Filesystem Mount API

Overview

The creation of new mounts is now to be done in a multistep process:

1. Create a filesystem context.
2. Parse the parameters and attach them to the context. Parameters are expected to be passed individually from userspace, though legacy binary parameters can also be handled.
3. Validate and pre-process the context.
4. Get or create a superblock and mountable root.
5. Perform the mount.
6. Return an error message attached to the context.
7. Destroy the context.

To support this, the file_system_type struct gains two new fields:

```
int (*init_fs_context)(struct fs_context *fc);
const struct fs_parameter_description *parameters;
```

The first is invoked to set up the filesystem-specific parts of a filesystem context, including the additional space, and the second points to the parameter description for validation at registration time and querying by a future system call.

Note that security initialisation is done *after* the filesystem is called so that the namespaces may be adjusted first.

The Filesystem context

The creation and reconfiguration of a superblock is governed by a filesystem context. This is represented by the fs_context structure:

```
struct fs_context {
        const struct fs_context_operations *ops;
        struct file_system_type *fs_type;
        void                    *fs_private;
        struct dentry           *root;
        struct user_namespace   *user_ns;
        struct net              *net_ns;
        const struct cred       *cred;
        char                    *source;
        char                    *subtype;
        void                    *security;
        void                    *s_fs_info;
        unsigned int            sb_flags;
        unsigned int            sb_flags_mask;
        unsigned int            s_iflags;
        unsigned int            lsm_flags;
        enum fs_context_purpose purpose:8;
        ...
};
```

The fs_context fields are as follows:

- `const struct fs_context_operations *ops`

  These are operations that can be done on a filesystem context (see below). This must be set by the ->init_fs_context() file_system_type operation.

- `struct file_system_type *fs_type`

  A pointer to the file_system_type of the filesystem that is being constructed or reconfigured. This retains a reference on the type owner.

- `void *fs_private`

  A pointer to the file system's private data. This is where the filesystem will need to store any options it parses.

- `struct dentry *root`

  A pointer to the root of the mountable tree (and indirectly, the superblock thereof). This is filled in by the ->get_tree() op. If this is set, an active reference on root->d_sb must also be held.

- `struct user_namespace *user_ns`
  `struct net *net_ns`

There are a subset of the namespaces in use by the invoking process. They retain references on each namespace. The subscribed namespaces may be replaced by the filesystem to reflect other sources, such as the parent mount superblock on an automount.

- `const struct cred *cred`

  The mounter's credentials. This retains a reference on the credentials.

- `char *source`

  This specifies the source. It may be a block device (e.g. /dev/sda1) or something more exotic, such as the "host:/path" that NFS desires.

- `char *subtype`

  This is a string to be added to the type displayed in /proc/mounts to qualify it (used by FUSE). This is available for the filesystem to set if desired.

- `void *security`

  A place for the LSMs to hang their security data for the superblock. The relevant security operations are described below.

- `void *s_fs_info`

  The proposed s_fs_info for a new superblock, set in the superblock by sget_fc(). This can be used to distinguish superblocks.

- `unsigned int sb_flags`
  `unsigned int sb_flags_mask`

  Which bits SB_* flags are to be set/cleared in super_block::s_flags.

- `unsigned int s_iflags`

  These will be bitwise-OR'd with s->s_iflags when a superblock is created.

- `enum fs_context_purpose`

  This indicates the purpose for which the context is intended. The available values are:

FS_CONTEXT_FOR_MOUNT,	New superblock for explicit mount
FS_CONTEXT_FOR_SUBMOUNT	New automatic submount of extant mount
FS_CONTEXT_FOR_RECONFIGURE	Change an existing mount

The mount context is created by calling vfs_new_fs_context() or vfs_dup_fs_context() and is destroyed with put_fs_context(). Note that the structure is not refcounted.

VFS, security and filesystem mount options are set individually with vfs_parse_mount_option(). Options provided by the old mount(2) system call as a page of data can be parsed with generic_parse_monolithic().

When mounting, the filesystem is allowed to take data from any of the pointers and attach it to the superblock (or whatever), provided it clears the pointer in the mount context.

The filesystem is also allowed to allocate resources and pin them with the mount context. For instance, NFS might pin the appropriate protocol version module.

The Filesystem Context Operations

The filesystem context points to a table of operations:

```
struct fs_context_operations {
        void (*free)(struct fs_context *fc);
        int (*dup)(struct fs_context *fc, struct fs_context *src_fc);
        int (*parse_param)(struct fs_context *fc,
                           struct fs_parameter *param);
        int (*parse_monolithic)(struct fs_context *fc, void *data);
        int (*get_tree)(struct fs_context *fc);
        int (*reconfigure)(struct fs_context *fc);
};
```

These operations are invoked by the various stages of the mount procedure to manage the filesystem context. They are as follows:

- `void (*free)(struct fs_context *fc);`

  Called to clean up the filesystem-specific part of the filesystem context when the context is destroyed. It should be

aware that parts of the context may have been removed and NULL'd out by ->get_tree().

- `int (*dup)(struct fs_context *fc, struct fs_context *src_fc);`

  Called when a filesystem context has been duplicated to duplicate the filesystem-private data. An error may be returned to indicate failure to do this.

  > **Warning**
  >
  > Note that even if this fails, put_fs_context() will be called immediately thereafter, so ->dup() *must* make the filesystem-private data safe for ->free().

- ```
  int (*parse_param)(struct fs_context *fc,
                     struct fs_parameter *param);
  ```

  Called when a parameter is being added to the filesystem context. param points to the key name and maybe a value object. VFS-specific options will have been weeded out and fc->sb_flags updated in the context. Security options will also have been weeded out and fc->security updated.

  The parameter can be parsed with fs_parse() and fs_lookup_param(). Note that the source(s) are presented as parameters named "source".

  If successful, 0 should be returned or a negative error code otherwise.

- `int (*parse_monolithic)(struct fs_context *fc, void *data);`

  Called when the mount(2) system call is invoked to pass the entire data page in one go. If this is expected to be just a list of "key[=val]" items separated by commas, then this may be set to NULL.

  The return value is as for ->parse_param().

  If the filesystem (e.g. NFS) needs to examine the data first and then finds it's the standard key-val list then it may pass it off to generic_parse_monolithic().

- `int (*get_tree)(struct fs_context *fc);`

  Called to get or create the mountable root and superblock, using the information stored in the filesystem context (reconfiguration goes via a different vector). It may detach any resources it desires from the filesystem context and transfer them to the superblock it creates.

  On success it should set fc->root to the mountable root and return 0. In the case of an error, it should return a negative error code.

  The phase on a userspace-driven context will be set to only allow this to be called once on any particular context.

- `int (*reconfigure)(struct fs_context *fc);`

  Called to effect reconfiguration of a superblock using information stored in the filesystem context. It may detach any resources it desires from the filesystem context and transfer them to the superblock. The superblock can be found from fc->root->d_sb.

  On success it should return 0. In the case of an error, it should return a negative error code.

  > **Note**
  >
  > reconfigure is intended as a replacement for remount_fs.

Filesystem context Security

The filesystem context contains a security pointer that the LSMs can use for building up a security context for the superblock to be mounted. There are a number of operations used by the new mount code for this purpose:

- ```
  int security_fs_context_alloc(struct fs_context *fc,
                                struct dentry *reference);
  ```

  Called to initialise fc->security (which is preset to NULL) and allocate any resources needed. It should return 0 on success or a negative error code on failure.

  reference will be non-NULL if the context is being created for superblock reconfiguration (FS_CONTEXT_FOR_RECONFIGURE) in which case it indicates the root dentry of the superblock to be reconfigured. It will also be non-NULL in the case of a submount (FS_CONTEXT_FOR_SUBMOUNT) in which case it indicates the automount point.

- `int security_fs_context_dup(struct fs_context *fc,`

```
                                        struct fs_context *src_fc);
```

Called to initialise fc->security (which is preset to NULL) and allocate any resources needed. The original filesystem context is pointed to by src_fc and may be used for reference. It should return 0 on success or a negative error code on failure.

- ```
  void security_fs_context_free(struct fs_context *fc);
  ```

  Called to clean up anything attached to fc->security. Note that the contents may have been transferred to a superblock and the pointer cleared during get_tree.

- ```
  int security_fs_context_parse_param(struct fs_context *fc,
                                      struct fs_parameter *param);
  ```

  Called for each mount parameter, including the source. The arguments are as for the ->parse_param() method. It should return 0 to indicate that the parameter should be passed on to the filesystem, 1 to indicate that the parameter should be discarded or an error to indicate that the parameter should be rejected.

  The value pointed to by param may be modified (if a string) or stolen (provided the value pointer is NULL'd out). If it is stolen, 1 must be returned to prevent it being passed to the filesystem.

- ```
  int security_fs_context_validate(struct fs_context *fc);
  ```

  Called after all the options have been parsed to validate the collection as a whole and to do any necessary allocation so that security_sb_get_tree() and security_sb_reconfigure() are less likely to fail. It should return 0 or a negative error code.

  In the case of reconfiguration, the target superblock will be accessible via fc->root.

- ```
  int security_sb_get_tree(struct fs_context *fc);
  ```

  Called during the mount procedure to verify that the specified superblock is allowed to be mounted and to transfer the security data there. It should return 0 or a negative error code.

- ```
  void security_sb_reconfigure(struct fs_context *fc);
  ```

  Called to apply any reconfiguration to an LSM's context. It must not fail. Error checking and resource allocation must be done in advance by the parameter parsing and validation hooks.

- ```
  int security_sb_mountpoint(struct fs_context *fc,
                             struct path *mountpoint,
                             unsigned int mnt_flags);
  ```

  Called during the mount procedure to verify that the root dentry attached to the context is permitted to be attached to the specified mountpoint. It should return 0 on success or a negative error code on failure.

VFS Filesystem context API

There are four operations for creating a filesystem context and one for destroying a context:

- ```
  struct fs_context *fs_context_for_mount(struct file_system_type *fs_type,
                                          unsigned int sb_flags);
  ```

  Allocate a filesystem context for the purpose of setting up a new mount, whether that be with a new superblock or sharing an existing one. This sets the superblock flags, initialises the security and calls fs_type->init_fs_context() to initialise the filesystem private data.

  fs_type specifies the filesystem type that will manage the context and sb_flags presets the superblock flags stored therein.

- ```
  struct fs_context *fs_context_for_reconfigure(
          struct dentry *dentry,
          unsigned int sb_flags,
          unsigned int sb_flags_mask);
  ```

  Allocate a filesystem context for the purpose of reconfiguring an existing superblock. dentry provides a reference to the superblock to be configured. sb_flags and sb_flags_mask indicate which superblock flags need changing and to what.

- ```
  struct fs_context *fs_context_for_submount(
          struct file_system_type *fs_type,
          struct dentry *reference);
  ```

  Allocate a filesystem context for the purpose of creating a new mount for an automount point or other derived superblock. fs_type specifies the filesystem type that will manage the context and the reference dentry supplies the
  ```

parameters. Namespaces are propagated from the reference dentry's superblock also.

Note that it's not a requirement that the reference dentry be of the same filesystem type as fs_type.

- `struct fs_context *vfs_dup_fs_context(struct fs_context *src_fc);`

  Duplicate a filesystem context, copying any options noted and duplicating or additionally referencing any resources held therein. This is available for use where a filesystem has to get a mount within a mount, such as NFS4 does by internally mounting the root of the target server and then doing a private pathwalk to the target directory.

  The purpose in the new context is inherited from the old one.

- `void put_fs_context(struct fs_context *fc);`

  Destroy a filesystem context, releasing any resources it holds. This calls the ->free() operation. This is intended to be called by anyone who created a filesystem context.

  > **Warning**
  >
  > filesystem contexts are not refcounted, so this causes unconditional destruction.

In all the above operations, apart from the put op, the return is a mount context pointer or a negative error code.

For the remaining operations, if an error occurs, a negative error code will be returned.

- ```
  int vfs_parse_fs_param(struct fs_context *fc,
                         struct fs_parameter *param);
  ```

  Supply a single mount parameter to the filesystem context. This includes the specification of the source/device which is specified as the "source" parameter (which may be specified multiple times if the filesystem supports that).

  param specifies the parameter key name and the value. The parameter is first checked to see if it corresponds to a standard mount flag (in which case it is used to set an SB_xxx flag and consumed) or a security option (in which case the LSM consumes it) before it is passed on to the filesystem.

  The parameter value is typed and can be one of:

fs_value_is_flag	Parameter not given a value
fs_value_is_string	Value is a string
fs_value_is_blob	Value is a binary blob
fs_value_is_filename	Value is a filename* + dirfd
fs_value_is_file	Value is an open file (file*)

  If there is a value, that value is stored in a union in the struct in one of param->{string,blob,name,file}. Note that the function may steal and clear the pointer, but then becomes responsible for disposing of the object.

- ```
  int vfs_parse_fs_string(struct fs_context *fc, const char *key,
                          const char *value, size_t v_size);
  ```

  A wrapper around vfs_parse_fs_param() that copies the value string it is passed.

- `int generic_parse_monolithic(struct fs_context *fc, void *data);`

  Parse a sys_mount() data page, assuming the form to be a text list consisting of key[=val] options separated by commas. Each item in the list is passed to vfs_mount_option(). This is the default when the ->parse_monolithic() method is NULL.

- `int vfs_get_tree(struct fs_context *fc);`

  Get or create the mountable root and superblock, using the parameters in the filesystem context to select/configure the superblock. This invokes the ->get_tree() method.

- `struct vfsmount *vfs_create_mount(struct fs_context *fc);`

  Create a mount given the parameters in the specified filesystem context. Note that this does not attach the mount to anything.

Superblock Creation Helpers

A number of VFS helpers are available for use by filesystems for the creation or looking up of superblocks.

- ```
  struct super_block *
  sget_fc(struct fs_context *fc,
          int (*test)(struct super_block *sb, struct fs_context *fc),
  ```
  ```

```
                int (*set)(struct super_block *sb, struct fs_context *fc));
```

This is the core routine. If test is non-NULL, it searches for an existing superblock matching the criteria held in the fs_context, using the test function to match them. If no match is found, a new superblock is created and the set function is called to set it up.

Prior to the set function being called, fc->s_fs_info will be transferred to sb->s_fs_info - and fc->s_fs_info will be cleared if set returns success (ie. 0).

The following helpers all wrap sget_fc():

- ```
  int vfs_get_super(struct fs_context *fc,
                    enum vfs_get_super_keying keying,
                    int (*fill_super)(struct super_block *sb,
                                      struct fs_context *fc))
  ```

  This creates/looks up a deviceless superblock. The keying indicates how many superblocks of this type may exist and in what manner they may be shared:

  1. vfs_get_single_super

     Only one such superblock may exist in the system. Any further attempt to get a new superblock gets this one (and any parameter differences are ignored).

  2. vfs_get_keyed_super

     Multiple superblocks of this type may exist and they're keyed on their s_fs_info pointer (for example this may refer to a namespace).

  3. vfs_get_independent_super

     Multiple independent superblocks of this type may exist. This function never matches an existing one and always creates a new one.

Parameter Description

Parameters are described using structures defined in linux/fs_parser.h. There's a core description struct that links everything together:

```
struct fs_parameter_description {
        const struct fs_parameter_spec *specs;
        const struct fs_parameter_enum *enums;
};
```

For example:

```
enum {
        Opt_autocell,
        Opt_bar,
        Opt_dyn,
        Opt_foo,
        Opt_source,
};

static const struct fs_parameter_description afs_fs_parameters = {
        .specs          = afs_param_specs,
        .enums          = afs_param_enums,
};
```

The members are as follows:

1. ```
   const struct fs_parameter_specification *specs;
   ```

   Table of parameter specifications, terminated with a null entry, where the entries are of type:

   ```
   struct fs_parameter_spec {
           const char              *name;
           u8                      opt;
           enum fs_parameter_type  type:8;
           unsigned short          flags;
   };
   ```

   The 'name' field is a string to match exactly to the parameter key (no wildcards, patterns and no case-independence) and 'opt' is the value that will be returned by the fs_parser() function in the case of a successful match.

   The 'type' field indicates the desired value type and must be one of:

TYPE NAME	EXPECTED VALUE	RESULT IN

TYPE NAME	EXPECTED VALUE	RESULT IN
fs_param_is_flag	No value	n/a
fs_param_is_bool	Boolean value	result->boolean
fs_param_is_u32	32-bit unsigned int	result->uint_32
fs_param_is_u32_octal	32-bit octal int	result->uint_32
fs_param_is_u32_hex	32-bit hex int	result->uint_32
fs_param_is_s32	32-bit signed int	result->int_32
fs_param_is_u64	64-bit unsigned int	result->uint_64
fs_param_is_enum	Enum value name	result->uint_32
fs_param_is_string	Arbitrary string	param->string
fs_param_is_blob	Binary blob	param->blob
fs_param_is_blockdev	Blockdev path	◦ Needs lookup
fs_param_is_path	Path	◦ Needs lookup
fs_param_is_fd	File descriptor	result->int_32

Note that if the value is of fs_param_is_bool type, fs_parse() will try to match any string value against "0", "1", "no", "yes", "false", "true".

Each parameter can also be qualified with 'flags':

fs_param_v_optional	The value is optional
fs_param_neg_with_no	result->negated set if key is prefixed with "no"
fs_param_neg_with_empty	result->negated set if value is ""
fs_param_deprecated	The parameter is deprecated.

These are wrapped with a number of convenience wrappers:

MACRO	SPECIFIES
fsparam_flag()	fs_param_is_flag
fsparam_flag_no()	fs_param_is_flag, fs_param_neg_with_no
fsparam_bool()	fs_param_is_bool
fsparam_u32()	fs_param_is_u32
fsparam_u32oct()	fs_param_is_u32_octal
fsparam_u32hex()	fs_param_is_u32_hex
fsparam_s32()	fs_param_is_s32
fsparam_u64()	fs_param_is_u64
fsparam_enum()	fs_param_is_enum
fsparam_string()	fs_param_is_string
fsparam_blob()	fs_param_is_blob
fsparam_bdev()	fs_param_is_blockdev
fsparam_path()	fs_param_is_path
fsparam_fd()	fs_param_is_fd

all of which take two arguments, name string and option number - for example:

```
static const struct fs_parameter_spec afs_param_specs[] = {
        fsparam_flag    ("autocell",    Opt_autocell),
        fsparam_flag    ("dyn",         Opt_dyn),
        fsparam_string  ("source",      Opt_source),
        fsparam_flag_no ("foo",         Opt_foo),
        {}
};
```

An addition macro, __fsparam() is provided that takes an additional pair of arguments to specify the type and the flags for anything that doesn't match one of the above macros.

2. `const struct fs_parameter_enum *enums;`

Table of enum value names to integer mappings, terminated with a null entry. This is of type:

```
struct fs_parameter_enum {
        u8              opt;
        char            name[14];
        u8              value;
};
```

Where the array is an unsorted list of { parameter ID, name }-keyed elements that indicate the value to map to,

e.g.:

```
static const struct fs_parameter_enum afs_param_enums[] = {
        { Opt_bar,   "x",       1},
        { Opt_bar,   "y",       23},
        { Opt_bar,   "z",       42},
};
```

If a parameter of type fs_param_is_enum is encountered, fs_parse() will try to look the value up in the enum table and the result will be stored in the parse result.

The parser should be pointed to by the parser pointer in the file_system_type struct as this will provide validation on registration (if CONFIG_VALIDATE_FS_PARSER=y) and will allow the description to be queried from userspace using the fsinfo() syscall.

Parameter Helper Functions

A number of helper functions are provided to help a filesystem or an LSM process the parameters it is given.

- ```
  int lookup_constant(const struct constant_table tbl[],
                      const char *name, int not_found);
  ```

  Look up a constant by name in a table of name -> integer mappings. The table is an array of elements of the following type:

  ```
  struct constant_table {
          const char      *name;
          int             value;
  };
  ```

  If a match is found, the corresponding value is returned. If a match isn't found, the not_found value is returned instead.

- ```
  bool validate_constant_table(const struct constant_table *tbl,
                               size_t tbl_size,
                               int low, int high, int special);
  ```

  Validate a constant table. Checks that all the elements are appropriately ordered, that there are no duplicates and that the values are between low and high inclusive, though provision is made for one allowable special value outside of that range. If no special value is required, special should just be set to lie inside the low-to-high range.

  If all is good, true is returned. If the table is invalid, errors are logged to the kernel log buffer and false is returned.

- ```
  bool fs_validate_description(const struct fs_parameter_description *desc);
  ```

  This performs some validation checks on a parameter description. It returns true if the description is good and false if it is not. It will log errors to the kernel log buffer if validation fails.

- ```
  int fs_parse(struct fs_context *fc,
               const struct fs_parameter_description *desc,
               struct fs_parameter *param,
               struct fs_parse_result *result);
  ```

  This is the main interpreter of parameters. It uses the parameter description to look up a parameter by key name and to convert that to an option number (which it returns).

  If successful, and if the parameter type indicates the result is a boolean, integer or enum type, the value is converted by this function and the result stored in result->{boolean,int_32,uint_32,uint_64}.

  If a match isn't initially made, the key is prefixed with "no" and no value is present then an attempt will be made to look up the key with the prefix removed. If this matches a parameter for which the type has flag fs_param_neg_with_no set, then a match will be made and result->negated will be set to true.

  If the parameter isn't matched, -ENOPARAM will be returned; if the parameter is matched, but the value is erroneous, -EINVAL will be returned; otherwise the parameter's option number will be returned.

- ```
  int fs_lookup_param(struct fs_context *fc,
                      struct fs_parameter *value,
                      bool want_bdev,
                      struct path *_path);
  ```

  This takes a parameter that carries a string or filename type and attempts to do a path lookup on it. If the parameter expects a blockdev, a check is made that the inode actually represents one.

  Returns 0 if successful and `*_path` will be set; returns a negative error code if not.
  ```