Towards Automatic File Systems

Kanchan Damle

Project Report
CS736 Advanced Operating Systems
The University of Wisconsin-Madison
damle@wisc.edu

Abstract

File Systems are hard to develop. File-system development is a long process and involves writing huge amount of kernel code by experienced and expert programmers. However, these do not allow customization. Their implementation is not tunable as per application and hence lacks flexibility. Various ways of incremental development have been proposed but they do not rule out kernel consideration completely. In order to aid for flexibility, efficient automatic file system compilers can play the trick. The idea is to develop a mechanism that will generate a file system as per user specifications. This paper presents the concept of automatic file-systems compilers and, implementation details of an installable file system –a step towards automatic file systems.

1 Introduction

Writing file systems or any kernel code is hard, as they contain main lines of complex operating systems code. **Table.1** lists the different types. The kernel is a complex environment to master and small mistakes can cause severe data corruption. **[1]** employs automatic detection and repair of errors in data structures enabling the program to continue and to execute productively even in face of otherwise crippling errors. Typically, data structure consistency is a big problem in complex kernel code. File systems, however, offer a clean data access mechanism that is transparent to the user applications, which is why developers always desire to add new features to the file systems.

Media Type	Common File System	Avg. Code Size (C lines)
Hard Disks	UFS, FFS, EXT2FS	5,000–20,000
Network	NFS	6,000–30,000
CD-ROM	HSFS, ISO-9660	3,000–6,000
Floppy	PCFS, MS-DOS	5,000–6,000

Table.1 Common Native Unix File Systems and Code Sizes for Each Medium.

Although Linux supports many file systems, they are pretty similar: disk-based file-systems, network-based file-systems, etc. They do not provide for tunable features thus not allowing any flexibility. Efficient utilization of resources is also not guaranteed. Attempts have been made for incremental development. Making a file-system stable and efficient takes years of effort and once its stable and working, throwing more features at it might break it down or lead to data corruption.

One of the efficient features of Linux is the division of file-system into a two layer stack. The lower layer is the actual file system implementation and the upper VFS. This two tier system allows having multiple file-systems with aid from VFS for system call redirection to the appropriate file-system. Stackable file systems add another layer of indirection; between VFS and the actual low level file system. Stackable file-systems promise to speed up the development by providing an extensible file system interface. This extensibility allows new features to be added incrementally. To improve performance, these stackable file-systems were designed to run in the kernel. **Fig.1** illustrates the concept of stackable file systems. Unfortunately, using these stackable interfaces often requires writing lots of complex kernel code that is specific to a single operating system platform and also difficult to probe. To avoid complex kernel coding a language was developed for implementing such stackable file-systems **[2].** It employs a language code generator in FiST that allows the developer to describe the core functionality of their file system and then develops the kernel file systems for several platforms. They also created a minimal stackable file system template that allowed adding only the additional features and thus relieving the code generator from dealing with any operating specific aspects of the file-system.

This will still restrict flexibility to enhancements in existing file systems. In order to allow the developer to give file system specifications in a more flexible manner, a wise way would be to have an automatic file system compiler that will generate file-

systems as per his specifications. This paper moves ***"Towards Automatic File Systems"***.

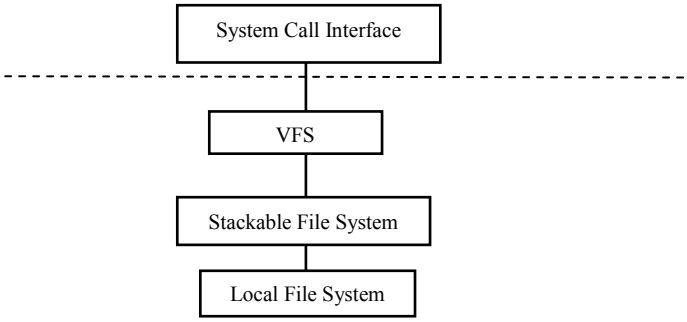

Fig.2 Stackable File Systems

The paper is organized as follows: Section 2 discusses -The 'what', 'why' and 'how' of automatic file systems. Section 3 describes a step towards automatic file systems – Installable file systems. Section 4 focuses on FUSE. Sections 5 and 6 are about implementation details and performance estimation respectively. Sections 7 and 8 discuss about future work and software engineering aspects. Post section 8 paper is summarized and concluded.

2 The 'What', 'Why' and 'How' of Automatic File Systems

Automatic file systems aim to focus on the need of the user. File systems can be automatically constructed by means of a compiler that will allow the user to specify his/her requirements of a file system. Traditional file systems (In this context the ones developed at kernel level) do not provide for flexibility and do not allow tuning of their features. For example the user may not want to have redundant copies of superblock like in FFS to save on disk space but does not want to go back to the old Linux file system to prevent problems on account of the small block size and the scattered free list. He may also accurately know his usage and may want to have lower maximum limits, as in the number of maximum files, size of files etc. Mixing and matching features from different file systems will give a tremendous amount of flexibility to the user. Automatic File System compiler will allow doing this. This is will also ensure efficient resource utilization from the systems perspective since it will allocate the resources, only as per need. This will also reduce overhead for maintaining huge file system metadata just for a few files in case of low utilization.

To have such a compiler in kernel code will increase complexity and will introduce series of problems as discussed earlier. It would be advantageous to have it in user-space since it will be simpler to develop. There are two important aspects hence to be considered about such file system development. First, allowing the user to specify. The compiler generates the file system code and API for it. Secondly, mounting it and allowing the standard file system calls to be redirected to the new file system methods. As mentioned earlier, Linux has implemented file systems by a two tier approach in order to support and adapt to simultaneously coexisting file systems. Refer **Fig.2.** The system level call like unlink() for example, gets translated to standard library calls like sys_unlink. The VFS layer then directs these calls to the appropriate registered file system method without actually knowing its internal operation. Hence, the file system calls for the generated file system should be directed via VFS for uniformity and ease of usage. The generated file system should have its methods registered with VFS hence.

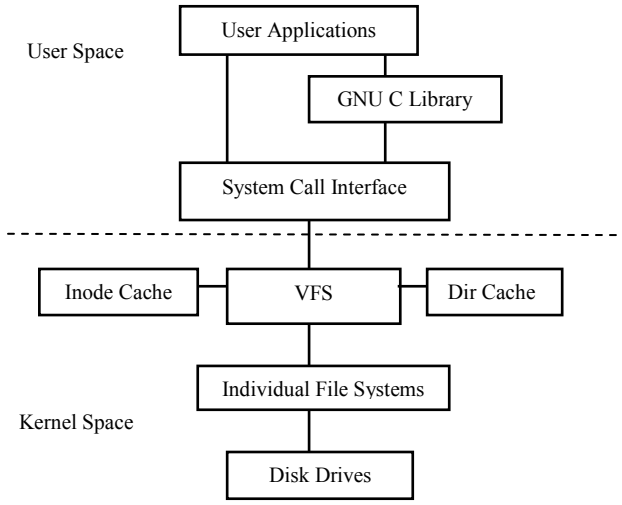

Fig.2 Architecture view of Linux file system components.

User needs to select from multiple options like, whether he will have a bitmap or a free list, or would have redundant data structures, if yes then how many copies, etc. He could also specify the maximum number of files he would use, their probable size etc. Such flexible file-systems can be then made over some predefined framework.

The next few sections elaborate about a step towards automatic file system generation. An installable file system is made, that would allow a file

system in user space without any kernel level privileges. The file system methods are registered with VFS via FUSE (File System in User Space) and hence standard library calls are supported. Accepting parameters dynamically and recompiling the code will make the file-system tunable and facilitate automatic file system generation.

3 An Installable File System

The file-system is developed along a simple framework. It has one superblock (without any redundant copies), inode and data block bitmaps, direct pointers (under the assumption that the file will be of moderate size) and provision to expand the file size by using the reserved blocks under special circumstances. **Fig.3** shows the framework of the file system.

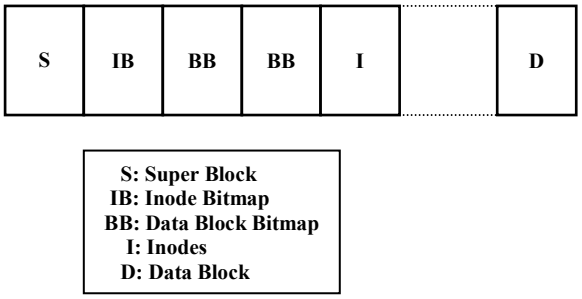

Fig.3 File System Framework

3.1 Implementation Details

3.1.1 On-Disk Structures

Let us assume that the user wants to implement a SFS (Simple File System). The block size is 512 bytes. There will be one super block, a maximum of hundred files and maximum file size to be moderately 15KB. The basic framework includes some reserved blocks by default so as to provide for file size expansion under special circumstances. The maximum number of files open at a time is limited to 25. This file system is implemented and made installable [±].

The superblock contains the file system's administrative information like the number of free inodes, number of free data blocks, magic number that testifies the liveliness of the file system, and maintains information about the maximum limits, etc. Following the superblock is the inode bitmap that holds one sector as the maximum number of files is limited to 100. The data block bitmaps come after that and occupy three sectors as the disk structure is specified to have 10,000 sectors. The inode size is 133 bytes and hence 3 inodes can fit in a sector. This will save the space wastage by allocating one sector per inode. Inodes occupy the disk space till sector 40 after which lie the data blocks. Each inode structure holds the information about the file/directory. It hold the name of the file, its type, size in bytes if file or number of entries if it's a directory, direct pointers to the data blocks where data is located and the permissions to the file/directory. **Fig.4** shows the inode structure. Each directory entry is of size 20 bytes and holds the child file/directory name and its corresponding inode number. Lastly, the filename size is restricted to 16 characters and the pathname to 256. All pathnames to the file system API are absolute and not relative. All these specifications are hardcoded in a header file and can be re-generated after computation in case user specifications change. Persistent storage for metadata structures and data is on a block file on existing file system. It can also be implemented on a raw disk partition like any file system.

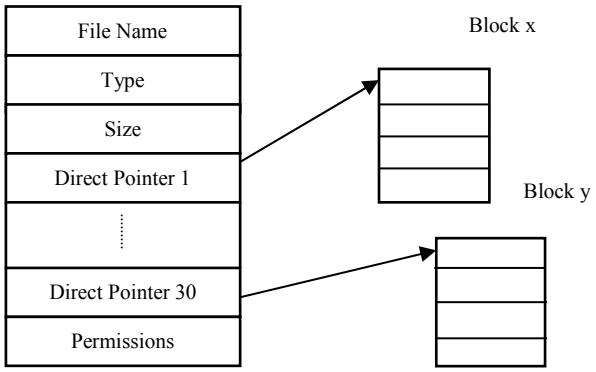

Fig.4 Inode Structure

3.1.2 File System API

The file system data and metadata can be accessed by file system API. The file system API is primarily divided into File level calls and directory level calls. There are few administrative calls that manage the on disk structures. This subsection briefs about these administrative calls handled by the file system transparent to the user and file system calls that can be accessed via the standard library once they are registered with the virtual file system (VFS) module. FUSE (File System in User Space) aids in the

[±] This file system is described throughout the paper, as the implementation has been done with mentioned specifications.

redirection of the standard library file system calls to the generated file system specific calls.

Administrative Calls

FS_Boot: This function initializes the data structures at install time and writes to persistent storage. It also loads the file system into memory during boot time. This enables caching hence reducing the number of writes to the file system. The file system data is flushed out to the persistent storage when any file is closed or on an explicit fsync() call.

FS_Sync: This function flushes the buffer file system from memory to persistent storage. It can be invoked by calling the fsync() function. As mentioned above, this is done at close time of files in order to avoid loss of data due to power loss hence updating persistent storage frequently.

Bitmap_Update: This updates the inode and data block bitmaps when they are allocated or released. The superblock is also updated, since the super block maintains the count of number of free inodes and data blocks. The same function is re-used for checking inode or data block allocation status by passing appropriate parameters to the function.

Find_Free: This function finds lowest free inode and data blocks for allocation. Inodes are required when file or directory is created and data blocks are required when files are written to or directory entries increase.

Traverse_Path: This function lookups in the file system tree and locates the file/directory pointed by ***path*** and returns its inodeno. It also keeps a check on the path length.

Read_Inode: This function locates the sector number and the byte location from its inodeno and reads or updates the inode. This function becomes handy for the file system when new files/directories are created, files/directories are updated, file/directories are deleted etc.

There are other administrative functions that the file system needs to perform. These involve reading or writing whole blocks to and from the disk. The file system calls as specified can be called by means of standard library calls. The file system specific API[β] is as follows:

sfs_getattr(char *path, struct inode): This function traverses the ***path*** and locates the inodeno of the file/directory. It then fills the inode structure. It is basically associated with fetching the file/directory attributes.

sfs_filecreate(char *path): This function creates a file or directory depending on specific parameter passed. It fails if a file/directory with the same name already exists in the same directory. It updates the parent directory, the inode bitmaps and hence the superblock. Inodes are also written to.

sfs_fileopen(char *path): This function traverses down the path specified and locates its inode. It checks the number of files open and opens the new one only if the number opened is below the limit for the file system. It makes the entry for the newly opened file in the open file table and returns a file descriptor to the calling process. This file descriptor is then used to reference the file for reading and writing.

sfs_readfile(int fd, (void *) buf, int size): This function reads size number of bytes from current offset, from the opened file referenced by file descriptor fd. The read bytes are written to the buffer buf. The offset is stored in the open file table once it is opened. The offset can be changed by the sfs_fileseek() function. If the size to be read is greater and the bytes available in the file (due to the end of file) the actual read bytes are less than the one specified. File offset is modified at the end of the read operation.

sfs_writefile(int fd, (void *) buf, int size): This function writes size number of bytes from current offset, to the opened file referenced by file descriptor fd. The bytes are read from buffer buf. The offset is stored in the open file table once it is opened. The offset can be changed by the sfs_fileseek() function. The file offset is updated at the end of the write. For modifying existing data offset has to be changed to the required position.

sfs_fileseek(int fd, int offset): This function updates the file offset to specified offset. The offset is updated in the open file table.

sfs_close(int fd): This function closes the opened file referenced by the file descriptor fd. It deletes the file entry from the open file table. It also invalidates the offset. Next time when the file is opened the file system offset is reset to 0.

[β] File system API calls are prefixed with sfs in order to segregate them as file system specific calls.

sfs_fileunlink(char *path): This function unlinks the file specified by ***path*** by removing its entry in its parent directory, zeroing all the data blocks it is pointing to, de-allocating them, clearing the inode data , de-allocating the inode and updating the corresponding bitmaps and superblock.

sfs_dirread(char *path, (void *) buf, int size): This function reads size number of entries from the directory referenced by path and copies them into the buffer. Each directory entry consists of the name and the inodeno for a particular file/directory. If the number of entries in the directory is more than the buffer size, then the function fails.

sfs_dirunlink(char *path): This function unlinks the directory specified by ***path*** by removing its entry in its parent directory, clearing the inode data , de-allocating the inode and updating the corresponding bitmaps and superblock. It however unlinks only empty directories.

File System calls to the generated file system should however be directed via the standard system library. Thus these methods are required to be registered with the VFS. This user level file system however requires some kernel level medium to do so. FUSE (File System in User Space) facilitates doing so. Next section briefs about basic concept of FUSE, its architecture, use and functioning. It also discusses the file-system and FUSE interface details.

4 FUSE (File System in User Space)

File System in User Space (FUSE) is a framework that aids development of file systems in user space. It registers the file system with VFS and redirects standard library file system calls to file system specific API. FUSE was developed for Linux, now has been extended for BSD and Mac OSX. It consists of two primary modules and one utility. A kernel module binds to VFS and redirects calls to the user level fuse library. The fuse library has user space file system methods registered with it. The mount utility allows mounting of the file system at specified mount point. Since, the file system is in user space, a specifying mount point like in fstab without root privileges is not possible. The mount utility is installed with root uid at install time so as to allow mounting of user space file system with user privileges. Care is taken that user cannot access any other information with any super user privilege. Many file systems are built over fuse base. mcachefs, Logic File Systems, Afuse , MountIo etc are some examples of them. The architectural features of FUSE are as in **Fig.5.**

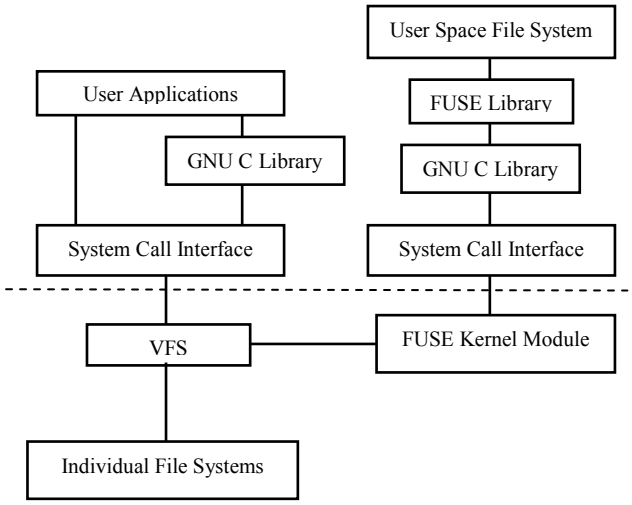

Fig.5 FUSE Architecture Details

The user application invoked file system calls get mapped to standard library calls. They further get translated to VFS calls. VFS layer then invokes the registered method for the corresponding file system. In case of the fuse based implementation, the calls from the VFS get transferred to the fuse kernel module. The kernel module in turn unblocks the blocking read call from the fuse library. The fuse library invokes the appropriate file system specific the kernel module. Data and control flows from VFS to kernel and data moves from kernel to VFS. The interface details are seen in **Fig.6**. The figure illustrates the complete call flow.

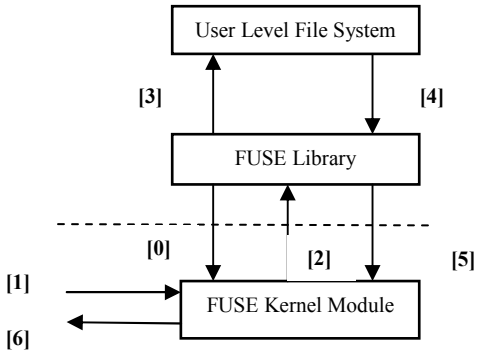

Fig.6 Interface Details & Call Transfer

The FUSE library has a blocked read call on the kernel module (path **[0]**). Suppose, some user application invokes a file system call on the user implemented file system, the VFS sends a request to the fuse kernel module via path **[1]**. The request is

then kept on the waiting queue. When kernel is ready to take up that request it unblocks the path **[0]** and forwards the request via a fuse_read call to the library via path **[2]**. The FUSE library further forwards this request to the user level file system by invoking the appropriate registered method via path **[3]**. The file system method returns the requested data via path **[4]** to the FUSE library. FUSE library then passes data to the kernel module through fuse_write call through path **[5]**. The fuse kernel forwards the request reply to VFS via path **[6].** All the calls are blocking calls hence queues like pending queues are required.

The kernel module is registered as a character device with VFS and the file system is of block access type. It is evident that handling calls is simple but there are multiple switches between user and kernel mode and hence the overload of switching. This however is true in any type of file system implemented at user level. It however gives flexibility by allowing the user to make file systems (or specify them in case of automatic file systems hence allowing generation). No kernel code re-compilation is required. It is also simple since the coding is in user space which is easier than in kernel space. The cost is overhead as discussed earlier.

5 Implementation Details

The file system described in section 3 has been developed. Its methods are registered with FUSE library and thus can be invoked by standard library calls redirected via the fuse kernel module. In systems, the classical solution has been always "redirection", here implemented via FUSE. The file system is compiled with dependent FUSE library and thus compacted as an installable package. Currently, the persistent storage is a file, which is created at install time in the installation directory on existing file system. The mount point can be specified at execution time. Raw device can also serve for persistent storage. The file system can be made tunable by accepting user specifications, computing the structure requirements and generating the limits header file dynamically. This would make the installable file system behave like a automatic file system compiler and give us flexibility benefit. This feature is not implemented so far. Future work will revolve around that area. The framework defined can also be changed for implementing other types of file systems. Performance of file systems would depend on their framework and their medium for persistent storage.

6 Performance Estimation

With the present implementation described above, the medium of persistent storage is a file. During file system initialization, the whole file will be read into memory and will be accessed from memory. Periodically the data is synced to the persistent storage file. This will ensure that data is not lost. Also, since the whole data is in memory, any block can be accessed with uniform access time (minute changes depending on the search time in cache etc) thus the performance will be good. Refer **Fig 7** and **8** for the performance measured in terms of access time for different API calls. It can be seen that Disk_Init (FS_Boot) and Disk_Sync (FS_Sync) complete in the order of msecs where actual disk access takes place, whereas other file system calls complete in orders of microseconds as file system in memory is accessed.

However, when the persistent storage medium changes to raw disk device, the performance will be greatly affected. Also, the framework is pretty naïve (chosen for simplicity) and does not take into account any type of locality. It does not take positional delay of disk into account and no sort of grouping is done. Neither spatial locality is exploited by keeping related files together like FFS **[3]** nor is temporal locality taken into consideration by buffering writes written around the same time and then writing them together on disk. If files are written completely then, better read and write performance can be achieved since adjacent disk blocks will be used. Further, it can be observed that the block size is only 512 bytes which is pretty unimpressive compared to current league of file systems. File system limits are also modest. Filename and pathname limits of 16 and 256 characters respectively seem too low. Maximum file limit of 100 plus their size limited to 15KB is certainly unusable with the amount of workload in current systems. Apart from that there are no checksums or redundancy implemented which may limit the file system reconstruction capability on a disk crash. Currently the file system is not suitable for sensitive data storage.

7 Future Work

Future work revolves around three aspects: changing the framework to a more robust and technology aware, implementing it on a raw device and generating the limits dynamically from user specifications. The new framework will revolve around exploiting the spatial locality or provide for shadowing and providing some kind of redundancy for security. Raw device implementation will give a

better estimate for performance and consideration of user specifications will make it change from a installable file system to a automatic file system compiler. This will not only give the user freedom to have the features of his choice and will still continue to give high performance combined with the factors mentioned above.

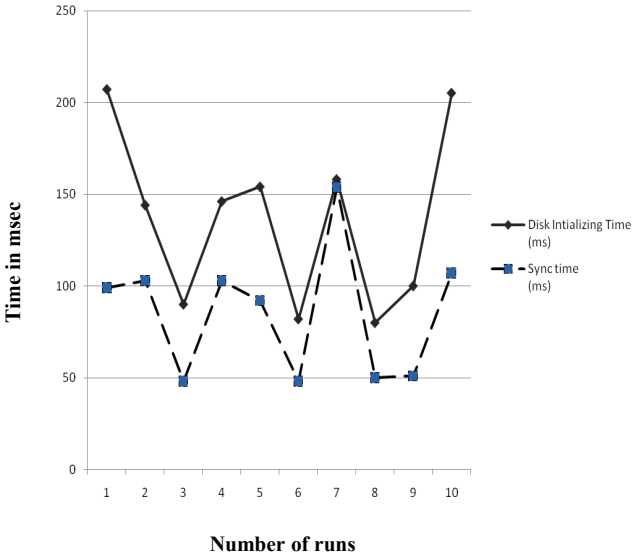

Fig. 7 Disk Initialize and Disk Sync time varying across different runs. (Disk positioning time effect)

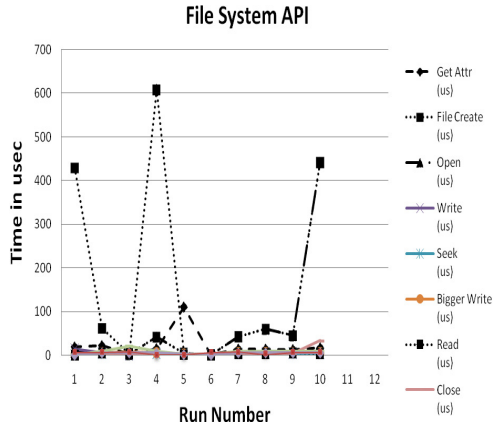

Fig.8 Performance of simple file system estimated in terms of completion time for various file system calls.

8 Software Engineering Aspects

The timeline of the project is shown in **Fig.9.** The initial phase of the project involved background reading. Later development process started. An estimate of available and required resources was made. Development environment was setup. This was however a recursive step till all requirements were satisfied. Primary environment was a Pentium IV machine with Centos 5.1 server. Due to software compatibility problems, later moved to a virtual machine with Fedora Code 8 Linux guest OS. It was followed by requirement analysis. Project development follows the waterfall model (**Fig 10**). The first phase involves requirement analysis followed by planning. Later phase is of designing followed by implementation. To ensure functionality later testing and debugging is done. The code has been commented and coding standards have been followed. Naming conventions have also been followed.

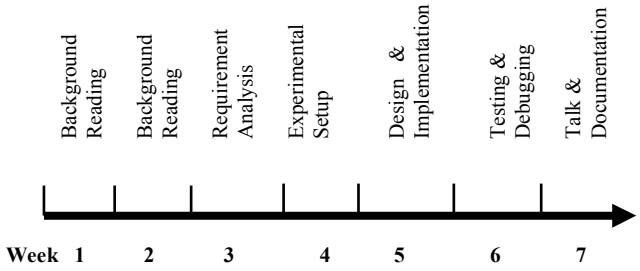

Fig.9 Project Timeline

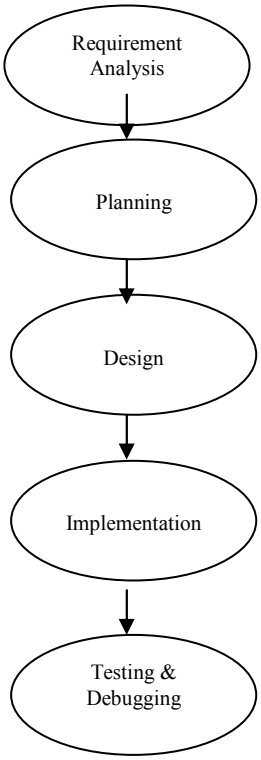

Fig.10 Waterfall Model

Summary

This project is a small step towards automating file systems that till date are very rigid, inflexible and complex. This implementation gives a glimpse of flexibility and customization that the users/experts can expect in the coming years from Systems domain.

Conclusion

Making file systems is a long, complex and tedious task. Stabilizing it is another challenge. Automating it further is altogether a new ballgame. This project gives a valuable insight into the complexities involved in file system design. Automatic file system compilers will give users and system administrators an efficient infrastructure for optimum resource utilization, flexibility, portability and performance.

References

[1] Demsky,B.,et al, "Automatic Detection and Repairs of Errors in Data Structures", OOPSLA 2003 , pp. 87-95.

[2] Zadok,E., et al, "FiST: A Language for Stackable File Systems", ATEC 2000, pp.5-5

[3] Joy,W.N., et al, " A Fast File System for Linux", TOCS 1984 , pp.181-197.

[4] Rosenblum, M., et al, "The design and Implementation of Log-Structured File System", TOCS 1992, pp.26-52.