Adaptation of the System V/386 Filesystem for Linux

By

PAUL B. MONDAY

A report submitted to accompany a project to fulfill
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science
December 1993

ADAPTATION OF THE SYSTEM V/386 FILESYSTEM FOR LINUX

Abstract by Paul B. Monday, M.S.
Washington State University
October 1993

Chair: K.C. Wang

Compatibility between operating systems and filesystems is an essential item when creating a robust operating system. The Linux operating system is taking the filesystem compatibility issue to a new level with its modular integration of filesystems into the Linux kernel. The project which accompanies this paper exploits the robust Linux filesystem to integrate System V/386 filesystem compatibility into Linux kernel. This paper will discuss issues relative to the integration of the System V/386 filesystem support.

Contents

1 Introduction

Operating systems which are in the marketplace and highly *commercial* usually include 1 or 2 filesystems to choose from to store and retrieve data. Creating efficient filesystems and making the structure proprietary is often a selling point for one operating system, while the action stifles coexistance with other operating systems. The Linux filesystem attempts to remedy this through simple, but highly successful, coding tricks which turns the filesystem into a modular block of code. Filesystems are treated as a set of high level and low level functions. Functions can be added and removed whenever the kernel is rebuilt. Creating new modules (filesystems) is simplified also if the programmer can understand a simple concept of containment which the Linux filesystem capitalizes on. A case study of how the System V/386 filesystem is built, followed by how it is integrated into the Linux kernel is examined here.

2 System V/386 Release 4.0 Filesystem

The key to adding a filesystem to Linux is differentiating between low level data handling, and the high level function which is duplicated in other filesystems. The System V/386 filesystem is much like other *Unix-like* filesystem, so there were many case studies already coded into Linux. A ground up approach was taken when adding the System V/386 filesystem.
First, the data was examined to determine the layout of a System V/386 filesystem. This consisted of a high-level overview of a diskette after a *mkfs* is completed. Next, finer grained examination of the data structures used on a filesystem is completed. Once both of these

tasks are done, algorithms can be varified on System V/386, then an examination of how to integrate the new algorithms into Linux must be completed.

I will document the most general cases of a System V/386 filesystem, as I have not intentionally set out to make special cases of *mkfs* work when coding the accompanying project (a user of System V/386 can radically change the structure of a filesystem with options on the *mkfs* command).

Throughout the coding of the System V/386 filesystem I used a reverse engineering approach. The implementation of the filesystem will be described in this way also. First a high level overview will be presented, and I will then work my way into the specific block data structures and organizations.

2.1 General Information

To understand how the filesystem algorithms manipulate the System V/386 filesystem, one must first understand how a disk is organized by System V/386. Below, I have listed the important aspects of a System V/386 filesystem. First, the *Superblock* is offset into the diskette by 512 bytes. This offset allows room for a bootblock and initialization. The inodes follow several blocks after the superblock. This gap allows for System V/386 to keep a list of bad blocks. I have not implemented this feature in the filesystem support.

The low-level design of System V/386 differs from many other flavors of Unix, particularly Minix, due to the fact that there are no bit maps or zone maps. Rather than keeping a bit aside for each data block and inode to indicate whether or not it is free, System V/386 keeps a linked list of free blocks and zeroes out the unused inodes. DOS is similiar to Minix in the way the FAT (File Allocation Table) is organized, see the Appendix for details of the Minix and DOS filesystem layouts.

The order of the filesystem is bootblock, superblock, bad block mapping, inodes, and data blocks. The layout is as follows (a sector is 512k, or 0.5 * block size).

- Sector 1: Bootblock
- Sector 2: Superblock
- Sector 3: Bad block mapping
- Sector 4 to x: Bad block mapping continued
- Sector x + 1 (1k aligned): Inodes
- Sector y: Continuation of inodes
- Sector y + 1 (1k aligned): Data Blocks
- Sector z: All sectors to end of disk are data blocks

2.2 Superblock

The System V/386 superblock is a 512 byte sector which holds information which is constantly changing. This differs from other operating systems in that the superblock must be repeatedly written to disk as blocks and inodes are allocated and deallocated.

The primary structure of the superblock is layed out in the list below, the offset is in terms of bytes (8 bits). Some structures which exist in the System V/386 superblock will not be used in the Linux implementation. These will not have corresponding field names. For the ones that I do use, field names will immediately follow the offset and will be in parenthesis. This makes it easier to reference the code which is included in an Appendix. An (*) indicates that the field is later abstracted out to the Linux Superblock.

- offset 0(isize): Number of blocks in inode list
- offset 2(fsize): Number of blocks in the volume
- offset 6(nfree): Number of addresses in free cache
- offset 8(free): Free block cache
- offset 208(ninode): Number of inodes in inode cache
- offset 210(inode): Free inode cache
- offset 410: Lock bit (set during block cache manipulation) (*)
- offset 411: Lock bit (set during inode cache manipulation) (*)
- offset 412: Super block modified flag (set when dirty) (*)
- offset 413: Read only flag (*)
- offset 414(time): Time of last super block modification (*)
- offset 418: Mounted device information (*)
- offset 426(tfree): Total free blocks on volume

- offset 430(tinode): Total free inodes on volume
- offset 432: File system name
- offset 438: File system pack name
- offset 444: Adjust this to make the size of filesystem
- offset 492: State the filesystem is in (*)
- offset 496: Filesystems magic number (0xfd187e20)
- offset 500: Type of new filesystem

The reason many fields are not used is simply because of duplication in the Linux operating system. The lock fields and modification fields are also contained in the main Linux superblock. Since the Linux superblock then contains a pointer to the System V/386 superblock (this will be described later), the System V/386 fields go unused.

Total free blocks and total free inodes must be tediously kept track of. The filesystem updates these fields in memory with each allocation of a block or an inode. The changes to the caches are not written to disk immediately, as this would put an unnecessary burden on resources and misuse one of the features of many unix-like filesystems, disk caching. The changes are written to disk upon a umount or when either the block or the inode cache is refilled or flushed. The block and inode caches in the filesystem drive much of the logic behind the algorithms used to maintain the superblock.

2.2.1 Free Block Cache

The free block cache is documented very well in [Ba86]. When the filesystem is made, the free blocks are organized into a linked structure. The System V/386 filesystem stores 50 addresses in the block cache. The last address read when blocks are being allocated is a block number which contains the next 50 addresses which are to be loaded into the cache. With this implemenatation, to reload the cache, the operating system loads the block pointed to in the cache, then transfers the addresses which are stored there into the superblock's cache. Although the initial overhead to build a filesystem around this idea may be slightly higher than a bit mapped method, the cache system is a very straightforward method of organizing the data. The algorithm for allocating blocks in the System V/386 filesystem follows, assume that a block is requested from the filesystem for an unknown reason.

```
• If tfree=0 then return FAIL
• If nfree=1 and tfree¡¿1 THEN
    – address=free[0]
    – Read block at address
    – free[0] to free[49] = block[0] to block[49]
• ELSE
    – address = free[nfree]
    – nfree = nfree - 1
• return address
```

The algorithm for freeing blocks in the System V/386 filesystem follows. Assume that the block to be freed resides at *address*.

```
• if nfree=50 THEN
    – read block at address
    – write 50 addresses in free to block
    – nfree = 1
    – free[0] = address
• ELSE
    – free[nfree] = address
    – nfree = nfree + 1
• tfree = tfree + 1
• return
```

There is one problem with this method which can slow down disk access. I have chosen to regularly write the superblock between allocations. A worst case scenario would as the scenario which is written below. This is a case where a few blocks get allocated which force the cache to be reloaded, then a few blocks get freed which forces the cache to be flushed. A scenario is as follows.

1. Initial Configuration of Scenario
    - Block Cache (*free*) contains 1 block (address 24)

   - $\mathit{nfree} = 1$

2. A block is requested by a user

   - $\mathit{nfree} = 1$ so cache must be reloaded

   - Reload cache then return block 24 for user to use

3. A block is returned by a user ($\mathit{address} = 100$) and *nfree* is still equal to 50

   - Superblock cache is full, write 50 addresses to block 100

   - Store the address 100 in the superblock cache and change *nfree* to 1

4. Go to step 2

The scenario above, although not dangerous, requires repeated disk writing for minimal erquests. I have implemented this algorithm merely for the sake of safety and to avoid problems which may occur if the cache is not functioning correctly in the prerelease kernel. [1]

2.2.2 Free Inode List

The free inode list is very similiar to the free block list with a couple of major exceptions. The designers of the System V/386 filesystem assumed that the inodes would not see as

[1] **Note:** After examining several books on the System V/386 filesystem after completing the coding portion of the project, differences in the implementation of writing the superblock to disk were noted. I now believe that it was *unwise* to repeatedly write the superblock to disk. The slowdown this causes can be quite substantial, plus, inode updates may not stay in sync with superblock updates. It would be better to throw out all changes in the case of a critical error, than keeping a list of where data *should* have been placed.

much activity as the data blocks. This assumption freed the designers to create a slightly more time consuming allocation method. The free inodes are not linked together as the data blocks are. When the time comes to reload the inode cache (100 inode addresses are kept in the superblock for a typical filesystem), a linear search is conducted until the cache is full or all inodes are exhausted. [2]

The current implementation and the true implementation differ in two ways. The first is in the use of the *nlink* field in the inode. My code sets all bits contained in an inode to zero. The System V/386 implementation sets the nlink field to zero, this indicates the inode is ready to be used again. The second difference is in the algorithm to free an inode. The differences are displayed in the free inode algorithm below.

- returnInode.nlink = 0 //This is currently a memset(returnInode,0,sizeof(returnInode))
- return //This return does not occur in the true implememtation
- if ninode = 100 then return
- superblock.ninode ++
- superblock.inode[superblock.ninode]=number of returnInode
- return

[2] **Note:** The implementation of the inode cache in the project versus the true implementation in the System V/386 filesystem differ somewhat, as I discovered just recently, see [Sh87].

2.3 Inodes

The internal inode representation, as with any Unix system, is the key to the organization of the filesytem. The System V/386 filesystem follows virtually the same format as any other Unix system, though with some important distinctions. The differences will be pointed out later in the document in the filesystem comparisons section.

Many of the fields in the inode structure which resides in the System V/386 filesystem are similiar or the same as Linux. Most importantly, the Unix filesystems appear to be compatible as far as the mode values go. This takes a level of abstraction away when trying to think in terms of one filesystem or another. For example, if the mode field contains an unsigned short value of 0x4000, this indicates a directory, all of the Unix systems I have run across so far use the same value. Again, an (*) indicates that the field is later abstracted out to the Linux inode.

- offset 0(mode): Mode and type of file (*)
- offset 2(nlink): Number of links to the file (*)
- offset 4(uid): File owner's userid (*)
- offset 6(gid): File's group id (*)
- offset 8(size): Size, in bytes, of the file (*)
- offset 12(addr): Disk block addresses
- offset 51(gen): File's generation number (*)
- offset 52(atime): Last time file accessed (*)

- offset 56(mtime): Last time file modified (*)

- offset 60(ctime): File creation time (*)

One of the most important topics to be addressed when talking about inodes is how the inode keeps track of file contents. The System V/386 filesystem uses a list of 10 direct blocks, followed by an indirect block, a double indirect block, and a triple indirect block. This allows for a maximum filesize of 33,834 blocks, or 34,646,016 bytes. .This creates an interesting problem when placing disk addresses into the inode and All direct addresses are stored as 3 byte values. Throughout the rest of the System V/386 filesystem, and most other filesystems, addresses are stored as long integers, giving four bytes of storage for a disk address. This peculiarity allows more addresses to be stored in the inode, thus a larger filesize. At the same time, the 3 byte addresses could easily be switched to 4 byte addresses, allowing for a smaller file, but a larger disk. As it stands, the largest drive that can be accessed by the inode is 16,777,216 blocks.

2.3.1 Direct Blocks

Figure 1 shows graphically how the block addressing from within the inode works. Direct blocks are controlled by grabbing the addresses from 0-9 (bytes 0-26 in the addr field) and reading the contents of the blocks referenced by each address.

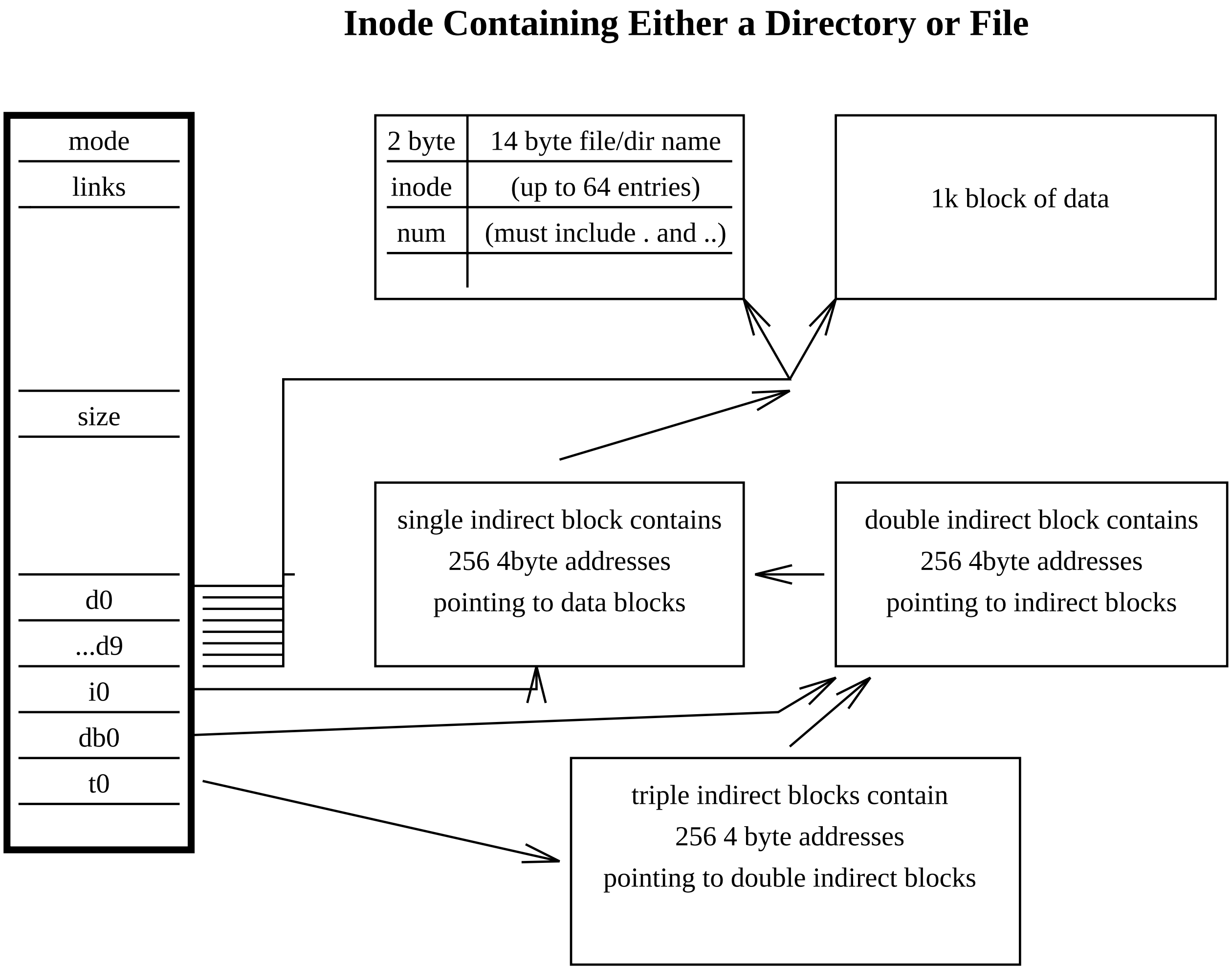


Figure 1: Inode and Data Blocks

2.3.2 Indirect (Single, Double and Triple) Blocks

Figure 1 also displays how the different levels of addressing link to blocks. All the indirect blocks really do is give us an extra 32 blocks to use for file space. The double blocks expand the filesize by 1024 blocks and the triple indirect blocks expand the filesize again by 32,768 blocks. One of the curious things about the System V/386 filesystem is that the addresses are stored as long integers in the indirect blocks. Here several addresses per block are lost to wasted space, since the direct inodes cannot use the 4th byte in each address. It is my belief that this was left to ease the algorithms which access addresses through indirect blocks, plus allow an easier way to expand the maximum file size, since the inodes could be changed relatively easily to allow for 4 byte addresses.

The algorithms follow for reading the various types of blocks.

2.4 Directory Blocks

Directory blocks are similiar to many other Unix systems. The data blocks pointed to by the direct and indirect blocks are filled with records which consist of a inode number and a filename. The record is a reference to an inode, this implies that the inode can consist of any type of Linux file (directory, data, symlink, etc....) Filenames are restricted to 14 characters. Figure 1 also displays how a directory block works with an inode. Note that it is exactly the same as the case that data is stored in the inode, the major difference is in the *mode* field, a different value is stored here.

2.5 System V/386 Filesystem Conclusion

The System V/386 filesystem is quite popular for schools to use due to the availability of the code in it's early life, and the volumes of literature which are written on the system as a whole. For this reason, the first half of the project was quite a bit of research, with some hands on experience. Once the theories were derived from books, it was a simple matter to write test programs to examine diskettes at various points as files are copied to and from the diskettes.

Once the System V/386 portions were understood, the second phase of the project was started. This was a matter of determining how the Linux filesystem used other filesystems, and how the System V/386 filesystem would fit into the Linux system.

3 Linux

Multiple levels of indirection, and robust/dynamic data structures in Linux creates a modular environment to code new filesystems into. At the same time, the initial configuration of Linux is allowed to leave out support that is not necessary for a particular user's needs. Currently, Linux contains support for the Microsoft DOS FAT filesystem, Minix, and several variations of Minix which are called extended filesystems. The structures which make this modular and robust system possible will be discussed here.

The Linux filesystem would take a considerable amount of time to describe in full detail. For the purpose of filesystem integration, only those parts which relate specifically to integrating new filesystem types will be described. Many other items, like how the Linux cache handles the filesystem, are left out since these are irrelevant to handling new filesystems.

3.1 General Information

The main Linux filesystem revolves around data buffers which remain in memory as long as a device is mounted. The structures which are kept in memory contain both the Linux version of a structure, and the original structure read in from the non-Linux filesystem.

The resulting *Linux-System V/386* filesystem is really a hybrid version of the System V/386 filesystem, customized to work alongside Linux' buffer implementations. It is best to look at Linux' specific structures to see exactly how this customization works.

I will start with the assumption that the user has typed in the correct mount command and Linux is passing control to the System V/386 specific routines. Only minor changes to the Linux kernel go into doing this, they will be glossed over in the section "Configuration of Linux to Include/Exclude Filesystems".

3.2 Superblock Handling

The function containing the setup of the Linux superblock is called sysv_read_super. Each filesystem will have a corresponding function. The purpose is to grab the superblock off the physical disk, and set up the Linux superblock which will reside in memory. The following list contains the fields in the Linux filesystem's superblock which are important to the project.

1. offset 0(dev): Device superblock is located on

2. offset 2(blocksize): Blocksize of blocks on device

3. offset 6(lock): Lock bit set when superblock in use

4. offset 7(rdonly): Read only bit set for read only filesystem

5. offset 8(dirt): Dirty bit set when superblock changed

6. offset 9(superop): Pointer to structure containing valid operations for the mounted filesystem

7. offset 13(flags): Various flags set at mount time (non-fs dependent) such as read-only, no-dev, no-suid, etc...
8. offset 17(magic): Magic number for filesystem
9. offset 23(time): Time filesystem was mounted
10. offset 27(covered): Pointer to inode of filesystem which was written over
11. offset 31(mounted): Pointer to root directory inode
12. offset 35(wait): Pointer to wait queue for superblock operations
13. offset 39(u): Union containing structures for subsets of superblocks of any other mountable filesystems

As can be seen from the superblock structure, it is essentially a container for other superblock structures and inodes. Locking mechanisms and various filesystem independant structures are added, but pointers to the superblock operations and filesystem dependant structures are left to be filled in at the time that the filesystem is mounted.

The union for the System V/386 specific portion contains most of the original superblock information. Unlike the inode (as seen later), it is necessary to retain a virtual copy of all superblock information for later reference.

3.2.1 Reading the Superblock

The superblock to be used is passed as a pointer to the read_super routine. First, a lock is put on the superblock so that the cache can be manipulated freely. I have been lax on locking up inodes and superblocks throughout the rest of the code. If race conditions occur, the modules should be examined to determine where to place further locks on the files and buffers.

1. The read_super does a direct read to the disk, grabs the zero block and places it into a buffer for manipulation.

2. Set a pointer to the System V/386 superblock to the correct location in the buffer (since the superblock starts at offset 512)

3. Check the magic number to make sure that the user did not err in calling the disk a System V/386 disk.

4. Read the root inode from the disk and verify that it really does exist. The root inode is then stored in mounted, in the superblock.

5. Copy the general System V/386 superblock info into the Linux superblock (time, uids, etc...)

6. Copy the free caches over into the Linux version of the System V/386 superblock.

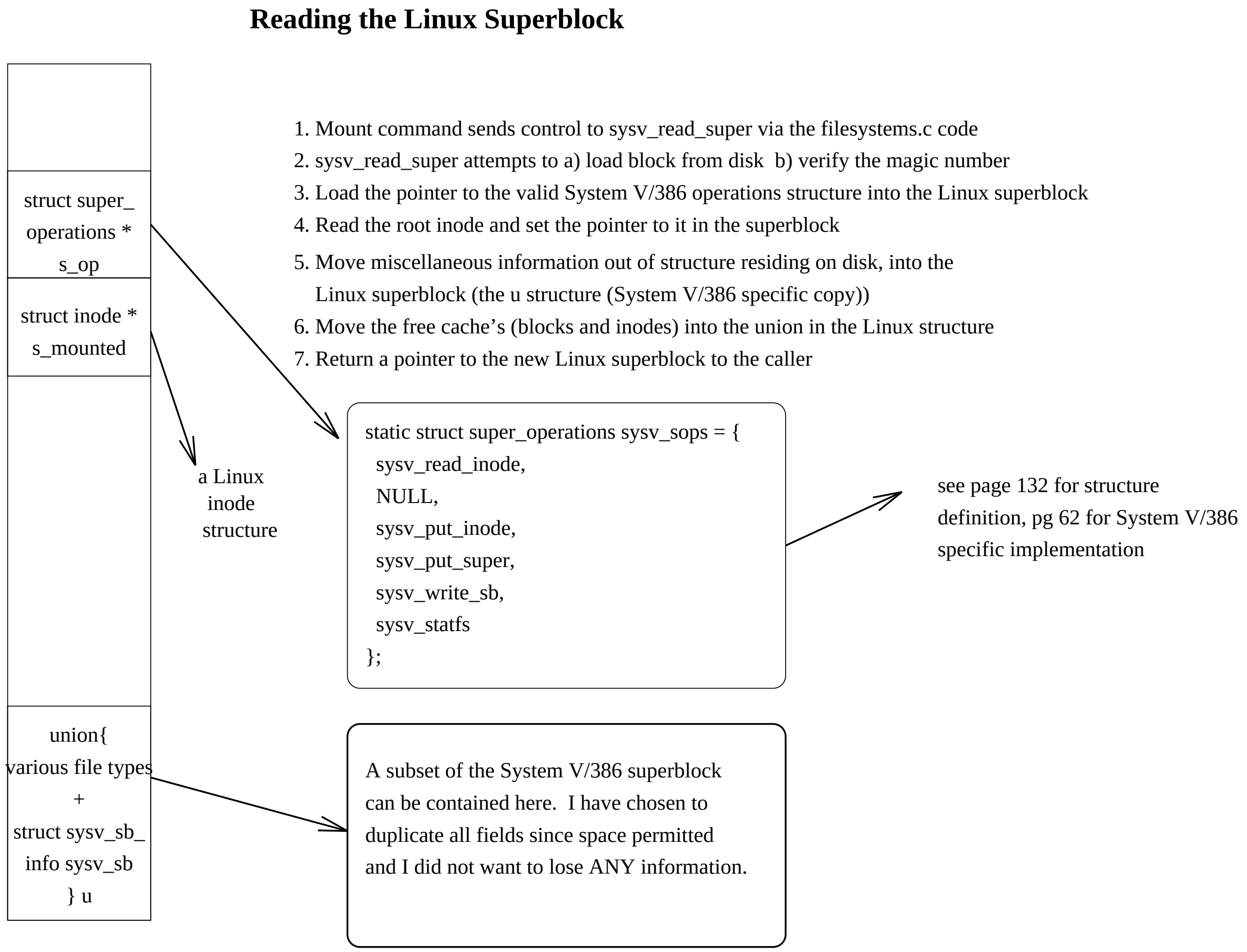
Upon return from the read_super routine, the Linux superblock contains the following:

- All fields specific to the Linux portion of the superblock were updated correctly. The fields include blocksize, magic number for the fs, and pointers to the filesystem specific operations (System V/386 function calls).

- The root inode for the system v filesystem is loaded into the mounted field in the superblock.

- The System V/386 specific portion of the Linux superblock was filled in with all necessary information taken from the superblock which resides on disk.

One major problem I cannot solve or find a reason for is where the initial location of the superblock is. All literature and code indicates that the superblock is always contained in block 0, offset 512 of a system v/386 filesystem. In practice I find it there only when using 5 1/4" diskettes formatted at 1.2 Meg. I have updated the code to search for the superblock before using it.



Reading the Linux Superblock

1. Mount command sends control to sysv_read_super via the filesystems.c code
2. sysv_read_super attempts to a) load block from disk b) verify the magic number
3. Load the pointer to the valid System V/386 operations structure into the Linux superblock
4. Read the root inode and set the pointer to it in the superblock
5. Move miscellaneous information out of structure residing on disk, into the Linux superblock (the u structure (System V/386 specific copy))
6. Move the free cache's (blocks and inodes) into the union in the Linux structure
7. Return a pointer to the new Linux superblock to the caller

```
static struct super_operations sysv_sops = {
  sysv_read_inode,
  NULL,
  sysv_put_inode,
  sysv_put_super,
  sysv_write_sb,
  sysv_statfs
};
```

see page 132 for structure definition, pg 62 for System V/386 specific implementation

A subset of the System V/386 superblock can be contained here. I have chosen to duplicate all fields since space permitted and I did not want to lose ANY information.

Figure 2: Reading the Superblock

3.2.2 Writing the Superblock

Unlike many varieties of Unix filesystems, the System V/386 filesystem needs to write the superblock periodically to disk. This operations is achieved by the sysv_write_sb routine. It is virtually the reverse of reading the superblock, though not as many verifications are done. The important part of this operation is that the variable data gets placed back onto the disk, primarily the cache specific fields of the superblock. The write_sb routine follows the following order.

1. Read the zero block of the superblock's device (pointed to by dev in the Linux superblock) into a buffer.
2. Find offset 512
3. Copy the time into the buffer
4. Update the cache fields in the buffer (free, inode, nfree, ninode)
5. Update the total free inodes and total free blocks fields
6. Mark the buffer as dirty
7. Release the buffer (this should implicitly write the buffer back onto the disk)
8. Change the superblock's dirty bit back to 0 to indicate it was written.

Primarily, the Linux implementation of the System V/386 filesystem completes the above operations when major cache fills are done, and when a filesystem is unmounted.

3.3 Inode Handling

Inode handling is very similiar to the superblock handling routines. This section will describe how a System V/386 inode is read into memory, then a bit about where and how subsequent changes are handled. The list below displays many of the fields in a Linux inode. Most of the fields are not listed though. A large number of fields only used for handling in memory. The bulk of the space allocated for a Linux inode contains fields and pointers for caching inodes. The list below does contain fields relevant to the filesystem. Again, like the superblock, the inode acts as a container for filesystem specific behavior, with the 'meta-inode' containing information which will be common to all inodes.

1. (dev): Device inode is mounted on
2. (ino): Number of inode on device
3. (mode): Mode of inode loaded
4. (link): Number of links to the inode
5. (uid): User ID of inode
6. (gid): Group ID of inode
7. (rdev): Device if inode refers to another inode on another device
8. (atime): Time inode was last accessed
9. (mtime): Time inode was last modified
10. (ctime): Time inode was created
11. (op): Pointer to valid operations for this particular inode
12. (lock): Bit set indicates inode locked
13. (mount): Bit set indicates inode mounted

14. (u): Union containing various possible inode structures which are dependent on the filesystem type the inode is associated with.

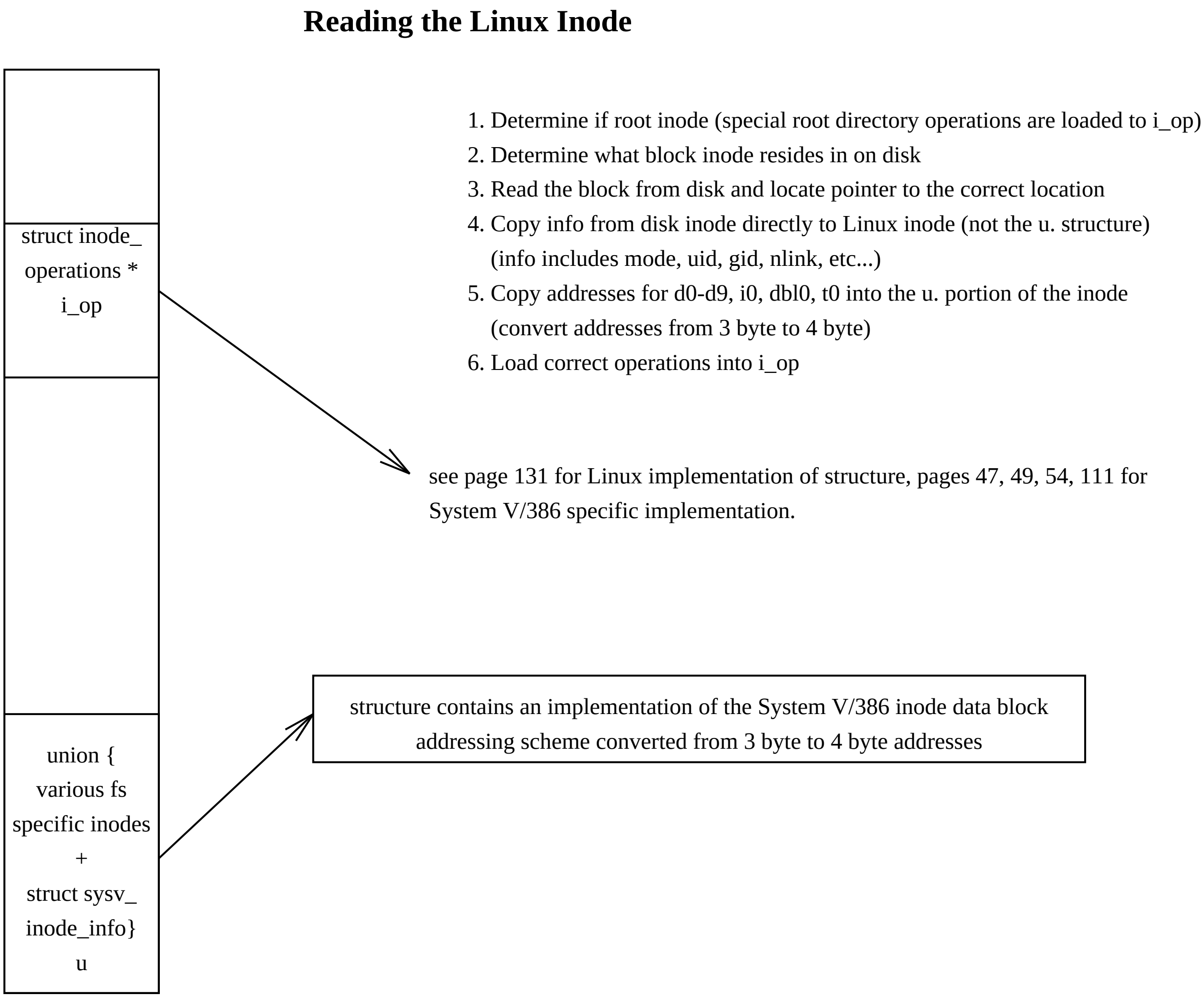
The union for the System V/386 filesystem contains the direct/indirect block mappings. All other information is contained in the Linux specific portion of the inode. The System V/386 direct/indirect block mappings is further altered when stored in the Linux inode. The addresses are converted from their native 3 byte addresses to an easier to handle four byte equivalent.

3.3.1 Reading an Inode

An inode is read in the sysv_read_inode routine. The routine receives a copy of the Linux inode when it is called, the number of the inode to be read is stored in the ino field of the inode structure. The following is done to grab the inode off the disk and return successfully.

1. Store the inode number
2. Zero out pointers to inode functions in the op field
3. Set the mode to 0
4. Check if reading the root inode, if so, set the op field to the operations which can be done on a root inode. Also set the mode to 777 and return to the caller.
5. Figure out the block that the inode is in if it's not the root inode
6. Read in the block, return if block can't be read
7. Move the inode structure into the correct portion of the buffer returned by the read

8. Set easy fields in the Linux inode such as time, mode, uid, etc....
9. Check what type of inode we have retrieved and set the inode op field appropriately (could have a directory, link, fifo, block char device, etc...)
10. Convert the block pointer addresses to 4 byte addresses and store in the Linux inode.
11. Return to the caller.


Reading the Linux Inode

1. Determine if root inode (special root directory operations are loaded to i_op)
2. Determine what block inode resides in on disk
3. Read the block from disk and locate pointer to the correct location
4. Copy info from disk inode directly to Linux inode (not the u. structure) (info includes mode, uid, gid, nlink, etc...)
5. Copy addresses for d0-d9, i0, dbl0, t0 into the u. portion of the inode (convert addresses from 3 byte to 4 byte)
6. Load correct operations into i_op

see page 131 for Linux implementation of structure, pages 47, 49, 54, 111 for System V/386 specific implementation.

structure contains an implementation of the System V/386 inode data block addressing scheme converted from 3 byte to 4 byte addresses

struct inode_ operations * i_op
union { various fs specific inodes + struct sysv_ inode_info} u

Figure 3: Reading an Inode

3.3.2 Writing an Inode

The primary purpose of the write_inode routine for System V/386 filesystem is to update the data block pointers if they have changed. The routine is simply a reverse of the read_systemv routine.

3.4 Coding for Filesystem Specific Routines

3.4.1 High Level Procedures

Most of the high level routine (mkdir, rename, etc...) were copied from other filesystems with slight modifications to allow for System V/386 versions of structures. The most complex of the high level routines to complete was the truncate functions. These are used to ensure that files are allocated to the data pointers within an inode correctly.

3.4.2 Low Level Disk i/o Coding

The low-level disk i/o is the only portion of the filesystem which must be built from scratch. It was best to maintain the same named functions for consistency with the rest of the filesystem, though much of the internals are changed. Direct manipulation of a System V/386 superblock occurs in the lowlev.c module. This module is where new inodes are allocated, new blocks are allocated, and inodes and blocks are freed. It is critical in the allocate/deallocate routines to keep the superblock which exists out on disk up to date. The algorithms follow directly from the description of the System V/386 filesystem in the first half of the paper.

While coding the lowlevel routines, the major thing to keep in mind is that the parameters passed in are Linux structures. The code uses a combination of both the Linux and the System V/386 structures to obtain the required results in low level disk i/o.

3.5 Configuration of Linux to Include/Exclude Filesystems

Changes to the Linux kernel are minimal to get a new filesystem up and running. Changes must be done to the following files, corresponding changes are documented with the list.

- /linux/fs/filesystems.c - The primary structure (file_systems) is contained in the fs.h file. This structure is used to send control upon a mount to the correct read_super routine. If a filesystem is not listed here, it will not be able to be mounted.
- /linux/include/linux/fs.h - Changes must be made here to include pointers to system v/386 data structures from the superblock and the inodes. These structures are maintained in memory and are accessed from most of the system v/386 filesystem routines.
- /linux/fs/Makefile - This must be changed to include the fs/sysv directory so that the code will be compiled.
- /linux/config.h - This is the script to configure the Linux system before compiling the kernel. It should be adjusted to allow the inclusion/exclusion of the system v/386 filesystem.

3.6 Conclusion (Linux System V/386 Implementation)

The superblock and inode in the Linux operating system are the key to the robust Linux filesystem. Abstraction of common data structures in filesystems, and subsequent containment of filesystem type dependent data structures allow for customization of Linux to use similiar, but not duplicate, filesystems.

There are benefits and drawbacks to this design. The major benefit of the design is easy inclusion/exclusion of filesystems in the kernel. A major drawback to the design is that each modular filesystem requeires unique code for all data access/handling routines, even if the functions are exact duplicates of each other.

Coding new filesystems for Linux is a relatively simple task. The largest part of coding is first understaninding how the new filesystem works. Once this is done, the Linux filesystem modifications are simple, primarily due to the abstraction Linux does. The native structures in the new filesystem are left intact so there is only a small learning curve to fit a filesystem into Linux.

It is also clear that this process is not for a simple user to appreciate. Adding and removing filesystems from a user's perspective would be extremely difficult. The option would be to include all of the filesystems which, in turn, creates a larger runtime kernel. From this perspective, Linux poses a problem if it were to ever enter a commercial market. From the perspective of a programmer and/or student, Linux makes a great case study to use to demonstrate filesystem implementations.

4 Appendix A - The DOS filesystem

1. Logical Sector 0 - Breakdown follows

   - 00h - 8086 Jump Instruction
   - 03h - OEM name & version
   - 0Bh - Bytes per sector

- 0Dh - Sectors per allocation unit
- 0Eh - Reserved sectors
- 10h - Number of FATs
- 11h - Number of root-directory entries
- 13h - Total sectors in logical volume
- 15h - Media descriptor byte
- 16h - Number of sectors per FAT
- 18h - Sectors per track
- 1Ah - Number of heads
- 1Ch - Number of hidden sectors
- 1Eh - Bootstrap routine

2. File Allocation Table (FAT) #1
3. Possible additional copies of FAT
4. Root disk directory
5. Files area (to the end of the disk)

The disk organization is built from the above structures, file accesses revolve around the FAT. Before the overview of the FAT, one must understand how MSDOS allocates space. Rather than going by blocks and sectors, MSDOS uses an *allocation unit*, also called a *cluster*. How many sectors per cluster on a disk is determined by the drive type. Sectors per cluster are determined in terms of powers of 2.

- Single Sided floppy 1 sector/cluster
- Double Sided floppy 2 sectors/cluster
- PC/AT type fixed disk 4 sectors/cluster
- PC/XT type fixed disk 8 sectors/cluster

Notice that serious fragmentations problems can occur on some types of fixed disks, in fact up to (512*8-1) bytes can be lost per file which is created. Contrast this with 1023 bytes lost for a typical Unix filesystem, assuming it was formatted with default values. The FAT keeps track of clusters and is simply a set of 12 bit or 16 bit hex numbers. Twelve bits are kept if there are under 4087 clusters, 16 bits are kept for over 4087 clusters. Each FAT table appears as follows.

- Byte 1 - Media Descriptor Byte
- Bytes 2-4 - (0)0FFh
- Bytes 4-end of FAT
  - (0)000h indicates the cluster is available
  - (F)FF0h - (F)FF6h indicates the cluster is reserved
  - (F)FF7h indicates the cluster is bad
  - (F)FF8h - (F)FFFh indicates the last cluster in a file
  - (x)xxxh indicates the next cluster in a series

The appearance of the FAT is similiar to many flavors of Unix, but whereas the bitmaps which will be described in the Minix system simply decide if a block is free or not, the FAT contains much information about how a file is assembled, plus bad block criteria.

The FAT makes file corruption checking very simple. Multiple copies of the FAT can be kept up to date. Occasionally, the copies of the FAT can be checked against each other to verify that a copy of the FAT has not been corrupted. Of course this is not in any way perfect as many other problems can occur, but it does keep the central data structure relatively intact.

In addition to the FAT, each volume in MSDOS contains a root directory structure. This is somewhat similiar to the idea behind the root inode, except that the root directory is not handled in the same manner as other subdirectories. Subdirectories other than the root appear as files with special attribute bytes, similiar to inodes. Only the root directory warrants special handling in DOS. This may be an offshoot of Versions 1 and 2 of DOS where only one directory was allowed, thus storage space was allotted at the beginning. All directory structures, root or subdirectories, have the same datastructure controlling it. This data structure does not allow for enough information to be kept for modern PC users, but has sufficed for a long time.

- 00h - 07h: Filename
- 08h - 0Ah: Extension
- 0Bh : Attribute
  - 0 = Read-only
  - 1 = Hidden

  - 2 = System
  - 3 = Volume label
  - 4 = Subdirectory
  - 5 = Archive bit
  - 6 = Reserved
  - 7 = Reserved

- 0Ch - 15h: Reserved
- 16h - 17h: Time created or last updated
- 18h - 10h: Date created or last updated
- 1Ah - 1Bh: Starting cluster
- 1Ch - 1Fh: File size

It is important to notice many of the problems that a directory entry such as this leaves out. There is no place to record an owner, or different security levels other than read only and hidden. In fact, there is NO way to maintain any type of security for a multi-user system. This is one of the major drawbacks of the MSDOS system. Any network file security must be implemented at a high level by application software.

In addition to security problems, the limited number of attributes limits what types of files are possible. It is not possible to symbolically link files or do many types of operations which Unix users take for granted. Also, even though subdirectories appear similiar to files, there are only three primitive functions which can be performed on an MSDOS subdirectory, CREATE, DELETE, and SELECT.

5 Appendix B - Minix

The minix operating system solves many problems which the System V/386 filesystem has, but produces problems and inefficiencies of its own. The Linux operating system is loosely based upon this, but will be described in better detail in the *Linux* section.

When looking at the layout of the Minix disk, there are 6 major parts that need to be understood.

- 0h - 1023h: Boot block
- 1024h - 2047h: Super block
- 2048h - ?????: I-node bitmaps
- end of inode bitmaps - ????: Zone bit maps
- end of zone bitmaps - end of disk: Data blocks

The inode and zone bitmap areas are simply bits with a 1-1 correspodence to inodes or data blocks. If the bit is a zero, the inode or block is not allocated, otherwise it is. This makes searches for free zones fairly simplistic and only a small amount of data must be searched to find free space. In the best case this is a linear search through the bits. The number of blocks which are dedicated to inode and zone bitmaps is variable, since the number of inodes which can be allocated is variable.

The super block is much larger than the System V/386 super block, but contains slightly more info, due to the need for pointers to where the bitmaps start and end. One can easily determine an algorithm for allocating free blocks to a file. The inodes reside in the first part of the data blocks. A typical inode is very similiar to the System V/386 inode, with slightly less information being kept as to update times.

A typical inode for Minix allows a 14 character filename. It must be kept in mind that there are extensions and variations of the Minix filesystem which allow for longer filenames, and various other changes.

References

[Ba86] Bach, Maurice *The Design of the Unix Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

[At89] *UNIX System V/386 Release 3.2.2 Operations//System Administration Guide*

[Du86] Duncan, Ray *Advanced MSDOS*, Microsoft Press, Redmond, WA, 1986.

[Du89] Duncan, Ray *Design Goals and Implementation of the New High Performance File System*, *Microsoft System Journal*, Microsoft Press, Redmond, WA, September, 1989.

[Sh87] Shaw, Myril Clement *Unix Internals: A Systems Operations Handbook*, Tab Books, Blue Ridge Summit, PA, 1987.

[Ta87] Tanenbaum, Andrew *Operating Systems: Design and Implementation* , Prentice-Hall International Editions, Englewood Cliffs, NJ, 1987.