CHUNKFS: A RECOVERY-DRIVEN FILE SYSTEM DESIGN APPROACH

by

AMIT GUD

Bachelor of Engineering, University of Pune, India, 2004

A THESIS

submitted in partial fulfillment of the
requirements for the degree

Master of Science

Department of Computing & Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2007

Approved by:

Major Professor
Dr. Daniel Andresen

Abstract

The recovery aspect of the file system has been largely ignored during file system design. Recovery of file system is becoming more and more significant as file system checking time is increasing due to increase in disk capacities with relatively low increase in disk bandwidth and insignificant improvement in seek time. With current file systems and the disk capacity trends, the file system checking time is likely to go up by a factor of 10 by 2013. Journaling and soft-updates avoid file system checks but only for incomplete metadata updates during a file system crash. Errors due to operating system and file system bugs and due to hardware malfunction, make file system checking unavoidable.

We take a recovery driven file system design approach by dividing the file system into small chunks, which are complete file systems by themselves. The spanning of the files and directories across multiple chunks is addressed in such a way that there is minimal cross-chunk dependencies thus allowing skipping of non-dirty chunks when checking the file system. This way the amount of file system that needs checking is reduced down to the number of chunks that are dirty. We explore what should be the chunk size for a given file system size, which is a trade-off between file system performance, in seek times, and reduction in file system checking time. We also explore the amount of overhead incurred for continuation of files across chunks in terms of number of continuations.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

"If I have seen further it is by standing on the shoulders of giants."

– Isaac Newton, Letter to Robert Hooke, February 5, 1675

It would be an understatement if I say I would like to extend my profound thanks to my major Professor Dr. Daniel Andresen for all the support and guidance throughout my Masters and thesis. I have learned a lot from him, both on the research and non-technical side, and all that will help me in my future. And I would also like to thank him for being generous for allowing me access to Nichols 119 lab, where I spent most of my Masters life.

I would like to thank my committee members, Dr. Gurdip Singh and Dr. Mitchell Neilsen for being patient while I worked on the thesis topic and also for supporting me during my Masters. Special thanks for the excellent graduate-level courses they offered.

I would like to extend sincere thanks to Valerie Henson for having faith in me while I explored the topic in the thesis further. Without her constant support and positive energy this work wouldn't have been possible.

I owe a lot to Rik van Riel for having enormous confidence in me and this idea. I was glad to have interacted with him on numerous occasions discussing ideas and learning. He was the guy I would run to whenever I had problems. I thank him for all his support. I also thank him for introducing me to number of Linux file system folks. I would also like to thank Arjan van de Ven, Zach Brown, Peter Staubach, Garth Gibson, and Stephen Tweedie for sharing their thoughts on this topic. All of the input was crucial.

Also thanks to Dave Jones and his blog that allowed me to have a wonderful summer in Boston interning with Red Hat, Inc., from where this work fruitioned.

On personal side, I would like to thank my parents, to whom this work is dedicated, for their unparalleled care and love. I hope, with all my work throughout my Masters, I'm able to make up for at least something for the time that I could not spend with you. Also I would like to thank rest of my family for all the laughter and being with me during all my times throughout my education so far.

I have thanked just a small fraction of people who have been instrumental for shaping my career so far and I ask forgiveness from those who have been omitted unintentionally.

Thank you all!

To my parents.

Chapter 1

Introduction

"If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time"

– Shimon Peres [3]

This chapter provides an introduction to file systems and its recovery. It also provides an overview of the dissertation, motivation for the work and its contributions.

1.1 Data Integrity & Recovery

File system is an integral part of an *operating system* (OS). Every data that is stored on the disk goes through a file system to be stored onto a set of *block*s [1] on the disk, collectively called as a *file*. Each file is associated with an on-disk metadata data structure traditionally called as *inode*. File Systems are required to store files and allow recovery of saved data at a later time. This component of the OS is crucial as it needs to abide by the following fundamental properties:

- allow a consistent interface to the data on the disk, by means of entities like files and directories

[1] a *block* is lowest logical I/O unit with which the file system driver communicates with the disk. It may or may not be equal to the size of the sector on the disk.

- provide uniquely identifiable namespace to the data on the disk, with the help of file names for example
- maintain the integrity of the data stored on the disk

While the properties above sounds more of a common sense, one issue that stands out and is completely uncompromisable is data integrity. Whatever data that goes into a file on the disk, same data is expected when the file is read back. The integrity of the data on the file system is at stake either due to mishandling of data, including a security breach, or due to system malfunction. Various techniques have been adopted to deal with the integrity of the file system in both such scenarios [4, 5]. In this document we will focus ourselves to file system damage due to system malfunction.

Malfunction could happen due to various reasons, like power failure, or OS crash, or file system driver bug or due to a hardware error. Whenever such malfunction occurs, there are chances that the file system is left in an unknown state. For example, half-writes, in which the size of the file on the disk is updated, but pointers to its data block are not, or vice-versa. On a possibility of a file system corruption, a file system scan needs to be carried out in order to *recover* [2] the file system back to a known consistent state. This is done usually by taking the disk offline and running a file system checking program designed for the particular file system that is on the disk. The file system checker reads every piece of metadata of the file system and checks for any anomaly. For example, if a link count is maintained in the inodes (which usually is) specifying how many other files refer to this file, then those number of files should actually refer to this file in some way. Another example could be the size of the file. The size, as specified in the inode, should match up with its calculated size depending on the number of data blocks it has[3]. Thus more the used metadata

[2]Terms 'recovery' and 'repair' are treated synonymous and are used interchangeably throughout this thesis

[3]The file size could be less than the number of blocks it has, multiplied by the file system block size.

on the file system, more time it will take to recover the file system. Thus the time complexity for a full file system scan is of the order of $\boldsymbol{O}$*(amount of used metadata on the file system).*

1.2 Thesis Objective

The objective of this work is to reduce the file system checking time. This problem is approached by adapting recovery-driven techniques at design-time of the file system, which allow speedy recovery when required.

1.3 Motivation

There are two approaches that can be taken to avoid spending time over the file system check. First is to completely eliminate the file system checks by not letting file system to go in an inconsistent state. And second, to reduce the file system checking time. The file systems taking the former approach deploy mechanisms like *journaling* [6] or $soft - updates$ [7]. But even these proactive measures prove insufficient and file system checks become inevitable mainly because of imperfect hardware [8], file system bugs [9], and operating system failures. There also exist *online* file system checking methodologies [10], but it requires expensive soft-update technique and for actual repair the file system has to be offline. Furtheremore, it does not reduce the file system checking time, which is the main objective of this study.

Thus said, there is no escape from occasional file system checks. And whenever it is required, the time required for checking directly accounts for the availability of the data and effectively the downtime.

Following subsections takes a closer look at why file system checking is becoming more frequent and more time consuming.

	2006	2009	2013	Change
Capacity (GB)	500	2000	8000	16x
Bandwidth (Mb/s)	1000	2000	5000	5x
Seek time (ms)	8	7.2	6.5	1.2x

Table 1.1: *Projected disk hardware trends [2]*

1.3.1 Increasing Disk Size

Table 1.1 shows predicted disk manufacturing projections over the coming years by Seagate Technology. It can be seen that the capacity of disk is growing at a much rapid pace than the disk I/O bandwidth. This entails more time to read the entire disk, since now the capacity has increased but the capacity of the I/O pipe reading and writing to the disk is not keeping up. Also since the seek times are almost going to be constant for the coming 6 years, it is unlikely that improvement in seek time is going to be of much help. File system checking time being proportional to the amount of used metadata and effectively the size of the disk, alarms of a situation in which file system check takes up more time than it does today. The analogy we use for this is as if the size of your milk shake is increased by a factor of 3, but your straw size stayed the same. With the projections mentioned in table 1.1, file system checking time would increase approximately by a factor of 10 by 2013 [11].

1.3.2 Increasing Per-Bit Error Rate

In future, not only the file system checking would take more time, it will also be required more frequently. This can be realized taking into account the disk failures and the error rates. Apart from *mean time to failure* and other popular metrics, *annualized failure rate* (AFR), which is the percentage of the disk drives fail scaled to a per year estimation, is often used by the disk drive manufacturers to specify disk drive failure rate [12]. AFR for the highest quality disks today is around 0.58% to 0.88% [13]. Another interesting metric, which is relevant to this study, is the *per-bit*

error rate [14], which is total number of errors scaled to the life-time of the disk divided by the total number of storage bits on the disk drive.

Though the per-bit error rate is improving, the improvement is not quite enough. Every time the disk capacity doubles, the per-bit error rate must be cut in half to keep the overall errors per bit constant; but it is not happening. The disk manufacturing companies specifies one uncorrectable read error rate (UER) every 10^{13} to 10^{16} bits read [8]. This specification is, however, under-estimated as compared with the actual results [8].

Even if it is assumed that the errors per square inch of the disk drive platter remains constant between now and 2013, a single error in 2013 would cause more damage to the data in the disk as more data will now be cramped per square inch of the platter, thus increasing UER.

This indicates that file system checks would be more often than today with the increase in disk capacity.

1.4 Approach In Brief

To reduce the file system checking time, the whole file system is divided into smaller parts, henceforth known as *chunks*. Each chunk is a file system in itself. It has characteristics as that of a regular file system. So file system is now a set of smaller file systems allowing fault isolation and reducing the work of the file system checker if it is able to determine exactly which chunks were *active* (or marked *dirty*) while the file system crashed, and only check those.

Now since the file system is being divided into smaller chunks, special care needs to be taken to allow seamless growth of files beyond the size of the chunk and also allow cross-chunk links, so that directory in one chunk can have entries of inodes in another chunk. This is done by introducing something called as *continuation inode* or *cnode*. A continuation inode is a glue between two files or directories in two different

chunks. It allows seamless use of the entire file system, but on the flip side it could cost expensive disk seeks increasing the I/O latency. Thus it is a good metric to know how the number of continuation inodes increase as the file system ages, or how ChunkFS scales in general. These are some of the questions that are attempted to be answered with this study.

1.5 Contributions

This work emphasizes recovery driven file system design in order to cope with the increasing file system checking time. This foundations of this approach was laid in previous publications [14] and [15]. However the technique was not actually implemented and evaluated so far. This work is extension to the work described in [14]. In this work, we explore the discussed technique further and implement it to evaluate the design decisions.

Main contributions of this work can be stated as under:

- to come up with a novel way of using recovery-driven file system design technique
- understand the current implementation of file systems and to find methods to deploy the technique
- understand the pitfalls and trade-offs following this path of file system design

Chapter 2

File Systems & Recovery

"If a tree falls in the forest and no one hears it, does it make a sound?"

– George Berkeley (1685 - 1753)

This chapter takes a technical overview of traditional file system. It elaborates what a file system actually is and takes a closer look at one of the file system, *ext*2, on which this work has been implemented. This chapter also discusses operations of file system checker program and its primary functions.

Since this work is designed keeping UNIX-based file systems in mind, we will borrow concepts and terminology for file systems from UNIX. But that said, the design discussed in the next chapter is very generic and is equally applicable to non-UNIX-based file systems. Most other non-UNIX systems also have rather equivalent but slightly modified terminology. The terms used throughout the dissertation are listed and defined in Appendix A.

2.1 What makes a file system?

File system is a methodology with which data is stored on the disk. The main aspect of a file system involves technique of laying out data on the disk. This on-disk layout is crucial, since it directly affects the file system efficiency and throughput. In this section we will go through some important file system data structures.

2.1.1 Superblock

After a block containing booting information, file system starts with a data structure, called *superblock*, which serves as an identifier for the file system and describes the file system in whole. It necessarily contains following key fields [16]:

- `file system magic` - identifier for the file system
- `file system size` - size of the file system in blocks
- `number of unused blocks`
- `number of total inodes`
- `number of unused inodes`
- `block size`
- `state` - maintains current state of the file system
- `pointers to inode and block on-disk data structures`
- `last-checked time` - last time, the file system was checked

Superblock is typically replicated across several locations within the file system as a backup, to be used incase the primary superblock fails. The *state* field of the superblock maintains information if the file system is currently *mounted* (in use), dirty or clean. This state is checked at the boot time to determine if the file system needs checking. Also, last-checked time stored in the superblock is used to enforce a periodical check (e.g. after 50 mounts or 120 days, which ever is earlier) even if the file system state is clean. This helps taking care of errors occurring without the notice of the file system.

2.1.2 Inode

As briefly mentioned in section 1.1, inode [1] is a primary data structure that holds metadata for a file. Following are some important fields in inode [17]:

- `file size` - size of the file
- `file type` - type of the file, e.g. *socket*, *pipe*, *block special* [2]
- `file permissions` - read, write, execute permissions
- `pointers to data block(s)` - actual data of the file
- `creation, modification, access times` - book-keeping information
- `link count` - number of other files that this inode is linked to

Apart from this, file system assigns a number to every inode, which is typically the index into the data structure used to maintain inodes, and uniquely identifies the inode. This number is, however, not stored within the inode data structure.

2.1.3 Links

The first file of every file system is a directory, called as *root directory* or simply root, and symbolized as "/". Directory is a special type of file [2]. It is exactly like regular files, but just that it stores data in a specific format and provide a hierarchical structure for the file system. Directories store entries of the files which include file name, inode number, and length of the entry. There are two special entries in every directory:

- "." (single dot) which refers to itself

[1] It is not clear where the term *inode* comes from, but at the 2003 conference of the International Association of Computer Investigative Specialists (IACIS), it was jokingly suggested that *inode* actually stood for "I'm Not Operating DOS Ever". For the matter of fact, term inode was coined much before DOS was even written.

[2] See Appendix A for definitions

- "**..**" (two dots) which refer to the parent directory, the directory which contains entry for this directory. For root, it is same as ".".

The number of entries that an inode has, determines its link count. If inode's link count is 1, it means that it is listed under 1 directory, i.e. it has 1 directory entry. Since, every file has a parent (root's parent is root itself), every file starts out with its link count set to 1. But for directories, since they refer themselves (with the "." entry stated above), they start out with the link count of 2.

Along with the basic data structure which are used to describe the file system and the file, on-disk file system layout contains data structures which are used to maintain used as well as unused blocks and inodes. This is where file system authors become innovative, and the reason why file systems differ. Typical data structures used for this are *bitmaps* [18, 19], *balanced trees* [20] and *B+ trees* [21, 22].

In the next section, we will study ext2 file system in detail and in further sections we will examine how file system checker is influenced by the file system design.

2.2 Second Extended File System (Ext2)

Extended File System is a file system designed as an extension to file system in Minix [23] operating system. Ext2 [18] is its descendant which improves on performance. Ext2 is native to Linux [24] operating system and is being used since early days of Linux.

This section explains ext2 file system and creates a background for understanding how file system checker works, which is explained in the next section.

2.2.1 Ext2 Disk Data Structures

Influenced by BSD file system [25], ext2 divides the disk into small groups called *block groups*, as shown in figure 2.1. Each block group contains copy of superblock and information about the inodes and blocks within that block group. Since ext2 uses

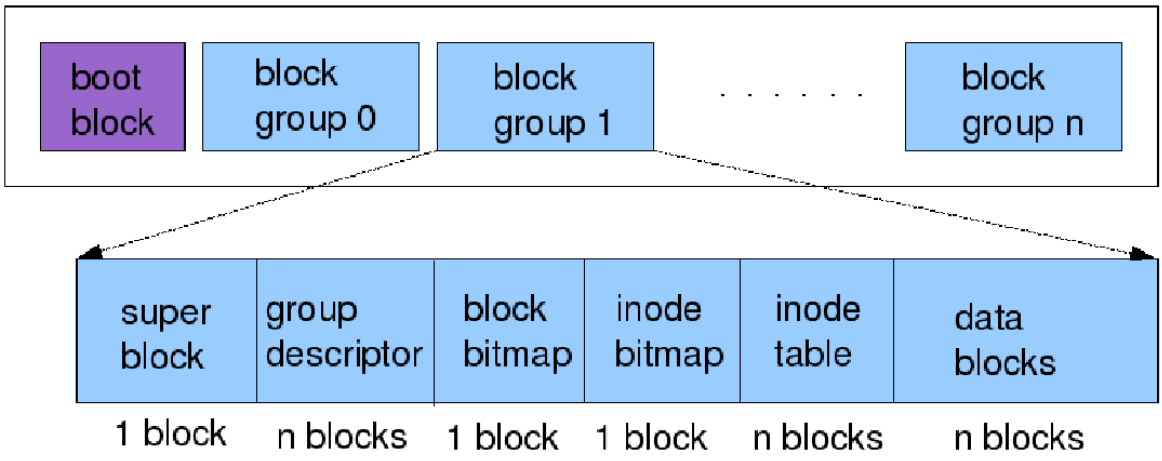

Figure 2.1: *Ext2 Disk Layout. It consists of block groups, which contain a copy of superblock, a descriptor for the block group, inode and block bitmaps and inode table along with the actual data blocks. Size of a block group can be determined while making the file system or by default is set to 8 * block size.*

bitmaps [3] to maintain information for blocks as well as inodes, every block group contains a set of bitmaps for inodes and blocks within it.

Size of a block and size of a block group can be specified when making a file system. Ext2 supports block sizes from 1024 to 4096 bytes. Default value for a block size is 1024 bytes, where as for block group it is 8 times size of the block [26].

2.2.2 Ext2 Operations

To understand, how disk data structures work and interoperate, let us take an example of creating a file. Creation of a file in ext2 file system involves following steps, we will exclude the error handling and details which are of no interest to us.

Create a file named foo in directory bar, assuming foo is not present already in bar.

1. Find a free inode for foo, starting from bar's block group [4]

[3] Bitmap uses a fixed length data type in which individual bits serve as an indicator, e.g. if it is used for managing free blocks, n^{th} bit is set to 1 if n^{th} block is used and 0 otherwise

[4] This is done to exploit the spatial locality and reduce the disk seek latency during an I/O

- find an unset bit, i in the inode bitmap

2. Mark the inode in use

   - set the bit i found in previous step

   - grab the i^{th} inode from the inode table of the group

3. Decrement unused inode count in the super block

4. Create a link with name "foo" in bar

   - allocate a data block for bar if needed

The ordering of these steps influences how file system checker should work and is crucial for file system reliability. We will revisit these steps in the next section.

Similarly for expanding a file, or appending to a file, following steps are involved:

Append k blocks of data to file foo.

1. Find a free block starting from the file's block group

   - find an unset bit, b in the block bitmap

2. Mark the block in use

   - set the bit b found in previous steps

3. Add the block number to the list of inode's data blocks

4. Repeat k times

Although the steps look logical and good enough, when unexpected crash is considered, the process becomes involved.

operation [27].

2.3 File System Errors and Fsck

In this section we will look at how file system errors occur and how those can be tackled with.

2.3.1 How Errors Happen?

The only atomicity primitive that hardware provides is of setting or resetting a bit. Due to this fact, ordering of the steps for a file system operation is important. For example, in file creation steps in subsection 2.2.2, if file system crashes for some reason after decrementing the unused inode count but before creating a link in the directory, i.e. before step 4, but after step 3 above, the newly created inode would remain marked as used, but it is not linked to any directory. If we were to determine where that inodes should be linked, we need to check for a directory within the file system that is likely to have such a link. And for doing this, every directory needs to be checked one at a time. Such unlinked used inodes are referred to as *orphan* inodes and need to be dealt with separately when file system is checked for.

Now if we swap steps 2 and 4 above, and if file system crashes after creating a link for the inode but before marking it as used, we will be left with a link to an inode which is not marked as used. Now it is relatively lesser work to bring the file system back in consistent state.

Similarly, while adding a block to a file, the process of marking it as used and adding it to the list of blocks is not atomic and the ordering influences how file system checker should operate. File system does not go in inconsistent state if it crashes while updating data, but it does if it crashes while updating metadata.

As briefly discussed in section 1.1, file system could corrupt due to multiple reasons. These reasons include:

- Operating system crash

- Crash due to power failure
- Operating system or file system bug
- Hardware error

There are techniques which effectively handle crashes and avoid file system checking, but for the other types of corruption, file system checking becomes unavoidable.

2.3.2 Fsck

Fsck [28] is a UNIX file system checking program designed in 80s originally for 4.2BSD and 4.3BSD file systems. Fsck is is an interactive file system check and repair program, which uses redundant structural information in the file system to perfrom several consistency checks. If any inconsistency is detected, the operator is prompted for further action on the inconsistency, who optionally may choose to fix or ignore it.

Fsck works in multiple passes, checking certain aspect of the file system and / or gathering more information required for latter passes. 5 passes were originally suggested for BSD file system [29] and Ext2 follows the same rules [18] along with optimizations suggested in [30].

Pass 1

Pass 1 of fsck goes through all the inodes in the file systems and checks for static consistency. That is, it checks inode as an unconnected objects in the file system. Fields such as mode and file type are checked. It also collects directories in a separate list for use in latter passes.

Fsck throughout its operation creates its own view of the file system and tries to match it up with the view actually represented by the file system. It creates its own set of bitmaps for blocks and inodes in each block group and in this pass, it marks block used whenever it is listed as inode's data block and the bit in inode bitmap as used for every used inode. These bitmaps created are then matched up with the

on-disk bitmaps and latter are rectified if any discrepancies are observed. During this operation if it encounters multiply-claimed blocks, with the consent of the user, it resolves the issue by either duplicating the blocks or by deallocating either of the inodes.

Pass 2

In pass 2, all the directories are static checked. The list created in pass 1 is traversed and for each directory, all its directory entries are checked. For every directory entry, it is checked if the inode number is within range, the file name is acceptable and if the directory contains sane entries for "." and "..". Since it iterates over every directory entry in the file system, this is the place to collect information about link counts of the individual inodes, which is maintained in a separate list. This list contains links counted for every inode.

Pass 1 and 2 and mostly IO bound and gather most of the information required for further passes. Collectively passes 3 through 5 take only 5-10 % of the total running time of fsck.

Pass 3

In pass 3, it is verified whether every directory is connected to some directory leading to the root directory "/". The directories not tracing back to root are placed in a special directory called $lost+found$, in which all the orphan inodes, discussed earlier, are also placed.

Pass 4

In pass 4, link counts for all the inodes in the file systems are verified. For every inode, the link count from the list created in pass 2 is matched up with the actual link count on the inode. Any inconsistencies are reported and prompted for action. Any used inodes with zero link counts (orphan inodes) are moved to lost+found directory.

Pass 5

In pass 5, fsck checks if the bitmaps it created for blocks and inodes match with the actual bitmaps on the file system's block group descriptor. Thus in this pass, the bitmaps are compared and on-disk copies are corrected if necessary.

2.4 Avoiding Fsck - Previous Work

With the background of how file systems can fail and how fsck attempts to rectify them, we will look at previous attempts to avoid file system checking.

2.4.1 Journaling

Journaling file systems add journal, or log, to an unmodified file system. This journal records changes to be made to the file system before changes are committed to the disk, in such a way that if file system crashes in the middle of doing some operation, after reboot that partially completed operation can be either carried out till completion or completely undone. The updates made to the journal are usually done with the help of synchronized writes, i.e. the journal is flushed as soon as something is written to them. Journals are usually in different disk or disk partition that the actual file system. This helps isolate faults.

Changes can be to the data or to the metadata. For example, if a file is being written, data is written but if the file size does not change, then it is data change. But if file size changes, then it is both data as well as metadata change. Based on this, journaling can be mainly of three types:

Data Journaling In this all the metadata changes as well as data changes are recorded. This is rather expensive logging and affects performance drastically, since every write needs to be done twice - once in the log and once on the disk.

Journaling With Writebacks In this only changes to metadata are logged and

data changes are done independently. This is relatively less expensive, but risk of data loss is not mitigated. Also out-of-order write scenarios could occur if metadata is updated first, and recorded in the journal, but actual data changes do not happen. So, for example, a file being appended could have garbage at its tail end after a crash recovery.

Ordered Journaling This is same as journaling with writebacks, but the sequence in which writebacks happen and metadata changes are logged is reversed. In this, data is written first before metadata is marked as committed in the journal.

In journaling file system, on reboot if file system is determined to be inconsistent state or uncleanly unmounted, the journal is consulted for any half updates to the file system. The changes noted in the journal are then either replayed or the partial changes to the disk are undone.

Although, journaling helps avoid fsck, it helps only during file system crashes during half updates. If the file system is corrupt due to file system bug or hardware bug, then journaling does not come to rescue and full file system check has to be performed if any anamoly is detected which is not recorded in the journal.

2.4.2 Soft Updates

Usually, both data and metadata after being updated in memory do not immediately synchronize with the disk. They are written back during the next *writeback* cycle later. Synchronous writes, as deployed in older file systems like DOS [31], could arguably be most robust solution to metadata updates. In synchronous writes, all the updates are written back to disk instantly. So the memory and the disk copy of metadata are almost coherent. But what comes with it is unacceptably poor performance, since the file system operates mainly with the disk speed as opposed to memory speed [32].

Developed originally for BSD file systems, *soft update* [33] is a mechanism that allows complete asynchronous operation without any need for a journal. Soft updates uses delayed writes to update metadata. This is achieved by tracking dependencies among buffers in the memory and enforcing proper write order for both data and metadata, such that the disk always contains a consistent copy.

Since data goes to the disk before metadata, this approach, may leave unclaimed used blocks and orphan inodes due to crash, but still file system is in consistent state and those can be cleaned away without any warning. For such 'garbage collection', a background fsck is run in BSD systems [10]. Fsck running in the background works with a file system snapshot and carries out the garbage collection. Thus, fsck can run in the background even if the file system is mounted and in use. However, for any anamoly detected in the file system and if the file system is to be repaired, whole file system has to be checked offline.

2.4.3 NVRAM

NVRAM, or non-volatile random access memory, also commonly known as flash memory is similar to traditional RAM, with only difference that the data on NVRAM is persistent even after power loss. This is same as having an uninterruptible power supply for the entire system. With NVRAM, number of different design techniques could be applied.

Journal on NVRAM One approach would be to store journal or logs on NVRAM [34]. During crash recovery, the uncommitted updates on NVRAM would allow bringing the file system back to consistent state.

Metadata on NVRAM NVRAM, on the other hand could be used to store the metadata and can be used in combination with file systems like [35].

NVRAM as disk cache NVRAM could well be used as disk cache [36], in which

case only NVRAM needs to be consistent and the writes to the disk could happen anytime.

In all of the above approaches, performance of file system with NVRAM far exceeds that of other file systems. But the drawback to this is mainly the cost of NVRAM. Also, failing of NVRAM may lead to a non-recoverable file system if data on NVRAM is crucial [34], e.g. all of the metadata. Other drawback include the need to have both the NVRAM and other disk during the crash recovery, which means it is not possible to move one component from a crashed system to other.

Moreover, overall fsck time still remains proportional to the disk size and amount of used metadata on the disk.

2.4.4 Other attempts

All the techniques discussed in this section, try to avoid running a file system checker program. But as mentioned earlier, it is the hardware malfunction and operating system and file system bugs that make fsck mandatory.

Although some efforts did went into actually reducing the file system time [30, 37], none of them could contribute a significant reduction compared with the growing disk capacity trends. More related work is described in chapter 6

Chapter 3

A Recovery Driven Design

"If anything can go wrong, it will, and usually at the most inopportune moment"

– Murphy's Law

In this section we describe our recovery-driven approach towards file system design. Based on the design, the file system we developed is called $ChunkFS$. The design is also described briefly in previous publications [14, 15].

3.1 Divide and Conquer

The main idea of ChunkFS is to divide the file system into smaller chunks, which are an independent file systems by themselves. Thus, every chunk in the file system has its own superblock, and an independent inode number as well as block number namespace.

The main intention of dividing the entire file system into smaller chunks is to isolate fault boundaries and allow file system checker to skip chunks of file system and only check those that were active at the time of the crash. Even if anomalies are detected in some part of the file system, only the concerned chunks would be checked and recovered, as opposed to entire file system.

As shown in figure 3.1, each chunk contains a complete file system and ChunkFS acts as a wrapper across the multiple chunks and export a holistic view to the user

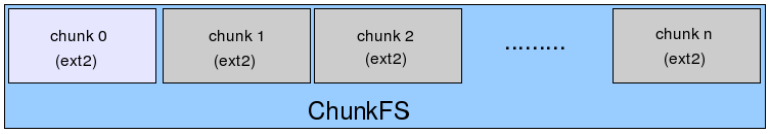

Figure 3.1: *A file system is divided in smaller chunks, which are a file systems in themselves.*

as if its one big file system. The individual file system within each chunk operates independently. The main goal of fault isolation is achieved by allowing file systems within each chunk to independently allocate inodes and blocks within its own chunk. The rules can be briefly and informally stated as under:

- **RULE I**: Every directory entry should only reference inode from the same chunk as that of the directory.
- **RULE II**: Every block of every inode should be from the same chunk as that of inode.

To formalize the constraints, let the chunks be denoted as C_i, where $0 \leq i \leq NR\_CHUNKS$, where NR_CHUNKS is the number of chunks. Let directories in chunk C_i be denoted as D_{ij}, where $0 < j \leq INODES_i$, $INODES_i$ being total number of inodes in C_i. Recollect that every chunk, or for that matter every UNIX based file system, contains at least one directory - root. Let a directory D_{ij} have directory entries DI_{ijk}, where k denotes number of directory entries which is bounded by the amount of disk space directory can have [1] and no less than 2 [2]. Let $Inode(D_{ijk})$ denote inode of directory entry D_{ijk}. Also let inodes for chunk C_i be denoted as I_{ij}, where $0 < i \leq NR\_CHUNKS$ and $0 \leq j \leq INODES_i$, and having blocks B_{ijk}, where j is bounded by number of blocks in the chunk and no less than 0. Finally, let $CHUNK(I_{ij})$ denote chunk of inode I_{ij}, i.e. C_i

Thus, above constraints can now be restated as:

[1] This is because a single file theoretically can have infinite hard links to a directory. Refer Appendix A for definition of a hard link.

[2] for "." and ".."

- **RULE I**: D_{ijk} such that $\forall$ i, j, k, $CHUNK(Inode(D_{ijk})) = C_i$.
- **RULE II**: B_{ijk} such that $\forall$ i, j, k, $CHUNK(I_{ij}) = C_i$

With these constraints, the cross chunk references are minimized and file system checker does not have to check chunks that are not dirty. Because they refer to only local data structures, and it is marked consistent. We will discuss this in greater detail in further sections.

3.2 Continuation Inodes

When file system is divided into smaller chunks, the file may want to grow beyond its chunk size, or file may want to grow but there is no space in its own chunk, but there is in some other chunk. Such files spanning multiple chunks has to be managed separately. Also directories should be allowed to have files and subdirectories across multiple chunks, since the chunk in which they are placed may run out of unused inodes. To allow such behavior *continuation inodes*, also referred to as *cnodes*, are introduced [14]. Before continuing further with the design, lets introduce some terminologies for convenience.

Continuation inode glues two files or directories allowing a seamless behavior. Whenever a file or directory is to be expanded, a continuation inode is created in a chunk having unused blocks or unused inodes. Thus, there can be multiple continuation inodes for a file as the file grows, forming a chain-like structure of continuation inodes CI_i, where $0 \leq i \leq (NR\_CHUNKS$ - 1). Here CI_0 is called *primary inode* of the file, and CI_i, $i > 0$, are continuation inodes. It is important that for a single file, $i \leq (NR\_CHUNKS$ - 1), i.e. total number of continuation inodes for any file should be less than total number of chunks or in other words there should be at most one continuation inode per file per chunk, excluding the chunk of the primary inode. We will determine the reason for this constraint in further sections. $PARENT(CI_{i+1})$

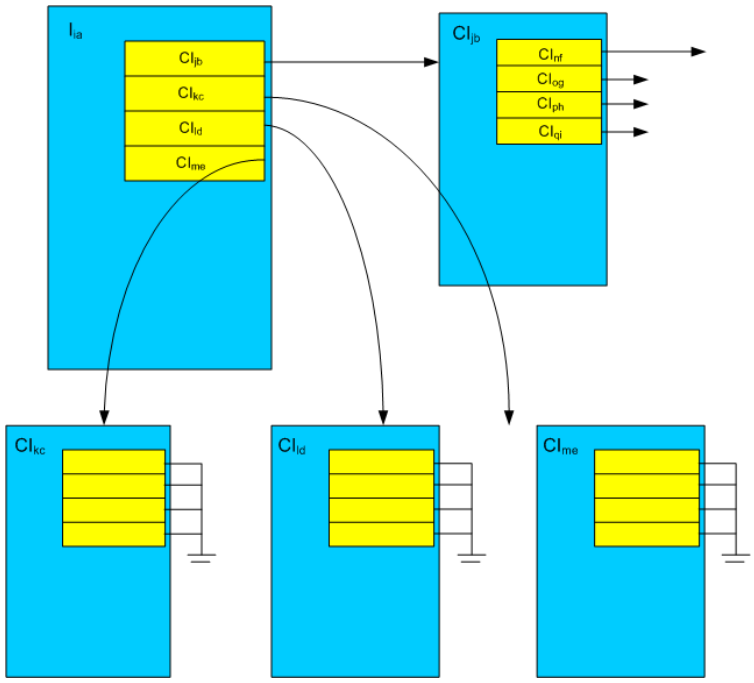

Figure 3.2: *A one-sided tree with fixed number of data slots per node is used to store continuation inodes. There is one tree per file.*

denotes continuation inode prior to CI_{i+1}, i.e. CI_i; and CI_{i+1} is called *parent* of CI_i. For example, in figure 3.3, $PARENT(CI_{jy})$ is I_{ix}, and I_{ix} is the primary inode for the file. For every continuation inode, there exist a $forward\ pointer$ from parent to the continuation inode and a *back pointer* in the reverse direction. In this way, continuation inodes know who their parent inode is.

Continuation inodes are very similar to inodes, except that they are marked as being a continuation inode with the help of a flag $INODE\_IS\_CNODE$ and contains a back reference to its parent inode. A parent inode can either be an inode or cnode, but for convenience sake, it shall referred as an inode.

3.3 Managing Continuation Inodes

For managing continuation inodes, data structure shown in figure 3.2 is used.

The data structure is one sided tree with multiple data slots within each node

of the tree. Every on-disk inode incorporates one node. The data slots in a node are limited to a finite number, 4 in the example shown in the figure. Each data slot represents a continuation inode and only first continuation inode in the node is allowed to be parent of a continuation inode.

This data structure is chosen due to following reasons:

Typical File Sizes Most file sizes on UNIX are small [38]. Because of this, typically, not many files are expected to span across chunks and typically would have small number of continuation inodes. Thus though time complexity for traversing the data structure is $\boldsymbol{O}(n)$, it can be very fast when successive traversals are done within memory by pinning down continuation inodes in memory instead of reading from the disk for every traversal. The pinned inodes can be freed when the primary inode is freed. Also size of a chunk is a controlling factor as to how many continuation inodes exists in the file system. This is dealt with in greater detail in section 3.5.

Non-recursive Implementation This data structure allows non-recursive traversal of the continuation inode chain. Also no additional data structures, like *queue* or *stack* is needed for traversal. This is helpful in OS environment.

Note that all the continuation inodes in one node have same parent, and are called *siblings*. Thus all the continuation inodes, CI_{jb}, CI_{kc}, CI_{ld} and CI_{me} are siblings having I_{ia} as their parent.

3.3.1 Expanding Files

File maintains pointers to its data blocks. Consider a scenario in which a file with inode I_{ix} needs to be expanded. Now when chunk C_i, in which the inode for the file is, runs out of space, and there exists another chunk C_j, which has unused blocks, the file needs to make use of unused data blocks in chunk C_j. This has to be done in

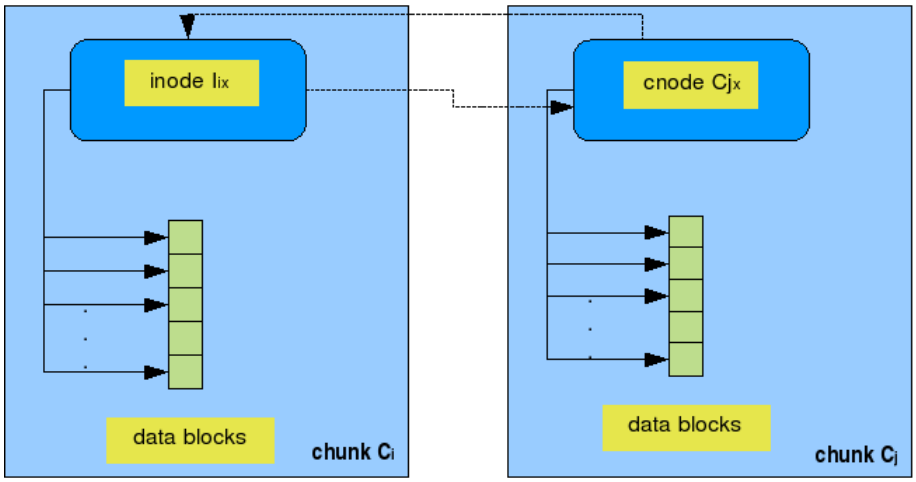

Figure 3.3: *A cnode is created in chunk C_j for file inode in chunk C_i.*

such a way that there is minimal dependency between chunk C_i and chunk C_j. We follow the approach depicted in figure 3.3.

For expanding a file, a continuation inode, CI_{jy}, is created in chunk C_j for the file to be expanded and forward and back pointers are placed accordingly. The continuation inode now can point directly to data blocks in its chunk, which are now accounted as inode I_{ix}'s data blocks.

3.3.2 Expanding Directories

As shown in figure 3.4, consider a scenario in which a new file, $newfoo$ is to be created in a directory. Let the inode of the directory be I_{ix}. Recall that for creating a new file, an unused inode is to be allocated. If C_i, I_{ix}'s chunk, does not have any free inodes, a continuation inode, CI_{jy}, is created for it in another chunk, C_j. After creating the continuation inode CI_{jy}, which is now extension of I_{ix}, is a normal directory. After creation of CI_{jy}, $newfoo$ is created and the directory entry is placed in CI_{jy}.

The entries in all the continuation inodes of I_{ix} appear in its directory listing [3].

[3]For example, *readdir* system call.

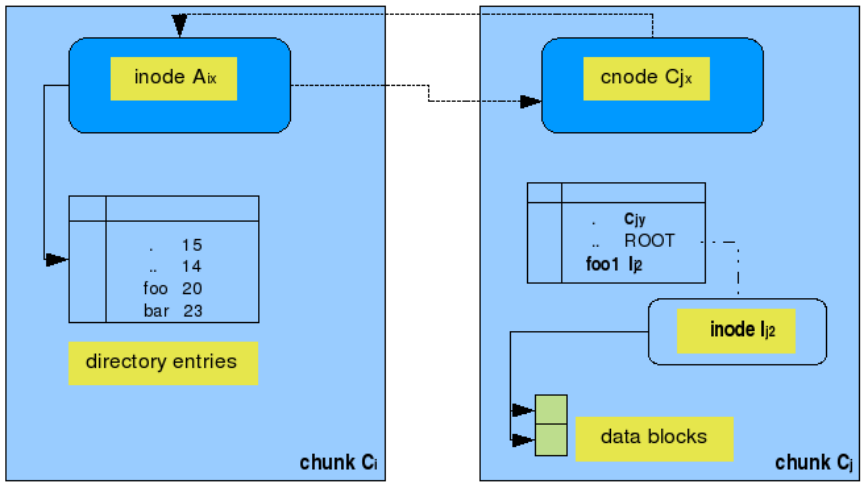

Figure 3.4: *A cnode is created in chunk C_j for directory inode in chunk C_i.*

3.3.3 Cross-chunk Hard Links

For creating a hard link for an inode, I_{ix}, in a directory, D_{jy}, a directory entry needs to be created in D_{jy}. In this particular case where the inode and directory are in different chunks, directory entry cannot be directly created as it would violate the **RULE II** stated in section 3.1. We deploy same technique as we do for expanding directories as described in subsection 3.3.2. If the inode and the directory are in different chunks, as is the case here, a continuation inode for directory is created, if it does not exist already, in inode's chunk. Thus in this case, a continuation inode would be created for D_{jy} in chunk C_i and then a directory entry for I_{ix} would be created.

It should be noted that for creating such hard links, directories are expanded into the inode's chunk which is to be hard linked. For doing this we allocate an inode and at least one data block for storing directory entries. Thus, if any of this is not available, we will not be able to expand the directory into another chunk, and thus hard links cannot be created even if there exist unused inodes and data blocks in the file system as a whole. This would clearly be a design drawback. This problem is

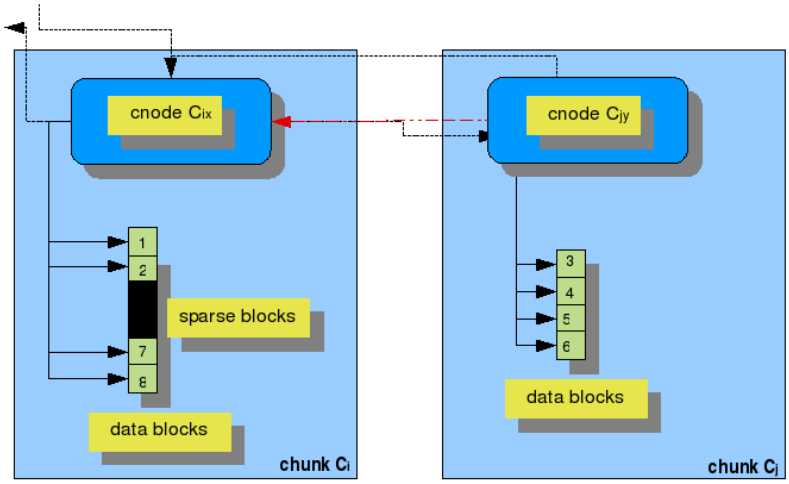

Figure 3.5: *A cnode is created in chunk C_j for inode in chunk C_i. Later the file is expanded again in chunk C_i. For this, a new continuation inode is not created, but a sparse section is added to the old inode.*

solved by reserving a small number of inodes and data blocks in every chunk just to handle the cross-chunk hard links. These reserved inodes and blocks are used only for creating hard-links if the chunk in concern is nearly full.

Although, we reserve a small set of inodes and data blocks for hard links in every chunk, it should be noted that most files in UNIX-based file systems have maximum of 1 hard link and thus such scenario is inherently rare.

3.3.4 Limiting Continuation Inodes

Continuation inodes are created whenever a file or directory needs to be expanded. A constraint is enforced which dictate upper limit to the number of continuation inodes for a file. The constraint is that there can be at most one continuation inode per file per chunk, excluding the chunk in which the file originally started out. This constraint prohibits waste of continuation inodes and unnecessary I/O operations for reading and writing continuation inodes.

The way to achieve this is to enable continuation inodes to have multiple *sparse* fragments. *Sparse* files are regular files which do not have data in between some

locations [4]. For illustration refer figure 3.5. The block numbers shown represent logical block numbers for the file, i.e. what location in the file a particular block holds data for.

The figure 3.5 is output of following sequence.

- expand file in C_i for 2 blocks.

- later, expand file more. Now if chunk C_i does not have free blocks, a continuation inode, C_{jy} is created in chunk C_j and blocks 7 through 10 and placed.

- finally, the file is expanded again and this time we are assigned chunk C_i again. Now, instead of creating a new continuation inode, a sparse fragment is created in the existing continuation inode C_i for blocks 11 and 12.

Limiting continuation inodes per file per chunk avoids pathological worst case of one single file in the file system with all the inodes in its own chunk as well as all other chunks as continuation inodes.

Continuation inodes are an important performance metric for ChunkFS. More the continuation inodes, more fragmented is the file system. Continuation inodes for a given file are inherently placed at a distance from each other, and most likely disk head seeks while reading them sequentially. Thus I/O latency is increased with the increase in the number of continuation inodes. Though large files are rare in UNIX-based file systems [38], it is important to limit the number of continuation inodes for better performance. This is achieved by selecting a suitable chunk allocation algorithm [27].

3.4 Fsck for ChunkFS

In this section, we examine what are the implications for file system checker (fsck) for ChunkFS of our recovery-driven file system design approach described in previous sections of this chapter.

[4] Refer Appendix A for more description.

Firstly, on detection of unclean unmount or detection of any file system anomaly, fsck would check only those chunks those are marked dirty. The whole idea of dividing file systems in smaller chunks, is to divide the amount of metadata on the disk scanned by a large constant factor, ideally by the number of chunks in the file system. Performance close to ideal can be achieved with careful selection of data-structures and some amount of redundancy, as explored further in the section.

3.4.1 Fsck Passes

Although, most part of fsck functionality remains same, changes in file system behavior and on-disk format requires changes in fsck. For ChunkFS, fsck performs passes 1 through 5 for all the chunks that are marked dirty, and performs an extra pass "Cross-chunk Pass" after all the chunks are through. Fsck maintins two lists for handling continuation inodes, IS_CNODE and $CHECK\_IF\_CNODE$. IS_CNODE list contains all the continuation inodes that are encountered and $CHECK\_IF\_CNODE$ list contains all the continuation inodes that any inode refers to.

Passes 1 through 5 of fsck described below have running time complexity of $\boldsymbol{O}$*(number of chunks dirty + amount of used metadata data on them)*. Though, theoretically $\boldsymbol{O}$*(number of chunks dirty + amount of used metadata data on them)* $\in$ $\boldsymbol{O}$*(amount of used data on the file system)*, practically it is much less as it depends on the number of chunks active during the crash or corrupted due to any other reason.

However, there are certain cases in "Cross-chunk Pass", that can have running time complexity of $\boldsymbol{O}$*(amount of used metadata on the file system)*, but this can be avoided with the help of redundancy. This complexity can happen due to the need of iterating through all the chunks in search of a continuation inode or continuation inode's parent. The time complexity can be reduced by deploying one or more of following methods:

Per chunk continuation inode bitmap This is same as inode bitmap, but the

only difference is that, a set bit would indicate a continuation inode. Essentially, this bitmap would be used for distinguishing a continuation inode from regular inodes, since only difference between them is $INODE\_IS\_CNODE$ flag. With this it would be possible to locate continuation inodes immediately instead of going through entire chunk looking for continuation inode.

Per chunk cross-chunk bitmap A set bit in this bitmap would indicate presence of a cross-reference between this chunk and the chunk indicated by the bit number. This would help reduce the number of chunks are checked while resolving a problem concerning a lost continuation inode or lost parent.

It should be noted that the on disk data structures above could as well be corrupted and wrong. But techniques like journaling, discussed in subsection 2.4.1, could be used for individual chunks to ensure reliability of these data structures. This is something that needs to be explored further and is within our future scope.

Different error scenarios and possible solutions are discussed in subsection 3.4.2.

Having background of fsck program described in subsection 2.3.2, changes in respective passes of fsck can be listed as under. However, it should be noted that these are only changes required for fsck for ChunkFS, all other fsck operations not referred to remain same as described in subsection 2.3.2 and in [29].

Pass 1

As mentioned in subsection 2.3.2, pass 1 does static check on all the inodes in the chunk. Apart from the things done for $non-chunked$ [5] file system in pass 1, fsck adds to two lists mentioned above. Going back to the figure 3.3, after fsck checks chunk C_i, the lists would look like:

$$IS\_CNODE = \{ \emptyset \} , CHECK\_IF\_CNODE = \{ I_{ix} \}$$

[5] File system which is not divided into chunks e.g., Ext2.

And after checking chunk C_j, the list would look like:

$$IS\_CNODE = \{\ C_{jy}\ \}\ ,\ CHECK\_IF\_CNODE = \{\ I_{ix}\ \}$$

This lists are checked in "Cross-chunk Pass".

Apart from adding to IS_CNODE list, pass 1 treats continuation inodes same as inodes and checks them like any other inode.

Pass 2

This pass goes through all the directories in the file systems and checks for their consistency. Since all the directory entries are for local inodes, no changes are required for this pass for ChunkFS.

Pass 3

In pass 3, directories are checked for connectivity by tracing them back to root. In case of continuation inode for directories, the parent is root of that chunk, so these are verified in same way as other directories. Any disconnected directories, however, are placed in $lost + found$ directory of first chunk.

Pass 4

In pass 4, link counts for inodes are verified. Continuation inodes do not have directory entries, so the links counted for continuation inode would be 0. But still since in-use inodes start with link count of 1, their link count is 1. Thus an exception is to be made for continuation inodes for this pass. In this pass, orphaned inodes are placed in $lost + found$ directory of first chunk, instead of local chunk's.

Pass 5

No design changes are required for pass 5. However, depending on implementation, bitmaps of blocks could be required to start at an offset, instead of 0.

3.4.2 Cross-chunk Pass

This pass happens when all the chunks are through passes 1 to 5. This pass is aimed at verifying that the continuation inode forward and backlinks are consistent.

In this pass, the lists created in the first pass are used to verify the cross-chunk connectivity. In this pass, the attempt is to validate if all the continuation inodes in $CHECK\_IF\_CNODE$ exist. List IS_CNODE contains continuation inodes that have been encountered so far. But, recollect that during fsck not all chunks would be checked. Only those chunks that are marked dirty would be checked. Thus further work required is to sanitize $CHECK\_IF\_CNODE \setminus IS\_CNODE$ continuation inodes.

In this pass, every continuation inode, CI_{ij}, in $(CHECK\_IF\_CNODE \setminus IS\_CNODE)$ is read and made sure that it is marked as being a continuation inode. Further sanity checks on CI_{ij} or its chunk are optional, since C_i would have been already checked if it was marked dirty, and if it was not, then there is no need to check it. Similarly, parent for every continuation inode in IS_CNODE is verified if it is correct.

To look at how different error conditions are resolved, let us take error cases one by one.

Forward pointer is corrupted

Forward pointer to a continuation inode can be treated as corrupted if continuation inode that it points to does not exist or is invalid. In this case continuation inode needs to be found. Following measures could be taken to rectify the problem:

- get the likely characteristics of the continuation inode that is lost, like starting logical block number, ending logical block number, if it can contain sparse fragments, etc.

- first iterate through all the dirty chunks for continuation inodes and see if a match is found.

- if no match is found iterate through all the chunks, other than where the file already has a continuation inode, and look for a continuation inode with matching characteristics.

- if match is found, correct the forward link, else report error.

Backward pointer is corrupted

Similar to the case above, backward pointer can be treated as corrupted, if the inode that a continuation inode points to as its parent either does not exist or is invalid. In this case, the parent inode can be found out by making use of one of the methods described in subsection 3.4.1.

3.5 ChunkFS Trade-offs

Chunk Size

Most important trade-off for ChunkFS is individual chunk size. If chunks are small, it would take lesser time for fsck to recover the disk. But, on the other hand, now more files would require continuation inodes. If chunks are too big, lot less continuation inodes would be needed, but now fsck would take more time to repair the file system. Thus it is important to have an appropriate chunk size largely depending on the file size pattern. Based on experimental data and contemporary file size trends [38], it is recommended to have chunk size approximately 1% of the file system size.

Cnode Slots

Another minor trade-off can be observed is for the data structure used to maintain continuation inodes described in section 3.3. More the slots, more easy is the repair and traversal, since less inodes would be required to be read. But would waste disk space for inodes not having continuation inodes. However, on the other hand if disk space is not to be wasted, at most we can use singly-linked list, which would have

least possible I/O performance. However, this trade-off is alleviated with a technique described in section 3.3

Chapter 4

ChunkFS Implementation

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."

– Stan Kelly-Bootle (First CS Graduate 1954)

Developing a file system involves developing following components of the file system:

- Tool for making the file system ($mkfs$)
- Actual driver making file system usable (implemented as a kernel driver in most OSs)
- Tool for recovery and repair of the file system (fsck)

All the tools above, understand the disk layout of the file system. If disk layout of a file system is changed, all the three components needs to be changed. It is worth mentioning that out of the 3 listed components, file system checker is the most non-exciting in terms of development and neglected component of UNIX-based file systems. [11]. This can also be used as an evidence for recovery aspect of a file system not being one of the prime concerns during file system design.

We have two implementations of ChunkFS. One is $fuse - chunkfs$, implemented using a utility called $FUSE$. And later using ext2 Linux kernel driver.

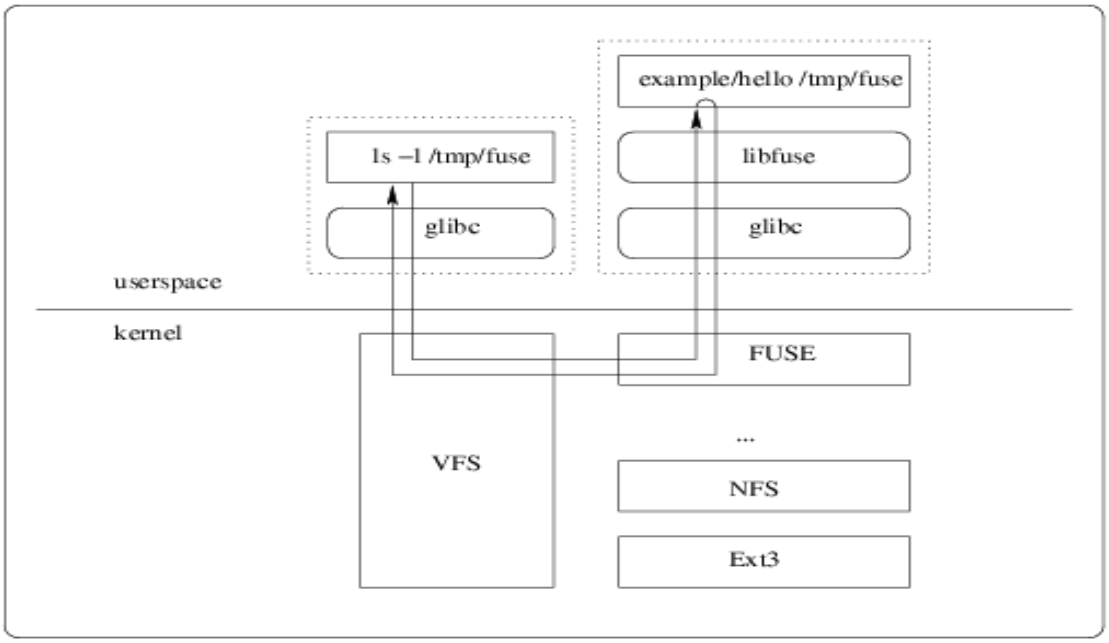

Figure 4.1: *FUSE Structure [1].*

4.1 Fuse-ChunkFS

FUSE, or "Filesystem in Userspace" [1], is a set of kernel supported utilities that allow implementing file systems in userspace. Succinctly, *userspace* is referred to as part of the virtual memory which runs user applications in restrictive environment.

File systems in FUSE are stand-alone programs written in userspace and linked with library *libfuse*. *libfuse* library forms a communication medium between FUSE file systems and the kernel. The file system program when run receives calls from kernel via libfuse. Figure 4.1 shows an example of what path a system call for FUSE file system takes within the entire system.

Figure shows execution path of `stat` system call. Here, as shown in the top-right dotted box, `example/hello` is the binary of the file system written in FUSE, and `/tmp/fuse` is the device that it is working on. 'FUSE' shown in the figure is a kernel module, which interacts with VFS to service the calls to the file system mounted with

FUSE. Thus, it requires minimum 4 context switches for servicing a system call in FUSE. Therefore, file systems written in FUSE do not perform well compared with native file systems implemented as a kernel driver, but are very useful for developing prototypes and for things that need not be in the kernel, but useful as a file system. FUSE homepage [1] lists number of projects putting FUSE to use.

4.1.1 Overview

$fuse{-}chunkfs$ is a simple proof-of-concept implementation on top of $fuse{-}ext2fs$ [39]. fuse-ext2fs is an implementation of ext2 file system in FUSE that we developed prior to starting with fuse-chunkfs. FUSE allows file systems implemented in userspace to be used in Linux. It mainly involves, using FUSE primitives to provide file system specific operations like `open`, `read`, `write`, `stat`, and `delete` for files, very similar to in-kernel file systems with only difference that the whole file system is in userspace and much of the racing and locking issues are taken taken care of by FUSE. The file system only has to worry about its own *reentrancy*, e.g. it cannot be assumed that only one thread would be executing in `open` call for a given file, since multiple processes can open same file. However, this is less of a hassle when compared with concurrency issues in th kernel and reduces the development time drastically.

fuse-chunkfs uses separate disks as chunks. Each chunk has an unmodified ext2 file system. By unmodified we mean that the on-disk layout is same as native ext2 file system. Fuse-chunkfs interacts with the chunks using E2fsprogs [26] library and provide unified namespace and view for the user. During development, *loopback file systems* were used, which allow having a file system on regular files, instead on disk. Not only this reduces development time but also provides flexibility.

Continuation inodes are simulated using regular files. So, in fuse-chunkfs, continuation inodes are actually files on the chunks. This is different than the design described in chapter 3, since here, continuation inodes have directory entries. Con-

tinuation inodes are placed in a doubly linked list for a file. Being in userspace, this is particularly inefficient, since no caching is done for continuation inodes, and hence walking down the list requires reading those many inodes. One design aspect that was learned from this prototype is to avoid frequent read and writes of continuation inodes. Since they are inherently placed at a distant location on the disk from eachother, seeking the disk head to a continuation inode entails increase in I/O latency.

Since most operations in fuse-chunkfs are very straight forward, we will not discuss those in much detail. However, we will discuss chunk allocation policy, since it affects the number of continuation inodes created.

4.1.2 Chunk Allocation Policy

Chunk allocation policy dictates in what chunk should the file or directory be allowed to grow. The file or directory may already have a continuation inode in the assigned chunk, or a new one is created.

fuse-chunkfs implements a chunk allocation policy in which each chunk is partially filled before moving to the next chunk. A global variable called `write_active_chunk` holds the identifier for the chunk in which new writes should happen. When a file or directory is being written, chunk is requested via function call to `write_active_chunk` with preferred chunk and requested resource type as parameters. This function returns the chunk identifier in which new writes should take place. Requested resource type could be either blocks, for expanding file or a directory or inodes, for creating new file or a directory. Another variable, `cfill_limit`, is used which stores normalized value upto which all the chunks are filled, before increasing it further. Thus maximum value `cfill_limit` could take is 1, at which the file system is full. Initial value of `cfill_limit` is set to 0.1 + normalized value of reserved chunk space for hard links as discussed in subsection 3.3.3.

Function `write_active_chunk` checks if the preferred chunk is filled more than `cfill_limit`. If no, it is returned. If yes, all the chunks in which the file or directory to be expanded has a continuation inode are checked and any chunk not filled more than the normalized value `cfill_limit` is returned. If no such chunk is found, all the chunks are scanned and any chunk matching the criteria is returned. In this chunk new writes happen. If no chunk is found not filled upto `cfill_limit`, value of `cfill_limit` is raised by 10% and the search is repeated.

Thus all the chunks are filled 10% of their capacity and then new chunk is assigned. This type of policy is set just for the sake of experimentation.

4.2 ChunkFS over Ext2

Second implementation of ChunkFS is done in ext2 file system driver of Linux kernel 2.6.18 [40]. This is a working prototype conforming to the ChunkFS design discussed in chapter 3. All the 3 components of file system, namely mkfs, fsck and the kernel driver have been implemented simultaneously.

In brief, the implementation largely consist of following.

- Changing on-disk superblock and inode data structures to incorporate chunk and continuation inode awareness.
- Changing the mount procedure to enable individual chunks to be independent file systems.
- Changing `get_block` routine for ext2 to allow both, reading of data from continuation inodes and creation of new continuation inode when the file to be written runs out of chunk space.
- Changing `readdir` routine to allow reading of directory entries from continuation inodes of directories.

- Changing the `create` routine to allow creating new files in a different chunk.
- Changing the `unlink` routine to handle proper deletion of continuation inodes.
- Changing the new inode allocation routine to allow directories spanning across chunks and maintaining inode uniqueness.
- Handling proper caching of continuation inodes for better performance.
- Changing unmount procedures to allow graceful shutdown of file system.
- Modifying fsck to handle continuation inodes.

It should be noted that this implementation is primitive one and there is scope for number of optimizations.

4.2.1 Making ChunkFS

For all UNIX-based block file systems, utility $mkfs$ is written to create a file system on a raw block disk device. This device could well be a regular file which then can be used with a loopback file system. mkfs lays out the disk pattern that is specific to the file system.

In ChunkFS mkfs accepts an additional command-line argument for number of chunks to be created in the given device. mkfs divides the file system by the given parameter thus creating chunks. It then iterates through each of the chunk creating ext2 file system.

For example, `mkfs -C 1000 /dev/sda5`, would create a file system with 1000 chunks in the block device `/dev/sda5`.

Though, this design does not talk about whether chunks should be division of block device, as described above, or should it be physical partitions or separate disks. Allowing both would enable online resizing of the file system. This aspect of ChunkFS, however, needs a closer examination and is left as a future work.

```
@@ -404,10 +435,12 @@ struct ext2_super_block {
        __u32   s_hash_seed[4];         /* HTREE hash seed */
        __u8    s_def_hash_version;     /* Default hash version to use */
        __u8    s_reserved_char_pad;
-       __u16   s_reserved_word_pad;
+       __u8    s_chunk_id;             /* chunk id of this chunk */
+       __u8    s_num_chunks;           /* number of chunks in this file system */
        __le32  s_default_mount_opts;
        __le32  s_first_meta_bg;        /* First metablock block group */
-       __u32   s_reserved[190];        /* Padding to the end of the block */
+       __le32  s_chunk_offset;         /* offset from where this chunk starts */
+       __u32   s_reserved[189];        /* Padding to the end of the block */
 };
```

Figure 4.2: *On-disk Superblock Changes for ChunkFS*

Superblock

New fields are added to the existing superblock of ext2 in order to allow chunks to collaborate with each other and with the *Virtual File System* (VFS) layer of Linux kernel. The Virtual File System is a kernel software layer that handles all system calls related to standard UNIX file system [41]. Its main strength is providing a common interface to several kinds of file systems.

Figure 4.2 shows the changes in superblock as produced by the output of `diff -up`. *diff* is an utility which outputs difference between two files that are given as command-line arguments. The lines added have $'+'$ sign in front of them and those that are deleted have $'-'$ sign. $'\_\_u'$ signifies unsigned data type, and the number following it signifies number of *bits* in the data type.

Thus, as seen in the figure, every chunk's superblock contains an unique chunk identifier assigned sequentially starting from 0 by mkfs, and stored as `s_chunk_id`. This is a 8-bit field, thus allowing maximum of 2^8 chunks. And `s_chunk_offset` contains the offset in number of blocks from which the chunk starts. If for example 200 Gigabyte disk is divided into 10 chunks, offset of first chunk would be 0, for second would be *200 Gb / `block size`*, for third it would be *(200 Gb / `block size`) * 2*, and so on. Thus offset of arbitrary chunk with identifier i, can be given as: *(`size of`*

```
@@ -236,6 +265,8 @@ struct ext2_inode {
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
+       __le32  i_parent;       /* parent inode - encoded */
+       __le32  i_cnodes[NR_CNODES * CNODE_SIZE]; /* list of continuation inodes */
        union {
                struct {
                        __u8    l_i_frag;       /* Fragment number */
```

Figure 4.3: *On-disk Inode Changes for ChunkFS*

*the device / block size) * i*. Field *s_num_chunks* stores total number of chunks. The fields with word 'reserved' are used as padding for fitting the superblock to the block boundary.

Inodes

In order to allow inodes to be continuation inodes and have links to and from continuation inodes, on-disk inode structure is modified. Figure 4.3 shows the modifications.

Every inode is a node in the data structure described in section 3.3. To allow this, as shown in the figure, every inode contains slot for NR_CNODES cnodes, statically set to 4. $CNODE\_SIZE$ contains size of individual slot in multiples of 32 bits, since the data type is 32 bit long. Field `parent` points to the parent of inode.

Apart from this, two flags, `EXT2_HAS_CNODE_FL` and `EXT2_IS_CNODE_FL` are declared which are used to signify whether an inode is a continuation inode, is a continuation inode or none. These flags are marked on `flags` field of inode (not shown in the figure).

Inode numbers in all the chunks start from 0 and grow sequentially by 1. Thus inode number namespace of every chunk completely overlaps with each other. In order to allow unique inode numbers, 8-bit chunk identifiers are used. Most significant 8 bits of an inode represent chunk number in which the inode is and remaining bits represent the actual inode number. This way all the inode numbers are unique. This,

however, reduced the total number of inodes that can be in a file system from 2^{32} to 2^{28} on a 32-bit machine. This limitation can be removed by having 64-bit inode numbers.

Bitmaps

Apart from the new field additions, mkfs changes the block bitmaps of all the block groups to reflect the block offset of the chunk. Important fields of bitmap include `start`, `end` and `map`. Here `map` is the actual bit vector, where as `start` and `end` specify the starting points within the `map`. mkfs only changes `start` and `end` to reflect the offset at which the chunk is. It simply entails adding `chunk_offset` to the fields.

Inode bitmaps are not altered and every chunk has inodes starting with inode 1. Blocks, however, needs to be changed because it is easier to be less intrusive with this change avoiding propagating s_chunk_offset knowledge to many parts of the code.

4.2.2 ChunkFS Kernel Driver

ChunkFS kernel driver puts the file system to use and does most of the work listed in the beginning of this section.

Mounting & Unmounting ChunkFS

VFS maintains its own in-memory representation of superblocks, inodes and directory entries for all the file systems that are currently mounted. These in-memory copies contain more information than their on-disk counterparts, and are required for run-time operations. When a file system is being mounted, as a consequence of a mount system call, request is sent to the file system driver to read and interpret the superblock for the given device and to create in-memory representation of it for VFS.

When such a request comes to ChunkFS, `chunkfs_fill_super` routine reads superblocks of all the chunks in the device and creates in-memory representation for VFS. With the help of on-disk information of `s_chunk_offset` and number of blocks

in the file system, superblock for chunks are located. These superblock structures are initialized and are attached to the global superblock list, `super_blocks`, maintained by the VFS for all the mounted file systems. By adding chunk superblocks to this list, all the chunks are now periodically flushed to the disk by the writeback mechanism. Along with this, in-memory list is maintained of superblocks of all chunks. This is required in order to redirect file system operations on a particular chunk to the respective superblock.

During unmount, routine `chunkfs_put_super` is called. In this routine all the chunks are flushed to disk and superblocks of all the chunks are removed from the `super_blocks` VFS list and deallocated.

Continuation Inodes For Files

Ext2 provides VFS with a routine `get_block`, which is responsible for fetching a given logical block of an inode into a `buffer_head` to be mapped on a page in inode's `page cache`. `buffer_head` can be treated as a block of VFS, and `page` is a group of such `buffer_heads`. `get_block` routine is responsible for traversing the continuation inodes for an inode and finding the right block. Also, new continuation inode is created here if file is being expanded and no space exist in the primary inode's chunk or in its last continuation inode's chunk.

This routine takes care of both reading and writing to continuation inodes.

As shown in algorithm 1, `get_block` fetches the logical block number `@iblock` [1] from `@inode`. Flag `@create` is set if new block should be created if not found. If new block allocation on line 19, new continuation inode is created on line number 23. It calls algorithm `alloc_cnode`, which creates new continuation inode. Flag `CNODE_EXPAND_FILE` is passed to indicate that new continuation inode is for a file. Algorithm `alloc_cnode` is a straight forward algorithm which creates a new continuation inode in a chunk allocated by the chunk allocator.

[1] As a coding notation, formal parameters are referenced starting with symbol '@'.

Algorithm 1 get_block(inode, iblock, create)

```
1: Locate block @iblock in @inode
2: if block is found then
3:         return 0
4: end if
5: blocks ← @iblock − data_blocks(@inode)
6: if @inode has continuation inodes then
7:         blocks ← blocks − data_blocks(@inode)
8:         for all continuation inode, CI_jy, of @inode do
9:                 if blocks < data_blocks(CI_jy) then
10:                        Fetch block blocks
11:                        return 0
12:                end if
13:                blocks ← blocks − data_blocks(CI_jy)
14:        end for
15: end if
16: if no @create flag then
17:        return 1
18: end if
19: Allocate new block for @inode at logical block number blocks
20: if allocation successful then
21:        return 0
22: end if
23: CI_kz = alloc_cnode(@inode, CNODE_EXPAND_FILE), for @inode
24: Allocate new block for CI_kz at logical block number blocks
25: if allocation successful then
26:        return 0
27: end if
28: return 1
```

Routine `data_blocks` used in the algorithm returns number of data blocks in an inode. For ext2 (and most other file systems), not all blocks that are allocated for an inode contain data, as some blocks are used for maintaining information about the block pointers, e.g. *indirect*, *double indirect* and *triple indirect* blocks [18]. While locating a logical block number for a file, these blocks should be subtracted from the number of blocks that have been allocated for an inode. For ext2, field `i_blocks` in on-disk inode structure store number of blocks allocated.

Loop on line 7 loops through the continuation inode data structure described in section 3.3 with the help of `i_cnodes` field added in every inode.

Function `get_block` returns 0 if the block is found, and non-zero value otherwise. The algorithm described here is implemented in function `chunks_get_block` in [40].

Continuation Inodes For Directories

Similar algorithms with minor implementation differences are implemented for traversing directories with continuation inodes and for expanding directories across chunks. These resembles functions `chunkfs_readdir` and `chunkfs_find_entry` respectively in [40].

Directories need to be expanded when directory's chunk runs out of unused inodes and then can no more create new files. In this case, as described in subsection 3.3.2, we create a continuation inode for the directory in some other chunk having unused inodes.

Algorithm 2 is for allocating new inode which traverses continuation inode data structure in the loop on line 4. It creates new continuation inode on line 11 with flag `CNODE_EXPAND_DIR` to indicate that continuation inode is being created for a directory. Continuation inode in created if no unreserved unused inode exist in previous continuation inodes. This algorithm resembles function `chunkfs_new_inode` in [40].

Algorithm 2 new_inode(dir)

```
1: Find unused inode in @dir's chunk
2: if unused inode found then
3:         return 0
4: end if
5: for all continuation inodes, CI_jy, of @dir do
6:         Find unused inode in @dirs chunk
7:         if unused inode found then
8:                 return 0
9:         end if
10: end for
11: CI_kz = alloc_cnode(@inode, CNODE_EXPAND_DIR), for @dir
12: Find unused inode in chunk C_k
13: if unused inode found then
14:         return 0
15: end if
16: return 1
```

Deleting Continuation Inodes

For now, since no files share continuation inodes, which could well be worth more thought, all the continuation inodes are unlinked or deleted when the primary inode is unlinked. Deletion of file with continuation inodes entails deleting data structure illustrated in figure 3.2.

To delete continuation inode tree or chain, we start deleting continuation inodes first. For this, we traverse till the end of the chain and on the way freeing all but first continuation inode on every node. This allows us to traverse back the chain to the primary inode. While traversing back, we free the remaining first continuation inodes in the chain.

For example, in figure 3.2 we free the continuation inode chain in following order: CI_{me}, CI_{ld}, CI_{kc}, CI_{og}, CI_{ph}, CI_{qi}, CI_{nf}, CI_{jb}, and finally the primary inode I_{ia}.

Caching Continuation Inodes

Continuation inodes are referred to whenever the file associated with them is referred. Due to this, and since the continuation inodes are rare, they are cached in the memory as long as all the references to the file and the primary inode is dropped. In Linux, inodes are by default cached in *icache*. A structure called *dentry* acts as controller for the inode cache. *Dentry* is an in-memory representation of directory entry that is used by VFS. Dentries are stored in dentry cache called *dcache*. As long as dentries are in dcache, inodes are in the icache. Now, since continuation inodes do not have directory entries, they do not have dentries and are thus not cached by default. Thus a separate caching mechanism is deployed in order to cache continuation inodes and avoid frequent reads and writes to them.

4.2.3 FSCK Implementation

Fsck is implemented as described in section 3.4 without any design changes. This is implemented using E2fsprogs version 1.39 [26].

Chapter 5

ChunkFS Evaluation

"Do not go where the path may lead, go instead where there is no path and leave a trail."

– Ralph Waldo Emerson (1803 - 1882)

The performance and effectiveness of ChunkFS can be proved if we can convince ourselves, that:

- cross-chunk links in the form of continuation inodes are rare, and form small part of the file system.
- fsck time is indeed reduced by a large factor.

The performance of ChunkFS depends on number of cross-chunk links in the file system and forms a critical metric for ChunkFS. Our estimate that continuation inodes are rare is testified by our experimental results which show that even for a very simple chunk allocation algorithm only 2.5% of all the files in the file system has continuation inodes.

We also show that ChunkFS indeed reduces the file system checking time as expected, which is the main aim of this work. For example, fsck on a terabyte hard drive with ChunkFS is expected to be faster than it having any other file system. However, instead of calculating reduction in wall clock time, we examine how many chunks are

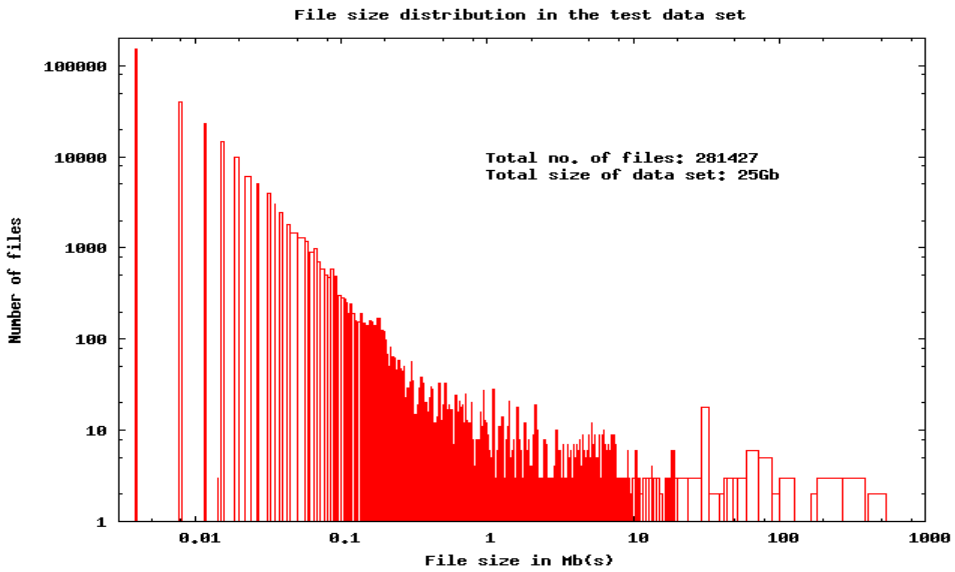

Figure 5.1: *File Size Distribution in Data Set*

actually fsck'ed on a file system with ChunkFS. This way we get an estimate of how much portion of file system was actually checked. Our initial experiments show that for a 300Gb file system with 90% data, only about 34% of the file system needs to be checked. These experiments can be improved and reverified rigorously with larger data sets.

5.1 Experimental Setup

Since we have two implementations, we experimented with both of them. For the FUSE implementation, we experimented with a 25Gb and 120Gb file system, and for the ext2 implementation we experimented with a 300Gb file system. For FUSE experiments, chunk size is set to 1Gb and for ext2, it is 3Gb. This implies, chunk size for FUSE experiments was 4% and 0.84% of the file system size, and 1% in case of experiments with ext2 implementation. All the file systems are 90% full.

The data set we used for the experiments was collected from author's machines and was replicated as necessary to fill up the file system. Practical file system benchmarks

are achieved with a file system that is realistically old and fragmented [42]. To achieve this, the script for filling up the file system with the data set occasionally deleted the data set contents.

The file size distribution for the data set used is shown in figure 5.1. This data set conforms with the typical file size distribution described in [38]. As seen, almost 98% of the files are less than 10Mb in size, and about 85% of them are less than a megabyte. Similar size distribution pattern was found in all the experimental setups.

5.1.1 Tools Used

Since the initial implementation of ChunkFS is in FUSE, we used libfuse APIs along with E2fsprogs libraries [26]. To start with, fuse-ext2fs [39] was developed which is based on [43] but is heavily rewritten.

For the ChunkFS kernel implementation, User Mode Linux (UML) was used for development, testing and experiments. UML is a kernel supported virtualization tool which allows running Linux kernel as a normal process under another Linux kernel. With UML, it is possible to use entire distribution in a single loopback file system, allowing easy changes and repair. Also use of UML makes compiling, executing and debugging kernel easier. As mentioned, UML was also used during experiments. Since the aim of the experiments was not to test the performance, use of UML did not affected the results.

GDB [44] was used for debugging purpose. And numerous shell scripts were written to manipulate the the test data obtained from the experiments.

5.2 Number of Continuation Inodes

As described in subsection 3.3.4, one of the important performance metric for ChunkFS is the number of continuation inodes it has. For the test data described above, we experimented with our FUSE implementation and found that only 2.5% of all the

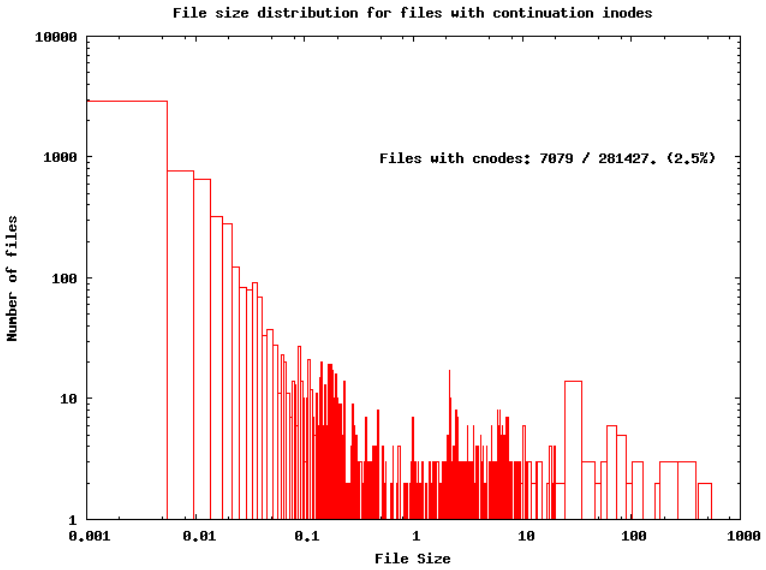

Figure 5.2: *File Size Distribution for Files with Continuation Inodes*

files have continuation inodes. The results are shown in figure 5.2.

As shown in the figure, it can be seen that there are more smaller files with continuation inodes than the larger files. Number of continuation inodes decreases as the file size increases. Our hypothesis, as stated in section 3.3, that smaller files have less continuation inodes, would falsify if we do not consider the percentage distribution of the file sizes having continuation inodes. It should be noted that files with smaller sizes are more in proportion to large files, as seen in figure 5.1.

For the sake of correct comparison, we plot a graph of percentage of files of same size having continuation inodes versus the file size. Figure 5.2 shows the graph. This graph answers the question, what percentage of files with size half megabyte have continuation inode? For example, as seen in the figure, 12.5% of files with size 280Kb has at least one continuation inode.

Graph in figure 5.2 shows percentage distribution of files with continuation inodes. This figure makes it clear that even though there are highest number of 4K files, those are only 1.9% of total of 151906 4K files having a continuation inode. Here it can be seen that as the file grows beyond 10Mb, it is likely to have a continuation inode.

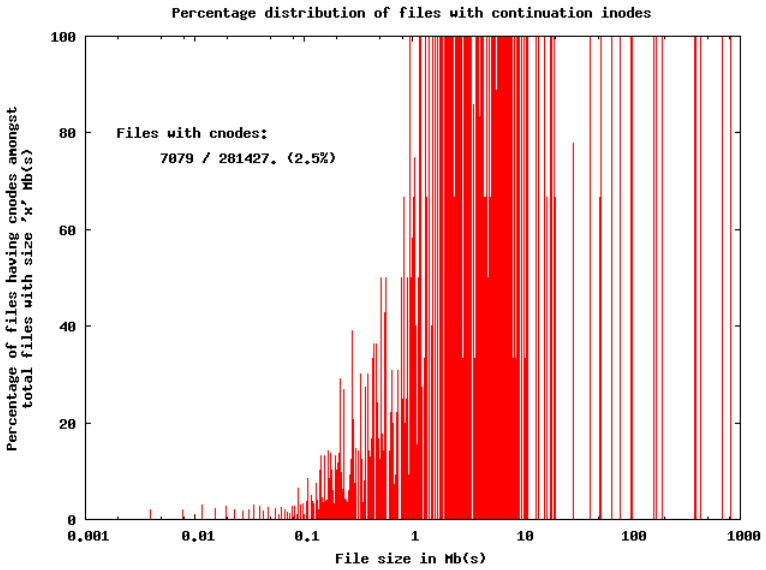

Figure 5.3: *Percentage distribution of files with continuation inode*

Although it should be understood that these figures are highly specific to the chunk size and chunk allocation policy. And even a slightly smarter allocation policy would make a big difference in the number of continuation inodes created.

We hoped that small number of files have continuation inodes and those having it do not have large number of them. To verify this we plotted a graph, shown in figure 5.2, showing variation in the number of continuation inodes as the file size having continuation inode increase. As it can be seen, files with size less than 100Mb rarely has more than one continuation inode, while a 800Mb file could have as many as 9 continuation inodes. This is not a huge number compared to number of files that are larger than 100Mb and total number of files in the file system. But, nonetheless, there is clearly some room for improvement.

In another experiment with file system size of 120Gb, we observed that 3.1% of 2046299 files had continuation inodes. From the previous experiments with file system size of 25Gb, we see only 0.6% rise in the number of files having continuation inodes. This rise could be made linear, if not constant, by adopting a better algorithm for chunk allocation and by making use of a program similar to the $defragmentor$ or fsr

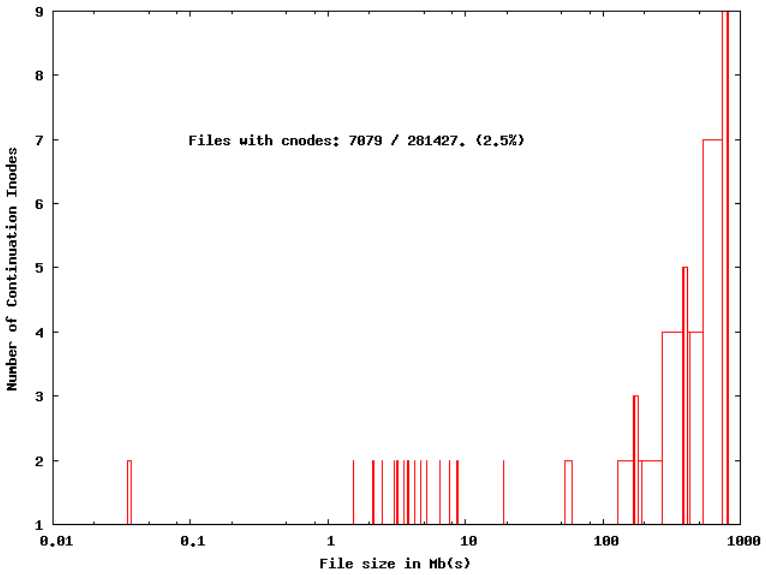

Figure 5.4: *Number of Continuation Inodes*

used in XFS file system [22]. With 120Gb similar file size distribution is observed for the files having continuation inodes.

5.3 Reduction in Fsck Time

For measuring the number of dirty chunks we used a 300Gb filled with data as mentioned in section 5.1. When the file system is 90% full, UML was crashed while some file system operation is running, leaving file system in inconsistent state. This file system was then checked using fsck developed for ChunkFS, described in section 3.4.

Graph in figure 5.3 shows the results. For 15 different crashes, on an average 102 out of 300 chunks were dirty and scanned per crash. Thus overall 33.8% of fsck time was reduced. Thus fsck on ChunkFS took 33.8% less time than on other file systems.

Even though, the preliminary results obtained look promising, more work is needed in order to validate and reexamine the behavior of ChunkFS in environments larger than 300Gb.

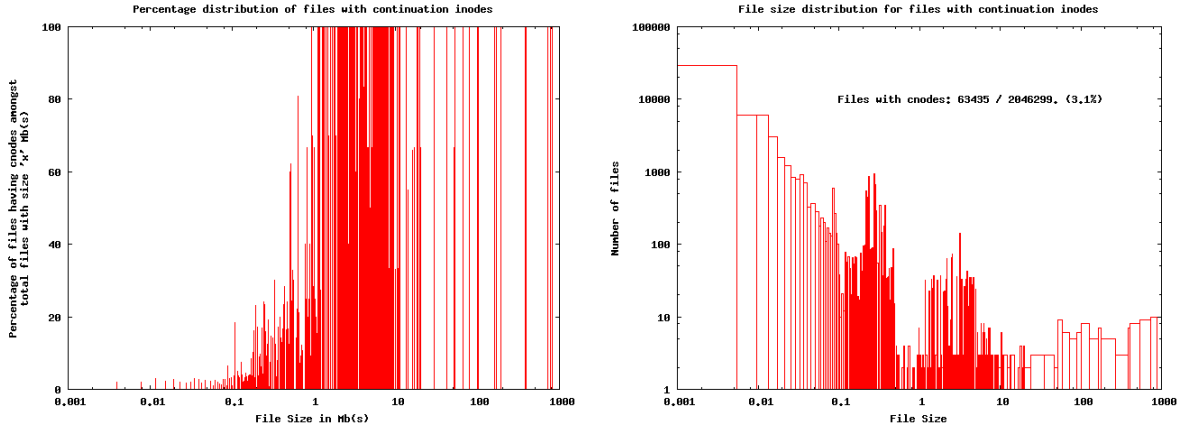

Figure 5.5: *Evaluation with file system size of 120Gb. Figure on the left shows percentage of files of same size having continuation inodes, and figure on the right shows the number of files having continuation inodes by file size.*

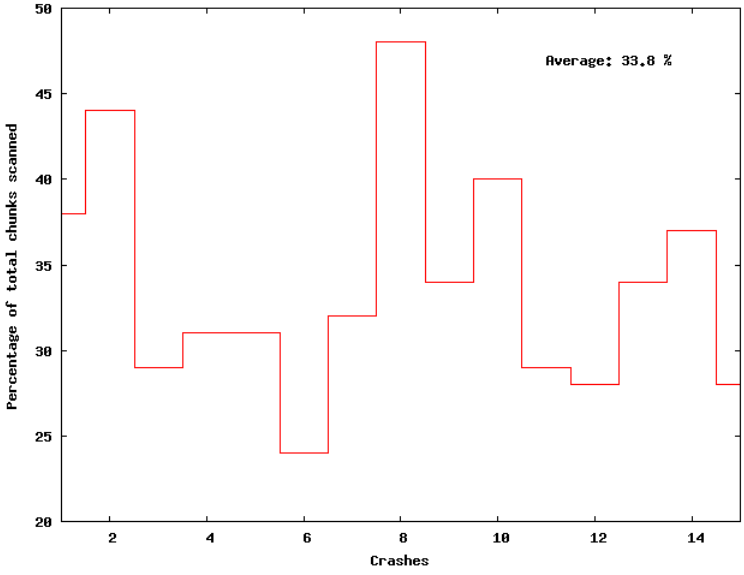

Figure 5.6: *Reduction in Amount of File System Checked for ChunkFS*

Chapter 6

Related Work

"We should not look back unless it is to derive useful lessons from past errors, and for the purpose of profiting by dearly bought experience"

– George Washington (1732 - 1799)

Most file systems designed so far focus on making data storage scalable, robust and more efficient in terms of performance. Though not directly addressed, attempts were made to have a faster recovery. The major problem most of the file systems face is that whole of the metadata is required to be read for file system checks, which makes the file system checking time $\boldsymbol{O}$*(amount of used metadata on the file system)*

Though most related techniques have been discussed in section 2.4, we will cover the remaining ones in this chapter.

6.1 Log-Structured File System

Developed in early 90s, one of the aspects of log-structured is to speed-up the file system recovery [45]. In log-structured file system, only data-structure that is kept on the disk is a sequential log. The disk is written out sequentially on the disk, which enables fast file reads and appends. The large area of file system is divided into segments for easy management.

In log-structured file system it is easy to recognize any half-updates that took place

before the crash - at the end of the log. This allows a very fast file system recovery on crash. Checkpoints are also used allowing instant recovery. These checkpoints are then used as reference marks whenever a file system is to be recovered after crash. This approach works, but any data written after the checkpoint is lost. To overcome this limitation, log-structured file system used a technique called $rolled-forward$. In this technique, segment summaries are used to determine last writes that happened to the disk. This effectively leads to checking entire metadata of the file system during file system checking.

Log-structured file systems are a type of $copy-on-write$ file systems, which do not overwrite data in place. Instead a copy is made and modified. The main problem of log-structured file system is that it requires large free segments of disk space, created by the cleaner thread, which runs periodically cleaning the old copies of the data as and when required. The overhead of cleaner thread turns out to be quite high making log-structured file system unscalable [15]. The file system repair time, however, remains proportional to the file system size and the size of data on it, as all of the metadata needs to be checked.

6.2 Copy-On-Write File Systems

COW file systems [46, 47], are structured as tree of blocks. When write happens, blocks are duplicated and changes are made to the the copy. The original copy is replaced when the changes are stabilized. This technique make snapshots very inexpensive and can be further used for online error detection. COW file systems, however, face similar problems as that of log-structured file systems as discussed above and as such do not reduce file system checking time.

6.3 Checksums

Checksums can be added at the metadata or even at the data level to ensure integrity of the data on the file system. Especially when used just for metadata, reliability of file system can be raised by a substantial amount [48]. But, despite run-time reliability, all of the metadata needs to be checked when the file system demands a repair. Thus, no gains are achieved for file system checker.

Chapter 7

Conclusion & Future Work

"Nothing will ever be attempted if all possible objections must be first overcome."

– Dr. Samuel Johnson (1709 - 1784)

Conclusion

Recovery-driven file system design is a promising step ahead. As the disk capacities explode overtime with unparalleled increase in bandwidth and flat seek times, it is mandatory for file systems to change themselves to be able to survive recovery time crunch for the coming decade. Throughout our experience with ChunkFS, we realized and observed that dividing the file system for better recovery performance is a definite way to reduce file system checking time as observed by the preliminary experimental evaluation of our prototypes. ChunkFS improves file system reliability, shortens repair time, and increases data availability by dividing the file system into chunks, small fault-isolation domains which can be checked and repaired almost entirely independently of other chunks.

Furture Work

This work leaves number of open questions, that needs further evaluation and experimentation. Aspects that needs to be studied are:

- Evaluation of other techniques to reduce file system recovery time like per-chunk

journaling, soft-updates and COW [6, 7, 46, 47, 48].

- Evaluation of different chunk allocation techniques [27].
- Experimentation with online resizing of chunks and ChunkFS.
- Experimentation with online defragmentation tool life XFS's fsr [49] to further reduce number of continuation inodes.
- Evaluation of moving ChunkFS to the VFS layer.

ChunkFS is an on-going activity and is likely to be continued to be developed and maintained by the author.

Bibliography

[1] Filesystem in userspace (fuse). http://fuse.sourceforge.net/.

[2] Mark Kryder. Future storage technologies: A look beyond the horizon. http://www.snwusa.com/documents/presentations-s06/markkryder.pdf.

[3] D. Rumsfeld. Rumsfelds rules: Advice on government, business and life. In *The Wall Street Journal Managers*, 2001.

[4] M. A. Halcrow. ecryptfs: An enterprise-class cryptographic filesystem for linux. In *The Linux Symposium*, Ottawa, Canada, 2005.

[5] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM Press.

[6] Greg Ganger and Y. Patt. Metadata update performance in file systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, page 4960, Monterey, CA, 1994.

[7] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 155–162, Austin, TX, 1987.

[8] Jim Gray and Catherine van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, 2007.

[9] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] Marshall K. McKusick. Running "fsck" in the background. In *Proceedings of the BSDCon 2002*, pages 55–64, Berkeley, CA, USA, 2002. USENIX Association.

[11] Val Henson. Repair driven file system design. http://infohost.nmt.edu/ val/review/repair.pdf, 2006.

[12] G. Cole. Estimating drive reliability in desktop computers and consumer electronics systems. Technical Report TP-338.1, Seagate, 2000.

[13] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[14] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Hot Topics in System Dependability*, 2006.

[15] Valerie Henson. The 2006 linux file systems workshop. http://lwn.net/articles/190222/, 2006.

[16] Robert Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.

[17] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1986.

[18] Remi Card, Ted Tso, and Stephen Tweedie. Design and implementation of the second extended file system. In *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[19] Stephen Tweedie. Ext3, journaling filesystem. In *The Linux Symposium*, 2000.

[20] Hans Reiser. Reiserfs white paper. http://www.namesys.com/.

[21] Hans Reiser. Reiser4 white paper. http://www.namesys.com/.

[22] Barry Naujok. Xfs filesystem structure white paper, second edition, revision 2. 2006.

[23] Andrew Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1987.

[24] The linux kernel homepage. http://www.kernel.org/.

[25] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. In *Computer Systems*, volume 2, pages 181–197, 1984.

[26] E2fsprogs release 1.39. http://e2fsprogs.sourceforge.net/.

[27] Keith A. Smith and Margo I. Seltzer. A comparison of FFS disk allocation policies. In *USENIX Annual Technical Conference*, pages 15–26, 1996.

[28] T. J. Kowalski. Fsckthe unix file system check program. In *UNIX Vol. II: research system (10th ed.)*, pages 581–592, Philadelphia, PA, USA, 1990. W. B. Saunders Company.

[29] M. McKusick and T. J. Kowalski. Fsckthe unix file system check program. In *4.2BSD UNIX Programmer's Manual, Vol 2c, Document #66*, 1983.

[30] Eric J. Bina and Perry A. Emrath. A faster fsck for bsd unix. In *USENIX Winter Technical Conference*, pages 173–185, Berkeley, CA, USA, 1989. USENIX Association.

[31] R Duncan. *Advanced MSDOS Programming*. Microsoft Press, Redmond, WA, 1986.

[32] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, pages 247–256, 1990.

[33] Gregory R. Ganger and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. pages 49–60, Berkeley, CA, USA, 1994. USENIX.

[34] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 239–249, Berkeley, CA, USA, 1999. USENIX Association.

[35] Juan Piernas, Toni Cortes, and José M. Garc. Dualfs toward a new journaling file system. In *Jornadas de Paralelismo*, volume 13, 9 2002.

[36] Bob Lyon and Russel Sandberg. Breaking through the nfs performance barrier. In *Sun Technical Journal 2(4)*, pages 21–27, 1989.

[37] Val Henson, Zach Brown, T. Tso, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *The Linux Symposium*, 2006.

[38] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems: then and now. In *SIGOPS Oper. Syst. Rev.*, volume 40, pages 100–104, New York, NY, USA, 2006. ACM Press.

[39] Amit Gud. [announce] fuse-ext2fs git tree. http://lwn.net/articles/199108/, 2006.

[40] Amit Gud. [rfc][patch] chunkfs: fs fission for faster fsck. http://lkml.org/lkml/2007/4/23/120, 2007.

[41] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[42] Keith A. Smith and Margo I. Seltzer. File system aging-increasing the relevance of file system benchmarks. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, New York, NY, USA, 1997. ACM Press.

[43] Jeff Garzik. fuse-ext2. http://www.kernel.org/pub/scm/fs/fuse/fuse-ext2.git.

[44] The gnu project debugger. http://sourceware.org/gdb/.

[45] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[46] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[47] Zfs at opensolaris.org. http://www.opensolaris.org/os/community/zfs.

[48] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, 2005.

[49] Dave Chinner and Jeremy Higdon. Exploring high bandwidth filesystems on large systems. In *The Linux Symposium*, Ottawa, Canada, 2006.

Appendix A

Terminology & Definitions

Bitmap Bitmap is a stream of bits, where each bit is treated as a flag for the resource associated with it, e.g. for inode bitmap, bit i would indicate whether i^{th} inode is used or unused.

Block group Block group is a subdivision with which an ext2 file system is splitted . Each block group has a copy of the superblock, block bitmap, inode bitmap, inode table and the actual data blocks. See figure 2.1.

Device file A device file is a special file which refer to some kind of device, either character or block. Instead of inode numbers, device files have a major and a minor number and provide raw access to the device they refer to. Character device special files are represented by letter c in the permissions field and block device special files are represented by letter b in the permissions field.

```
crw-rw-rw- 1 root root 1, 5 2006-05-02 01:45 /dev/zero
brw-r----- 1 root disk 8, 1 2007-05-08 04:09 /dev/sda1
```

Directory Directory is a special kind of file which contains listing of other files and directories. Directories provide the hierarchical structure for the file system.

Hard link A file is said to be hard-linked with a directory, if the directory contains an entry for it. The link count field in the inode specifies number of hard

links that an inode has. Since a directory entry is required, hard links cannot be created across file systems. Hard links can be created with *ln* command without the $-s$ option.

Inode Inode is an on-disk data structure used to store information about a file. It contains fields like file type, link count, various timestamps, permissions and size. See subsection 2.1.2 for more details.

Pipe Pipe is a special kind of file which connect input of one process with the output of another. This is helpful in the interprocess communication. This are also called as *named pipes*. Pipes can be created with the command $mkfifo$. Pipes are represented by letter p in the permissions field.

```
prw-r--r-- 1 gud users 0 2007-05-08 09:47 mypipe
```

Regular file This is the most common file type in UNIX. A regular file is a file which contains actual data, i.e. a text or a binary file. Regular file is represented by no symbol or $-$ in the permissions field.

```
-rw-r--r-- 1 gud users 12982 2006-09-07 10:10 tux.jpg
```

Socket Socket is a special kind of file which is used for interprocess communication in UNIX. Using sockets process can send data as well as open file descriptors. Sockets can be created using `socket` and `bind` system calls. Sockets are represented by letter s in the permissions field.

```
srw-r--r-- 1 gud users 0 2007-05-08 09:47 mysock
```

Sparse files Sparse files is a technique to efficiently store a regular files to which space has been allocated but not all data is filled. Sparse files have intermediate blocks with no data. Size of sparse files as stored in its inode differs from the size of file in terms of number of blocks.

```
11M -rw-r--r-- 1 gud users 10G 2007-05-08 10:10 sparse.file
```

Superblock Superblock is an on-disk data structure containing information about the file system, like file system identifier, number of free blocks and number of free inodes. Refer subsection 2.1.1 for more details.

Symbolic link Symbolic link is a special kind of file which points to another file. In a symbolic link, or *symlink*, the link contains path of the file that it is pointing to. Thus symbolic links could be across file systems or devices. Symbolic links can be created with $ln - s$ command. Symbolic link is represented by letter l in the permissions field of the file, and usually with an arrow pointing to the file it is referencing.

```
lrwxrwxrwx 1 root root 10 2006-05-03 12:12 tmp -> ../var/tmp
```