CHAPTER 1

INTRODUCTION

1.1 MOTIVATION AND OVERVIEW

ReactOS is an open-source, alpha-stage operating system intended to be binary-compatible with application software and device drivers made for Microsoft Windows NT versions 5.x and up (Windows 2000 and its successors). It is composed entirely of free software, by means of a complete clean room reverse engineering process. ReactOS is primarily written in the C programming language, with some elements, such as ReactOS Explorer, written in C++. Various components of ReactOS are licensed under the GNU General Public License, the GNU Lesser General Public License, and the BSD License.

Currently ReactOS is restricted to the FAT filesystem and has its inherent disadvantages. The objective of our project is to integrate the Virtual Filesystem layer, as in Linux kernel, in the ReactOS kernel.

1.2 LITERATURE SURVEY

The latest stable version of ReactOS (version 0.3.9 alpha) is designed to work with the FAT filesystem. The FAT filesystem was a commonly used filesystem with Windows till Windows 98, NT and XP. The Microsoft identified some limitations of the FAT filesystem and chose to discontinue its use paving the way for the NTFS filesystem. The inherent limitations of FAT filesystem (FAT 12/ FAT 16/ FAT 32) are:

1. You cannot format a volume larger than 32 gigabytes (GB) in size using the FAT32 file system.
2. FAT has a fixed maximum number of clusters per partition, which means as the hard disk gets bigger the size of each cluster must increase, creating more slack space.
3. FAT slows down as the number of files on the disk increases.
4. FAT usually fragments files more.
5. You cannot create a file larger than 2^{32}-1 bytes on a FAT32 partition.

All these limitations come into play with ReactOS too. So by adding a VFS layer as with the Linux kernel it would be possible to mount any filesystem type with ReactOS. This means that the ReactOS kernel would be compatible with most of the filesystem like the Extended filesystems (EXT2, EXT3), NTFS, NFS etc. The integration of the VFS layer is focused of increasing the exposure, compatibility and reliability of the ReactOS.

1.3 THE PROPOSED VFS DRIVER

The functionality of VFS layer of the Linux kernel is extracted and is to be ported into the ReactOS kernel. The VFS functionality is to be added as a driver into the ReactOS kernel. The structure of a ReactOS driver is similar to a Windows driver as it is based on the Windows NT architecture.

The Windows driver programming is done using the Windows Driver Development Kit (WDK) that includes all the building environment, compilers, linkers, and tools required. The Windows Driver that is equivalent to the ReactOS

driver is written using the Visual C++. The driver is an executable file with extension .SYS.

It may also be done by implementing the driver in the main code and compiling it with the ReactOS kernel. By doing this the use of propreitory programs like WDK or use of languages like visual C++ can be avoided.

CHAPTER 2

REQUIREMENTS

2.1 HARDWARE REQUIREMENTS

The project requires a normal computer that has x86-compatible processor (Pentium or later), atleast 512 MB RAM, IDE hard disk of at least 500 MB. The main partiontion of the ReactOS system can be FAT16 or FAT32 boot partition. The Linux can be run together for build environment. So an EXT 3 partition for Linux is also required.

2.2 SOFTWARE REQUIREMENTS

The ReactOS 0.3.7 source code can be compiled to get a bootable disk image. Qemu software is used for emulating a system on a GNU/Linux machine. GNU/Linux 2.6.26 is also required for running ReactOS Build Environment – RosBE 1.2. This build environment is required to compile the system.

CHAPTER 3

THE VIRTUAL FILESYSTEM

The Virtual File System (otherwise known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to user space programs. It also provides an abstraction within the kernel which allows different filesystem implementations to co-exist. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

The Virtual Filesystem handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

For instance, let us assume that a user issues the shell command:

$ cp /floppy/TEST /tmp/test

where /floppy is the mount point of an MS-DOS diskette and /tmp is a normal Ext2 (Second Extended Filesystem) directory. The VFS is an abstraction layer between the application program and the filesystem implementations. Therefore, the cp program is not required to know the filesystem types of /floppy/TEST and /tmp/test. Instead, cp interacts with the VFS by means of generic system calls.

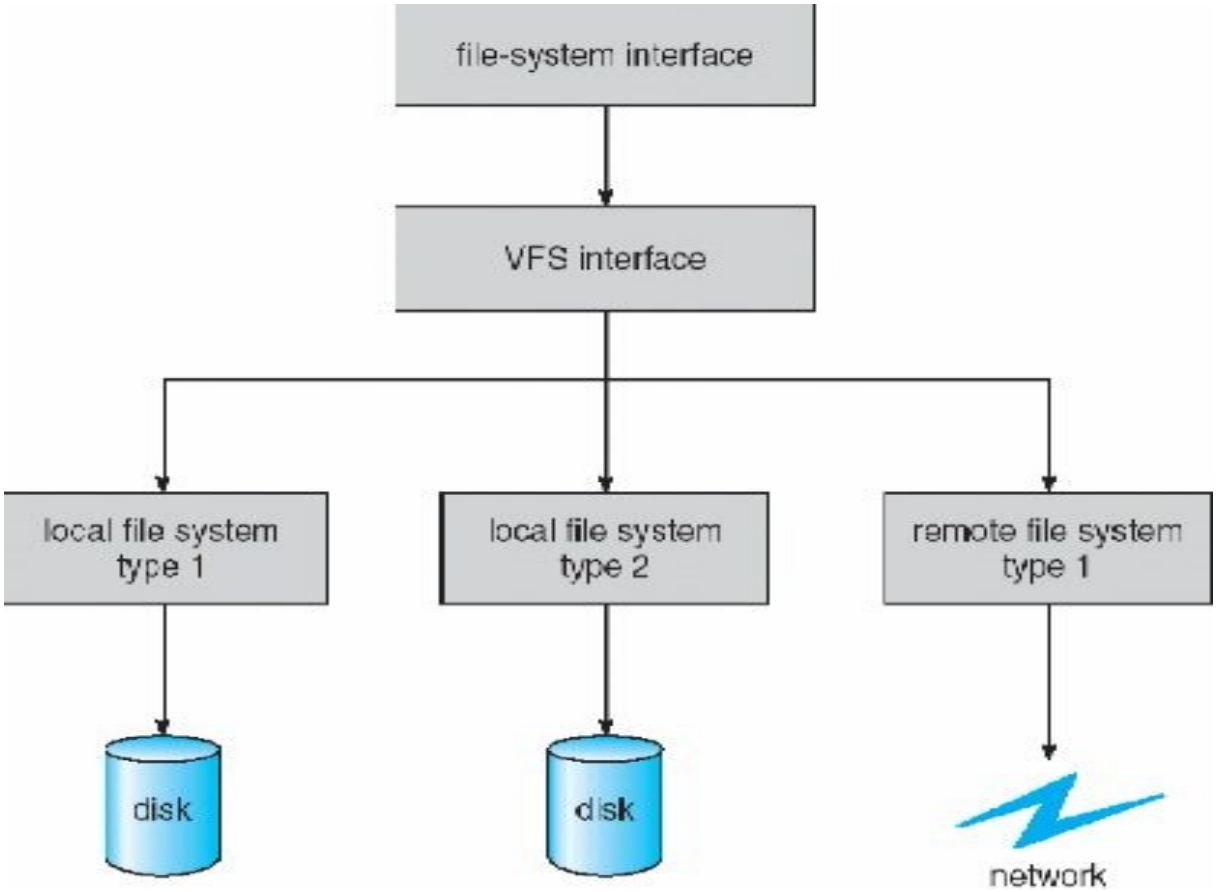

Fig 3.1 Schematic view of Virtual Filesystem

3.1 PRINCIPLE

When a process issues a file oriented system call, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical filesystem code, which is responsible for handling the structure dependent operations. Filesystem code uses the buffer cache functions to request I/O on devices. This scheme is illustrated in this figure:

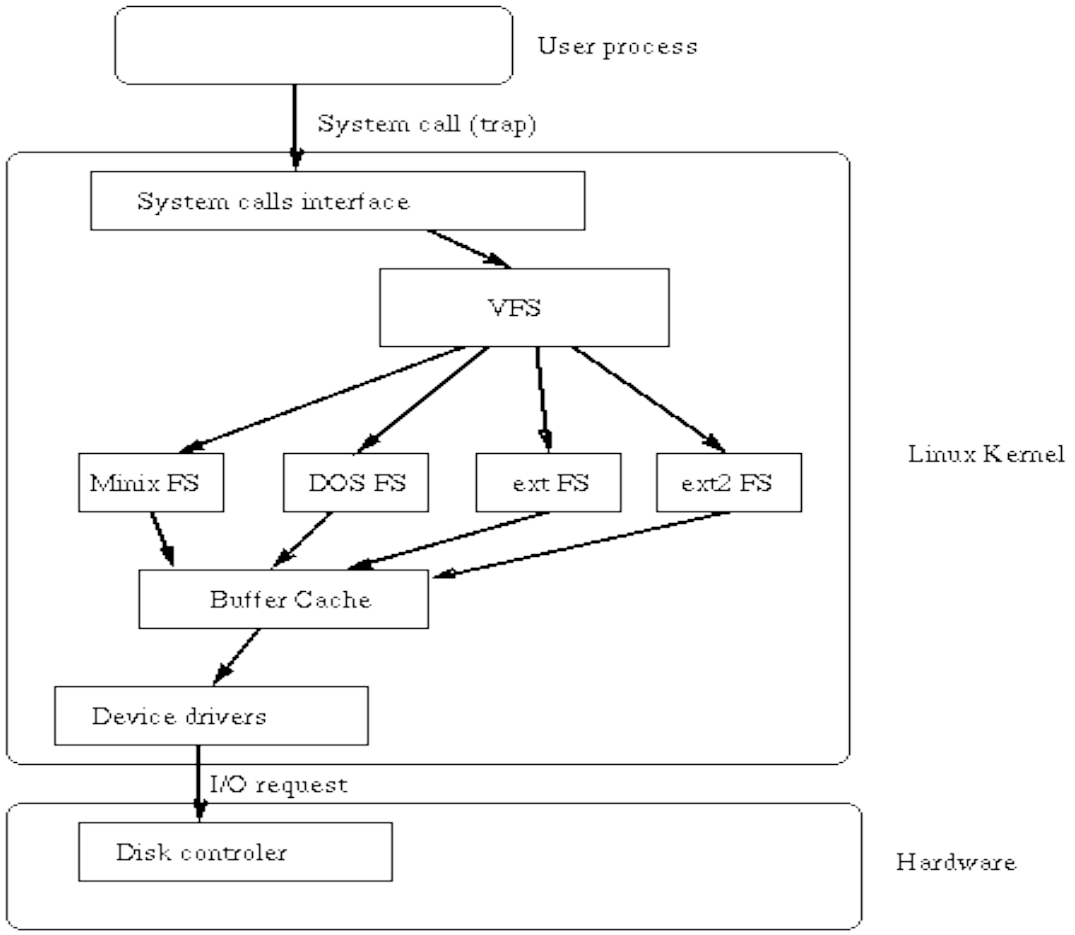

Fig 3.2: Schematic view of Linux Kernel

As each file system is initialized, it registers itself with the VFS. This happens as the operating system initializes itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module, and then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure.

The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 [7] specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the "/" directory.

The VFS implements the open(2), stat(2), chmod(2) 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.

3.2 VFS STRUCTURE AND OPERATIONS

The VFS defines a set of functions that every filesystem has to implement. This interface is made up of a set of operations associated to three kinds of objects: filesystems, inodes, and open files. 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.

The dcache is meant to be a view into the entire filespace. In order to resolve the 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. Once the VFS has the required dentry (and hence the inode),it can do operations like open(2) the file, or stat(2) it to peek at the inode data. The stat(2) operation is fairly simple: once the VFS has the dentry, it peeks at the inode data and passes some of it back to userspace.

All VFS system calls (i.e. open(2), stat(2), read(2), write(2), chmod(2) and so on) are called from a process context. It is assumed 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. The access to shared resources is protected by appropriate locks.

3.3 REGISTERING A FILESYSTEM

In order to support a new kind of filesystem in the kernel, call the system call register_filesystem(). It passes a structure describing the filesystem implementation (struct file_system_type) which is then added to an internal table of supported filesystems. To see what filesystems are currently available on your system do:

```
% cat /proc/filesystems
```

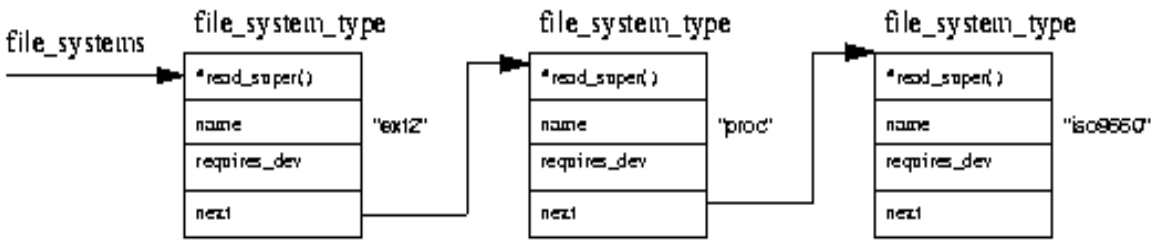

Fig 3.2: Registered filesystem list

3.4 MOUNTING A FILESYSTEM

When the superuser attempts to mount a file system, the Linux kernel must first validate the arguments passed in the system call. Although mount does some basic checking, it does not know which file systems this kernel has been built to support or that the proposed mount point actually exists. Consider the following mount command:

$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom

This mount command will pass the kernel three pieces of information; the name of the file system, the physical block device that contains the file system and, thirdly, where in the existing file system topology the new file system is to be mounted. The first thing that the Virtual File System must do is to find the file system. To do this it searches through the list of known file systems by looking at each file_system_type data structure in the list pointed at by file_systems.

If it finds a matching name it now knows that this file system type is supported by this kernel and it has the address of the file system specific routine for reading this file system's superblock. If it cannot find a matching file system name then all is not

lost if the kernel is built to demand load kernel modules. In this case the kernel will request that the kernel daemon loads the appropriate file system module before continuing as before.

Next if the physical device passed by mount is not already mounted, it must find the VFS inode of the directory that is to be the new file system's mount point. This VFS inode may be in the inode cache or it might have to be read from the block device supporting the file system of the mount point. Once the inode has been found it is checked to see that it is a directory and that there is not already some other file system mounted there. The same directory cannot be used as a mount point for more than one file system.

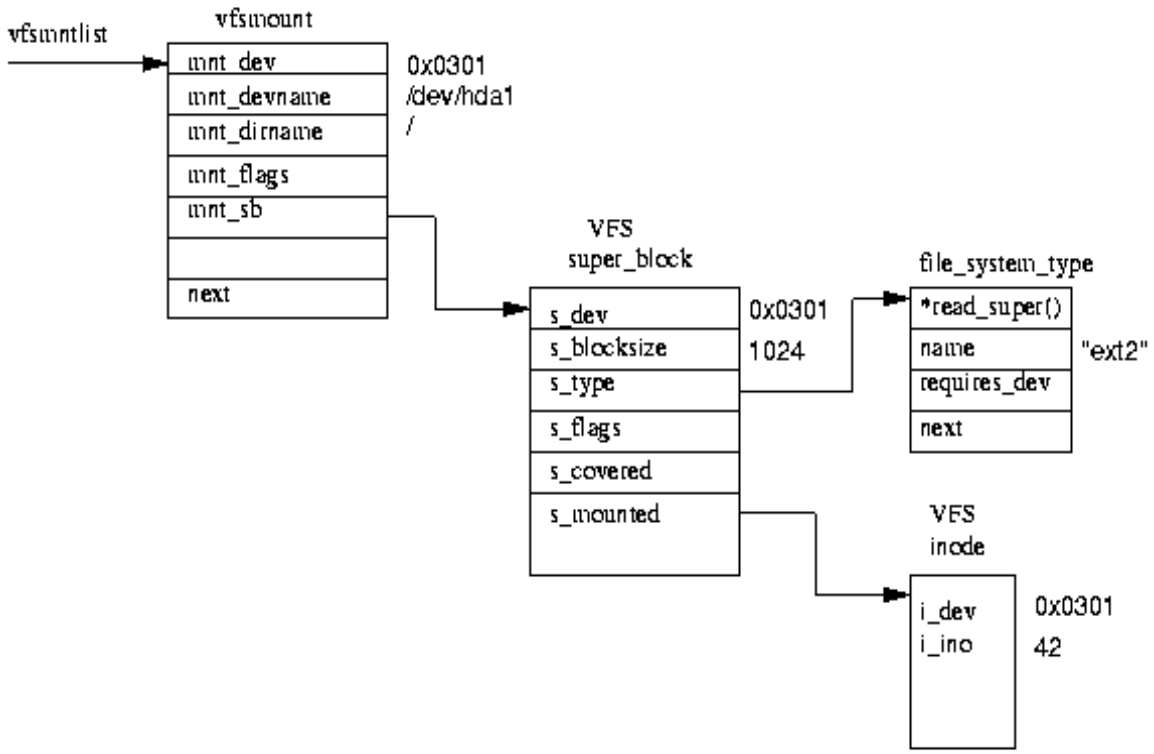

Fig 3.3: A mounted filesystem

At this point the VFS mount code must allocate a VFS superblock and pass it

the mount information to the superblock read routine for this file system. All of the system's VFS superblocks are kept in the super_blocks vector of super_block data structures and one must be allocated for this mount. The superblock read routine must fill out the VFS superblock fields based on information that it reads from the physical device. Whatever the file system, filling out the VFS superblock means that the file system must read whatever describes it from the block device that supports it. If the block device cannot be read from or if it does not contain this type of file system then the mount command will fail. Each mounted file system is described by a vfsmount data structure. These are queued on a list pointed at by vfsmntlist.

Another pointer, vfsmnttail points at the last entry in the list and the mru_vfsmnt pointer points at the most recently used file system. Each vfsmount structure contains the device number of the block device holding the file system, the directory where this file system is mounted and a pointer to the VFS superblock allocated when this file system was mounted. In turn the VFS superblock points at the file_system_type data structure for this sort of file system and to the root inode for this file system. This inode is kept resident in the VFS inode cache all of the time that this file system is loaded.

3.5 FINDING A FILE IN THE VIRTUAL FILE SYSTEM

To find the VFS inode of a file in the Virtual File System, VFS must resolve the name a directory at a time, looking up the VFS inode representing each of the intermediate directories in the name. Each directory lookup involves calling the file system specific lookup whose address is held in the VFS inode representing the parent directory. This works because we always have the VFS inode of the root of

each file system available and pointed at by the VFS superblock for that system. Each time an inode is looked up by the real file system it checks the directory cache for the directory. If there is no entry in the directory cache, the real file system gets the VFS inode either from the underlying file system or from the inode cache.

3.6 UNMOUNTING A FILE SYSTEM

A file system cannot be unmounted if something in the system is using one of its files. So, for example, it is not possible to umount /mnt/cdrom if a process is using that directory or any of its children. If anything is using the file system to be unmounted there may be VFS inodes from it in the VFS inode cache, and the code checks for this by looking through the list of inodes looking for inodes owned by the device that this file system occupies. If the VFS superblock for the mounted file system is dirty, that is it has been modified, then it must be written back to the file system on disk. Once it has been written to disk, the memory occupied by the VFS superblock is returned to the kernel's free pool of memory. Finally the vfsmount data structure for this mount is unlinked from vfsmntlist and freed.

3.7 IMPLEMENTATION OF VIRTUAL FILESYSTEM IN LINUX KERNEL

In order to support multiple filesystems, Linux contains the special kernel interface level called VFS (Virtual Filesystem Switch). The software layer in the kernel provides the filesystem interface to userspace programmes. It also provides an abstraction within the kernel which allows different filesystem implementations to co-exist.

3.7.1 FILESYSTEM REGISTRATION/UNREGISTRATION

The Linux kernel provides a mechanism for new filesystems to be written with minimum effort. The code to implement a filesystem can be either a dynamically loadable module or statically linked into the kernel, and the way it is done under Linux is very transparent. All that is needed is to fill in a struct file_system_type structure and register it with the VFS using the register_filesystem() function.

The struct file_system_type is declared in include/linux/fs.h:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt;
        struct file_system_type * next;
};
```

The fields thereof are explained thus:

- **name**: human readable name, appears in /proc/filesystems file and is used as a key to find a filesystem by its name; this same name is used for the filesystem type in **mount(2)**, and it should be unique: there can (obviously) be only one filesystem with a given name. For modules, name points to module's address

spaces and not copied: this means cat /proc/filesystems can oops if the module was unloaded but filesystem is still registered.

- **fs_flags**: one or more (ORed) of the flags: FS_REQUIRES_DEV for filesystems that can only be mounted on a block device, FS_SINGLE for filesystems that can have only one superblock, FS_NOMOUNT for filesystems that cannot be mounted from userspace by means of mount(2) system call: they can however be mounted internally using kern_mount() interface, e.g. pipefs.

- **read_super**: a pointer to the function that reads the super block during mount operation. This function is required: if it is not provided, mount operation (whether from userspace or inkernel) will always fail except in FS_SINGLE case where it will Oops in get_sb_single(), trying to dereference a NULL pointer in fs_type->kern_mnt->mnt_sb with (fs_type->kern_mnt = NULL).

- **owner**: pointer to the module that implements this filesystem. If the filesystem is statically linked into the kernel then this is NULL. You don't need to set this manually as the macro THIS_MODULE does the right thing automatically.
- **kern_mnt**: for FS_SINGLE filesystems only. This is set by kern_mount() (TODO: kern_mount() should refuse to mount filesystems if FS_SINGLE is not set).

- **next**: linkage into singly-linked list headed by file_systems (see fs/super.c). The list is protected by the file_systems_lock read-write spinlock and functions register/unregister_filesystem() modify it by linking and unlinking the entry from the list.

The unregister_filesystem function is used to remove the file_system_type entry of a particular filesystem from the INIT_LIST_HEAD doubly linked list.

3.7.2 OPEN(), READ(), WRITE() AND CLOSE() SYSTEM CALLS

Opening a file requires an operation: allocation of a file structure (this is the kernel-side implementation of file descriptors). The freshly allocated file structure is initialized 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 its 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 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. As long as the file is open, it keeps the dentry "open" (in use), which means that the VFS inode is still in use.

The read() call when received by the VFS layer invokes the generic_file_read() function in /usr/src/linux-2.6-2.6.18.dfsg.1/mm/filemap.c that redirects the call. The actual reading is performed by generic_file_aio_read(). This is the read routine for all filesystems that can use the page cache directly.

The write() calls received by the VFS layer calls the function generic_file_write() in the "/usr/src/linux-2.6-2.6.18.dfsg.1/mm/filemap.c" which

invokes a generic_file_write_nolock() that initializes I/O Control block. The actual write operation is done by the __generic_file_aio_write_nolock().

CHAPTER 4

REACTOS

ReactOS is an open-source, alpha-stage operating system intended to be binary-compatible with application software and device drivers made for Microsoft Windows NT versions 5.x and up (Windows 2000 and its successors). It is composed entirely of free software, by means of a complete clean room reverse engineering process.

While the ReactOS kernel has been written from scratch, the userland is mostly based on the Wine compatibility layer for Unix-like operating systems. The user-mode part of ReactOS is almost entirely WINE-based.

ReactOS is primarily written in the C programming language, with some elements, such as ReactOS Explorer, written in C++. Various components of ReactOS are licensed under the GNU General Public License, the GNU Lesser General Public License, and the BSD License.

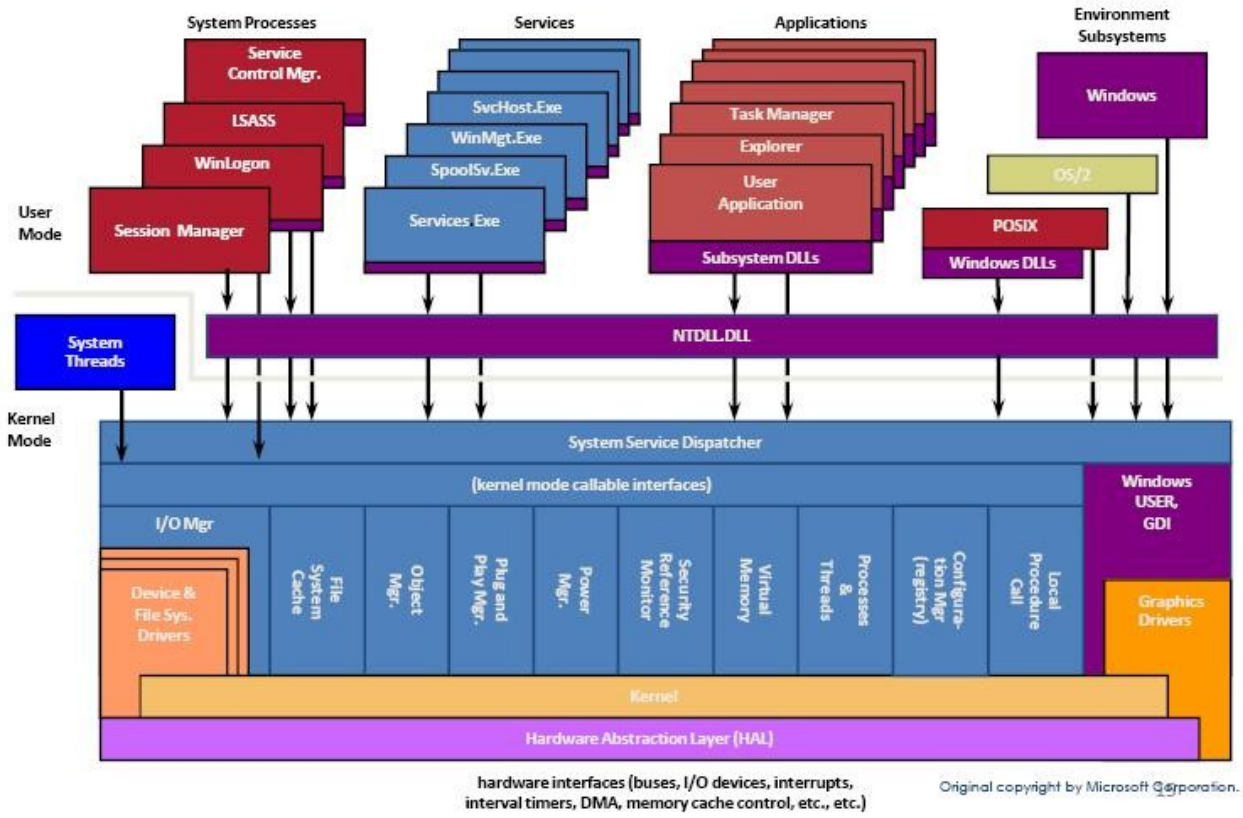

Fig 4.1: Windows NT architecture followed by ReactOS

It is aimed at providing the users who likes a Windows like environment but an open source kernel to work on. ReactOS is a replacement for Windows users who want a Windows replacement that behaves just like Windows.

The Windows family has always gone out of its way to ensure a stable API and backwards compatibility. By its design, ReactOS will also follow the philosophy of backwards compatibility with existing and future applications designed for the Windows NT family.

4.1 CURRENT AND FUTURE DEVELOPMENT

The ReactOS developers are currently working on support for USB. Development is taking place to add networking, multimedia, plug-and-play hardware support, and improving the GUI system. Java and .NET support (through Mono) has also been stubbed. Provisions for DOS, OS/2, and POSIX subsystems have also been made, similarly to the Windows NT subsystems.

The developers aim to make the kernel more compatible with Windows NT versions 5 and 6, and add support for more applications. Improved USB, networking, and other hardware support may also be available, while support for file sharing services with SMB and NTFS file system support may be added.

4.2 FEATURES

- Compatible

Actually, the ReactOS project re-implements a state-of-the-art and open NT alike operating system based on the NT architecture-design. It comes of course with a Win32 subsystem, NT driver compatibility and a handful of useful applications and tools. ReactOS combines the power and strengths of the NT kernel - which is well known for its extensibility, portability, reliability, robustness, performance and compatibility with Win32 compatibility.

- Secure

Despite statements to the contrary, NT is secure by design. It was the first mainstream operating system with a proper implementation of a very flexible security model

based on access control lists. ReactOS will incorporate proper default security settings. ReactOS has been designed for high security; it doesn't share common security flaws with other operating system.

- Lightweight

In short, ReactOS is designed to be powerful and lightweight. You can think of the term "lightweight" in the good old fashion of Win95, a consistent user interface and small bundle of very common and useful tools. In contrast, ReactOS offers a lot more, an up-to-date experience as well as built from scratch on a rock solid NT core.

4.3 REACTOS SUBSYSTEMS

The ReactOS architecture is based on that of Microsoft Windows 2003 Server. The ReactOS architecture, like NT, is monolithic but able to load modules. At the lowest layer is the Executive. The executive includes everything that runs in kernel mode. Above the executive are the Protected Subsystems. These subsystems provide implementations of different Operating System personalities.

- The Executive

The Executive is all the code that runs in kernel mode. The executive can roughly be broken up into the following components: Hardware Abstraction Layer (HAL), Device Drivers, The Kernel, System Services (including the Win32k subsystem) These components all run in kernel mode. The HAL, Kernel, System Services and Device Drivers are collectively referred to as the Executive.

- Hardware Abstraction Layer

The HAL makes it possible for the x86 ReactOS kernel and HAL to run on different x86 motherboards. The HAL abstracts motherboard specific code from the kernel, so that different motherboards do not require changes in the kernel.

- Device Drivers

Device drivers are hardware specific extensions to the ReactOS Executive. They allow the Operating System to interact with certain devices and vice versa.

- Communication

Device drivers use packets to communicate with the kernel and with other drivers. Packets are sent via the I/O Manager (System Service) and make use of IRPs (I/O Request Packets).

- Kernel

The kernel design is based on that of Microsoft Windows 2003 Server. It implements kernel mode Asynchronous Procedure Calls (APCs), Deferred Procedure Calls (DPCs), processes, threading, mutexes, semaphores, spinlocks, timing code, and more.

- System Services

System services include: IO Manager, Configuration Manager, Plug and Play, Power Manager, Memory Manager, Executive Support, Object Manager, security reference monitor, process structure, local procedure call, Win32 Subsystem.

- Protected Subsystems

The Protected Subsystems allow different Operating System personalities to run on top of the ReactOS Executive. The initial target for ReactOS was the Win32k subsystem -- however, the Win32k subsystem runs in kernel mode as part of the Executive and is not featured here. For other graphical subsystems, there exists an Interface for Subsystems via the Win32k Subsystem.

4.4 NATIVE API ARCHITECTURE

The Native API Architecture calls for user mode code to call kernel mode services in a standard manner. It is the equivalent to the System Call Interface used by most UNIXes. The Native API Architecture is implemented in NTDLL.dll. Aside from containing Native API user mode entry points, NTDLL.dll also contains process startup and module loading code. These entry points call KiSystemService in kernel mode, which looks up the kernel mode service in a system table – KiSystemServiceTable.

4.5 BUILDING REACTOS

To build ReactOS, a suitable build environment is needed. As the current ReactOS Source Code is only compatible with specific versions of the compiler tools, only the official ReactOS Build Environment (RosBE) is supported. A Subversion client is needed to obtain the current source from the ReactOS Subversion repository. SVN is the most recent evolution of source code control and versioning systems. SVN is open-source software.

4.5.1 COMMANDS FOR INVOKING A BUILD

make

This command builds all binaries of ReactOS. They will be placed in the directory specified by the ROS_OUTPUT environment variable. (default: *output-i386*) All source files, which did not change since the last build, will not be built again.

make bootcd

This command works like *make*, but also generates a bootable ReactOS ISO file in the base of the working copy. This is usually created for use with qemu.

make livecd

This command generates ReactOS-LiveCD.iso in the base of the working copy. This is the ReactOS Live-CD that runs completely from the CD-ROM

make install

This command copies all the ReactOS binaries to their proper installation directory as specified in the ROS_INSTALL environment variable. (default: *reactos*)

4.5.2 OTHER COMMANDS

clean

This command cleans all files of your working copy except the generated ISO files (if any). The next build you make will be completely clean then.

make depmap

This command generates a simple dependency map for all ReactOS components.

make vreport

This command generates a version report for all ReactOS components, whose source files have appropriate information for that.

4.5.4 BUILDING MODULES

In "reactos/modules" or "reactos\modules", depending on whether you're on a UNIX or Windows system, are two files, *empty.rbuild* and *directory.rbuild*. The Build system will read the *directory.rbuild* file and look if the subfolders named in this file exist. If a folder exists, the Build System will process its *directory.rbuild* file. Otherwise it will fall back to the *empty.rbuild* file.

CHAPTER 5

VIRTUAL FILESYSTEM DRIVER DEVELOPMENT

5.1 REACTOS DRIVER DEVELOPMENT

The ReactOS kernel is based on the Windows NT architecture except that it allows for the addition of modules. The driver for the ReactOS kernel needs to be developed similar to the Windows driver in the Driver Development Kit (DDK). The VFS driver is to be developed with the DDK. The DDK consists of the necessary compiler, linker, and all other tools necessary for the driver development.

Yet another way of doing this is by compiling the whole VFS module anlong with the kernel. This is more advantages and as this module shall then be a part of the release rather than just another external module introduced. Though this beomes a complex task when taking into consideration the dificulty in debugging, we can avoid the use of DDK or Visual Studio.

5.2 LAYERED DRIVER ARCHITECTURE

Windows operating systems support layered driver architecture. Every device is serviced by a chain of drivers, typically called a driver stack. Each driver in the stack isolates some hardware-dependent features from the drivers above it.

The following figure shows the types of drivers that could potentially be in a driver stack for a hypothetical device. In reality, few (if any) driver stacks contain all these types of drivers. And sometimes there can be more layers also within each of

these layers.

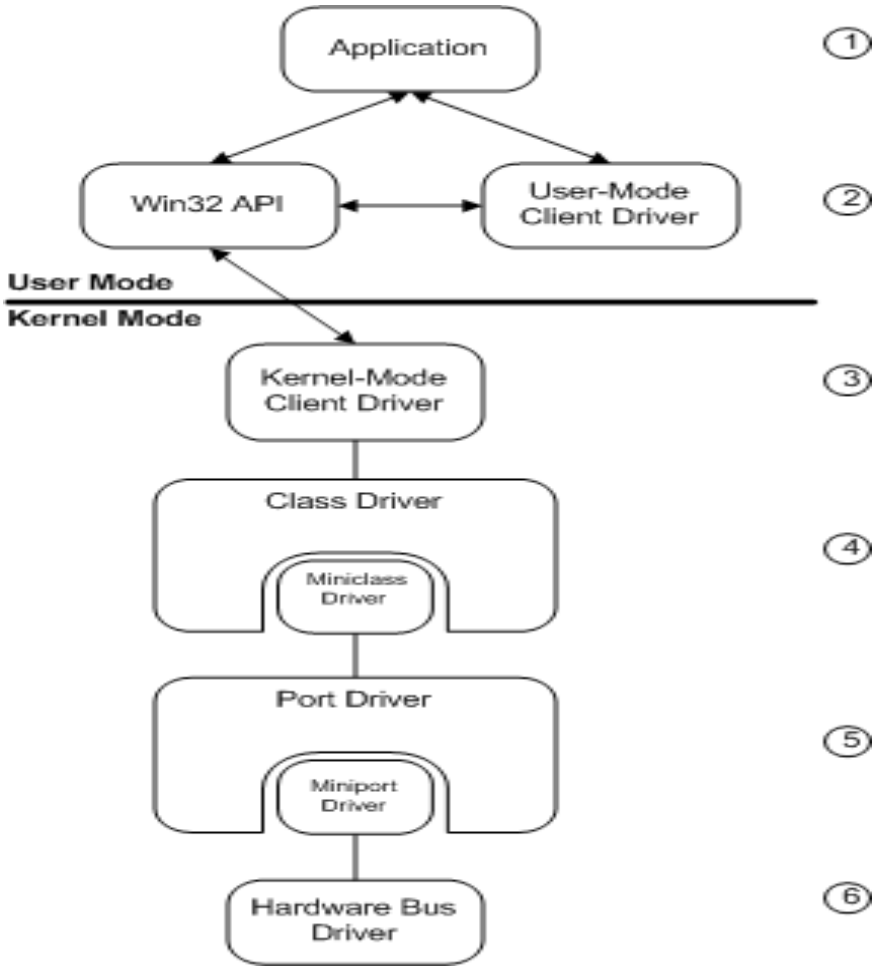

Fig 5.1: Layered architecture of Windows driver

As the preceding figure shows:

1. Above the driver stack is an application. The application handles requests from users and other applications, and calls either the Win32 API or a routine that is exposed by the user-mode client driver.

2. A user-mode client driver handles requests from applications or from the Win32 API. For requests that require kernel-mode services, the user-mode

client driver calls the Win32 API, which calls the appropriate kernel-mode client or support routine to carry out the request. User-mode client drivers are usually implemented as dynamic-link libraries (DLL).

3. A kernel-mode client driver handles requests similar to those handled by the user-mode client, except that these requests are carried out in kernel mode, rather than in user mode.

4. A device class and miniclass driver pair provides the bulk of the device-specific support. The class driver supplies system-required but hardware-independent support for a particular class of device.

5. A corresponding port driver (for some devices, this is a host controller or host adapter driver) supports required I/O operations on an underlying port, hub, or other physical device through which the device attaches. Whether any such drivers are present depends on the type of device and the bus to which it eventually connects.

6. At the bottom of the figure is the hardware bus driver.

5.3 INF FILES

Microsoft Windows drivers must have an INF file in order to be installed. An INF file is a text file that contains all the information necessary to install a device, such as driver names and locations, registry information, version information that is used by the Setup components. This is based on a standard developed by the

Microsoft for creating a uniform driver installation method. The latest DDK adheres to this model.

But when it is being compiled along with the kernel there is no need of an INF file as it is coming pre-installed with the system. Thus there is no installation assosiated with it this way.

CHAPTER 6

DESIGN AND IMPLEMENTATION

The VFS layer is present in the Linux kernel. This is extracted as such, or in other words the data structures, the functions and their functionalities are all re-implemented in the ReactOS. In actual case the code was actually copied as such, including the header files. This thus provided the basic framework of the VFS.

Now these functionalities are to be provided in the ReactOS code. For this a VFS driver is created which is aware of both worlds. The ReactOS provides a driver entry function that is called first while initialising the driver. The driver registers itself with the kernel before actually becoming functional. This is attained by filling up certain data structure. This structure holds the data like which function must be called when a particular action needs to be done.

These functions shall be mostly present within the main driver code. But for easier implementation and maximum re-use of the existing code, the better way was to access the code from the driver. This way the driver tried to access inodes, for example, that were defined in the code imported from the Linux kernel. The basic requirements of a driver must be satisfied along with the working as a filesystem handler.

CHAPTER 7

FUTURE ENHANCEMENTS

There is a large room for enhancement for the VFS driver. Typically, there are some further work to be performed in the driver. The driver needs to be completed with all the operations and facilities provided in Linux VFS layer. The ported layer has to be tested thoroughly before releasing it with an official release of ReactOS.

The other filesystems like extended filesystems (EXT2/EXT3), NTFS etc could be mounted on the virtual filesystem layer. For this the driver for these filesystem has to be written for the VFS driver we have already created. Porting of NFS is not a necessity but will give the full benifit of the VFS driver.

Cleaning the ReactOS subsystem from the limitations in drive naming can also be corrected as part of this. Support of commands like mount can also be implemented.

CHAPTER 8

REFERENCES

- ReactOS source code: http://www.reactos.org/
- ReactOS documentation: http://www.reactos.org/deoxygen/
- NT Kernel Internals, Microsoft Press.