Writing a Samba VFS
Richard Sharpe
2-Oct-2011

0 Document History

2-Oct-2011	Version 1.0 Completed xx-Nov-2011

1 Introduction

This document was written to fill a void that I saw in information about the Samba Virtual File System (VFS). There seems to be no single document that those who want to write a Samba VFS module can use to obtain enough understanding to make the process run smoothly.

In the rest of this document I:

1. Provide an outline of the Samba VFS and show the interactions between the main Samba code, the VFS layer, VFS modules, and the underlying OS.
2. Discuss two different types of file systems that module writers might want to write a VFS module for.
3. Provide more detail on actually writing a Samba VFS and some of the functions and macros Samba makes available to help you.
4. Discuss some existing VFS modules, especially in the context of the two file system types outlined above.
5. Give details on the steps module writes will have to take to add their code and build their module.
6. Provide some information on adding additional VFS routines over and above those already provided.

Please note that this document currently only discusses the **Samba 3 VFS layer**!

2 The Samba VFS

The Samba VFS provides a mechanism to allow programmers to extend the functionality of Samba in useful ways. Some examples are:

- Convert NTFS ACLs to NFSv4 ACLs for storing in a file system that supports them. The GPFS VFS module does this and the same could be done for Linux when RichACL support is complete.
- Support features that a vendor has implemented in their file system that Linux file systems do not support. The OneFS VFS module from Isilon interfaces with their in-kernel distributed file system which provides more complete NTFS functionality, including four file times, etc.
- Implement features like Alternate Data Streams.
- Support user-space file systems, perhaps accessible via a shared memory interface or via a user-space library (eg, Ceph's libceph.)

A Samba VFS is a shared library (xxx.so), or module, that implements some or all of the functions that the Samba VFS interface make available and provides the desired functionality. In addition VFS modules can be stacked (if they have been written for that), and there is a default VFS (source3/modules/vfs_default.c) that provides the default Samba functionality for those functions that are not implemented higher in the stack or that earlier modules also call.

The following diagrams help illustrate some of the concepts in more detail.

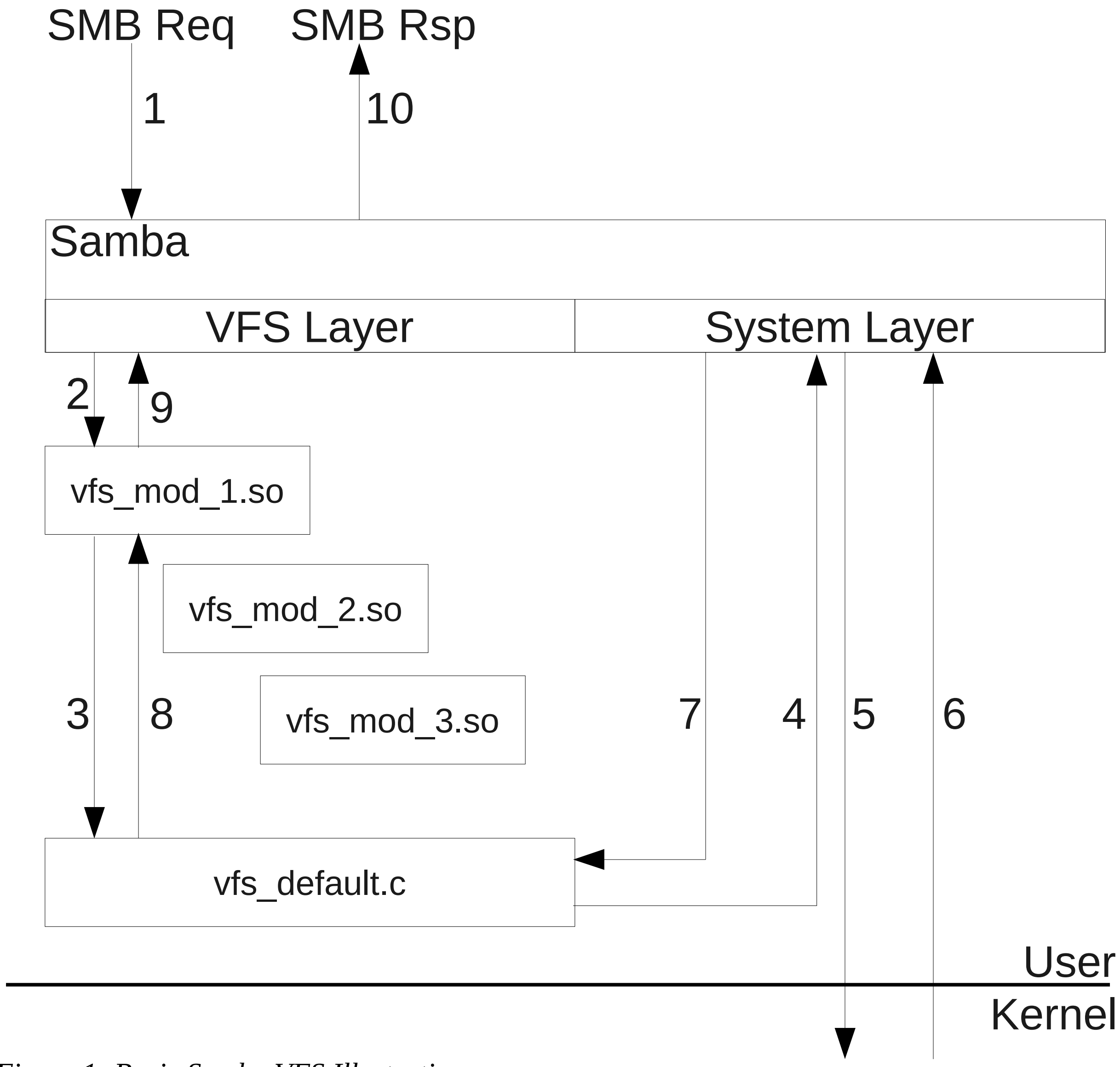

Figure 1: Basic Samba VFS Illustration

Figure 1 shows how control flows within Samba and through a VFS module. The steps are similar to the following:

1. An SMB request comes into Samba, which results in samba calling a VFS routine. The call is via a macro in the source code that looks like SMB_VFS_XXX, eg, SMB_VFS_STAT to retrieve file metadata.

2. The VFS layer calls the entry point in the first VFS module in the stack that implements the requested function.
3. If the called function needs the functionality provided by other modules in the stack, it calls VFS_NEXT_XXX, which in the illustration ends up in the default VFS module, vfs_default.c.
4. The entry points in the default VFS module typically call functions in the system layer, eg, sys_stat.
5. The system module calls into the kernel via a system call, eg, the stat system call.
6. The system call returns to the system module, which
7. Returns to the function in vfs_default.c that called the system layer, which
8. Returns up the stack to the VFS module, which
9. Returns to the main Samba code
10. Which formats and sends an SMB Response.

It should be noted that the Samba VFS interface contains some 120 different functions and that a VFS does not have to implement them all (with an exception noted below.) If a module does not implement a particular VFS function, the required function within vfs_default.c will be called.

The Samba VFS functions can be separated into the following classes:

1. Disk, or file system operations, like mounting and unmounting functions (actually called connect and disconnect), quota and free space handling routines, a statvfs function, and so forth.
2. Directory operations, like opendir, readdir, mkdir, etc.
3. File operations. This is the largest class of VFS functions, and includes functions for opening and closing files, reading and writing files, obtaining metadata information, and all the other operations you can perform on a file.
4. NT ACL operations, like setting and getting an NT ACL on a file or directory. These functions actually deal in security descriptors, which can contain ACLs.
5. POSIX ACL operations, for setting POSIX acls on files.
6. Extended Attribute operations, for setting and retrieving XATTRs on files.
7. AIO operations, for handling asynchronous operations.
8. Offline operations, for handling offline operations.

You tell Samba about any VFS modules you want used for a share in the smb.conf file. You do this with the `vfs objects` parameter for those shares you want to use VFS modules for.

For example:

```
[global]

...

[share1]
   path = /some/path
   vfs objects = acl_xattr my_vfs_obj
   ....
```

In this example we have specified that the share `share1` uses two VFS objects in the order they are listed:

1. A VFS object called acl_xattr. Any VFS functions this object implements will be called first. If they call a NEXT function, that function in the next module in the stack will be called. See below for more details on the NEXT function.
2. A VFS object called my_vfs_obj. Functions in the my_vfs_obj VFS module will be called if they are not implemented in the acl_xattr module, or if the acl_xattr module explicitly calls the NEXT function and there is one in the my_vfs_obj VFS module.

Any VFS function not implemented in any VFS module in the stack is handled in vfs_default.c.

3 Two Types of File Systems

From the point of view of Samba there are two types of file systems:

1. A file system that is accessed via system calls and for which the system provides file descriptors, and
2. A file system that is accessed from user space, typically via a user-space library.

The reason for distinguishing between these two types of file system is the following. Many Samba VFS routines deal with file descriptors (FDs). Any VFS for a user-space file system **provides file descriptors that the kernel does not understand** (it possibly supplies an index into a table of objects that are managed by the VFS.) For that reason, a VFS module for a user-space file system must implement all VFS routines and cannot forward any requests to the default VFS module, because the default VFS module will eventually result in calling a system call with a file descriptor that the kernel knows nothing about.

This also means that a VFS module for a user-space file system must be the last module in the stack.

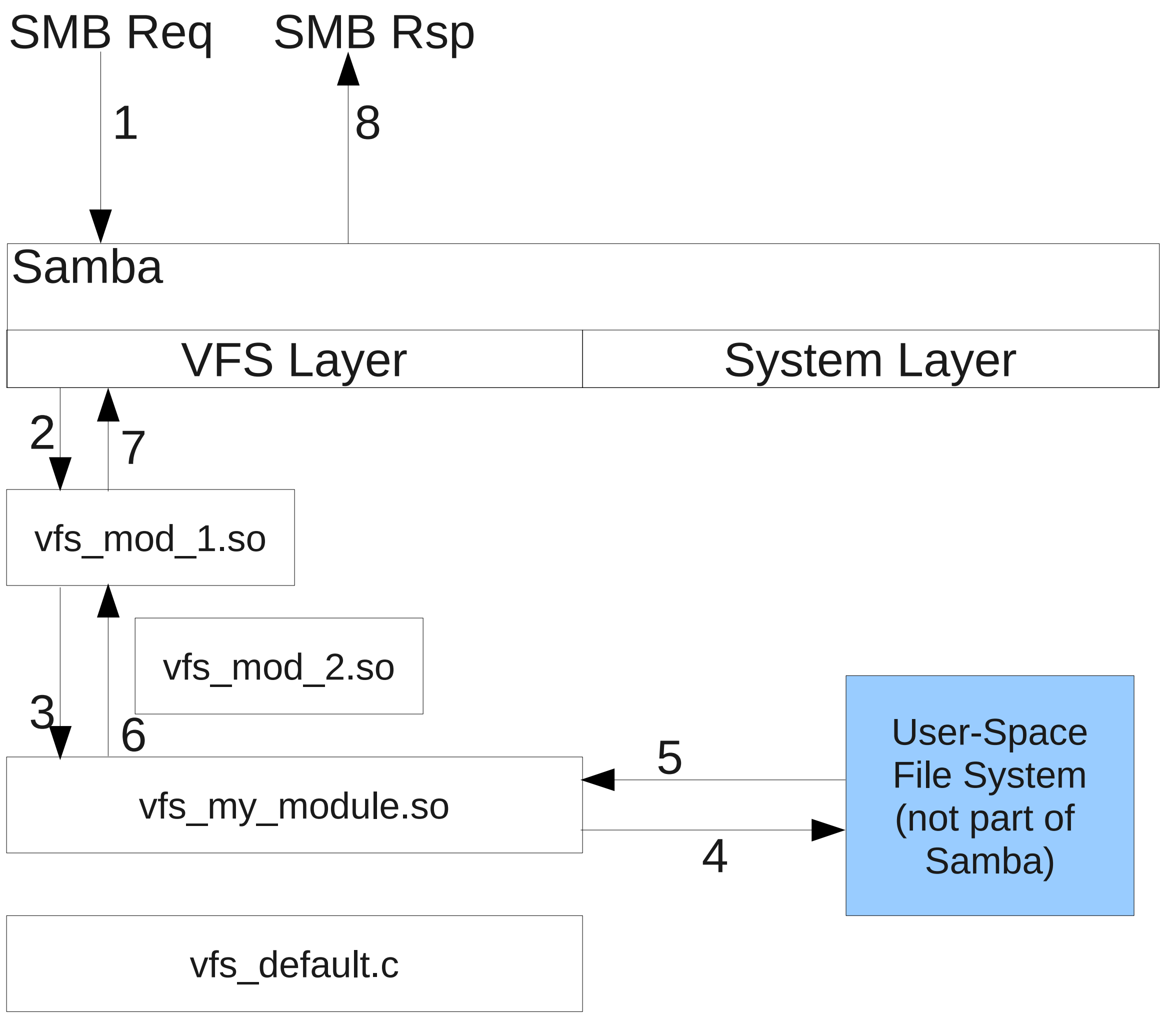

Figure 2: Accessing a File System in User Space

Figure 2 illustrates a VFS module for accessing a file system in user space. Such a file system might be accessed via NFS requests directly to an NFS server (on the same computer, or a different computer) or via a shared memory segment, etc. The essential point is that such a module must implement all VFS functions and not let any fall through to vfs_default.c.

Note: From the point of view of Samba a FUSE file system is not a user-space file system, since it is accessed by system calls just like file systems implemented in the kernel and it makes available kernel-visible FDs.

4 Writing a VFS Module

Before writing your own Samba VFS module have a look at the existing modules to see if any combination (stack) of existing modules supplies the functionality you need, or if any existing module supplies some of the functionality you need. For example, if you are thinking of storing Security

Descriptors (AKA NT ACLS) in XATTR-like objects in your file system, there is already a module for doing that called acl_xattr. As long as you provide it with a way to store XATTRs, and do a few other things, it should work and already does all the hard work for you. The source code for all the VFS modules is in source3/modules.

When you write a VFS module you supply three things:

1. A module initialization routine that tells Samba what VFS routines are handled by this module. This routine is called something like vfs_my_module_init, and its signature is specified below.
2. A VFS function pointers structure (vfs_fn_pointer) to the VFS routines implemented in this module. By using standard C89 initialization, you only initialize this structure with pointers to the functions you actually implement.
3. The actual VFS functions you implement along with any supporting functions, etc.

However, first you have to give it a name and place the code in a file. If you are building your module within the Samba source tree it will need to be placed in the directory **source3/modules**, and the main file (the one that contains your module's initialization routine as mentioned below) must be called:

vfs_<module_name>.c

For example, vfs_my_module.c. The remainder of this document will use this name in the examples. The rest of this section deals with:

1. VFS Module Initialization,
2. VFS Function Pointer Structure,
3. Include Files,
4. The VFS Functions,
5. The Life Cycle of a VFS.

4.1 VFS Module Initialization

Your module **must** contain and entry point called **vfs_my_module_init**, which the build system will actually convert to **samba_module_init**.

The initialization routine has one simple task to perform: Register itself along with the set of functions it implements. The following is an example:

```
NTSTATUS vfs_my_module_init(void)
{
	return smb_register_vfs(SMB_VFS_INTERFACE_VERSION, "my_module",
				&vfs_my_module_fns);
}
```

The things to note are:

1. As mentioned above, this function must be called vfs_<module_name>_init, it returns an NTSTATUS and does not take any parameters.
2. It returns the result of calling smb_register_vfs with three variables as shown.
3. You can name the variable that contains the functions you implement anything you want, however the practice has been to name it as shown.

4. If registration fails, none of the routines in your module will be called, but there are likely to be bigger problems, in that case.

This code can be cut from an existing module and pasted into yours with the appropriate changes made.

NOTE! If your module has undefined symbols, then Samba will not even call your module's init function, and attempts to connect to the share will fail.

The next section deals with how to declare and initialize the function pointer structure that you need to declare.

4.2 VFS Function Pointer Structure

Your module must declare and initialize a `struct vfs_fn_pointers` structure. The following is an example.

```
static struct vfs_fn_pointers vfs_my_module_fns = {
        .getxattr = my_module_getxattr,
        .fgetxattr = my_module_fgetxattr,
        .setxattr = my_module_setxattr,
        .fsetxattr = my_module_fsetxattr,
        .listxattr = my_module_listxattr,
};
```

The variable must be declared static so that it does not cause conflicts with any symbol exported by Samba or any other module. In addition, you only need to initialize pointers to just those VFS functions you are implementing (using the C89 initialization syntax.)

You would generally declare this variable before you declare the init function discussed above.

4.3 Include Files

Your module will need to invoke some include files. You will need `includes.h`, but you might also need to include a few more:

- `system/filesys.h` if you need access to many of the file system calls, like `fcntl`, etc. See `lib/replace/system/filesys.h` to determine what system include files this file pulls in.
- `smbd/smbd.h` if you need access to definitions for NT ACLs etc.

These will all need to be included before your code.

4.4 VFS Functions

These are the meat of your Samba VFS module and I can only provide generic information here.

Functions in Samba modules return several different types:

1. int return values, in which case a value less than zero means an error has occurred, and the error value is available in errno, or

2. NTSTATUS return value. Here, if the underlying functions you are calling communicate errors through errno then you have to convert them to NTSTATUS values using map_nt_error_from_unix, or
3. Pointers to things like SMB_STRUCT_DIR where you return NULL to indicate an error and set errno to a UNIX error.

If your functions are adding functionality to that already provided by Samba or existing modules in the stack (after your module) you will generally make to calls to SMB_VFS_NEXT_XXX, where XXX is the name of the function you are providing (eg, UNLINK if you are providing UNLINK functionality, in which case you will call SMB_VFS_NEXT_UNLINK.

You can also call any other VFS function that is relevant, eg SMB_VFS_STAT, but you will have to ensure that you pass the correct parameters, eg:

```
ret = SMB_VFS_STAT(handle->conn, smb_fname_cpath);
```

This brings us to the parameters that your functions will have to deal with. The first parameter passed to each Samba VFS function is a pointer to vfs_handle_struct, which contains information you might need, like the connection structure (share, etc) that the request relates to, and so forth. Another parameter you might receive is a pointer to a files_struct or a struct filename_struct. Others that you might also receive include character strings for paths, integer values, etc. You should peruse existing Samba VFS functions to see some of the values you might receive.

In addition, you should be aware that Samba has an extended STAT structure, SMB_STRUCT_STAT. In some versions of Samba (3.6.0 and above, I think) you can use init_stat_ex_from_stat to convert a normal Unix struct stat variable into an SMB_STRUCT_STAT for return to Samba. However, if the underlying module you are extending has its own extended stat structure that is not compatible with SMB_STRUCT_STAT you will have to supply a routine to convert your stat struct to an SMB_STRUCT_STAT (see, for example, `modules/onefs_streams.c::onefs_fstat` for an example.)

4.5 The Life Cycle of a VFS

When a client issues a TREE_CONNECT request (either because of a NET USE command or mapping a network drive) samba calls SMB_VFS_CONNECT which results in the connect_fn in your VFS module (if defined) being called.

The connect_fn has the following signature (the name of the function can be anything you like):

```
static int my_module_connect(vfs_handle_struct *handle,
                             const char *service,
                             const char *user)
```

This call gives you the opportunity to create and save context information for calls to other functions.

As mentioned above, the first parameter to all VFS functions is the vfs_handle_struct, which is unique for each share and module, so you can store context information in the structure pointed to by the handle. You can save information in the 'handle' in the following way:

```
config = talloc_zero(handle->conn, struct my_module_config_data);
if (!config) {
      SMB_VFS_NEXT_DISCONNECT(handle);
      DEBUG(0, ("talloc_zero() failed\n")); return -1;
}
SMB_VFS_HANDLE_SET_DATA(handle, my_module_context_data,
                              NULL, struct my_module_config_data,
                              return -1);
```

SMB_VFS_HANDLE_SET_DATA is a macro, and its arguments are:

1. handle, the VFS handle.
2. A pointer to some data that you want to associate with the handle.
3. A pointer to a function to free the data you are saving. It is set to NULL above, which means that this VFS module will explicitly free the data (in a disconnect function.)
4. The data type of the structure that param 2 points to.
5. A command to be executed if handle is NULL.

You can use this handle data to keep track of information relating to the file system backing the share, or to maintain parameters related to this instance of the share, or both. It is a pointer to a structure you declare.

You can retrieve handle data in your VFS functions subsequently using the following macro:

```
SMB_VFS_HANDLE_GET_DATA(handle, config,
                      struct my_module_config_data, return next);
```

You should also be aware of the macros SMB_VFS_HANDLE_FREE_DATA and SMB_VFS_HANDLE_TEST_DATA. Check the include file source3/include/vfs.h.

You might also want to retrieve module-specific parameters from the smb.conf file in your connect function. This can be done using:

```
config->some_bool_param = lp_parm_bool(SNUM(handle->conn),
                              "my_module", "someboolparam", true);
```

These parameters should be entered in the smb.conf file in the format:

```
[global]
      ...
```

```
my_module:someboolparam = yes
...
```

They can also appear in share sections.

If your module is not designed to be the last in the stack then your connect_fn should give other modules a chance to capture connection information as well, using:

```
int ret = SMB_VFS_NEXT_CONNECT(handle, service, user);
```

of course, you should check the return code and cleanup if an error occurs in a lower module.

When the client disconnects from the share that your VFS module is connected to, Samba will call your disconnect function:

```
static void my_module_disconnect(vfs_handle_struct *handle)
{
        /* Perform whatever actions are needed here */
}
```

In general you do not need to clean up memory allocated with talloc in your connection module if that memory was allocated using the connection structure (handle->conn) as a context, as it will all be cleaned up when the connection structure is freed with TALLOC_FREE.

Of course, if your module has no need to capture connection and disconnection events, you do not need to define these routines.

In addition to the above functions, you can extend Samba's files_struct with an extension of your own. Each module in the stack can add their own extension, but only one extension can be added per file per module. You add the extension with:

```
p_var = (struct my_struct *) VFS_ADD_FSP_EXTENSION(handle,
                        fsp,
                        struct my_struct,
                        NULL);
```

after which you can update the fields in the structure that you now have a pointer to.

You can fetch an extension with:

```
p_var = (struct my_struct *)VFS_FETCH_FSP_EXTENSION(handle,
                        fsp);
```

There is also VFS_REMOVE_FSP_EXTENSION and VFS_MEMCTX_FSP_EXTENSION, which can be found in source3/include/vfs.h, although they reduce to functions in source3/smbd/vfs.c.

4.6 AIO Handling in a VFS Module

Samba supports the use of AIO and provides three VFS functions to allow VFS module writers to also support AIO. These are:

SMB_VFS_AIO_READ	This is used to initiate an AIO read request. If all went well initiating the request, return 0, otherwise return -1 after setting errno to something appropriate.
SMB_VFS_AIO_WRITE	This is used to initiate an AIO write request. If all went well initiating the request, return 0, otherwise return -1 after setting errno to something appropriate.
SMB_VFS_AIO_RETURN	This is used to retrieve the returned status from a successfully initiated AIO operation. That is, whether it ultimately succeeded or failed.
SMB_VFS_AIO_CANCEL	This is used to cancel an already initiated AIO operation. If you managed to do so, return AIO_CANCELED or AIO_NOTCANCELED, AIO_ALLDONE or -1 as appropriate and set errno where appropriate.
SMB_VFS_AIO_ERROR	This is used to retrieve the status of AIO operations that were successfully initiated. Return EINPROGRESS, ECANCELED or an other error to indicate an error, or 0 to indicate that the operation has successfully completed.
SMB_VFS_AIO_FSYNC	Samba 3 does not currently use this VFS routine.
SMB_VFS_AIO_SUSPEND	This is used to clean up initiated AIO operations when a client drops a connection. Consult the Samba code for more details, specifically source3/smbd/aio.c.
SMB_VFS_AIO_FORCE	This is used to tell Samba whether or not the read or write operation Samba is about to initiate via AIO should be performed via AIO. That is, your module gets to veto the initiation of AIO requests on a request by request basis if it wants to. Return FALSE if you are happy to allow the operation to be an AIO operation, otherwise return TRUE if you don't want that operation being sent via AIO.

The default behavior is to call the standard system AIO routines, aio_read/aio_read64, aio_write/aio_write64 and aio_return/aio_return64.

The main thing to be aware of here is that if you support AIO in your VFS module, and you do not simply pass them on to normal kernal AIO routines (either via sys_aio_xxx routines or directly via system calls) then you must simulate the normal AIO completion behavior. That is, you must signal RT_SIGNAL_AIO somewhere in your module (perhaps in the async threads) when the operations ultimately complete.

5 Some Existing VFS Modules

You should peruse the existing Samba VFS modules to get some idea of how others have written VFS modules and also to find those modules that already implement functionality that you want, so you don't have to re-implement it. The judicious use of module stacking can save you a lot of coding and testing. You can find them all in source3/modules.

One module that is particularly interesting in this regard is the acl_xattr (and acl_tdb) module. This module stores security descriptors (AKA NT ACLs) as blobs in XATTRs attached to files (security.NTACL), however, it does so in an interesting way. The process it uses is to retrieve any underlying ACL on the file by calling the next module in the stack. This underlying ACL might be synthesized from UNIX permissions, or converted from RichACLs, etc. It then hashes the underlying security descriptor, and when storing a new NT ACL, it includes the hash of the underlying file system security descriptor in the blob it stores. This way, when the NT ACL is subsequently retrieved by Windows, the acl_xattr module can check to see if anyone changed the underlying security descriptor (permissions) and return those instead. This must be an attempt to provide some level of interoperability with NFS.

6 Building, Installing and Debugging your VFS Module

6.1 Building your VFS Module

If you are adding your own VFS module, you can add it in the Samba source tree for building, or you can build it outside of the Samba source tree. Here I will give suggestions for what you need to change to get your VFS module to build in the Samba source tree.

There are two different cases, though:

1. You are building from a git source tree. That is, you did a git-clone and have checked out a particular branch.
2. You are building from a released source tarball, eg samba-3.5.10.tar.gz.

Each of these will be dealt with in turn, however, they involve modifications to configure or configure.in and Makefile.in, depending on which route you take.

Building in a git Tree

If you are working within a clone of the Samba git repos (and you have created your own branch off of an existing branch, haven't you – git checkout -b my-branch [<start-point>]) then you need to modify configure.in to include your module in the default modules list if you want your module built by default, and then you need to provide instructions in Makefile.in telling the Samba build system (make for the moment) how to build your module.

Each of these are pretty much boilerplate changes.

Change source3/configure.in by searching for the symbol default_shared_modules and adding your module to the end. You will find it looks something like this:

```
dnl These are preferably build shared, and static if dlopen() is not available

default_shared_modules="vfs_recycle vfs_audit vfs_extd_audit vfs_full_audit
vfs_netatalk vfs_fake_perms vfs_default_quota vfs_readonly vfs_cap vfs_expand_msdfs
vfs_shadow_copy vfs_shadow_copy2 charset_CP850 charset_CP437 auth_script
```

```
vfs_readahead vfs_xattr_tdb vfs_streams_xattr vfs_streams_depot vfs_acl_xattr
vfs_acl_tdb vfs_smb_traffic_analyzer vfs_preopen vfs_catia vfs_scannedonly"
```

Simply add `vfs_my_module` to the end of the list.

Next, you have to add several short sections to Makefile.in.

Search for the last module listed above in Makefile.in, the one before your module. You should find it looking like this:

```
PERFCOUNT_ONEFS_OBJ = modules/perfcount_onefs.o

PERFCOUNT_TEST_OBJ = modules/perfcount_test.o

VFS_DIRSORT_OBJ = modules/vfs_dirsort.o

VFS_SCANNEDONLY_OBJ = modules/vfs_scannedonly.o


PLAINTEXT_AUTH_OBJ = auth/pampass.o auth/pass_check.o
```

and add a line that says `VFS_MY_MODULE_OBJ = modules/vfs_my_module.o` after the last module and before the empty line that signals the end of the list of modules that are known about. It might look a little different than shown depending on whether or not there are modules that are not built by default.

Then, search further into Makefile.in for the last module again, scannedonly in this case. You will find something like this:

```
bin/dirsort.@SHLIBEXT@: $(BINARY_PREREQS) $(VFS_DIRSORT_OBJ)
      @echo "Building plugin $@"
      @$(SHLD_MODULE) $(VFS_DIRSORT_OBJ)

bin/scannedonly.@SHLIBEXT@: $(BINARY_PREREQS) $(VFS_SCANNEDONLY_OBJ)
      @echo "Building plugin $@"
      @$(SHLD_MODULE) $(VFS_SCANNEDONLY_OBJ)
```

Add a similar section for your module, which should look something like this:

```
bin/scannedonly.@SHLIBEXT@: $(BINARY_PREREQS) $(VFS_SCANNEDONLY_OBJ)
      @echo "Building plugin $@"
      @$(SHLD_MODULE) $(VFS_SCANNEDONLY_OBJ)
bin/my_module.@SHLIBEXT@: $(BINARY_PREREQS) $(VFS_MY_MODULE_OBJ)
      @echo "Building plugin $@"
      @$(SHLD_MODULE) $(VFS_MY_MODULE_OBJ)
```

After that, simply rerun autogen.sh (to regenerate configure from configure.in) and then rerun configure and then run make.

Note. If you do not want to modify configure.in to have your module built by default, simply make the changes specified above to Makefile.in and have your build system build your module by using `make bin/my_module.so` (or whatever the shared library suffix is on your system.)

Building in a Source Tarball Tree

If you are building from a source tarball, consider building your module outside the Samba source tree. Instructions on doing this will be provided some time :-)

However, here is how to hack the configure script and Makefile.in to achieve what you need.

Follow the instructions for building in a git source tree, but edit configure rather than configure.in, because the symbols you are searching for are the same. Then modify Makefile.in as described above.

Finally, rerun configure, and then run make.

6.2 Installing your VFS Module

Your VFS module will be installed in (or should be copied to):

1. /usr/lib64/samba/vfs if you build for an RPM-based Linux system,
2. /usr/local/samba/lib/vfs if you build for an FHS-based Linux system.

It might also be installed elsewhere depending on your environment.\

6.3 Debugging your VFS Module

It is relatively easy to debug your VFS module, and here are some steps that might prove useful:

1. The first thing to check is that your module is attaching and detaching correctly. You can do this with smbclient, eg:

   ```
   smbclient //localhost/some-share -Usomeuser%somepass
   ```

   After then, check that you get a normal smbclient prompt, exit, and then check the log files for errors or crashes in your VFS module. You might want 'debug level = 10' in the smb.conf file for this. If you get a core file, use gdb to inspect the core file.
2. Once that piece works, use smbclient commands to check that the basics work, eg, that you can list directories with the ls command, or that you can copy files.
3. Finally, from a Windows client, connect to the same share as above and perform the appropriate tests.

Some versions of Samba on Linux would not generate core files (because of a bug to do with Linux not allowing core files by default after programs have used setreuid et al), although that has been fixed in recent versions of Samba. In any event, if you find that you are not getting core files, you can use this alternative. Specify a 'panic action = sleep 999999' in the smb.conf file. This causes Samba to pause for a long time in its SIGSEGV handler, which will allow you time to find the errant process and attach with gdb so you can get stack traces etc.