[Articles Home]

Note: This file has been modified. The code works for 2.6 kernels. You can get the old code/article here.

Writing a Simple File System

Author: Ravi Kiran UVS

1. Objective

We will write a file system with very basic functionality. This is to understand the working of certain kernel code paths of the Virtual File System (VFS). This filesystem has only one file 'hello.txt'. We can read/write into it.

2. Introduction

The Linux VFS supports multiple file systems. The kernel does most of the work while the file system specific tasks are delegated to the individual file systems through the handlers. Instead of calling the functions directly the kernel uses various Operation Tables, which are a collection of handlers for each operation (these are actually structures of function pointers for each handlers/callbacks). The kernel calls the handler present in the table for the operation. This enables different file systems to register different handlers. This also enables the common tasks to be done before calling the handlers. This reduces the burden on the handlers which can then focus on the operation specific to that file system.

File systems are identified by their names. The supported file systems can be seen using 'cat /proc/filesystems'. The first step is to register the file system with the kernel. Since we are using a kernel module, the file system registration is done during the module initialization. This registers handlers which will be called to fill the super block structure while mounting, a handler to do the cleanup during unmounting the file system. There are other handlers but these two are essential.

The super block operations are set at the time of mounting. The operation tables for inodes and files are set when the inode is opened. The first step before opening an inode is lookup. The inode of a file is looked up by calling the lookup handler of the parent inode. But what about the root-most inode of the new file system? This has to be allocated at the time of mounting i.e., during the super block initialization.

Once the operation tables are set on the data structures, the kernel calls the handlers depending on the operation.

3. Data Structures

This is a brief description about the data structures used in implementing our file system.

a. File System Type (struct file_system_type)
Definition found in *include/linux/fs.h*

This structure is used to register the filesystem with the kernel. This data structure is used by the kernel at the time of mounting a file system. We have to fill the 'name' field with the name of our file system (example "rkfs") and the handlers get_sb and kill_sb to allocate and release the super block objects.

b. Super Block (struct super_block)
Definition found in *include/linux/fs.h*

This stores the information about the mounted file system. The important fields to be filled are the operation table (s_ops field) and the root dentry (s_root). At the time of mounting a file system, the kernel calls the get_sb field of the file_system_type object (it identifies the correct file_system_type object based on the file system name) to get a super block object.

c. Super Block Operations (struct super_operations)
Definition found in *include/linux/fs.h*

Super block operations table.

d. Inode (struct inode)
Definition found in *include/linux/fs.h*

Inode object is the kernel representation of the low level file. We return the dentry of the root of our file system. We have to attach a proper inode also to the dentry.

This structure has two operation tables i_op, i_fop i.e., inode operations and file operations respectively. We will implement one operation in the inode_operations - lookup.

This is called when the kernel is resolving a path. The kernel starts from the ancestor (this can be the current working directory for relative paths or the root most directory for the absolute paths) and gets the dentry (also the inode) of a name component of the path from its parent. This is achieved by calling inode_operations.lookup on the inode of the parent entry.

For example, when the kernel is resolving /parentdir/subdir, the lookup operation reaches the root most inode of the file system. This was already allocated during the super block initialization and stored in the s_root field. To resolve the 'parentdir' under the root most inode, the kernel creates a new dentry object, sets the name as 'parentdir' and calls lookup handler on inode of the root most inode. The handler is supposed to attach the inode to the dentry using d_add and return NULL if it was successful or an error code otherwise. Similarly, the lookup for 'subdir' is done by the parentdir inode. The dentry cache and the inode cache saves repeated lookups and boosts the performance.

It is important for us to implement the lookup callback. This will be called by the open system call.

e. Inode Operations (struct inode_operations)
Definition found in *include/linux/fs.h*

This is the inode operations table with each field corresponding to a function pointer to handle the task. It has fields like mkdir, lookup etc. We are interested in lookup.

f. Address Space Operations (struct address_space_operations)
Definition found in *include/linux/fs.h*

Address space operations table.

g. DEntry (struct dentry)
Definition found in *include/linux/dcache.h*

The kernel uses dentries to represent the file system structure. dentries point to inode objects. This has pointers to store the parent-child relationship of the files. Inodes and files do not store any information about the hierarchy.

h. File (struct file)
Definition found in *include/linux/fs.h*

File object is used to store the process's information about the file. We dont have to fill any fields of the files directly. The kernel takes care of filling the proper fields but we have to implement the file operation callbacks. We register the file operation table when we return the inode object during lookup. The file operations are copied from the i_fop field of the inode object to the file object by the kernel.
We will implement readdir in case of directories (while returning the inode, we have to set the file operation table based on the type of the file) and read/write in the case of regular files. We will have two file operation tables one for directories and the other for regular files.

The relationship between files, dentries and objects is like this:

File **--->** DEntry **--->** Inode

The following figure shows the important relationships between various VFS data structures. It does not show all the relationships though. Note that super block structure has a list of all open file objects, a list of dirty inodes and another list of locked inodes of the file system. Also the lists used for cache, lru and free lists are not shown.

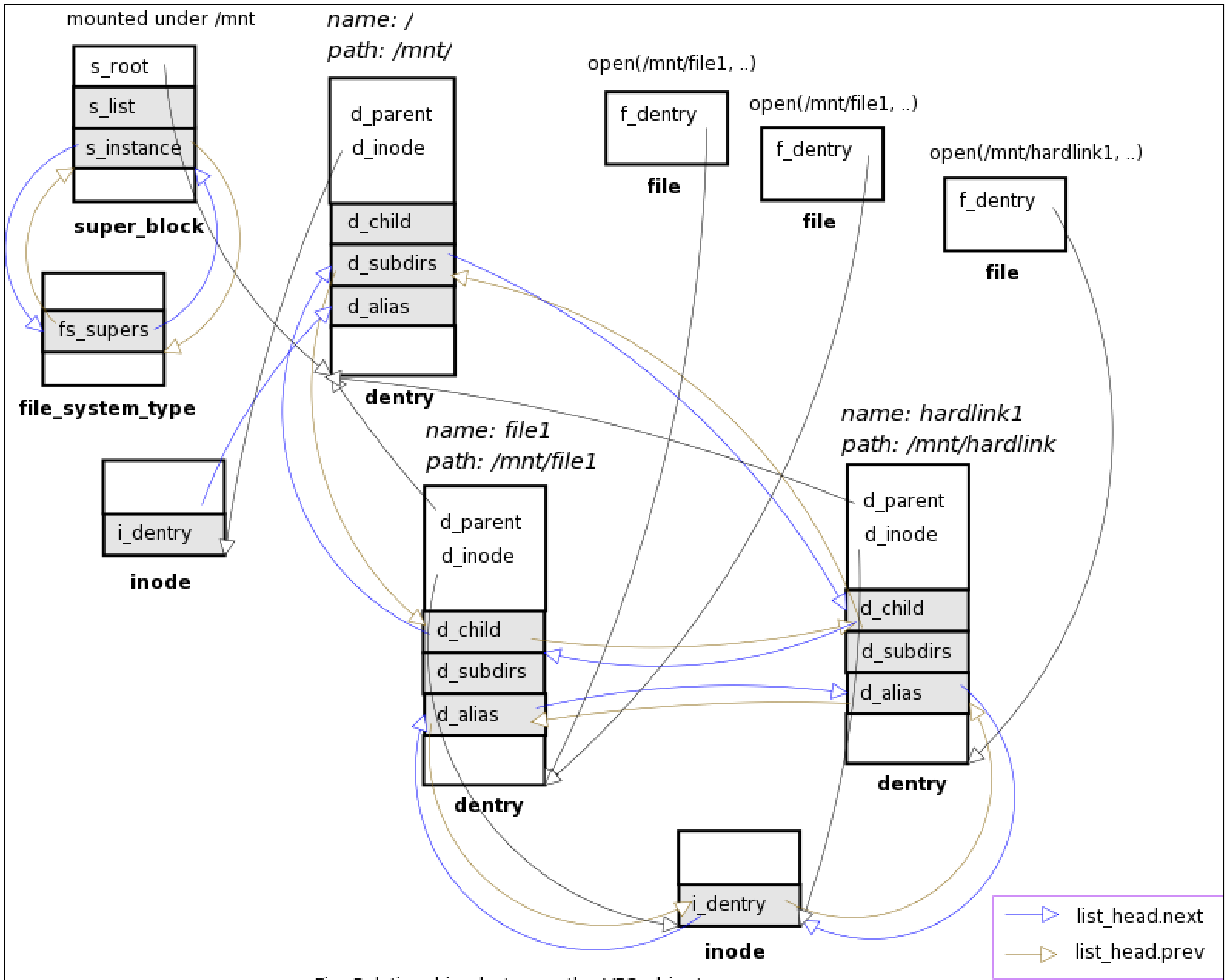

Fig: Relationships between the VFS objects

i. File Operations (struct file_operations)
Definition found in *include/linux/fs.h*

This is the file operations table with each field corresponding to a function pointer to handle the task. It has fields like read, write, readdir, llseek etc.

All these structures have fields used by the kernel in maintaining internal data structures like lists and hash tables etc. So, we cannot use local/global obects. Kernel allocates the object and passes it to our functions so that we can fill the required fields. If we have to allocate the objects, we need to use the corresponding allocator functions.

3. Implementation

File System Type
The filesystem is registered with the kernel during the module initialization. During this step, two handlers get_sb and kill_sb. get_sb is called at the time of mounting the file system. kill_sb is called at the time of unmounting the file system. The get_sb handler uses the kernel helper function get_sb_single to allocate the super block. We pass a callback, rkfs_fill_super, to this function which will be called to fill the super block structure. Since we dont have any specific task to do in the the kill_sb handler, we can use the kernel helper function kill_anon_super.

Super Block Operations
We'll register the read_inode and write_inode callbacks in the super operations table. read_inode will be called when the inode object is newly allocated. Inodes are identified by the inode numbers. Based on the inode number, the file system will have to resolve the inode object on the storage and fill the fields of the inode with the contents on the storage. In our case it is simple as we are implementing a memory based file system. File systems like ext2 will have to read the inode from the disk using the inode

number. Depending on the file type, different handlers can be registered for the file operations table.

write_inode handler will be called to sync the inode contents to the storage. In our case, we will update the file size variable in memory.

Inode Operations
We'll register the lookup callback in the inode operations table. A dentry object will be allocated by the kernel and passed to the handler. The name component is set on the dentry by the kernel. In the handler, we have to check whether an entry by that name exists under the parent inode. If an entry exists, the inode object is obtained by passing the super block object and the inode number to the function iget. This uses the inode cache and will allocate a new inode if it is not avaiable in the inode cache. (Note that when the inode is newly allocated, the read_inode handler of the super block is called to fill the inode). This inode is added to the dentry object using d_add. The return value should be a NULL on successful lookup or an error code otherwise.

File Operations
We'll register read, write and readdir handlers in the file operations table. read and write handlers are called to read and write data into the file. readdir is called to read the contents of a directory.

The following table shows the fields we need to fill in the above data structures.

File System Type	**Super Block**	**Inode**	**DEntry**
• name • *get_sb* • *kill_sb* • s_op	• s_root	• i_ino • i_mode • i_op • i_fop	• d_inode

The following table shows the operation tables and the handlers used.

Super Operations	**File Operations**	**Inode Operations**	**Address Space Operations**
• *read_inode* • *write_inode*	• *read* • *write* • *readdir*	• *lookup*	• *readpage* • *writepage* • *prepare_write* • *commit_write*

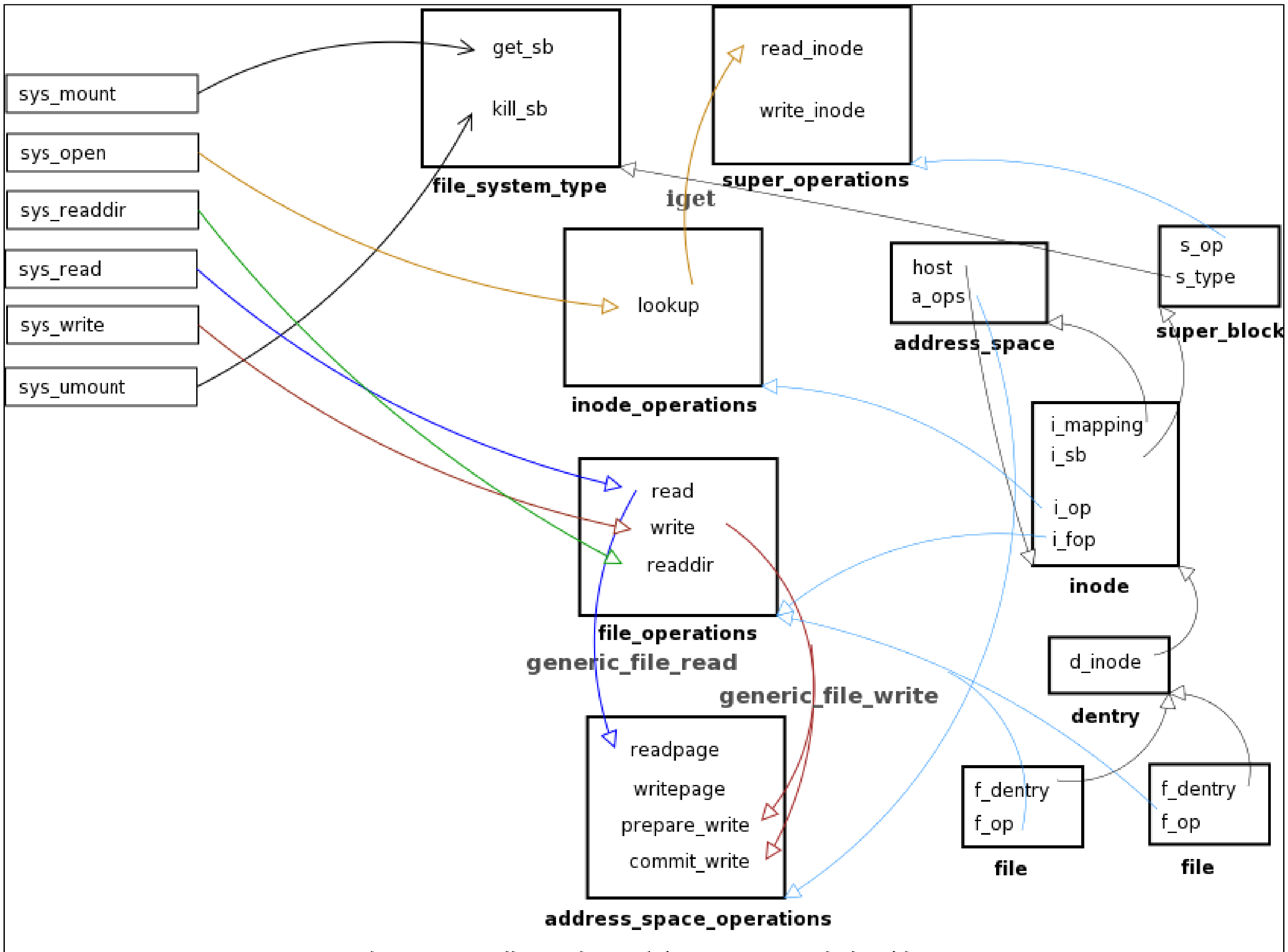

Fig: System call mapping and data structure relationships

4. Entry points

a. init_module

This is called when the module is loaded. We have to register our file system here. Fill the file_system_type strucure with name and read_super fields and call register_filesystem with the structure. For example,

```
static struct file_system_type rkfs = {
    name:       "rkfs",
    get_sb:    rkfs_get_sb,
    kill_sb: rkfs_kill_sb,
    owner:      THIS_MODULE
};

int init_module(void) {
    int err;
    err = register_filesystem( &rkfs );
    return err;
}
```

b. file_system_type.get_sb

This will be called when the file system is mounted. We have to return a super block. We use the helper function get_sb_simple to do the super block allocation and also passing rkfs_fill_super callback to fill the super block object. The s_op field is set with the address of the super block operations table rkfs_sops. The root most inode of the file system has to be allocated at this stage. The dentry for it should be set on the s_root field of the super block. As mentioned earlier, this is the entry point of lookup operations into the file system.

The inode object is allocated using the function iget. After initializing the inode, the dentry is allocated using the function d_alloc_root. This dentry is set to the s_root field of the super block.

```
static int
rkfs_fill_super(struct super_block *sb, void *data, int silent)
{
    sb->s_blocksize = 1024;
    sb->s_blocksize_bits = 10;
    sb->s_magic = RKFS_MAGIC;
    sb->s_op = &rkfs_sops; // super block operations
    sb->s_type = &rkfs; // file_system_type

    rkfs_root_inode = iget(sb, 1); // allocate an inode
    rkfs_root_inode->i_op = &rkfs_iops; // set the inode ops
    rkfs_root_inode->i_mode = S_IFDIR|S_IRWXU;
    rkfs_root_inode->i_fop = &rkfs_fops;

    if(!(sb->s_root = d_alloc_root(rkfs_root_inode))) {
        iput(rkfs_root_inode);
        return -ENOMEM;
    }

    return 0;
}

static struct super_block *
rkfs_get_sb(struct file_system_type *fs_type, int flags, const char *devname, void *data) {
     /* rkfs_fill_super this will be called to fill the superblock */
    return get_sb_single(
        fs_type,
        flags,
        data,
        &rkfs_fill_super);
}
```

Let us assume that our file system is mounted under /mnt/rkfs.

c. file_system_type.kill_sb

This is called at the time of unmounting the file system. We use the helper function kill_anon_super for this.

d. super_operations.read_inode

Inodes objects should be allocated using the iget function. This uses the inode cache, adjusts the relationships of the inode with various data structures, updates the count etc. If the inode is not cached, it allocates a new inode and calls the read_inode handler of the super block if present. Inodes are identified by the inode numbers. The handler is expected to initialize the inode with the contents of the inode on the backend.

e. super_operations.write_inode

This will be called when the dirty inodes are flushed. The handler has to sync the inode contents to the backend.

d. inode_operations.lookup

This will be called when the kernel is resolving a path. The lookup handler of the inode operation table of the parent inode is called to resolve a child. Remember that the dentry for the root most inode is already available in s_root field of the super block.

For example, after mounting the file system under '/mnt/rkfs' if we want to see the contents using 'ls /mnt/rkfs', the kernel has to create a file object for the inode '/mnt/rkfs'. The kernel will create a file object with the dentry of the root most inode of the file system. For the command 'ls -l /mnt/rkfs/hello.txt', the kernel name lookup reaches the root most inode and the lookup handler will be called to set the inode of 'hello.txt'. The kernel allocates the dentry object and passes to the handler. If an inode exists for the name component, the inode has to be added to the dentry using d_add and NULL should be returned. If there is some problem, a suitable error code has to be returned.

```
static struct dentry *
rkfs_inode_lookup(struct inode *parent_inode, struct dentry *dentry, struct nameidata *nameidata) {
    struct inode *file_inode;
    if(parent_inode->i_ino != rkfs_root_inode->i_ino)
        return ERR_PTR(-ENOENT);
    if(dentry->d_name.len != strlen("hello.txt") ||
       strncmp(dentry->d_name.name, "hello.txt", dentry->d_name.len))
      return ERR_PTR(-ENOENT);

    file_inode = iget(parent_inode->i_sb, FILE_INODE_NUMBER);
    if(!file_inode)
        return ERR_PTR(-EACCES);
    file_inode->i_size = file_size;
    file_inode->i_mode = S_IFREG|S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;
    file_inode->i_fop = &rkfs_fops;
    //  file_inode->i_fop
    d_add(dentry, file_inode);
    return NULL;
}
```

e. file_operations.readdir

This will be called when the kernel wants to read the contents of a directory. The readdir handler of the file operations table of the file will be called to show the contents of the directory. The kernel passes the file object, the dirent structure and a callback to fill the dirent structure with the values of the contents of the directory. The values are added to the dirent structure using the 'filldir' callback. Since we support one hardcoded file 'hello.txt', we just have to return the values, '.', '..' and 'hello.txt'. File systems like ext2 will have to fetch the contents from the disk.

```
int rkfs_f_readdir( struct file *file, void *dirent, filldir_t filldir ) {
    int err;
    struct dentry *de = file->f_dentry;

    printk( "rkfs: file_operations.readdir called\n" );
    if(file->f_pos > 0 )
        return 1;
    if(filldir(dirent, ".", 1, file->f_pos++, de->d_inode->i_ino, DT_DIR)||
      (filldir(dirent, "..", 2, file->f_pos++, de->d_parent->d_inode->i_ino, DT_DIR)))
        return 0;
    if(filldir(dirent, "hello.txt", 9, file->f_pos++, FILE_INODE_NUMBER, DT_REG ))
        return 0;
    return 1;
}
```

In our file system, we are supporting only one file i.e., hello.txt. So, the result of 'ls /mnt/rkfs' will be

```
. .. hello.txt
```

f. file_operations.read

This will be called when the kernel gets a read request for a file in our file system. The file object of the file to be read, the user-space buffer address, the maximum size of the buffer and the address of the offset (which contains the current offset and which has to be updated after successful read operation) are passed. The contents of the file have to be written to the buffer. Note that this is in the user-space. The data is copied to the user-space buffer using the function copy_to_user.

There are two ways of supporting this operation. One way is to provide a read handler which writes the data to the buffer. But the drawback is that we cannot take advantage of the page cache. Files can also be read/written using the mmap (memory mapping). With this approach, we cannot support the mmap way of accessing the file (or it is very difficult to provide transparency i.e., file written after mapping and read with 'read' system call).
The second way is to provide a unified way to read/write to the file for both the approaches i.e., calling the system calls directly or by mapping the file and reading/writing the contents in memory). This takes the advantage of page cache also. This is applicable to the write operation also.

Let us take the second approach (the code for the first approach is also provided later). The approach is slightly different in this case. The contents of an inode are seen as chunks of pages and represented by addess_space object. This 'mapping' between the inode and the address space object is stored in the i_mapping field of the inode. To read some data from the page, the corresponding chunks/pages which holds the data are loaded into memory.

Address Space Operations table is used to perform different operations on the address space object (a_ops field). The readpage handler of the table is used to read the contents of a page of the inode into memory. For example, if the page size is 4096, the data from 5000 to 6000 bytes is present in the 2nd page of the inode (similarly, the data from 4000 to 5000 is present in the pages 1 and 2).

Since the actual work of reading the data is moved to address_space_operations.readpage handler, we can use the generic_file_read helper function as the read handler. This function get the pages of the data and copies to the user-space buffer. If the pages are not in the cache, it waits till the pages are loaded with the data using the 'readpage' handler.

g. address_space_operations.readpage

The readpage handler has to fill the page with the contents of the inode. The index of the page is obtained from the 'index' field of the page structure.

```
static int
rkfs_readpage(struct file *file, struct page *page) {
    printk("RKFS: readpage called for page index=[%d]\n", (int)page->index);
    if(page->index > 0) {
        return -ENOSPC;
    }

    SetPageUptodate(page);
    memcpy(page_address(page), file_buf, PAGE_SIZE);
    if(PageLocked(page))
        unlock_page(page);
    return 0;
}
```

h. file_operations.write

We register generic_file_write as the write handler. This allocates pages and calls prepare_write handler on the address space object so that buffer head objects can be allocated for the page to perform the I/O to the device later. It copies the data from the user-space to the pages and calls commit_write on the address space object.

File systems like ext2 generally have a specific implementation of prepare_write. The allocation of buffer head objects for the write operation will be same for most of the file systems except for the location of the buffer heads (buffer head is the kernel's copy of a disk block). In this case, they use the helper/wrapper function block_prepare_write and passing a callback (in case of ext2, it is ext2_get_block) which will give the block number for the file offset.

Since writing the buffers associated with the pages to the device is similar to most of the file systems, they normally use generic_commit_write helper function. This marks the buffer as dirty so that it will be flushed later to the device by the block device layer.

This writes the data from user-space to the pages. The pages are synced later. Note that the read/write happens on the cached pages.

i. address_space_operations.commit_write

The generic_commit_write helper function sets up the buffers to be written to the disk. Since we are not using any device, this has been modified to write into the memory buffer of the file.

```
static int
rkfs_commit_write(struct file *file, struct page *page,
            unsigned from, unsigned to) {
    struct inode *inode = page->mapping->host;
    loff_t pos = ((loff_t)page->index << PAGE_CACHE_SHIFT) + to;

   if(page->index == 0) {
       memcpy(file_buf, page_address(page), PAGE_SIZE);
       ClearPageDirty(page);
   }

    /*
     * No need to use i_size_read() here, the i_size
     * cannot change under us because we hold i_sem.
     */
    if (pos > inode->i_size) {
        i_size_write(inode, pos);
        mark_inode_dirty(inode);
    }


    SetPageUptodate(page);
    return 0;
}
```

i. address_space_operations.writepage

This will be called when the dirty pages are flushed.

```
static int
rkfs_writepage(struct page *page, struct writeback_control *wbc) {
    printk("[RKFS] offset = %d\n", (int)page->index);
    memcpy(file_buf, page_address(page), PAGE_SIZE);
    ClearPageDirty(page);
    if(PageLocked(page))
      unlock_page(page);
    return 0;
}
```

k. cleanup_module

This will be called when the module is removed. We have to unregister our file system at this point. The module count will be incremented and decremented by the file system calls. So the module will not be removed if the module use count is not zero. The kernel takes care of this, so we need not do anything to check if our file system is in use.

5. code

Note: this has some debugging messages... you can remove all the 'printk's.

rkfs.c

6. Instructions to use the code

This code works on the 2.6 kernels. I haven't tested it on 2.4 kernels.

Compile using the 2.6 build system. Create a Makefile like this: (or you can download the code here)

```
ifneq (${KERNELRELEASE},)
obj-m += rkfs.o
else
KERNEL_SOURCE := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C ${KERNEL_SOURCE} SUBDIRS=$(PWD) modules
clean :
    rm *.o *.ko
endif
```

```
make
```

This generates a file rkfs.o. Load the module as root using

```
insmod rkfs.o
```

Mount the file system using

```
mount -t rkfs none /mnt/rkfs
```

Unmount using

```
umount /mnt/rkfs
```

Unload the module using

```
rmmod rkfs
```