VFS over the Years: An Efficient Change Log and System Call for Kernel Developers

Bubai Das

Assistant Professor, Department of Computer Science, J. K. College, Purulia

ABSTRACT

"*The Linux Kernel is getting bloated and huge, it's a problem. Sometimes it's a bit sad that we are definitely not the streamlined, small, hyper-efficient kernel that I envisioned 15 years ago ... The kernel is huge and bloated*"

– Linus Torvalds, Linuxcon2009 **(**Roundtable - The Linux Kernel: Straight From the Source)**.** [1]

The Linux Kernel is getting large day by day, both in terms of lines of code and complexity of features and functions. This includes rapid changes in naming conventions of kernel functions, variables and even the flow of data within the code. Moreover, these changes are not well supported by adequate documentation. These issues account to incremental difficulties faced by developers of system code across the globe. This paper is a study of internal working and changes of virtual File System specific code of Linux kernel versions 2.4, 2.6. and 3.0. We have tried to introduce various changes and new addition of several features of the Linux kernel. This survey regarding virtual file system among various kernel versions may play an important role for system developers during system development to gather information about file system in any specific or several kernel versions.

KEY WORDS: File System, Virtual File system (VFS), *Superblock object, File object, Inode* object, dcache, System calls with a path name argument, File descriptor argument and I/O operation, Namespace.

1. Introduction

A *file system* is the methods, hierarchical structure and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. Also it is the way of storing information on a computer that usually consists of a hierarchy of directories that is used to organize files. The file system is also used to refer to a partition or disk that is used to store the files and another use that is using the extended file system, meaning the type of the file system. Ext file system is the first file system which was implemented in April 1992 for created specifically Linux kernel[10]. The Extended File System or ext has metadata structure inspired by the traditional Unix File System. This file system can handle up to 2 gigabytes in size. It was the first implementation that used the virtual file system. There are other members in the extended file system family Ext2(second extended file system), Ext3(third extended file system) and Ext4(fourth extended file system). This paper is organized as below: The first section compares among various kernel versions on the basis of extended file system, next section specified about virtual file system model based on super block operation, file object, node object and dentry operation. The next section focuses on system calls implemented in the virtual file system layer. Lastly, the operations of namespaces and mounting are consider as well.

2. DIFFERENCE BETWEEN V 2.4, 2.6 AND 3.0 OF THE LINUX KERNEL ON THE BASIS OF EXTENDED FILE SYSTEM

2.1 Ext2

Second Extended File System (Ext2) is used by v2.4 Linux kernel.Ext2 was introduced or developed for overcome some limitation of original extended file system which is used earlier Linux kernel version .Ex2 file system is more useful for multiple user environment operating system to store the large number of block and also provide various kind of feature in operating system [17].

2.2 Ext3

Third Extended File System (Ext3) is used by v2.6 Linux kernel. It play importance role in the file system at compared to Ex2 file system because several features are provided by this Extended file system, some of this not present in earlier file system (Ext2).

2.3 Ext4

Fourth Extended File System (Ext4) is used by v3.0 Linux kernel. Various features are introduced in Ext4 like multiple block allocation, delayed allocation, journal checksum, fast fsck, etc. This is the latest extended file system now days. This Extended file system (Ext4) is also capable to store the large number of block and it is too much useful for large volume of operating system [17].

2.4Table1: comparison among ext2 ext3 and ext4 File System in various Linux kernel versions (v2.4, v2.6, v3.0).

Please do NOT insert biographies into your paper. Grant information and other acknowledgements may be placed in the “Acknowledgement(s)” section (see next page).

Table 1

Feature	Ext2	Ext3	Ext4
1.implemented Linux kernel	V2.4	V2.6	V3.0
version: 2. Journaling feature (Journal, Ordered, Writeback).	X	✓	Provide facility to “ON” or“OF”
3. Maximum individual file size	16 GB to 2 TB	16 GB to 2 TB	16 GB to 16 TB
4. Overall	2 TB to 32 TB	2 TB to 32 TB	1 EB. 1 EB = 1024 PB. 1 PB

maximum file system size			= 1024TB.
5. The max number of sublevel-directories	31998	32,000	64,000
	low	average	High
6.Availability 7.Integrity	Data Integrity low	Data Integrity high	Data Integrity high
8.Speed	Faster through put compare to ext.	Faster through put compare to ext2.	Faster through put compare to ext3.
9.Transition among Ext2,Ex3,Ex4.	X	✓	✓

3. THE VIRTUAL FILE SYSTEM (VFS)

The virtual file system (also known as virtual file system switch or VFS) is kernel software layer that handles all system call related to a standard Unix file system. It's main strength is providing a common interface to several kind of file systems, that is, the VFS is an abstraction layer between the application programs and the file system implementations. Data flow in between several layer of operating system has been defined below. In large volume of operating system, only layer of virtual file system is being considered and subsequent part in this paper has been described code label enhancement and changes among various kernel versions also describe newly added and changes various system calls implemented through virtual file system.

3.1 THE VFS FILE MODEL

3.1.1 Superblock Object

- Stores information concerning a mounted file system.
- Holds things like device, blocksize, dirty flags, list of dirty inodes etc.
- Super operations like read/write/delete/clear inode etc.
- Gives pointer to the ***root inode*** of this FS
- Superblock manipulators: mount/umount

3.1.1.1 struct super_operations:

This describes how the VFS can manipulate the superblock of the file system. As of kernel 2.6.22 and also 3.0, the following members are newly defined with existing members:(2.4 and 2.6 version are same).

1. Struct inode *(*alloc_inode)(struct super_block *sb):
Description:- Alloc_inode:-This method is called by inode_alloc() to allocate memory for struct inode and initialize it.If this function is not defined, a simple 'struct inode' is allocated. Normally alloc_inode will be used to allocate a larger structure which contains a 'struct inode' embedded within it[3][6].

2. **void (*destroy_inode)(struct inode *);**
Description:- **destroy_inode**:- This method is called by destroy_inode() to release resources allocated for struct inode. It is only required if ->alloc_inode was defined and simply undoes anything done by ->alloc_inode[11].

3. **Void (*dirty_inode) (struct inode *, int flags);**
Description :- **dirty_inode:** This method is called by the VFS to mark an inode dirty.

4. **void (*drop_inode) (struct inode *);**
Description :- **drop_inode:** called when the last access to the inode is dropped, with the inode->i_lock spinlock held[12].

5**. int (*sync_fs)(struct super_block *sb, int wait);**
Description :- **sync_fs**: called when VFS is writing out all dirty data associated with a superblock. The second parameter indicates whether the method should wait until the write out has been completed. Optional[3][6].

6. **int (*freeze_fs) (struct super_block *);**
Description: - **freeze_fs:** called when VFS is locking a file system and forcing it into a consistent state. This method is currently used by the Logical Volume Manager (LVM**)[3][6].**

7. int (*unfreeze_fs) (struct super_block *);
Description :- **unfreeze_fs:** called when VFS is unlocking a filesystem and making it writable again[11].

8. **ssize_t (*quota_read)(struct super_block *, int, char *,size_t, loff_t);**
Description :- **quota_read**: called by the VFS to read from filesystem quota file.

9. **ssize_t(*quota_write)(struct super_block *, int, const char *, size_t, loff_t);**
Description :- **quota_write:** called by the VFS to write to filesystem quota file[12].

3.1.2 File object

- Stores information about the interaction between an open file and a process.
- File pointer points to the current position in the file from which the next operation will take place.

3.1.2.1 struct file_operations

This describes how the VFS can manipulate an open file. As of kernel 2.6.22 and also 3.0, the following members are newly defined with existing members: (2.4 and2.6 version are same)

1. ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
Description :- **aio_read:** called by io_submit(2) and other asynchronous I/O operations

2. ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
Description :- **aio_write**: called by io_submit(2) and other asynchronous I/O operations[11].

3. long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
Description :- **unlocked_ioctl:** called by the ioctl(2) system call[8].

4. long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
Description:- **compat_ioctl:** called by the ioctl(2) system call when 32 bit system calls are used on 64 bit kernels[3][6][5].

5. int (*aio_fsync) (struct kiocb *, int datasync);
Description:- **fsync:** called by the fsync(2) system call

6. ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
Description :- **sendfile:** called by the sendfile(2) system call

7. int (*check_flags)(int);
Description :- **check_flags:** called by the fcntl(2) system call for F_SETFL command[11].

8. dir_notify(file,arg)int (*flock) (struct file *, int, struct file_lock *);
Description:- **flock:** called by the flock(2) system call

9. ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, size_t, unsigned int);
Description :- **splice_write:** called by the VFS to splice data from a pipe to a file. This method is used by the splice(2) system call[3][6].

10. ssize_t(*splice_read)(struct file *, struct pipe_inode_info *, size_t, unsigned int);
Description :- **splice_read:** called by the VFS to splice data from file to a pipe. This method is used by the splice(2) system call

3.1.3 Inode Object

- stores general information about a specific file.
- Linux keeps a cache of active and recently used inodes**.**
- All inodes within a file system are accessed by file-name.
- Linux's VFS layer maintains a cache of currently active and recently used names, called *dcache*

3.1.3.1 struct inode_operation

This describes how the VFS can manipulate an inode in your file system. As of kernel 2.6.22 and also 3.0, the following members are newly defined with existing members: (same as 2.4 and 2.6 version).

1. **void (*put_link) (struct dentry *, struct nameidata *, void *)**

Description:- put_link: called by the VFS to release resources allocated by follow_link(). The cookie returned by follow_link() is passed to this method as the last parameter. [3][5][6].

2. **int (*check_acl)(struct inode *, int, unsigned int);**

3. **int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);**

Description :- **setxattr:** called by the VFS to set an extended attribute for a file. Extended attribute is a name:value pair associated with an inode. This method is called by setxattr(2) system call[9][11].

4. **ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);**

Description :- **getxattr:** called by the VFS to retrieve the value of an extended attribute name. This method is called by getxattr(2) function call.

5. **ssize_t (*listxattr) (struct dentry *, char *, size_t);**

Description:- **listxattr:** called by the VFS to list all extended attributes for a given file. This method is called by listxattr(2) system call.

6. **int (*removexattr) (struct dentry *, const char *);**

Description:- **removexattr:** called by the VFS to remove an extended attribute from a file. This method is called by removexattr(2) system call[3][6].

7. **void (*truncate_range)(struct inode *, loff_t, loff_t);**

Description :- truncate_range: a method provided by the underlying filesystem to truncate a range of blocks , i.e. punch a hole somewhere in a file.

3.1.4 Dcache

- structured in memory as a tree.
- each entry or node in tree (*dentry)* points to an inode.
- it is not a complete copy of a file tree

3.1.4.1 struct dentry_operations

This structure describes how a file system can overload the standard dentry operations. Dentries and the dcache are the domain of the VFS and the individual file system implementations. Device drivers have no business here. These methods may be set to NULL, as they are either optional or the VFS uses a default. As of kernel 2.6.22 and 3.0, the following members are newly added with previous members :(2.4 and 2.6 version are also same).

1. **char *(*d_dname)(struct dentry *, char *, int);**

Description:- Useful for some pseudo filesystems (sockfs, pipefs, ...) to delay pathname generation. (Instead of doing it when dentry is created, it's done only when the path is needed.). Real files ystems probably do not want to use it, because their dentries are present in global dcache hash, so their hash should be an invariant. As no lock is held, d_dname() should not try to modify the dentry itself, unless appropriate SMP safety is used[3][6].

2. struct_vfsmount*(*d_automount)(struct path *);

Description :- This should create a new VFS mount record and return the record to the caller.The caller is supplied with a path parameter giving the automount directory to describe the automount target and the parent VFS mount record to provide inheritable mount parameters. NULL should be returned if someone else managed to make the automount first. If the vfsmount creation failed, then an error code should be returned[3][6].

3. int (*d_manage)(struct dentry *, bool);

Description :- This allows autofs, for example, to hold up clients waiting to explore behind a 'mountpoint' whilst letting the daemon go past and construct the subtree there. 0 should be returned to let the calling process continue[3][5][9].

4. SYSTEM CALLS' IMPLEMENTATION THROUGH VFS

This section represents newly added and changes of existing several system calls implemented by VFS among the kernel version 2.4, 2.6 and 3.0 with synopsis and description.

4.1 System Calls With A Path Name Argumen

1. Name : **setxattr, lsetxattr, fsetxattr** - set an extended attribute value.

Synopsis : #include <sys/types.h>
#include <attr/xattr.h>

int setxattr(const char *path, const char *name, const void *value, size_t size, int flags);
int lsetxattr(const char *path, const char *name, const void *value, size_t size, int flags);
int fsetxattr(int fd, const char *name, const void *value, size_t size, int flags)

Description: **setxattr():** sets the value of the extended attribute identified by name and associated with the given path in the file system. The size of the value must be specified.
lsetxattr(): is identical to setxattr(), except in the case of a symbolic link, where the extended attribute is set on the link itself, not the file that it refers to.
fsetxattr(): is identical to setxattr(), only the extended attribute is set on the open file referred to by fd (as returned by open(2)) in place of path[3][9][11].

2. Name : **removexattr, lremovexattr, fremovexattr** - remove an extended attribute

Synopsis : #include <sys/types.h>
#include <attr/xattr.h>

int removexattr(const char *path, const char *name);
int lremovexattr(const char *path, const char *name);
int fremovexattr(int fd, const char *name)

Description:- **removexattr():** removes the extended attribute identified by name and associated with the given path in the file system. lremovexattr() is identical to removexattr(), except in the case of a symbolic link, where the extended attribute is removed from the link itself, not the file that it refers to. fremovexattr() is identical to removexattr(), only the extended attribute is removed from the open file referred to by fd (as returned by open(2)) in place of path[3][9].

3. Name : **listxattr, llistxattr, flistxattr** - list extended attribute names

Synopsis : #include <sys/types.h>
#include <attr/xattr.h>
ssize_t listxattr(const char *path, char *list, size_t size);
ssize_t llistxattr(const char *path, char *list, size_t size);
ssize_t flistxattr(int fd, char *list, size_t size);
Description:- **listxattr():** retrieves the list of extended attribute names associated with the given path in the file system. The list is the set of (null-terminated) names, one after the other.
llistxattr(): is identical to listxattr(), except in the case of a symbolic link, where the list of names of extended attributes associated with the link itself is retrieved, not the file that it refers to.
flistxattr(): is identical to listxattr(), only the open file referred to by fd (as returned by open(2)) is interrogated in place of path[3][9][18].

4. Name : **rename** - change the name or location of a file
Synopsis : #include <stdio.h>
int rename(const char *oldpath, const char *newpath);
Description:-rename(): renames a file, moving it between directories if required. Any other hard links to the file (as created using link(2)) are unaffected. If new path already exists it will be atomically replaced, so that there is no point at which another process attempting to access new path will find it missing. If new path exists but the operation fails for some reason rename() guarantees to leave an instance of new path in place[2][9].

5. Name : **lookup_dcookie** - return a directory entry's path.
Synopsis : **int lookup_dcookie(u64 cookie, char *buffer, size_t len);**
Description :- Look up the full path of the directory entry specified by the value cookie.The cookie is an opaque identifier uniquely identifying a particular directory entry. The buffer given is filled in with the full path of the directory entry. For lookup_dcookie() to return successfully, the kernel must still hold a cookie reference to the directory entry[3][9].

6. Name : **oldfstat, oldlstat, oldstat**, - obsolete system calls
synopsis : Obsolete system calls.
Description : The Linux 2.0 kernel implements these calls to support old executables. These calls return structures which have grown since their first implementation, but old executables must continue to receive old smaller structures. Current executables should be linked with current libraries and never use these calls[9].

7. Name: **fstat64, lstat64, stat64** -- get file status
Synopsis : **int fstat64(int fildes, struct stat64 *buf);**
int lstat64(const char *restrict path, struct stat64 *restrict buf);
int stat64(const char *restrict path, struct stat64 *restrict buf);
Description: The stat() function obtains information about the file pointed to by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be search able. The lstat() function is like stat() except in the case where the named file is

a symbolic link; lstat() returns information about the link, while stat() returns information about the file the link references[3][9][11].

8. Name :**getxattr, lgetxattr, fgetxattr** - retrieve an extended attribute value
Synopsis : #include <sys/types.h>
#include <attr/xattr.h>
ssize_t getxattr(const char *path, const char *name, void *value, size_t size);
ssize_t lgetxattr(const char *path, const char *name, void *value, size_t size);
ssize_t fgetxattr(int fd, const char *name, void *value, size_t size);
Description:- **getxattr()** retrieves the *value* of the extended attribute identified by *name* and associated with the given *path* in the file system. The length of the attribute *value* is returned. lgetxattr() is identical to getxattr(), except in the case of a symbolic link, where the link itself is interrogated, not the file that it refers to.fgetxattr() is identical to getxattr(), only the open file referred to by *fd* (as returned byopen(2)) is interrogated in place of p*ath*[3][9].

9. Name : **getdents** - get directory entries
Synopsis:**int getdents(unsigned int fd, struct linux_dirent *Dirp,Unsigned Int Count);**
Description:- The system call getdents() reads several linux_dirent structures from the directory referred to by the open file descriptor fd into the buffer pointed to by dirp. The argument count specifies the size of that buffer[3][9][18].

10. Name : **umount2** - unmount file system
Synopsis : #include <sys/mount.h>
int umount2(const char *target, int flags);
Description : umount() and umount2() remove the attachment of the (topmost) file system mounted on target. MNT_EXPIRE (since Linux 2.6.8), Mark the mount point as expired. If a mount point is not currently in use, then an initial call to umount2() with this flag fails with the error EAGAIN, but marks the mount point as expired. The mount point remains expired as long as it isn't accessed by any process. A second umount2() call specifying MNT_EXPIRE unmounts an expired mount point. This flag cannot be specified with either MNT_FORCE or MNT_DETACH. UMOUNT_NOFOLLOW (since Linux 2.6.34) Don't dereference target if it is a symbolic link. This flag allows security problems to be avoided in set-user-ID-root programs that allow unprivileged users to unmount file systems[3][9].

4.2 SYSTEM CALLS WITH FILE DESCRIPTOR ARGUMENT

1. Name **: statfs, fstatfs** - get file system statistics
Synopsis : #include <sys/vfs.h> /* or <sys/statfs.h> */
int statfs(const char *path, struct statfs *buf);
int fstatfs(int fd, struct statfs *buf);
Description : The function statfs() returns information about a mounted file system. path is the pathname of any file within the mounted file system. buf is a pointer to a statfs structure.
fstatfs() returns the same information about an open file referenced by descriptor fd[3][9].

2. name : fact - manipulate file descriptor

Synopsis : #include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */);
Description : fcntl() performs one of the operations described below on the open file descriptor fd. The operation is determined by cmd[3][18].

3. Name : truncate, ftruncate - truncate a file to a specified length
Synopsis : #include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
Description : The truncate() and ftruncate() functions cause the regular file named by *path* or referenced by *fd* to be truncated to a size of precisely *length* bytes. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0')[9][18].

4. Name : sendfile - transfer data between file descriptors
Synopsis : #include <sys/sendfile.h>
ssize_t sendfile(int *out_fd*, int *in_fd*, off_t * offset ", size_t" " count");
Description : sendfile() copies data between one file descriptor and another. Because this copying is done within the kernel, sendfile() is more efficient than the combination ofread*(2)* and write*(2)*, which would require transferring data to and from user space.*in_fd* should be a file descriptor opened for reading and *out_fd* should be a descriptor opened for writing. If *offset* is not NULL, then it points to a variable holding the file offset from which sendfile() will start reading data from *in_fd[9]*.

5. Name : mmap2 - map files or devices into memory
Synopsis : #include <sys/mman.h>
void *mmap2(void ****addr*, size_t** ***length*****, int** ***prot*****, int*****flags*****, int** ***fd*****, off_t** ***pgoffset*****);**
Description : This is probably not the system call you are interested; instead, see ,mmap(2) which describes the glibc wrapper function that invokes thissystem call.The **mmap2**() system call provides the same interface as mmap(2), except thatthe final argument specifies the offset into the file in 4096-byte units(instead of bytes, as is done by mmap(2)).

6.Name: remap_file_pages - create a nonlinear file mapping
Synopsis :#include<sys/mman.h>
intremap_file_pages(void**addr***,size_t*****size*****,int*****prot*****,ssize_t*****pgoff*****,int*****flags*****);**
Description : The **remap_file_pages**() system call is used to create a nonlinear mapping, that is, a mapping in which the pages of the file are mapped into a non sequential order in memory. The advantage of using **remap_file_pages**() over using repeated calls to mmap(2) is that the former approach does not require the kernel to create additional VMA (Virtual Memory Area) data structures[11].

7.Name: pread, pwrite - read from or write to a file descriptor at a given offset

Synopsis #include <unistd.h>
ssize_t pread(int *fd*, void **buf*, size_t *count*, off_t *offset*);
ssize_t pwrite(int *fd*, const void **buf*, size_t *count*, off_t *offset*);
Description : pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*. The file offset is not changed. pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset*. The file offset is not changed. The file referenced by *fd* must be capable of seeking. On Linux, the underlying system calls were renamed in kernel 2.6: pread() became pread64(), and pwrite() became pwrite64(). The system call numbers remained the same. The glibc pread() and pwrite() wrapper functions transparently deal with the change[9].

8.Name : setns - reassociate thread with a namespace
Synopsis : #include <sched.h>
int setns(int *fd*, int *nstype*);
Description : Given a file descriptor referring to a namespace, reassociate the calling thread with that namespace. The *fd* argument is a file descriptor referring to one of the namespace entries in a */proc/[pid]/ns/* directory[8].

9.Name : dup, dup2, dup3 - duplicate a file descriptor
Synopsis: #include<unistd.h>
intdup(int*oldfd*);
intdup2(int*oldfd*,int*newfd*);
#include<unistd.h>
intdup3(int*oldfd*,int*newfd*,int*flags*);
Description : These system calls create a copy of the file descriptor *oldfd*. **dup**() uses the lowest-numbered unused descriptor for the new descriptor. **dup2**() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary. **dup3**() is the same as **dup2**(), except that: The caller can force the close-on-exec flag to be set for the new file descriptor by specifying **O_CLOEXEC** in *flags[9][11]*.

4.3 SYSTEM CALLS WITH I/O OPERATION

1. Name : io_setup - create an asynchronous I/O context
Synopsis : #include<libaio.h>
intio_setup(unsigned*nr_events*,aio_context_t**ctxp*);
Description : io_setup() creates an asynchronous I/O context capable of receiving at least*nr_events*. *Ctxp* must not point to an AIO context that already exists, and must be initialized to 0 prior to the call. On successful creation of the AIOcontext, **ctxp* is filled in with the resulting handle[3][11].

2. Name : io_submit - submit asynchronous I/O blocks for processing
Synopsis : #include<libaio.h>
intio_submit(aio_context_t*ctx_id*,long*nr*,structiocb*iocbpp*)**
Description:io_submit() queues *nr* I/O request blocks for processing in the AIO context *ctx_id*. *iocbpp* should be an array of *nr* AIO control blocks, which will be submitted to context *ctx_id[8]*.

3. Name : io_getevents - read asynchronous I/O events from the completion queu**e**
Synopsis :#include<linux/time.h>
#include<libaio.h>
intio_getevents(aio_context_t*ctx_id*,long*min_nr*,long*nr*,
structio_event**events***,structtimespec****timeout***);**
Description : io_getevents() attempts to read at least *min_nr* events and up to *nr* events from the completion queue of the AIO context specified by *ctx_id*[3].

4. Name : io_cancel - cancel an outstanding asynchronous I/O operation
Synopsis : #include<libaio.h>
intio_cancel(aio_context_t*ctx_id*,structiocb**iocb***, structio_event****result***);**
Description :io_cancel() attempts to cancel an asynchronous I/O operation previously submitted with io_submit(2)[9].

5. Name : io_destroy - destroy an asynchronous I/O context
Synopsis : #include<libaio.h>
intio_destroy(aio_context_t*ctx*);
Description :io_destroy() removes the asynchronous I/O context from the list of I/O contexts and then destroys it. io_destroy() can also cancel any outstanding asynchronous I/O actions on *ctx* and block on completion[11].

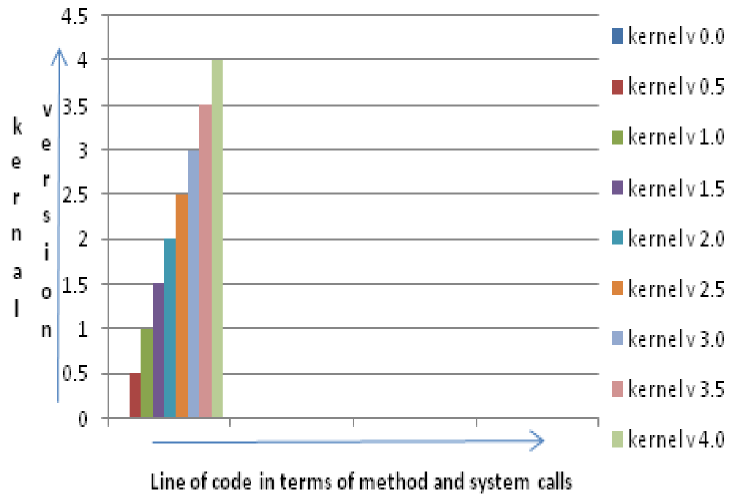
4.4 Table2: System call implementation in VFS has been summarized in a tabular representation among the various kernel versions (V2.4, V2.6, V3.0)

Operation type	**System call name**	**2.4V**	**2.6V**	**3.0V**
Path name argument	1.Setxattr,lsetxattr,fsetxattr	x	✓	✓
	2.removexattr,lremovexattr, fremovexattr.	x	✓	✓
	3.rename	✓	✓	✓
	4.lookup_dcookie	x	✓	✓

	5.fstate64,lstate64,state64	fsate, lstate ,state	✓	✓
	6.getxattr,fgetxattr,lgetxattr	x	✓	✓
	7.getdents	✓	✓	✓
	8.unmount2	unmount	✓	✓
File Descriptor argument	1.statfs ,fstatfs	✓	✓	✓
	2.fact	x	✓	✓
	3.truncate	✓	✓	✓
	4.sendfile	✓	✓	✓
	5.mmap2	mmap	✓	✓
	6.remap_file_pages	x	✓	✓
	7.pread,pwrite	✓	pread64 ,pwrite 64	pread64, pwrite64
	8.setns	x	x	✓
	9.dup, dup2,dup3	Dup, dup2	Dup,dup2	✓
System calls with I/O	1.io_setup	x	✓	✓
	2.io_submit	x	✓	✓
	3.io_getevents	x	x	✓
	4.io_cancel	x	x	✓

5. Enrichment of Linux kernel with time has been implemented in a graphical representation.

Linux kernel is The Linux Kernel is getting bloated and huge with time by system developer. Line of code is increasing to implement various method and system calls among various versions.



6. ABOUT NAMESPACE

Linux , as of about 2.6.29, has solid support for a kernel feature called namespaces. Basically, a namespace makes it possible for a process and all its descendants to have their own private view of what is normally a globally shared resource in the kernel. Some of the more interesting resources that can be namespace-ified include[12][15]:

- the network stack (ports, sockets, interfaces)
- processes and pids
- the mount table

6.1 VFS File System Mounting with namespace

In a traditional Unix system, there is only one tree of mounted file systems: starting from the system's root file system, each process can potentially access every file in a mounted file system by specifying the proper pathname. In this respect, Linux 2.6.29 introduced a new concept that, every process might have its own tree of mounted file systems the so-called namespace of the process version 3.0 also have the same things[15].Two basic operations must be performed before making use of a file system, registration and mounting. Registration is done either when the system boots or when the module implementing the file system is being loaded. Once a file system has been registered, its specific functions are available to the kernel, so that kind of file system can be mounted on the system's directory tree. Each file system has its own *root directory*. The file system whose root directory is the root of the system's directory tree is called *root file system*. Other file systems can be mounted on the system's directory tree: the directories on which they are inserted are called *mount points*[8].

- A *bind mount* allows any file or directory to be accessible from any other location.
- *File system namespaces* are completely separate file system trees associated with different processes.

A process requests a copy of its current file system tree at clone(2) , after which the new process has an identical copy of the original process's file system tree. After the copy is made, any mount action in either copy of the tree is not reflected in the other copy. While per-process file system namespaces were very useful in theory, in practice the complete isolation between them was too restrictive. Usually most processes share the same namespace, which is the tree of mounted file systems that is rooted at the system's root file system and that is used by the init process[3]. However, a process gets a new namespace if it is created by the clone( ) system call with the CLONE_NEWNS flag set (see the section . The new namespace is then inherited by children processes if the parent creates them without the CLONE_NEWNS flag. When a process mounts or unmounts a file system, it only modifies its namespace. Therefore, the change is visible to all processes that share the same namespace, and only to them. A process can even change the root file system of its namespace by using the Linux-specific pivot_root( ) system call[3][13].The namespace of a process is represented by a namespace structure pointed to by the namespace field of the process descriptor. The fields of the namespace structure are follow

- atomic_t count

Usage counter (how many processes share the namespace)

- struct vfsmount * root

Mounted files ystem descriptor for the root directory of the namespace

- struct list_head list

Head of list of all mounted file system descriptors

- struct rw_semaphore sem

Read/write semaphore protecting this structure

7. CONCLUSION

This paper introduces newly added concepts and changes regarding virtual file system and system calls implemented in the virtual file system among various versions. This paper aims to fill some of the gap between operating system algorithms and practical coding principles and implementation of Linux kernel, across various versions. Also, it is worth mentioning that there are no well-defined versioning control systems with change logs for open source distributions like the Linux kernel. This paper tried to cover up all areas of file system specific code, mainly focused on virtual file system, as given below:

- super block, file object, inode and dentry: newly added structures and functions
- system call (path name related, file descriptor and I/O operation)

This paper may serve as a well-documented change log across different kernel versions, namely v2.4, 2.6 and v3.0. This research work may be helpful to researchers and system developers including security analysts and auditors, as a single point of reference.

8. REFERENCES

1. http://streaming.linux-magazin.de/en/archive-linuxcon09.htm
2. Daniel P. Bovet and Marco Cesati. Understanding Linux Kernel. O'Reilly Media, Inc., 2nd edition edition, December 2002. This book covers kernel 2.4.
3. Daniel P. Bovet and Marco Cesati. Understanding Linux Kernel. O'Reilly Media, Inc., 3rd edition edition, November 2005. This book covers kernel 2.6.
4. Linux Kernel 2.4 : http://lxr.free-electrons.com/source/Documentation/filesystems/vfs.txt?v=2.4
5. Linux Kernel 2.6:http://lxr.free-electrons.com/source/Documentation/filesystems/vfs.txt?v=2.6
6. Linux Kernel 3.0: http://lxr.free-electrons.com/source/Documentation/filesystems/vfs.txt?v=3.0
7. Jaroslav Šoltýs Linux Kernel 2.6 Documentation Master thesis Thesis advisor: RNDr. Jaroslav Janácek Bratislava

8. Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best,"File-Access Characteristics of Parallel Scientific Workloads," IEEE Transactions on Parallel and Distributed Systems, 7(10):1075--1089, October 1996.
9. Linux man page
10. Linux kernel From Wikipedia, the free encyclopediahttp://en.wikipedia.org/wiki/Linux_kernel
11. http://lxr.linux.no
12. Linux Kernel Newbies: http://kernelnewbies.org
13. Applying mount namespaces http://www.ibm.com/developerworks/linux/library/l-mount-namespaces/index.html
14. PID namespaces in the 2.6.24 kernel http://lwn.net/Articles/259217/
15. Network namespaceshttp://lwn.net/Articles/219794/
16. Linux Namespace at Arista https://eos.aristanetworks.com/2011/06/linux-namespaces-at-arista/
17. Linux File Systems: Ext2 vs Ext3 vs Ext4**http://www.thegeekstuff.com/2011/05/ext2-ext3-ext4/**
18. Linux Cross Reference Free Electrone Embedded Linux experts http://lxr.free-electrons.com
19. Index of/pub/linux/kernel https://www.kernel.org/pub/linux/kernel
20. Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor, "Parallel Access to Files in the Vesta File System," Proceedings of Supercomputing '93, pages 472--481, Portland, OR, 1993. IEEE Computer Society Press.
21. http://www.linuxjournal.com/article
22. http://www.maenad.net/geek/di8k-debian/node29.html