The Linux Virtual Filesystem Implementation

In this wiki, I will introduce about Linux file system sub system and how we can investigate a bug related with a file system.

Chapter 1: Linux virtual file system basic

Linux kernel is a combined of few major components. Each major component is called subsystem.

Example of kernel subsystem:

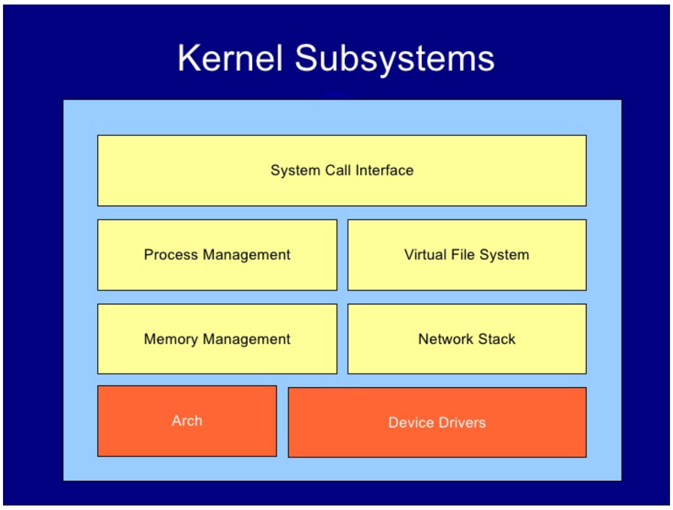
+ Process managerment.

+ Virtual file system.

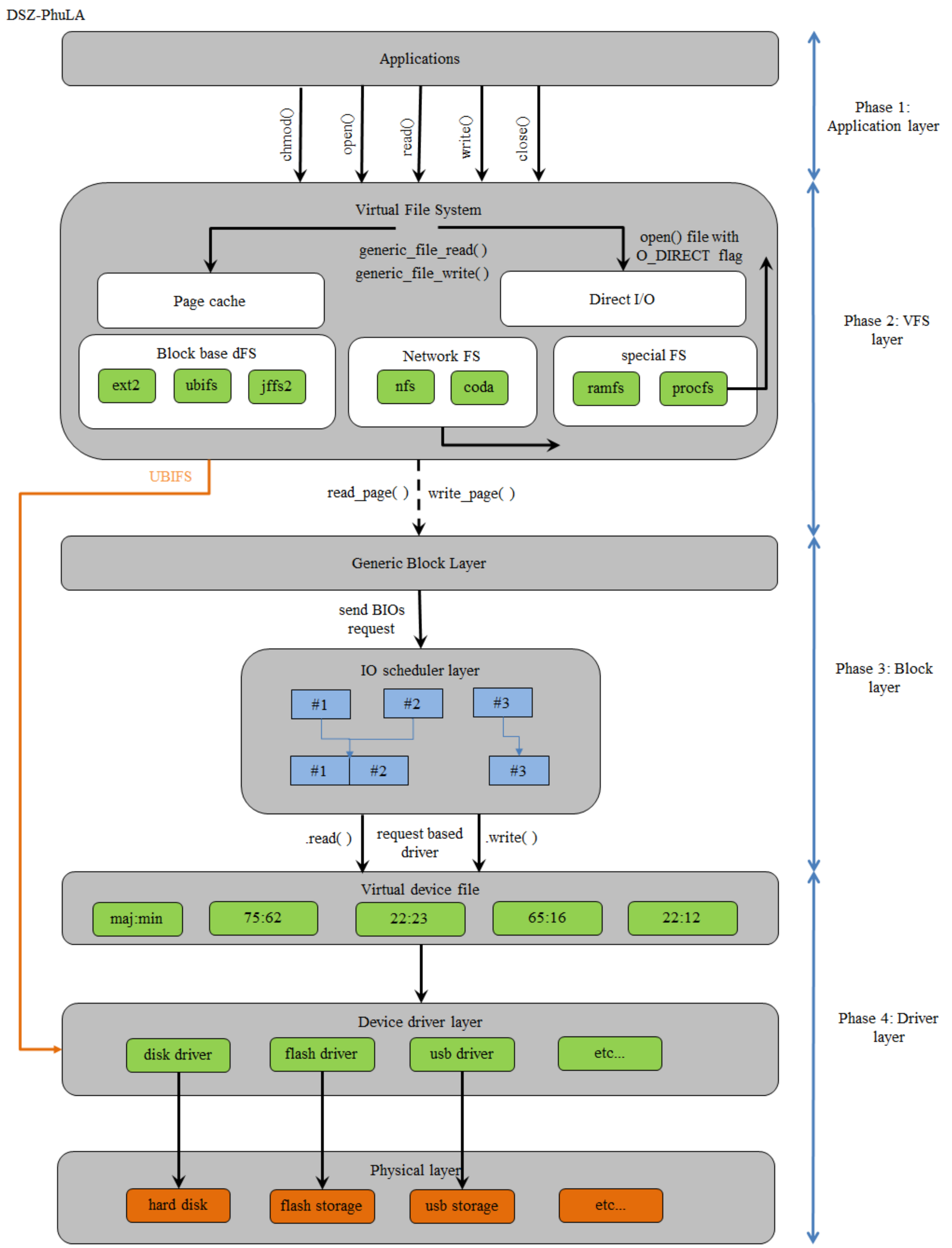
+ Memory managerment.

+ Network stack.

+ etc....



We can see a model of VFS subsystem in bellow picture:



The Linux virtual file system subsytem include 4 basically layers. Each layer service request of above layer and send request to bellow layer.

You can think the idea of VFS layer is similar with TCP/IP model layer.

Each layer have some basically object types. All operation in each layer will access its object type.

+ **Application layer**: Provide file system tree. It hidden the different type of files in its file system tree. It provide a interface fo human. Ex: File name, directory, file attribute ....

In this layer, it have only one basically object type called: File descriptor. This object include 2 important field: file name and file operation pointer.

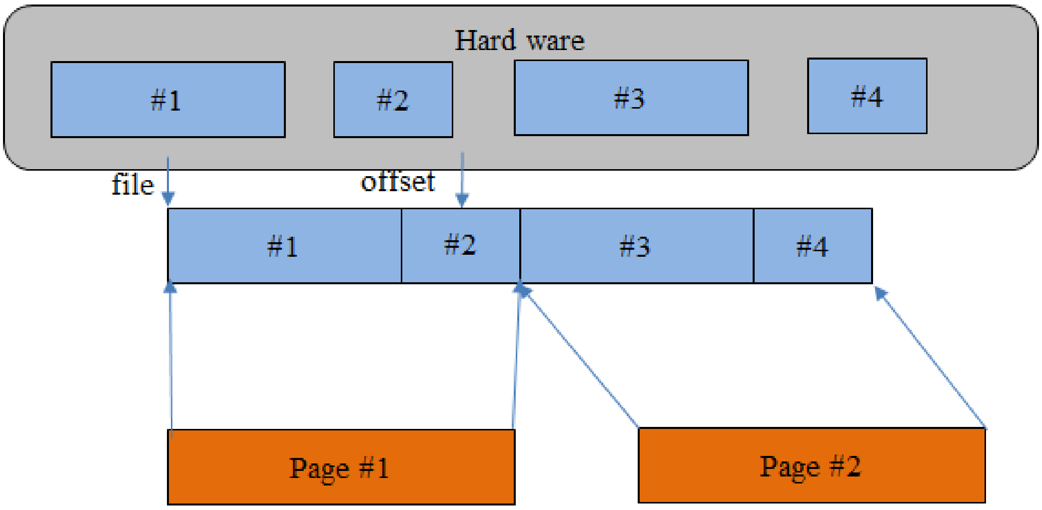
```
struct file {
 struct path  f_path; // Struct store the file name.
 struct inode  *f_inode; // Struct store the identify of a file. It is
unique.
 const struct file_operations *f_op; //Struct store all function
pointers, they point to defination of file operation like read, write,
open ...

 atomic_long_t  f_count; // count reference with this file.
 unsigned int   f_flags; // Flag set when we open a file.
 fmode_t   f_mode; // File mode
 loff_t   f_pos;
 struct fown_struct f_owner; // File owner
 /* needed for tty driver, and maybe others */
 void   *private_data; // Store private data, it belong each file sytem.
Will be send to file when kernel send a request to it.
};
```

+ **Virtual filesystem layer**: This layer service request recevied from application layer. In this layer include many specific file system. So it have to know this request is belong which file system and send recevied request to this file system.

Morever, in this layer, it maps a file with memory address space. It try to reduce I/O requests in a device, instead, it use a cached memory. With cached memory, it ignore the different between each storage format.

It maps file and offset into pages cached.



To implement feature maps file tp page cached, VFS layer provide a set of general file functions:

- generic_file_read()
- generic_file_write()
- generic_file_llseek()
- etc .....

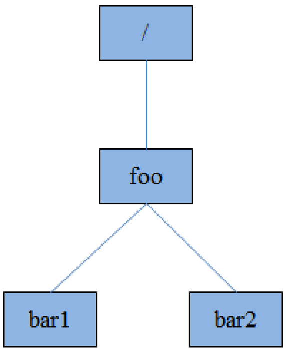
But, to call these above functions or not, it belong each specific file system.

```
const struct file_operations ubifs_file_operations = {
 .llseek         = generic_file_llseek,
 .read           = do_sync_read,
 .write          = do_sync_write,
 .aio_read       = generic_file_aio_read,
 .aio_write      = ubifs_aio_write,
 .mmap           = ubifs_file_mmap,
 .fsync          = ubifs_fsync,
};
```

In VFS layer, it have 4 basically object types:

- file object: It is created when a process open a file.
- inode object: Stores general information about a specific file. Each inode object is associated with an inode number, which uniquely identifies the file within the filesystem.
- dentry object: Stores information about the linking of a directory entry with the corresponding file.
- superblock object: Stores information concerning a mounted filesystem.

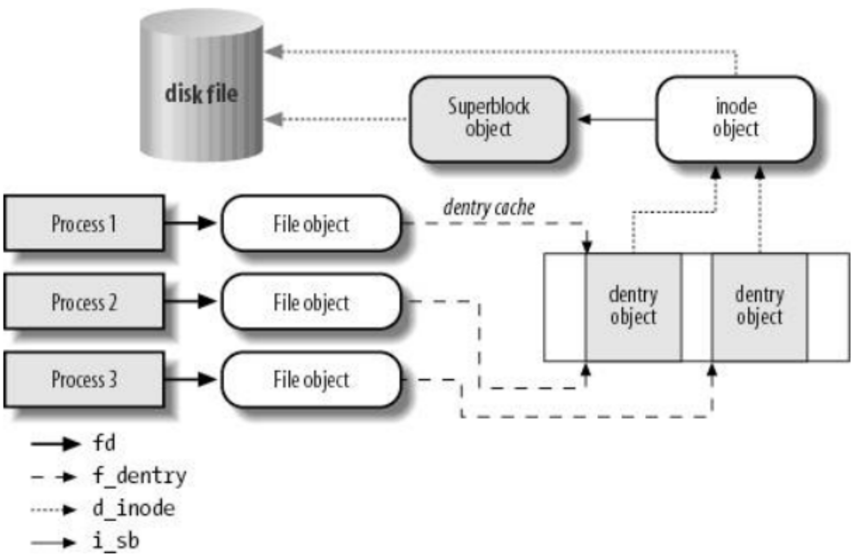
Example, in this tree:



It is represented by four inodes: one each for foo, bar, and bar2, and the root. And 3 dentries: one linking bar to foo, one linking bar2 to foo, and one linking foo to the root

The first entry object include name "bar" in its content.

Figure 12-2. Interaction between processes and VFS objects



+ Block layer: It it designed for improve performance when storage is hard disk. So in some modern file system for flash storage or usb storage, the file system send request directly to hard ware and skip this layer.

This layer have purpose to delay the request sent from VFS layer, and try to combine two or more request to be one request if these request access to neare physical location in disk.

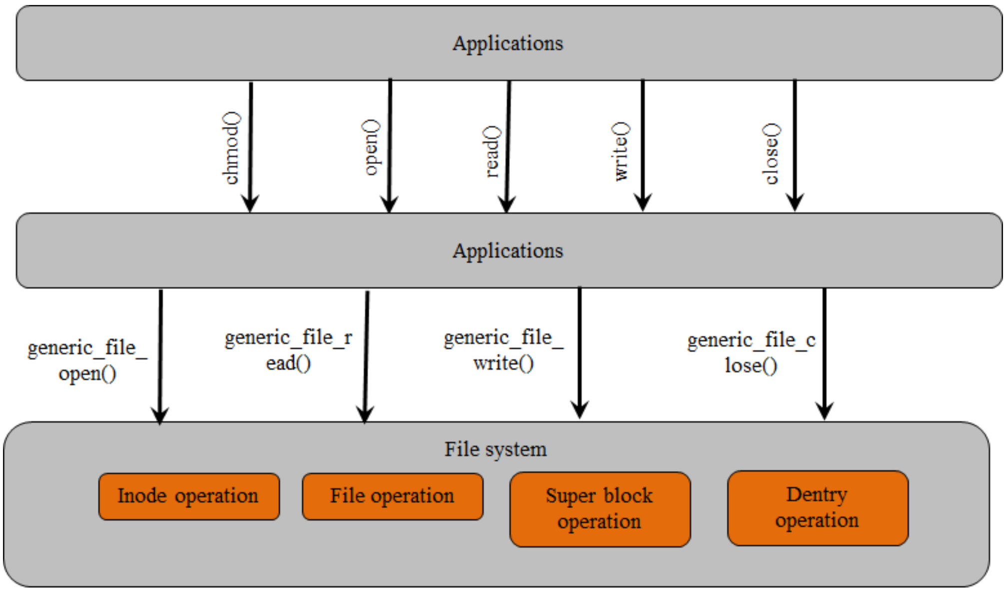
After combined request, it will schedule request and chose a request to service.

Basically object in this layer is BIO structure:

```
struct bio {
 struct block_device *bi_bdev;
 struct bvec_iter bi_iter;
 /* Number of segments in this BIO after
  * physical address coalescing is performed.
  */
 unsigned int  bi_phys_segments;
 /*
  * To keep track of the max segment size, we account for the
  * sizes of the first and last mergeable segments in this bio.
  */
 unsigned int  bi_seg_front_size;
 unsigned int  bi_seg_back_size;
 atomic_t  __bi_remaining;
 bio_end_io_t  *bi_end_io;
 void   *bi_private;
};
```

Chapter 2: Debuging a file system.

Basic component of a file system:



+ Inode operation:

```
const struct inode_operations ubifs_file_inode_operations = {
 .setattr     = ubifs_setattr,
 .getattr     = ubifs_getattr,
#ifdef CONFIG_UBIFS_FS_XATTR
 .setxattr    = ubifs_setxattr,
 .getxattr    = ubifs_getxattr,
 .listxattr   = ubifs_listxattr,
 .removexattr = ubifs_removexattr,
#endif
};
```

+ File operation:

```
const struct file_operations ubifs_file_operations = {
 .llseek         = generic_file_llseek,
 .read           = do_sync_read,
 .write          = do_sync_write,
 .aio_read       = generic_file_aio_read,
 .aio_write      = ubifs_aio_write,
 .mmap           = ubifs_file_mmap,
 .fsync          = ubifs_fsync,
 .unlocked_ioctl = ubifs_ioctl,
 .splice_read = generic_file_splice_read,
 .splice_write = generic_file_splice_write,
#ifdef CONFIG_COMPAT
 .compat_ioctl   = ubifs_compat_ioctl,
#endif
};
```

+ Super block operation:

```
const struct super_operations ubifs_super_operations = {
 .alloc_inode   = ubifs_alloc_inode,
 .destroy_inode = ubifs_destroy_inode,
 .put_super     = ubifs_put_super,
 .write_inode   = ubifs_write_inode,
 .delete_inode  = ubifs_delete_inode,
 .statfs        = ubifs_statfs,
 .dirty_inode   = ubifs_dirty_inode,
 .remount_fs    = ubifs_remount_fs,
 .show_options  = ubifs_show_options,
 .sync_fs       = ubifs_sync_fs,
};
```

+ Page cached operation:

```
const struct address_space_operations ubifs_file_address_operations = {
 .readpage       = ubifs_readpage,
 .writepage      = ubifs_writepage,
 .write_begin    = ubifs_write_begin,
 .write_end      = ubifs_write_end,
 .invalidatepage = ubifs_invalidatepage,
 .set_page_dirty = ubifs_set_page_dirty,
 .releasepage    = ubifs_releasepage,
};
```

Base on type of file operation, kernel will call corresponding function pointer.

Example:

- fopen("/foo/bar", "w"); alloc_inode(), f_op→open()
- fwrite(fd, buff, len): f_op→write()
- fclose(fd): f_op→close(), .write_page(), .write_inode.

File system initialization:

Purpose: When kernel mount a file system, it register its operation for kernel, so when kernel need, it can call a corresponding function of this file system.

```c
static int __init ubifs_init(void)
{
	int err;

	err = register_filesystem(&ubifs_fs_type);
```

```c
static struct file_system_type ubifs_fs_type = {
	.name    = "ubifs",
	.owner   = THIS_MODULE,
	.get_sb  = ubifs_get_sb,
	.kill_sb = kill_anon_super,
};
```

```c
static int ubifs_get_sb(struct file_system_type *fs_type, int flags,
			const char *name, void *data, struct vfsmount *mnt)
{
	struct ubi_volume_desc *ubi;
	struct ubi_volume_info vi;
	struct super_block *sb;
	int err;

	sb->s_flags = flags;
	/*
	 * Pass 'ubi' to 'fill_super()' in sb->s_fs_info where it is
	 * replaced by 'c'.
	 */
	sb->s_fs_info = ubi;
	err = ubifs_fill_super(sb, data, flags & MS_SILENT ? 1 : 0);
```

```c
static int ubifs_fill_super(struct super_block *sb, void *data, int silent)
{
	struct ubi_volume_desc *ubi = sb->s_fs_info;
	struct ubifs_info *c;
	struct inode *root;
	int err;

	sb->s_op = &ubifs_super_operations;
	mutex_lock(&c->umount_mutex);
	err = mount_ubifs(c);
```