# Miscellaneous Device control operations for the autofs kernel module

## The problem

There is a problem with active restarts in autofs (that is to say restarting autofs when there are busy mounts).

During normal operation autofs uses a file descriptor opened on the directory that is being managed in order to be able to issue control operations. Using a file descriptor gives ioctl operations access to autofs specific information stored in the super block. The operations are things such as setting an autofs mount catatonic, setting the expire timeout and requesting expire checks. As is explained below, certain types of autofs triggered mounts can end up covering an autofs mount itself which prevents us being able to use open(2) to obtain a file descriptor for these operations if we don't already have one open.

Currently autofs uses "umount -l" (lazy umount) to clear active mounts at restart. While using lazy umount works for most cases, anything that needs to walk back up the mount tree to construct a path, such as getcwd(2) and the proc file system /proc/<pid>/cwd, no longer works because the point from which the path is constructed has been detached from the mount tree.

The actual problem with autofs is that it can't reconnect to existing mounts. Immediately one thinks of just adding the ability to remount autofs file systems would solve it, but alas, that can't work. This is because autofs direct mounts and the implementation of "on demand mount and expire" of nested mount trees have the file system mounted directly on top of the mount trigger directory dentry.

For example, there are two types of automount maps, direct (in the kernel module source you will see a third type called an offset, which is just a direct mount in disguise) and indirect.

Here is a master map with direct and indirect map entries:

```
/-      /etc/auto.direct
/test   /etc/auto.indirect
```

and the corresponding map files:

```
/etc/auto.direct:

/automount/dparse/g6  budgie:/autofs/export1
/automount/dparse/g1  shark:/autofs/export1
and so on.
```

/etc/auto.indirect:

```
g1    shark:/autofs/export1
g6    budgie:/autofs/export1
and so on.
```

For the above indirect map an autofs file system is mounted on /test and mounts are triggered for each sub-directory key by the inode lookup operation. So we see a mount of shark:/autofs/export1 on /test/g1, for example.

The way that direct mounts are handled is by making an autofs mount on each full path, such as /automount/dparse/g1, and using it as a mount trigger. So when we walk on the path we mount shark:/autofs/export1 "on top of this mount point". Since these are always directories we can use the follow_link inode operation to trigger the mount.

But, each entry in direct and indirect maps can have offsets (making them multi-mount map entries).

For example, an indirect mount map entry could also be:

```
g1  \
/        shark:/autofs/export5/testing/test \
/s1      shark:/autofs/export/testing/test/s1 \
/s2      shark:/autofs/export5/testing/test/s2 \
/s1/ss1  shark:/autofs/export1 \
/s2/ss2  shark:/autofs/export2
```

and a similarly a direct mount map entry could also be:

```
/automount/dparse/g1 \
    /       shark:/autofs/export5/testing/test \
    /s1     shark:/autofs/export/testing/test/s1 \
    /s2     shark:/autofs/export5/testing/test/s2 \
    /s1/ss1 shark:/autofs/export2 \
    /s2/ss2 shark:/autofs/export2
```

One of the issues with version 4 of autofs was that, when mounting an entry with a large number of offsets, possibly with nesting, we needed to mount and umount all of the offsets as a single unit. Not really a problem, except for people with a large number of offsets in map entries. This mechanism is used for the well known "hosts" map and we have seen cases (in 2.4) where the available number of mounts are exhausted or where the number of privileged ports available is exhausted.

In version 5 we mount only as we go down the tree of offsets and similarly for expiring them which resolves the above problem. There is somewhat more detail to the implementation but it isn't needed for the sake of the problem explanation. The one important detail is that these offsets are implemented using the same mechanism as the direct mounts above and so the mount points can be covered by a mount.

The current autofs implementation uses an ioctl file descriptor opened on the mount point for control operations. The references held by the descriptor are accounted for in checks made to determine if a mount is in use and is also used to access autofs file system information held in the mount super block. So the use of a file handle needs to be retained.

## The Solution

To be able to restart autofs leaving existing direct, indirect and offset mounts in place we need to be able to obtain a file handle for these potentially covered autofs mount points. Rather than just implement an isolated operation it was decided to re-implement the existing ioctl interface and add new operations to provide this functionality.

In addition, to be able to reconstruct a mount tree that has busy mounts, the uid and gid of the last user that triggered the mount needs to be available because these can be used as macro substitution variables in autofs maps. They are recorded at mount request time and an operation has been added to retrieve them.

Since we're re-implementing the control interface, a couple of other problems with the existing interface have been addressed. First, when a mount or expire operation completes a status is returned to the kernel by either a "send ready" or a "send fail" operation. The "send fail" operation of the ioctl interface could only ever send ENOENT so the re-implementation allows user space to send an actual status. Another expensive operation in user space, for those using very large maps, is discovering if a mount is present. Usually this involves scanning /proc/mounts and since it needs to be done quite often it can introduce significant overhead when there are many entries in the mount table. An operation to lookup the mount status of a mount point dentry (covered or not) has also been added.

Current kernel development policy recommends avoiding the use of the ioctl mechanism in favor of systems such as Netlink. An implementation using this system was attempted to evaluate its suitability and it was found to be inadequate, in this case. The Generic Netlink system was used for this as raw Netlink would lead to a significant increase in complexity. There's no question that the Generic Netlink system is an elegant solution for common case ioctl functions but it's not a complete replacement probably because its primary purpose in life is to be a message bus implementation rather than specifically an ioctl replacement. While it would be possible to work around this there is one concern that lead to the decision to not use it. This is that the autofs expire in the daemon has become far to complex because umount candidates are enumerated, almost for no other reason than to "count" the number of times to call the expire ioctl. This involves scanning the mount table which has proved to be a big overhead for users with large maps. The best way to improve this is try and get back to the way the expire was done long ago. That is, when an expire request is issued for a mount (file handle) we should continually call back to the daemon until we can't umount any more mounts, then return the appropriate status to the daemon. At the moment we just expire one mount at a time. A Generic Netlink implementation would exclude this possibility for future development due to the requirements of the message bus architecture.

## autofs Miscellaneous Device mount control interface

The control interface is opening a device node, typically /dev/autofs.

All the ioctls use a common structure to pass the needed parameter information and return operation results:

```
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];
};
```

The ioctlfd field is a mount point file descriptor of an autofs mount point. It is returned by the open call and is used by all calls except the check for whether a given path is a mount point, where it may optionally be used to check a specific mount corresponding to a given mount point file descriptor, and when requesting the uid and gid of the last successful mount on a directory within the autofs file system.

The union is used to communicate parameters and results of calls made as described below.

The path field is used to pass a path where it is needed and the size field is used account for the increased structure length when translating the structure sent from user space.

This structure can be initialized before setting specific fields by using the void function call init_autofs_dev_ioctl(`struct autofs_dev_ioctl *`).

All of the ioctls perform a copy of this structure from user space to kernel space and return -EINVAL if the size parameter is smaller than the structure size itself, -ENOMEM if the kernel memory allocation fails or -EFAULT if the copy itself fails. Other checks include a version check of the compiled in user space version against the module version and a mismatch results in a -EINVAL return. If the size field is greater than the structure size then a path is assumed to be present and is checked to ensure it begins with a "/" and is NULL terminated, otherwise -EINVAL is returned. Following these checks, for all ioctl commands except AUTOFS_DEV_IOCTL_VERSION_CMD, AUTOFS_DEV_IOCTL_OPENMOUNT_CMD and AUTOFS_DEV_IOCTL_CLOSEMOUNT_CMD the ioctlfd is validated and if it is not a valid descriptor or doesn't correspond to an autofs mount point an error of -EBADF, -ENOTTY or -EINVAL (not an autofs descriptor) is returned.

## The ioctls

An example of an implementation which uses this interface can be seen in autofs version 5.0.4 and later in file lib/dev-ioctl-lib.c of the distribution tar available for download from kernel.org in directory /pub/linux/daemons/autofs/v5.

The device node ioctl operations implemented by this interface are:

### AUTOFS_DEV_IOCTL_VERSION

Get the major and minor version of the autofs device ioctl kernel module implementation. It requires an initialized struct autofs_dev_ioctl as an input parameter and sets the version information in the passed in structure. It returns 0 on success or the error -EINVAL if a version mismatch is detected.

### AUTOFS_DEV_IOCTL_PROTOVER_CMD and AUTOFS_DEV_IOCTL_PROTOSUBVER_CMD

Get the major and minor version of the autofs protocol version understood by loaded module. This call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to a valid autofs mount point descriptor and sets the requested version number in version field of struct args_protover or sub_version field of struct args_protosubver. These commands return 0 on success or one of the negative error codes if validation fails.

### AUTOFS_DEV_IOCTL_OPENMOUNT and AUTOFS_DEV_IOCTL_CLOSEMOUNT

Obtain and release a file descriptor for an autofs managed mount point path. The open call requires an initialized struct autofs_dev_ioctl with the path field set and the size field adjusted appropriately as well as the devid field of struct args_openmount set to the device number of the autofs mount. The device number can be obtained from the mount options shown in /proc/mounts. The close call requires an initialized struct autofs_dev_ioct with the ioctlfd field set to the descriptor obtained from the open call. The release of the file descriptor can also be done with close(2) so any open descriptors will also be closed at process exit. The close call is included in the implemented operations largely for completeness and to provide for a consistent user space implementation.

### AUTOFS_DEV_IOCTL_READY_CMD and AUTOFS_DEV_IOCTL_FAIL_CMD

Return mount and expire result status from user space to the kernel. Both of these calls require an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call and the token field of struct args_ready or struct args_fail set to the wait queue token number, received by user space in the foregoing mount or expire request. The status field of struct args_fail is set to the errno of the operation. It is set to 0 on success.

### AUTOFS_DEV_IOCTL_SETPIPEFD_CMD

Set the pipe file descriptor used for kernel communication to the daemon. Normally this is set at mount time using an option but when reconnecting to a existing mount we need to use this to tell the autofs mount about the new kernel pipe descriptor. In order to protect mounts against incorrectly setting the pipe descriptor we also require that the autofs mount be catatonic (see next call).

The call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call and the pipefd field of struct args_setpipefd set to descriptor of the pipe. On success the call also sets the process group id used to identify the controlling process (eg. the owning automount(8) daemon) to the process group of the caller.

### AUTOFS_DEV_IOCTL_CATATONIC_CMD

Make the autofs mount point catatonic. The autofs mount will no longer issue mount requests, the kernel communication pipe descriptor is released and any remaining waits in the queue released.

The call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call.

### AUTOFS_DEV_IOCTL_TIMEOUT_CMD

Set the expire timeout for mounts within an autofs mount point.

The call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call.

### AUTOFS_DEV_IOCTL_REQUESTER_CMD

Return the uid and gid of the last process to successfully trigger a the mount on the given path dentry.

The call requires an initialized struct autofs_dev_ioctl with the path field set to the mount point in question and the size field adjusted appropriately. Upon return the uid field of struct args_requester contains the uid and gid field the gid.

When reconstructing an autofs mount tree with active mounts we need to re-connect to mounts that may have used the original process uid and gid (or string variations of them) for mount lookups within the map entry. This call provides the ability to obtain this uid and gid so they may be used by user space for the mount map lookups.

### AUTOFS_DEV_IOCTL_EXPIRE_CMD

Issue an expire request to the kernel for an autofs mount. Typically this ioctl is called until no further expire candidates are found.

The call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call. In addition an immediate expire that's independent of the mount timeout, and a forced expire that's independent of whether the mount is busy, can be requested by setting the how field of struct args_expire to AUTOFS_EXP_IMMEDIATE or AUTOFS_EXP_FORCED, respectively . If no expire candidates can be found the ioctl returns -1 with errno set to EAGAIN.

This call causes the kernel module to check the mount corresponding to the given ioctlfd for mounts that can be expired, issues an expire request back to the daemon and waits for completion.

### AUTOFS_DEV_IOCTL_ASKUMOUNT_CMD

Checks if an autofs mount point is in use.

The call requires an initialized struct autofs_dev_ioctl with the ioctlfd field set to the descriptor obtained from the open call and it returns the result in the may_umount field of struct args_askumount, 1 for busy and 0 otherwise.

### AUTOFS_DEV_IOCTL_ISMOUNTPOINT_CMD

Check if the given path is a mountpoint.

The call requires an initialized struct autofs_dev_ioctl. There are two possible variations. Both use the path field set to the path of the mount point to check and the size field adjusted appropriately. One uses the ioctlfd field to identify a specific mount point to check while the other variation uses the path and optionally in.type field of struct args_ismountpoint set to an autofs mount type. The call returns 1 if this is a mount point and sets out.devid field to the device number of the mount and out.magic field to the relevant super block magic number (described below) or 0 if it isn't a mountpoint. In both cases the device number (as returned by new_encode_dev()) is returned in out.devid field.

If supplied with a file descriptor we're looking for a specific mount, not necessarily at the top of the mounted stack. In this case the path the descriptor corresponds to is considered a mountpoint if it is itself a mountpoint or contains a mount, such as a multi-mount without a root mount. In this case we return 1 if the descriptor corresponds to a mount point and also returns the super magic of the covering mount if there is one or 0 if it isn't a mountpoint.

If a path is supplied (and the ioctlfd field is set to -1) then the path is looked up and is checked to see if it is the root of a mount. If a type is also given we are looking for a particular autofs mount and if a match isn't found a fail is returned. If the located path is the root of a mount 1 is returned along with the super magic of the mount or 0 otherwise.