Storage Management File System

Peter Braam, Mike Pershin

14 February 2005

Contents

1 Requirements

In this document SMFS (Storage Management File System) is described. It is required to do the following:

- works as transparent pseudo-filesystem on the top of real filesystem
- have access to Lustre's API like llog and btree
- be portable to other file systems than ext3
- be able to call side-functions from other modules and plugins using own API.

All features can exist together and should not affect bottom filesystem in any way.

2 Architectural overview

In order to achieve requirements SMFS is expected to have the following major functionality divisions:

Backstore FS - main module that is placed in Lustre stack on top of backing FS. It is pseudo filesystem that just pass all operations to real one and can invoke side-functions for each filesystem operation. There is plugin API that can be used by special modules (plugins) to register list of functions for various filesystem operations.

Plugins - major reason for SMFS existence is possibility to call side-functions for any filesystem operation. Let's call module with such functionality as SMFS plugin. Each plugin is initialized via options when SMFS is mounted and corresponds to certain method tables. Plugins should register in SMFS using plugins API after that SMFS will call their functions for specified filesystem operations.

Upcalls - very similar with plugins but more generic. Modules can use this API to place own hooks in SMFS. It can be needed for notification or taking control from SMFS by upper-level.

3 High Level Design

3.1 Functional Specification

3.1.1 Backstore FS operations

SMFS creates copy of backstore FS objects when needed but setup own super/inode/file operations and save pointer to original one. SMFS should not affect the Lustre IO path. Backstore FS should remains operational in case of SMFS error as long as it possible.

3.1.2 SMFS plugins/upcalls managment

SMFS can registers/deregisters plugin. Plugins are activated after SMFS initialization and can register own functions in SMFS using special API.

All plugins are linked in the list and their functions are invoked for several filesystem operations if defined.

Deactivated process starts when SMFS uninitialization occur or ioctl is received from user. Ioctl interface or procfs (sysfs later) can be used also for plugin management.

Upcall are similar to plugins but more generic. There is upcall API in SMFS. Upcalls are also organized in linked list and called one by one where needed.

3.1.3 SMFS plugin API

Plugins should be able to register and deregister with SMFS. For these purposes SMFS has two methods

Prototype:

```
int smfs_register_plugin(struct smfs_plugin);
int smfs_deregister_plugin(int type);
```

Before calling this function plugin should fill struct smfs_plugin with own data. Registration process will check it and add to the list if possible. Deregistration takes type of plugin, search it in list and delete it.

Plugins can insert own function hooks into SMFS. Two major functions that can be setted by plugins are pre- and post-hook. They will be called before filesystem operation and after it.

Prototype:

```
/* this is prototype for plugin hook */
int smfs_hook_func (int hook_opcode, void * parameters);
```

Here we pass to plugin hook_opcode and several parameters. For each opcode paramaters can be different.

Hook_ opcode is a number from enumerated list of all possible operations where hooks are used.

3.1.4 SMFS upcall API

Plugin API is defined for using with filesystem operations specially. For cases where more generic functionality is needed upcall API can be used.

Prototype:

```
int smfs_upcall (int type, void * arg);
```

Here *type* is number from enumerated list of upcalls, *arg* is parameter or structure with parameters related to this upcall. Upcalls can be used for notification purposes. Upcalls are also widely used with plugins to get some data from they or do some action.

Due to upcall origin there can be no some modules while SMFS already is loaded. Therefore it is possible that not all functionality presents at the moment. Upcall API should notify caller about this.

3.2 Use cases

There are several plugins that use plugin API. They should fit in proposed API well.

3.2.1 KML

KML module designed to support writeback caching. KML is quite simple plugin that receive notification about FS operations and write they to the log using llog API.

KML use post_op function for logging, that are called after FS operations. It should have also pre_ops for transaction handling. The SMFS functionality is sufficient for it.

3.2.2 LRU

LRU is designed for cache management in local MDS on client, but can be used also for HSM in future. LRU module defines both pre_op and post_op. Pre_op is quit simple and is used mostly to check do we need purge cache or not. Post_op does cache operations. There is special thread for purging cache, it can be awaken from pre_op but. SMFS plugin API functionality is sufficient for this module.

3.2.3 COW

This module supports snapfs working over SMFS. This module use all functionality of SMFS plugin API - pre_op and post_op. Meanwhile it doesn't need extra functionality and fits in existent API.

3.3 Logic Specification

3.3.1 Backstore FS operations

For succefull functioning SMFS should maintain copies of many filesystem objects such as superblock, inodes, files, etc. and replace their operations with own. Therefore each filesystem object in SMFS has its origin object in backstore FS and there is pointer on it. While operating SMFS should maintain own object in consistency with real objects.

Example of superblock operation handling:

```
smfs_dirty/write_inode(struct inode * inode)
{
        /* getting saved backstore fs inode */
        backfs_inode = I2CI(inode);
        /* here can be some pre-work */
        ...
        /* call real operation */
        backfs_sb->s_op->dirty/write_inode(backfs_inode);

        /* postprocessing */
        duplicate_inode(inode, backfs_inode);
}
```

3.3.2 Fsfilt operations handlilng

Fsfilt operations are handled like ordinary FS operations. SMFS store fsfilt operations that corresponds to backstore FS, fsfilt module just use operations declared for SMFS. SMFS calls real operation with some framework. There are several common steps to pass fsfilt operations to real FS:

- get backfs sb and inode from SMFS struct inode
- get backfs fsfilt_operations from superblock
- call needed operation using backfs objects

Transactions in SMFS are handled using fsfilter operation for backstore FS. SMFS make fs_start and fs_commit in all cases where it is needed. Therefore these all extra changes will be included into current transaction:

```
smfs_trans_start()
...
SMFS_HOOK()
...
smfs_trans_commit()
```

Transaction handler are passed to plugin in hook and can be used by plugin to do proper transaction handling when it is needed.

3.3.3 SMFS plugin activation/deactivation

Now all plugins are compiled in SMFS module and are initialized via options when SMFS is mounting.

Options should be passed as mountfsoptions like this: *... [kml/cache/snap...]*

SMFS parses module options and call initialization method for selected plugins. It is possible to activate/deactivate plugins using ioctl (procfs or sysfs) later.

While initialization plugins should registers with SMFS:

```
smfs_register_plugin(parameters);
```

Parameters are struct smfs_plugin filled with valid data:

- type of plugin - to distinguish it from anothers,
- pre_op function - function that will be called before fs operation,
- post_op function - function that will be called after fs operation,
- helper function - helper function. See Plugin API,
- any private data - data which will be return to plugin in each call.

3.3.4 SMFS hook invokation

SMFS place special wrapper in own filesystem operations to call plugins hook:

```
/* wrapper that calls all hooks walking through the list */
#define SMFS_HOOK(opcode,...)    \
do {                                                          \
       smfs_hooks_op * hops;                                  \
       ...                                                    \
       list_for_each_entry(plugin_list, hops, smh_list) {     \
              ...                                             \
              rc = hops->smh_hook_op(opcode, ...);            \
       }                                                      \
        ...                                                   \
} while(0)
```

Hooks are placed in SMFS methods before and after calling backstore FS operations:

```
/* this is how to SMFS uses hooks */
smfs_some_op()
{
        struct inode * backfs_inode = I2CI(inode);
        struct smfs_file_info *sfi;

        SMFS_HOOK(..., hook_opcode , ..., PRE_HOOK, ...);
        backfs_inode->i_fop->some_op(sfi->c_file, ...);
        SMFS_HOOK(..., hook_opcode, ..., POST_HOOK, ...);
}
```

3.3.5 SMFS upcall usage

Upcalls can be placed in any place in SMFS, where it is needed.

```
/* example of upcalls handling */
int smfs_upcall(int type, void * arg)
{
        list_for_each(upcall_list)
                entry->upcall(type, arg);
}
/* example of using upcall  */
int check_uninitialization()
{
```

```
        ...
        smfs_upcall(SMFS_UP_SMTH, &parameter);
        ...
}
```

4 Plugins API

4.1 Functional Specification

Each plugin has own unique name and corresponding type. There is enumerated list of plugins types. Any new plugin should use next sequence number as type and name which not used yet by other plugins. SMFS exports two functions for plugins: *smfs_register_plugin()* and *smfs_deregister_plugin()*.

SMFS defines format of function that can be used by plugins to invoke own methods. Plugins should pass pointers on desired function while registration.

After successfull registration plugin's function will be called when selected FS operations are occur. SMFS passes to plugin inode structre, dentry, hook opcode and fsfilt transaction handle if exists.

Plugin should return zero value in case of successfull completion and error code otherwise. Plugin should decide how to handle error by itself. In case of critical error it can even stops operating or deregisters itself from SMFS. If plugin have no function for selected *hook_opcode* it has to return also *0*.

4.2 Logic Specificaion

Plugin identification There are defined types of plugins:

```
#define SMFS_PLG_KML 0x01
#define SMFS_PLG_LRU 0x02
...
```

Plugin data Main structure for communication between plugin and SMFS.

```
struct smfs_plugin {
        struct list_head plist;
        int type;
        smfs_hook_func * pre_op;
        smfs_hook_func * post_op;
        smfs_helper * helper;
        void * plugin_data;
}
```

Where

- *type* - plugins type,
- *pre_hook_func* - function to handle pre-operations,
- *post_hook_func* - function to handle post-hook operation,
- helper - call to various helper functions needed by SMFS,
- plugin_data - private plugin data.

Plugin registration SMFS provide following method for registration:

```
int smfs_register_plugin (struct smfs_plugin *);
```

SMFS will return 0 in case of successfull registration or error code otherwise.

Plugin deregistration

SMFS provide following method for deregistration:

```
void * smfs_deregister_plugin (int type);
```

SMFS will return private plugin data from struct smfs_plugin in case of successfull registration or NULL pointer otherwise.

Plugins operation Plugin's hook function must have the same type:

```
typedef int (*smfs_plg_hook) (int opcode, void * parameter, void * plg_private);
```

Parameter can contains:

- inode - SMFS inode
- dentry - SMFS dentry
- new_inode - used in rename
- new_dentry - used in rename
- opcode - hook code
- handle - fsfilt transaction if exists

Plg_private is private plugin data.
There are opcodes for now:

```
#define HOOK_CREATE       1
#define HOOK_LOOKUP       2
#define HOOK_LINK         3
#define HOOK_UNLINK       4
#define HOOK_SYMLINK      5
#define HOOK_MKDIR        6
#define HOOK_RMDIR        7
#define HOOK_MKNOD        8
#define HOOK_RENAME       9
#define HOOK_SETATTR      10
#define HOOK_WRITE        11
#define HOOK_READDIR      12
#define HOOK_MAX          12
```

Plugin may use opcode as index for array of some type. It can be table of functions for each hook type for example:

```
static  cache_hook_op cache_space_hook_ops[HOOK_MAX + 1] = {
        [HOOK_CREATE]     cache_space_hook_create,
        [HOOK_LOOKUP]     cache_space_hook_lookup,
        [HOOK_LINK]       cache_space_hook_link,
        [HOOK_UNLINK]     cache_space_hook_unlink,
        [HOOK_SYMLINK]    cache_space_hook_create,
        [HOOK_MKDIR]      cache_space_hook_mkdir,
        [HOOK_RMDIR]      cache_space_hook_rmdir,
        [HOOK_MKNOD]      cache_space_hook_create,
        [HOOK_RENAME]     cache_space_hook_rename,
        [HOOK_SETATTR]    NULL,
        [HOOK_WRITE]      NULL,
        [HOOK_READDIR]    NULL,
};
```

Helpers

Plugins can register helper function. There are several places where helpers are needed:

- smfs uninitialization - after deleting plugin from list it can be usefull to invoke plugin deactivation
- init_inode_info - several plugins can need to initialize own info on per-inode basis
- calculate size for transaction - if plugin will participate into transaction it should return extra size value.
- etc.

```
smfs_helper(int code, void * parameter, void * plg_private);
```

Where *code* is type of helper:

```
#define SMFS_HLP_EXIT   0x01
#define SMFS_HLP_INODE  0x02
#define SMFS_HLP_TSIZE  0x03
```

4.3 Use cases

SMFS returns zero value in case of successfull registration/deregistration or error code otherwise.

Plugins will return zero value in case of successfull execution hook function or error code otherwise.

New plugin design example

```
//add new type
#define SMFS_PLG_NEW 0x03
//prepare hook functions
int new_plugin_pre_function(int opcode, void * parameter)
{
        //let it be just statistic collector
        array[opcode]++;
}
int new_plugin_init ()
{
        struct smfs_plugin data;
        //prepare data
        data.type = SMFS_PLG_NEW;
        data.pre_op = &new_plugin_pre_function;
        data.post_op = NULL;
        //regisration here
        err = smfs_register_plugin(&data);
}
```

5 Upcall API

SMFS provides the ability to place a call to other modules. This can be needed when some notification needed from SMFS for example.

There are quite a few things to keep in mind:

1. The module that causes the mount of smfs, like the low level OSS and MDS server will want to register methods for upcalls.
2. Other modules that are not mounting need to be able to find smfs and register upcall methods.
3. When these registration of upcall methods have not happened, we should get reasonable error handling. Oopses and hangs are not permitted.

5.1 Functional Specification

We can use mechanism similar to plugin API but it is more generic. Upcall is invoked in particular place or when some conditions are occure. So each upcall has own type. There can be several upcalls with one type simultaneously. Therefore we can introduce set of types for various upcalls and make generic call function like *smfs_ upcall(int type, void * arg)*. Module should register upcall handler in SMFS with help of registering function. SMFS has list of upcalls and will call functions for each upcall type walking through list.

5.2 Logic Specification

There are defined types of upcalls:

```
#define SMFS_UP_SMTH 0x01
#define SMFS_UP_...  0x02
...
```

SMFS provide following method for registration/deregistration:

```
int smfs_register_upcall (struct smfs_upcall *);
int smfs_deregister_upcall (struct smfs_upcall *);
```

SMFS will return 0 in case of successfull registration or error code otherwise.

SMFS will invoke function for each upcall:

```
typedef int smfs_upcall_func (int upcall_type, void * arg);
```

Here:

- upcall_type - upcall type. It is define where upcall was invoked and under what conditions;
- arg - any desired arguments.

5.3 Use cases

SMFS returns zero value in case of successfull registration/deregistration or error code otherwise.

Modules should be aware about conditions under what upcall is called - locking, for example.