autofs - how it works

Purpose

The goal of autofs is to provide on-demand mounting and race free automatic unmounting of various other filesystems. This provides two key advantages:

1. There is no need to delay boot until all filesystems that might be needed are mounted. Processes that try to access those slow filesystems might be delayed but other processes can continue freely. This is particularly important for network filesystems (e.g. NFS) or filesystems stored on media with a media-changing robot.
2. The names and locations of filesystems can be stored in a remote database and can change at any time. The content in that data base at the time of access will be used to provide a target for the access. The interpretation of names in the filesystem can even be programmatic rather than database-backed, allowing wildcards for example, and can vary based on the user who first accessed a name.

Context

The "autofs" filesystem module is only one part of an autofs system. There also needs to be a user-space program which looks up names and mounts filesystems. This will often be the "automount" program, though other tools including "systemd" can make use of "autofs". This document describes only the kernel module and the interactions required with any user-space program. Subsequent text refers to this as the "automount daemon" or simply "the daemon".

"autofs" is a Linux kernel module which provides the "autofs" filesystem type. Several "autofs" filesystems can be mounted and they can each be managed separately, or all managed by the same daemon.

Content

An autofs filesystem can contain 3 sorts of objects: directories, symbolic links and mount traps. Mount traps are directories with extra properties as described in the next section.

Objects can only be created by the automount daemon: symlinks are created with a regular *symlink* system call, while directories and mount traps are created with *mkdir*. The determination of whether a directory should be a mount trap is based on a master map. This master map is consulted by autofs to determine which directories are mount points. Mount points can be *direct*/*indirect*/*offset*. On most systems, the default master map is located at */etc/auto.master*.

If neither the *direct* or *offset* mount options are given (so the mount is considered to be *indirect*), then the root directory is always a regular directory, otherwise it is a mount trap when it is empty and a regular directory when not empty. Note that *direct* and *offset* are treated identically so a concise summary is that the root directory is a mount trap only if the filesystem is mounted *direct* and the root is empty.

Directories created in the root directory are mount traps only if the filesystem is mounted *indirect* and they are empty.

Directories further down the tree depend on the *maxproto* mount option and particularly whether it is less than five or not. When *maxproto* is five, no directories further down the tree are ever mount traps, they are always regular directories. When the *maxproto* is four (or three), these directories are mount traps precisely when they are empty.

So: non-empty (i.e. non-leaf) directories are never mount traps. Empty directories are sometimes mount traps, and sometimes not depending on where in the tree they are (root, top level, or lower), the *maxproto*, and whether the mount was *indirect* or not.

Mount Traps

A core element of the implementation of autofs is the Mount Traps which are provided by the Linux VFS. Any directory provided by a filesystem can be designated as a trap. This involves two separate features that work together to allow autofs to do its job.

DCACHE_NEED_AUTOMOUNT

If a dentry has the DCACHE_NEED_AUTOMOUNT flag set (which gets set if the inode has S_AUTOMOUNT set, or can be set directly) then it is (potentially) a mount trap. Any access to this directory beyond a "*stat*" will (normally) cause the *d_op->d_automount()* dentry operation to be called. The task of this method is to find the filesystem that should be mounted on the directory and to return it. The VFS is responsible for actually mounting the root of this filesystem on the directory.

autofs doesn't find the filesystem itself but sends a message to the automount daemon asking it to find and mount the filesystem. The autofs *d_automount* method then waits for the daemon to report that everything is ready. It will then return "*NULL*" indicating that the mount has already happened. The VFS doesn't try to mount anything but follows down the mount that is already there.

This functionality is sufficient for some users of mount traps such as NFS which creates traps so that mountpoints on the server can be reflected on the client. However it is not sufficient for autofs. As mounting onto a directory is considered to be "beyond a *stat*", the automount daemon would not be able to mount a filesystem on the 'trap' directory without some way to avoid getting caught in the trap. For that purpose there is another flag.

DCACHE_MANAGE_TRANSIT

If a dentry has DCACHE_MANAGE_TRANSIT set then two very different but related behaviours are invoked, both using the *d_op->d_manage()* dentry operation.

Firstly, before checking to see if any filesystem is mounted on the directory, d_manage() will be called with the *rcu_walk* parameter set to *false*. It may return one of three things:

- A return value of zero indicates that there is nothing special about this dentry and normal checks for mounts and automounts should proceed.

  autofs normally returns zero, but first waits for any expiry (automatic unmounting of the mounted filesystem) to complete. This avoids races.

- A return value of *-EISDIR* tells the VFS to ignore any mounts on the directory and to not consider calling *->d_automount()*. This effectively disables the **DCACHE_NEED_AUTOMOUNT** flag causing the directory not be a mount trap after all.

  autofs returns this if it detects that the process performing the lookup is the automount daemon and that the mount has been requested but has not yet completed. How it determines this is discussed later. This allows the automount daemon not to get caught in the mount trap.

  There is a subtlety here. It is possible that a second autofs filesystem can be mounted below the first and for both of them to be managed by the same daemon. For the daemon to be able to mount something on the second it must be able to "walk" down past the first. This means that d_manage cannot *always* return -EISDIR for the automount daemon. It must only return it when a mount has been requested, but has not yet completed.

  d_manage also returns *-EISDIR* if the dentry shouldn't be a mount trap, either because it is a symbolic link or because it is not empty.

- Any other negative value is treated as an error and returned to the caller.

  autofs can return

  - -ENOENT if the automount daemon failed to mount anything,
  - -ENOMEM if it ran out of memory,
  - -EINTR if a signal arrived while waiting for expiry to complete
  - or any other error sent down by the automount daemon.

The second use case only occurs during an "RCU-walk" and so *rcu_walk* will be set.

An RCU-walk is a fast and lightweight process for walking down a filename path (i.e. it is like running on tip-toes). RCU-walk cannot cope with all situations so when it finds a difficulty it falls back to "REF-walk", which is slower but more robust.

RCU-walk will never call *->d_automount*; the filesystems must already be mounted or RCU-walk cannot handle the path. To determine if a mount-trap is safe for RCU-walk mode it calls *->d_manage()* with *rcu_walk* set to *true*.

In this case *d_manage()* must avoid blocking and should avoid taking spinlocks if at all possible. Its sole purpose is to determine if it would be safe to follow down into any mounted directory and the only reason that it might not be is if an expiry of the mount is underway.

In the *rcu_walk* case, *d_manage()* cannot return -EISDIR to tell the VFS that this is a directory that doesn't require d_automount. If *rcu_walk* sees a dentry with DCACHE_NEED_AUTOMOUNT set but nothing mounted, it *will* fall back to REF-walk. *d_manage()* cannot make the VFS remain in RCU-walk mode, but can only tell it to get out of RCU-walk mode by returning *-ECHILD*.

So *d_manage()*, when called with *rcu_walk* set, should either return -ECHILD if there is any reason to believe it is unsafe to enter the mounted filesystem, otherwise it should return 0.

autofs will return *-ECHILD* if an expiry of the filesystem has been initiated or is being considered, otherwise it returns 0.

Mountpoint expiry

The VFS has a mechanism for automatically expiring unused mounts, much as it can expire any unused dentry information from the dcache. This is guided by the MNT_SHRINKABLE flag. This only applies to mounts that were created by *d_automount()* returning a filesystem to be mounted. As autofs doesn't return such a filesystem but leaves the mounting to the automount daemon, it must involve the automount daemon in unmounting as well. This also means that autofs has more control over expiry.

The VFS also supports "expiry" of mounts using the MNT_EXPIRE flag to the *umount* system call. Unmounting with MNT_EXPIRE will fail unless a previous attempt had been made, and the filesystem has been inactive and untouched since that previous attempt. autofs does not depend on this but has its own internal tracking of whether filesystems were recently used. This allows individual names in the autofs directory to expire separately.

With version 4 of the protocol, the automount daemon can try to unmount any filesystems mounted on the autofs filesystem or remove any symbolic links or empty directories any time it likes. If the unmount or removal is successful the filesystem will be returned to the state it was before the mount or creation, so that any access of the name will trigger normal auto-mount processing. In particular, *rmdir* and *unlink* do not leave negative entries in the dcache as a normal filesystem would, so an attempt to access a recently-removed object is passed to autofs for handling.

With version 5, this is not safe except for unmounting from top-level directories. As lower-level directories are never mount traps, other processes will see an empty directory as soon as the filesystem is unmounted. So it is generally safest to use the autofs expiry protocol described below.

Normally the daemon only wants to remove entries which haven't been used for a while. For this purpose autofs maintains a "*last_used*" time stamp on each directory or symlink. For symlinks it genuinely does record the last time the symlink was "used" or followed to find out where it points to. For directories the field is used slightly differently. The field is updated at mount time and during expire checks if it is found to be in use (ie. open file descriptor or process working directory) and during path walks. The update done during path walks prevents frequent expire and immediate mount of frequently accessed automounts. But in the case where a GUI continually access or an application frequently scans an autofs directory tree there can be an accumulation of mounts that aren't actually being used. To cater for this case the "*strictexpire*" autofs mount option can be used to avoid the "*last_used*" update on path walk thereby preventing this apparent inability to expire mounts that aren't really in use.

The daemon is able to ask autofs if anything is due to be expired, using an *ioctl* as discussed later. For a *direct* mount, autofs considers if the entire mount-tree can be unmounted or not. For an *indirect* mount, autofs considers each of the names in the top level directory to determine if any of those can be unmounted and cleaned up.

There is an option with indirect mounts to consider each of the leaves that has been mounted on instead of considering the top-level names. This was originally intended for compatibility with version 4 of autofs and should be considered as deprecated for Sun Format automount maps. However, it may be used again for amd format mount maps (which are generally indirect maps) because the amd automounter allows for the setting of an expire timeout for individual mounts. But there are some difficulties in making the needed changes for this.

When autofs considers a directory it checks the *last_used* time and compares it with the "timeout" value set when the filesystem was mounted, though this check is ignored in some cases. It also checks if the directory or anything below it is in use. For symbolic links, only the *last_used* time is ever considered.

If both appear to support expiring the directory or symlink, an action is taken.

There are two ways to ask autofs to consider expiry. The first is to use the **AUTOFS_IOC_EXPIRE** ioctl. This only works for indirect mounts. If it finds something in the root directory to expire it will return the name of that thing. Once a name has been returned the automount daemon needs to unmount any filesystems mounted below the name normally. As described above, this is unsafe for non-toplevel mounts in a version-5 autofs. For this reason the current *automount(8)* does not use this ioctl.

The second mechanism uses either the **AUTOFS_DEV_IOCTL_EXPIRE_CMD** or the **AUTOFS_IOC_EXPIRE_MULTI** ioctl. This will work for both direct and indirect mounts. If it selects an object to expire, it will notify the daemon using the notification mechanism described below. This will block until the daemon acknowledges the expiry notification. This implies that the "*EXPIRE*" ioctl must be sent from a different thread than the one which handles notification.

While the ioctl is blocking, the entry is marked as "expiring" and *d_manage* will block until the daemon affirms that the unmount has completed (together with removing any directories that might have been necessary), or has been aborted.

Communicating with autofs: detecting the daemon

There are several forms of communication between the automount daemon and the filesystem. As we have already seen, the daemon can create and remove directories and symlinks using normal filesystem operations. autofs knows whether a process requesting some operation is the daemon or not based on its process-group id number (see getpgid(1)).

When an autofs filesystem is mounted the pgid of the mounting processes is recorded unless the "pgrp=" option is given, in which case that number is recorded instead. Any request arriving from a process in that process group is considered to come from the daemon. If the daemon ever has to be stopped and restarted a new pgid can be provided through an ioctl as will be described below.

Communicating with autofs: the event pipe

When an autofs filesystem is mounted, the 'write' end of a pipe must be passed using the 'fd=' mount option. autofs will write notification messages to this pipe for the daemon to respond to. For version 5, the format of the message is:

```
struct autofs_v5_packet {
        struct autofs_packet_hdr hdr;
        autofs_wqt_t wait_queue_token;
        __u32 dev;
        __u64 ino;
        __u32 uid;
        __u32 gid;
        __u32 pid;
        __u32 tgid;
        __u32 len;
        char name[NAME_MAX+1];
};
```

And the format of the header is:

```
struct autofs_packet_hdr {
        int proto_version;              /* Protocol version */
        int type;                       /* Type of packet */
};
```

where the type is one of

```
autofs_ptype_missing_indirect
autofs_ptype_expire_indirect
autofs_ptype_missing_direct
autofs_ptype_expire_direct
```

so messages can indicate that a name is missing (something tried to access it but it isn't there) or that it has been selected for expiry.

The pipe will be set to "packet mode" (equivalent to passing *O_DIRECT*) to _pipe2(2)_ so that a read from the pipe will return at most one packet, and any unread portion of a packet will be discarded.

The *wait_queue_token* is a unique number which can identify a particular request to be acknowledged. When a message is sent over the pipe the affected dentry is marked as either "active" or "expiring" and other accesses to it block until the message is acknowledged using one of the ioctls below with the relevant *wait_queue_token*.

Communicating with autofs: root directory ioctls

The root directory of an autofs filesystem will respond to a number of ioctls. The process issuing the ioctl must have the CAP_SYS_ADMIN capability, or must be the automount daemon.

The available ioctl commands are:

- **AUTOFS_IOC_READY**:
  a notification has been handled. The argument to the ioctl command is the "wait_queue_token" number corresponding to the notification being acknowledged.
- **AUTOFS_IOC_FAIL**:
  similar to above, but indicates failure with the error code *ENOENT*.
- **AUTOFS_IOC_CATATONIC**:
  Causes the autofs to enter "catatonic" mode meaning that it stops sending notifications to the daemon. This mode is also entered if a write to the pipe fails.
- **AUTOFS_IOC_PROTOVER**:
  This returns the protocol version in use.
- **AUTOFS_IOC_PROTOSUBVER**:
  Returns the protocol sub-version which is really a version number for the implementation.
- **AUTOFS_IOC_SETTIMEOUT**:
  This passes a pointer to an unsigned long. The value is used to set the timeout for expiry, and the current timeout value is stored back through the pointer.
- **AUTOFS_IOC_ASKUMOUNT**:
  Returns, in the pointed-to *int*, 1 if the filesystem could be unmounted. This is only a hint as the situation could change at any instant. This call can be used to avoid a more expensive full unmount attempt.
- **AUTOFS_IOC_EXPIRE**:

  as described above, this asks if there is anything suitable to expire. A pointer to a packet:

  ```
  struct autofs_packet_expire_multi {
          struct autofs_packet_hdr hdr;
          autofs_wqt_t wait_queue_token;
          int len;
          char name[NAME_MAX+1];
  };
  ```

  is required. This is filled in with the name of something that can be unmounted or removed. If nothing can be expired, *errno* is set to *EAGAIN*. Even though a *wait_queue_token* is present in the structure, no "wait queue" is established and no acknowledgment is needed.
- **AUTOFS_IOC_EXPIRE_MULTI**:

  This is similar to **AUTOFS_IOC_EXPIRE** except that it causes notification to be sent to the daemon, and it blocks until the daemon acknowledges. The argument is an integer which can contain two different flags.

  AUTOFS_EXP_IMMEDIATE causes *last_used* time to be ignored and objects are expired if the are not in use.

  AUTOFS_EXP_FORCED causes the in use status to be ignored and objects are expired ieven if they are in use. This assumes that the daemon has requested this because it is capable of performing the umount.

  AUTOFS_EXP_LEAVES will select a leaf rather than a top-level name to expire. This is only safe when *maxproto* is 4.

Communicating with autofs: char-device ioctls

It is not always possible to open the root of an autofs filesystem, particularly a *direct* mounted filesystem. If the automount daemon is restarted there is no way for it to regain control of existing mounts using any of the above communication channels. To address this need there is a "miscellaneous" character device (major 10, minor 235) which can be used to communicate directly with the autofs filesystem. It requires CAP_SYS_ADMIN for access.

The 'ioctl's that can be used on this device are described in a separate document *autofs-mount-control.txt*, and are summarised briefly here. Each ioctl is passed a pointer to an *autofs_dev_ioctl* structure:

```
struct autofs_dev_ioctl {
        __u32 ver_major;
        __u32 ver_minor;
        __u32 size;             /* total size of data passed in
                                 * including this struct */
        __s32 ioctlfd;          /* automount command fd */

        /* Command parameters */
        union {
                struct args_protover            protover;
                struct args_protosubver         protosubver;
                struct args_openmount           openmount;
                struct args_ready               ready;
                struct args_fail                fail;
                struct args_setpipefd           setpipefd;
                struct args_timeout             timeout;
                struct args_requester           requester;
                struct args_expire              expire;
                struct args_askumount           askumount;
                struct args_ismountpoint        ismountpoint;
        };

        char path[0];
};
```

For the **OPEN_MOUNT** and **IS_MOUNTPOINT** commands, the target filesystem is identified by the *path*. All other commands identify the filesystem by the *ioctlfd* which is a file descriptor open on the root, and which can be returned by **OPEN_MOUNT**.

The *ver_major* and *ver_minor* are in/out parameters which check that the requested version is supported, and report the maximum version that the kernel module can support.

Commands are:

- **AUTOFS_DEV_IOCTL_VERSION_CMD**:
  does nothing, except validate and set version numbers.
- **AUTOFS_DEV_IOCTL_OPENMOUNT_CMD**:
  return an open file descriptor on the root of an autofs filesystem. The filesystem is identified by name and device number, which is stored in *openmount.devid*. Device numbers for existing filesystems can be found in */proc/self/mountinfo*.
- **AUTOFS_DEV_IOCTL_CLOSEMOUNT_CMD**:
  same as *close(ioctlfd)*.
- **AUTOFS_DEV_IOCTL_SETPIPEFD_CMD**:
  if the filesystem is in catatonic mode, this can provide the write end of a new pipe in *setpipefd.pipefd* to re-establish communication with a daemon. The process group of the calling process is used to identify the daemon.
- **AUTOFS_DEV_IOCTL_REQUESTER_CMD**:
  path should be a name within the filesystem that has been auto-mounted on. On successful return, *requester.uid* and *requester.gid* will be the UID and GID of the process which triggered that mount.
- **AUTOFS_DEV_IOCTL_ISMOUNTPOINT_CMD**:
  Check if path is a mountpoint of a particular type - see separate documentation for details.

- **AUTOFS_DEV_IOCTL_PROTOVER_CMD**
- **AUTOFS_DEV_IOCTL_PROTOSUBVER_CMD**
- **AUTOFS_DEV_IOCTL_READY_CMD**
- **AUTOFS_DEV_IOCTL_FAIL_CMD**
- **AUTOFS_DEV_IOCTL_CATATONIC_CMD**
- **AUTOFS_DEV_IOCTL_TIMEOUT_CMD**
- **AUTOFS_DEV_IOCTL_EXPIRE_CMD**
- **AUTOFS_DEV_IOCTL_ASKUMOUNT_CMD**

These all have the same function as the similarly named **AUTOFS_IOC** ioctls, except that **FAIL** can be given an explicit error number in *fail.status* instead of assuming *ENOENT*, and this **EXPIRE** command corresponds to

AUTOFS_IOC_EXPIRE_MULTI.

Catatonic mode

As mentioned, an autofs mount can enter "catatonic" mode. This happens if a write to the notification pipe fails, or if it is explicitly requested by an *ioctl*.

When entering catatonic mode, the pipe is closed and any pending notifications are acknowledged with the error *ENOENT*.

Once in catatonic mode attempts to access non-existing names will result in *ENOENT* while attempts to access existing directories will be treated in the same way as if they came from the daemon, so mount traps will not fire.

When the filesystem is mounted a _uid_ and _gid_ can be given which set the ownership of directories and symbolic links. When the filesystem is in catatonic mode, any process with a matching UID can create directories or symlinks in the root directory, but not in other directories.

Catatonic mode can only be left via the **AUTOFS_DEV_IOCTL_OPENMOUNT_CMD** ioctl on the */dev/autofs*.

The "ignore" mount option

The "ignore" mount option can be used to provide a generic indicator to applications that the mount entry should be ignored when displaying mount information.

In other OSes that provide autofs and that provide a mount list to user space based on the kernel mount list a no-op mount option ("ignore" is the one use on the most common OSes) is allowed so that autofs file system users can optionally use it.

This is intended to be used by user space programs to exclude autofs mounts from consideration when reading the mounts list.

autofs, name spaces, and shared mounts

With bind mounts and name spaces it is possible for an autofs filesystem to appear at multiple places in one or more filesystem name spaces. For this to work sensibly, the autofs filesystem should always be mounted "shared". e.g.

```
mount --make-shared /autofs/mount/point
```

The automount daemon is only able to manage a single mount location for an autofs filesystem and if mounts on that are not 'shared', other locations will not behave as expected. In particular access to those other locations will likely result in the *ELOOP* error

```
Too many levels of symbolic links
```