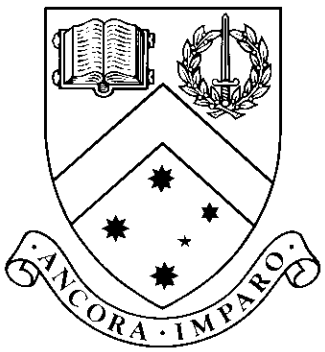
An Object Store for The Walnut Kernel

by

Stewart Smith, BCompSc



Thesis

Submitted by Stewart Smith

in partial fulfillment of the Requirements for the Degree of

Bachelor of Computer Science with Honours (1608)

in the School of Computer Science and Software Engineering at

Monash University

Monash University

November, 2003

© Copyright

by

Stewart Smith

2003

To all those times where a data store has irritated you, and all the hours you spent fscking the fscking disk, and those long hours spent mkfsing with ext2, please remember those bad blocks that made ReiserV3 go to goo.

When CVS irritated you heaps, its branching has been known to make grown men weep.

But most of all, we must remember the accidental delete and save that left you without that important something, that you just couldn't get...

And that paragraph that you had last week, which you just wish you had now...

This thesis is definitely not dedicated to fsck.

Contents

List of Figures . . . . . **vii**

Abstract . . . . . **x**

Acknowledgments . . . . . **xii**

1 Introduction **1**

2 Literature Review **3**

2.1 Persistent Systems 3

2.1.1 The Walnut Kernel 4

2.1.2 Mungi 9

2.1.3 Consistency in Persistent Systems 10

2.2 Existing File Systems 10

2.2.1 BSD FFS 11

2.2.2 Linux ext2 15

2.2.3 Linux ext3 16

2.2.4 Network Appliance's WAFL 16

2.2.5 Reiser FS 17

2.2.6 BeOS BeFS 19

2.2.7 SGI's XFS 20

2.2.8 MacOS and MacOS X's HFS and HFS Plus 22

2.2.9 BSD LFS 23

2.3 Revision Tracking 24

2.4 Conclusions 25

3 Constraints on a Walnut Object Store . . . 26
3.1 Constraints from attributes of physical disks . . . 26
3.2 Walnut Requirements . . . 27
3.3 Walnut niceties . . . 28
3.3.1 Networked Volumes and Caching . . . 28
3.3.2 Backup . . . 28
3.4 Inter-Object Dependencies . . . 28
3.5 Generally good requirements . . . 29
3.5.1 Magic Numbers . . . 29
3.6 Revision Tracking . . . 29
3.6.1 Flexibility and Extensibility . . . 30

4 Design of a Walnut Object Store . . . 32
4.1 Volume Identification - the Super Block . . . 32
4.2 Block Addressing . . . 34
4.2.1 Block Runs/Extents . . . 34
4.2.2 Allocation (Cylinder) Groups . . . 35
4.2.3 Block Size . . . 35
4.3 Free Block Tracking . . . 36
4.3.1 Block Bitmap . . . 36
4.3.2 Block B+Tree . . . 37
4.3.3 Our Implementation . . . 37
4.4 Storing Objects . . . 37
4.5 Object Node (o-node) . . . 37
4.5.1 In o-node data . . . 38
4.5.2 Object Forks and Meta-Data design . . . 38
4.5.3 Access control to meta-data . . . 40
4.6 Revision Tracking . . . 41
4.6.1 Temporal Windowing . . . 44
4.7 Object Indexing . . . 46
4.7.1 Hashing Object IDs . . . 46
4.7.2 Sparse File approach . . . 47

4.7.3 B+Tree approach . . . 47

5 Implementation of a Walnut Object Store . . . 50

5.1 General Implementation Notes . . . 50

5.1.1 Endianness . . . 50

5.2 Super Block . . . 51

5.2.1 Superblock fields . . . 51

5.2.2 Superblock Constants . . . 53

5.2.3 FCFS Super Block Flags . . . 54

5.3 Block Run . . . 54

5.3.1 Special cases . . . 56

5.3.2 Corruption . . . 56

5.4 O-Node . . . 57

5.4.1 O-Node fields . . . 57

5.4.2 O-Node Flags . . . 58

5.5 Forks . . . 58

5.6 O-Node Index . . . 60

5.6.1 O-node Index Node . . . 60

5.6.2 O-node Index Leaf . . . 60

5.7 Revision Tracking . . . 62

5.7.1 Revision Information fork . . . 62

5.7.2 Revision Data Fork . . . 64

6 Conclusion . . . 66

References . . . 68

Simulation Source Code . . . 72

.1 Source Code License . . . 72

.2 Testkit Block Device Simulator . . . 79

.3 FCFS Source . . . 99

List of Figures

2.1 Structure of a Walnut Kernel Capability 4
2.2 Walnut Disk Layout 7
2.3 Survey of disk usage on various UNIX based systems 8
2.4 Direct, Indirect, Doubly Indirect and Triply indirect blocks in file systems such as FFS and ext2 12
2.5 FFS Cylinder Group 14
2.6 User Access to Snapshots in WAFL 17
2.7 WAFL Before and After Snapshot 17
2.8 BeFS Block Run 20
2.9 BSD LFS IFILE 23

3.1 Separated Magic Numbers 31

4.1 FCFS Design thought process 33
4.2 FCFS Allocation Group on disk layout 35
4.3 Reference to data block vs. data inside the o-node block 39
4.4 Meta-Data Design 1 41
4.5 Meta-Data Design 2 42
4.6 FCFS O-Node layout 43
4.7 Revision Tracking 45
4.8 Revision Tracking - On Disk 46
4.9 Sparse object index wasted space 47
4.10 FCFS O-Node Index Tree 49

Listings

5.1 struct fcfs_sb . . . 52
5.2 FCFS Super Block Defines . . . 55
5.3 enum fcfs_sb_flags . . . 55
5.4 enum fcfs_sb_location . . . 55
5.5 struct fcfs_block_run . . . 56
5.7 O-node1 fork node . . . 58
5.8 O-node1 fork leaf . . . 58
5.6 O-node Revision 1 . . . 59
5.10 O-node1 fork flags . . . 60
5.9 O-node1 fork . . . 61
5.11 O-node Index Node . . . 61
5.12 O-node Index Leaf . . . 63
5.13 O-node fork Revision Information Fork . . . 63
5.14 O-node fork Revision Information Fork Operations . . . 65
1 testkit/Makefile . . . 79
2 testkit/bitops.h . . . 80
3 testkit/types.h . . . 82
4 testkit/block_dev.h . . . 83
5 testkit/block_dev.c . . . 89
6 testkit/blktest.c . . . 97
7 fcfs/Makefile . . . 99
8 fcfs/disk.h . . . 103
9 fcfs/onode.h . . . 105
10 fcfs/onode_versioned.h . . . 106

11 fcfs/onode_index.h 108
12 fcfs/space_bitmap.h 111
13 fcfs/super_block.h 112
14 fcfs/fcfs_vfs.h 117
15 fcfs/disk_testkit.c 118
16 fcfs/super_block.c 121
17 fcfs/space_bitmap.c 123
18 fcfs/onode.c 127
19 fcfs/onode_index.c 139
20 fcfs/mount_testkit.c 151
21 fcfs/mkfile.c 154
22 fcfs/mkfs.c 156
23 fcfs/volinfo.c 163
24 fcfs/fcfs_newobj.c 170

An Object Store for The Walnut Kernel

Stewart Smith, BCompSc(Hons)
Monash University, 2003

Supervisor: Dr. Ron Pose
Associate Supervisor: Dr. Carlo Kopp

Abstract

The Walnut Kernel(Castro, 1996) is a persistent, capability based operating system kernel developed at and extending the previous work in the School of Computer Science and Software Engineering at Monash University. Building on work in the Password-Capability System(Anderson, Pose and Wallace, 1986), Walnut has some very unique features such as its capability architecture, charging scheme and persistence. Walnut requires a slightly specialised set of features from its equivalent of a file system, the Walnut object store. Various deficiencies with the original design were found, and it was decided that alternative systems should be explored.

This thesis extends initial work on the Walnut kernel by examining the issues related to its object store. Other operating systems use a file system as their method of on disk storage and various file systems are closely examined for their efficiency, reliability and possible applicability to a Walnut system.

As processes are persistent in Walnut, care must be taken to ensure these can be restored from disk in a consistent state. That is, the image on disk should match the appearance of these objects at a specific time, in essence a "snapshot" of the process and all objects it is using. We call this the problem of inter-object dependencies as the process object depends on a specific version of the objects it has loaded.

The addition of the revision tracking of objects is examined with the aim towards helping solve the problem of inter-object dependencies and remove the consequences of user error. A discussion on the possible Walnut interface and possible security implications is included and it is found that there needs to be further exploration of "Temporal Windowing" before access to previous revisions of objects can be done in both a flexible and secure manner.

The near complete design of a new object store for the Walnut is presented providing a flexible volume format that can be extended to encompass future ideas and features. Knowledge gained from other systems helps us in determining that this system has the capacity to fulfil the listed requirements. A proof of concept implementation in a simulation environment is presented with support for most of the features discussed and reasonable performance considering its stage in development. The implementation has been written so that it can be extended and eventually incorporated into Walnut.

An Object Store for The Walnut Kernel

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Stewart Smith
November 4, 2003

Acknowledgments

This thesis would not have been possible without the help and support of my supervisors, Dr Ronald Pose and Dr Carlo Kopp. Their guidance, input and knowledge of The Walnut and related systems has proven invaluable over the course of the year.

A deep thank you goes out to everyone who has let me discuss with them file systems, data storage and The Walnut Kernel. Even if you didn't understand a word of it, you at least helped me formulate my own thoughts and ideas.

Stewart Smith

Monash University
November 2003

Chapter 1

Introduction

The Walnut Kernel(Castro, 1996) is a persistent, capability based operating system kernel developed at and extending the previous work in the School of Computer Science and Software Engineering at Monash University. Building on work in the Password-Capability System(Anderson et al., 1986), Walnut has some very unique features such as its capability architecture, charging scheme and persistence.

The basic unit of storage in Walnut is an object. A capability gives the holder a set of access rights on a view of an object. A view can be all, or part of an object. A process accesses objects by asking the kernel to load a capability, and if it is valid, the kernel memory maps the view into the process' address space.

From examining the current implementation and design of The Walnut Kernel(Castro, 1996) it became apparent that there was significant room for improvement in how Walnut stored objects on disk as the previous method lacked features that were important for Walnut to be a fast, reliable and resilient system.

Several file systems for UNIX like systems have solved some of the problems with the current Walnut object store in a variety of ways. Although varying in complexity and features, no existing system really covered all of the requirements of a Walnut system. This is partly due to the fact that in Walnut, processes are persistent.

Processes being persistent raises several problems in relation to restoring them after a crash. If there is to be a way to automatically resume processes, there needs to be a way of quickly locating process objects. Also, to successfully restore a process, you must restore all objects it was interacting with to the exact same state as they were when you wrote the process object to disk.

Other problems include: identifying a volume, tracking free disk space, tracking where objects are stored on disk, recording information about objects, dealing with object meta-data, ensuring consistency and tracking where information about objects is located on disk.

Two major additions to the Walnut object store are explored: the addition of arbitrary amounts of meta-data and tracking revisions to objects. The addition of these extra features to the Walnut object store both raise new research topics. Access control to meta-data can largely be reduced to a choice of policy but the issue of access to different revisions of an object, the problem of temporal windowing requires significant further exploration than what can be provided here.

This thesis focuses on the design and implementation issues of a fast, efficient, feature complete and extensible object store suitable for Walnut. Design choices and decisions are examined and evaluated along with new approaches and variations to problems in object store design.

Chapter 2

Literature Review

The Walnut Kernel(Castro, 1996) is a persistent, capability based operating system developed at and extending the previous work in the School of Computer Science and Software Engineering at Monash University. Building on work in the Password-Capability System(Anderson et al., 1986), Walnut has some unique features such as its capability architecture, charging scheme and persistence. Walnut requires a slightly specialised set of features from its equivalent of a file system, the Walnut object store. Various deficiencies with the original design were found, and it was decided that alternative systems should be explored.

Most modern Operating Systems base their file semantics on those of 4.4BSD UNIX with a largely POSIX compliant programmers interface. This system is well tested and its faults and shortcomings are well known. However, many advances in data storage methods have been made on such systems and they deserve close attention.

2.1 Persistent Systems

Persistence is the property of a system where created objects continue to exist and retain their values between runs of the system. An example of a persistent system is a persistent programming language, in such a language the contents of variables are preserved across runs of the program.

A persistent operating system is different from most popular operating systems as (ideally) no state is lost after an (expected or unexpected) system restart. In Walnut, processes are first class objects and are backed to disk. Although persistence has existed in various forms for a long time, occasionally making its way into a production system(Liedtke, 1993), they have not received widespread usage. More recently, there has been an interest in persistence in distributed systems(Elnozahy, Johnson and Zwaenepoel, 1992). There have been many

interesting, and rather complete, persistent research operating systems developed including the focus of our research, The Walnut Kernel.

2.1.1 The Walnut Kernel

The Walnut Kernel(Castro, 1996) builds on previous work in the Password-Capability System(Anderson et al., 1986) to create a persistent and secure operating system for general use. Walnut is designed to be portable[1] across different architectures but currently only runs on the Intel i486 compatible CPUs with limited support for other PC peripheral hardware. It has been shown that it is possible to implement ANSI stdio compatible inter-process communication to Walnut(Kopp, 1996) and gain most of the functionality of a modern UNIX like Inter-Process Communications (IPC) system.

The basic unit of storage abstraction in Walnut is an object and all data made a available to user programs is made available through objects. An object is a series of pages of which any number of these are memory mapped into a process' address space. In the current implementation, the page size is that of the host CPU architecture. This design makes it very difficult to exchange data between architectures with dissimilar page sizes. This is a problem and one which future work on Walnut will one day have to address.

Volumes represent physical media on which Walnut Objects are stored. In the current system, Objects are permanently associated with a volume (i.e. you cannot transparently move an object to another volume) and objects cannot span more than one volume. Each volume has a unique identifier, known as the **volume number**. The current system only allows 32bits for the volume number, so if Walnut were to become widely used, collisions are bound to occur. There is currently no facility to deal with this.

A **Serial** number is the unique identifier for an object on a volume - rather similar to the UNIX i-node number. A serial is only unique for a specific volume and should be as random as possible as part of Walnut's security is the difficulty of guessing valid object identifiers.

A **Password Capability** is associated with a set of attributes: a set of rights and a view. The capability is used to identify what access permissions the holder has to an object.

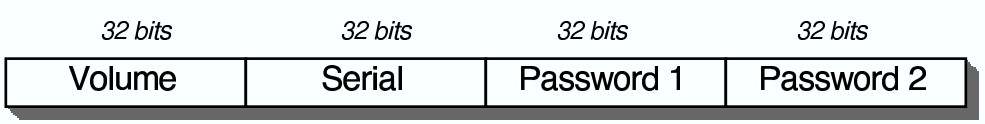


Figure 2.1: Structure of a Walnut Kernel Capability

All Walnut objects and processes (except the `init` process) are persistent. It is thought(Wallace, Pose, Castro, Kopp, Pringle, Gunawan and Jan..., 2003) that any clean up of user device

[1] Although the current implementation lacks certain infrastructure and code design to accomplish this

drivers could either be handled by the `init` process on startup or by receiving a specific type of message on system reboot.

Walnut objects have the following properties(Castro, 1996):

- They are permanently associated with a volume. They cannot be moved to another volume or span more than one volume.
- Pages are allocated on the first reference to them.
- If the number of guaranteed pages has been exceeded, and there are unreserved pages on the volume, then additional pages may be allocated to the object.
- If there are not any unreserved pages available, an exception will occur.
- Attempts to access beyond the limit of an object will cause an exception.
- The main memory acts as a cache of objects.

The backup and restoration of individual objects becomes problematic when an object is permanently associated with one volume (as it is in Walnut). Unsolved problems include: how to backup a volume without compromising the security of the system, how to restore a volume without compromising system security, how to backup a single object and how to restore a single object (especially if the original has been removed) without compromising security. Currently, the accepted attitude(Wallace et al., 2003) is that it is each users duty to back up their own objects. They could do this by periodically running a backup process and giving it capabilities to objects they want backed up. This does not, however, facilitate restoration of objects and capabilities.

Walnut builds on the Password-Capability System's(Anderson et al., 1986) concept of rental, which is used for the garbage collection of objects(Wallace and Pose, 1990). To continue to exist, an object must be able to afford to pay for the resources it uses. Each object has an amount of money (stored in its **money word**) which may be drawn by those holding capabilities with withdraw rights or by the Walnut Kernel for the use of storage space. Processes also have a **cash word** which stores money used to pay for kernel services. The **rent collector** process periodically deducts rent for disk space occupied by objects. It is commonly accepted(Wallace et al., 2003) that the rent collector should be flexible, possibly charging different amounts at times of high load or high disk usage.

Charging for services also increases the security of the system(Wallace and Pose, 1990). If each attempt to load a capability is charged for, then attempting to guess a valid capability becomes an extremely expensive operation. It is thought(Wallace et al., 2003) that charging more for failed attempts could also be useful.

Walnut Disk Structures

A Walnut Volume has three components: the **Disk-ID-block**, the **bitmap** and all other blocks which are used for storing objects. It is important to note that Walnut does not currently provide any form of journaling, consistency checking utility or guaranteed synchronous updates. This means that robustness of the current Walnut data store does not compare favourably to that of modern file systems in the event of a crash, reflecting its origins as a research platform.

On disk, Walnut objects are split into two parts: the **body** and the **dope**. The dope is all the system information related to an object including the object Header, the capability table and a set of page tables. The body is the contents of the object. The dope may grow or shrink in size separately from that of the body. A process object is the only type which enforces structure on the body but apart from that there is no distinction between processes and other objects.

Walnut does not currently keep a separate index data structure for translating object serials into disk locations of the header blocks. Instead, Walnut uses the least significant bits of the object's serial number as the block number of the **first header block** of the object. To prevent a possible attack on the security of the system by forging a serial number and a **first header block** to gain unauthorised access to other objects, the volume bitmap tracks which blocks are legitimate first header blocks. The bitmap contains a two bit value describing each block on the volume. A block can be **free**, a **first header block**, **in use** or **bad**. A full diagram of the walnut disk structure can be seen in Figure 2.2.

This method of locating objects has several drawbacks; the **rent collector** needs to operate on all objects on a volume and the only way to currently do this is to search the volume bitmap for blocks marked as **header blocks**, seek to that disk block, read the header blocks for the object, update the money word and write the header blocks back to disk. This is going to cause a lot of seeking and become problematic during periods of high disk IO. The current model also reduces what serial numbers you can store on a specific volume without having collisions with other object headers or data blocks.

The most critical drawback of this indexless scheme is the problem of automatically resuming of processes. As there is no quick way to locate process objects on a volume, the header blocks for **all** objects on a volume must be read and examined to see if they represent a process. On small volumes, this IO is fairly insignificant, but with the size of today's disks and the number of objects that todays users store on disk, the amount of IO needed to query each object on a volume to determine if it's a process becomes staggering.

As the implementation of a UNIX like environment on Walnut is possible(Kopp, 1996) and is generally viewed as the quickest and most effective way to port software to the system, a comparison with the workloads of other UNIX based systems would seem fair. It should also be noted that a 4.2BSD environment was implemented on Walnut's predecessor, the Multi although nothing was ever published about this achievement. A quick survey on the number of files present on a variety of UNIX based systems (Table 2.1.1) shows that the

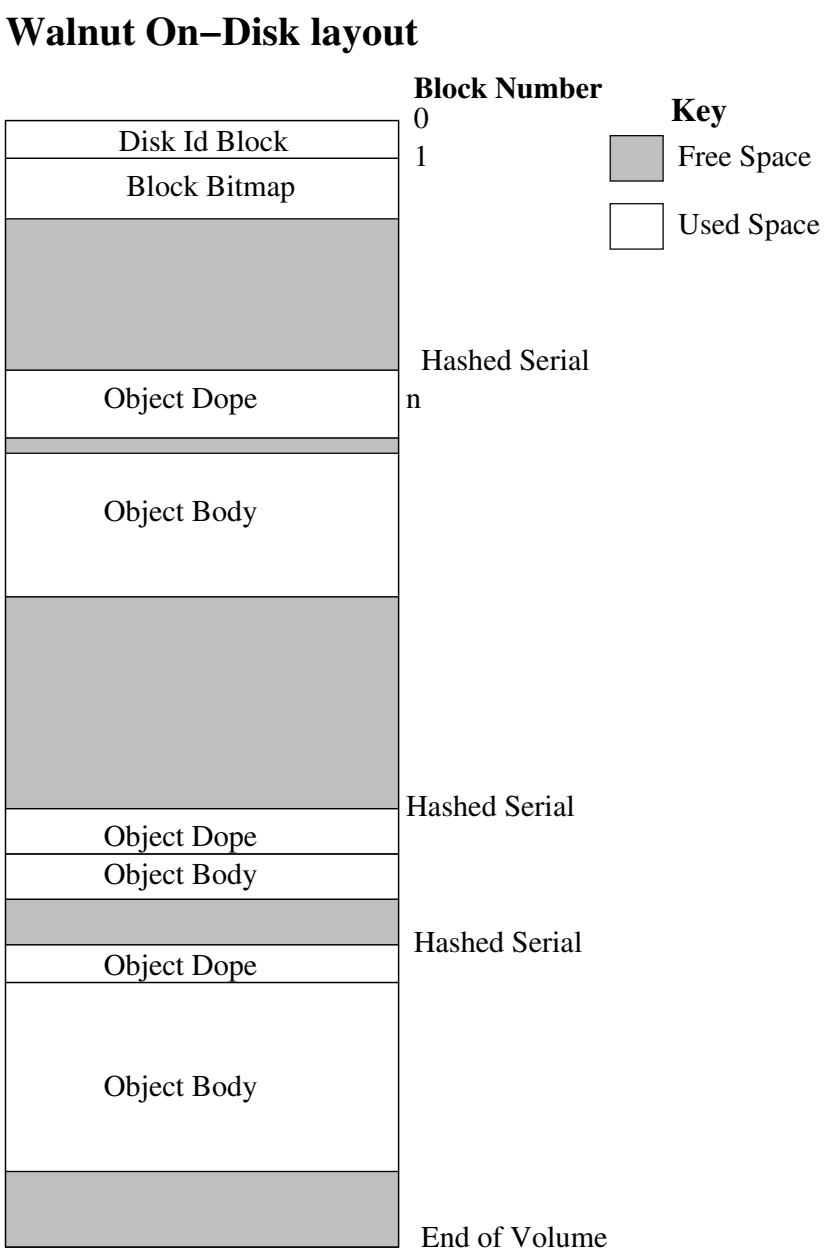


Figure 2.2: Walnut Disk Layout

Machine	Used Disk Space	Number of Objects (files)
pheonix	1.7GB	109,365
saturn	23GB	299,528
crashtestdummy	810MB	44,344
willster	34GB	422,095
cancer	71.3GB	3,065,590
yoyo	35.8GB	1257217

The machines pheonix, saturn, crashtestdummy, cancer and willster all run Debian GNU/Linux and are my personal machines while yoyo is a shared system with 50 to 60 regularly active users and several hundred rather inactive users.

Figure 2.3: Survey of disk usage on various UNIX based systems

number of objects even on small disks is quite large. With even the smallest disk examined, scanning header blocks for over 100,000 objects will consume significant disk bandwidth and even in the best conditions for disk I/O, this would take more than a few seconds of real time.

There is one area in which the current system completely fails and that is providing any form of inter-object consistency after an unclean shutdown. Since persistence is meant to be transparent to processes, the data they access should not change between the process being suspended and the process being resumed. This means that the image on disk must be a snapshot of the process and all of its data at a specific moment in time as any variation would have the same effect of modifying a running processes data which will in all likelihood lead to unpredictable results(Smith, 2003).

The Current State of the Walnut Kernel

The current Walnut implementation is unfortunately not up to the standard of other operating system projects and strongly shows its immaturity. There has been little work on the kernel since its original implementation and a lot of knowledge about how parts of it work has been lost over time. Although recent work by Stanley Gunawan has updated the code and build system so that development can be done on recent releases of FreeBSD (namely 4.4 and 4.5), problems have arisen in trying to get user mode processes to execute. There is very little hardware support and any change in hardware configuration requires source code modifications.

The Walnut source code is sparsely documented and is believed(Wallace et al., 2003)) to use "reserved" and "unused" fields in some data structures for internal scratch space. There is an overuse of constructs such as goto that have the effect of greatly obsfucating large parts of the kernel. While such constructs are can be useful in speed optimisation and occasionally even code clarity, this is not the case in much of the Walnut code. In many of the subsystems it is a complex operation to understand what is being done and how it is

being done. If a port to a 64bit architecture were to be attempted, almost the entire code base would need to be rewritten or closely audited to remove assumptions on word size.

Walnut has proven great theory, spawning several papers and other projects but the current implementation would need a large amount of work, or indeed a total rewrite before extensions to the original design could be pursued within a reasonable time frame on the existing code-base.

2.1.2 Mungi

One of the larger research projects regarding persistent systems is Mungi. Mungi(Heiser, Elphinstone, Vochteloo, Russell and Liedtke, 1998) is a persistent, Single-Address-Space Operating System (SASOS) developed on top of the L4(Ceelen, 2002) Micro kernel at the University of New South Wales. The system was designed to be a pure SASOS without sacrificing features such as protection, encapsulation and orthogonal persistence. The base abstractions that Mungi provides are: capability, object, task, thread and protection domain. Mungi also has the concept of a bank account which is (similar to Walnut's concept of money) is used to limit and control resource use.

Like Walnut, an object in Mungi is the basic storage abstraction. All objects exist within a 64bit address space and since Mungi was designed for 64bit platforms (such as MIPS and Alpha) the single address space was not deemed an unreasonable limitation on the storage capacity of a single system. The single address space approach does mean that an implementation on a 32 bit architecture would Beverly limit the amount of storage accessible compared to modern standards.

Mungi attempts to improve efficiency by having three classes of data objects:

- Transient and unshared
- Transient and shared
- Persistent

Persistent system purists would argue that Mungi is not a pure persistent system due to its support for non-persistent objects. Mungi supporters would rebut this with claims of performance improvements. Since there is not a variety of completed persistent systems to perform real-world performance tests, the claims of added performance cannot be proven over other techniques. It has been indicated(Ceelen, 2002) that transient objects in Mungi would only be used for device drivers and other objects which cannot easily be restored after a system reboot.

A Mungi object exists until it is explicitly destroyed. For each process, Mungi keeps a **kill list** of objects which that process has created. When the process finishes, Mungi will remove all processes on the **kill list**. There exists a system call to remove an object from the **kill list** so that it may outlive its creator. This approach contrasts sharply with the

Walnut view of money and paying for services (including storage). The Mungi approach means that it is possible for a buggy process to create objects, not reference them and have them exist for long periods of time (or forever if the process never quits). In Walnut, these objects would be garbage collected if they are unable to pay rent.

Unfortunately, the currently available implementations of Mungi do not have persistence implemented. Additionally, the requirement of 64bit hardware supported by L4/MIPS or L4/Alpha greatly reduces the range of machines able to run Mungi and totally eliminate commodity PC hardware.

2.1.3 Consistency in Persistent Systems

One of the main problems with persistent systems is how the on disk representation of the system should be kept consistent in the event of a crash. Since processes are persistent and will usually reference objects other than themselves, it is important that these objects were all written to disk in one atomic operation, or data a process is using could mysteriously change or disappear (due to it not have being flushed to disk)(Smith, 2003).

One method to ensure on disk consistency is Checkpointing(Skoglund, Ceelen and Lidtke, 2000). Checkpointing works by taking a snapshot of the contents of memory at a specific moment and writing that snapshot atomically to disk. The last (successful) snapshot written to disk will be the state in which the system is restored.

It has been shown(Elnozahy et al., 1992) with distributed applications that the overhead for checkpointing can be quite minimal (less than 1% for six out of eight applications, with the highest overhead at 5.8%). It is natural that **incremental checkpointing** is used to reduce the number of disk writes for each snapshot. Using **copy-on-write** memory protection, it is possible for a snapshot to be written to disk while processes continue to execute, hence having asynchronous checkpointing(III and Singhal, 1993). Such techniques were also raised during Wallace et al. (2003) were thought to be worth investigating for Walnut.

In the worst case scenario, each time a snapshot is taken (Elnozahy et al. (1992) used two minutes as an interval) the entire contents of memory is dirty and must be written to disk. It is possible that such methods as partially writing snapshots in between the time they are taken or a variation in time when snapshots are taken could help alleviate this bottleneck. It has also been shown that relaxing consistency(Janssens and Fuchs, 1993) can help in reducing the overhead of checkpointing.

2.2 Existing File Systems

Since most popular Operating Systems use an explicit storage system and a variation on UNIX file semantics, an examination of how file systems work, their performance and reliability could help in understanding what features a Walnut data store requires.

Although there has been many papers and books published on data structures, using these structures on disk is rather different than using them in memory. The overwhelming majority of texts focus on in-memory structures and not on how to optimise these for use on disk. Folk and Zoellick (1987) discusses several of the differences between data structures in memory and on disk (within files). It is clear from Folk and Zoellick (1987) that many trade offs are made in the design of file formats, it also becomes clear that it is the same way with file systems.

2.2.1 BSD FFS

The Berkeley Software Distribution's Fast File System (FFS)(*A Fast File System for UNIX*, 1984) dramatically improved file system performance over existing systems. Improvements over previous UNIX File Systems introduced by FFS include:

- Larger block sizes (at least 4096 bytes)
- the use of *cylinder groups* to exploit the physical properties of a disk to reduce seek times.
- improved reliability though careful ordering of file system meta-data writes

A lot of the performance gain of FFS was due to the use of larger disk blocks (4096 byte or larger instead of the more common 512 byte), allowing more files to fit into a single disk read. However, with 4096 byte blocks *A Fast File System for UNIX* (1984) reported that with a set of files about 775MB in size, about 45.6% of the disk space was used by the file system. The solution FFS chose was to split each block into fragments of 512 bytes (or more commonly 1024 bytes) and allocate fragments instead of disk blocks. This allowed the speed increases of having a larger block size while not wasting space with small files.

FFS keeps with the basic UNIX file system abstractions. Each i-node represents a file on disk, it contains information such as the length of the file (in bytes) and what disk blocks are being used by it. FFS i-nodes have direct, indirect and doubly indirect block pointers (See Figure 2.4 (Card, Ts'o and Tweedie, n.d.)).

This means that for large files it is faster to locate disk blocks for the beginning of files than it is towards the end. In this way, FFS is biased towards small files, where only the direct block pointers are used. This is a valid assumption for many UNIX systems as there tends to be a large amount of small files.

Directories are files containing a list of names and i-node numbers. The special entry '.' means the current directory (and should be the currently i-node number) and '..' means the directory above the current directory. The directory structure is a tree. I-nodes may appear multiple times within the tree as FFS allows for "hard links"; i-nodes can appear under more than one name.

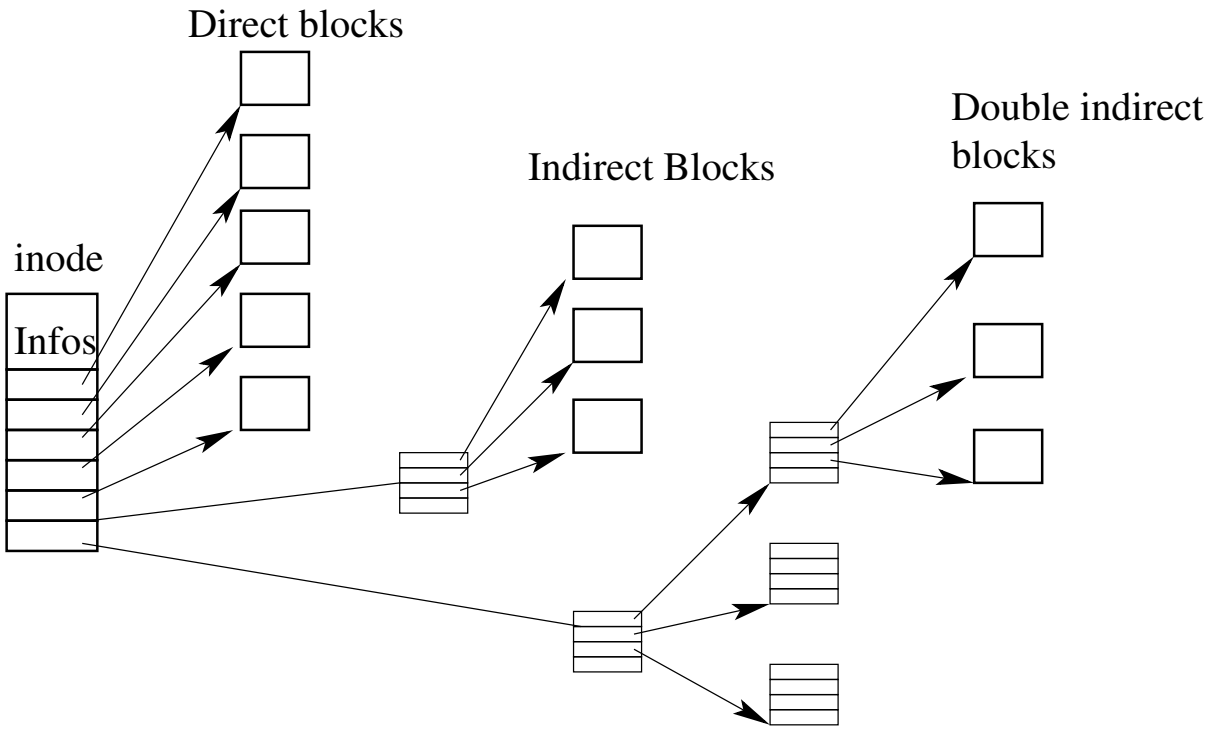


Figure 2.4: Direct, Indirect, Doubly Indirect and Triply indirect blocks in file systems such as FFS and ext2

For a file (e.g. /home/test/testfile) on an FFS volume to be accessed, the following sequence of events must take place (assuming the volume is mounted):

- Read root directory i-node
- Read root directory blocks, searching for path entry ('home')
- Read sub-directory's i-node
- read sub-directory's blocks, searching for path entry ('test')
- read directory's i-node
- read directory's blocks, searching for path entry ('testfile')
- Read file's i-node
- read file's blocks.

If all directories in this path occupied one disk block, there will need to be 7 blocks read from disk *before* any of the file's content is read.

However, the advantage FFS has over many of its predecessors is *cylinder groups* (Figure 2.5). A cylinder group is a collection of one or more cylinders on a disk. Each cylinder group has its own i-node table, block bitmap and backup super block. The purpose of a cylinder group is to exploit the benefits of locality. Files and directories used together should be in the same cylinder group, reducing seek time. The users also has the advantage of being able to do simultaneous updates to the file system meta data if the implementation supports fine grained locking as different threads could operate on different cylinder groups.

However, most modern disks hide their physical geometry from the operating system, instead preferring a *Logical Block Addressing* (LBA) scheme where disk blocks are not referenced by Cylinder, Head and Sector but by a block number. In these systems, block N+1 will be the next block to N (possibly read by a different head, but this is transparent), following the direction that the disk rotates and the transition between cylinders is transparent.

Due to the array of i-nodes being separate from the file data, seeks are inevitable between reading an i-node and reading the data. Cylinder groups reduced the distance considerably but FFS performance suffers from this, as do all systems which keep i-nodes separate from data. The speed of being able to look up an i-node in O(1) time (i-node array start block number + i-node number) is at the expense of a seek to i-node data. Caching of i-nodes by the Operating System helps overcome this seek time for frequently or recently accessed i-nodes, but initial lookup does not benefit.

Since FFS prefers to store all the files in a directory within the same cylinder group and if we make the assumption that files within a directory will be accessed at roughly the same

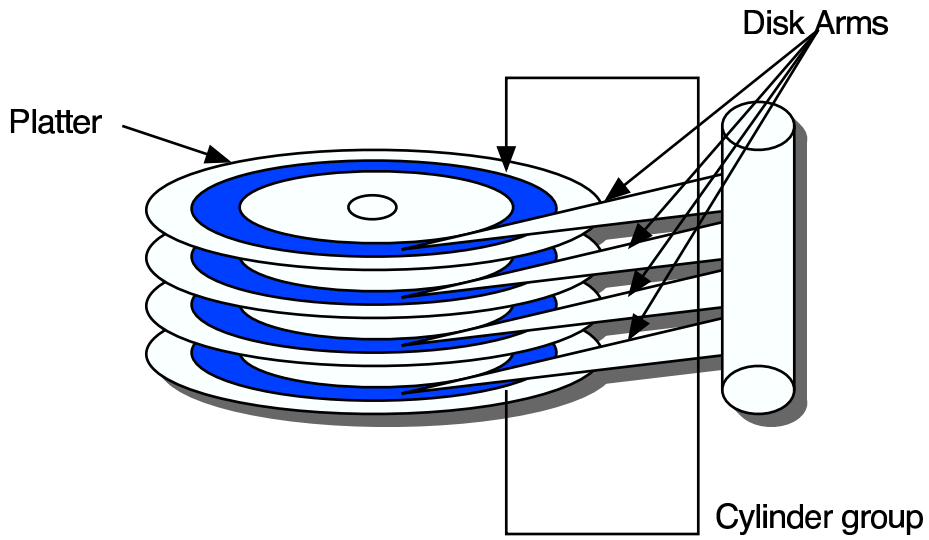


Figure 2.5: FFS Cylinder Group

time (for example, opening a Maildir style mailbox), a read ahead by the operating system could greatly reduce disk seeks from the i-node table to file data.

The reliability of FFS is largely based on carefully ordered synchronous meta data writes. Meta data writes are carefully ordered so that in the event of a crash, the `fsck` (File System CheckK) utility can relatively easily recover the file system to a sane state. However, improvements have been made which allows the file system check operation to happen in the background. These improvements are known as Soft Updates.

Soft Updates

Soft Updates(McKusick and Ganger, 1999) is a new and rather interesting way to help improve the reliability and recovery time of BSD's FFS and remove the need to wait for a file system check after an unclean mount. The principle behind Soft Updates is to track meta data dependencies and do carefully ordered writes to disk to ensure that on disk meta data is consistent. The only operation required after a crash with Soft Updates is a `fsck` process that can reclaim any allocated but unused disk blocks. This process can be run in the background as freeing blocks does not require exclusive access to the file system.

Because of the added complexity of tracking meta data dependencies, it is admitted by the authors of the Soft Updates code in FreeBSD(*The FreeBSD Handbook*, 1995-2003) there is a higher risk of bugs in the code. There is also a higher memory penalty for using soft updates but the main advantages are the little (or no) change to the on disk format and unlike a journaling filesystem, meta data is not written to disk twice. There has been much debate over which approach (soft updates or journaling) provides better performance or is

a cleaner approach although little data backing up any of the arguments has been produced and there are not any (easily locatable) convincing benchmarks.

2.2.2 Linux ext2

The ext2 file system(Card et al., n.d.) has been, for many years, the defacto Linux file system. Recently, most distributions have replaced it with the ext32.2.3 due to its added reliability. The design of the ext2 file system was heavily influenced by FFS. The data structures are rather similar in design, the most notable exception being the absence of fragments.

The main difference with ext2 is the relaxed data and meta-data integrity. All writes to the ext2 file system only make it to disk when the Operating System chooses to flush dirty blocks to disk. There is the option to do synchronous meta-data updates, but it is seldom enabled due to the loss in performance. This relaxed attitude to on-disk consistency accounts for many of ext2's speed advantages over other file systems (Giampaolo, 1999). The down side is the higher risk of file system corruption in the event of a crash and the added complexity in recovery for `fsck` utilities. With the work in Tweedie (1998), journaling has been added to the ext2 file system without a significant performance penalty.

There has also been some work to solve the problem of users accidently deleting files with the ext2 undelete project(Crane, 1999). The ext2 undelete, like undelete programs for other file systems, attempt to re-link a directory entry with the (now deallocated) i-node. This solution is not ideal, as it depends on the file system being in much the same state as when the file was removed for the i-node not to have been re-used. It also requires exclusive write access to the file system, something that is unacceptable in a multi-user environment. Debate on if this is a job that a file system should attempt to solve carries on and is unlikely to be resolved.

The ext2 file system, like FFS before it, does not cope very well with large directories. When FFS and ext2 were designed, this wasn't too much of a problem as disks were not large enough to hold enough files for this to become a real problem. However, with today's large disks and systems such as the Maildir mailbox format (where each email message is a file in a directory) it is conceivable to have directories with 60,000 files in them[2].

There has been work to overcome the inefficiencies in directory lookup in the ext2 filesystem. The Directory Index(Phillips, 2001) project has managed to devised method of directory indexing that is both backwards compatible with existing ext2 file systems and forwards compatible so that older ext2 code can fully access directory indexed volumes.

It is clear from ext2 that extensibility of the file system is very important for its continued use. There have been several important features added over time without breaking compatibility backwards compatibility.

[2]Real-world example of the linux-kernel mailing list messages from January-October 2003

2.2.3 Linux ext3

The ext3 filesystem is the same as the ext2 filesystem except for the addition of a transactional meta data journal(Tweedie, 1998). The implementation is fairly standard, employing the expected optimisations such as batched transactions. The on disk format is compatible with ext2, the journal simply being another file on the disk (albeit with a special i-node number). The goal of the ext3 project was to not destabilise the ext2 codebase but to add one new feature.

2.2.4 Network Appliance's WAFL

Network Appliance's (NetApp) WAFL(Hitz, Lau and Malcolm, n.d.) system grew out of the desire to have a solid and reliable system for networked file servers. WAFL (Write Anywhere File Layout) uses Snapshots (read only clones of the active file system) to provide access to historical data (for example, how the file system looked yesterday) and to ensure on disk consistency.

The requirements for their NFS server were(Hitz et al., n.d.):

- provide a fast NFS service
- support large file systems (tens of GB) that grow dynamically as disks are added
- high performance while supporting RAID
- restart quickly, even after an unclean shutdown due to power failure or system crash

Snapshots are made available to users through the `.snapshots` directory. Figure 2.6 (the example from (Hitz et al., n.d.)) shows how a user may recover what was in their file at the time of any of the snapshots being taken. The main advantage to system administrators is being able to take a live file system and reliably back up its contents.

The WAFL layout is best thought of as a tree where for each snapshot, the root node is duplicated and any modified nodes are copied to new locations and referenced to by the snapshot's root node. Figure 2.7 (adapted from (Hitz et al., n.d.)) illustrates the layout of the file system before and after snapshot creation, and modification.

Because WAFL only duplicates modified blocks, it is able to create a snapshot every few seconds to allow quick recovery after unclean shutdowns. When the new snapshot is created, the old one is marked as consistent. When the system starts up, it uses the most recent snapshot that was marked as consistent. Combined with a small (NVRAM based) log, this provides rapid crash recovery.

Batched disk writes and the ability to write any data to any part of the disk lead WAFL to perform rather well in NFS benchmarks. Hitz et al. (n.d.) debates the validity of these bench marks, it is fair to assume that the WAFL approach achieves good performance.

```
spike% ls -lut .snapshot/*/todo
-rw-r--r-- 1 hitz 52880 Oct 15 00:00
.snapshot/nightly.0/todo
-rw-r--r-- 1 hitz 52880 Oct 14 19:00
.snapshot/hourly.0/todo
-rw-r--r-- 1 hitz 52829 Oct 14 15:00
.snapshot/hourly.1/todo
...
-rw-r--r-- 1 hitz 55059 Oct 10 00:00
.snapshot/nightly.4/todo
-rw-r--r-- 1 hitz 55059 Oct 9 00:00
.snapshot/nightly.5/todo
```

Figure 2.6: User Access to Snapshots in WAFL

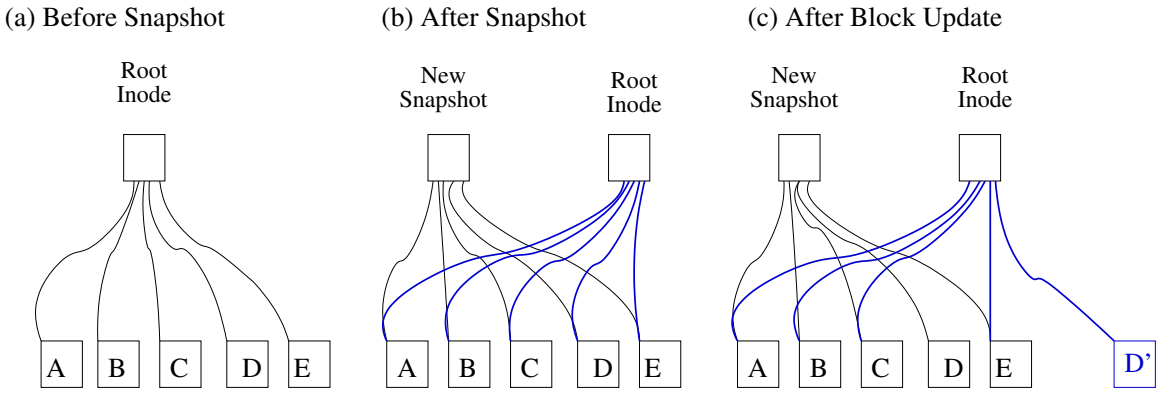


Figure 2.7: WAFL Before and After Snapshot

2.2.5 Reiser FS

ReiserFS has grown around Hans Reiser's desire to integrate the name space in the operating system. The desire was to create a fast and reliable filesystem for Linux to replace the ext2 file system. ReiserFS is a relatively new filesystem and some of it's tools (such as the file system check utility) have not yet reached maturity. However, many users like it due to its speed and efficiency, especially with small files.

Reiser (2001) claims that through the use of balanced trees, it is possible to get large performance improvements in file system throughput and fewer disk seeks during tree traversal. Reiser (2001) also believes that existing literature focuses too much on the worst case scenario, where none of the tree is cached. Reiser (2001) believes that this is not a useful thing to study as in the real world, parts of the tree will be cached for most of the time and performance in this case is more important.

The Reiser tree is designed to allow efficient handling of directories with thousands or hundreds of thousands of files, something that was previously inefficient. ReiserFS will also pack many small files into one block, so 100 byte files do not occupy one disk block each. BeFS(Giampaolo, 1999) can store a small amount of file data within the i-node, but each file will still use one full disk block. The reiserfs approach packs multiple i-nodes and data into a block, thus wasting less space. Nodes on disk which contain these small files are known as formatted nodes.

The use of formatted notes does mean that when a file outgrows the space it has in a formatted node, the system must spend extra time copying to elsewhere on disk. Eventually, as formatted nodes become more fragmented, a repacker will have to be run to help clean things up. This is the same fragmentation problem that has plagued file systems, the exception being that the reiserfs repacker should work on mounted file systems and work transparently.

The other problem with formatted nodes (which is mainly because of the design of the Linux kernel) is that memory mapping files stored in formatted nodes requires extra work. The Linux kernel can only memory map on page boundaries, which are usually the same size as a disk block (on i386, this is 4k). The computational overhead is an accepted tradeoff for the added disk space savings.

ReiserFS uses a block bitmap and some intelligent searching algorithms for allocating disk space. Reiser (2001) has empirically shown that their method, which could require extra reads from disk to traverse the tree gains good allocation and locality. With Reiser4(MacDonald, Reiser and Zarochentcev, 2002), in memory transactions use a duplicate of the block bitmap with changes only being committed once transactions enter the COMMIT stage. This allows for greater parallelism in writes to the file system as each transaction does not need to lock the single copy of the block bitmap.

Reiser (2001) argues that the speed advantages FFS gained through cylinder groups was more to do with placing semantically adjacent nodes close together on disk. Reiser (2001) found that this was an excellent approximation of the optimisations of placing data on disk according to actual cylinder boundaries (which are now often hidden from the programmer). Reiser differs from FFS in its implementation of locality is separated from the semantic layer, and it is (theoretically) possible to introduce smarter locality logic to produce a better locality_id for each object_id. A possible improvement could be to monitor usage patterns and lay out objects on disk accordingly. It was found that allocating blocks in the direction the disk spins (in increasing block numbers) significantly boosted performance over other policies of just allocating blocks 'near' each other..

Reiser (2001) debates the merits of whether files should be block aligned, and argues that the ReiserFS model of not block aligning files gives higher performance for small files. Reiser (2001) also makes the conjecture that the current usage patterns of file systems are because of how the file systems have been designed. Current systems are not optimised for small objects, and at layers above the file system, much aggregation of objects is done. If it were

more optimal to have many 100 byte files than one 4kb file, then maybe each option in a configuration file would become a separate object on disk. This is very interesting when considering that it is impossible to get any usage statistics for Walnut (as so few systems exist) and user code is limited. Current Walnut implementations limit objects to multiples of page size, but this is seen as a major limitation.

Reiser (2001) does state that the most interesting features are yet to come, and some of these are starting to appear in the next revision of ReiserFS, Reiser4.

Reiser FS 4

The main difference between Reiser FS Version 3 and Version 4 is that Version 4 is an atomic filesystem(Reiser, 2002-2003). Each operation (including writes) happens atomically, giving the impression of full data journaling. Reiser actually uses a similar system to WAFL(Hitz et al., n.d.), Wandering Logs.

Wandering logs means that any (free) area on disk can become part of the on disk journal. This is used so that each file write can be journaled physically close to the file(MacDonald et al., 2002). This can dramatically reduce the number of disk writes needed to achieve atomic writes as instead of copying blocks from the log into the file, we can simply rewrite the block pointers in the file system to point to the blocks in the log. The disadvantage is that it is possible for files to become more fragmented after an update on a disk where the free space is highly fragmented. Although, subsequent operations will not suffer this fragmentation and over time, this could actually decrease the fragmentation of files on disk.

The Reiser (2002-2003) tree, known as a 'dancing tree', which means it is only balanced in response to memory pressures triggering a flush to disk or as a result of a transaction closure (which forces nodes to be flushed to disk)(Reiser, 2002-2003). The speed improvements come from balancing the tree less often than with every update

This is similar to optimisations other systems (XFS in particular) make just before they commit to disk. It becomes clear that batching file system operations and optimising their flush to disk improves both read and write performance. This stems from how the designers look at the problem, reiserfs and xfs designers view the file system problem as one of many disk operations, not just a sequence of single disk operations and their design (and increased performance) indicates this.

2.2.6 BeOS BeFS

In early versions of the Be Operating System, extra information about files was kept in a database file on top of the filesystem. The separate database design was chosen due to the engineers desire to keep as much code in user space as possible. With wider use of the system, especially the POSIX support (which did not interface with the database

```
typedef struct block_run
{
        int32 allocation_group;
        uint16 start;
        uint16 len;
} block_run;
```

Figure 2.8: BeFS Block Run

file) problems were seen with keeping the database in sync with the contents of the file system.(Giampaolo, 1999)

The Be Filesystem (BeFS) was created out of a need for the BeOS to have a unified filesystem interface (VFS layer) and a fast, 64bit, journaled and database like filesystem(Giampaolo, 1999). Because of the nature of BeOS and their target audience, the ability to handle media files was a priority.

The BeFS is also interesting in that two engineers made the first beta release in only nine months with the final release shipping a month later. This is especially interesting given the high regard the BeOS file system is held in by many users of it. It's indexing capabilities are the envy of users of other systems even over six years after it's initial release and several years after the demise of the company.

The journaling implementation in BeFS is rather interesting as it does support the journaling of file data, but due to the limited size of the log file, only directory data is actually journaled. It would theoretically be possible to add data journaling to BeFS by allowing the journal to dynamically change size.

The BeFS **block_run** structure (Figure 2.8, from (Giampaolo, 1999) P47-48) is a unique way to address disk blocks. It takes advantage of the optimal (and common) case of several sequential blocks being allocated to a single file.

2.2.7 SGI's XFS

Silicon Graphics started the XFS project to replace their aging EFS file system under their version of UNIX, IRIX. The Project Description(Doucette, 1993c) listed the goals of XFS as: scalability (especially for large systems), compatibility among all supported SGI machines (especially small machines), support the functionality of EFS, to outperform EFS, high availability, quick recovery from failures and the system must be able to be extended in the future(Anderson, Doucette, Glover, Hu, Nishimoto, Peck and Sweeney, 1993).

The requirements to satisfy SGI's customers were rather unique at the time. SGI produced both high end (compared to the general PC industry) and very high end (1024 processors) machines and had many customers in the media business who needed to store large files,

and lots of them. There was also the scientific community who often needed to operate on large sets of data, including large, sparsely populated arrays. There was also the need to have good performance on small files (less than 1kb) as most / and /usr file systems have many such files(Doucette, 1993c).

XFS was designed to be a 64 bit file system and SGI had to deal with the problems of integrating support for 64 bit file offsets into a system that largely relied on 32 bit offsets(Sweeney, 1993a). There was the unfortunate consequence that user code had to be changed to support the larger offsets and extra system calls were added. This has been the way that 64 bit file offsets have been implemented in several UNIX variants (including Linux) and the general consensus of the community is that this is the best way.

XFS gains great scalability(Sweeney, Doucette, Hu, Anderson, Nishimoto and Peck, 1996) through its use of kernel threads(Doucette, 1993a) and message system(Doucette, 1993b) as well as fine grained locking throughout the code. The purpose behind this is to allow highly parallel access to the filesystem. Allowing multiple open transactions allows many processes to be updating the disk at once, a great benefit on large multi processors(Sweeney, 1993c)(Nishimoto, 1994).

For all its speed and parallelism advantages, the size of the XFS code is larger than any other file system discussed here. At about 120,000 lines of code, it is about fifteen times the size of ext3 and five times than of reiserfs. On small scale embedded devices, this size may still matter, but the ever decreasing cost of memory makes this point moot on most systems. Maintainability of such a large code base could become a problem, although the XFS project does not seem to have made any of these problems public.

XFS, like other systems, benefited greatly from a good simulation environment during development(Doucette, 1993d) and this method is echoed in Giampaolo (1999) as a good method to debug core file system code. This method is taken to the extreme in the User Mode Linux project, designed to enable testing of the entire kernel within a user process.

The XFS Namespace(Anderson, 1993b) is that of a traditional UNIX system. The main difference is using B*Trees for large directories. They have shown that optimising for small and large directories leads to increased performance, mainly due to a decreased number of disk reads. The in-node directories are a good example of this.

Like FFS, XFS splits the disk into cylinder (allocation) groups(Doucette, 1993e). XFS does this for increased parallel access to the file system as opposed to FFS's reasoning of increased locality. This works well for XFS as the hardware it was designed to run on is highly parallel and since the XFS transaction mechanism allows for multiple simultaneous transactions, this allows multiple simultaneous disk space allocations and deallocations(Sweeney, 1993c)(Nishimoto, 1994).

The super block of a file system is modified by most transactions and since there is data in the super block that must remain consistent, we have to journal changes to it. If we journal the entire super block, this limits us to commiting transactions serially which will no longer allow the optimisation of the order of transactions being committed to disk. This

also becomes a rather obvious bottleneck for parallel file system updates. XFS uses a clever technique to bypass these problems and instead of journaling the super block itself, XFS will journal the changes to super block fields(Sweeney, 1993b).

XFS also supports named meta-data to be associated with files, known as Extended Attributes
(Anderson, 1993a). Each attribute is a name and value pair, with a separate namespace also available for the administrator if they so wish to populate this separately from the user visible namespace. Unlike BeFS, XFS does not offer the ability to index these attributes, and unlike HFS Plus's ability to have arbitrarily sized meta-data streams, XFS limits the size of the meta-data. This severely limits what can be stored as meta-data on an XFS volume, but at the time XFS was being designed not many other systems supported any form of extended attributes.

2.2.8 MacOS and MacOS X's HFS and HFS Plus

The Macintosh's HFS and the updated HFS Plus(Inc., n.d.) volume formats are rather different from the UNIX based filesystems discussed here. It is clear from its data structures that it was designed to support a graphical environment, something which the MacOS was designed to be.

This is evident with the unique forks concept. Each file on a HFS (or HFS Plus) volume has two forks: a data fork and a resource fork. The format and access methods of the Resource Fork are specific to the MacOS and is documented in (*Inside Macintosh: More Macintosh Toolbox*, 1993). Each fork is a stream of bytes and either fork can be of zero length. This is effectively the inverse of UNIX hard links. Hard links are many names for one stream of bytes, while forks are one name for many streams of bytes.

HFS Plus has an attribute file which is intended to support an arbitrary number of named forks sometime in the future. The goal behind this was to be able to attach extra data and meta-data to files and directories that is moved and removed with that object - an advantage over the more traditional method of adding hidden files to a directory. The MacOS currently only uses two such streams for its traditional Data Fork and Resource Fork.

A common way for MacOS based word processors to store documents was to have plain text in the data fork (so that the raw information could be easily read by other applications, even on other computing platforms) and all the formatting information in a resource in the Resource Fork. The MacOS provided the Resource Manager(*Inside Macintosh: More Macintosh Toolbox*, 1993) set of APIs to manipulate a simple two-level namespace within the resource fork. It was also common for User Interface specific information to be stored within the resource fork. One such application was to store the name of the program used to create a document so that somebody trying to open it who did not possess the necessary software could be notified of what software they needed to open the document.

In contrast, the Be Operating System (BeOS) used indexed attributes to store and access meta-data for meta-data such as MIME types of files, Artist and Album titles of music files and the status of email messages. The query interface meant that users could perform complex searches of the contents of attributes. The BeOS engineers found that keeping the attributes associated with the file, on the file system as opposed to in a separate "attributes file" was of immense benefit in the general efficiency of the storage system(Giampaolo, 1999).

2.2.9 BSD LFS

The BSD Log Structured File System(Seltzer, Bostic, McKusick and Staelin, 1993) is rather different from the traditional file system. The disk is treated as a log file, with each write being to new disk blocks, never overwriting previously used ones. This approach is believed to increase file system write performance and this is cited as the main reason to use a log based file system.

Although LFS does sequential writes, a set of FFS style index structures are maintained to support efficient random retrieval. The *i-node map* maps i-node numbers to disk addresses. An LFS disk is divided into fixed sized segments, typically of 512kb. When dirty (modified) blocks are to be written to disk, LFS will write a segment, or a *partial segment* (for when there are not enough dirty buffers to fill a segment) to disk. Each segment contains a summary block which contains the i-node and logical block number of each block in the segment. The *ifile* structure (Figure 2.9) is a read-only file on disk which contains the segment usage table and is used by the cleaner.

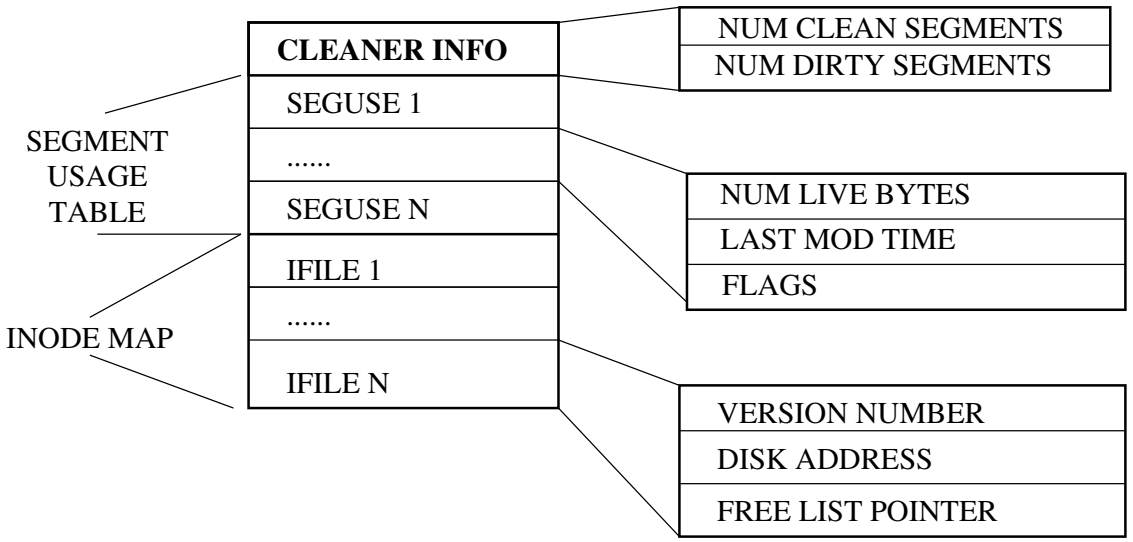


Figure 2.9: BSD LFS IFILE

There is a garbage collection process (the *cleaner*) which will reclaim disk blocks that belong to deleted files or have been superseded by more recent writes. This does lead to a performance hit when the disk is near full as the cleaner must run often. The kernel maintains a *segment usage table* which tracks the last modified time and the number of "live" bytes in each segment; the cleaner uses this table to find segments to clean.

Because of the log structure and the no-overwrite policy, it is possible to implement a facility to request previous revisions of a file. Although the BSD-LFS does not implement this, Seltzer et al. (1993) does indicate that this, among other features could be easily added to LFS.

For an "unrm" (un-remove) utility, the primary problem to be solved would be locating the old inode on disk. It is theorised that this would be a rather trivial problem to solve(Seltzer et al., 1993).

The Seltzer et al. (1993) paper shows favourable performance benchmarks, not showing any significant performance penalty over the use of FFS. Indeed, in some areas, such as simultaneous random updates, LFS has a large performance advantage over other systems. Real world use of LFS has been rather small, possibly due to peoples general reluctance to switch to something radically different than that which they are used to.

2.3 Revision Tracking

There have been various attempts to integrate revision tracking into the file system. The VMS operating system saved each revision of a file into a separate file with a version number appended to the name ('file;1','file;2' etc). This can (partially) help the user reverse errors such as accidental deletion of files (or part of the content of a file) or wanting to change something back to the way it was. LFS shows that it could be possible to add revision tracking to an existing system without a large performance penalty, but no implementation exists to do this and the LFS user base is rather small despite having existed for nearly 10 years.

The best revision tracking most systems currently hope for is an undelete utility. Traditional undelete utilities for existing file systems are not guaranteed to work reliably or on active file systems(Crane, 1999). Some file systems lend themselves to effective undelete utilities, and others do not. They may have a preference to reuse recently deallocated disk blocks and i-nodes (such as XFS does) or dynamically allocate i-nodes anywhere on disk, making a search for the i-nodes of delete files a very expensive operation.

Many users have resorted to using version control systems such as RCS and CVS(Tichy, 1985) to store and track changes to their documents. Typically, these have required more of an expert knowledge than many end users have, and have not been used widely outside the developer community (who are used to such tools for Source Code Management). More mainstream applications such as Microsoft Word and AbiWord have started to integrate simple graphical interfaces to store and retrieve document revisions, but they have yet to

gain common usage. The possibility of integrating such a feature into the file system is desirable as it would mean that revision tracking would be transparent to applications and instantly available to all the tools users are familiar with.

In both Tichy (1985) and MacDonald (n.d.), it is shown that the delta of two file revisions can be computed in reasonable time and stored efficiently. Systems such as WAFL(Hitz et al., n.d.) and LFS(Seltzer et al., 1993) have provided limited support for snapshots of the entire file system and in the case of WAFL, allowing users to access previous revisions of files. The WAFL approach uses significantly less disk than the VMS approach of revision tracking as WAFL only duplicates blocks that differ (and applicable meta-data blocks) between revisions.

The XDFS(MacDonald, n.d.) approach is rather different to that of WAFL, XDFS stores all deltas between a file's revisions. At any time, any revision of any file on an XDFS volume can be accessed, not just snapshots of the entire system.

2.4 Conclusions

Existing file systems have several things in common:

- the name resolution system is separate from how files are stored on disk
- Files are represented on disk by a structure generally referred to as an i-node containing a small amount of set meta-data and pointers to disk blocks containing file data
- Extensible systems have had features added when needed and remained popular.

There are also several trends in file system design:

- There is a trend to support arbitrary amounts of meta-data
- Speed for large files and large numbers of files is increasingly important
- Quick crash recovery is now a requirement
- A trend towards data journaling to ensure the contents of files are not corrupted after a crash.

Chapter 3

Constraints on a Walnut Object Store

In this chapter we define the constraints placed on the design of a new object store for Walnut. We examine the restraints placed on us by the physical properties of disks, the requirements stemming from the design of the Walnut kernel as well as the possibility of supporting extra features that would be of benefit to Walnut. Following chapters examine the design and implementation of a system to meet these constraints.

3.1 Constraints from attributes of physical disks

The physical attributes of any kind of modern disk place constraints on the design of any on disk structures. Although the physical design of the disk (Cylinders, Heads, Sectors) is now usually hidden from the programmer, the basic physics still apply:

- Disk seeking is expensive and should be avoided
- The seek time from block a to b is proportional to $|b - a|$
- Accessing N blocks at once is quicker than accessing N blocks one at a time.
- It is quicker to access blocks in the direction the disk spins (i.e. with the head at block N, block number N+1 is quicker to access than N-1)
- Areas of a disk degrade over time and some blocks become 'bad', that is they can no longer be reliably used to store data.

The ideal access pattern for disks is where all required data is sequentially on disk and the software accessing it can process it the same speed that the disk can read it. The same is

true of disk writes. Unfortunately, this can never be the case, so some compromises have to be made. The closer an object store can get to this ideal, the better its performance.

These constraints have remained true through the entire history of magnetic disks. The same limitations also apply to optical media such as CD-ROMs and will also to emerging storage technologies such as holographic storage. The situation is different with flash media as there are not the physical delays of head movement and platter rotation and any optimisations made to avoid these will be void.

Flash devices also have the added restriction of a limited number of writes before failure and a larger block size (128KiB for NOR flash and 8KiB for NAND(Woodhouse, 2001)). Systems such as the Journalling Flash File System(Woodhouse, 2001) have been specifically designed for these devices and their (typically) embedded environments, so it is viewed that a storage system for these devices should be designed around their specific needs as they are different enough from disks to warrant such treatment.

Devices such as PCMCIA flash cards do not need a special system designed for them as the standards specify methods to emulate the behaviour of more traditional block devices and to cycle the use of blocks so that no one set of blocks gets worn out before any other.

3.2 Walnut Requirements

The system the object store is primarily designed for is the Walnut Kernel(Castro, 1996). The design of Walnut places several design constraints on the system.

- Objects must be easily memory mapped
- Object lookup must be fast
- The storage system must be robust
- The storage system must ensure both data and meta-data consistency.
- The storage system must support a persistent system
- The storage system must not leak information (one of Walnut's goals is security)
- The storage system must be suitable for implementation in a micro-kernel environment.
- The storage system must support caching of objects from other devices (Walnut is meant to be distributed)
- The storage system must be able to support Walnut's password-capability architecture.

3.3 Walnut niceties

3.3.1 Networked Volumes and Caching

Since Walnut is designed to be distributed, it makes sense that volumes will be mounted over a network. Traditional caching of remote objects has been limited to RAM as a local backing store.

In modern networked environments, most client machines have a local disk that is largely unused (usually storing just an operating system and applications software) and user documents are stored on a networked volume. Other techniques have been developed to manage many machines of the same configuration such as disk imaging and automated software updates from a central server. What a lot of setups really want is all workstations to have a copy of the networked disk, but use the local disk as a large cache. Using protocols such as rsync(Tridgell, 1999) it would be possible to update objects in the local cache on demand and efficiently in both speed and bandwidth.

3.3.2 Backup

The problem of backup and restoration in a secure environment has been long standing and many believe it has not been solved satisfactorily. This is untrue for the backup and restoration of an entire volume, which can be accomplished with the creation of UNIX like `dump` and `restore` utilities. Care must be taken to not expose sensitive user information to the party running the `dump` or `restore` utilities. Features such as limiting how many times the capability for the disk may be loaded simultaneously and various locking mechanisms could aid in keeping the security of the system.

If we make the assumption that the primary interaction between users and objects is going to be via a name-server, the restoration of individual objects can be approached by inserting a new object (with the contents from backup) with the same key into the name-server's index. This is much the way that existing per-file backup strategies work on other platforms.

In the design of the new storage system for Walnut, we should provide mechanisms for the efficient backup and restoration of entire volumes but not dictate a policy on how this is to be done.

3.4 Inter-Object Dependencies

When resuming a process object, all objects mapped by it must be consistent with what the process last saw. Persistence is meant to be transparent to user processes which means that they should not need to do any sanity checking after an unclean shutdown. If the objects on-disk that were being used by a process are not from exactly the same time as

the process, it will be as if memory has been altered underneath the process, creating very unpredictable results.

3.5 Generally good requirements

Maintaining file-system (or indeed, object store) meta-data consistency after an unclean shutdown is now considered a requirement of any system as lengthy consistency checks can be both problematic and immensely time consuming. Most existing systems solve this by using some form of transactional meta-data journal, and since this is not a new area, the implementation details of journaling are not covered and left as an exercise for a future revision.

In the case of disaster, where parts of the volume have been damaged, possibly in very unpredictable and catastrophic ways, it is important to provide every mechanism possible for tools (or people) performing volume recovery. A common method of reconstructing a volume involves scanning the volume for data that could belong to a file system data structure. The use of magic numbers can be of great use, especially if they are unusual values and at unusual offsets within blocks.

3.5.1 Magic Numbers

Magic numbers are often used in data structures to aid in debugging. When dealing with on-disk structures, they can also greatly help in disaster recovery. By searching a disk for blocks containing magic numbers in specific locations it can aid us in reconstructing a partially destroyed volume.

By separating magic numbers withing data structures (e.g. Figure 3.1) it allows better detection of corruption within a specific area of a data structure. For example (from Figure 3.1), if a buffer overflow was to write 11 words into value4[], hence overwriting MAGIC2, we would still be able to see the MAGIC1 number and examine the possibility of this being a `data_structure`.

It is for these reasons that we have carefully designed some data structures and the placement of magic numbers within them to help aid volume recovery tools.

3.6 Revision Tracking

Tracking revisions to objects could be of great benefit to users of the system. It is widely acknowledged that users make mistakes and that program error can be time consuming to correct. The speed of such a feature is dependent on the efficiency of computing and storing deltas, the differences between revisions.

Revision tracking also raises important questions about access permissions and time. Objects (and with Walnut, capabilities) change over time. Questions about capabilities and their validity period must be answered to maintain the security of the system.

3.6.1 Flexibility and Extensibility

The most successful file systems have been those which have been shown to be extensible either through defined parts of the design or incompatible updates to the disk format.

The Linux ext2(Card et al., n.d.) file system has had many extensions implemented, one of which is journaling(Tweedie, 1998) (ext3 is ext2+journaling). More ambitious additions such as directory indexing(Phillips, 2001) aim to remove initial limitations of the ext2 file system. Apple's HFS Plus volume format also leaves part of the specification to be determined for "future use".

The design for any new storage system should be easily extensible both in on disk format and design. The power and flexibility of the system should exist in the data structures, not in the algorithms required to manipulate them. This will enable shorter and simpler code, which is in turn easier to debug and leave room for future expansion.

```
1 struct a_data_structure {
2         u64 MAGIC1;
3         u32 value;
4         u32 value2;
5         u32 value3;
6         u32 value4[10];
7         u64 MAGIC2;
8         };
```

Figure 3.1: Separated Magic Numbers

Chapter 4

Design of a Walnut Object Store

In this chapter we explore the overall design of a new Walnut object store and the reasoning behind design decisions. Figure 4.1 is an outline of some of the problems and possible solutions to them which are explored in this chapter. The more implementation specific details are left for the next chapter which covers more specific data structures. At various points the new object store may be referred to by its code name, FCFS .

The object store is implemented on top of a flat, logical block addressing model of a disk where any re-mapping of blocks that may be done by volume managers or disk firmware is ignored and the constraints from Section **??** are assumed to hold true.

4.1 Volume Identification - the Super Block

The super block details some global information about the object store and where certain critical data structures can be found on disk. Designed to be easily identifiable (especially by relatively simple boot-loaders) as both a FCFS volume and the specific revision of the FCFS system which last wrote to the volume, and if the code is compatible with it (and if it is read-only or read/write compatible).

A copy of the super block is placed at volume creation time at the start of each allocation group. Since it would be too costly in seeks and writes to expect to keep all of these copies consistent all of the time, we will only ever guarantee that the primary super block (the one at the start of the disk) is consistent. Such consistency would be achieved through journaling which is not discussed in this thesis.

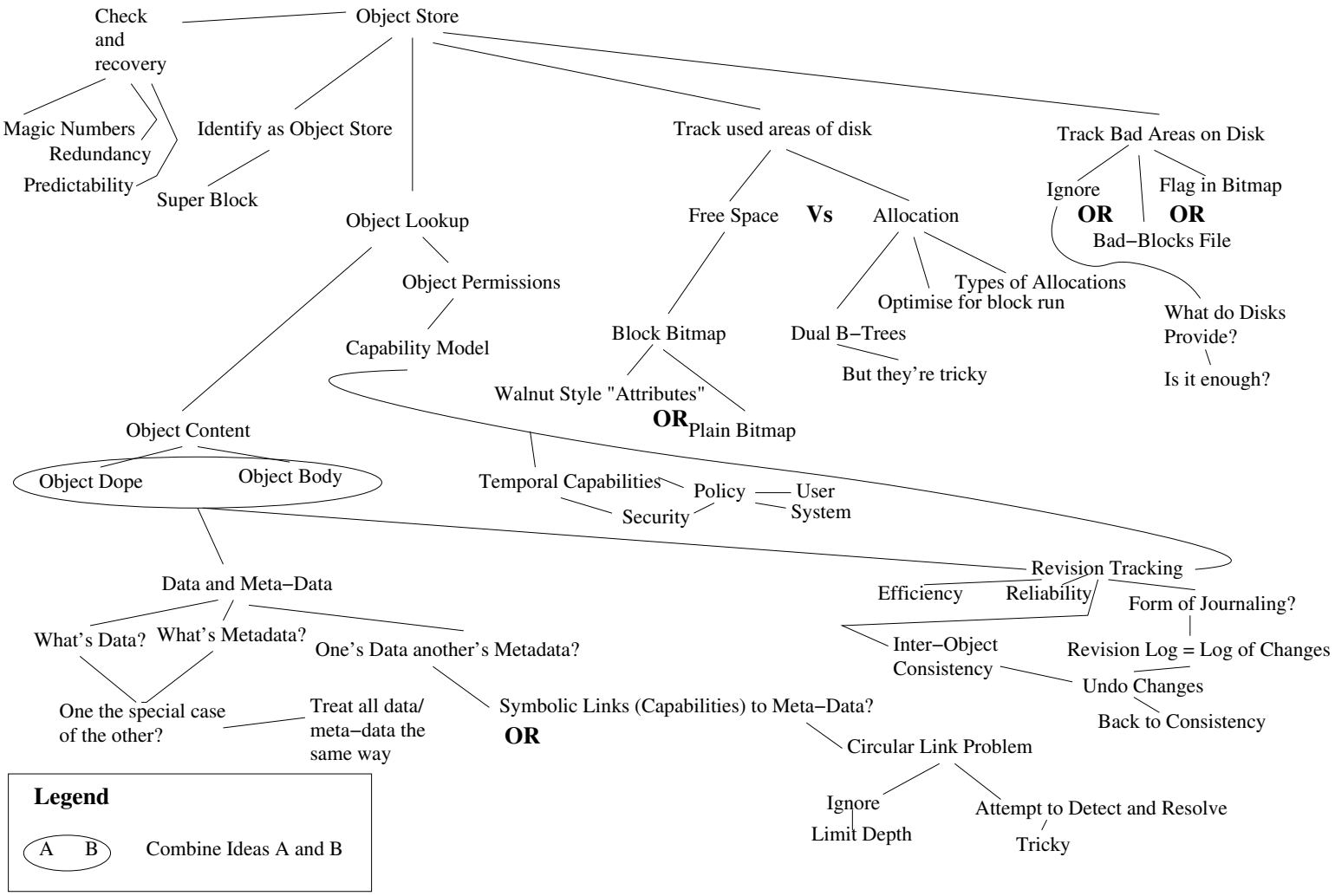


Figure 4.1: FCFS Design thought process

4.2 Block Addressing

There are a number of ways of addressing blocks in use today. The way we (usually) have to reference them to the block device layer of the Operating System is by a logical block number from the start of the block device. Most systems also have the ability to specify an amount of read-ahead that will proceed asynchronously after reading the block we requested.

Traditional UNIX file systems track what parts of the disk are being used by a file in its i-node structure. There is an attempt to optimise for the (traditionally) prevalent small file on UNIX systems. Thus, on very large files it is quicker to access the first few blocks of a file than it is to access the last one. Traditional UNIX systems (e.g. FFS, ext2, ext3) the i-node has space for several block numbers (direct blocks), several indirect blocks (that is, a pointer to a structure that lists block numbers), less doubly indirect blocks (a pointer to a structure of pointers to structures with block numbers) and sometimes even an entry for triply indirect blocks (see Figure 2.4).

This system has several disadvantages when dealing with larger files. Although a block at the start of a file will not require a seek to find, a block at the end of a file may require up to four extra seeks to find. Also, this method provides little advantage in the optimal case of all blocks in a file being sequential on disk.

4.2.1 Block Runs/Extents

Since the optimal way to interact with disks is with large sequences of blocks, it makes sense to try and optimise a data store for this case. Most block allocation algorithms will attempt to allocate the blocks in a file to sequential blocks on disk. It makes sense to optimise the recording of which blocks used by a file for this optimal case.

BeFS(Giampaolo, 1999) uses a structure called a Block Run (Figure 2.8) instead of block numbers to reference locations on disk. This has the distinct advantage of optimising for the best-case scenario where each file is a contiguous run of blocks on disk (a single structure can address up to 64MB with a 1kb block size). The `allocation_group` member refers to the number of the BeFS allocation group.

We use a variation of the BeFS `block_run` structure, removing several of its limitations. The original structure (Figure 2.8) was limited to 2^{48} blocks on a volume (with 1K block size, a maximum volume size of 2^{58} bytes). The FCFS block run (Figure 5.5) has an addressing limit of 1.84×10^{19} blocks. Considering that even with the (relatively small) block size of 1K, this would address over 16.7 million Terra-bytes of data, it is commonly accepted that this will not be a problem anywhere in the near future.

The Block Run (often referred to as extent by some systems) is used to address a series of blocks on disk. The term block run is used over extent purely due to personal preference.

4.2.2 Allocation (Cylinder) Groups

We split the volume into several allocation groups (similar to XFS and BeFS) so that is possible to have several threads updating different parts of the disk without contending for the same lock. We assume that the disk, the disk controller or the driver software performs the optimisations related to cylinders (so that the logical blocks 1,2 and 3 are on cylinders 1,2 and 3 respectively).

Although there is no specific data structure for the allocation group, several other structures are used. The first block is a backup copy of the super block (See section 4.1) as it was at allocation group creation time. For most systems, this will be the volume creation time but in future releases it is intended that allocation groups could be added on the fly to expand the size of the volume. There is then the block bitmap (see Section **??**).

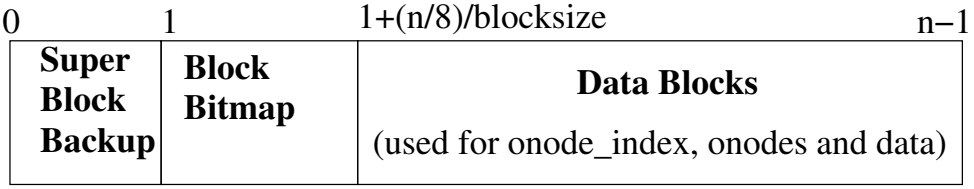


Figure 4.2: FCFS Allocation Group on disk layout

4.2.3 Block Size

Some studies have shown that for multimedia applications, a larger block size can improve throughput for these throughput sensitive applications. This is especially true when trying to read several media streams from disk. An example block size of 256kb has been shown to improve performance for streaming media. Since media files are usually quite large, the internal block fragmentation of an average 128kb per file is insignificant. However, for small files, this wastage can become problematic.

We believe that the same performance gains can be achieved through better disk read-ahead, batching of commands to disk and better disk scheduling algorithms. It would be better to improve these areas as the benefits will be available to all applications accessing the disk. The use of a block run structure can be thought of as a better solution to this problem, as it allows prefetching to easily be done on a much larger amount of data.

There is the option of having different allocation groups have different block sizes, but the added implementation complexity and the removal of some of the parallelism advantages of allocation groups, such a feature is not deemed worthy to pursue.

Systems such as XFS and BeFS allow data for small files to reside inside the i-node disk block. Since the i-node data structure is often much smaller than the disk block size, many files consuming less than one block of space can fit in the same block as the i-node. Often,

the space used for referencing what disk blocks are being used by a file are also used to store in i-node file data. This is a relatively trivial feature to implement, and we shall use it in our system to varying degrees and plan to expand its use in future revisions.

ReiserFS(Reiser, 2001) goes for something slightly different allowing several files to share a single block. For our improved Walnut object store, it is debatable if the extra effort involved in implementing this is worth the limited benefit it may provide us. For initial revisions of the object store, the ability to have in o-node data is perceived to be adequate but further exploration of the ReiserFS approach should be done if in o-node data alone proves inadequate. It should be noted that initial benchmarks of Reiser4(Reiser, 2002-2003) show a negligible penalty for storing several objects in the one disk block, an improvement over Reiser3.

The new Walnut storage system will allow the creation of volumes with large block sizes, which allows for the rare circumstance where there may be a measurable performance improvement in doing so. The time needed and complexity in implementation of mapping multiple objects into one disk block is viewed too great for this initial system, although if usage patterns show that many tiny objects are being used we may have to reconsider this decision.

4.3 Free Block Tracking

Two common scenarios exist for allocating a number of disk blocks. A request for a large sequence of blocks indicates that the preference is for a contiguous chunk of disk space (this will often be the case when creating or appending to media files) while modifications to files such as documents will usually require only a few blocks, but close to the rest of the file to avoid large amounts of seeking. Any free space tracking and allocation system should perform well in these two scenarios. Unfortunately, many systems have chosen ease of implementation over efficiency during runtime and we are currently no different.

4.3.1 Block Bitmap

The standard method of tracking used blocks has been to use a block bitmap. Each bit in the bitmap represents one block on disk; when set, the block is in use and when not set, it is free. The main reason that the block bitmap has been so popular is because of its ease of implementation. This is the reason that it has been used here and the design choice is regrettable and should be corrected in a later revision of the system.

The main disadvantage to the block bitmap method is the efficiency of allocating blocks. Finding a free block which is near another block (n) is relatively simple, a simple search through the bitmap until a free block is found works quite well where there are free blocks nearby. In the worst case scenario, where the only free block is $n - 1$ this algorithm will scan through every block on the disk, a rather expensive operation. It may be worth

implementing extra logic to scan for free blocks behind n once a certain threshold of scanning forwards is reached, but real world tests would be needed to see if this gives any better performance.

The other scenario, looking for a sequence of n free disk blocks is not well supported by a block bitmap. The same linear search must be employed as for the first scenario. However, the ease of implementation is the main advantage, and this is why it was chosen over the more efficient method of keeping two B+Trees.

4.3.2 Block B+Tree

SGI's XFS system keeps two B+Trees which index free space on disk. One of them is keyed by block number, the other by the length of the extent. This means that any lookup for free space will occur in $O(nlogn)$ time and always find the best location on disk. Unfortunately, the added complexity in implementation means that such a mechanism is not commonplace.

4.3.3 Our Implementation

For ease of initial implementation, a simple block bitmap has been implemented. It will be replaced by dual B-Tree structures in a later revision of the system as it is viewed to be a more optimal approach. There is a separate bitmap for each allocation group, starting in the second block (Block 1) and using a minimum of one disk block.

The bitmap size is dictated by 1 bit per block in the allocation group. In the event of the last allocation group being smaller than the rest, the bitmap should be large enough to expand this allocation group to the same size as the others. This is to allow the (future) feature of dynamically expanding the size of a volume.

4.4 Storing Objects

To store objects on disk we need a data structure to record where the content of the object can be found on disk and any meta-data that the operating system requires about that object.

4.5 Object Node (o-node)

The o-node is the name we give to the data structure describing an object on disk. It is similar to the UNIX file system concept of an i-node but contains a lot less meta-data as most of the meta-data is specific to UNIX. The o-node has been designed to be flexible and extensible so that when new features or requirements need to be added, it is easy to integrate

them. The O-Node contains several other data structures and concepts, a complete view of the o-node and related data structures can be found in Figure 4.6.

The initial design of the o-node structure has been simplified in certain areas for ease of implementation and debugging. As there is room for future improvements - mainly related to the efficiency of packing data into blocks or the o-node block - the current structure has been labelled **fcfs_onode1** with the intention that when a more efficient o-node is implemented, backwards compatibility can be maintained.

The key:value pair in the object index is the o-node Id and the block number of the o-node data structure. Each o-node contains a B+Tree of the forks in the object. It is thought that most objects will not have many forks and that it is unlikely for an o-node to contain anything but a leaf of the forks tree. We optimise for this case by including the fork information inside the o-node block so that extra seeks and reads are minimised.

Each fork contains a B+Tree of block runs (again, highly unlikely to ever be anything but a leaf as the majority of objects on disks are contiguous). It is also possible for the fork data structure to hold some data instead of the B+Tree to optimise for forks holding only small amounts of data. In the initial implementation, this data is limited in size but it would make sense to explore the possibility of allowing variable amounts of data inside the o-node block.

4.5.1 In o-node data

Although technically this should be titled "in o-node fork data", this title is more in tune with how other systems describe what we do. If the content of a fork is small enough to fit within the o-node's disk block, we will pack it in there so that we do not waste an extra block of disk space and require a seek (Figure 4.3). The current implementation limits how much data this is, but a future revision should allow up to the remainder of the o-node block to function as this "small data" area.

This is to allow more efficient access to small forks, which are likely to be meta-data such as the Walnut object header. On other systems, such data is stored directly in the i-node and by allowing small forks to be stored inside the o-node block, we achieve the same level of performance as these systems.

The ReiserFS approach of tail-packing(Reiser, 2001) is probably a more efficient and flexible design, but is considered too complex for the time constraints of this project. Such approaches should be investigated for future revisions of the object store.

4.5.2 Object Forks and Meta-Data design

The fork terminology (as used by HFS+(Inc., n.d.)) means that an object (in HFS+ terminology, a file) can have more than one stream of data. The advantage of having

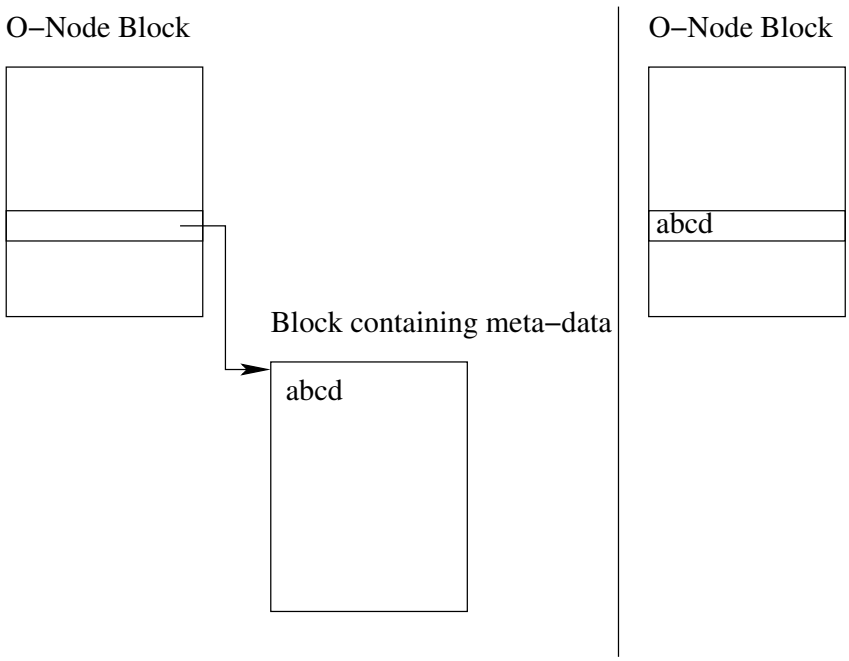


Figure 4.3: Reference to data block vs. data inside the o-node block

multiple data streams associated with one object is that extra meta-data can easily be added without affecting the volume format or content of the object's main data stream.

There have also been attempts to speed up the "search by content" feature of some operating systems. Such a feature typically extracts and indexes human decipherable data from files/objects and searches this index instead of crawling through every file on disk (a very time consuming operation). The most common method is to periodically index files on disk and store these indexes in separate files. This method has two main disadvantages: upon each run the system must scan every object on disk to check if it has changed or been removed and the database is always out of date. If the index (or some information relating to) was stored along with the object on disk and the appropriate programming hooks were available, it would be possible to create a list of updated objects for the "search by content" software to look at in addition to its existing indexes.

Having multiple data streams for an object can also greatly simplify the implementation of extra features such as revision tracking (Section 4.6). The Reiser (2002-2003) system is using plugins to provide similar functionality and the HFS Plus volume specification does allow for named forks other than the traditional MacOS Resource and Data forks although these are not currently used by much software.

It is because of these points that we have decided to allow objects to have an arbitrary number of data streams (forks) associated with them. It is important not to confuse our usage of the word fork with the MacOS concept of a Resource Fork, we use fork to mean

an arbitrary length string of bytes. In a system such as Walnut, one fork could contain the content of the object and another the Walnut specific header information. If our storage system were to be geared towards a UNIX like environment, one fork could contain the meta-data (usually contained in the i-node) and another fork could hold the content of the file. A MacOS implementation could use one fork as the MacOS Resource Fork and the other as the Data Fork. It is clear that this design choice allows lots of flexibility and potential compatibility with other systems.

With various systems supporting various types and amounts of meta-data, the line between data and meta-data can become very blurred. We argue that the difference can purely be a matter of perspective. To a Word Processor application, the text and formatting information is the data it cares about while the icon and file name are auxiliary information, meta-data. However, for the same word processor file, but to a file manager style application, the icon and file name is the data it cares about and the text and formatting information can be viewed as auxiliary information (perhaps used in generating a preview or summary of the document).

We take the view that "one persons' data is anothers' meta-data" and believe that the object store should be designed to efficiently support this principal. Our initial design was to support the one o-node to many meta-data streams (forks) with one of these being the "data" fork (Figure 4.4). To help improve this design, allowing for inter-object meta-data relationships to be realised, the idea of allowing a meta-data fork to be a link (capability) to another is thought to aid in realising this (Figure 4.5).

A real world example could be an application object having a link to its manual page, documentation and to the object which describes the complete installation of the application. The behaviour of these links would be much how symbolic links are handled in a UNIX like environment. Although not currently implemented, the capability structure could easily fit within the o-node block, making implementation trivial as well as rather efficient.

4.5.3 Access control to meta-data

If the Walnut model were to be expanded to allow objects to have an arbitrary number of meta-data forks, there would have to be a method of referencing these forks and a good method of access control.

One simple method is to expand the password-capability structure to accommodate a `fork_type` field (which would match with that in the `fcfs_fork` structure). The master capability would provide access to all forks, and restrictions on which forks could be accessed could be added to the capability mechanism.

Another method could involve a separate master capability for each meta-data fork. Each of these methods as advantages and disadvantages and further work is needed to come up with a good system.

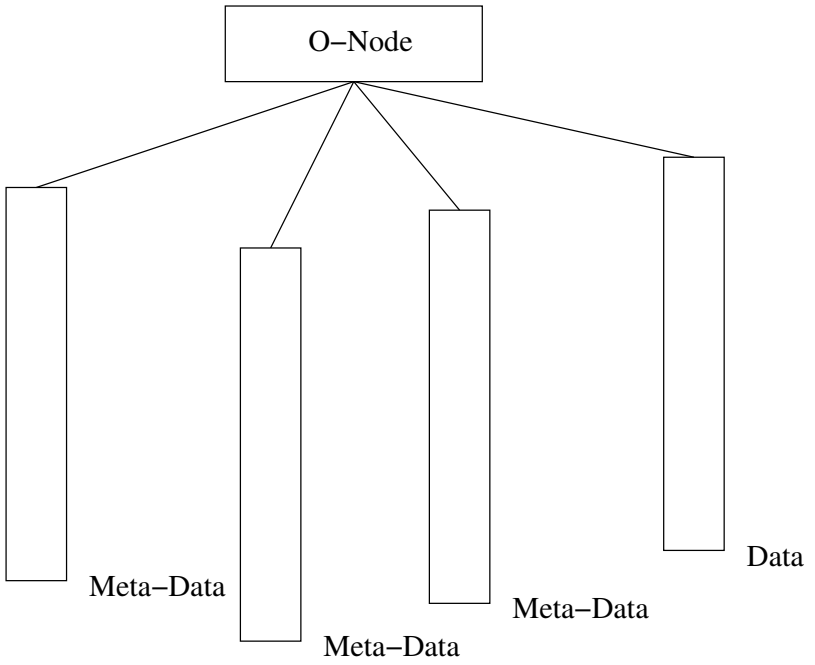


Figure 4.4: Meta-Data Design 1

4.6 Revision Tracking

We make the assumption that the most recent version of an object is going to be the most commonly accessed version and make optimisations for this case. Other revision control systems such as RCS(Tichy, 1985) also make this assumption, and this is consistent with the use of systems that do not track revisions as on these systems the most recent version is the **only** version available.

To be able to access older revisions we either need to store the old revision in full or store a set of operations to be used to reconstruct it. The difference between two revisions is called a **delta**. If a delta, when applied to a new revision, constructs an old revision, it is termed a **reverse delta** as it reverses the effects of time. A delta which makes transforms an old revision into a more recent one is known as a **forward delta**. If we make the assumption that the most recent version will be the version most frequently accessed, then it makes sense to store this version on-disk so that the revision tracking adds no overhead. We can then store a series of reverse deltas on this version to make it possible to reconstruct any previous revision.

To maintain consistency, the writing of deltas and changes to objects should occur in a specific order:

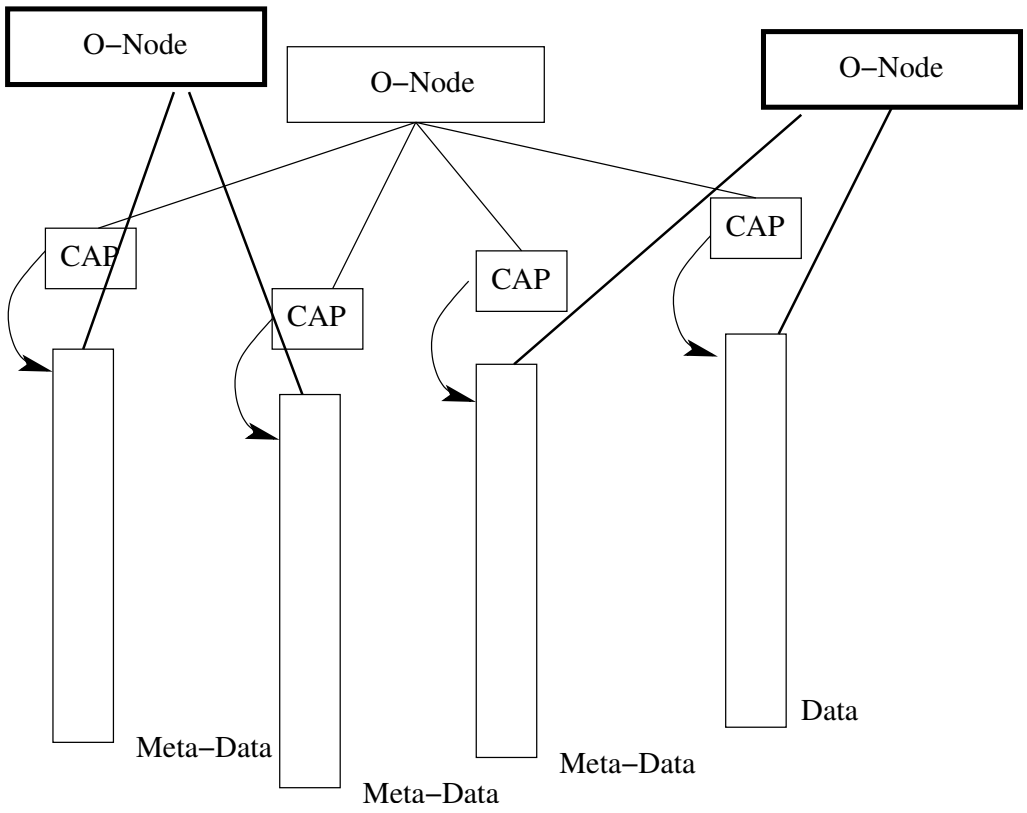


Figure 4.5: Meta-Data Design 2

- Firstly, the information about the reverse delta should be written to disk. (e.g. append data, replace data)
- Secondly, the data part of the delta (what data to append or replace) should be written to disk
- Finally, the "current" revision on disk should be updated to reflect the (new) current revision.

If any of these steps does not complete successfully it is possible to reverse previous steps so that the object is in its previous (consistent) state. This gives us the effect of data journaling - a guarantee that the content of an object is consistent after a crash. This is essential in a Walnut system where processes are persistent and any damage to a process object could yield undesirable, even catastrophic results.

The concept of a "stable" or `consistent_revision` has been introduced to allow the partial writing of deltas in case of memory pressures. It also allows the delta between two revisions to contain more than one operation using simple data structures. Only when all

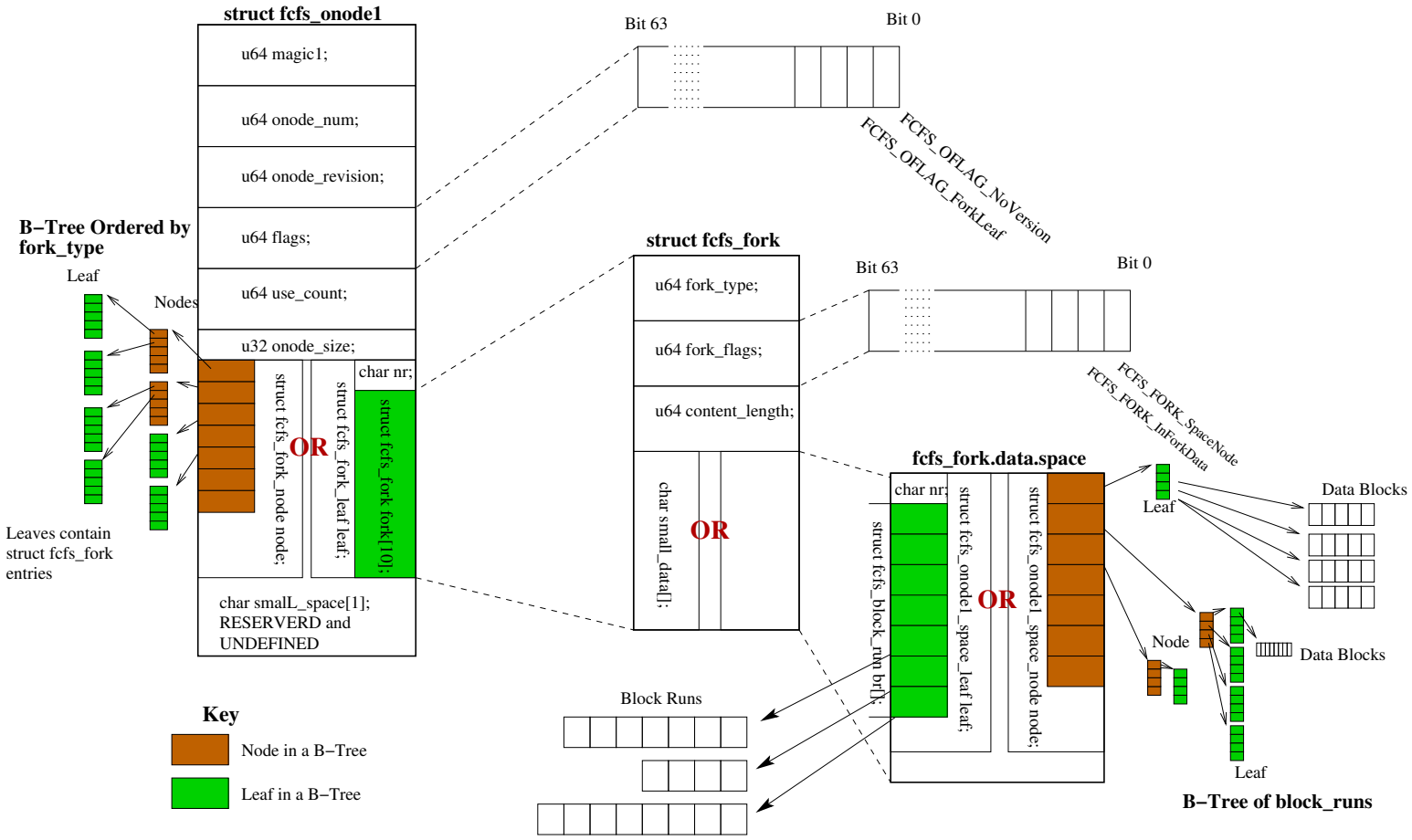


Figure 4.6: FCFS O-Node layout

the operations involved with a change in version have been written to disk shall a revision be marked as consistent. In low memory situations, it is possible to write operations out to disk without compromising the consistency of an object. Figure 4.7 shows how previous revisions are reconstructed by applying reverse deltas and how deltas may be applied upon recovery to revert the object to a consistent state.

Instead of adding complexity to the o-node structure to support revision tracking, it was decided to track the revisions to each fork separately using two other forks to store the revision information. This allows the flexibility of having forks which aren't under revision control should the need arise as well as an implementation of revision tracking using existing, predesigned and simple infrastructure. One fork is a list of a simple data structure detailing the operation being performed (See Section 5.7.1), while the other is data which may be required to complete the operation (e.g. an INSERT operation with the data to insert). This is illustrated in Figure 4.8 for the tracking of one fork in an o-node.

To allow the retrieval of multiple forks from a specific time to get a view of an entire o-node as it was in the past, the `time` field from the `onode_fork_rev` structure (Listing 5.13) can be used to synchronise revision numbers of the different forks. This method may prove unworkable due to the reliance on a working system clock and methods involving either adding a separate field to the `onode_fork_rev` structure or making the revision number global across the o-node, possibly with a relationship to the o-node revision field may need to be explored.

The choice to not support concepts such as branching in the revision tracking system is a conscious one as the added complexity for implementation and especially for users having to reference branches is considered too high for this initial design. There is no reason why extensions could be made to support branching, perhaps in the form of a **clone** operation which preserves a link to the revision of the cloned object as part of the new object's revision history.

4.6.1 Temporal Windowing

Tracking revisions to objects is of little use if a mechanism of accessing previous revisions is not provided. In providing such a mechanism, careful attention must be made to ensure the security of the system. In the life of an object, its contents, size and access controls can be changed. Questions must be raised as to the validity of capabilities over time, and these are questions which are not easily answered. Several possible approaches include:

Current capabilities are valid for all revisions Advantages are that it is a simple approach to implement it has several large disadvantages. The simple process of "remove the confidential information and make it publicly accessible" becomes problematic and error prone.

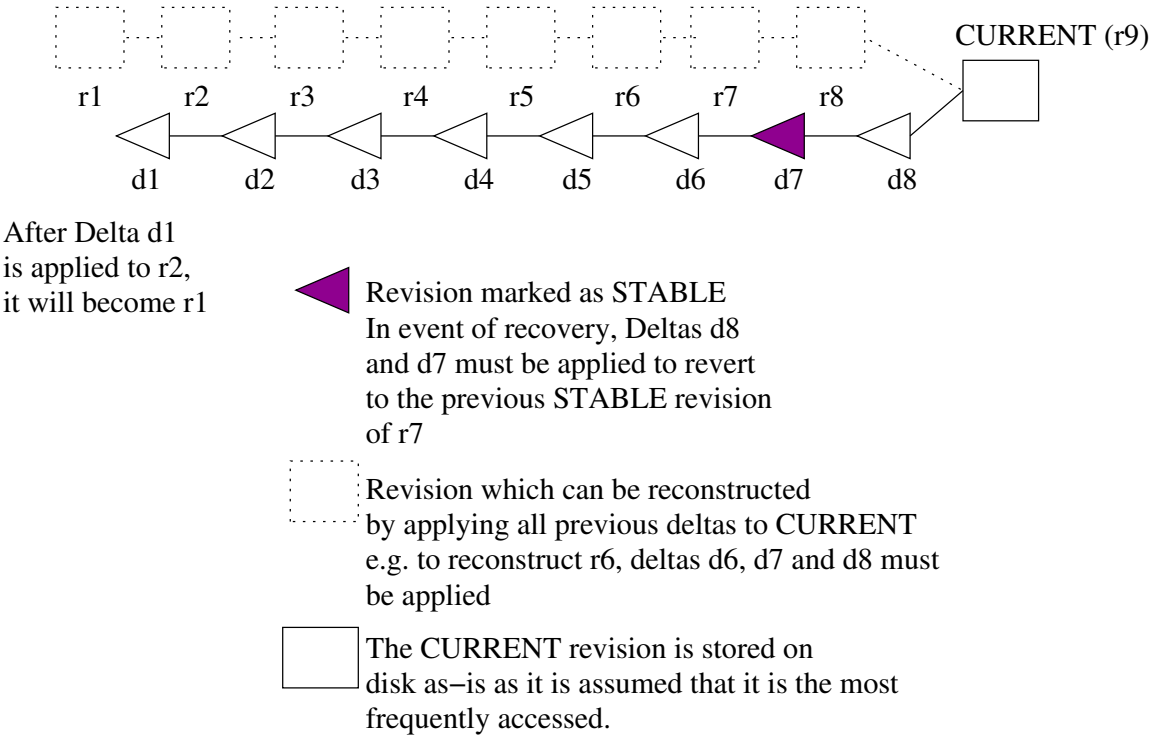


Figure 4.7: Revision Tracking

Previous revisions only accessible to the holder of the master capability This would mean that you must have all access rights to the object before you can retrieve a previous revision. This may prove limiting if you wish to grant others similar rights with different capabilities.

Revision information is accessible only by a separate master capability Provides a large amount of flexibility, but removes the elegance of a single master capability.

An as yet undetermined way to extend capabilities into the fourth dimension This is probably going to yield the best results, but requires much further research.

The correct way to control access to objects over time is unclear. The few simple methods suggested here hold some merit in certain scenarios but do not stand out as a universal solution. Further research into methods to control access to objects over time is needed, possibly with the aim of extending the password capability model to cover temporal windowing.

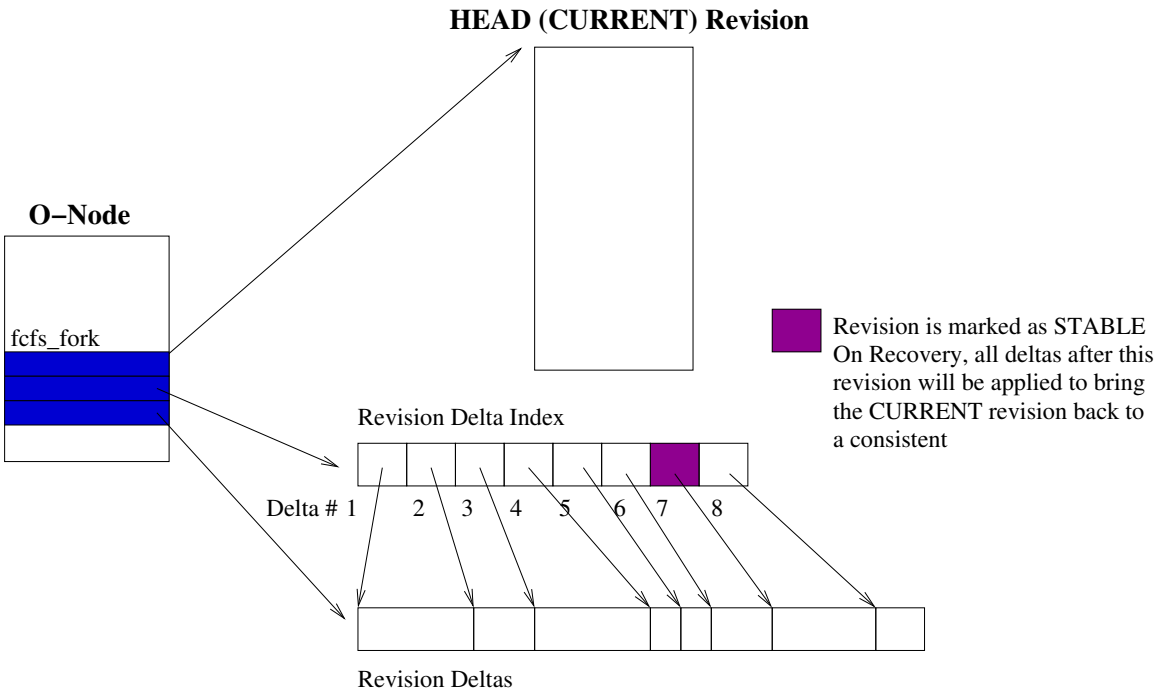


Figure 4.8: Revision Tracking - On Disk

4.7 Object Indexing

The function of the object catalogue is to translate object ids into disk blocks. There are several approaches to this with various advantages and disadvantages to each.

4.7.1 Hashing Object IDs

The approach which the current Walnut implementation takes is that a hash function is applied to the object id which gives us the block number of the object header. This means that object lookup is an $O(1)$ operation, which is a good thing but the disadvantages of this mechanism outweigh this single constant time operation.

The resize operation on a volume using this scheme is extremely expensive as (most likely) all object headers would need to be moved. Worst case scenario, data blocks would also have to be moved so that the headers could be put in the correct locations.

Another major disadvantage is that it limits what object ids can be stored on a disk. Since in Walnut part of the security of the system is based on the object id, this method reduces the overall security of the system. One possible work around would be to have a separate Walnut object id to on-disk object id index, but this would mean that we loose the advantage of the $O(1)$ lookup operation that a hash provides.

4.7.2 Sparse File approach

Sparse objects are objects where blocks are not allocated for unwritten blocks. If there was an object where the first 10,000 blocks where never written to and it was sparse, these blocks would not be allocated on disk but simply marked as empty in the block run structure (see Section 5.3.1 for details). Sparse objects are extremely useful for efficiently storing sparsely populated arrays, which is exactly what a catalogue of object ids is.

With this kind of setup (maybe in addition to hashing the object ids before placement into the sparse object) the limiting factor becomes how efficiently the system can compute offsets in a sparse object. Space efficiency could also be an issue as the most obvious way to implement sparse objects is on a block-by-block basis. This means that a write of a single value (the block number of the object header) will mean the allocation of a complete disk block, potentially wasting lots of disk space if the keys chosen are evenly spaced inside the address space (See Figure 4.9). It is because of this potential wastage of space that we chose not to use this method.

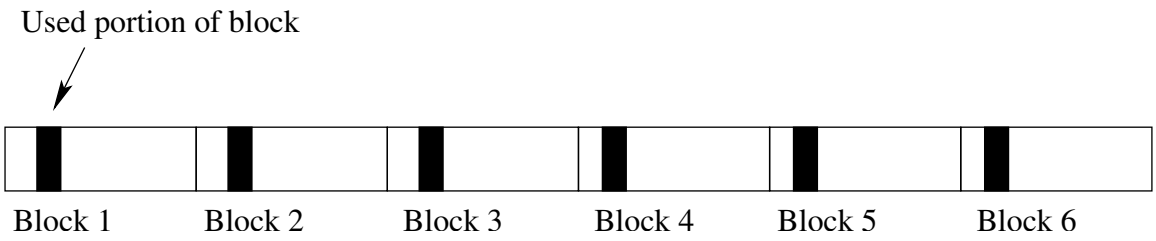


Figure 4.9: Sparse object index wasted space

4.7.3 B+Tree approach

The use of B-Trees as indexes on file systems has been commonplace for many years and is one of the most common methods of storing indices. This is because of the relatively cheap insert and lookup operations as well as the data structures being easily adaptable and efficient for large page based storage (such as disks). It is because of the good lookup performance that we have chosen to use a (slight variant) on the standard B+Tree for the object index.

B+Trees are chosen over other forms of B-Trees as the property of B+Trees having all data in the leaves leads to higher disk cache hits during lookup. This is because during key lookup we are more likely to want the branching information from the internal nodes than the value (many branches are needed, but only one value). Thus it is better to already have as many branch information (which is what all the data inside a B+Trees internal node is) in memory than lots of values (from the key-value pair).

Although algorithms exist for deletion from B+Trees(Jannink, 1995), they can be either expensive (computationally or IO) or reduce the ability of parallel access to the B+Tree.

Although with the planned support of revision tracking (Section 4.6), efficiency of deletion may not matter as the deletion operation would happen very rarely.

Loosely balanced B+Tree Approach

Here we define a "loosely balanced B+Tree" as a B+Tree where we are a little bit more relaxed about balancing during insert operations. Since we can select the keys for an object index, a pseudo-random number generator will generate numbers in a roughly equal distribution. This means that the splitting of nodes should be fairly limited operation especially if we do not mind that a single split makes the tree one level deeper in a single area. The logic being that the random selection of keys will even up the height of the tree over time as more keys are inserted.

The goal of the o-node index is to translate o-node ids to disk blocks so that the o-node may then be read from disk. The structures are designed so that they are easily cachable and will allow a the largest number of cache hits per cache size.

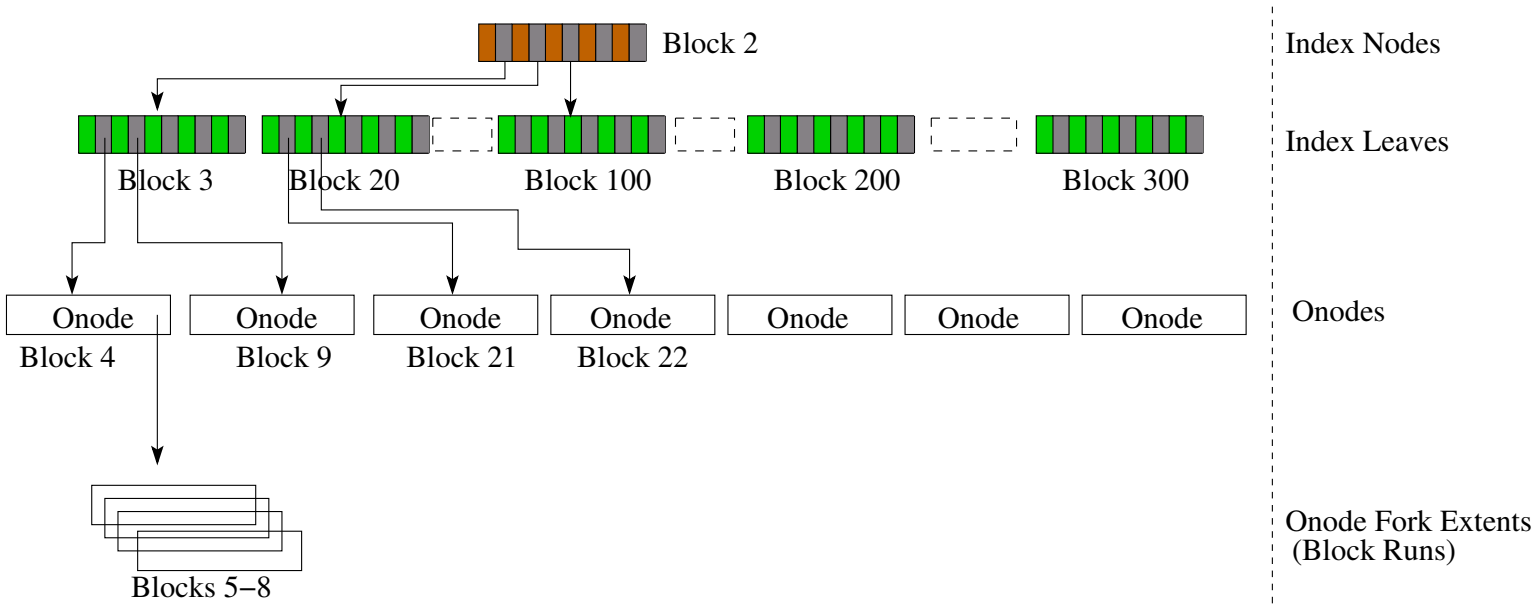


Figure 4.10: FCFS O-Node Index Tree

Chapter 5

Implementation of a Walnut Object Store

This chapter provides extra detail on some of the design decisions in Chapter 4. Information presented in this chapter has a focus on implementation rather than overall design. Detailed explanation as to the function, content of, validity of values and reasons behind data structures and their fields are presented, along with behaviour required by software accessing the volume directly. Where enough information and reasoning was presented in Chapter 4, it has not been repeated here.

5.1 General Implementation Notes

5.1.1 Endianness

All on disk data structures will be kept in Big Endian format. The benchmarks performed by SGI during the development of XFS((tes@sgi.com), 2002) show a minimal performance impact from this decision.

The current implementation has been tested on big endian architectures (such as PowerPC) and limited testing shows that the implementation is also functional on little endian systems (such as i386). Currently the volumes produced on one system are incompatible with code on another. Implementation and testing of endian conversion was not considered unique enough to be part of this project. In the future it may be desirable to provide support for little endian on disk formats, but there is little motivation to do so.

5.2 Super Block

The main copy of the super block data structure (Listing 5.1) is stored at the start of the first block of the volume. If as part of its boot requirements, a system needs space in the first block of a disk partition, the super block may appear in the second block of the partition.

5.2.1 Superblock fields

magic1 The first magic number in the superblock, must be set to `FCFS_SB_MAGIC1` (0x46436673).

version The version of the volume format used on this volume. It is left to the software to determine if it can adequately access a volume with this version number. This allows software to gracefully return an error in the case a future revision of the volume format is not backwards compatible.

bits The base number of bits for file system fields. This is currently used only as a sanity check (the value should be 64) but in the future may be used to indicate an extension of the volume format which only differs in field size. It may be useful in embedded systems to save space by only using a maximum of 32 bits throughout the object store. See the more detailed description of the `FCFS_SB_BITS` constant in Section 5.2.2.

sblength Length of the super block (in bytes). This is set to `sizeof(struct fcfs_sb)` for the specific version of the volume format. This is used as a sanity check.

flags Various global settings for the volume such as if it was cleanly unmounted, if journaling is enabled or if revisions to objects should be recorded. Flags are further defined in Section 5.2.3 and the enum structure in Listing 5.3.

bsize The block size of the volume (in bytes). This field (along with the fields: `blocksnr`, `allocation_groupsnr` and `ag_blocksnr`) can be used to check the sanity of the volume geometry as reported by any partitioning or block device software.

blocksnr The number of blocks (of size `bsize`) in the volume.

name A user defined null terminated ASCII string that is a user chosen, (theoretically) human readable name for the volume up to `FCFS_NAME_LENGTH` characters (including the terminating NULL). This field should be UTF-8 but current code only deals with ASCII and this should be considered a bug in the current implementation.

magic2 The value of `FCFS_SB_MAGIC2` (0x3f8ec2a1). This value has no special or hidden meaning, it's just strange.

allocation_groupsnr Number of alloction groups on volume.

ag_blocksnr Blocks per allocation group (except the last one which may be smaller).

Listing 5.1: struct fcfs_sb

```
1  struct fcfs_sb {
2    u32 magic1;     // first magic number.
3    u32 version;    // version of FS
4    u32 bits;       // number of bits to use as a 'base' bits. (64
       default)
5    u32 sblength;   // length of sb (bytes)
6
7    u64 flags;
8
9    u32 bsize;      // block size (bytes)
10   u64 blocksnr;   // number of blocks
11
12
13   char name[FCFS_NAME_LENGTH];   // human readable name of volume
14   u32 magic2;     // magic2
15
16   u32 allocation_groupsnr;       /* Number of allocation groups on
       volume */
17   u32 ag_blocksnr;               /* Blocks per allocation group */
18
19   u64 num_mounts;                /* Number of successful read/
       write mounts */
20   u64 num_dirtymounts;           /* Num dirty mounts */
21   u64 time_created;              /* Time volume created*/
22   u64 time_clean;                /* Time volume last unmounted
       cleanly*/
23   u64 time_dirtymount;           /* Time volume last mounted dirty
       */
24
25   u64 onode_index_blocknr;       /* Where we find our primary
       index */
26
27   u64 onindex_num_onodes;        /* Num onodes in index */
28   u64 onindex_used;              /* Num of used inodes in index */
29   u64 onindex_free;              /* Num of free onodes in index */
30   u64 onindex_next_id;           /* Next ID to use */
31   u32 onindex_node_size;         /* In number of keys */
32
33   union space_tracking {
34
35     char padding[50];
36   }space;
37 };
```

num_mounts Records the number of successful read/write mounts where the volume was cleanly unmounted prior to mounting. Read only mounts are not recorded due to their "read only" nature. In case of overflow, the value should wrap around to zero.

num_dirtymounts Records the number of successful times the volume has been mounted after having been uncleanly unmounted (i.e. with the dirty flag set). Read only mounting of an unclean volume will not change this field and in itself should be strongly discouraged.

time_created Set by the volume creation utility, this is the time volume created. The value is seconds since the Epoch (00:00:00 UTC, January 1, 1970).

time_clean Time volume last unmounted cleanly, in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

time_dirtymount Time the volume was last successfully mounted after an unclean unmount, in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

onode_index_blocknr The block number of the root node of the o-node index.

onindex_num_onodes The number of o-nodes in the o-node index.

onindex_used The number of used entries in o-node index leaves.

onindex_free The number of unused entries in o-node index leaves.

onindex_next_id Currently unused in any implementation but the aim is to restore the seed of the o-node id generator.

onindex_node_size The number of keys inside an o-node index node.

Padding Reserved for future use, should be created blank (filled with null bytes) by the creation utility and unmodified by any volume manipulation software.

5.2.2 Superblock Constants

The constants (Listing 5.2) for this revision of FCFS include two magic numbers, where the second one (`FCFS_SB_MAGIC2`) may change with major revisions of the system, indicating total incompatibility with previous code. This is to allow software to gracefully return an error indicating that although the volume appears to be valid, more recent software is required to access it. The `FCFS_SB_MAGIC1` value will remain constant across major revisions of the on disk format. The `FCFS_SB_VERSION1` field shall change in future incompatible revisions of the storage system. The `FCFS_SB_BITS` field indicates the base number of bits used in on disk data structures. Currently the system is designed to be 64 bits but this field enables a future revision of the code to scale up (or down) to other sizes. It is currently only used as a sanity check. The `FCFS_NAME_LENGTH` value is a #define purely for convenience.

5.2.3 FCFS Super Block Flags

The flags field in the super block (Listing 5.1) is populated by setting the bits as per the enum in Listing 5.3.

FCFS_FLAG_SBLocation1, FCFS_FLAG_SBLocation2, FCFS_FLAG_SBLocation3 Are defined according to the `enum fcfs_sb_location` (Listing 5.4) to indicate where this copy of the superblock is (or at least should be) on disk. This can be used to aid in volume recovery where the volume geometry is unknown.

FCFS_FLAG_Dirty When set, the volume was not cleanly unmounted and requires a consistency check or journal replay. We do not currently do nothing with this flag apart from warn the user.

FCFS_FLAG_Experimental Any code that is of an experimental nature (e.g. in active development and debugging of the volume format) should set this flag on a volume. Once set, this flag may **never** be unset, not even after a complete consistency check of the system.

Upon mounting of a volume with this flag set, the user should be warned that they are playing with fire and are going to be in trouble. This flag should be thought of as a "don't come whining to us when all your data suddenly vanishes" flag.

FCFS_FLAG_JournalMeta When set, the volume journals meta data. This flag is a placeholder as no implementation or specification of a journal format exists.

FCFS_FLAG_JournalData When set, the volume journals object data. This flag is a placeholder as no implementation or specification of a journal format exists.

FCFS_FLAG_Versioned When set, all revisions to all forks of all objects are tracked unless they explicitly state otherwise (or are the forks used by the revision tracking system).

5.3 Block Run

The `fcfs_block_run` structure (Listing 5.5) is capable of addressing 2^{32} blocks (via the `len` field), which may be up to 2^{32} bytes in size. In reality, no block size is ever going to be 2^{32} bytes long, a more typical size is 4kb (4096 bytes) as this fits in nicely with the IA32 processor family's page size. With this block size, a single `fcfs_block_run` structure will address a maximum of 2^{44} bytes of data. Although the probability of an object ever needing to have several `fcfs_block_run`s of this length, the alternative of having a 16 bit length field seemed overly limiting when considering the rate at which disk and file sizes have been increasing over time.

Listing 5.2: FCFS Super Block Defines

```
1  #define FCFS_SB_MAGIC1    0x46436673   /* FCFS */
2  #define FCFS_SB_VERSION1  0x00010001   /* V1 SB, V1 FS */
3  #define FCFS_SB_BITS      0x00000040   /* We're 64 bit - but could
       be bigger...*/
4
5  #define FCFS_NAME_LENGTH  128          /* Human Readable Name */
6  #define FCFS_SB_MAGIC2    0x3f8ec2a1   /* There is no meaning, just
       weird num */
```

Listing 5.3: enum fcfs_sb_flags

```
1  enum fcfs_sb_flags {
2    FCFS_FLAG_SBLocation1,          /* Three flags make number */
3    FCFS_FLAG_SBLocation2,          /* see: enum fcfs_sb_location */
4    FCFS_FLAG_SBLocation3,
5    FCFS_FLAG_Dirty,                /* Was uncleanly unmounted*/
6    FCFS_FLAG_Experimental,         /* Has been written by
         experimental code */
7    FCFS_FLAG_JournalMeta,          /* Journal meta-data */
8    FCFS_FLAG_JournalData,          /* Journal all data writes*/
9    FCFS_FLAG_Versioned,            /* Is versioned system */
10 };
```

Listing 5.4: enum fcfs_sb_location

```
1  enum fcfs_sb_location {
2    FCFS_SBloc_start_volume,        /* Start of volume */
3    FCFS_SBloc_start_ag,            /* Start of allocation group */
4    FCFS_SBloc_end_volume           /* Final block of volume */
5  };
```

The `allocation_group` and `start` fields can represent any of the 2^{64} possible blocks on a volume. Although the `fcfs_block_run` structure s the major cause of the restriction of the number of blocks in an `allocation_group`, the 2^{32} limit is not viewed as overly restrictive.

The alternative of having an absolute, 64bit block number and a 32 bit length field was considered, but rejected as this removes our concept of allocation groups which have proven to be very useful for parallelism in other systems such as XFS.

The BeFS block run (Figure 2.8) had the advantage of being 64 bits long, the same as an absolute block number on a 64 bit system. Although we use an extra 32 bits per block run, the advantage of

Listing 5.5: struct fcfs_block_run

```
1 struct fcfs_block_run {
2   u32 allocation_group;          /* Allocation group */
3   u32 start;                     /* Start Block */
4   u32 len;                       /* Length (blocks) */
5 };
```

5.3.1 Special cases

A block run with an `allocation_group` of zero and a `start` value of zero should be considered part of a sparse object. That is, when read, `NULL` values are returned and when written to, disk space is allocated for that block.

The other special case for a `fcfs_block_run` is where part of the disk has gone bad and can no longer be referenced. In this case, the `allocation_group` will be `0xFFFFFFFF` and the `start` block will be `0xFFFFFFFF`. This will be set in the case where blocks previously occupied were found to be defective or when an object was copied from a volume where part of the media was defective. It is left up to the host system the exact behaviour of relaying this message to users - if there is a warning on initial access to the object or if there are errors when accessing these parts of the object. These values should also be used for when part of a `fcfs_block_run` was found to be corrupted.

5.3.2 Corruption

A `fcfs_block_run` is corrupt when:

- The `allocation_group` is larger than the `allocation_groupsnr` value in the super block and is not zero (for sparse) or `0xFFFFFFFF` (for was corrupt)
- The `start` value is larger than the `ag_blocksnr` field in the super block or in the last allocation group, the `start` value is greater than $blocksnr - (allocation_groupsnr \times ag_blocksnr)$ (all values stored in the super block)

- The *start* + *len* values (from the `fcfs_block_run` structure) are greater than the ag_blocksnr value from the super block, or in the case of the last allocation group, greater than $blocksnr - (allocation_groupsnr \times ag_blocksnr)$ (all values stored in the super block)

5.4 O-Node

5.4.1 O-Node fields

The purpose of each field in the `fcfs_onode1` structure (Listing 5.6) is:

magic1 Identifies this disk block as containing an O-Node by storing the value of `FCFS_ONODE_MAGIC1`, which is the 64 bit number `0x4f4e6f4465563101` (the first seven bytes of which represent the ASCII string "ONoDeV1"). This can aid in volume recovery and debugging by being easily identifiable and uncommon to be at the beginning of a disk block.

onode_num A 64 bit identifier for the o-node which is unique to this volume. May be allocated in any way, but should be tied in with the implementation of the o-node index as this will allow close optimisation of the o-node index. This value will commonly be referred to as the o-node Id.

onode_revision Incremented each time the o-node is updated. This may be useful in networked environments when trying to determine if a cached copy of the o-node is current. This field may also be useful in volume recovery if an o-node has been relocated on disk, the one with the higher revision will be more recent.

flags See section 5.4.2

use_count Number of times this o-node is referenced in indices. Future revisions of the volume format may choose to support more than one o-node index using data other than the Id as the key. Such indices could be useful in indexing meta-data. When the o-node is deleted from the main index, this should drop to zero and the o-node should be written to disk. This will aid any undelete or volume recovery utility in locating deleted o-nodes. Although, with revision tracking (Section 4.6) it is doubtful if the removal of an o-node will ever happen.

onode_size The size (in bytes) of the o-node structure. Mainly used as a sanity check but could also be used in a consistency checker or volume recovery tool to help identify an o-node.

5.4.2 O-Node Flags

FCFS_OFLAG_NoVersion When set, no fork in the object should have its revisions tracked. This is a global override and is intended for debugging purposes only as the revision tracking code should never be run (for this o-node) when this flag is set.

FCFS_OFLAG_ForkLeaf The `forks` union part of the o-node structure contains a `fcfs_fork_leaf` structure and not a `fcfs_fork_node`. This should be the case when there are 10 or less forks as the `fcfs_fork_leaf` structure (Listing 5.8) can hold up to 10 forks.

5.5 Forks

Fork Node

The `fcfs_fork_node` structure (Listing 5.7) is a node in the B+Tree of forks belonging to an o-node. To support more than one level of nodes (the one in the o-node), a magic number will need to be added to the data structure so that a block may be easily identified as a node or leaf. Due to time constraints, this is left as an exercise for a future revision.

offset the maximum `fork_type` in the subtree. Zero if there is no subtree.

block the block number of the node or leaf. Zero if there is no subtree.

Listing 5.7: O-node1 fork node

```
1 struct fcfs_fork_node {
2   u64 offset[7];
3   u64 block[7];
4 };
```

Fork Leaf

The `fcfs_fork_leaf` structure (Listing 5.8) contains `nr` number of `fcfs_fork` structures up to a maximum of 10. In future revisions, this limit should be based on the size of the block or space in the o-node and not the arbitrary constant of 10.

Listing 5.8: O-node1 fork leaf

```
1 struct fcfs_fork_leaf {
2   char nr;
3   struct fcfs_fork fork[10];
4 };
```

Listing 5.6: O-node Revision 1

```
1  #define FCFS_ONODE_MAGIC1  0x4f4e6f4465563101ULL          /*
       ONoDeV1 0x01 */
2
3  enum fcfs_onode_flags {
4    FCFS_OFLAG_NoVersion,          /* Don't version track this onode
         */
5    FCFS_OFLAG_ForkLeaf            /* We have a fork leaf, not a
       node */
6  };
7
8  struct fcfs_onode1 {
9    u64 magic1;                    /* Identify as O-Node */
10   u64 onode_num;                 /* FS Specific Unique ID */
11   u64 onode_revision;            /* Revision of onode */
12   u64 flags;                     /* fcfs_onode_flags */
13   u64 use_count;                 /* Reference Counter (for indices
       ) */
14   u32 onode_size;                /* Length of o-node structure. */
15   union forks {
16     struct fcfs_fork_leaf leaf;
17     struct fcfs_fork_node node;
18   } forks;
19   char small_space[1];           /* used for in-onode data (or
       metadata) */
20 };
```

Fork

The `fcfs_fork` data structure (Listing 5.9) contains the following fields:

fork_type What type of data the fork contains. Systems may define their own values for specific types of data (e.g. Resource fork, i-node equivalent data) and may choose to use an external index to map `fork_type` values to human readable names.

fork_flags See Section 5.5

content_length Length (in bytes) of the data stored in this fork.

Fork Flags

The flags for a `fcfs_fork` structure (Listing 5.9) are:

FCFS_FORK_InForkData When set, the `data` union of the `fcfs_fork` structure contains `small_data` and not the `space` union.

FCFS_FORK_SpaceNode When set, the `space` union (which is part of the `data` union) in the `fcfs_fork` structure contains a `fcfs_onode1_space_node` and not a `fcfs_onode1_space_leaf`.

Listing 5.10: O-node1 fork flags

```
1 enum fcfs_fork_flags {
2   FCFS_FORK_InForkData,          /* We have data, not space nodes
      */
3   FCFS_FORK_SpaceNode            /* Space is node, not leaf */
4 };
```

5.6 O-Node Index

5.6.1 O-node Index Node

The node of the o-node index (Listing **??**) has the number of `struct fcfs_onode_index_node_items` in the `items` array as defined in the `onindex_node_size` field of the super block (See Section 5.2.1).

5.6.2 O-node Index Leaf

The O-Node Index Leaf (Listing 5.12) contains an ordered list of o-node IDs (the key) and the o-node block number (the value). The `block` field is the number of the disk block this structure is in and is used for sanity checking. The `used` field indicates how many items have been used in this leaf.

Listing 5.9: O-node1 fork

```
1  #define FCFS_FORK_SMALL_DATA_SIZE 112
2
3  struct fcfs_fork {
4    u64 fork_type;
5    u64 fork_flags;
6    u64 content_length;
7    union data {
8      union space {
9        struct fcfs_onode1_space_leaf leaf;
10       struct fcfs_onode1_space_node node;
11     } space;
12
13     char small_data[FCFS_FORK_SMALL_DATA_SIZE]; /* I wish this
           was neater, as part of the leaf */
14   } data ;
15 };
```

Listing 5.11: O-node Index Node

```
1  #define FCFS_ONODE_INDEX_NODE_MAGIC1 0x4f6e4944784e4445ULL  /*
       OnIDxNDE */
2
3  struct fcfs_onode_index_node_item {
4    u64 key;
5    u64 node_blocknr;
6  };
7
8  struct fcfs_onode_index_node {
9    u64 magic1;                    /* FCFS_ONODE_INDEX_NODE_MAGIC1
         */
10   u64 id;                        /*Not necessarily unique, used
         for locking */
11   u64 block;
12   u64 used;
13   struct fcfs_onode_index_node_item items[1];
14 };
```

5.7 Revision Tracking

5.7.1 Revision Information fork

Is a series of `struct fcfs_onode_fork_rev` structures (Listing 5.13). When a new operation is written to disk, a `fcfs_onode_fork_rev` is appended to the revision information fork. Once written, no editing of these structures should be performed.

Revision Information Fork Fields

revision Revision that this operation relates to.

consistent_revision The last consistent revision written to disk.

time The time this revision was created. Note that this is **not** the time this revision was committed.

operation The operation this structure describes. See Section 5.7.1 and Listing 5.14.

magic1 `FCFS_ONODE_FORK_REV_MAGIC1`

rev_offset Offset in revision data fork for data to use in the `operation`.

rev_length Length of the data in the revision data fork.

offset Offset in the fork to apply the operation to.

space Reserved for future use.

Revision Operations

Currently, the operations supported (Listing 5.14) are:

FCFS_ONODE_FORK_REV_REPLACE Replace `rev_length` bytes starting at `offset` with `rev_length` bytes starting from `rev_offset` in the Revision Data Fork (Section **??**)

FCFS_ONODE_FORK_REV_INSERT Insert `rev_length` bytes at `offset` with `rev_length` bytes starting from `rev_offset` in the Revision Data Fork (Section **??**)

FCFS_ONODE_FORE_REV_APPEND Append `rev_length` bytes starting from `rev_offset` in the Revision Data Fork (Section **??**)

FCFS_ONODE_FORK_REV_TRUNCATE Truncate `rev_length` bytes starting from `offset`. The term truncate is used but the structure of the command could also accommodate the deletion of a length of bytes.

Listing 5.12: O-node Index Leaf

```
1  #define FCFS_ONODE_INDEX_LEAF_MAGIC1 0x4f6e4944784c6665ULL  /*
       OnIDxLfe */
2
3  struct fcfs_onode_index_leaf_item {
4    u64 key;
5    u64 onode_blocknr;
6  };
7
8  struct fcfs_onode_index_leaf {
9    u64 magic1;                    /* FCFS_ONODE_INDEX_LEAF_MAGIC1
         */
10   u64 id;                        /* Not necessarily unique, used
         for locking */
11   u64 block;
12   u64 used;
13   struct fcfs_onode_index_leaf_item items[1];
14 };
```

Listing 5.13: O-node fork Revision Information Fork

```
1  #define FCFS_ONODE_FORK_REV_MAGIC1 0xF0526B76   /* 0xF0 'R' 'k' '
       v' */
2
3  struct fcfs_onode_fork_rev {
4    u64 revision;                  /* Increment for each revision */
5    u64 consistent_revision;       /* Used to note last consistent
         revision */
6    u64 time;                      /* Time revision was done (not
         committed) */
7    u32 operation;                 /* enum fcfs_onode_fork_rev_op */
8    u32 magic1;                    /* FCFS_ONODE_FORK_REV_MAGIC1 */
9    u64 rev_offset;                /* Offset in revision fork */
10   u64 rev_length;                /* Length of revision fork data
         */
11   u64 offset;                    /* Offset in real fork to apply
         to*/
12   char space[16];                /* Space for future revisions */
13 };
```

FCFS_ONODE_FORK_REV_GROW fill `rev_length` bytes with null bytes at `offset`.

5.7.2 Revision Data Fork

The revision data fork contains the data referenced by the Revision Information Fork (Section 5.7.1). This fork is only ever appended to or read from, no other modifications should ever be made.

Listing 5.14: O-node fork Revision Information Fork Operations

```
1 enum fcfs_onode_fork_rev_op {
2     FCFS_ONODE_FORK_REV_REPLACE,
3     FCFS_ONODE_FORK_REV_INSERT,
4     FCFS_ONODE_FORE_REV_APPEND,
5     FCFS_ONODE_FORK_REV_TRUNCATE,
6     FCFS_ONODE_FORK_REV_GROW,
7 };
```

Chapter 6

Conclusion

The work in this thesis was motivated by the need to supply the Walnut kernel with a fast, efficient and robust object store. It has focused on solving problems specific to Walnut as well as more general problems faced by systems implementing new features as part of their object stores.

This thesis has presented the design and implementation details of an improved object store for the Walnut kernel. The design is based on proven techniques from other systems and strong theoretical and practical knowledge.

We identify a volume by a super block, a data structure containing information on where other structures are stored on disk and other statistics about the volume such as the date it was created and last used. Copies are kept on other parts of the disk in case the primary super block is damaged. This is a solid and proven method that is used widely among similar systems.

Although no direct structure exists for them, we separate the disk into allocation groups which allows for increased parallelism. We reference disk blocks by a block run structure, referred to by some systems as extents. It allows us to optimise for the optimal case where blocks are allocated in a contiguous block.

Free blocks are tracked in a block bitmap, implemented because it is relatively easy to code in a bug free manner. In the future, a set of B+Trees is viewed as the alternative that should be used so that allocating blocks based on locality or number of contiguous blocks is a simpler operation.

Objects are described by the o-node data structure which allows great flexibility and extensibility in the future. By having the o-node point to an arbitrary number of arbitrarily sized forks we are able to use these forks to implement additional features of the object store such as revision tracking on top of a well tested and simple base, avoiding needless complexity and possible bugs. For the existing Walnut model, an object would consist of two forks: one containing the dope and the other the body. Since we make optimisations

for small forks by storing them in the same disk block as the o-node (and the dope is a small structure), there will be little or no performance penalty over the current method of writing objects to disk.

Although not implemented, we have discussed various possible methods to have fine grained access control to meta-data. We introduce a method to support links to meta-data, providing an efficient way for relationships between objects to be realised and potentially optimised for.

By using B+Trees to reference data structures (such as forks) and in the o-node index we provide scalability to large data sets which makes our system competitive with the best of existing systems. Because we do not statically assign o-nodes to disk blocks, we can write objects to disk to optimise for locality dramatically improving performance over the existing Walnut object store.

We have discussed and implemented a basic form of revision tracking into the object store, providing a facility to ensure processes and their data objects are in a consistent state before they are resumed and providing real functionality to users which was previously only available with specialised software.

From examining the implications of implementing revision tracking we realised that there needs to be further study in the area of temporal windowing. That is, a method to securely provide access to previous revisions of objects.

Although our current implementation is only at a proof-of-concept stage, it has allowed us to test the design of the system and preliminary tests and code optimisations have only reinforced the theory that the system presented here is fast, efficient and reliable. There is no reason why future work could not advance the implementation to a stage where it could be used as a live object store for an operating system such as Walnut.

References

A Fast File System for UNIX (1984). University of California, Berkeley.

Anderson, C. (1993a). xfs attribute manager design, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Anderson, C. (1993b). xfs namespace manager design, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Anderson, C., Doucette, D., Glover, J., Hu, W., Nishimoto, M., Peck, G. and Sweeney, A. (1993). xfs project architecture, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Anderson, M., Pose, R. D. and Wallace, C. S. (1986). A Password-Capability system, *The Computer Journal* **1-8**(1).

Card, R., Ts'o, T. and Tweedie, S. (n.d.). Design and implementation of the second extended filesystem, *Proceedings of the First Dutch International Symposium on Linux*, number ISBN 90 367 0385 9, Laboratoire MASI — Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh. *http://web.mit.edu/tytso/www/linux/ext2intro.html

Castro, M. D. (1996). *The Walnut Kernel: A Password-Capability Based Operating System*, PhD thesis, Monash University.

Ceelen, C. (2002). Implementation of an orthogonally persistent l4 microkernel based system, *Technical report*, Universitat Karlsruhe. Supervisor: Cand. Scient and Espen Skoglund. *http://i30www.ira.uka.de/teaching/thesisdocuments/ceelen_study_studi.pdf

Crane, A. (1999). Ext2 undeletion, *Technical report*. *http://www.praeclarus.demon.co.uk/tech/e2-undel/

Doucette, D. (1993a). xfs kernel threads support, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Doucette, D. (1993b). xfs message system design, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Doucette, D. (1993c). xfs project description, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Doucette, D. (1993d). xfs simulation environment, *Technical report*, Silicon Graphics. *http://oss.sgi.com/projects/xfs/

Doucette, D. (1993e). xfs space manager design, *Technical report*, Silicon Graphics.

Elnozahy, E. N., Johnson, D. B. and Zwaenepoel, W. (1992). The performance of consistent checkpointing, *Proceedings of the 11th Symposium on Reliable Distributed Systems*, IEEE Computer Society, Houston, Texas 77251-1892, pp. 39–47. *http://www.cs.rice.edu/ dbj/ftp/srds92.ps

Folk, M. J. and Zoellick, B. (1987). *File Structures: A Conceptual Toolkit*, Addison-Wesley Publishing Company.

Giampaolo, D. (1999). *Practical FileSystem Design with the Be File System*, Morgan Kaufmann Publishers, Inc., chapter 1.

Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S. and Liedtke, J. (1998). The Mungi single-address-space operating system, *Software Practice and Experience* **28**(9): 901–928. *citeseer.nj.nec.com/heiser98mungi.html

Hitz, D., Lau, J. and Malcolm, M. (n.d.). File system design for an nfs file server. *http://www.netapp.com/tech_library/3002.html

III, G. G. R. and Singhal, M. (1993). Using logging and asynchronous checkpointing to implement recoverable distributed shared memory, *Technical report*, Department of Computer and Information Science, The Ohio State University. *http://camars.kaist.ac.kr/ yjkim/ftsdsm/richard93using.pdf

Inc., A. C. (n.d.). Hfs plus volume format, *Technical report*, Apple Computer Inc. Technical Note TN1150.

Inside Macintosh: More Macintosh Toolbox (1993). Addison-Wesley.

Jannink, J. (1995). Implementing deletion in B+-trees, *SIGMOD Record (ACM Special Interest Group on Management of Data)* **24**(1): 33–38. *citeseer.nj.nec.com/jannink95implementing.html

Janssens, B. and Fuchs, W. K. (1993). Relaxing consistency in recoverable distributed shared memory, *Technical report*, Center for Reliable and High Performance Computing Coordinated Science Laboratory, University of Illinois. *http://citeseer.nj.nec.com/rd/35597259%2C83458%2C1%2C0.25%2CDownload/http://citeseer.nj.ne

Kopp, C. (1996). *An i/o and stream inter-process communications library for password capability system*, Master's thesis, Department of Computer Science, Monash University.

Liedtke, J. (1993). A persistent system in real use: Experiences of the first 13 years, *German National Research Center for Computer Science* .
*http://citeseer.nj.nec.com/liedtke93persistent.html

MacDonald, J. P. (n.d.). File system support for delta compression, *Technical report*, University of California at Berkeley.

MacDonald, J., Reiser, H. and Zarochentcev, A. (2002). Reiser4 transaction design document, *Technical report*, Namesys.
*http://www.namesys.com/txn-doc.html

McKusick, M. K. and Ganger, G. R. (1999). Soft updates: A technique for eliminating most synchronous writes in the fast filesystem, *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, The USENIX Association.

Nishimoto, M. (1994). The log manager (xlm), *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Phillips, D. (2001). A directory index for ext2, *Technical report*.
*http://people.nl.linux.org/ phillips/htree/paper/htree.html

Reiser, H. (2001). Reiserfs v.3 whitepaper, *Technical report*, NameSys, 6979 Exeter Dr., Oakland, CA 94611-1625.
*http://www.namesys.com/content_table.html

Reiser, H. (2002-2003). Reiser4, *Technical report*, Namesys. Accessed: 23/4/2003.
*http://www.namesys.com/v4/v4.html

Seltzer, M., Bostic, K., McKusick, M. K. and Staelin, C. (1993). An implementation of a log-structured file system for unix, USENIX.

Skoglund, E., Ceelen, C. and Lidtke, J. (2000). Transparent orthogonal checkpointing through user-level pagers, *Technical report*, System Architecture Group, University of Karlsruhe.
*http://uhlig-langert.de/publications/files/l4-checkpointing.pdf

Smith, S. (2003). Interim honors presentation: The walnut kernel, *Technical report*, Monash University.

Sweeney, A. (1993a). 64 bit file access, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Sweeney, A. (1993b). xfs superblock management, *Technical report*, Silicon Graphics.
*http://oss.sgi.com/projects/xfs/

Sweeney, A. (1993c). xfs transaction mechanism, *Technical report*, Silicon Graphics.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G. (1996). Scalability in the xfs file system, *USENIX Conference*.

(tes@sgi.com), T. S. (2002). Re: Question about litte and big endian?, Electronic Mailing List - linux-xfs@oss.sgi.com.
*http://linux-xfs.sgi.com/projects/xfs/mail_archive/200209/msg00409.html

The FreeBSD Handbook (1995-2003). The FreeBSD Documentation Project.
*http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/configtuning-disk.html

Tichy, W. F. (1985). RCS - a system for version control, *Software - Practice and Experience* **15**(7): 637–654.
*citeseer.nj.nec.com/tichy91rcs.html

Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*, PhD thesis, The Australian National University.
*http://samba.org/ tridge/phd_thesis.pdf

Tweedie, S. C. (1998). Journaling the linux ext2fs filesystem, *Technical report*, LinuxExpo 98.

Wallace, C. and Pose, R. (1990). Charging in a secure environment.
*http://www.csse.monash.edu.au/courseware/cse4333/rdp-material/Bremen-paper.pdf

Wallace, C. S., Pose, R., Castro, M., Kopp, C., Pringle, G., Gunawan, S. and Jan... (2003). Informal discussions relating to the walnut kernel.

Woodhouse, D. (2001). Jffs: The journalling flash file system, *Ottawa Linux Symposium*, RedHat Inc.
*http://sources.redhat.com/jffs2/jffs2.pdf

Simulation Source Code

The source code is also available from `http://www.flamingspork.com/honors/` and is provided here for reference and in support of the thesis.

.1 Source Code License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you

distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or

collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is

void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing

to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) year name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it

```
    under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than ‘show w’ and ‘show c’; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```

.2 Testkit Block Device Simulator

Listing 1: testkit/Makefile

```
1  all: testkit.o blktest doc
2
3  GLIB_CFLAGS = ‘glib-config --cflags glib‘
4  GLIB_LIB = ‘glib-config --libs glib‘
5  CFLAGS:=-g
6
7  blktest: testkit.o blktest.c
8          $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB) $(CFLAGS)
9
10 testkit.o: block_dev.c
11         $(CC) -c $< $(GLIB_CFLAGS) $(CFLAGS) -o testkit.o
12
13 .PHONY: clean
14
15 clean:
16         rm -f blktest testkit.o
```

Listing 2: testkit/bitops.h

```
1  /*
2    bitops.h
3    --------
4    $Id: bitops.h,v 1.3 2003/07/06 12:28:03 stewart Exp $
5
6    functions similar to those present in include/asm/bitops.h
7    There is no *real* guarantee of being atomic, except for the
8    "we're not doing threads so go away" thing.
9
10   (C)2003 Stewart Smith
11   Distributed under the GNU Public License.
12
13   Some data structures have been constructed out of those
14   present in the Linux Kernel (v2.5.69). They are copyright
15   of their respective owners.
16
17   The API here is very much based on Linux kernel stuff.
18   Except that we don't do inline ASM to do this 'atomic' stuff.
19   bah - we live on the edge baby!
20  */
21
22 #ifndef __BITOPS_H__
23 #define __BITOPS_H__
24
25 #define ADDR (*(volatile long *) addr)
26
27 static inline void set_bit(int nr, volatile void * addr)
28 {
29   ADDR = ADDR | 1UL << nr;
30 }
31
32 static inline int test_and_set_bit(int nr, volatile void * addr)
33 {
34   int oldbit;
35   oldbit = ADDR & 1UL << nr;
36   set_bit(nr,addr);
37   return oldbit;
38 }
39
40 static inline void clear_bit(int nr,volatile void * addr)
41 {
```

```
42    ADDR = ADDR & ~(1UL << nr);
43  }
44
45  static inline int test_and_clear_bit(int nr,volatile void * addr)
46  {
47    int oldbit;
48    oldbit = ADDR & (1UL << nr);
49    clear_bit(nr,addr);
50    return oldbit;
51  }
52
53  static inline int test_bit(int nr,volatile void * addr)
54  {
55    return ADDR & (1UL << nr);
56  }
57
58  #endif
```

Listing 3: testkit/types.h

```
1  /*
2     types.h
3     -------
4     $Id: types.h,v 1.4 2003/07/06 12:27:37 stewart Exp $
5
6     Mostly glue around types that would usually be defined by the
          system.
7     This is more portable (unfortunately).
8   */
9  #ifndef __TYPES_H__
10 #define __TYPES_H__
11
12 #include <sys/types.h>
13
14 typedef int16_t s16;
15 typedef u_int16_t u16;
16 typedef int32_t s32;
17 typedef u_int32_t u32;
18 typedef int64_t s64;
19 typedef u_int64_t u64;
20
21 typedef u64 sector_t;
22 typedef int atomic_t;
23
24 #endif
```

Listing 4: testkit/block_dev.h

```
1  /*
2    block_dev.h
3    ~~~~~~~~~~~
4    $Id: block_dev.h,v 1.13 2003/10/01 05:42:06 stewart Exp $
5
6    Designed to be rather similar to the Linux Buffer Cache,
7    i.e. with functions like bread() et al that function similarly.
8    Should allow testing and development of filesystems purely in
9    userspace and purely in an application.
10   User Mode Linux could be used for this, but this suite is
       designed
11   for even earlier in the design process.
12
13   (C)2003 Stewart Smith
14   Distributed under the GNU Public License.
15
16   Some data structures have been constructed out of those
17   present in the Linux Kernel (v2.5.69). They are copyright
18   of their respective owners.
19  */
20
21 #ifndef __BLOCK_DEV_H__
22 #define __BLOCK_DEV_H__
23
24 #include "types.h"
25 #include "bitops.h"
26 #include <glib.h>
27
28 /*
29   struct block_device
30   ~~~~~~~~~~~~~~~~~~~
31   A really simple block_device structure.
32   $Id: block_dev.h,v 1.13 2003/10/01 05:42:06 stewart Exp $
33  */
34 struct block_device {
35   char* name;
36   int file_on_disk;
37   u64 block_size;
38   u64 num_blocks;
39
40   /* Internal */
```

```
41    u64 cache_hit;
42    u64 cache_hit_clear;
43    u64 cache_miss;
44    u64 cache_miss_clear;
45  };
46
47  struct super_block {
48    //struct list_head    s_list;         /* Keep this first */
49    //dev_t               s_dev;          /* search index;
          _not_ kdev_t */
50          unsigned long           s_blocksize;
51          unsigned long           s_old_blocksize;
52          unsigned char           s_blocksize_bits;
53          unsigned char           s_dirt;
54          unsigned long long      s_maxbytes;     /* Max file size
              */
55    //struct file_system_type     *s_type;
56    //struct super_operations     *s_op;
57    //struct dquot_operations     *dq_op;
58    //struct quotactl_ops *s_qcop;
59    //struct export_operations *s_export_op;
60          unsigned long           s_flags;
61          unsigned long           s_magic;
62    //struct dentry               *s_root;
63    //struct rw_semaphore s_umount;
64    //struct semaphore    s_lock;
65          int                     s_count;
66          int                     s_syncing;
67          int                     s_need_sync_fs;
68    //atomic_t            s_active;
69          void                    *s_security;
70
71    //struct list_head    s_dirty;        /* dirty inodes */
72    //struct list_head    s_io;           /* parked for writeback
          */
73    //struct hlist_head   s_anon;         /* anonymous dentries for
          (nfs) exporting */
74    //struct list_head    s_files;
75
76          struct block_device     *s_bdev;
77    //struct list_head    s_instances;
```

```
78    //struct quota_info    s_dquot;        /* Diskquota specific
          options */
79
80          char s_id[32];                          /* Informational
                name */
81
82    //struct kobject           kobj;          /* anchor for sysfs
          */
83          void                    *s_fs_info;     /* Filesystem
                private info */
84
85          /*
86           * The next field is for VFS *only*. No filesystems have
                any business
87           * even looking at it. You had been warned.
88           */
89    //     struct semaphore s_vfs_rename_sem;      /* Kludge */
90  };
91
92  /*
93     bh_state-bits
94    ---------------
95     straight from include/linux/buffer_head.h (kernel 2.5.69)
96   */
97  enum bh_state_bits {
98     BH_Uptodate,      /* Contains valid data */
99     BH_Dirty,         /* Is dirty */
100    BH_Lock,          /* Is locked */
101    BH_Req,           /* Has been submitted for I/O */
102
103    BH_Mapped,        /* Has a disk mapping */
104    BH_New,           /* Disk mapping was newly created by get_block
          */
105    BH_Async_Read,    /* Is under end_buffer_async_read I/O */
106    BH_Async_Write,   /* Is under end_buffer_async_write I/O */
107    BH_Delay,         /* Buffer is not yet allocated on disk */
108
109    BH_Boundary,      /* Block is followed by a discontiguity */
110    BH_PrivateStart,/* not a state bit, but the first bit available
111                     * for private allocation by other entities
112                     */
113 };
```

```
114
115 /*
116   buffer_head
117   -----------
118   straight from include/linux/buffer_head.h (kernel 2.5.69)
119   We've comment out things we don't really care too much about.
120   $Id: block_dev.h,v 1.13 2003/10/01 05:42:06 stewart Exp $
121  */
122 struct buffer_head {
123
124   unsigned long b_state;           /* buffer state bitmap (see
          above) */
125   atomic_t b_count;                /* users using this block */
126   struct buffer_head *b_this_page;/* circular list of page's
          buffers */
127   //  struct page *b_page;           /* the page this bh is
          mapped to */
128
129   sector_t b_blocknr;              /* block number */
130   u32 b_size;                      /* block size */
131   char *b_data;                    /* pointer to data block */
132
133   struct block_device *b_bdev;
134   //  bh_end_io_t *b_end_io;         /* I/O completion */
135   //  void *b_private;               /* reserved for b_end_io */
136   //  struct list_head b_assoc_buffers; /* associated with
          another mapping */
137 };
138
139 /*
140   Macro tricks to exapnd the set_buffer_foo(), clear_buffer_foo()
141   and buffer_foo() functions
142 */
143 #define BUFFER_FNS(bit,name) \
144 static inline void set_buffer_##name(struct buffer_head *bh)     \
145 { \
146   set_bit(BH_##bit, &(bh)->b_state);\
147 } \
148 static inline void clear_buffer_##name(struct buffer_head *bh) \
149 { \
150   clear_bit(BH_##bit, &(bh)->b_state); \
151 } \
```

```
152 static inline int buffer_##name(struct buffer_head *bh) \
153 { \
154   return test_bit(BH_##bit, &(bh)->b_state); \
155 }
156
157 /*
158    test_set_buffer_foo() and test_clear_buffer_foo()
159 */
160 #define TAS_BUFFER_FNS(bit, name) \
161 static inline int test_set_buffer_##name(struct buffer_head *bh)
        \
162 { \
163   return test_and_set_bit(BH_##bit, &(bh)->b_state); \
164 } \
165 static inline int test_clear_buffer_##name(struct buffer_head *bh
      ) \
166 { \
167   return test_and_clear_bit(BH_##bit, &(bh)->b_state); \
168 }
169
170 /*
171   Emit the buffer bitops functions.
172 */
173 BUFFER_FNS(Uptodate, uptodate)
174 BUFFER_FNS(Dirty, dirty)
175 TAS_BUFFER_FNS(Dirty, dirty)
176 BUFFER_FNS(Lock, locked)
177 TAS_BUFFER_FNS(Lock, locked)
178 BUFFER_FNS(Req, req)
179 BUFFER_FNS(Mapped, mapped)
180 BUFFER_FNS(New, new)
181 BUFFER_FNS(Async_Read, async_read)
182 BUFFER_FNS(Async_Write, async_write)
183 BUFFER_FNS(Delay, delay)
184 BUFFER_FNS(Boundary, boundary)
185
186
187
188 /*
189   for submit_bh
190 */
191 #define READ 1
```

```
192 #define WRITE 2
193
194 /*
195   our function definitions. Mostly glue.
196  */
197 int block_dev_new(struct block_device* b, const char* file, u64
      block_size, u64 num_blocks);
198 int block_dev_close(struct block_device *b);
199
200 struct buffer_head *bnew(struct block_device *b, sector_t block,
      int size);
201
202 /*
203   function definitions pretty much straight out of include/linux/
        block_head.h
204  */
205 struct buffer_head *bread(struct block_device *b, sector_t block
      , int size);
206
207 /*
208   reads block as per super block info
209  */
210 static inline struct buffer_head *
211 sb_bread(struct super_block *sb, sector_t block)
212 {
213         return bread(sb->s_bdev, block, sb->s_blocksize);
214 }
215
216 void block_dev_init();
217
218 void submit_bh(int rw,struct buffer_head * bh);
219
220 #endif
```

Listing 5: testkit/block_dev.c

```
1  /*
2    block_dev.c
3    ___________
4    $Id: block_dev.c,v 1.15 2003/10/01 05:41:57 stewart Exp $
5    (C)2003 Stewart Smith
6    Distributed under the GNU Public License
7
8      Some data structures have been constructed out of those
9    present in the Linux Kernel (v2.5.69). They are copyright
10   of their respective owners.
11  */
12
13 #define _GNU_SOURCE
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <sys/types.h>
18 #include <sys/stat.h>
19 #include <fcntl.h>
20 #include <unistd.h>
21
22 #include "block_dev.h"
23 #include <glib.h>
24
25
26
27 GList* lru_list;
28 GHashTable* blk_hash;
29
30 /* hash_bh
31    _______
32    Hashes a buffer head. Poor hash function, but
33    okay for our purposes as we generally only
34    use one block device at once in simulation.
35
36    Used with the Glib hash.
37 */
38 guint hash_bh(gconstpointer a)
39 {
40   struct buffer_head *bh = (struct buffer_head*)a;
```

```
41    return (bh->b_bdev->file_on_disk % 32768)+(bh->b_blocknr
          % 32768);
42  }
43
44  /* bh_equal
45     --------
46     true if the contents of two buffer heads are
47     the same (i.e. device, block number and block size)
48   */
49  gint bh_equal(gconstpointer a,gconstpointer b)
50  {
51    struct buffer_head *bha,*bhb;
52    bha = (struct buffer_head*)a;
53    bhb = (struct buffer_head*)b;
54
55    if(bha==NULL||bhb==NULL)
56      return 0;
57
58    return (bha->b_bdev == bhb->b_bdev
59            && bha->b_blocknr==bhb->b_blocknr
60            && bha->b_size == bhb->b_size
61            );
62  }
63
64  /* block_dev_init
65     --------------
66
67     Initializes the block device simulator.
68
69     Should be called before any other function.
70     On failure, aborts
71   */
72  void block_dev_init()
73  {
74    lru_list = NULL;
75    if((blk_hash = g_hash_table_new(hash_bh,bh_equal))==NULL)
76      { fprintf(stderr,"Failed to create blk_hash\n"); abort();}
77  }
78
79  /* block_dev_new
80     -------------
```

```
81      Fills out the block_device structure. doesn't allocate memory
          for it.
82
83      On failure of opening teh device, aborts.
84   */
85  int block_dev_new(struct block_device *b, const char* file, u64
      block_size, u64 num_blocks)
86  {
87    if( (b->file_on_disk=open(file,O_RDWR|O_CREAT,S_IRUSR|S_IWUSR))
        < 0)
88      {
89        fprintf(stderr,"Unable to open device file %s\n",file);
90        abort();
91      }
92    b->block_size = block_size;
93    b->num_blocks = num_blocks;
94    b->name = file;
95
96    b->cache_hit = 0;
97    b->cache_hit_clear = 0;
98    b->cache_miss = 0;
99    b->cache_miss_clear = 0;
100
101   return 1;
102 }
103
104 /* block_dev_close
105    ---------------
106    Closes our simulated block device. Warns if any dirty buffers
107    removes buffers from the buffer cache.
108 */
109 int block_dev_close(struct block_device *b)
110 {
111   struct buffer_head* bh;
112
113   if(g_list_first(lru_list))
114     {
115       bh = (struct buffer_head*)(g_list_first(lru_list)->data);
116       do
117         {
118           if(bh->b_bdev == b)    /* Our block device */
119             {
```

```
120                 if(buffer_dirty(bh))       /* Destroying without
                        flush */
121                   fprintf(stderr,      /* So we warn */
122                           "block_dev_close: bdev has Dirty buffers
                                (%x)\n",
123                           bh->b_blocknr);
124                 g_hash_table_remove(blk_hash,bh);
125                 lru_list = g_list_remove(lru_list,bh);
126                 fprintf(stderr,"Removing block from cache 0x%08llx\
                        n",bh->b_blocknr);
127               }
128             else
129               fprintf(stderr,"bh->b_dev = %p not %p\n",bh->b_bdev,b
                    );
130             if(g_list_first(lru_list))
131               bh = (struct buffer_head*)g_list_first(lru_list)->
                    data;
132             else
133               bh = NULL;
134           }while(bh!=NULL);
135       }
136     fprintf(stderr,"Closed Block Device: %s\n",b->name);
137     fprintf(stderr,"%s Stats:\n\t%lld hit\n\t%lld miss\n\n\t%lld
          Clear Hit\n\t%lld Clear Miss\n",b->name,b->cache_hit,b->
          cache_miss,b->cache_hit_clear,b->cache_miss_clear);
138 }
139
140 struct buffer_head *bread(struct block_device *b, sector_t block
       , int size)
141 {
142   struct buffer_head *bh , *bh2;
143   int i;
144
145 #ifdef DEBUG_VERBOSE
146   fprintf(stderr,"Attempting to get block 0x%llx\t",block);
147 #endif
148
149   if(block >= b->num_blocks)
150     {
151       fprintf(stderr,"block out of device range. b=%d\n",block);
152       return NULL;
153     }
```

```
154
155     if((bh = (struct buffer_head*) malloc(sizeof(struct buffer_head
            )))==NULL)
156       {
157         fprintf(stderr,"Cannot allocate struct buffer_head\n");
158         abort();
159       }
160     bh->b_blocknr = block;
161     bh->b_size = size;
162     bh->b_bdev = b;
163
164     if((bh2 = g_hash_table_lookup(blk_hash,bh))!=NULL)
165       {
166         //  fprintf(stderr,"Found block in cache 0x%llx at 0x%x\n",
                block,hash_bh(bh2));
167         free(bh);
168         lru_list = g_list_remove(lru_list,bh2);
169         lru_list = g_list_append(lru_list,bh2);
170         b->cache_hit++;
171         return bh2;
172       }
173
174     bh->b_state=0;
175     bh->b_count=0;
176
177     if((bh->b_data = (char*)malloc(sizeof(char)*size))==NULL)
178     {
179       fprintf(stderr,"Unable to Allocate data buffer\n");
180       abort();
181     }
182
183     lseek(b->file_on_disk,b->block_size*block,SEEK_SET);
184     if((i=(read(b->file_on_disk,bh->b_data,(size_t)size)!=size)))
185       {
186         fprintf(stderr,"An unexpected number of bytes was read: %d\
                n\n",i);
187         abort();
188       }
189
190     lru_list = g_list_append(lru_list,bh);
191     if(g_hash_table_lookup(blk_hash,bh)!=NULL && !bh_equal(
          g_hash_table_lookup(blk_hash,bh),bh))
```

```
192       fprintf(stderr,"HASH COLLISION!!!!!\n\n");
193     g_hash_table_insert(blk_hash,bh,bh);
194 #ifdef DEBUG_VERBOSE
195     fprintf(stderr,"blocks read: %d, hash size: %d hashed as: 0x%x\
          n",g_list_length(lru_list),g_hash_table_size(blk_hash),
          hash_bh(bh));
196 #endif
197
198     b->cache_miss++;
199
200     return bh;
201 }
202
203 struct buffer_head *bnew(struct block_device *b, sector_t block,
       int size)
204 {
205     struct buffer_head *bh , *bh2;
206     int i;
207
208 #ifdef DEBUG_VERBOSE
209     fprintf(stderr,"Attempting to get block 0x%llx\t",block);
210 #endif
211
212     if(block >= b->num_blocks)
213       {
214         fprintf(stderr,"block out of device range. b=%d\n",block);
215         return NULL;
216       }
217
218     if((bh = (struct buffer_head*) malloc(sizeof(struct buffer_head
          )))==NULL)
219       {
220         fprintf(stderr,"Cannot allocate struct buffer_head\n");
221         abort();
222       }
223     bh->b_blocknr = block;
224     bh->b_size = size;
225     bh->b_bdev = b;
226
227     if((bh2 = g_hash_table_lookup(blk_hash,bh))!=NULL)
228       {
```

```
229         //  fprintf(stderr,"Found block in cache 0x%llx at 0x%x\n",
                block,hash_bh(bh2));
230         free(bh);
231         lru_list = g_list_remove(lru_list,bh2);
232         lru_list = g_list_append(lru_list,bh2);
233         memset(bh2->b_data,0,size);
234         b->cache_hit_clear++;
235         return bh2;
236       }
237
238     bh->b_state=0;
239     bh->b_count=0;
240
241     if((bh->b_data = (char*)malloc(sizeof(char)*size))==NULL)
242     {
243       fprintf(stderr,"Unable to Allocate data buffer\n");
244       abort();
245     }
246
247     memset(bh->b_data,0,size);
248
249     lru_list = g_list_append(lru_list,bh);
250     if(g_hash_table_lookup(blk_hash,bh)!=NULL && !bh_equal(
          g_hash_table_lookup(blk_hash,bh),bh))
251       fprintf(stderr,"HASH COLLISION!!!!!\n\n");
252     g_hash_table_insert(blk_hash,bh,bh);
253 #ifdef DEBUG_VERBOSE
254     fprintf(stderr,"blocks read: %d, hash size: %d hashed as: 0x%x\
          n",g_list_length(lru_list),g_hash_table_size(blk_hash),
          hash_bh(bh));
255 #endif
256
257     b->cache_miss_clear++;
258
259     return bh;
260 }
261
262 void submit_bh(int rw,struct buffer_head * bh)
263 {
264     int result;
265
266     switch(rw)
```

```
267     {
268     case READ:
269       break;
270
271     case WRITE:
272       if(!buffer_dirty(bh))
273         fprintf(stderr,"Warning: Non dirty buffer being written.\
              n");
274 #ifdef DEBUG_VERBOSE
275       fprintf(stderr,"---BLOCK WRITE--- %d %lld\n",bh->b_size,bh
            ->b_blocknr);
276 #endif
277       lseek(bh->b_bdev->file_on_disk,bh->b_size*bh->b_blocknr,
            SEEK_SET);
278       result = write(bh->b_bdev->file_on_disk,bh->b_data,bh->
            b_size);
279       if(result <= 0)
280         { fprintf(stderr,"Error Writing Block, result = %d\n",
              result);}
281       clear_buffer_dirty(bh);
282       fdatasync(bh->b_bdev->file_on_disk);
283       break;
284     default:
285       fprintf(stderr,"Invalid mode to submit_bh (mode=%x)\n",rw);
286       abort();
287       break;
288     };
289 }
```

Listing 6: testkit/blktest.c

```
1  /*
2     blktest.c
3     ---------
4     Test driver for the block device simulator.
5
6     $Id: blktest.c,v 1.3 2003/07/06 12:28:03 stewart Exp $
7
8     (C) 2003 Stewart Smith
9     Distributed under the GNU Public License
10   */
11
12 #include "block_dev.h"
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16
17 int main(int argc,char* argv[])
18 {
19   struct block_device b;
20   struct buffer_head* bh;
21   int a;
22   int i;
23   int bsize = atoi(argv[2]);
24
25   if(argc<3)
26     {
27       fprintf(stderr,"Usage:\n\t./blktest device blocksize
           blockcount\n\n");
28       exit(0);
29     }
30
31   block_dev_init();
32   block_dev_new(&b,argv[1],atoi(argv[2]),atoi(argv[3]));
33
34   for(i=0;i<atoi(argv[3])/2;i++)
35     {
36       bh = bread(&b,i,atoi(argv[2]));
37       a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
38     }
39
40   for(i=atoi(argv[3])/4;i<atoi(argv[3])/2;i++)
```

```
41      {
42        bh = bread(&b,i,atoi(argv[2]));
43        a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
44      }
45
46    for(i=0;i<atoi(argv[3]);i++)
47      {
48        bh = bread(&b,i,atoi(argv[2]));
49        a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
50      }
51
52    for(i=atoi(argv[3])/4;i<atoi(argv[3])/2;i++)
53      {
54        bh = bread(&b,i,atoi(argv[2]));
55        a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
56      }
57
58    bh = bread(&b,0,bsize);
59    set_buffer_dirty(bh);
60    strcpy(bh->b_data,"Shut your fucking face, Uncle Fucker.");
61    submit_bh(WRITE,bh);
62
63    return 0;
64  }
```

.3 FCFS Source

Listing 7: fcfs/Makefile

```
1  #
2  # Makefile for FCFS
3  #------------------
4  # $Id: Makefile,v 1.14 2003/10/29 04:19:33 stewart Exp $
5  #
6  # (C)2003 Stewart Smith
7  # Distributed under the GNU Public License
8  #
9
10 CC:=gcc
11 CFLAGS:=-g -Wall -pg -fprofile-arcs
12 GLIB_CFLAGS = `glib-config --cflags glib`
13 GLIB_LIB = `glib-config --libs glib`
14
15 LIBTOOL:=ar r
16
17 .PHONY: clean doc
18
19 all: mkfs volinfo mkfile fcfs_newobj fcfs_readobj fcfs_updateobj
20
21 mkfs: mkfs.o testkit/testkit.o fcfs.a
22         $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB)
23
24 volinfo: volinfo.o testkit/testkit.o fcfs.a
25         $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB) $(CFLAGS)
26
27 fcfs_newobj: fcfs_newobj.o mount_testkit.o fcfs.a testkit/testkit
      .o
28         $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB) $(CFLAGS)
29
30 fcfs_readobj: fcfs_readobj.o mount_testkit.o fcfs.a testkit/
      testkit.o
31         $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB) $(CFLAGS)
32
33 fcfs_updateobj: fcfs_updateobj.o mount_testkit.o fcfs.a testkit/
      testkit.o
34         $(CC) -o $@ $^ $(GLIB_CFLAGS) $(GLIB_LIB) $(CFLAGS)
35
36 fcfs.a: space_bitmap.o disk.o onode.o onode_index.o super_block.o
```

```
37          $(LIBTOOL) $@ $^
38
39  mkfile: mkfile.c
40          $(CC) -o $@ $^ $(CFLAGS)
41
42  ################
43  # Documentation
44  ################
45
46  doc:
47          rm -rf doc/html; mkdir doc/html
48          rm -rf doc/latex; mkdir doc/latex
49          cxref *.c *.h testkit/*.c testkit/*.h -xref-all -index-
                all -html32-src -I. -I testkit/ $(GLIB_CFLAGS) -Odoc/
                html -all-comments
50          cxref *.c *.h testkit/*.c testkit/*.h -xref-all -index-
                all -I. -Itestkit/ $(GLIB_CFLAGS) -Odoc/latex -latex2e
51
52  #################
53  #
54  # And now.... C file compilation (yay)
55  #
56  #################
57
58  fcfs_newobj.o: fcfs_newobj.c
59          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
60
61  fcfs_readobj.o: fcfs_readobj.c
62          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
63
64  fcfs_updateobj.o: fcfs_updateobj.c
65          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
66
67  mount_testkit.o: mount_testkit.c
68          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
69
70  volinfo.o: volinfo.c
71          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
72
73  mkfs.o: mkfs.c
74          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
75
```

```
76  onode.o: onode.c
77          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
78
79  onode_index.o: onode_index.c
80          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
81
82  space_bitmap.o: space_bitmap.c
83          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
84
85  super_block.o: super_block.c
86          $(CC) -c $< $(CFLAGS) $(GLIB_CFLAGS)
87
88  disk.o: disk_testkit.c
89          $(CC) -c $< -o $@ $(CFLAGS) $(GLIB_CFLAGS)
90
91
92  ###############
93  # clean target
94  ###############
95
96  clean:
97          rm -f mkfs volinfo mkfile fcfs_newobj fcfs_readobj
                fcfs_updateobj
98          rm -f fcfs.a
99          rm -f mkfs.o space_bitmap.o disk.o onode.o onode_index.o
                volinfo.o super_block.o fcfs_newobj.o mount_testkit.o
                fcfs_readobj.o
100
101
102
103 #################
104 # TESTING
105 #################
106
107 testmkfs: mkfs
108         ./mkfile test 4096 10240
109         ./mkfs test 4096 10240 blergh
110
111 hexdump:
112         hexdump -C test
113
114 testvolinfo: volinfo testmkfs
```

```
115             ./volinfo test 4096 10240
```

Listing 8: fcfs/disk.h

```
1  /* disk.h
2     ------
3
4     FCFS Disk abstraction.
5
6     $Id: disk.h,v 1.7 2003/10/20 07:18:11 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10 */
11
12 #ifndef __DISK_H__
13 #define __DISK_H__
14
15 #include "testkit/block_dev.h"
16 #include "testkit/types.h"
17
18 struct fcfs_disk {
19   sector_t blocksnr;              /* number of blocks */
20   u32 bsize;                      /* Size of blocks */
21   struct fcfs_disk_block *sb_block;
22   struct fcfs_sb* sb;             /* The Disk SuperBlock */
23   void* os_private;               /* Used by OS for it's structure
        */
24 };
25
26 struct fcfs_disk_block {
27   sector_t blocknr;               /* Number of block*/
28   u32    bsize;                   /* Size of block*/
29   char* data;                                 /* Block data */
30   struct fcfs_disk *disk;         /* The disk the block is from */
31   void* os_private;               /* The OS data structure */
32 };
33
34 #define BR_SECTOR_T(disk,br) (((disk)->sb->ag_blocksnr * (br)->
     allocation_group) + (br)->start)
35
36 struct fcfs_disk* disk_new(struct block_device *bdev);
37 struct fcfs_disk* disk_free(struct fcfs_disk* disk);
38
```

```
39 struct fcfs_disk_block* disk_newblock(struct fcfs_disk *disk,
       sector_t block);
40 struct fcfs_disk_block* disk_getblock(struct fcfs_disk *disk,
       sector_t block);
41 //static inline struct fcfs_disk_block* disk_freeblock(struct
       fcfs_disk_block* block);
42 //static inline struct fcfs_disk_block* disk_writeblock(struct
       fcfs_disk_block* block);
43
44 /* When finished using a disk block */
45 static inline struct fcfs_disk_block* disk_freeblock(struct
       fcfs_disk_block* block)
46 {
47   ((struct buffer_head*)(block->os_private))->b_count--;
48   return block;
49 }
50
51 /* Write a disk block immediately. */
52 static inline struct fcfs_disk_block* disk_writeblock(struct
       fcfs_disk_block* block)
53 {
54   set_buffer_dirty((struct buffer_head*)block->os_private);
55   submit_bh(WRITE,(struct buffer_head*)block->os_private);
56   return block;
57 }
58
59
60 #endif
```

Listing 9: fcfs/onode.h

```
1  /* onode.h
2     ———————
3
4     Header for O-Node Manipulation
5
6     $Id: onode.h,v 1.9 2003/10/20 07:18:11 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10  */
11
12 #ifndef __ONODE_H__
13 #define __ONODE_H__
14
15 #include "testkit/types.h"
16
17 struct fcfs_onode1 *onode1_new(struct fcfs_disk *disk);
18 struct fcfs_onode1 *onode1_free(struct fcfs_onode1 *onode);
19 int onode1_fork_new(struct fcfs_disk *disk, struct fcfs_block_run
       *onode_br, u64 fork_type, u64 content_length, void* content,
     int allow_internal);
20 int onode1_fork_grow(struct fcfs_disk *disk, struct
     fcfs_block_run *onode_br, int forknr, u64 blocksnr);
21
22 u64 onode1_fork_length(struct fcfs_onode1 *onode,int forknr);
23 int onode1_fork_write_versioned(struct fcfs_disk *disk,struct
     fcfs_onode1 *onode, struct fcfs_block_run *onode_br,int
     forknr,u64 pos,u64 content_length,void* content);
24 int onode1_fork_write(struct fcfs_disk *disk, struct
     fcfs_block_run *onode_br,int forknr, u64 pos, u64
     content_length, void* content);
25 u64 onode1_fork_read(struct fcfs_disk *disk, struct fcfs_onode1 *
     onode,int forknr, u64 pos, u64 content_length, void* content);
26 struct fcfs_disk_block *onode1_fork_getblock(struct fcfs_disk *
     disk, struct fcfs_onode1 *onode,int forknr,u64 blocknr);
27
28 #endif
```

Listing 10: fcfs/onode_versioned.h

```
1  /* onode_versioned.h
2     ------------------
3
4     Data structures for versioned onode forks,
5     keeps track of the revision log.
6
7     $Id: onode_versioned.h,v 1.2 2003/10/20 07:18:11 stewart Exp $
8
9     (C)2003 Stewart Smith
10    Distributed under the GNU GPL
11 */
12
13 #ifndef _ONODE_VERSIONED_H_
14 #define _ONODE_VERSIONED_H_
15
16 #include "testkit/types.h"
17
18 enum fcfs_onode_fork_rev_op {
19   FCFS_ONODE_FORK_REV_REPLACE,
20   FCFS_ONODE_FORK_REV_INSERT,
21   FCFS_ONODE_FORE_REV_APPEND,
22   FCFS_ONODE_FORK_REV_TRUNCATE,
23   FCFS_ONODE_FORK_REV_GROW,
24 };
25
26 #define FCFS_ONODE_FORK_REV_MAGIC1 0xF0526B76    /* 0xF0 'R' 'k' '
      v' */
27
28 struct fcfs_onode_fork_rev {
29   u64 revision;                  /* Increment for each revision */
30   u64 consistent_revision;       /* Used to note last consistent
        revision */
31   u64 time;                      /* Time revision was done (not
        committed) */
32   u32 operation;                 /* enum fcfs_onode_fork_rev_op */
33   u32 magic1;                    /* FCFS_ONODE_FORK_REV_MAGIC1 */
34   u64 rev_offset;                /* Offset in revision fork */
35   u64 rev_length;                /* Length of revision fork data
       */
36   u64 offset;                    /* Offset in real fork to apply
        to*/
```

```
37     char space[16];                 /* Space for future revisions */
38  };
39
40  #endif
```

Listing 11: fcfs/onode_index.h

```
1  /* onode_index.h
2     ---------------
3     A casually demented B+Tree designed to stay balanced
4     and be optimized for access from disk (and in-core cache)
5     and not to have to write lots of times.
6
7     Some of the ideas for this came from Reiser4 stuff, except
8     I'm allowing unbalanced stuff to make it to disk if needed.
9     I'm thinking that if things get really bad, we can run a
10    tree repacker :)
11
12    $Id: onode_index.h,v 1.9 2003/10/20 07:18:11 stewart Exp $
13
14    (C)2003 Stewart Smith
15    Distributed under the GNU Public License
16 */
17
18 #ifndef __ONODE_INDEX_H__
19 #define __ONODE_INDEX_H__
20
21 #include "testkit/types.h"
22
23 /* In memory representation of onode_index */
24 struct fcfs_onode_index {
25   //  u64 root_blocknr; /* block run for root of index */
26   struct fcfs_onode_index_node *root; /* the node */
27   struct fcfs_disk_block *root_block; /* disk block for node */
28   struct fcfs_disk *disk;        /* disk the index is on */
29 };
30
31 #define FCFS_ONODE_INDEX_NODE_MAGIC1 0x4f6e4944784e4445ULL  /*
     OnIDxNDE */
32
33 struct fcfs_onode_index_node_item {
34   u64 key;
35   u64 node_blocknr;
36 };
37
38 struct fcfs_onode_index_node {
39   u64 magic1;                    /* FCFS_ONODE_INDEX_NODE_MAGIC1
        */
```

```
40    u64 id;                          /*Not necessarily unique, used
          for locking */
41    u64 block;
42    u64 used;
43    struct fcfs_onode_index_node_item items[1];
44  };
45
46  #define FCFS_ONODE_INDEX_LEAF_MAGIC1 0x4f6e4944784c6665ULL  /*
        OnIDxLfe */
47
48  struct fcfs_onode_index_leaf_item {
49    u64 key;
50    u64 onode_blocknr;
51  };
52
53  struct fcfs_onode_index_leaf {
54    u64 magic1;                      /* FCFS_ONODE_INDEX_LEAF_MAGIC1
          */
55    u64 id;                          /* Not necessarily unique, used
          for locking */
56    u64 block;
57    u64 used;
58    struct fcfs_onode_index_leaf_item items[1];
59  };
60
61  u64 onode_index_new_id(struct fcfs_onode_index *index);
62  struct fcfs_onode_index* onode_index_read(struct fcfs_disk *disk)
        ;
63  struct fcfs_onode_index* onode_index_new(struct fcfs_disk *disk);
64  int onode_index_write_leaf(struct fcfs_onode_index *index, struct
        fcfs_onode_index_leaf *leaf);
65  struct fcfs_disk_block *onode_index_new_node(struct
      fcfs_onode_index *index);
66  struct fcfs_onode_index_leaf *onode_index_leaf_new(struct
      fcfs_onode_index *index, struct fcfs_onode_index_node *parent
      , int position);
67  struct fcfs_onode_index_node* onode_index_read_node(struct
      fcfs_onode_index *index, u64 blocknr);
68  int onode_index_free_node(struct fcfs_onode_index *index, struct
      fcfs_onode_index_node* node);
69  int onode_index_write_node(struct fcfs_onode_index *index, struct
        fcfs_onode_index_node* node);
```

```
70
71 int onode_index_new_root(struct fcfs_onode_index *index);
72 int onode_index_leaf_insert(struct fcfs_disk *disk,struct
      fcfs_onode_index_leaf *leaf, u64 key, u64 value);
73 struct fcfs_block_run* onode_index_insert(struct fcfs_onode_index
      *index, struct fcfs_onode1 *onode);
74 struct fcfs_disk_block *onode_index_lookup(struct
      fcfs_onode_index *index,struct fcfs_block_run *onode_br, u64
      id);
75 struct fcfs_onode_index* onode_index_delete(struct
      fcfs_onode_index* index);
76
77 #endif
```

Listing 12: fcfs/space_bitmap.h

```
1  /* space_bitmap.h
2     ---------------
3
4     Header file for the bitmap block allocator
5
6     $Id: space_bitmap.h,v 1.3 2003/10/20 07:18:11 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10  */
11
12 #ifndef _SPACE_BITMAP_H_
13 #define _SPACE_BITMAP_H_
14
15 #include "testkit/types.h"
16
17 #define BLOCK_AG(disk,block) (block/disk->sb->ag_blocksnr)
18
19 u32 space_bitmap_size(struct fcfs_sb *sb, int ag);
20 int space_bitmap_allocate_block(struct fcfs_disk *disk, u32
      allocation_group,u32 block);
21 struct fcfs_block_run *ag_allocate_block(struct fcfs_disk *disk,
      u32 allocation_group, u32 near, u32 blocksnr);
22 int ag_block_free(struct fcfs_disk *disk,u32 allocation_group,
      u32 block);
23
24 #endif
```

Listing 13: fcfs/super_block.h

```
1  /*
2   super_block.h
3   ---------------
4   $Id: super_block.h,v 1.13 2003/10/20 07:18:11 stewart Exp $
5
6   Part of the FCFS object store.
7
8   This file describes the super block and related data structures.
9
10  (C) 2003 Stewart Smith
11  Distributed under the GNU Public License
12 */
13 #ifndef __FCFS_SUPER_BLOCK_H__
14 #define __FCFS_SUPER_BLOCK_H__
15
16 #include "disk.h"
17
18 /* fcfs_block_run
19    ---------------
20    We have a max of ~4 billion alloction groups,
21    within these, we can have up to 4 billion blocks.
22    and we can address these with *one* block_run structure.
23    This means we don't have a lot of block address lookup on
         large files
24    or contiguous files. Should lead to fast IO...
25  */
26 struct fcfs_block_run {
27   u32 allocation_group;          /* Allocation group */
28   u32 start;                     /* Start Block */
29   u32 len;                       /* Length (blocks) */
30 };
31
32 #define FCFS_SB_MAGIC1    0x46436673   /* FCFS */
33 #define FCFS_SB_VERSION1  0x00010001   /* V1 SB, V1 FS */
34 #define FCFS_SB_BITS      0x00000040   /* We're 64 bit - but could
       be bigger...*/
35
36 #define FCFS_NAME_LENGTH 128           /* Human Readable Name */
37 #define FCFS_SB_MAGIC2    0x3f8ec2a1   /* There is no meaning, just
       weird num */
38
```

```
39 enum fcfs_sb_flags {
40   FCFS_FLAG_SBLocation1,          /* Three flags make number */
41   FCFS_FLAG_SBLocation2,          /* see: enum fcfs_sb_location */
42   FCFS_FLAG_SBLocation3,
43   FCFS_FLAG_Dirty,                /* Was uncleanly unmounted*/
44   FCFS_FLAG_Experimental,         /* Has been written by
       experimental code */
45   FCFS_FLAG_JournalMeta,          /* Journal meta-data */
46   FCFS_FLAG_JournalData,          /* Journal all data writes*/
47   FCFS_FLAG_Versioned,            /* Is versioned system */
48 };
49
50 enum fcfs_sb_location {
51   FCFS_SBloc_start_volume,        /* Start of volume */
52   FCFS_SBloc_start_ag,            /* Start of allocation group */
53   FCFS_SBloc_end_volume           /* Final block of volume */
54 };
55
56 struct fcfs_sb {
57   u32 magic1;     // first magic number.
58   u32 version;    // version of FS
59   u32 bits;       // number of bits to use as a 'base' bits. (64
       default)
60   u32 sblength;    // length of sb (bytes)
61
62   u64 flags;
63
64   u32 bsize;      // block size (bytes)
65   u64 blocksnr;  // number of blocks
66
67
68   char name[FCFS_NAME_LENGTH];  // human readable name of volume
69   u32 magic2;     // magic2
70
71   u32 allocation_groupsnr;        /* Number of alloction groups on
       volume */
72   u32 ag_blocksnr;                /* Blocks per allocation group */
73
74   u64 num_mounts;                 /* Number of successful read/
       write mounts */
75   u64 num_dirtymounts;            /* Num dirty mounts */
76   u64 time_created;               /* Time volume created*/
```

```
77    u64 time_clean;                 /* Time volume last unmounted
          cleanly*/
78    u64 time_dirtymount;            /* Time volume last mounted dirty
          */
79
80    u64 onode_index_blocknr;        /* Where we find our primary
          index */
81
82    u64 onindex_num_onodes;         /* Num onodes in index */
83    u64 onindex_used;               /* Num of used inodes in index */
84    u64 onindex_free;               /* Num of free onodes in index */
85    u64 onindex_next_id;            /* Next ID to use */
86    u32 onindex_node_size;          /* In number of keys */
87
88    union space_tracking {
89
90      char padding[50];
91    }space;
92  };
93
94  #define FCFS_ONODE_MAGIC1  0x4f4e6f4465563101ULL          /*
        ONoDeV1 0x01 */
95
96  enum fcfs_onode_flags {
97    FCFS_OFLAG_NoVersion,           /* Don't version track this onode
          */
98    FCFS_OFLAG_ForkLeaf             /* We have a fork leaf, not a
          node */
99  };
100
101 #define FSFS_ONODE1_SPACE_LEAF_MAXNR 10
102
103 struct fcfs_onode1_space_leaf {
104   char nr;
105   struct fcfs_block_run br[FSFS_ONODE1_SPACE_LEAF_MAXNR];
106 };
107
108 struct fcfs_onode1_space_node {
109   u64 offset[7];
110   u64 block[7];
111 };
112
```

```
113 enum fcfs_fork_flags {
114   FCFS_FORK_InForkData,          /* We have data, not space nodes
        */
115   FCFS_FORK_SpaceNode            /* Space is node, not leaf */
116 };
117
118 #define FCFS_FORK_SMALL_DATA_SIZE 112
119
120 struct fcfs_fork {
121   u64 fork_type;
122   u64 fork_flags;
123   u64 content_length;
124   union data {
125     union space {
126       struct fcfs_onode1_space_leaf leaf;
127       struct fcfs_onode1_space_node node;
128     } space;
129
130     char small_data[FCFS_FORK_SMALL_DATA_SIZE]; /* I wish this
          was neater, as part of the leaf */
131   } data ;
132 };
133
134 struct fcfs_fork_leaf {
135   char nr;
136   struct fcfs_fork fork[10];
137 };
138
139 struct fcfs_fork_node {
140   u64 offset[7];
141   u64 block[7];
142 };
143
144 struct fcfs_onode1 {
145   u64 magic1;                    /* Identify as O-Node */
146   u64 onode_num;                 /* FS Specific Unique ID */
147   u64 onode_revision;            /* Revision of onode */
148   u64 flags;                     /* fcfs_onode_flags */
149   u64 use_count;                 /* Reference Counter (for
        indicies) */
150   u32 onode_size;                /* Length of o-node structure. */
151   union forks {
```

```
152       struct fcfs_fork_leaf leaf;
153       struct fcfs_fork_node node;
154     } forks;
155     char small_space[1];          /* used for in-onode data (or
            metadata) */
156 };
157
158
159 /* Functions for manipulating the Super Block */
160 int fcfs_write_sb(struct fcfs_disk *disk);
161 int fcfs_sb_mark_dirty(struct fcfs_disk *disk);
162 int fcfs_sb_mark_clean(struct fcfs_disk *disk);
163
164 #endif
```

Listing 14: fcfs/fcfs_vfs.h

```
1  /*
2     fcfs_vfs.h
3     ----------
4     VFS operations for FCFS
5
6     $Id: fcfs_vfs.h,v 1.1 2003/10/28 16:17:54 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GPL
10 */
11
12 #ifndef __FCFS_VFS_H__
13 #define __FCFS_VFS_H__
14
15 #include "disk.h"
16 #include "super_block.h"
17 #include "onode.h"
18
19
20 struct fcfs_disk *fcfs_mount(char *name);
21 int fcfs_umount(struct fcfs_disk *disk);
22
23
24 #endif
```

Listing 15: fcfs/disk_testkit.c

```
1  /* disk_testkit.c
2     ---------------
3
4     Disk access functions used under Stewart Smith's Testkit.
5
6     $Id: disk_testkit.c,v 1.7 2003/10/20 07:18:11 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include "testkit/block_dev.h"
16 #include "disk.h"
17
18 /* disk_new
19    --------
20    new disk object from testkit block device.
21 */
22 struct fcfs_disk* disk_new(struct block_device *bdev)
23 {
24   struct fcfs_disk* disk;
25
26   disk = (struct fcfs_disk*)malloc(sizeof(struct fcfs_disk));
27   disk->os_private = bdev;
28   disk->bsize = bdev->block_size;
29   disk->blocksnr = bdev->num_blocks;
30   disk->sb = NULL;
31   return disk;
32 }
33
34 /* disk_free
35    ---------
36    Close and free the disk.
37  */
38 struct fcfs_disk* disk_free(struct fcfs_disk* disk)
39 {
40   block_dev_close(disk->os_private);
41   free(disk);
```

```
42 }
43
44 /* Return an empty disk_block for the block. Does not read from
       disk */
45 struct fcfs_disk_block* disk_newblock(struct fcfs_disk *disk,
       sector_t block)
46 {
47   struct fcfs_disk_block* disk_block;
48
49   disk_block = (struct fcfs_disk_block*)malloc(sizeof(struct
       fcfs_disk_block));
50   if(disk_block==NULL)
51     {
52       fprintf(stderr,"Unable to allocate disk_block\n");
53       abort();
54     }
55   disk_block->os_private = bnew(disk->os_private,block,disk->
       bsize);
56   disk_block->blocknr = block;
57   disk_block->bsize = disk->bsize;
58   disk_block->data = ((struct buffer_head*)(disk_block->
       os_private))->b_data;
59   ((struct buffer_head*)(disk_block->os_private))->b_count++;
60   disk_block->disk = disk;
61
62   return disk_block;
63 }
64
65 /* Allocate and return a disk block */
66 struct fcfs_disk_block* disk_getblock(struct fcfs_disk *disk,
       sector_t block)
67 {
68   struct fcfs_disk_block* disk_block;
69
70   disk_block = (struct fcfs_disk_block*)malloc(sizeof(struct
       fcfs_disk_block));
71   if(disk_block==NULL)
72     {
73       fprintf(stderr,"Unable to allocate disk_block\n");
74       abort();
75     }
76
```

```
77    //   fprintf(stderr,"DISK_GETBLOCK: 0x%llx\n",block);
78
79    disk_block->os_private = bread(disk->os_private,block,disk->
          bsize);
80    disk_block->blocknr = block;
81    disk_block->bsize = disk->bsize;
82    disk_block->data = ((struct buffer_head*)(disk_block->
          os_private))->b_data;
83    ((struct buffer_head*)(disk_block->os_private))->b_count++;
84    disk_block->disk = disk;
85
86    return disk_block;
87  }
```

Listing 16: fcfs/super_block.c

```
1  /* super_block.c
2     ---------------
3     $Id: super_block.c,v 1.2 2003/09/22 09:02:16 stewart Exp $
4
5     The long awaited super_block.c
6     i wish i had coded this before, gah, i thought i had.
7     maybe i should give up the booze and late nights up coding.
8
9     This file contains all those functions dealing with the super
10    block of the FCFS Object Store.
11
12
13    One day, we need a better name than FCFS, but really - come up
14    with one. StewStore just seems lame.
15
16    (C)2003 Stewart Smith
17    Distributed under the GNU Public License.
18 */
19
20 #include <stdio.h>
21 #include <stdlib.h>
22 #include <string.h>
23 #include "testkit/types.h"
24
25 #include "disk.h"
26 #include "super_block.h"
27
28 int fcfs_write_sb(struct fcfs_disk *disk)
29 {
30   struct fcfs_disk_block *block;
31   block = disk_getblock(disk,0);
32   disk_writeblock(block);
33   disk_freeblock(block);
34   return 1;
35 }
36
37 int fcfs_sb_mark_dirty(struct fcfs_disk *disk)
38 {
39   set_bit(FCFS_FLAG_Dirty,&(disk->sb->flags));
40   fcfs_write_sb(disk);
41   return 1;
```

```
42 }
43
44 int fcfs_sb_mark_clean(struct fcfs_disk *disk)
45 {
46   clear_bit(FCFS_FLAG_Dirty,&(disk->sb->flags));
47   fcfs_write_sb(disk);
48   return 1;
49 }
```

Listing 17: fcfs/space_bitmap.c

```
1  /* space_bitmap.c
2     ---------------
3     Simple space allocator using a bitmap
4
5     $Id: space_bitmap.c,v 1.11 2003/10/20 07:18:11 stewart Exp $
6
7     (C)2003 Stewart Smith
8     Distributed under the GNU Public License
9  */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include "testkit/block_dev.h"
14 #include "testkit/bitops.h"
15 #include "super_block.h"
16 #include "disk.h"
17 #include "space_bitmap.h"
18
19 #define MAX_DIV(a,b) (((a%b)==0)?(a/b):(a/b)+1)
20
21 u64 next_free = 300;
22
23 u32 space_bitmap_size(struct fcfs_sb *sb, int ag)
24 {
25   /* if last ag, is remaining no. of blocks */
26   if(ag == sb->allocation_groupsnr-1)
27     return MAX_DIV(MAX_DIV((sb->ag_blocksnr+(sb->blocksnr%sb->
         allocation_groupsnr)),8),sb->bsize);
28   else
29     /* 1 bit per block */
30     return MAX_DIV(MAX_DIV(sb->ag_blocksnr,8),sb->bsize);
31 }
32
33 int space_bitmap_allocate_block(struct fcfs_disk *disk, u32
     allocation_group,u32 block)
34 {
35   struct fcfs_disk_block *b;
36
37 #ifdef DEBUG_SPACE_BITMAP
38   printf("ALLOC %u %u\n",allocation_group,block);
39 #endif
```

```
40
41    b = disk_getblock(disk,
42                      (allocation_group)*disk->sb->ag_blocksnr
43                      +(block/(disk->sb->bsize*8))
44                      +1);
45    set_bit(block%8,
46            (b->data+((block%(disk->sb->bsize*8))/8)));
47
48    disk_writeblock(b);
49    disk_freeblock(b);
50    return 0;
51  }
52
53  struct fcfs_block_run *ag_allocate_block(struct fcfs_disk *disk,
      u32 allocation_group, u32 near, u32 blocksnr)
54  {
55    struct fcfs_block_run *br;
56    int i;
57
58  #ifdef DEBUG_SPACE_BITMAP
59    printf("%u %u\n",allocation_group,near);
60  #endif
61
62    br = (struct fcfs_block_run*)malloc(sizeof(struct
        fcfs_block_run));
63    if(!br)
64      {
65        fprintf(stderr,"ag_allocate_block: Unable to allocate
            block_run\n");
66        abort();
67      }
68
69    br->start = 0;
70    br->len = 0;
71
72    if(near+blocksnr > disk->sb->ag_blocksnr)
73      blocksnr = disk->sb->ag_blocksnr - near;
74
75    for(i=1;i<=blocksnr+1;i++)
76      {
77        if(near+i>=disk->sb->ag_blocksnr)
```

```
78          {allocation_group++;near=0;i=0; br->start=0;br->len=0;/*
                printf("RESTARTING BLOCK SEARCH IN NEW ALLOCATION
                GROUP\n");*/}
79        if(ag_block_free(disk,allocation_group,near+i))
80          {
81 #ifdef DEBUG_SPACE_BITMAP
82            fprintf(stderr,"BLOCK %lu %lu is free\n",
                allocation_group,near+i);
83 #endif
84            if(br->start==0)
85              {  br->start=near+i; br->len=1;}
86            else
87              if((br->start+br->len)==(near+i-1))
88                br->len++;
89          }
90        else
91          {
92            //fprintf(stderr,"BLOCK %lu %lu isn't free\n",
                allocation_group,near+i);
93            near++;
94            i=0;
95          }
96      }
97
98    br->allocation_group = allocation_group;
99
100   for(i=0;i<br->len;i++)
101     space_bitmap_allocate_block(disk,br->allocation_group,br->
          start+i);
102
103   next_free = br->start + br->len;
104
105   return br;
106 }
107
108 u64 ag_block_free_last;
109 struct fcfs_disk_block *ag_block_free_last_block;
110
111 int ag_block_free(struct fcfs_disk *disk,u32 allocation_group,
      u32 block)
112 {
113   struct fcfs_disk_block *b;
```

```
114    int retval;
115
116 #ifdef DEBUG_SPACE_BITMAP
117    printf("\t%u %u\n",allocation_group,block);
118 #endif
119
120    if(block > disk->sb->ag_blocksnr)
121      {
122        fprintf(stderr,"ERROR: ag_black_free() can't check block
               that's out of this AG\n");
123        abort();
124      }
125
126    if(ag_block_free_last==0 ||
127       ag_block_free_last!=
128       (allocation_group)*disk->sb->ag_blocksnr+(block/(disk->sb->
              bsize*8))+1
129       )
130      {
131        if(ag_block_free_last!=0) disk_freeblock(
               ag_block_free_last_block);
132        ag_block_free_last = (allocation_group)*disk->sb->
               ag_blocksnr
133          +(block/(disk->sb->bsize*8))
134          +1;
135        ag_block_free_last_block = b = disk_getblock(disk,
               ag_block_free_last);
136      }
137    else
138      {
139        b = ag_block_free_last_block;
140      }
141
142
143    retval =  !test_bit(block%8,
144                        (b->data+((block%(disk->sb->bsize*8))/8)));
145
146
147    return retval;
148 }
```

Listing 18: fcfs/onode.c

```
1  /* onode.c
2     ———————
3
4     Code for manipulating FCFS O–Nodes and their forks.
5     Maybe the Fork stuff should be in a seperate file,
6     you're welcome to patch :)
7
8     $Id: onode.c,v 1.15 2003/10/20 07:18:11 stewart Exp $
9
10    (C)2003 Stewart Smith
11    Distributed under the GNU Public License
12
13    Debugged with Beer(TM)
14  */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #include <assert.h>
20 #include <time.h>
21
22 #include "testkit/types.h"
23 #include "super_block.h"
24 #include "disk.h"
25 #include "onode.h"
26 #include "onode_index.h"
27 #include "space_bitmap.h"
28 #include "onode_versioned.h"
29
30 struct fcfs_onode1 *onode1_new(struct fcfs_disk *disk)
31 {
32   struct fcfs_onode1 *node;
33
34   node = (struct fcfs_onode1*)malloc(sizeof(struct fcfs_onode1));
35   if(!node)
36     {
37       fprintf(stderr,"cannot allocate onode1, exiting\n");
38       abort();
39     }
40
41   node->magic1 = FCFS_ONODE_MAGIC1;
```

```
42    node->onode_num = 0;
43    node->onode_revision = 0;
44    set_bit(FCFS_OFLAG_ForkLeaf,&(node->flags));
45    node->use_count = 0;
46    node->onode_size = sizeof(struct fcfs_onode1);
47    node->forks.leaf.nr = 0;
48
49    return node;
50  }
51
52  struct fcfs_onode1 *onode1_free(struct fcfs_onode1 *onode)
53  {
54    free(onode);
55    return NULL;
56  }
57
58  /*
59     FIXME: the allow_internal is nasty hack, should be removed and
           made elegant
60  */
61  int onode1_fork_new(struct fcfs_disk *disk, struct fcfs_block_run
       *onode_br, u64 fork_type, u64 content_length, void* content,
      int allow_internal)
62  {
63    struct fcfs_disk_block *block;
64    struct fcfs_onode1 *onode;
65
66    block = disk_getblock(disk,BR_SECTOR_T(disk,onode_br));
67    onode = (struct fcfs_onode1*)block->data;
68
69    onode->forks.leaf.nr++;          /* should be atomic */
70    onode->forks.leaf.fork[onode->forks.leaf.nr-1].fork_type =
         fork_type;
71    onode->forks.leaf.fork[onode->forks.leaf.nr-1].content_length
         = content_length;
72    if(content_length <= FCFS_FORK_SMALL_DATA_SIZE &&
         content_length > 0 && allow_internal)
73      {                              /* Let's make it a small_data
           node */
74        onode->forks.leaf.fork[onode->forks.leaf.nr-1].fork_flags
             = FCFS_FORK_InForkData;
75        if(content!=NULL)
```

```
76          memcpy(onode->forks.leaf.fork[onode->forks.leaf.nr-1].
                data.small_data, content,content_length);
77        else
78          memset(onode->forks.leaf.fork[onode->forks.leaf.nr-1].
                data.small_data, 0,FCFS_FORK_SMALL_DATA_SIZE);
79      }
80    else                              /* We're going to need more
        space! */
81      {
82        onode->forks.leaf.fork[onode->forks.leaf.nr-1].fork_flags
              = 0;
83        onode->forks.leaf.fork[onode->forks.leaf.nr-1].data.space.
            leaf.nr = 0;
84
85        if(content_length > FCFS_FORK_SMALL_DATA_SIZE && content!=
            NULL)
86          fprintf(stderr,"**FIXME** Need larger objects\n");
87        else
88          {
89  #ifdef DEBUG_ONODE
90            fprintf(stderr,"PREALLOCATING ONODE FORK SPACE\n\n");
91  #endif
92          onode1_fork_grow(disk,onode_br,onode->forks.leaf.nr-1,
                content_length/disk->bsize + ((content_length%disk->
                bsize)?1:0));}
93      }
94
95    disk_writeblock(block);
96    disk_freeblock(block);
97
98    return onode->forks.leaf.nr-1;
99  }
100
101 struct fcfs_disk_block *onode1_fork_getblock(struct fcfs_disk *
      disk, struct fcfs_onode1 *onode,int forknr,u64 blocknr)
102 {
103   int i;
104   u64 cur_blocknr;
105   struct fcfs_disk_block *block;
106
107   block = NULL;
108
```

```
109     if(forknr > onode->forks.leaf.nr)
110       {
111         fprintf(stderr,"Invalid Fork Number for onode\n");
112         abort();
113       }
114
115     //  fprintf(stderr,"\n\n\t\t\tGETTING blocknr %llu from leaf.nr
            %d\n\n",blocknr,onode->forks.leaf.fork[forknr].data.space.
            leaf.nr);
116
117     cur_blocknr = 0;
118     for(i=0;i<onode->forks.leaf.fork[forknr].data.space.leaf.nr;i
          ++)
119       {
120         if(cur_blocknr+onode->forks.leaf.fork[forknr].data.space.
              leaf.br[i].len
121            >= blocknr)
122           {                             /* blocknr in current br */
123             block = disk_getblock(disk,
124                       BR_SECTOR_T(disk,
125                                   &(onode->forks.leaf.fork[forknr].
                                        data.space.leaf.br[i]))
126                                   +(blocknr-cur_blocknr));
127             //       fprintf(stderr,"GOT BLOCK #%llu\n",block->
                    blocknr);
128
129             break;
130           }
131         else
132           {
133             cur_blocknr+=onode->forks.leaf.fork[forknr].data.space.
                    leaf.br[i].len;
134           }
135       }
136
137     return block;
138   }
139
140   u64 onode1_fork_length(struct fcfs_onode1 *onode,int forknr)
141   {
142     return (onode->forks.leaf.fork[forknr].content_length);
143   }
```

```
144
145 int onode1_fork_write_versioned(struct fcfs_disk *disk,struct
      fcfs_onode1 *onode, struct fcfs_block_run *onode_br,int
      forknr,u64 pos,u64 content_length,void* content)
146 {
147   void* olddata;
148   struct fcfs_onode_fork_rev rev;
149
150   rev.magic1 = FCFS_ONODE_FORK_REV_MAGIC1;
151   rev.time = time(NULL);
152   rev.revision = onode->onode_revision; /* FIXME */
153   rev.consistent_revision = onode->onode_revision; /* FIXME */
154   rev.operation = FCFS_ONODE_FORK_REV_REPLACE;
155   rev.rev_offset = onode1_fork_length(onode,forknr+1);
156   rev.rev_length = content_length;
157   rev.offset = pos;
158   memset(rev.space,0,16);
159
160   olddata = malloc(content_length);
161   if(!olddata)
162     {fprintf(stderr,"No memory for updating content\n");abort();}
163   onode1_fork_read(disk,onode,forknr,pos,content_length,olddata);
164   onode1_fork_write(disk,onode_br,forknr+1,onode1_fork_length(
        onode,forknr+1),content_length,olddata);
165   onode1_fork_write(disk,onode_br,forknr+2,onode1_fork_length(
        onode,forknr+2),sizeof(struct fcfs_onode_fork_rev),&rev);
166   onode1_fork_write(disk,onode_br,forknr,pos,content_length,
        content);
167   return 1;
168 }
169
170 int onode1_fork_write(struct fcfs_disk *disk, struct
      fcfs_block_run *onode_br,int forknr, u64 pos, u64
      content_length, void* content)
171 {
172   struct fcfs_disk_block *onode_block;
173   struct fcfs_disk_block *block;
174   struct fcfs_onode1 *onode;
175   u64 write_length;
176   u64 done_length;
177
178   onode_block = disk_getblock(disk,BR_SECTOR_T(disk,onode_br));
```

```
179     onode = (struct fcfs_onode1*)onode_block->data;
180
181     if(forknr > onode->forks.leaf.nr)
182       {
183         fprintf(stderr,"Invalid Fork Number for onode\n");
184         abort();
185       }
186     if(content_length==0)
187       {
188         fprintf(stderr,"Must have some content to write\n");
189         abort();
190       }
191
192     if((pos+content_length) > onode->forks.leaf.fork[forknr].
          content_length)
193       {                                 /* Grow Onode Fork */
194         if(content_length > (disk->bsize - onode->forks.leaf.fork[
              forknr].content_length%disk->bsize) || onode->forks.leaf
              .fork[forknr].content_length%disk->bsize==0)
195           {
196 #ifdef DEBUG_ONODE_WRITE
197             fprintf(stderr,"GROWING in WRITE\n");
198 #endif
199             onode1_fork_grow(disk,onode_br,forknr,
200                              (pos+content_length
201                               - onode->forks.leaf.fork[forknr].
                                    content_length)
202                              / disk->bsize
203                              );
204           }
205         else
206           fprintf(stderr,"ENOUGH SPACE IN EXISTING BLOCK\n");
207       }
208
209     done_length = pos%disk->bsize;
210     do
211       {
212         //       fprintf(stderr,"DEBUG---- content %llu  done %llu\n
              ",content_length,done_length);
213
214         if(content_length<= disk->bsize)
```

```
215           if((disk->bsize - done_length%disk->bsize) <
                  content_length)
216             {//fprintf(stderr,"\n\n\t\t1\n\n");
217               write_length = disk->bsize - done_length%disk->bsize
                      ;}
218           else
219             {//fprintf(stderr,"\n\n\t\t2\n\n");
220               write_length = content_length;}
221         else
222           {//fprintf(stderr,"\n\n\t\t3\n\n");
223             write_length = disk->bsize - (done_length%disk->bsize)
                    ;}
224
225 #ifdef DEBUG_ONODE_WRITE
226         fprintf(stderr,"&&&&&&&&&&&&WRITE LENGTH = %llu, %llu, %llu\
              n",write_length,pos,done_length);
227 #endif
228
229         //        fprintf(stderr,"^^^^^^^^^^^^GETTING BLOCK = %llu\n",
              done_length/disk->bsize);
230         block = onode1_fork_getblock(disk,onode,forknr,(pos + (
              done_length - pos%disk->bsize)) / disk->bsize);
231
232         memcpy(block->data+(done_length%disk->bsize),content+(
              done_length - pos%disk->bsize),write_length);
233
234         onode->forks.leaf.fork[forknr].content_length+=write_length
              ;
235         content_length-=write_length;
236         done_length+=write_length;
237
238         disk_writeblock(block);
239         disk_freeblock(block);
240         //        fprintf(stderr,"!!!!!!!!!!!!!!LOOPING WITH %llu, %
              llu\n",content_length,done_length);
241       } while(content_length > 0);
242
243     //  fprintf(stderr,"\n\nDONE WRITE()\n\n");
244
245     disk_writeblock(onode_block);
246     disk_freeblock(onode_block);
247 }
```

```
248
249 u64 onode1_fork_read(struct fcfs_disk *disk, struct fcfs_onode1 *
    onode,int forknr, u64 pos, u64 content_length, void* content)
250 {
251   struct fcfs_disk_block *block;
252   u64 read_length;
253   u64 done_length;
254
255   if(forknr > onode->forks.leaf.nr)
256     {
257       fprintf(stderr,"Invalid Fork Number for onode\n");
258       abort();
259     }
260   if(content_length==0)
261     {
262       fprintf(stderr,"Must have some content to read\n");
263       abort();
264     }
265
266   if(pos > onode->forks.leaf.fork[forknr].content_length)
267     {
268       fprintf(stderr,"ERROR IN READ(): pos > content_length\n");
269       return 0;
270     }
271
272   if((pos+content_length) > onode->forks.leaf.fork[forknr].
       content_length)
273     {
274       fprintf(stderr,"ERROR IN READ(): pos+content_length >
           content_length\n");
275       content_length = onode->forks.leaf.fork[forknr].
           content_length - pos;
276     }
277
278   done_length = pos%disk->bsize;
279   do
280     {
281 #ifdef DEBUG_ONODE_READ
282       fprintf(stderr,"DEBUG---- content %llu done %llu\n",
           content_length,done_length);
283 #endif
284
```

```
285         if(content_length<=disk->bsize)
286           if((disk->bsize - done_length%disk->bsize) <
                 content_length)
287             {read_length = disk->bsize - done_length%disk->bsize;}
288           else
289             {read_length = content_length;}
290         else
291           {read_length = disk->bsize - (done_length%disk->bsize);}
292
293 #ifdef DEBUG_ONODE_READ
294         fprintf(stderr,"&&&&&&&&&&&READ LENGTH = %llu\n",
              read_length);
295
296         fprintf(stderr,"^^^^^^^^^^^GETTING BLOCK = %llu\n",
              done_length/disk->bsize);
297 #endif
298         block = onode1_fork_getblock(disk,onode,forknr,(pos + (
              done_length - pos%disk->bsize)) / disk->bsize);
299
300         memcpy(content+(done_length - pos%disk->bsize),block->data
              +(done_length%disk->bsize),read_length);
301
302         content_length-=read_length;
303         done_length+=read_length;
304
305         disk_freeblock(block);
306 #ifdef DEBUG_ONODE_READ
307         fprintf(stderr,"!!!!!!!!!!!!!!LOOPING WITH %llu, %llu\n",
              content_length,done_length);
308 #endif
309       } while(content_length > 0);
310 #ifdef DEBUG_ONODE_READ
311     fprintf(stderr,"\n\nDONE READ()\n\n");
312 #endif
313     return (done_length - pos%disk->bsize);
314 }
315
316
317 int onode1_fork_grow(struct fcfs_disk *disk, struct
       fcfs_block_run *onode_br, int forknr, u64 blocksnr)
318 {
319     struct fcfs_block_run *newbr;
```

```
320     struct fcfs_onode1 *onode;
321     struct fcfs_disk_block *block;
322     struct fcfs_disk_block *blockz;
323     int leafnr;
324     int i;
325
326     block = disk_getblock(disk,BR_SECTOR_T(disk,onode_br));
327     onode = (struct fcfs_onode1*)block->data;
328     if(test_bit(FCFS_FORK_InForkData,&(onode->forks.leaf.fork[
          forknr].fork_flags)))
329       {
330         /* We're changing a InFork Fork - and we shouldn't be! */
331         fprintf(stderr,"ERROR: Trying to grow InForkData fork!\n");
332         abort();
333       }
334     else
335       {
336         /* We're changing a normal fork */
337         if(onode->forks.leaf.fork[forknr].data.space.leaf.nr >=
              FSFS_ONODE1_SPACE_LEAF_MAXNR)
338           {
339             fprintf(stderr,"FIXME-ERROR: Growing fork beyond leaf
                  size\n");
340             abort();
341           }
342         else
343           {
344 #ifdef ONODE_GROW_DEBUG
345             fprintf(stderr,"ONODE GROW\n");
346 #endif
347             leafnr = onode->forks.leaf.fork[forknr].data.space.leaf
                  .nr;
348             //       printf("onode_fork_grow\n");
349             if(leafnr)
350               newbr = ag_allocate_block(disk,
351                                         onode->forks.leaf.fork[
                                              forknr].data.space.leaf.
                                              br[leafnr-1].
                                              allocation_group,
352                                         onode->forks.leaf.fork[
                                              forknr].data.space.leaf.
                                              br[leafnr-1].start,
```

```
353                                             //+onode->forks.leaf.fork[
                                                    forknr].data.space.leaf.
                                                    br[leafnr-1].len-1,
354                                             //onode_br->start,
355                                             blocksnr);
356             else
357               newbr = ag_allocate_block(disk,
358                                             onode_br->allocation_group,
359                                             300,
360                                             blocksnr);
361
362
363 #ifdef ONODE_GROW_DEBUG
364             fprintf(stderr,"\n\t\t\t\t\tFORKNR INFO %u %u %u %u\n\n
                    ",
365                     onode->forks.leaf.fork[forknr].data.space.leaf.
                        br[leafnr-1].allocation_group,
366                     onode->forks.leaf.fork[forknr].data.space.leaf.
                        br[leafnr-1].start,
367                     onode->forks.leaf.fork[forknr].data.space.leaf.
                        br[leafnr-1].len,newbr->start);
368 #endif
369
370             if(leafnr>0 &&
371                onode->forks.leaf.fork[forknr].data.space.leaf.br[
                    leafnr-1].allocation_group == newbr->
                    allocation_group
372                && onode->forks.leaf.fork[forknr].data.space.leaf.br
                    [leafnr-1].start
373                    + onode->forks.leaf.fork[forknr].data.space.leaf.
                        br[leafnr-1].len
374                == newbr->start
375                )
376               {                        /* Extend current block_run */
377 #ifdef ONODE_GROW_DEBUG
378                 fprintf(stderr,"Extended Existing Block Run! by %u\
                      n",newbr->len);
379 #endif
380                 onode->forks.leaf.fork[forknr].data.space.leaf.br[
                      leafnr-1].len+=
381                   newbr->len;
382               }
```

```
383             else
384               {
385                 onode->forks.leaf.fork[forknr].data.space.leaf.nr
                      ++;
386                 leafnr = onode->forks.leaf.fork[forknr].data.space.
                      leaf.nr;
387
388                 onode->forks.leaf.fork[forknr].data.space.leaf.br[
                      leafnr-1].allocation_group = newbr->
                      allocation_group;
389                 onode->forks.leaf.fork[forknr].data.space.leaf.br[
                      leafnr-1].start = newbr->start;
390                 onode->forks.leaf.fork[forknr].data.space.leaf.br[
                      leafnr-1].len = newbr->len;
391 #ifdef ONODE_GROW_DEBUG
392                 fprintf(stderr,"ONODE GROWN WITH BR: %u %u %u #%u\n
                      ",
393                         newbr->allocation_group,
394                         newbr->start,
395                         newbr->len,
396                         onode->forks.leaf.fork[forknr].data.space.
                            leaf.nr
397                         );
398 #endif
399               }
400             for(i=BR_SECTOR_T(disk,newbr);i<BR_SECTOR_T(disk,newbr)
                  +newbr->len;i++)
401               {
402                 blockz = disk_newblock(disk,i);
403                 //disk_writeblock(blockz); // we don't write, as we
                      're about
404                                          // to anyway (after
                                               return)
405                 disk_freeblock(blockz);
406               }
407             free(newbr);
408           }
409       }
410     disk_writeblock(block);
411     disk_freeblock(block);
412     return 1;
413 }
```

Listing 19: fcfs/onode_index.c

```
1  /* onode_index.c
2     ---------------
3     A casually demented B+Tree designed to stay balanced
4     and be optimized for access from disk (and in-core cache)
5
6     $Id: onode_index.c,v 1.15 2003/10/20 07:18:11 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10 */
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include "testkit/types.h"
15
16 #include "disk.h"
17 #include "super_block.h"
18 #include "space_bitmap.h"
19 #include "onode_index.h"
20
21 extern u64 next_free;
22
23 u64 onode_index_new_id(struct fcfs_onode_index *index)
24 {
25
26 }
27
28 struct fcfs_onode_index* onode_index_read(struct fcfs_disk *disk)
29 {
30   struct fcfs_onode_index *index;
31   struct fcfs_disk_block *block;
32
33   index = (struct fcfs_onode_index*)malloc(sizeof(struct
        fcfs_onode_index));
34   if(index==NULL)
35     {fprintf(stderr,"Cannot allocate onode index structure to
          read into\n");
36     abort();}
37   block = disk_getblock(disk,disk->sb->onode_index_blocknr);
38   index->root_block = block;
39   index->root = (struct fcfs_onode_index_node*)block->data;
```

```
40
41    index->disk = disk;
42
43    return index;
44  }
45
46  struct fcfs_onode_index* onode_index_new(struct fcfs_disk *disk)
47  {
48    struct fcfs_onode_index *index;
49
50    index = (struct fcfs_onode_index*)malloc(sizeof(struct
        fcfs_onode_index));
51    if(index==NULL)
52      {fprintf(stderr,"Cannot allocate onode index structure\n");
          abort();}
53
54    index->root = NULL;
55    index->disk = disk;
56
57    return index;
58  }
59
60  int onode_index_write_leaf(struct fcfs_onode_index *index, struct
       fcfs_onode_index_leaf *leaf)
61  {
62    struct fcfs_disk_block *block;
63
64    block = disk_getblock(index->disk,leaf->block);
65  #ifdef DEBUG_ONODE_INDEX
66    fprintf(stderr,"--WRITE LEAF--\n");
67  #endif
68    disk_writeblock(block);
69    disk_freeblock(block);
70
71    return 1;
72  }
73
74  struct fcfs_onode_index_leaf *onode_index_leaf_new(struct
      fcfs_onode_index *index, struct fcfs_onode_index_node *parent
      , int position)
75  {
76    int i;
```

```
77    struct fcfs_block_run *br;
78    struct fcfs_onode_index_leaf *leaf;
79    struct fcfs_disk_block *block;
80
81    //  printf("onode_index_leaf_new\n");
82    br = ag_allocate_block(index->disk,
83                           BLOCK_AG(index->disk,parent->block),
84                           parent->block%index->disk->sb->
                                ag_blocksnr,
85                           1);
86    block = disk_newblock(index->disk,BR_SECTOR_T(index->disk,br));
87    leaf = (struct fcfs_onode_index_leaf*)block->data;
88
89 #ifdef DEBUG_ONODE_INDEX
90    fprintf(stderr,"LEAF: AG: %d, Start: %d, Len: %d\n",br->
         allocation_group,br->start,br->len);
91 #endif
92
93    leaf->magic1 = FCFS_ONODE_INDEX_LEAF_MAGIC1;
94    leaf->id = index->disk->sb->onindex_next_id++;
95    leaf->block = BR_SECTOR_T(index->disk,br);
96    leaf->used = 0ULL;
97
98    for(i=0;i<index->disk->sb->onindex_node_size;i++)
99      { leaf->items[i].key = 0; leaf->items[i].onode_blocknr = 0; }
100
101   onode_index_write_leaf(index,leaf);
102
103   parent->items[position].node_blocknr = BR_SECTOR_T(index->disk,
         br);
104   if(parent->used < position)
105     parent->used = position;
106
107   onode_index_write_node(index,parent);
108
109   free(br);
110   return leaf;
111 }
112
113 struct fcfs_onode_index_node* onode_index_read_node(struct
      fcfs_onode_index *index, u64 blocknr)
114 {
```

```
115   struct fcfs_onode_index_node* node;
116   struct fcfs_disk_block *block;
117
118   block = disk_getblock(index->disk, blocknr);
119
120   node = (struct fcfs_onode_index_node*) block->data;
121
122   return node;
123 }
124
125 int onode_index_free_node(struct fcfs_onode_index *index, struct
      fcfs_onode_index_node* node)
126 {
127   struct fcfs_disk_block *block;
128   block = disk_getblock(index->disk,node->block);
129   disk_freeblock(block);
130   free(node);
131   return 1;
132 }
133
134 int onode_index_write_node(struct fcfs_onode_index *index, struct
      fcfs_onode_index_node* node)
135 {
136   struct fcfs_disk_block *block;
137
138   /* Write root to disk */
139   block = disk_getblock(index->disk,node->block);
140   disk_writeblock(block);
141 #ifdef DEBUG_ONODE_INDEX
142   fprintf(stderr,"--WRITE NODE--\n");
143 #endif
144   disk_freeblock(block);
145
146   return 1;
147 }
148
149 /* Writes root back to disk */
150 static inline int onode_index_write_root(struct fcfs_onode_index
      *index)
151 {
152   return onode_index_write_node(index,index->root);
153 }
```

```
154
155 /* The ever belated onode_index_new_node */
156 struct fcfs_disk_block *onode_index_new_node(struct
       fcfs_onode_index *index)
157 {
158   struct fcfs_block_run *br;
159   struct fcfs_disk_block *block;
160   struct fcfs_onode_index_node *node;
161   int i;
162
163   br = ag_allocate_block(index->disk,0,1,1);
164   if((br->len) < 1)
165     {
166       fprintf(stderr,"Unable to allocate Onode Index Node\n");
167       fprintf(stderr,"\tAG: %d, Start: %d, Len: %d\n",br->
             allocation_group,br->start,br->len);
168       abort();
169     }
170
171   fprintf(stderr,"Onode Index node created at: AG: %d, Start: %d
        , Len: %d\n",br->allocation_group,br->start,br->len);
172
173   block = disk_newblock(index->disk,BR_SECTOR_T(index->disk,br));
174   node = (struct fcfs_onode_index_node*)block->data;
175
176   node->magic1 = FCFS_ONODE_INDEX_NODE_MAGIC1;
177   node->id = index->disk->sb->onindex_next_id++;
178   node->used = 0ULL;
179   node->block = br->start;
180
181   for(i=0;i<index->disk->sb->onindex_node_size;i++)
182     { node->items[i].key = 0; node->items[i].node_blocknr = 0;}
183
184   return block;
185 }
186
187 /* Creates a new root node for an onode_index */
188 int onode_index_new_root(struct fcfs_onode_index *index)
189 {
190   struct fcfs_onode_index_node *node;
191   struct fcfs_disk_block *block;
192
```

```
193     block = onode_index_new_node(index);
194     index->root = (struct fcfs_onode_index_node*)block->data;
195     index->root_block = block;
196
197     fprintf(stderr,"****ONODE INDEX**** %llu\n",block->blocknr);
198
199     /* Move br to index structure */
200     /* Also copy into super block, and write to disk */
201     /* update the SB records */
202     index->disk->sb->onode_index_blocknr = block->blocknr;
203     index->disk->sb->onindex_free = index->disk->sb->
          onindex_node_size;
204     fcfs_write_sb(index->disk);
205     onode_index_write_root(index);
206
207     return 1;
208   }
209
210   int onode_index_leaf_insert(struct fcfs_disk *disk,struct
        fcfs_onode_index_leaf *leaf, u64 key, u64 value)
211   {
212     int pos;
213     int i;
214
215     if(leaf->used==disk->sb->onindex_node_size)
216       {                             /* Need to split leaf */
217         fprintf(stderr,"onode_index_leaf_insert: full
              onode_index_leaf. Growing of leaves not yet implemented\
              n");
218         abort();
219       }
220
221     if(key > leaf->items[leaf->used-1].key)
222       {                             /* Append to end of leaf */
223         leaf->used++;               /* FIXME: should be atomic */
224         leaf->items[leaf->used-1].key = key;
225         leaf->items[leaf->used-1].onode_blocknr = value;
226         return 1;
227       }
228     else
229       {
230         for(i=0;i<leaf->used && leaf->items[i].key < key;i++)
```

```
231           ;                             /* FIXME: Make binary search */
232         pos = i;
233         leaf->used++;
234         for(i=leaf->used;i>pos;i--) /* Shuffle everything down */
235           {
236             leaf->items[i].key = leaf->items[i-1].key;
237             leaf->items[i].onode_blocknr = leaf->items[i-1].
                  onode_blocknr;
238           }
239         leaf->items[pos].key = key; /* Add our key */
240         leaf->items[pos].onode_blocknr = value; /* Add our value */
241         return 1;                     /* Success! */
242       }
243
244     return 1;
245   }
246
247   /* Insert an onode into an index */
248   struct fcfs_block_run* onode_index_insert(struct fcfs_onode_index
        *index, struct fcfs_onode1 *onode)
249   {
250     struct fcfs_block_run *onode_br;
251     struct fcfs_disk_block *block;
252     struct fcfs_onode_index_node *node;
253     struct fcfs_onode_index_node *parent;
254     struct fcfs_onode_index_leaf *leaf;
255     int i,node_pos;
256
257     if(index->root==NULL)
258       onode_index_new_root(index);
259
260     /* Determine ID for onode */
261     //  onode->onode_num = ((~0ULL)/index->disk->sb->
          onindex_node_size) * (i+1);
262     onode->onode_num = random();
263     //index->disk->sb->onindex_next_id++;
264     fcfs_write_sb(index->disk);
265
266   #ifdef DEBUG_ONODE_INDEX
267     fprintf(stderr,"STORE ONODE ID: %llu\n",onode->onode_num);
268   #endif
269
```

```
270     /* Revision 1 now that we're going onto disk */
271     onode->onode_revision = 1;
272
273     /* Use count = 1, in onode_index */
274     onode->use_count = 1;
275
276     /* Allocate room for onode */
277     //  printf("onode_index_insert\n");
278     onode_br = ag_allocate_block(index->disk,0,10,1);
279
280 #ifdef DEBUG_ONODE_INDEX
281     fprintf(stderr,"ONODE created at: AG: %d, Start: %d, Len: %d\n"
            ,onode_br->allocation_group,onode_br->start,onode_br->len);
282 #endif
283
284     /* Write onode to disk */
285     block = disk_newblock(index->disk,BR_SECTOR_T(index->disk,
            onode_br));
286     //  memset(block->data,0,index->disk->bsize);
287     memcpy(block->data,onode,sizeof(struct fcfs_onode1));
288     disk_writeblock(block);
289 #ifdef DEBUG_ONODE_INDEX
290     fprintf(stderr,"ONODE Written\n");
291 #endif
292     disk_freeblock(block);
293
294     /* Put it in the index properly */
295     node = index->root;
296      parent = NULL;
297
298      /* Check current node for free space/location */
299      for(i=0;i<node->used && node->items[i].key < onode->onode_num;
            i++)
300        ;                               /* FIXME: This should be a binary
                search */
301
302      if(i==index->disk->sb->onindex_node_size)
303        i = 0;
304
305      node_pos = i;                              /* position in the node
            */
306
```

```
307 #ifdef DEBUG_ONODE_INDEX
308     fprintf(stderr,"\n\n\ti = %d\n\n",i);
309 #endif
310 
311     if(node->items[i].node_blocknr!=0)
312       {
313         /* Lookup subtree, see if node or leaf */
314         parent = node;
315         //       node = ;
316         block = disk_getblock(index->disk,node->items[i].
              node_blocknr);
317         if(*((u64*)block->data)== FCFS_ONODE_INDEX_LEAF_MAGIC1)
318           {                           /* is a leaf */
319 #ifdef DEBUG_ONODE_INDEX
320             fprintf(stderr,"FOUND A LEAF!\n\n");
321 #endif
322             leaf = (struct fcfs_onode_index_leaf *)block->data;
323 #ifdef DEBUG_ONODE_INDEX
324             fprintf(stderr,"GOING TO ADD TO LEAF 0x%llu\n",leaf->
                id);
325             fprintf(stderr,"\tblock %llu\n",leaf->block);
326             fprintf(stderr,"\tused  %llu\n",leaf->used);
327 #endif
328 
329             onode_index_leaf_insert(index->disk,leaf,
330                                     onode->onode_num,
331                                     BR_SECTOR_T(index->disk,
332                                                 onode_br));
333 
334             disk_writeblock(block);
335             parent->items[node_pos].key = onode->onode_num;
336             onode_index_write_node(index,parent);
337             index->disk->sb->onindex_used++;
338             index->disk->sb->onindex_free--;
339             fcfs_write_sb(index->disk);
340           }
341         else if(*((u64*)block->data)==
              FCFS_ONODE_INDEX_NODE_MAGIC1)
342           {
343             fprintf(stderr,"FOUND A NODE!\n\n");
344             fprintf(stderr,"**FIXME** We don't handle nodes yet\n"
                );
```

```
345           }
346         else
347           {
348             fprintf(stderr,"WARNING-CORRUPT: node pointer points
                   to corrupt node or leaf!\n");
349             fprintf(stderr,"\tCORRUPT node->items[%d].node_blocknr
                   = %llu\n",i,node->items[i].node_blocknr);
350             fprintf(stderr,"\tMAGIC is %llx\n",*((u64*)block->data
                   ));
351             abort();
352           }
353         disk_freeblock(block);
354       }
355     else
356       {
357         /* Create leaf */
358         leaf = onode_index_leaf_new(index, node,i);
359         leaf->items[0].key = onode->onode_num;
360         leaf->items[0].onode_blocknr = BR_SECTOR_T(index->disk,
              onode_br);
361         leaf->used = 1;
362
363 #ifdef DEBUG_ONODE_INDEX
364         fprintf(stderr,"LEAF: Should be writing with key=%lld,
              block=%lld\n",leaf->items[0].key,leaf->items[0].
              onode_blocknr);
365 #endif
366
367         node->items[i].node_blocknr = leaf->block;
368         node->items[i].key = onode->onode_num;
369         node->used = i+1;
370
371         onode_index_write_root(index);
372         onode_index_write_leaf(index,leaf);
373       }
374
375     return onode_br;
376   }
377
378 struct fcfs_disk_block *onode_index_lookup(struct
      fcfs_onode_index *index,struct fcfs_block_run *onode_br, u64
      id)
```

```
379 {
380   struct fcfs_disk_block *block;
381   struct fcfs_disk_block *onode_block;
382   struct fcfs_onode_index_node *node;
383   struct fcfs_onode_index_node *parent;
384   struct fcfs_onode_index_leaf *leaf;
385
386   int i;
387
388   node = index->root;
389
390   for(i=0;i<node->used && node->items[i].key < id ;i++)
391     ;
392
393   /* Lookup subtree, see if node or leaf */
394   parent = node;
395   //       node = ;
396   block = disk_getblock(index->disk,node->items[i].node_blocknr);
397   if(*((u64*)block->data)== FCFS_ONODE_INDEX_LEAF_MAGIC1)
398     {                       /* is a leaf */
399       fprintf(stderr,"FOUND A LEAF!\n\n");
400       leaf = (struct fcfs_onode_index_leaf *)block->data;
401       fprintf(stderr,"READ LEAF 0x%llu\n",leaf->id);
402       fprintf(stderr,"\tblock %llu\n",leaf->block);
403       fprintf(stderr,"\tused  %llu\n",leaf->used);
404       for(i=0;i<leaf->used && leaf->items[i].key < id;i++)
405         ;
406       fprintf(stderr,"leaf i = %d, %llu %llu\n",i,leaf->items[i].
            key,leaf->items[i].onode_blocknr);
407       onode_block = disk_getblock(index->disk,leaf->items[i].
            onode_blocknr);
408       onode_br->allocation_group = BLOCK_AG(index->disk,leaf->
            items[i].onode_blocknr);
409       onode_br->start = leaf->items[i].onode_blocknr%index->disk
            ->sb->ag_blocksnr;
410       onode_br->len = 1;
411       disk_freeblock(block);
412     }
413   else
414     abort();
415
416   return onode_block;
```

```
417 }
418
419 struct fcfs_onode_index* onode_index_delete(struct
      fcfs_onode_index* index)
420 {
421   if(index->root)
422     disk_freeblock(index->root_block);
423   free(index);
424   index = NULL;
425   return index;
426 }
```

Listing 20: fcfs/mount_testkit.c

```
1  /* mount_testkit.c
2     ---------------
3     $Id: mount_testkit.c,v 1.2 2003/09/22 09:03:26 stewart Exp $
4
5     This is the code to mount a fcfs volume using the testkit
          stuff.
6     a bit icky, but it should work.
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10  */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <sys/stat.h>
15 #include <unistd.h>
16 #include <fcntl.h>
17 #include <string.h>
18 #include <time.h>
19
20
21 #include "testkit/block_dev.h"
22 #include "testkit/types.h"
23 #include "testkit/bitops.h"
24 #include "disk.h"
25
26 #include "super_block.h"
27 #include "onode.h"
28 #include "onode_index.h"
29 #include "space_bitmap.h"
30
31
32 struct fcfs_disk *fcfs_mount(char *name)
33 {
34   struct block_device bdev,*bdevproper;
35   struct fcfs_disk *disk,*diskproper;
36   struct fcfs_disk_block *block;
37   u32 bsize;      // block size (bytes)
38   u64 blocksnr;   // number of blocks
39
```

```
40    bdevproper = (struct block_device*)malloc(sizeof(struct
          block_device));
41    if(bdevproper==NULL)
42      {
43        fprintf(stderr,"No memory for bdevproper\n");
44        abort();
45      }
46
47    block_dev_init();
48    block_dev_new(&bdev,name,4096,1);
49    disk = disk_new(&bdev);
50
51    block = disk_getblock(disk,0);
52    disk->sb = (struct fcfs_sb*)block->data;
53
54    bsize = disk->sb->bsize;
55    blocksnr = disk->sb->blocksnr;
56
57    /* clean up after initial detection */
58    disk_freeblock(block);
59    disk_free(disk);
60
61    /* Let's use our knowledge to make a proper disk */
62    block_dev_new(bdevproper,name,bsize,blocksnr);
63    diskproper = disk_new(bdevproper);
64
65    /* Put proper superblock into disk */
66    block = disk_getblock(diskproper,0);
67    disk->sb_block = block;
68    disk->sb = (struct fcfs_sb*)block->data;
69
70    fcfs_sb_mark_dirty(disk);
71
72    return disk;
73  }
74
75  int fcfs_umount(struct fcfs_disk *disk)
76  {
77    fcfs_sb_mark_clean(disk);
78    disk_freeblock(disk->sb_block);
79    disk_free(disk);
80    return 0;
```

```
81  }
```

Listing 21: fcfs/mkfile.c

```
1  /*
2     mkfile.c
3     ---------
4     Real stupid program to make an empty file.
5     Should take advantage of sparse stuff too!
6
7     $Id: mkfile.c,v 1.3 2003/10/12 12:58:53 stewart Exp $
8
9     (C)2003 Stewart Smith
10    Distributed under the GNU Public License
11   */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16 #include <sys/types.h>
17 #include <sys/stat.h>
18 #include <fcntl.h>
19
20 typedef u_int64_t u64;
21
22 u64 atou64(const char *nptr)
23 {
24   u64 out,i;
25   for(out=0,i=0;nptr[i]!='\0';i++)
26     {
27       out*=10;
28       out+=nptr[i]-'0';
29     }
30
31   return out;
32 }
33
34 int main(int argc,char* argv[])
35 {
36   FILE* file;
37
38   if(argc<4)
39     {
40       fprintf(stderr,"Usage:\n");
41       fprintf(stderr,"\t%s file bsize blocksnr\n\n",argv[0]);
```

```
42          exit(1);
43        }
44
45      file = fopen(argv[1],"w");
46      fseek(file,atou64(argv[2])*atou64(argv[3]),SEEK_SET);
47      fprintf(file,"%c",0);
48      fclose(file);
49      return 0;
50  }
```

Listing 22: fcfs/mkfs.c

```
1  /*
2     mkfs.c
3     ------
4     $Id: mkfs.c,v 1.23 2003/11/03 19:29:16 stewart Exp $
5     (C) 2003 Stewart Smith
6     Distributed under the GNU Public License
7
8     This is the 'mkfs' utility for FCFS - the new Walnut object
          store.
9     The name mkfs is kept purely 'cause it says what it does.
10
11    Some data structures have been constructed out of those
12    present in the Linux Kernel (v2.5.69). They are copyright
13    of their respective owners.
14   */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <sys/stat.h>
19 #include <unistd.h>
20 #include <fcntl.h>
21 #include <string.h>
22 #include <time.h>
23
24
25 #include "testkit/block_dev.h"
26 #include "testkit/types.h"
27 #include "testkit/bitops.h"
28 #include "disk.h"
29
30 #include "super_block.h"
31 #include "onode.h"
32 #include "onode_index.h"
33 #include "space_bitmap.h"
34
35 #define EXPERIMENTAL
36
37 u64 atou64(const char *nptr)
38 {
39    u64 out,i;
40    for(out=0,i=0;nptr[i]!='\0';i++)
```

```
41      {
42        out*=10;
43        out+=nptr[i]-'0';
44      }
45
46    return out;
47  }
48
49  int make_sb(void *sb_v, u32 bsize,u64 bcount, char* name)
50  {
51    struct fcfs_sb *sb = (struct fcfs_sb*) sb_v;
52
53    sb->magic1 = FCFS_SB_MAGIC1;
54    sb->version = FCFS_SB_VERSION1;
55    sb->bits = 64;
56    sb->sblength = sizeof(struct fcfs_sb);
57    sb->flags = 0;
58    set_bit(FCFS_FLAG_JournalMeta,&(sb->flags));
59  #ifdef EXPERIMENTAL
60    set_bit(FCFS_FLAG_Experimental,&(sb->flags));
61  #endif
62
63    sb->bsize = bsize;
64    sb->blocksnr = bcount;
65    strcpy(sb->name,name);
66    sb->magic2 = FCFS_SB_MAGIC2;
67
68    sb->time_created = time(NULL);
69    sb->time_clean = time(NULL);
70    sb->time_dirtymount = 0x0ULL;
71
72    sb->num_mounts = 0x0ULL;
73    sb->num_dirtymounts = 0x0ULL;
74
75    /* We simply steal the number of allocation groups from XFS. :)
76       They use 8, so we'll steal that.
77     */
78    sb->allocation_groupsnr = 8;
79    sb->ag_blocksnr = bcount / sb->allocation_groupsnr;
80
81    fprintf(stderr,"ALLOCATION GROUPS: 0x%x, each 0x%x\n",
82            sb->allocation_groupsnr,
```

```
83              sb->ag_blocksnr);
84
85      sb->onindex_num_onodes = 0;     /* Num onodes in index */
86      sb->onindex_used = 0;           /* Num of used inodes in index */
87      sb->onindex_free = 0;           /* Num of free onodes in index */
88      sb->onindex_next_id = 0;        /* Next ID to use */
89
90      /* FIXME: Potentially do real nasty things that can't be
            described
91       * in polite circles when our structs change size... */
92      sb->onindex_node_size =  ((sb->bsize - 4*sizeof(u64)) / sizeof(
            u64))/2;
93      /* In number of keys */
94
95      return 1;                       /* success, although we don't
            check it*/
96  }
97
98  int write_superblocks(struct fcfs_disk* disk)
99  {
100     struct fcfs_disk_block *block;
101     u32 i;
102
103     /* Write start of volume master SB */
104     block = disk_getblock(disk,0);
105
106     disk->sb->flags = disk->sb->flags | FCFS_SBloc_start_volume;
            /* start of volume*/
107
108     disk_writeblock(block);
109     disk_freeblock(block);
110     fprintf(stderr,"SuperBlock at Block 0\n");
111
112     /* Write end of volume Backup SB */
113     block = disk_newblock(disk,disk->blocksnr-1ULL);
114
115
116     //  disk->sb->flags = disk->sb->flags & ~0x03ULL;
117     disk->sb->flags = disk->sb->flags ^ FCFS_SBloc_start_volume;
            /* start of volume*/
118     disk->sb->flags = disk->sb->flags | FCFS_SBloc_end_volume;  /*
            end of volume*/
```

```
119
120     memcpy(block->data,disk->sb,disk->bsize);
121
122     disk_writeblock(block);
123     disk_freeblock(block);
124     fprintf(stderr,"BACKUP SuperBlock at Block 0x%llx\n",disk->
          blocksnr-1ULL);
125
126     disk->sb->flags = disk->sb->flags ^ FCFS_SBloc_end_volume;  /*
          end of volume*/
127
128     for(i=1;i<disk->sb->allocation_groupsnr;i++)
129       {
130         block = disk_newblock(disk,i*disk->sb->ag_blocksnr);
131
132         disk->sb->flags = disk->sb->flags | FCFS_SBloc_start_ag;
              /* start of ag*/
133         memcpy(block->data,disk->sb,disk->bsize);
134         disk_writeblock(block);
135         disk_freeblock(block);
136         fprintf(stderr,"BACKUP SuperBlock at Block 0x%llx (offset 0
              x%llx)\n",i*disk->sb->ag_blocksnr,(i*disk->sb->
              ag_blocksnr)*disk->bsize);
137       }
138     return 0;
139   }
140
141   int write_block_bitmaps(struct fcfs_disk* disk)
142   {
143     struct fcfs_disk_block *block;
144     struct fcfs_block_run br;
145     int i,j;
146
147     /* Create initial empty Block Bitmap */
148     for(i=0;i<(disk->sb->allocation_groupsnr);i++)
149       {
150         /* For each ag, blank one of the right size */
151         for(j=0;j<space_bitmap_size(disk->sb,i);j++)
152           {
153             block = disk_newblock(disk,i*disk->sb->ag_blocksnr+2);
154             memset(block->data,0,disk->bsize);
```

```
155             fprintf(stderr,"BITMAP: Zeroing 0x%x (offset %x) length
                    %u\n",
156                     i*disk->sb->ag_blocksnr+2,
157                     (i*disk->sb->ag_blocksnr+2)*disk->bsize,
158                     space_bitmap_size(disk->sb,i));
159             disk_writeblock(block);
160             disk_freeblock(block);
161           }
162       }
163
164     for(i=0;i<(disk->sb->allocation_groupsnr);i++)
165       {
166         /* Setup our Block Run */
167         br.allocation_group = i;
168         br.start = 1;
169         br.len = space_bitmap_size(disk->sb,i);
170
171         space_bitmap_allocate_block(disk,i,0);
172
173         for(j=0;j<space_bitmap_size(disk->sb,i);j++)
174           space_bitmap_allocate_block(disk,i,j+1);
175
176       }
177
178     return 0;
179 }
180
181 int main(int argc,char* argv[])
182 {
183     struct block_device bdev;
184     struct fcfs_disk *disk;
185     struct fcfs_disk_block *block;
186     struct fcfs_sb *sb;
187     struct fcfs_onode_index *index;
188     u64 bsize;
189     u64 bcount;
190     char* buffer;
191     int i;
192
193     if(argc<5)
194       {
195         fprintf(stderr,"FCFS Make Object Store Utility (mkfs)\n");
```

```
196         fprintf(stderr,"----------------------------------------\n");
197         fprintf(stderr,"$Id: mkfs.c,v 1.23 2003/11/03 19:29:16
               stewart Exp $\n\n");
198         fprintf(stderr,"Written by: Stewart Smith (
               stewart@flamingspork.com)\n\n");
199         fprintf(stderr,"Usage:\n\t./mkfs device blocksize
               blockcount name [initial objects]\n\n");
200         exit(0);
201       }
202
203     bsize =  atou64(argv[2]);
204     bcount = atou64(argv[3]);
205
206     fprintf(stderr,"Going to create volume '%s' with 0x%llx blocks
          at 0x%llx bytes each\n\n",argv[4],bcount,bsize);
207
208     block_dev_init();
209     block_dev_new(&bdev,argv[1],bsize,bcount);
210     disk = disk_new(&bdev);
211
212     if((buffer = (char*)malloc(sizeof(char*)*bsize))==0)
213       {
214         fprintf(stderr,"Unable to allocate buffer\n");
215         abort();
216       }
217
218     /* Make a new Super Block */
219     block = disk_newblock(disk,0);
220     sb = (struct fcfs_sb*)block->data;
221
222     if(!make_sb(sb,disk->bsize,disk->blocksnr,argv[4]))
223       {
224         fprintf(stderr,"Make_sb failed\n");
225         abort();
226       }
227
228     disk->sb = sb;
229
230     write_superblocks(disk);
231
232     write_block_bitmaps(disk);
233
```

```
234     index = onode_index_new(disk);
235     onode_index_new_root(index);
236
237     for(i=5;i<argc;i++)
238     {
239       struct fcfs_onode1 *node;
240       struct fcfs_block_run *onode_br;
241       int forknr;
242       char data[1024]; int j;
243       FILE* a;
244
245       node = onode1_new(disk);
246       onode_br = onode_index_insert(index,node);
247       //     onode1_grow(disk,onode_br,10);
248       //     onode1_grow(disk,onode_br,20);
249       onode1_fork_new(disk,onode_br,0x42,strlen(argv[i]),argv[i],1)
              ;
250       forknr = onode1_fork_new(disk,onode_br,0x69,0,NULL,0);
251       fprintf(stderr,"+++++++GOING TO GO AND WRITE %s++++++++\n",
              argv[5]);
252       a = fopen(argv[i],"r");
253       j=0;
254       while(!feof(a))
255         {
256           fgets(data,1000,a);
257           fprintf(stderr,"*******WRITING %d bytes*******\n",strlen(
                data));
258           onode1_fork_write(disk,onode_br,forknr,j,(u64)strlen(data
                ),data);
259           j+=strlen(data);
260         }
261       fclose(a);
262     }
263
264     disk_freeblock(block);
265
266     /* Clean up and exit */
267     block_dev_close(&bdev);
268
269     return 0;
270   }
```

Listing 23: fcfs/volinfo.c

```
1  /* volinfo.c
2     ---------
3
4     Dumps a lot of information about a volume to the console.
5
6     $Id: volinfo.c,v 1.7 2003/09/22 09:05:18 stewart Exp $
7
8     (C)2003 Stewart Smith
9     Distributed under the GNU Public License
10 */
11
12 // Block Device includes
13 #include "testkit/types.h"
14 #include "testkit/block_dev.h"
15 #include "testkit/bitops.h"
16 #include "disk.h"
17
18 // UNIX includes
19 #include <stdio.h>
20 #include <stdlib.h>
21 #include <sys/stat.h>
22 #include <unistd.h>
23 #include <fcntl.h>
24 #include <string.h>
25
26 // FCFS Includes
27 #include "super_block.h"
28 #include "onode.h"
29 #include "onode_index.h"
30 #include "space_bitmap.h"
31
32 void print_sb_flags(struct fcfs_sb *sb, int desired_loc)
33 {
34   if((sb->flags & 0x03)!=desired_loc)
35     printf("WARNING-CORRUPT: ");
36   if((sb->flags & 0x03) == FCFS_SBloc_start_volume)
37     printf("This is the start of a volume\n");
38   else
39     if((sb->flags & 0x03) == FCFS_SBloc_end_volume)
40       printf("This is the End of a volume\n");
41     else
```

```
42        if((sb->flags & 0x03) == FCFS_SBloc_start_ag)
43          printf("This is the start of an Allocation Group\n");
44        else
45          printf("Volume is corrupt - unknown Super Block Location\
              n");
46
47    printf("FLAGS: 0x%llx\n",sb->flags);
48
49    if(test_bit(FCFS_FLAG_Dirty,&sb->flags))
50      printf("\t- Volume is Dirty\n");
51    if(test_bit(FCFS_FLAG_Experimental,&sb->flags))
52      printf("\t- Volume has been used by EXPERIMENTAL code. It's
            probably b0rked.\n");
53    if(test_bit(FCFS_FLAG_JournalMeta,&sb->flags))
54      printf("\t- Volume journals Meta Data\n");
55    if(test_bit(FCFS_FLAG_JournalData,&sb->flags))
56      printf("\t- Volume journals Data\n");
57    if(test_bit(FCFS_FLAG_Versioned,&sb->flags))
58      printf("\t- Volume is Versioned\n");
59  }
60
61  int print_sb(struct fcfs_disk *disk)
62  {
63    struct fcfs_disk_block *block, *block2;
64    struct fcfs_sb *sb, *sb2;
65
66
67    block = disk_getblock(disk,0);
68    sb = (struct fcfs_sb*)block->data;
69    if(sb->magic1!=FCFS_SB_MAGIC1)
70      fprintf(stderr,"WARNING: Primary SB MAGIC1 mismatch. Corrupt
            Volume.\n");
71    if(sb->magic2!=FCFS_SB_MAGIC2)
72      fprintf(stderr,"WARNING: Primary SB MAGIC2 mismatch. Corrupt
            Volume.\n");
73    if(sb->bits!=64)
74      fprintf(stderr,"WARNING: Weird number of filesystem base bits
            . Corrupt Volume.\n");
75    if(sb->version!=FCFS_SB_VERSION1)
76      fprintf(stderr,"WARNING: Unknown Version of FS & SB.\n");
77
78    printf("Primary Superblock\n");
```

```
79    printf("----------\n");
80    printf("Volume name is: %s\n",sb->name);  /* FIXME: CHECK NAME
          !!! */
81    printf("MAGIC1:  0x%x\tMAGIC2: 0x%x\nVERSION: 0x%x\tBITS: %u\t
          LENGTH: %u bytes\n",sb->magic1,sb->magic2,sb->version,sb->
          bits,sb->sblength);
82    printf("Block Size: %u\tNo. Blocks: %llu\n",sb->bsize,sb->
          blocksnr);
83
84    // sb flags
85    print_sb_flags(sb,FCFS_SBloc_start_volume);
86
87    printf("Number of Clean Mounts:   %llu\n",sb->num_mounts);
88    printf("Number of UnClean Mounts: %llu\n",sb->num_dirtymounts);
89    printf("Time Created:             %llu\n",sb->time_created);
90    printf("Time last cleanly mounted:%llu\n",sb->time_clean);
91    printf("Time last dirty mounted:  %llu\n",sb->time_dirtymount);
92
93    printf("\n");
94
95    printf("Onode Index:\n");
96    printf("------------\n");
97    printf("ONode Index location: %llu\n",sb->onode_index_blocknr);
98    printf("Number of onodes: %llu used, %llu available, %llu total
          \n",
99          sb->onindex_used,
100         sb->onindex_free,
101         sb->onindex_num_onodes);
102   printf("Next ID: %llu\tNode Size: %lu\n",
103         sb->onindex_next_id,
104         sb->onindex_node_size);
105
106
107   block2 = disk_getblock(disk,disk->blocksnr-1);
108   sb2 = (struct fcfs_sb*)block2->data;
109
110   printf("\n\nChecking end of disk super block backup...\n");
111
112   if(sb2->magic1!=sb->magic1)
113     printf("WARNING-CORRUPT: End of disk SB magic1 is incorrect 0
          x%x\n",sb2->magic1);
114   if(sb2->magic2!=sb->magic2)
```

```
115       printf("WARNING-CORRUPT: End of disk SB magic2 is incorrect 0
              x%x\n",sb2->magic2);
116     if(sb2->version!=sb->version)
117       printf("WARNING-CORRUPT: End of disk SB version differs 0x%x\
              n",sb2->version);
118     if(sb2->bits!=sb->bits)
119       printf("WARNING-CORRUPT: End of disk SB bits differ %u\n",sb2
              ->bits);
120     if(sb2->sblength!=sb2->sblength)
121       printf("WARNING-CORRUPT: End of disk SB length differs %u\n",
              sb2->sblength);
122
123     print_sb_flags(sb2,FCFS_SBloc_end_volume);
124
125     if((sb2->flags & (~0x03ULL))!=(sb->flags & (~0x03ULL)))
126       {
127         printf("WARNING-CORRUPT: End of disk SB flags differ\n");
128         printf("End of Disk SB FLAGS: 0x%llx  0x%llx\n",sb2->flags
                ,(sb2->flags & (~0x03ULL)));
129       }
130
131     if(sb2->bsize!=sb->bsize)
132       printf("WARNING-CORRUPT: Block size differs %u vs %u\n",sb->
              bsize, sb2->bsize);
133     if(sb2->blocksnr!=sb->blocksnr)
134       printf("WARNING-CORRUPT: Number of blocks differs %llu vs %
              llu\n",sb->blocksnr,sb2->blocksnr);
135
136     if(strcmp(sb2->name,sb->name)!=0)
137       printf("WARNING-CORRUPT: Volume names differ.\n");
138
139     disk_freeblock(block2);
140
141     disk_freeblock(block);
142   }
143
144   int print_used_blocks(struct fcfs_disk *disk)
145   {
146     int ag,blk;
147     struct fcfs_disk_block *block;
148     struct fcfs_sb *sb;
149
```

```
150     block = disk_getblock(disk,0);
151     sb = (struct fcfs_sb*)block->data;
152
153     printf("Checking used blocks...\n");
154     printf("-------------------------");
155
156     for(ag=0;ag<sb->allocation_groupsnr;ag++)
157       {
158         printf("\nAG %u: ",ag);
159         for(blk=0;blk<sb->ag_blocksnr;blk++)
160           if(!ag_block_free(disk,ag, blk))
161             printf("%u,",blk);
162       }
163     printf("\n");
164     return 0;
165 }
166
167 int print_onode_index_leaf(struct fcfs_disk *disk,u64 blocknr)
168 {
169     struct fcfs_disk_block *block;
170     struct fcfs_onode_index_leaf *leaf;
171     int i;
172
173     block = disk_getblock(disk,blocknr);
174     leaf = (struct fcfs_onode_index_leaf *)block->data;
175
176     if(leaf->magic1!=FCFS_ONODE_INDEX_LEAF_MAGIC1)
177       printf("WARNING-CORRUPT: Bad onode_index_leaf MAGIC1\n");
178     printf("\tOnode Index LEAF: %llu in block %llu\n",leaf->id,leaf
          ->block);
179     printf("\tUsed: %llu/%lu\n",leaf->used,disk->sb->
          onindex_node_size);
180     printf("\t   key:block\n");
181     for(i=0;i<leaf->used;i++)
182       printf("\t   %llu:%llu\n",leaf->items[i].key,
183              leaf->items[i].onode_blocknr);
184
185     disk_freeblock(block);
186     return 0;
187 }
188
189 int print_onode_index(struct fcfs_disk *disk)
```

```
190 {
191   struct fcfs_disk_block *block;
192   struct fcfs_onode_index_node *index_node;
193   int i;
194
195   block = disk_getblock(disk,disk->sb->onode_index_blocknr);
196
197   index_node = (struct fcfs_onode_index_node*)block->data;
198   if(index_node->magic1!=FCFS_ONODE_INDEX_NODE_MAGIC1)
199     printf("WARNING-CORRUPT: Bad onode_index_node MAGIC1\n");
200   printf("Onode Index Node: %llu in block %llu\n",index_node->id,
        index_node->block);
201   if(disk->sb->onode_index_blocknr!=index_node->block)
202     printf("WARNING-CORRUPT: onode_index_node->block != sb->
          onode_index_blocknr\n");
203   printf("Used: %llu/%lu\n",index_node->used,disk->sb->
        onindex_node_size);
204   printf("  key:block\n");
205   for(i=0;i<index_node->used;i++)
206     printf("  %llu:%llu\n",index_node->items[i].key,
207            index_node->items[i].node_blocknr);
208
209   for(i=0;i<index_node->used;i++)
210     print_onode_index_leaf(disk,index_node->items[i].node_blocknr
          );
211
212   disk_freeblock(block);
213   return 0;
214 }
215
216 int main(int argc, char* argv[])
217 {
218   struct block_device bdev;
219   struct fcfs_disk *disk;
220   struct fcfs_disk_block *block;
221
222   if(argc<4)
223     {
224       fprintf(stderr,"FCFS Volume Info Utility (volinfo)\n");
225       fprintf(stderr,"-----------------------------------\n");
226       fprintf(stderr,"$Id: volinfo.c,v 1.7 2003/09/22 09:05:18
            stewart Exp $\n\n");
```

```
227         fprintf(stderr,"Written by: Stewart Smith (
                stewart@flamingspork.com)\n\n");
228         fprintf(stderr,"Usage:\n\t./volinfo device blocksize
                blockcount\n\n");
229         exit(0);
230       }
231
232     block_dev_init();
233     block_dev_new(&bdev,argv[1],atoi(argv[2]),atoi(argv[3]));
234     disk = disk_new(&bdev);
235
236     block = disk_getblock(disk,0);
237     disk->sb = (struct fcfs_sb*)block->data;
238
239     print_sb(disk);
240
241     print_used_blocks(disk);
242
243     print_onode_index(disk);
244
245     /* Clean up and exit */
246     block_dev_close(&bdev);
247
248     return 0;
249 }
```

Listing 24: fcfs/fcfs_newobj.c

```
1  /*
2     fcfs_newobj.c
3     ---------------
4
5     Program that inserts an object onto an FCFS volume
6     from a unix file.
7
8     $Id: fcfs_newobj.c,v 1.5 2003/10/20 07:18:11 stewart Exp $
9
10    (C)2003 Stewart Smith
11    Distributed under the GPL
12 */
13
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <sys/stat.h>
17 #include <unistd.h>
18 #include <fcntl.h>
19 #include <string.h>
20 #include <time.h>
21
22
23 #include "testkit/block_dev.h"
24 #include "testkit/types.h"
25 #include "testkit/bitops.h"
26 #include "disk.h"
27
28 #include "super_block.h"
29 #include "onode.h"
30 #include "onode_index.h"
31 #include "space_bitmap.h"
32 #include "fcfs_vfs.h"
33
34 #define EXPERIMENTAL
35
36 int main(int argc, char* argv[])
37 {
38    struct fcfs_disk* disk;
39    struct fcfs_onode_index* index;
40    int i;
41
```

```
42    if(argc<3)
43      {
44        fprintf(stderr,"fcfs_newobj\n");
45        fprintf(stderr,"------------\n");
46        fprintf(stderr,"$Id: fcfs_newobj.c,v
              1.5 2003/10/20 07:18:11 stewart Exp $\n");
47        fprintf(stderr,"Usage:\n");
48        fprintf(stderr,"\t%s volume file [file2...]\n\n",argv[0]);
49        exit(1);
50      }
51
52    disk = fcfs_mount(argv[1]);
53
54    index = onode_index_read(disk);
55    for(i=2;i<argc;i++)
56    {
57      struct fcfs_onode1 *node;
58      struct fcfs_block_run *onode_br;
59      int forknr;
60      char data[8200]; int j;
61      int infile;
62      int rlen;
63      struct stat statbuf;
64
65      node = onode1_new(disk);
66      onode_br = onode_index_insert(index,node);
67
68      infile = open(argv[i],O_RDONLY);
69      fstat(infile,&statbuf);
70
71      /* UNIX source File name fork  */
72      onode1_fork_new(disk,onode_br,0x42,strlen(argv[i]),argv[i],1)
            ;
73
74      /* Current revision fork */
75      forknr = onode1_fork_new(disk,onode_br,0x69,statbuf.st_size,
            NULL,0);
76      onode1_fork_new(disk,onode_br,0x6A,0,NULL,0); /* Revision
            history fork */
77      onode1_fork_new(disk,onode_br,0x6B,0,NULL,0); /* Revision
            history data fork */
78
```

```
79      fprintf(stderr,"%s in fork %d\n",argv[i],forknr);
80
81      j=0;
82      while(0 < (rlen = read(infile,data,8192)))
83        {
84          //      fprintf(stderr,"*******WRITING %d bytes*******\n
                    ",rlen);
85          onode1_fork_write(disk,onode_br,forknr,j,(u64)rlen,data);
86          j+=rlen;
87        }
88      close(infile);
89      free(node);
90      free(onode_br);
91    }
92
93    fcfs_umount(disk);
94
95    return 0;
96  }
```