

ABSTRACT

Dissertation Title: Design and Implementation of Journaling File System for HURD Operating System

Name of student: N. SivaramaKrishnan

Supervisor: Mr. K. B. Rajashekar

Semester: Fourth

ID No: 2002HZ12334

This dissertation presents design and implementation ideas for a new journaling file system, *Siva File System (SFS)* for the GNU HURD operating system. The HURD is a collection of servers that run on the Mach micro kernel to implement file system, network protocols, file access control and other features implemented by the UNIX kernel or similar kernels (such as LINUX). File systems provide a mechanism for successfully storing and retrieving data on a computer system. The data structures that define the organization of a file system must be correct when a file system is being used.

Journaling file systems keep a record of the changes made to the file system in a separate part of the disk called a *journal* or *log*. If the computer system crashes while making changes to the file system, the operating system can use the information in the log to bring the file system up-to-date by replaying the log, which is usually done when it remounts the journaling file system or verifies its consistency.

The grounding for the project is analyzing the other journaling file systems ext3, ReiserFS, JFS, XFS and providing a generic design and implementation ideas for the new file system *Siva File System (SFS)* for the GNU HURD Operating System.

ACKNOWLEDGEMENT

I take this opportunity to express my profound gratitude and deep regards to my guide and mentors Mr. K.B. Rajashekar and Mr. A.G. Sumesh for their exemplary guidance, monitoring and constant encouragement throughout the course of this Dissertation work.

I would like to thank all my friends and colleagues who helped me directly or indirectly during the course of this work.

Last but not the least I would also like to thank my family for giving me constant support throughout the project.

N. SivaramaKrishnan

LIST OF ABBREVIATIONS USED

SFS	Siva File System
HURD	Hird of Unix-Replacing Daemons
HIRD	Hurd of Interfaces representing Depth
JFS	Journaling File System

LIST OF FIGURES

Figure 1 SFS Architecture.....	10
Figure 2 SFS Layout	12
Figure 3 Data blocks before de-fragmenting (Scenario 1).....	18
Figure 4 Data blocks after de-fragmenting (Scenario 1).....	19
Figure 5 Data blocks before de-fragmenting (Scenario 2).....	19
Figure 6 Data blocks after de-fragmenting (Scenario 2).....	19

Table of contents

1.	Introduction	6
1.1	Overview of GNU Mach	6
1.2	Overview of HURD	6
1.3	Overview of non-journaling file system.....	6
1.4	Overview of journaling file system.....	7
2.	Problem Description.....	9
2.1	Background	9
2.2	Requirements Description	9
3.	Architecture of <i>SFS</i>	10
3.1	Attribute Manager	10
3.2	Space Manager	10
3.3	Log Manager	11
3.4	Volume Manager.....	11
4.	<i>SFS</i> layout	12
5.	Data Structures	13
5.1	Super block header format	13
5.2	Data block header format	13
5.3	Inode format	14
5.4	Journal / log header format.....	14
5.5	Journal / log format	14
6.	Design of Attribute Manager.....	16
7.	Design of Space Manager	17
7.1	Space Manager	17
7.2	De-fragmenter	18
8.	Design of Log Manager.....	20
9.	File system formatting.....	22
10.	Implemented HURD interfaces.....	23
11.	Limitations / Future Enhancements.....	25
12.	Discussion	26
13.	Conclusion.....	27
14.	References	28

1. Introduction

1.1 Overview of GNU Mach

GNU Mach is the micro kernel of the GNU project. It is the base of the operating system and provides its functionality to the Hurd servers and all user applications. The Mach kernel provides abstractions of the underlying hardware resources like devices and memory. Mach also provides interfaces for inter-process communication.

1.2 Overview of HURD

The fundamental purpose of an operating system is to enable a variety of programs to share a single computer efficiently and productively. This demands memory protection, preemptively scheduled timesharing, coordinated access to I/O peripherals, and other services. On today's computer systems, programmers usually implement these goals through a large program called *kernel*. Since this program must be accessible to all user programs, it is the natural place to add functionality to the system. A traditional system allows users to add components to a kernel only if they both understand most of it and have privileged status within the system. Testing the new components becomes a difficult task and cannot be done while others are using the system. Bugs usually cause fatal system crashes, further disrupting others' use of the system

The GNU Hurd, by contrast, is designed to make the area of system code as limited as possible. Programs are required to communicate only with a few essential parts of the kernel and the rest of the system can be replaced dynamically. Users can easily add components to the existing system without disrupting others from using the system. This is accomplished by identifying those system components that users *must* use in order to communicate with each other.

The Hurd uses Mach ports for communicating between users and servers (A Mach port is a communication point on a Mach task where messages are sent and received).

1.3 Overview of non-journaling file system

All computer applications need to store and retrieve information. The file systems exist to store, retrieve and manipulate data. In order to do this, file system maintains an internal data structure so that the data is organized and can be readily accessed. This internal data structure is called *meta-data*. In order for the file system to work properly, the *meta-data* is expected to be in reasonable, consistent, non-corrupted state. The file

system consistency is verified by checking the *meta*-data, every time the file system is mounted. Unfortunately, verifying the correctness of file system *meta*-data actually involves checking a number of different points:

- Each allocation unit (whether it is a block or an extent) belongs only to a single file or directory, or is marked as being used
- No file or directory contains a data block marked as being unused in the file system bitmap
- Each file or directory in the file system is referenced in some other directory in that file system. From a user's point of view, this means that there is a directory path to each file or directory in the file system.
- Each file has only as many parent directories as the reference count in its inode indicates. Although each file exists only in a single physical location on the disk, multiple directories can contain references to the inode that holds information about this file. These references are known as hard links. The file can therefore be accessed through any of these directories, and deleting it from any of these directories decrements the link count. A file is actually deleted only when its link count is 0 — in other words, when any directory no longer references it.

Verifying all these relationships may take a while if it's necessary to manually check each of them. Whether this consistency check is necessary is the fundamental difference between journaling and non-journaling file systems. Eliminating this sort of delay when restarting a file system is one of the primary motivations for adopting journaling file systems.

1.4 Overview of journaling file system

Journaling file systems keep a record of the changes made to the file system *meta*-data in a separate part of the disk called a *journal* or *log*. The *journal* is an on-disk data structure. If the computer system crashes while making changes to the file system, the operating system can use the information in the log to bring the file system up-to-date by replaying the log, which is usually done when it remounts the journaling file system or verifies the consistency. A journaling file system transaction treats a sequence of changes as a single, atomic operation – but instead of tracking updates to tables, the journaling file system tracks changes to the file system *meta*-data and/or user data. The transaction guarantees that either all or none of the file system updates are done.

For example, the process of creating a new file modifies several *meta*-data structures. Before the file system makes those changes, it creates a transaction that describes what it's about to do. Each transaction is associated with a unique id and is stored as part of the *journal*. Once the transaction has been recorded (on disk), the file system goes ahead and modifies the metadata. The *journal* in a journaling file system is simply a list of transactions. The transaction id will be used to remove the entries from the log / journal when:

- The transaction is completed
- The transaction is not able to be logged in the log / journal and all the records with respect to this transaction needs to be removed

In the event of a system failure, the file system is restored to a consistent state by replaying the *journal*. Rather than examine all *meta-data*, the file system inspects only those portions of the *meta-data* that have recently changed and recovery is much faster.

2. Problem Description

2.1 Background

Some of the more inspired among us may have kept (or still keep) a journal to record the changes in our lives. Journals and diaries help us keep track of exactly what's happening to us, and also often come in handy when we need to look back and see what was happening at a specific point in time. Though not nearly so melodramatic as personal journals, this is almost exactly the same model used by journaling file systems, which keep a record of the changes made to the file system in a special part of the disk called a journal or log. If the computer system crashes while actually making those changes to the file system, the operating system can use the information in the log to bring the file system up-to-date by replaying the log, which is usually done when it remounts the journaling file system or verifies its consistency.

2.2 Requirements Description

- The file system shall possess journaling capabilities that will aid in fast recovery incase of system failure
- The file system shall support both large and small files
- The file system shall support both large and small number of directories and files
- The file system shall not predefine the number of inodes – they shall be dynamically chosen and created
- The file system shall not limit the block size of a file – they shall be dynamically chosen both at the mount time and run-time

3. Architecture of SFS

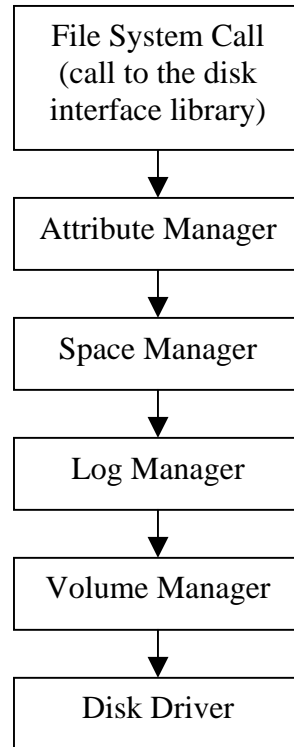


Figure 1 SFS Architecture

3.1 *Attribute Manager*

The *attribute manager* implements file system attribute operations viz., storing and retrieving of files (both regular file and directory) and manipulation of inodes. An attribute is stored internally attaching it to the inode of the referenced object. The attribute manager manages the attributes structures that are associated with inodes. The attribute manager handles creation, modification and deletion of inodes.

3.2 *Space Manager*

The *space manager* manages the allocation of disk space within the file system. It is responsible for mapping a file into a sequence of disk blocks. The space manager controls the internal structures of the file system – data groups, inodes, and free space management. Each file system is divided into log, *meta-data* and data.

The *space manager* divides each file system *meta-data* and data into a number of *data groups*. Each *data group* has a collection of inodes and data blocks, and data

structures to control their allocation. The blocks containing inodes are allocated dynamically from the data block pool, to permit more efficient use of disk space. Knowledge of sequence of inode blocks for a *data group* is kept the same as it is for ordinary files. *Space Manager* is also responsible for de-fragmenting the partition so that blocks of free and allocated blocks are stored contiguously.

Storage for files is represented in one of three ways, depending on the size and contiguity of the file. For small files, the data in the file is stored in the inode. For medium sized files, the inode contains pointers to extents containing the file data. For large files, the inode contains the root block of a B-tree indexed by logical position in the file.

3.3 Log Manager

All changes to the file system *meta-data* are serially logged to a separate area of the disk space. There is separate log for each file system. The log allows fast reconstruction of a consistent and correct file system if a crash intervenes before the *meta-data* blocks are written to disk. The log space is allocated independently from the file system space for safety – the volume manager manages this separation. The *space manager* sends login requests to the *log manager*.

After a crash, the log must be recovered before the file system can be used. Operations that are recovered and are complete in the log but are not yet stored in the data area of the file system are re-done so that the file system data reflects a correct and consistent state. The *log manager's* role in this is to identify the log records and to call other pieces of the file system to perform recovery operations.

3.4 Volume Manager

The *volume manager* is responsible for translating logical addresses in the linear address spaces into real disk addresses.

4. SFS layout

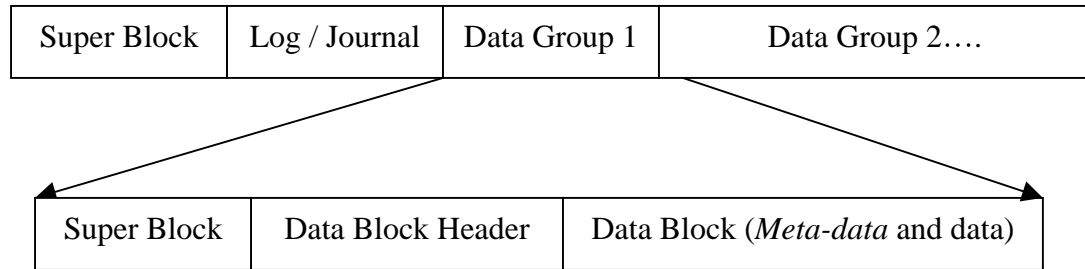


Figure 2 SFS Layout

The above figure represents the layout of the file system. The file system is divided into 3 categories:

- Super block
- Log / Journal
- Data group
 - Data block header
 - Meta-data (inodes)
 - Data blocks

The super block is a centralized resource for every file system and it contains the information that is required for the operation of the file system. The super block is modified quite often as it contains information about the number of inodes in the file system, total number of free inodes in the file system and the total number of free data blocks in the file system. The super block is replicated in all the data groups for recovery purpose. Every time the super block is changed the changes are also done in replicated super block in the data groups.

The log / journal is the meta-data about the meta-data and it is updated every time the data blocks and inode contents are modified. It does not record the changes done in the super block and this is done considering the performance, as the super block will be modified more frequently.

The data part of the file system is divided into groups called as *data groups*. The super block is also maintained in every data blocks. Even though this introduces redundant data storage, this would be helpful when the super block at the start of the file system get corrupted and the super block can easily be recovered from the data groups. Each data group contains header information that would provide pointers to the free & allocated inodes and data blocks.

5. Data Structures

5.1 *Super block header format*

- File system unique id
- Number of data blocks
- File system magic number (SFS_SB_MAGIC)
- Header version magic number (SFS_HV_1_0)
- Root inode number
- Size of an allocation group
- Number of allocation groups
- Size of a data block
 - Value 0 indicates that the size of the allocated data blocks is dynamically allocated
 - The other values being 256, 512, 1024 and 2048
- Size of the file system
- Size of the inode (256 KB)
- Total number of inodes
 - Value of 0 indicates that the file system supports unlimited number of inodes
 - Any value greater than 0 is given as input when the file system is created
- Number of allocated inodes
- Number of free inodes
- Free data blocks
- Super block lock
- Last allocated inode number

5.2 *Data block header format*

- File system magic number (SFS_SB_MAGIC)
- Data block header magic number (SFS_DB_1_0)
- Allocation group number
- Data block header lock
- Total number of allocated inodes in the data block
- Total number of free inodes
- Pointer to the first allocated inode
- Pointer to the first free inode
- Pointer to the free block

5.3 ***Inode format***

- Inode magic number (SFS_IN_MAGIC)
- Inode version (SFS_IN_1_0)
- Pointer to the next allocated inode
- Pointer to the previous allocated inode
- Inode number
- Status of the inode
- Inode access rights
- Password for the file or directory
- Inode lock
- Inode dirty flag
- Length of the file or directory
- Name of the file or directory (Shall not contain *new-line* characters)
- Inline data
- Data group number
- Pointer to the extents
- Pointer to the root node of the B-tree
- Format of the file
- Number of links to the file
- Owner's user id
- Owner's group id
- Inode last accessed
- Inode last modified
- Inode created
- Number of times inode modified

5.4 ***Journal / log header format***

- Log header magic number
- Log header version number
- Size of the log
- Pointer to first free log data
- Pointer for first allocated log data

5.5 ***Journal / log format***

- Log sequence number for this record
 - Value 0 indicates that the current block is empty
- Operation Id
 - SFS_JOURNAL_META (000)
 - SFS_JOURNAL_DATA (001)
 - SFS_JOURNAL_EXT (010)
 - SFS_JOURNAL_EXT (011)
 - SFS_JOURNAL_BROOT (100)

- Data group number that is modified
- Inode number
- Inode snapshot (does not include the contents of extents and B-tree)
- Transaction Id
- Pointer to the next log data
 - Pointer to the next free log data block if the current data block is empty
 - Pointer to the next allocated / occupied data block if the current data blocks is not empty

6. Design of Attribute Manager

The *attribute manager* implements file system attribute operations viz., storing and retrieving of files (both regular file and directory) and manipulation of inodes. An attribute is stored internally attaching it to the inode of the referenced object. The *attribute manager* manages the attributes structures that are associated with inodes. The *attribute manager* handles creation, modification and deletion of inodes. It does not handle deletion of files or directories i.e. the data blocks are not directly manipulated (both for files' data blocks and inodes' data blocks).

Each inode is associated with various security information and file specific information. The inode contains the access rights, user & group id and password for the file or directory. The user and group id indicates the user and the group having privileges to access the file or directory. The access right will provide the read, write and access rights for the files and directories. The password field will indicate the associated password with every file or directory. If the password field is empty it indicates that the file can be accessed without any password.

7. Design of Space Manager

7.1 Space Manager

The *space manager* manages the allocation of disk space within the file system. It is responsible for mapping a file into a sequence of disk blocks. The space manager controls the internal structures of the file system – data groups, inodes, and free space management. Each file system is divided into *log*, *meta-data* and data.

The *space manager* divides each file system *meta-data* and data into a number of *data groups*. Each *data group* has a collection of inodes and data blocks, and data structures to control their allocation. The blocks containing inodes are allocated dynamically from the data block pool, to permit more efficient use of disk space. Knowledge of sequence of inode blocks for a *data group* is kept the same as it is for ordinary files.

Files are stored in three ways, depending on the size and contiguity of the file. For small files, the data in the file is stored in the inode. For medium files, the inode contains pointers to extents containing the file data. For large files, the inode contain the root block of a B-tree indexed by logical position in the file.

When the file / directory is created for the first time the contents are stored as inline data in the inode itself. The size of the inline data in the inode is 128 KB. If the size of the file is more than 128 KB, then the files are stored in extents. Each extent will have 4 data blocks and each data block will be of 2048 KB. To start with, the extent will have data block of size 256 KB. If the file contents extends further then another 256 KB is allocated. The following is the order of the size of the data blocks in which it is allocated:

1. 256 KB
2. $256 + 256 = 512$ KB
3. $512 + 512 = 1024$ KB
4. $1024 + 1024 = 2048$ KB

Each inode will have pointers to 8 extents. So a single extent can store a file of $4 * 2048 = 8192$ KB. So the total size that an inode can support with just extent is:

$$128 + 8 * 8192 = 65664 \text{ KB} \approx 64 \text{ GB}$$

If the file content still increases, then the files are stored in the B-tree.

When the data groups are created a total of 64 inodes is allocated (if the total number of inodes are unlimited). Every time a data group is short of inodes, a block of 64 inodes is allocated. If the free blocks in a data group are not able to accommodate the 64 inodes, then the number is reduced to 32 inodes and reduced till 1 inode.

If there are free inodes in any data group and with no free data blocks, then the data contents of the inodes can be made to point to the data block of another data group. It is also possible for a file to be a part of more than one data group wherein a few number of data blocks are present in one data group and the remaining being in another data group.

The free space information is also stored in the data group header. The pointer to the first free data block is present in the data group header. Each free data block will be of size 256 KB. Each data block is linked together to provision easy access of the free data blocks.

7.2 De-fragmenter

Space manager is responsible for de-fragmenting the file system. As the files and directories might be constantly modified and deleted, the free and occupied data blocks will be scattered. If the occupied data blocks are scattered, the time required to retrieve them will be more due to the random movement of the diskette header. The same is also true for free data blocks if they are scattered when they are required to be allocated. The de-fragmenter also collects the empty space from every data blocks to form free data blocks.

The de-fragmenter is run periodically so that the free spaces are checked and aligned. Whenever the de-fragmenter takes control, it is executed on a data block that has less free space or recently modified. The de-fragmenter does not affect the file(s) that are currently in use and will be done the next time the data block is de-fragmented.

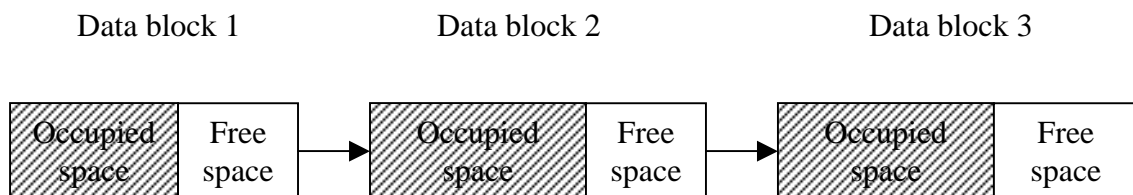


Figure 3 Data blocks before de-fragmenting (Scenario 1)

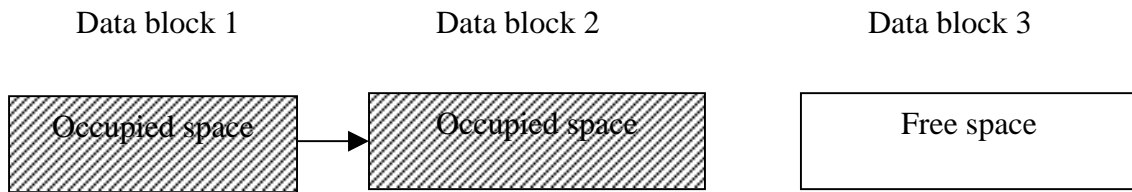


Figure 4 Data blocks after de-fragmenting (Scenario 1)

[Figure 3](#) and [Figure 4](#) represents the scenario of de-fragmenting wherein the data blocks contains some free space even though the file is spanned across 3 data blocks. Let us assume that the occupied data can very well be occupied in 2 data blocks itself and the free spaces would account for a complete free block. When the file is de-fragmented the 3 free spaces in the corresponding 3 data blocks are collected separately and the data from the 3 data blocks are moved accordingly across these blocks so that they fit in exactly 2 data blocks. The free space data block is then linked to the set of available free blocks.



Figure 5 Data blocks before de-fragmenting (Scenario 2)



Figure 6 Data blocks after de-fragmenting (Scenario 2)

[Figure 5](#) and [Figure 6](#) represents another scenario of de-fragmenting the free and the occupied data blocks are scattered. This shall be executed after Scenario 1 is completed and the data blocks are either completely full or empty. The occupied data blocks are moved to a single space so that there is minimal movement of the diskette header for any file access. The same is done for the free data blocks, that they are moved to a common place so they are present contiguously. In many cases, the movement of occupied data blocks is done in such a way that the data blocks of a file are always present in the same locality so as to avoid some varying movement of the diskette header.

Every time a file or directory is modified the count on the inode (Number of times inode modified) is incremented. And once it reaches a value of 25, de-fragmentation is invoked on that particular file.

8. Design of Log Manager

All changes to the file system *meta-data* are serially logged to a separate area of the disk space. There is separate log for each file system. The log allows fast reconstruction of a consistent and correct file system if a crash intervenes before the *meta-data* blocks are written to disk. The log space is allocated independently from the file system space for safety – the volume manager manages this separation. The *space manager* sends login requests to the *log manager*.

After a crash, the log must be recovered before the file system can be used. Operations that are recovered and are complete in the log but are not yet stored in the data area of the file system are re-done so that the file system data reflects a correct and consistent state. The *log manager's* role in this is to identify the log records and to call other pieces of the file system to perform recovery operations.

The *log manager* logs only the changes done to the *meta-data* and data blocks and does not log the changes done in super block. This is done considering the performance, as the super block will be modified frequently. Even in the *meta-data* (inode), the changes done for the access time, modification time and creation time are not logged.

The *log manager* logs the modification done to the files and / or directories in the *log* or *journal*. The changes done are categorized as:

- Changes done to the *meta-data*
- Changes done to the inline data
- Changes done to the extents
- Changes done to the B-tree

Before making any modifications, the changes are first logged in the *journal*. The inode number and the contents of the inode of the file or directory is taken as a snapshot and recorded in the *log / journal*. For every operation, a *transaction id* is generated and this will also be stored in the *log / journal*. Once the operation is complete, the *transaction id* is used to locate the transaction and is removed from the *journal*.

When the file system is mounted the *dirty flag* in the super block is set to indicate that the file system is in use. The value will be unset when the file system is un-mounted. If the system crashes or the file system is not un-mounted properly, then the *dirty flag* will be set. When the file system is mounted for the next time, this *dirty flag* is read, which would indicate that the file system was not un-mounted the previous time. The *log / journal* is read, which would have the transactions that might not have been stored properly. The *log manager* just replaces the contents of the current inode (that are present in the *journal*) with the snapshot that is present in the *log / journal*.

The *log / journal* shall have only the modification done to the file / directory and will not record if the file / directory is just viewed or read. If the file / directory is just

viewed / read, it's only the *access time* that will get modified and this will anyway not get logged.

The *log manager* does not record the contents of extents or B-tree before modification. So when the *log / journal* is read when the file system is mounted and if found to be dirty, then it's only the inode data and the inline data of the file that will be recovered. If the extents' contents or B-tree contents are modified, they will not be recovered.

The *log manager* can be used only when the file system is not un-mounted properly wherein the *meta-data* is not proper. There are other chances for the file system to get corrupted, viz.,

- The complete file system itself is corrupted and in such case the whole partition or the file system needs to be formatted
- The super block is corrupted and in such cases the replicated super block (from the data groups) can be read and restored.

9. File system formatting

Formatting a file system involves deciding the super block header, log / journal header, allocating log / journal space, allocating data groups, data group headers and data blocks. The following shall be decided when the file system is formatted:

- Size of a data block
- Total number of inodes

If the number of inodes is less than or equal to 5 (1 to 5), then so many number of data blocks will be allocated and each inode will be assigned to every data block. This is to handle the requirement to support very large files. This does not block the data block from one data group to branch to another data block if there is not enough space in other data blocks. If the user does not provide the total number of inodes, then the total number of inodes is set to 0 and the total number of inodes is unlimited. The user shall increase the number of inodes even after formatting. The user needs to un-mount the file system, increase the number of inodes and then mount the file system. This does not affect the data in the file system. However the user shall not reduce the number of inodes.

The size of the data block can also be assigned so that whenever a file requires extra space, the data block that is allocated will be of the predefined size. This data has to be carefully chosen, as lot of fragments would be created for large values of data block size. If the size of a data block is not given during formatting, the value is set to 0 and the size of the data block is dynamically calculated. The user shall change the size of a data block even after formatting. The user needs to un-mount the file system, change (increase / decrease) the size of the data blocks and then mount the file system for the changes to be effective. If the user does not provide the size then the default is 256KB. The allowed values for the size of the data block shall be 256, 512, 1024 and 2048.

10. Implemented HURD interfaces

The following are the libraries that shall be implemented:

- libdiskfs
- libpager

The following are the interfaces present in libdiskfs (of HURD) shall be implemented for the new file system (*SFS*):

1. diskfs_null_dirstat
2. diskfs_lookup_hard
3. diskfs_direnter_hard
4. diskfs_dirremove_hard
5. diskfs_dirrewrite_hard
6. diskfs_dirempty
7. diskfs_drop_dirstat
8. count_dirents
9. diskfs_get_directs
10. diskfs_cached_lookup
11. diskfs_node_norefs
12. diskfs_try_dropping_softrefs
13. diskfs_lost_hardrefs
14. diskfs_new_hardrefs
15. read_disknode
16. diskfs_node_reload
17. diskfs_validate_author_change
18. diskfs_create_symlink_hook
19. diskfs_read_symlink_hook
20. diskfs_node_iterate
21. diskfs_write_disknode
22. diskfs_set_statfs
23. diskfs_set_translator
24. diskfs_get_translator
25. diskfs_shutdown_soft_ports
26. diskfs_alloc_node
27. diskfs_free_node
28. diskfs_set_hypermetadata
29. diskfs_readonly_changed
30. diskfs_file_update
31. diskfs_get_filemap
32. diskfs_pager_users
33. diskfs_max_user_pager_prot
34. diskfs_get_filemap_pager_struct
35. diskfs_shutdown_pager

36. diskfs_sync_everything
37. diskfs_truncate
38. diskfs_grow

The following are the interfaces present in libpager (of HURD) shall be implemented for the new file system (*SFS*):

1. pager_read_page
2. find_address
3. pager_write_page
4. pager_unlock_page
5. pager_report_extent
6. pager_clear_user_data
7. pager_dropweak
8. create_disk_pager

11. Limitations / Future Enhancements

- Currently, the password field in every inode is always kept empty i.e. all the files / directories can be accessed without any password and shall be considered as a future enhancement.
- No separate utilities are provided to access the files / directories with passwords and is considered as a future enhancement. Since there is no password the normal utilities shall be used to access the file or directory.
- The size of the log / journal is limited to 512 KB and if the size of the operations to be logged exceeds this size, and then they will not be logged. As part of future enhancement the log / journal can either be stored as a separate file system or there should be a provision for expansion of the log size.
- B-tree implementation is not done and is considered as a future enhancement. So the maximum size of a file that *SFS* can support is limited to approximately 64 GB.
- The file or directory name is limited to 512 chars.
- If any data blocks is attached neither to free data blocks nor to any inode, then there is no utility to recover them (might happen when the file system is not un-mounted properly) and shall be considered as a future enhancement.
- Logging the contents of extents or B-tree is not handled in the *log manager* and shall be considered as a future enhancement
- Right now the data block size is fixed as 256 KB and the dynamic change in size of a data block shall be considered as a future enhancement
- No separate utility is provided to recover the super block and shall be considered as a future enhancement

12. Discussion

The journaling of a file system means that the file system can be recovered easily in case of improper file system shutdown. This is actually achieved by storing the *meta-data* about the *meta-data* itself so that it is only the *meta-meta-data* is verified in case of any error.

Other factors like the large sized files / directories, large and small number of files / directories and varying number of inodes are also achieved.

13. Conclusion

This dissertation provided information on the design and implementation ideas of a new journaling file system (*SFS*) for the GNU HURD operating system.

The first step in achieving this design was to study the design and implementation of the existing file systems

- Ext2 (non-journaling file system)
- Ext3 (journaling file system)
- ReiserFS (journaling file system)
- JFS (journaling file system)
- XFS (journaling file system)

Then the GNU HURD and GNU Mach were studied to have an understanding of the kernel, OS and the disk interface libraries. Then the requirements and design of the new file systems are framed that would suit HURD OS. This idea brings forth the needs for large file systems, large files, large number of files and large directories.

14. References

1. Leffler, McKusick, Karels, Quarterman "The Design and Implementation of the 4.2 BSD UNIX Operating System"
2. The GNU HURD – <http://www.gnu.org/software/hurd/>
3. The GNU HURD reference manual – http://www.gnu.org/software/hurd/doc/hurd_toc.html
4. The GNU HURD hacking guide – <http://www.gnu.org/software/hurd/hacking-guide/hhg.html>
5. The GNU HURD User's guide – http://www.gnu.org/software/hurd/users-guide/using_gnuhurd.html
6. Tanenbaum "Modern Operating Systems"
7. Knuth "The Art of Computer Programming – Volume 3", 2nd Edition
8. Tanenbaum, Woodbull "Operating Systems – Design and Implementation"
9. Mukesh Singhal, Niranjana G Shivaratri "Advanced Concepts in Operating Systems"
10. Kernighan, Ritchie "The C Programming Language"
11. EXT3 file system - <http://www.zipworld.com.au/~akpm/linux/ext3>
12. EXT2 file system - <http://www.nongnu.org/ext2-doc/>
13. JFS file system - <http://oss.software.ibm.com/jfs>
14. XFS file system - <http://oss.sgi.com/projects/xfs>
15. ReiserFS file system - <http://www.namesys.com>
16. Debian – <http://www.debian.org>

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.