YaNFS: Yet Another Network File System

Zhao Ruizhe
School of EECS, Peking University

Abstract

As the cloud services become more and more famous recently, and personal user level data clouds become cheaper, developing practical techniques which could build a "virtual environment" for those applications is quite necessary. With the help of Secure File Transfer Protocol(SFTP), and the easy to understand and augment Linux Virtual File System(VFS) API, it's possible to conquer that task. In this report, a brief introduction to the file system, which has been named as YaNFS will be listed first, and the implementation details and the *communication protocal* between the *kernel mode* and the *user mode* will be illustrated next. Although this project is finished, there's still a chance to improve the performance. All the optimization techniques will be addressed in the end.

Chapter 1

Introduction

1.1 File System Architecture

1.1.1 Module and File System

Obviously, implementing a file system on Linux is just writing a *module*. Why is that? The idea is, you'll never know there's such a file system unless you read some files in it, write data to it, or listing the directories under the mounting point of that file system. All these functionalities are provided by the standard *virtual file system(VFS)* APIs, and the way we define a unique file system is by referring the fuction pointers to our self-defined functions. For example, if you want to count the number of executed read operations, you could define a read function:

```c
// definition
static ssize_t read(struct file* flip, char __user *buf,
                        size_t len, loff_t *ppos)
{
        ...
        printk(KERN_INFO "Trying␣to␣read\n");
        ...
}
```

And refer to it:

```c
// refering
struct file_operations file_operations = {
        .read = read,
        ...
};

file->f_op = file_operations;
```

How could Linux VFS know there's such a `read` function? *By defining it in the module.* So that's why I say: "implementing a file system is just writing a module."

1.1.2 Mounting

After we've defined the function pointers' reference, we need to put our file system under one directory and register the file system. *Mounting* is the procedure of making the files or directories under the *mounting point* directory use the VFS APIs provided by our file system, and registering is just a simple procedure to let the operating system know that there's such a file system.

Where to do the mounting and registering procedure? In the `init` call of our module:

```
static int __init init_fs(void )
{
    int err;

// Here the register_filesystem() function will call
// our mounting function().
    err = register_filesystem(&fs_type);

    return err;
}
```

It's worthy to denote that there're 2 types of mounting: `mount_bdev` and `mount_nodev`. If you use the previous one, then all the `read` and `write` functions could operate on a block device, while the later one couldn't.

1.2 Kernel-User Communication

Linux VFS functions should be run under the *kernel mode*, which has the highest previledge, while at the same time, has less utilities. Several useful networking libraries are all under *user mode*. Due to this need, we have to build a communication mechanism between the *kernel mode* and the *user mode*.

Socket could do the job, but we want it to be simpler. Then we have *Netlink* module. *Netlink* is a special family of socket protocols, which is designed to provide communication between *kernel mode* and *user mode*.

1.3 SSH and SFTP

Secure Shell(SSH) is one of the most useful tool for programmers: It could create a remote shell for you to operate on a remote machine, at the same time, the communication is quite secure. Secure File Transfer Protocol(SFTP) is a secure version of FTP, which you could do file operations through the SSH session. SSH session will be created once you've logged on your remote machine.

LibSSH could help you do a lot of tasks based on SSH and SFTP, and our application is built on that library. It's mechanism is quite simple: After you've created a SSH session by doing the authetication and specifying the host and port, you could call any kind of remote operations afterwards.

Chapter 2

Implementation

2.1 Design Philosophy

The goal of our YaNFS has 3 points, which have been listed previously:

Virtual Environment The host machine couldn't know any other files on the remote machine, which means you need to build your own directory tree on the host("directory tree" is not a rigorous definition, it's just referring the inodes and dentries).

Kernel-User Mode The file system is built in kernel mode while the remote connection is built in user mode.

SSH Connection The remote connection is established through SSH and SFTP protocol.

And the whole project will be built on 3 different layers: *file system*, *communication* and *SSH connection*. It's rather good to understand this through an example.
Here I'd like to read a file on the remote machine:

1. Call `read()` in kernel mode.
2. Translate from dentry hierachy links to file pathname string.
3. Build message string "read [filepath]", and stall the kernel mode `read()`.
4. SSH daemon come to ask if there's any command to execute, and the message string found.
5. According to the message string, call `sftp_read(filepath, buf, size)` to store the read data to `buf`.
6. Send back read data to the kernel.
7. Kernel mode `read()` found the message, and copy to the `buf` parameter provided by the function call by `copy_to_user()`.

2.2 File System Implementation

2.2.1 File System Type

It's required to provide a `file_system_type` parameter if you want to register your file system.

```
static struct file_system_type sshfs_fs_type = {
    .name     = "kernelsshfs",
    .mount    = sshfs_mount,
    .kill_sb  = sshfs_kill_sb,
    .fs_flags = FS_USERNS_MOUNT, // what for?
};

struct dentry *sshfs_mount(struct file_system_type *fs_type,
                           int flags,
                           const char *dev_name,
                           void *data)
{
    return mount_nodev(fs_type, flags, data, sshfs_fill_super);
}

static void sshfs_kill_sb(struct super_block *sb)
{
    kill_litter_super(sb);
}
```

2.2.2 Super Block

Look at the `sshfs_mount` function, which will do `mount_nodev`. `mount_nodev` denotes our file system does not need a backend block device. And `mount_nodev` need a function `sshfs_fill_super` to fulfill the properties of the super block of our file system.

It's the definition of the *super block operations*. The functions are provided by default Linux VFS super block operations.

```
static const struct super_operations sshfs_ops = {
    .statfs       = simple_statfs,
    .drop_inode   = generic_delete_inode,
    .show_options = generic_show_options,
};
```

Here the `sshfs_get_inode` will not be covered in this document, you could easily find it in the `sshfs_inode.c` file. So the basic operations `sshfs_fill_super` does are just setting:

1. `s_op`
2. `s_root`
3. `s_magic`

```c
int sshfs_fill_super(struct super_block *sb, void *data,
        int silent)
{
    sb->s_blocksize      = PAGE_CACHE_SIZE;
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic          = SSHFS_MAGIC;
    sb->s_op             = &sshfs_ops;

    inode = sshfs_get_inode(sb, NULL,
                S_IFDIR|SSHFS_DEFAULT_MODE, 0);

    sb->s_root = d_make_root(inode);
        ...
}
```

2.2.3 File and Inode Operations

In order to satisfy our "virtual environment" requirement, we need to:

1. build `inode` both on host and remote.
2. only do file operations on remote.
3. querying the host-only inodes.

And the code definition of inode operations, and only the first 4 operations are self-defined. All the other operations are default operations which means they will not visit the remote information.

inode operations

```c
static const struct inode_operations
        sshfs_dir_inode_operations = {
    .mknod  = sshfs_mknod,
    .mkdir  = sshfs_mkdir,
    .rmdir  = sshfs_rmdir,
    .create = sshfs_create,
    .lookup = simple_lookup,
    .link   = simple_link,
    .unlink = simple_unlink,
    .rename = simple_rename,
};
```

File operations are much more simplified, only `read`, `write` and `llseek` are defined.

file operations

```c
const struct file_operations
        sshfs_file_operations = {
    .read    = sshfs_file_read,
    .write   = sshfs_file_write,
    .llseek  = sshfs_llseek,
    .fsync   = noop_fsync,
};
```

All the self-defined operations are based on one function, `netlink_send_command`, to send the command message like "read dir1/file1" to the netlink module. I'll cover this in the next section.

2.3 Netlink Communication

Only 2 APIs are important in this section.

2.3.1 Send Command and Stall

```
static int netlink_send_command(cmdtype cmd,
                                char *params,
                                struct iovec *iov)
{
        ...
    cmd_msg_buffer.cmd    =
                kzalloc(sizeof(char) * cmdstr_len, GFP_KERNEL);
    cmd_msg_buffer.cmdlen = cmdstr_len;
    cmd_msg_buffer.exec   = false;
    cmd_msg_buffer.msglen = 0;
    cmd_msg_buffer.msg    = NULL;

    strcpy(cmd_msg_buffer.cmd, cmdstr);

    cmd_msg_buffer.ready  = true;
    while(!cmd_msg_buffer.exec)
        msleep(1000);
        ...
    iov->iov_len = cmd_msg_buffer.msglen+1;
    iov->iov_base =
         (char *) kzalloc((iov->iov_len), GFP_KERNEL);
    strcpy(iov->iov_base, cmd_msg_buffer.msg);

        ...
    kzfree(cmd_msg_buffer.msg);
    cmd_msg_buffer.msg = NULL;

    return 0;
}
```

In this function, we'll first fulfill the `cmd` buffer in the global parameter: `cmd_msg_buffer`, and set the `ready` bool to `true`. Then this procedure will stall until the `exec` is `true`.

After the stall is finished, this function will copy back the message from the `msg` buffer. How to use that message is specified by different methods. Take the `sshfs_file_read` as an instance:

file read

```
static ssize_t sshfs_file_read(struct file* flip,
        char __user *buf, size_t len, loff_t *ppos)
{
        ...
    netlink_send_command(Cmd_read, params, iov);

    if(copy_to_user(buf, (void*)iov->iov_base, iov->iov_len))
        return -EFAULT;

    *ppos += iov->iov_len;
    ret = (ssize_t) iov->iov_len - 1;

    kzfree(params);
    kzfree(iov);
    return ret;
}
```

This one just copy the received message to the buffer required by the `read()` VFS API.

2.3.2 Message Receiving

It's necessary to issue that there're 3 steps in this module:

1. First of all, this method will be called only if there's a message sent from SSH daemon. The message is contained in `struct sk_buff *skb`.

2. This method will parse the message from SSH daemon next, there're 3 kind of status:

Status	Meaning
READY	SSH daemon is working and has nothing to do.
message	SSH daemon is transferring message and the content of received message have not yet finished.
DONE	SSH daemon has finished transferring.

3. At last, send the response to SSH Daemon by `nlmsg_unicase`.

Status	Meaning
NOTHING	If SSH daemon is ready and the file system has no workload
COPIED	If SSH daemon is transferring message and the data is copied.
OK	SSH daemon has finished transferring and response to it.

The codes are listed below:

Message Receiving

```c
static void netlink_recv_msg(struct sk_buff *skb)
{
    struct nlmsghdr *nlh;
    int pid;
        ...

    nlh = (struct nlmsghdr*) skb->data;
    pid = nlh->nlmsg_pid;
    recvmsg = (char *) nlmsg_data(nlh);

    if (!strcmp(recvmsg, recv_ready_str))
        ... // When SSH Daemon is Ready
    else if (!strcmp(recvmsg, recv_done_str))
        ... // When SSH Daemon is Done
    else
        ... // When SSH Daemon is transferring data.
    skb_out = nlmsg_new(msg_size, 0);
    nlh = nlmsg_put(skb_out, 0, 0, NLMSG_DONE, msg_size, 0);
    NETLINK_CB(skb_out).dst_group = 0;
    strncpy(nlmsg_data(nlh), msg, msg_size);

    res = nlmsg_unicast(nl_sk, skb_out, pid);
}
```

2.4 SSH Daemon

The SSH daemon is just like a bridge, which lies between the kernel mode `netlink` module, and the user mode `libssh` library.

2.4.1 Communicate with Netlink

Kernel mode netlink module could only response once SSH daemon send message to it. And SSH send message function follows these steps:

1. Initialize **`struct`** `nlmsghdr` by assigning `nlmsg_len`, `nlmsg_pid` and `nlmsg_flags`.
2. Initialize message itself. Send the message and stall.
3. Once received the message from the kernel, do the parsing work to get what to do next.

Communicate with Netlink

```c
char * send_recv_netlink_portal(char *msg_str, int msg_len)
{
    nlh = (struct nlmsghdr *) malloc(NLMSG_SPACE(MAX_PAYLOAD));
    memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
    nlh->nlmsg_len   = NLMSG_SPACE(MAX_PAYLOAD);
    nlh->nlmsg_pid   = getpid();
    nlh->nlmsg_flags = 0;

    strncpy(NLMSG_DATA(nlh), msg_str, msg_len);

    iov.iov_base    = (void *)nlh;
    iov.iov_len     = nlh->nlmsg_len;

    msg.msg_name    = (void *)&dest_addr;
    msg.msg_namelen = sizeof(dest_addr);
    msg.msg_iov     = &iov;
    msg.msg_iovlen  = 1;

    sendmsg(sock_fd, &msg, 0);
        \\ will stall here
    recvmsg(sock_fd, &msg, 0);

    return (char *)NLMSG_DATA(nlh);
}
```

There's a infinite loop in the `main()` function of SSH daemon:

Main loop

```c
while (1)
{
    char *recv =
        send_recv_netlink_portal("READY", strlen("READY"));
    if (!strcmp(recv, "ERROR"))
        ...
    else if (!strcmp(recv, "NOTHING"))
        sleep(1);
    else
    {
        struct iovec *iov =
            sftp_execute(ssh_s, sftp_s, recv);
        ...
        recv =
            send_recv_netlink_portal("DONE", strlen("DONE"));
        if (strcmp(recv, "OK"))
            exit(1);
        sleep(1);
    }
}
```

It's clear that this daemon will first send `READY`, and do different tasks by specifying the different responses. If SSH daemon is sure that it'll do some work on the remote machine, it'll call `sftp_execute`.

2.4.2 Communicate with SSH Library

There's no need to show some details about the libSSH library, so here I'll just take one function for instance: `sftp_execute_mkdir`. This function will be called only if the command sent by netlink module is something like: `mkdir dir1`.

Remote Mkdir

```
int sftp_execute_mkdir(ssh_session ssh,
                       sftp_session sftp,
                       char *dirname)
{
    int rc;

    rc = sftp_mkdir(sftp, dirname, S_IRWXU);
    if (rc != SSH_OK)
    {
        if (sftp_get_error(sftp) != SSH_FX_FILE_ALREADY_EXISTS)
        {
            fprintf(stderr, "Can't␣create␣directory:␣%s\n",
                    ssh_get_error(ssh));
            return rc;
        }
    }

    return rc;
}
```

As you can see, the key function is `sftp_mkdir`.
Another example is `sftp_execute_read`:

Remote Read

```
int sftp_execute_read(ssh_session ssh,
                      sftp_session sftp,
                      char *params, char **msg)
{   ....
    file = sftp_open(sftp, filename, access_type, 0);
    sftp_seek(file, offset);
    for (;;)
    {
        nbytes = sftp_read(file, buffer, sizeof(buffer));
        ....
        if (nbytes == 0) break;
        else if (nbytes < 0) return SSH_ERROR;

        int buflen = strlen(buffer);
        int msglen = (*msg != NULL) ? strlen(*msg) : 0;
        msglen += buflen;
        msglen ++;
        ....
    }
    rc = sftp_close(file);
    return rc;
}
```

The data received from remote operation `sftp_read` will be inserted into local buffer, and send back to netlink once the transfer is finished.

After the netlink received the message, it will copy the content to the `read()` API's buffer.