expiry run list to reflect these changes.

The proposed solution to this would see child mountpoints recursively associated as being part of an expiry when the parent mountpoint is linked into the expiry list. These associations will need to be cleared when any mountpoint manipulation occurs on the child mountpoints. They will be verified when checking the active state of the parent mountpoint to determine whether a child mountpoint is part of the parent mountpoint's expiry. The consistency of these associations will need to be managed by the VFS layer, which will simply remove any associations when a mountpoint is modified (possibly via a bind or a mountpoint move operation). The exception to this occurs when a namespace is cloned. In this case, any markings will need to be updated to remain consistent within the new namespace.

The following sequence of events and descriptions attempts to describe the semantics described above by example:

```
mount -o vfsexpire=10 /dev/hda1 /usr
```

The mountpoint at /usr is set to expire after ten seconds.

```
mount /dev/hda2 /usr/src
```

The mountpoint at /usr cannot expire because it is held busy by the filesystem mounted at /usr/src.

```
mount -o remount,vfsexpire=20 /usr
```

The mountpoint at /usr will now expire along with /usr/src after 20 seconds of both mountpoints being inactive. They will expire together atomically; e.g. Under no circumstances will /usr/src be unmounted by an expiry run without also removing the mountpoint at /usr.

```
mount /dev/hda3 /usr/local
```

The mountpoint at /usr cannot expire because it now has a new child mountpoint that is not associated with the expiry.

```
mount --move /usr/local /local
```

The mountpoint at /usr can now expire along with /usr/src after 20 seconds because it no longer has any child mountpoints that aren't associated with the expiry.

```
mount --move /usr/src /src
```

The mountpoint that was at /usr/src will no longer expire. Its association with the expiry of /usr is lost. The mountpoint at /usr will continue to expire after 20 seconds of inactivity.

```
mount --move /src /usr/src
```

The mountpoint at /usr will not expire because it is held busy by the mountpoint at /usr/src.

```
mount -o remount,vfsexpire=0 /usr
```

The mountpoint has its expiry disabled.

6.3 Cloning super_block

When a namespace is cloned, all the super_blocks for each of the currently mounted filesystems are shared between both old and new namespaces. Because filesystem-specific mount options are stored at the super_block layer, this creates the problem that changes to a mounted filesystem will affect all occurrences of the associated super_block. Sharing a super_block across namespaces opens the door to cross namespace tampering and contradicts our goal of keeping namespace configurations as isolated as possible.

The implications are less apparent with other types of filesystems. For example, given that an ext3 filesystem may be mounted in several places, it is a fundamental requirement that there only exists one running configuration of the ext3 filesystem at a given time, i.e. you wouldn't want to mount the filesystem in one place with data=journal and in another location with data=ordered (two contradicting options). This running configuration is represented as a single super_block, and the VFS layer ensures that only one super_block exists for any block device-backed filesystem. There is no such requirement for pseudo-device filesystems (those which do not have block devices backing them).

In order to allow namespaces to be cloned without letting changes within one namespace effect the other, we must develop a way for mount options to be kept distinct across the clone. Several alternatives are possible, some more immediate than others:

1) Do nothing. Allow cloned namespaces to share automount configuration within shared super_blocks.

Pros:

- No special work needs to be done

Cons:

- Can never be sure if a super_block is associated with a different namespace. This is a breach of isolation between namespaces.
- It becomes impossible to clone a namespace and update the automount configuration without affecting other namespaces save unmounting all autofs filesystem occurrences and replacing them with new instances.

Unfortunately, this option is not very viable as it does not achieve our goal of isolating automount configuration across cloned namespaces. A more complex method needs to be devised:

2) Allow a super_block to clone itself for the purposes of namespace cloning. This is preferably implemented as a new optional callback in super_operations. When called, the callback will generate a new super_block instance with the same configuration as the input instance. All directory entries (dentries) and inodes of the input super_block will also need to be duplicated so that filesystems mounted on top of the cloned filesystem may be stitched into the new namespace.

   Pros

   - Allows completely distinct automount triggers across cloned-namespaces.
   - Filesystems that are mounted within a cloned super_block will still be accessible within the new namespace.

   Cons

   - Duplicating all dentries and inodes for a given super_block in a consistent manner is not feasible given the locking and coherency semantics involved.

Unfortunately, the second option does not lend itself to dealing with cloning any sub-mountpoints easily. Mountpoints are internally dependent on dentries, which in turn are dependent on super_blocks. In order to clone a complete namespace while allowing the cloning of super_blocks as discussed in the second option above, we would have to not only clone the super_block, but also recreate any dentries and inodes associated with the super_block. This is a very difficult task to accomplish given the locking and coherency semantics involved.

This method is the only possible way conceived of guaranteeing the isolation of automount trigger configurations across cloned namespaces. The capability to clone super_blocks is needed and further investigation as to how this can be accomplished is required.

6.3.1 The –bind problem

When a mountpoint is bound (using mount(8)'s –bind option), the system is left in a state where two mountpoints exist that both use the same super_block. This leads to questionable behavior. Should remount options on one mountpoint affect the other? These semantics are currently being worked out, especially with the soon-to-be introduced per-mountpoint read-only mount option.

For the sake of simplicity, we may choose not to clone super_blocks for mountpoints when the mount bind operation occurs. However, this leads to strange semantics when mixed with the cloning of namespaces. For example, consider an autofs filesystem located at /foo. Super_blocks are shared on bind operations, so,

```
mount --bind /foo /bar
```

would result in two mountpoints sharing the same super_block. This allows any configuration changes performed on /foo to also affect /bar.

Assuming we naïvely clone super_blocks for autofs filesystems and a new namespace is then created, each of the mountpoints mentioned would each get its own super_block. With independent super_blocks for each mountpoint, changes to /foo would no longer affect the autofs mountpoint on /bar. The semantic of blindly cloning super_blocks for each mountpoint regardless of the number of mountpoints using the super_block results in a derived namespace that does not behave in exactly the same way as its parent namespace.

For these reasons, we extend the semantic description of cloning super_blocks when cloning namespaces. Instead of simply cloning the super_blocks that require it as we traverse the namespace, we keep a list of the cloned super_block pairs and re-use the newly cloned super_blocks for each mountpoint duplicated that referred to the ancestor super_block. This solves the –bind problem by ensuring that any mountpoints that referred to a single super_block will continue referring to a single super_block within the new namespace and that the two namespaces will continue to behave alike.

7 Scalability

Moving from the customary practice of using a daemon to using a usermode helper to perform automounting brings up the question about scalability. In this design, a new process is created every time a trigger occurs. This may lead to many small processes being created that have a very short lifespan. As such, the problem of having a lot of process overhead becomes a possible issue. The memory footprint for running a lot of small processes also becomes an issue.

The argument against these claims is that the process overhead in Linux is comparatively small, and is far outweighed by any network communication that will be occurring as part of the automount process. The time spent communicating with networked nameservices

(such as NIS or LDAP),latency spent in communicating with networked nameservices (such as NIS or LDAP) as well as network communication with a remote NFS server is many magnitudes larger than the overhead introduced by spawning a new process.

There does, however, remain the possibility of a denial of service attack by a user attempting to simultaneously trigger all of the automount triggers in a large system. Appropriate countermeasures to such activities can be put in place, such as defining a maximum possible number of simultaneous automounts triggered by a given user. This kind of issue remains an area of research and suggestions are welcome in dealing with this problem.

8 Conclusion

Linux automounting has always lacked full support for Solaris-style automount maps. This has long been the case due to technical limitations imposed by design as well as to lack of interest and time by the primary developers. It is our goal to make Linux able to support Solaris-style automounter maps completely and reliably. In order to achieve this goal, we need to redesign the way automounting works.

Namespaces provide a new and exciting way of dealing with security concerns, however, they make the problem space of automounting much more complex. By using a usermode helper in lieu of a daemon, we gain namespace accessibility. Namespace-local automount configuration and mount operations are at our disposal. We also gain the benefit of no longer having to maintain state in userspace, a task which is vulnerable to subtle changes in semantics[iii].

We also take the opportunity to define the semantics of automounting across cloned namespaces. These semantics require the ability to clone super_blocks in order to isolate automount configurations across namespaces. This appears at first to be an ugly hack, but in reality it makes sense considering the options that are available.

Another automounting task that has always caused problems in the past is the expiry of mountpoints. By moving mountpoint expiry into the VFS layer where it belongs, we eliminate any possible races. Expiring mountpoints also becomes available to anyone wishing to do so, whether it be part of the automount process or not.

Related to expiry is the ability for userspace to reliably navigate mountpoints so that covered mountpoints may be accessed and remounted. We've outlined a possible solution that will accommodate this need. The semantics involved are not yet completely defined and require insight from the primary consumers of such an interface.

It is hoped that the design outlined in this document is thorough enough to spark discussion as to how automounting should be implemented in the future. By implementing the core kernel facilities listed above, it is felt that a complete automount solution may be developed. This implementation would be completely capable of

handling Solaris-style automount maps and would continue to work reliably in a multi-namespace environment.

[i] “The Use of Name Spaces in Plan 9” Rob Pike et al. “http://plan9.bell-labs.com/sys/doc/names.html”

[ii] “A Uniform Name Service for Spring's UNIX Environment” Michael N. Nelson / Sanjay R. Radia http://www.usenix.org/publications/library/proceedings/sf94/full_papers/nelson.ps

[iii] “"simultaneous" mounts causing weird behavior” http://linux.kernel.org/pipermail/autofs/2003-November/000367.html