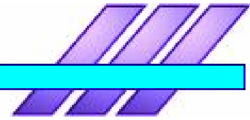
Chapter 10
File System Framework



Design and Implementation of SysPak OS

Abstract

The Virtual File System (otherwise known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programmes. It also provides an abstraction within the kernel which allows different filesystem implementations to co-exist.

10.1 Why Have a File System?

The file system is essential component of all operating systems; it provides the mechanism for the storage and retrieval of files and a hierarchical directory structure for naming of multiple files.

A basic file system enables the operating system to do the following:

- Create and delete files
- Open files for reading and writing
- Seek within a file
- Close files
- Create directories to hold groups of files
- List the contents of a directory
- Removes files from directory

10.2 SysPak OS File System Framework

SysPak OS includes a framework, the virtual file system framework, under which multiple file system types can be implemented.

10.3 Virtual File System - VFS

In this section I'll briefly describe how things work, before launching into the details. I'll start with describing what happens when user programmes open and manipulate files, and then look from the other view which is how a filesystem is supported and subsequently mounted.

10.3.1 Opening a File

The VFS implements the open, stat, chmod and similar system calls. The pathname argument is used by the VFS to search through the directory entry cache (dentry cache or "dcache"). This provides a very fast lookup mechanism to translate a pathname (filename) into a specific dentry.

An individual dentry usually has a pointer to an inode. Inodes are the things that live on disc drives, and can be regular files (you know: those things that you write data into), directories, FIFOs and other beasts. Dentries live in RAM and are never saved to disc: they exist only for performance. Inodes live on disc and are copied into memory when required. Later any changes are written back to disc. The inode that lives in RAM is a VFS inode, and it is this which the dentry points to. A single inode can be pointed to by multiple dentries (think about hardlinks).

The dcache is meant to be a view into your entire filespace. Unlike Linus, most of us losers can't fit enough dentries into RAM to cover all of our filespace, so the dcache has bits missing. In order to resolve your pathname into a dentry, the VFS may have to resort to creating dentries along the way, and then loading the inode. This is done by looking up the inode.

To lookup an inode (usually read from disc) requires that the VFS calls the lookup() method of the parent directory inode. This method is installed by the specific filesystem implementation that the inode lives in. There will be more on this later.

Once the VFS has the required dentry (and hence the inode), we can do all those boring things like open the file, or stat it to peek at the inode data. The stat operation is fairly simple: once the VFS has the dentry, it peeks at the inode data and passes some of it back to userspace.

Opening a file requires another operation: allocation of a file structure (this is the kernel-side implementation of file descriptors). The freshly allocated file structure is initialised with a pointer to the dentry and a set of file operation member functions. These are taken from the inode data. The open() file method is then called so the specific filesystem implementation can do it's work. You can see that this is another switch performed by the VFS.
The file structure is placed into the file descriptor table for the process.
Reading, writing and closing files (and other assorted VFS operations) is done by using the userspace file descriptor to grab the appropriate file structure, and then calling the required file structure method function to do whatever is required.
For as long as the file is open, it keeps the dentry "open" (in use), which in turn means that the VFS inode is still in use. All VFS system calls (i.e. open, stat, read, write, chmod and so on) are called from a process context. You should assume that these calls are made without any kernel locks being held. This means that the processes may be executing the same piece of filesystem or driver code at the same time, on different processors. You should ensure that access to shared resources is protected by appropriate locks.

10.3.2 Registering and Mounting a Filesystem

If you want to support a new kind of filesystem in the kernel, all you need to do is call register_filesystem(). You pass a structure describing the filesystem implementation (struct file_system_type) which is then added to an internal table of supported filesystems.
When a request is made to mount a block device onto a directory in your filespace the VFS will call the appropriate method for the specific filesystem. The dentry for the mount point will then be updated to point to the root inode for the new filesystem.

10.3.3 File System Calls

These are the file system calls that every file system must provide for registering with framework.

```
struct fs_calls {
      int  (*fs_mount)(fs_cookie *fs, fs_id id, const char *device, void
*args, vnode_id *root_vnid);
      int (*fs_unmount)(fs_cookie fs);
      int (*fs_sync)(fs_cookie fs);

      int  (*fs_lookup)(fs_cookie  fs,  fs_vnode  dir,  const  char  *name,
vnode_id *id);

      int (*fs_getvnode)(fs_cookie fs, vnode_id id, fs_vnode *v, bool r);
      int (*fs_putvnode)(fs_cookie fs, fs_vnode v, bool r);
      int (*fs_removevnode)(fs_cookie fs, fs_vnode v, bool r);

      int  (*fs_open)(fs_cookie  fs,  fs_vnode  v,  file_cookie  *cookie,
stream_type st, int oflags);
      int (*fs_close)(fs_cookie fs, fs_vnode v, file_cookie cookie);
      int (*fs_freecookie)(fs_cookie fs, fs_vnode v, file_cookie cookie);
      int (*fs_fsync)(fs_cookie fs, fs_vnode v);

      ssize_t (*fs_read)(fs_cookie fs, fs_vnode v, file_cookie cookie, void
*buf, off_t pos, ssize_t len);
      ssize_t  (*fs_write)(fs_cookie  fs,  fs_vnode  v,  file_cookie  cookie,
const void *buf, off_t pos, ssize_t len);
      int  (*fs_seek)(fs_cookie  fs,  fs_vnode  v,  file_cookie  cookie,  off_t
pos, seek_type st);
      int (*fs_ioctl)(fs_cookie fs, fs_vnode v, file_cookie cookie, int op,
void *buf, size_t len);
```

```
      int (*fs_canpage)(fs_cookie fs, fs_vnode v);
      ssize_t (*fs_readpage)(fs_cookie fs, fs_vnode v, iovecs *vecs, off_t
pos);
      ssize_t (*fs_writepage)(fs_cookie fs, fs_vnode v, iovecs *vecs, off_t
pos);

      int  (*fs_create)(fs_cookie  fs,  fs_vnode  dir,  const  char  *name,
stream_type st, void *create_args, vnode_id *new_vnid);
      int (*fs_unlink)(fs_cookie fs, fs_vnode dir, const char *name);
      int (*fs_rename)(fs_cookie fs, fs_vnode olddir, const char *oldname,
fs_vnode newdir, const char *newname);

      int (*fs_rstat)(fs_cookie fs, fs_vnode v, struct file_stat *stat);
      int (*fs_wstat)(fs_cookie fs, fs_vnode v, struct file_stat *stat, int
stat_mask);
};
```

Table 10.1: File System Calls

10.3.4 VFS calls for Virtual Memory System

These calls are used by VM system for performing paging operations:

```
int vfs_get_vnode_from_fd(int fd, bool kernel, void **vnode);
int vfs_get_vnode_from_path(const char *path, bool kernel, void **vnode);
int vfs_put_vnode_ptr(void *vnode);
void vfs_vnode_acquire_ref(void *vnode);
void vfs_vnode_release_ref(void *vnode);
ssize_t vfs_canpage(void *vnode);
ssize_t vfs_readpage(void *vnode, iovecs *vecs, off_t pos);
ssize_t vfs_writepage(void *vnode, iovecs *vecs, off_t pos);
void *vfs_get_cache_ptr(void *vnode);
int vfs_set_cache_ptr(void *vnode, void *cache);
```

10.4 Stream Types

Following types of streams are supported by the VFS

```
      STREAM_TYPE_ANY = 0,
      STREAM_TYPE_FILE,
      STREAM_TYPE_DIR,
      STREAM_TYPE_DEVICE,
      STREAM_TYPE_PIPE
```