DIPLOMARBEIT

dtfs
A Log-Structured Filesystem For Linux

ausgeführt am Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
o. Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Dr. Anton Ertl
als verantwortlich mitwirkenden Universitätsassistenten

durch

Christian Czezatke

Schiffmannstraße 47
2191 Atzelsdorf

Atzelsdorf, am 3. Dezember 1998

Abstract

This thesis discusses the design and implementation of dtfs, a log-structured filesystem for Linux. dtfs features a generic core providing logging facilities that are filesystem-independent and a "filesystem personality" that borrows heavily from the Linux ext2 filesystem. Furthermore, the dtfs design supports the placement of multiple filesystems (even of different filesystem personalities) on top of one dtfs filesystem device and the creation of snapshots and different versions for these filesystems.

I have also made a first implementation of dtfs using the Linux 2.0.33 kernel and investigated the performance effects caused by a log-structured filesystem. The results show that this implementation of dtfs is already approximately on par with the 2.0.33 ext2 filesystem performance-wise. This also illustrates that traditional approaches have been closing the performance gap during the last years, especially when dealing with write and metadata update operations. However, other qualtitative improvements offered by the dtfs design, such as fast crash recovery or the ability to create consistent backups without restricting user access to the filesystem, cannot be added to traditional approaches easily.

Contents

Acknowledgements

Many people deserve special thanks for making this thesis possible. First of all, I owe a lot to the staff of the Computer Language Department at the Technische Universität Wien; especially to my advisor Anton Ertl. He was always there for discussing ideas and problems and asked the right questions when I tended to make things unnecessary complex.

I also wish to thank the numerous contributors to the Linux kernel, especially Linus Torvalds, the key architect of the kernel and author of the original Minix filesystem implementation for it, as well as Stephen Tweedie and Ted Tso, the key designers of ext2 filesystem.

Many other people have also influenced this project by providing design ideas and suggesting useful features to consider. Cornelius "Kees" Cook has been especially helpful during the first stage of designing dtfs. Martin Baehr provided valuable insights when discussing possible log-structured filesystem applications that go far beyond the way filesystems are used today. Martin Sperl's ideas on very large filesystem support have also been inspiring.

I also want to acknowledge the contributions of Markus Würzl for proofreading and suggestions for the paper and for providing "moral support". Furthermore, working together with Gerhard Reithofer during the last few years has allowed me to get some insight into commercial software development techniques that were extremely useful when designing and implementing dtfs.

This thesis would not have been possible without the support of my parents. Without their support this thesis would have never been written.

Finally, I also thank Andreas Haumer of xS+S for providing the test system on which the dtfs performance measurements were done.

Part I

Design

Chapter 1

Introduction

1.1 Traditional Unix Filesystem Designs

1.1.1 The Purpose Of A Filesystem

The main purpose of a filesystem is to provide a layer of abstraction that allows the user to deal with files and directories on top of a block-oriented storage medium. Thereby, filesystem has to bridge the gap between applications using concepts such as files and a hierarchical directory structure and the basic block read/block write services offered by the device.

1.1.2 Basic Unix Filesystem Concepts

This section introduces a few basic filesystem concepts that can be found in all Unix-like operating systems. Only issues required for a deeper understanding of certain dtfs design decisions are outlined here; so this section is far from being a complete tutorial to Unix filesystem internals.

Publications are available that discuss several approaches to managing filesystem data taken by different operating systems [Sta92], while others focus on Unix-specific filesystem issues in more detail [Vah96].

Inodes

The key concept in every Unix filesystem is the concept of an *inode*. Every filesystem entity is represented by exactly one inode. Information that is common to all the different kinds of filesystem objects is represented in this data structure. The following entries can usually be found in an inode:

- the *inode number*, a filesystem-wide unique identifier for the inode;[1]
- the kind of the filesystem entity it represents (like file, directory, device, pipe, symbolic link, etc...);
- how to retrieve the data represented by the filesystem entity;
- the user and group owning it;
- access permissions;
- the number of times the filesystem entity is referenced in directories (the link count);

- various timestamps.

Directories and Inodes

Directories can be considered to be lists that associate user-visible names to inode numbers. The number of times an inode is referenced by such directory entries must correspond to the link count of the respective inode.

When the link count of an inode drops to zero, the inode is not referenced by the filesystem anymore. This means that the information associated with it is no longer accessible and can be discarded.

This behavior is illustrated by the fact that Unix operating systems do not provide a call for deleting filesystem entities. The only thing that is present for such purposes is the *unlink* functionality: Unlinking a filesystem entity means to remove a certain reference from a directory pointing to its inode. It is up to the operating system to actually release the resources allocated for the particular filesystem entry once the link count of its inode has dropped to zero.

Mapping Everything to the Filesystem

Filesystems in a typical Unix environment do not only hold user-data. They are also used for representing hardware devices and interprocess communication facilities.

Every hardware device like a harddisk, audio I/O equipment or a serial communication line is mapped to a special file, a so-called *device node*. This approach allows the user to access raw device data using the same set of commands that is used for ordinary files. One common example for the usage of this feature is the way the standard Unix `cat` commando can be used in order to play back and to record Sun Audio Files on some platforms: Playing back such files is simply done using the command `cat audiofile.au > /dev/audio`.

A similar technique is used for mapping certain interprocess communication facilities to the filesystem.

Filesystem Metadata

A filesystem must usually maintain a few on-disk data structures that are not visible from the filesystem's higher level interface. This information is used for internal housekeeping duties, such as

- locating all the data blocks of a file or directory on the device;
- keeping track of free disk space so that new, unallocated blocks can be found fast;
- finding free inodes.

Data blocks are usually located by using information that is stored in the inode directly. However, for bigger files it is necessary to employ some kind of indirect addressing scheme that allows to address a larger number of blocks.

[1] Please note that this unique identifier does not necessarily have to be explicitly stated in the inode data structure. In ext2, for example, this inode identifier is derived from the inode's physical location on the underlying device.

1.2 Other Filesystem Design Issues

1.2.1 Handling Block Devices

As already ountlined in section 1.1, a filesystem must act as a mediator between a block-oriented storage device and a higher level view of the information stored on it. While only the concepts used in this higher level interface have been discussed so far, a good filesystem implementation must also take some peculiarities of the underlying storage medium into account.

Generally, filesystems are used on block devices, such as harddisks. A harddisk basically consists of one or more rotating magnetic disks that are used for actually storing the data. The information stored on these disks is laid out in concentric tracks that are again subdivided into sectors. This division into tracks and sectors partitions the medium into a number of equal-sized blocks of data. Such blocks that always read in or written out as a whole resulting in a block-oriented view of the data stored on the medium.

The information is read and written using magnetic heads that can be moved radially with respect to the spinning disk. So any block of data can be accessed by positioning the disk head over the appropriate track and waiting for the desired block to apper under the magnetic head as the medium rotates.

Therefore, accessing blocks that are randomly distributed across the device takes much longer than reading in the same number of blocks sequentially. This is due to the fact that reading in sequential disk blocks requires only one initial head movement and only one period of rotational delay while random reads encounter these penalties for every block to be accessed.

So one major goal of a filesystem designer is to lay out the data on the underlying device in a way that reads and writes can always be performed in large, sequential chunks in oder to achieve good performance.

1.2.2 New Challenges For Filesystem Designs

Apart from finding an efficient on-disk placement policy for the data to be written, other challenges for filesystem designs emerge as the underlying storage technology evolves.

High Availability

Fast Crash Recovery As secondary storage devices are getting bigger, new ways must be found to get the filesystem into a consistent state after it has not been unmounted cleanly.[2]

Exhaustive metadata consistency checks as they are required for traditional filesystem designs are not an option for multi-gigabyte devices since the downtime resulting from a filesystem check can be unacceptably long for many applications.

Low Maintenance Overhead Traditional filesystems require many maintainence tasks to be performed with no user access going on in parallel. This is particularly true for backing up the current state of the filesystem. However, in many applications the downtime resulting from such maintainence tasks are unacceptible and have to be avoided.

Therefore a filesystem should provide mechanisms that allow many regular maintainence tasks to be performed without restricting user access.

[2] This might be the case because of a power failure or an operating system crash, for example.

Undoing Filesystem Changes

A filesystem should also provide means of undoing recent filesystem changes. This does not only involve the ability to recover accidentially deleted files; undoing recent (errorneous) updates to file is also desirable.

Such requirements will become more and more stringent as Unix systems are used by people who are unaware of the potentially destructive nature of many filesystem operations.

Dynamically Resizing an Existing Filesystem

Filesystem implementations should also be aware of the fact that the underlying block device interface they use for communicating with the storage device is getting more and more sophisticated. Many commercial Unix vendors have added features such as logical volume management to their products. Amogst other things, a logical volume manager allows to dynamically grow and shrink a device with a filesystem on it.

In order to be able to fully exploit the benefits of a logical volume manager, it must be supplemented with a filesystem design that is able to react to changes to the size of the underlying device without loosing its data.

1.3 Key Log-Structured Filesystem Design Issues

1.3.1 Append-Only Log Writes

Log-structured filesystems add a new layer of abstraction between the physical device and the filesystem implementation. Instead of a block-oriented view where the contents of every block may be changed at any time, the underlying device is presented to the filesystem as a continuous log. All writes are done to the tail of the log in an append-only manner. Furthermore, write operations are performed in a way that they get transaction characteristics: A write is either completed as a whole or not recognized at all.

This approach has some severe consequences for the way filesystem information is laid out on disk. One noteworthy difference gets obvious when it comes to altering existing filesystem data, for example: While traditional filesystem implementations do data modifications using an "update in place" policy, log-structured filesystems re-write the altered information to a new disk location. As a consequence of that, all the metadata used for referencing the information has to be updated (i.e.: appended to the log), too.

This append-only strategy is required because simply updating existing data could compromise the transaction-like semantics of log writes, since failed writes would corrupt old data that is already considered to be stored permanently. Furthermore, never overwriting old information, is also of vital importance for implementing many services (such as those outlined in section 1.2) that set log-structured filesystems apart from other designs.

This log-aware strategy also has another effect on the way filesystem data is laid out on disk. While traditional filesystems try to group filesystem entities together that are *logically* related to each other, log-structured filesystems actually feature a *temporal locality* since all new information is written to the end of the log.

1.3.2 Reclaiming Free Disk Space

Viewing the underlying device as an infinite, append-only log is convenient when implementing a log-structured filesystem. However, since disk space is actually limited, the log will finally reach the end of the device.

Simply deleting filesystem entities does not immediately free up disk space in a log-structured environment, as it would do for a traditional filesystem. If a file is removed from a log-structured filesystem, the blocks belonging to it are still present in the log, but the data is not referenced anymore.

A separate tool, the so-called *cleaner*, that finally reclaims free disk space by discarding such unreferenced filesystem information is needed. This is similar to the task performed by a memory garbage collector found in many modern programming languages.

In order to ease free space management, log-structured filesystems divide the underlying disk medium into *segments*. Segments are continuous areas of disk space (typically 512KB in size or larger) that constitute the unit of cleaning: A segment is either considered as being part of the log or as being unused. The task of the cleaner is to find segments that are part of the log and contain mostly unreferenced data. Then the data that is still referenced ("live") has to be copied to the end of the log. After that, the segment contains no live information anymore and can therefore be removed from the log and marked as free again.[3]

Furthermore, the partition of the device into even-sized segments that are either considered as used or unused, is the basis for the integration with a logical volume manager that allows to dynamically resize a filesystem without loosing the data stored on it: Enlarging the filesystem can be mapped to adding additional free segments to the filesystem, while shrinking it can be done by removing free segments.

[3]Another cleaning strategy would be not to simply discard unreferenced blocks, but to move them to a tertiary storage device thereby creating a hierarchical storage management. This would allow to automatically archive old versions of filesystem entities to a tertiary storage device, for example.

Chapter 2

Basic dtfs Design

2.1 Overview

This thesis presents a design for a Linux filesystem called "dtfs" that addresses the issues outlined in section 1.2. They are achieved by using a log-structured approach that brings transaction-like semantics — similar to databases — to disk writes. The basic dtfs design is based on principles developed for Sprite LFS [Ros92] and the 4.4BSD log-structured filesystem [SBMS93].

2.2 Design Goals

As already discussed in section 1.2, dtfs has been designed with qualitative enhancements to filesystems in mind that cannot be easily integrated into other approaches, such as

- fast crash recovery;
- undoing even multiple levels of filesystem changes;
- dynamically resizing of existing filesystems without loosing the data on it;
- unrestricted access even during maintenance tasks, such as backing up the filesystem.

Furthermore, a few more requirements have been introduced, that are not particular for log-structured filesystem designs. These issues are

- handling of large storage devices;
- "year 2037 compliance";[1]
- platform-independence.

[1]Unix-like operating systems use a standardized internal time representation: The current time is represented as the number of elapsed seconds since the begin of the "epoch" (Jan. 1^{st}, 1970, 00:00GMT). By default, this value is represented as a 4 byte, signed integral number, so that this counter will overflow in 2037.

dtfs Design Goal	Implemented by
Fast Crash Recovery	log-structured approach per-se, efficient checkpointing
Undoing Filesystem Changes	log-structured approach per-se, hierarchic storage management
Resizing Filesystems	log-structured approach per-se, support for logical volume management
Large Storage Devices	64 bit data structures, multiple logical filesystems
Unrestricted User Access	versioning
Year 2037 Compliance	64 bit data structures
Platform Independence	standardized byte order

Table 2.1: Overview of techniques used by dtfs in order to meet the requirements stated in 2.2.

2.3 Implementing These Goals

This section presents the techniques used by dtfs in order to meet the requirements stated above. While some design goals can be addressed by using a log-structured approach per-se, others require special mechanisms to be provided or are general mechanisms that are useful regardless to the nature of the filesystem implemented. Table 2.1 lists a summary of basic dtfs requirements and the mechanisms used for providing them.

Efficient Checkpointing

Existing LFS implementations like BSD LFS or Sprite LFS use sophisticated techniques to avoid possible inconsistencies in the filesystem after a crash resulting from only partial recovery of some operations that are not self-contained. An example of such an operation would be the creation of a file requiring changes to the directory structure as well as the allocation of an inode.

When the filesystem is being recovered after a crash, it must be made sure that either both of these operations (directory modification, inode allocation) are performed or none of them or the filesystem will be in an inconsistent state.

Other LFS implementations use sophisticated recovery algorithms based on additional information in order to detect such problems.

dtfs tries to avoid this problem by adhering to a simple convention when it comes to reconstructing a filesystem after a crash: Recovery is only done on a checkpoint-to-checkpoint base.

In order to compensate for that, dtfs introduces more lightweight checkpoint data structures that allow checkpoints to be written more frequently without big performance penalties. This is described in section 7.

A Simple Hierarchic Storage Management

Generally speaking, a log-structured filesystem writes its data in an append-only way. Since the disk space that is actually available on a harddisk is limited, a special user-space program (the cleaner) is required in order to reclaim disk space occupied by blocks that are not referenced by any current checkpoint anymore.

A conventional cleaner will simply discard this information in order to reclaim free segments . However instead of simply deleting them, it would also be possible to transfer such segments to a tertiary storage medium. This allows the reconstruction of the filesystem's state at any checkpoint at any given time by using the information on this tertiary storage medium, such as magnetic tapes.

dtfs supports this policy by numbering segments as they are written. Segment numbers are ever increasing, unique 64 bit identifiers. Together with the knowledge of the original physical location of the segment on the secondary storage device, these segment numbers can be used to reconstruct the state of the filesystem at any checkpoint.

The knowledge of the original location of the segment is required since filesystem-specific metadata structures are generally unaware of such issues and use some kind of physical addressing scheme based on disk-block numbers. Therefore this addressing scheme would not work without the knowledge of the original on-disk location of every segment.

Therefore a modified cleaner supporting this kind of hierarchical storage management would also have to build a segment number to original physical location translation table which would serve a similar purpose as the logical to physical address translation table in a virtual memory system.

However, the exact layout of such a translation table is not specified by this document since it is not part of a core dtfs filesystem.

Easy Integration With A Logical Volume Manager

Many operating systems use some kind of a logical volume manager that allows to dynamically resize logical volumes containing filesystems without loosing the data on the filesystem [IBM93]. The segment-oriented approach to free space management in a log-structured filesystem and the ability to explicitely clean segments in a given area can be used to add or to remove parts of the underlying "device block address space".

64 bit Wide Data Structures

dtfs incorporates 64 bit wide data items into crucial data structures of the dtfs core in order to avoid potential problems with running out of address space in the near future (as it is currently the case for ext2 with the 2GB file size limitation). Furthermore, dtfs' core data structures are already "year 2037 compliant": Most current Unix implementations use a 4 byte representation of the system time that will overflow in the year 2037 causing similar problems to the ones currently faced with software that is not "year 2000 compliant".[2] An eight byte representation has been chosen in dtfs for the following data items:

- "logical" dtfs time stamps;
- disk block addresses and disk sizes;
- segment numbers.

A four byte representation has been chosen for "natural" time values. However, whenever such a time field appears in a dtfs data structure, it is always preceded by four reserved bytes that are required to be set to zero by current dtfs implementations. Furthermore the start of these reserved bytes is always aligned to an eight byte boundary respective to the beginning of the data structure. So these four

[2] Although it is rather unlikely that dtfs will still be around in 2037, it should be avoided to make the same mistake again that has been made by some Cobol programmers assuming that their software will not be in use anymore at the turn of the millennium. The designers of 4.4BSD have also already decided to use a 8 byte representation of the system time in their filesystem [SBMS93].

bytes together with the currently used four byte representation of the item can also be seen as an eight byte word conforming to the RFC 1014 XDR specification.[3]

Multiple Logical Filesystems

The decision to support multiple logical filesystems and different versions of a filesystem in dtfs has had a major impact on dtfs design. The key concept for this feature is the separation of dtfs into a generic core and code that is specific to a filesystem residing within a dtfs partition.

Furthermore, different *filesystem personalities* can be implemented using this generic core. A filesystem personality is an adapted version of a *traditional filesystem*, such as ext2, that has been modified to work with the generic dtfs core.

It is possible to place more than one filesystem (that even use different filesystem personalities) within one dtfs filesystem. From a user's point of view this means that a partition containing a dtfs filesystem could export more than one root directory. Each root directory would then be the starting point for an independent filesystem tree, a *logical filesystem*.

dtfs Versioning

Probably more useful than placing multiple filesystem in one dtfs filesystem is the ability to do *versioning*. This feature allows the definition of *versions* of a logical filesystem at any point in time. Such a version can then be turned read-only (*snapshot*) or again be used for read/write access (*clone*) resulting in two different versions of a filesystem existing from that point in time onwards.

Having the ability to define read-only snapshots of a filesystem while users can still access the filesystem in read/write mode is quite convenient for backup purposes. A backup is meant to represent the status of a filesystem at a certain point in time.[4] In reality the problem with backups is that it takes a considerable amount of time (minutes to hours) to create them since the speed of current backup media is rather limited.

If the filesystem is accessible in read/write mode by users while the backup is in progress, the state of the backup being generated is not well-defined since a user might alter files while the backup is going on and the actual file version being saved on the backup medium is subject to a race condition between the user process altering the file and the backup process reading it in and copying it to the backup medium.

In order to avoid that problem, the system administrator restricts access to a filesystem for ordinary users during backup time. Sometimes, systems are even rebooted or brought to single-user mode before the backup is started just to ensure that there are no users left writing to the filesystem in question.

As a consequence of that, some important services are temporarily unavailable while the backup is taken. — This is one reason why backups are often performed at a time when only very little user activity can be expected. However, in many environments such a "denial of service approach" can still be unacceptable.

A dtfs implementation with versioning support can circumvent that problem by defining a read-only version of the filesystem to be backed up at a certain point of time t. Creating such a snapshot does not take much longer than creating any other dtfs checkpoint, so the delay is within a limit that is expected for normal

[3] GCC eases the implementation of 64 bit integral data types by already supporting the long long data type on 32 bit platforms. GCC can handle long long add, subtract and compare instructions, but requires library support for long long divides that is of course not available in kernel space.
However, dtfs gets along with long long adds and comparisons in kernel space pretty well.

[4] This is required for consistency reasons.

filesystem operations. This snapshot is then used as a starting point for a backup that is guaranteed to match the filesystem status as it was at the time t. Users can still continue to access the filesystem in read/write mode while the snapshot is being backed up.

Standardized Byte-Order

Since Linux is an OS that runs on a variety of different hardware platforms, a filesystem for it should be designed in a way that eases interplatform operability: A filesystem created on one particular hardware running Linux should be accessible from any other hardware platform supported, too.

Because Linux supports a variety of different processors with different byte orders and different native word sizes, it is necessary to specify a platform-independent layout for all the on-disk metadata structures. dtfs uses byte ordering according to the RFC 1014 External Data Representation (XDR) specification [Sun87] on all platforms. Therefore a dtfs filesystem should be readable on any platform regardless to the platform on which it has been created.

However, this causes a problem with filesystem personalities derived from traditional filesystem implementations that are not specified in platform-independent way.[5] In order to cope with that problem, dtfs maintains a flag for each filesystem created within a dtfs partition that indicates whether the filesystem in question has been created on a little-endian platform and therefore does not conform to the RFC 1014 XDR standard.

The filesystem personality implementation can then take care of changing the byte order of its own metadata structures in a separate translation layer, should this be required. This approach has been chosen over the one of stating that traditional filesystems have to conform to the RFC 1014 XDR specification in order to minimize the changes necessary to existing filesystems when porting them to dtfs. Furthermore this should also help to minimize the amount of code that cannot be shared between the native implementation of a traditional filesystem and its dtfs version.

2.4 Design Summary

Investigating the original goals to be achieved and the mechanisms outlined above that implement them, I've arrived at the following key decisions about the overall architecture of dtfs:

- Separation into a generic core and a filesystem personality implementation.

  In order to be able to provide advanced features, such as versioning and hosting of multiple filesystems on one dtfs partition, dtfs must have a generic core, called *dtfs core*, that implements the continuous log for the dtfs filesystem personalities. Filesystems can use the core for reserving disk space and for writing blocks to the log. They accumulate blocks to be written and hand them over in a large chunk, called *write cluster*, to the log. The log then tries to write out this write cluster in a large, sequential chunk.

  Furthermore, the log must be able to locate all the information required for mounting a particular version of a particular filesystem.

- Adaption of the existing ext2 implementation to turn it into a dtfs filesystem personality.

[5] ext2 uses a platform-dependent byte-order up to kernel 2.0.33. However, this seems to have changed with 2.0.34pre16.

> Using the existing Linux ext2 code as a basis for a dtfs filesystem personality implementation allows to re-use a big amount of code that is already very well tested and therefore extremely stable for dtfs. Furthermore, future enhancements to ext2, such as the addition of access control lists (ACL) or the introduction of a more sophisticated directory structure should be available for dtfs, too. Furthermore, re-using dtfs code greatly reduces the amount of work required for getting a first dtfs filesystem personality implementation up and running.

So the next tasks to be performed were to come up with a design for the generic dtfs core and an outline of modifications required to be able to adapt the existing ext2 implementation to it. This basically involved the definition of suitable data structures for the core and the ext2 filesystem personality.

These tasks will be discussed in the next two chapters.

Chapter 3

Designing The Core

3.1 Introduction

As outlined in the previous section, the dtfs core has to fulfill two different tasks, namely

- providing the abstraction of an infinite log to filesystem personalities;

  This involves the task of managing free disk space as far as the underlying block device is concerned. Furthermore, services must be offered for the filesystem personalities for writing data to the log.

- locating the information necessary for mounting a given version of a certain filesystem within a dtfs partition.

The design of the data structures used by the core is heavily influenced by the need to support more than one filesystem on a given dtfs partition.

3.2 Implementing The Infinite Log

3.2.1 Free Space Management

In order to keep track of disk space allocation, dtfs takes the approach of dividing the whole underlying block device into even-sized segments, as described in section 1.3. Such a segment is considered as either to be available for writing log information to it or as being already part of the log.

Segments in the Log Form a Doubly-Linked List

dtfs uses a doubly linked list of segments that are already in the log. This is beneficial for various filesystem tasks, such as crash recovery and cleaning: Making every segment in the log point to its successor eases the task of finding the tail of the log after a dtfs filesystem has not been unmounted cleanly. It prevents the dtfs implementation from having to scan the entire device in order to be able to find the tail of the log.

Besides that, the task of a cleaner can be eased by having every segment in the log point to its predecessor, too. This is convenient whenever a cleaner wants to remove a segment from the log that does not contain live information anymore: The header of the preceding segment must be modified to point to the successor of the segment being cleaned, too, in order to remove any reference to the segment to be cleaned from the log. However, care must be taken when implementing the removal of a segment not to compromise the append-only nature of the log.

A Segment Usage Bitmap Keeps Track of Allocated Segments

In order to be able to keep track of allocated and unallocated segments, dtfs maintains a segment usage bitmap. Every bit in the segment usage bitmap corresponds to one segment in the filesystem. If the bit is set, the corresponding segment is in use. The dtfs kernel implementation sets bits in the segment usage bitmap as writes to the log fill the corresponding segments, while the cleaner will re-set bits to 0 as it frees the corresponding segments.

Actually, the segment usage bitmap contains only redundant information, since a scan across the entire device can be used in order to reconstruct the information stored there. However, this would lead to unacceptable performance for log writes as the underlying device gets filled and the dtfs implementation is trying to find another unallocated segment to write to.

3.2.2 Writing To The Log

After having discussed the way that is used for managing free disk space, it is necessary to describe how the data filling up the free disk space is actually getting there.

Forming a Partial Segment Write

From the point of view of the log, there are one or more versions of one or more filesystem personalities present, that write to it. Every now and then, such a version of a filesystem personality will decide to write a certain amount of dirty blocks to the log. Such a number of blocks is called a *write cluster*.

If the log is asked to put a write cluster to disk, it also starts asking all the other filesystem personality versions that are currently active, for blocks to be written out. This is done for efficiency reasons: Firstly, writing out many blocks in one large, sequential write operation is less time consuming; secondly the administrative overhead required is reduced. Since every write to the log has to be committed with a checkpoint block, writing out as many blocks as possible in one log write operation reduces the overhead required for the log.

After the log has acquired all the write clusters from all the active filesystem versions, it starts writing them out one after each other. Finally, such a write operation is committed by writing out one single block, the *checkpoint block*.

The task of writing out a number of write clusters from different filesystem versions and committing this write with a checkpoint block is called a *partial segment write*.

Checkpoint Block Overview

The log also adds some metadata to every partial segment that gets written out: For every block in every partial segment write, there is a short description present in the checkpoint block. This description holds information required by the cleaner to perform a live block test. The outcome of this test for all the blocks in a segment is the basis for a cleaner's decision of whether or not a certain segment is to be freed.

Furthermore, the information stored here can also be used for making checkpointing more efficient in some circumstances, as it will be described in section 7.

In addition to that, every filesystem version that has placed a write cluster in a certain partial segment, is allowed to attach custom information to its write cluster. This is necessary since every filesystem personality implementation is responsible for maintaining its own metadata structures, such as inodes.

It can use this additional information as a hook for accessing filesystem personality-specific metadata.

The discussion of the ext2 personality design for dtfs in chapter 4 will show how this additional data can be used by a filesystem personality implementation for this purpose.

3.3 Finding a Filesystem Version

3.3.1 Three Levels of Metadata

dtfs allows to place more than one filesystem that can have more than one version on a single partition. This has great influence on the data structures required for storing information about the various filesystems. Three different levels of information can be distinguished in dtfs:

- Data that must be maintained for every dtfs formatted partition.

  This refers to the dtfs super block that holds some geometry information about the dtfs filesystem as a whole, for example. Furthermore, a segment usage bitmap must be maintained on aper-device base.

- Information about a particular filesystem residing on a dtfs partition.

  It consists of additional read-only information that a filesystem personality might need to maintain, such as a super block for every filesystem on the dtfs partition.

- Information that is specific for a particular version of a particular file system.

  This data basically consists of a pointer to the latest checkpoint block containing a write cluster for the particular filesystem version.

Unfortunately, placing all this information into the log itself is not an option, since a fast way to find the latest checkpoint block holding a write cluster for a particular filesystem version must be provided. Therefore I've decided to place only filesystem version specific information into the log at all. This means that the segment usage bitmap is not written to the log, too, since it is not filesystem-version specific.

On the other hand, it must be made sure that updates to data structures not residing in the log do not compromise the commit semantic of append-only log writes. So these data structures are duplicated and used alternately for updating. If updating the current version of such a data structure fails, there is still a valid copy left for use.

3.3.2 Basic dtfs Data Structures

This design decision has lead to the need for a few metadata block structures that are not mapped to the log by dtfs. The following enumeration lists all different metadata structures required for dtfs and classifies them according to the type of metadata information they hold according to the criteria introduced in 3.3.1:

1. dtfs super block (per dtfs partition);
2. filesystem super block (per filesystem);
3. checkpoint area (per filesystem, holding version-specific information);

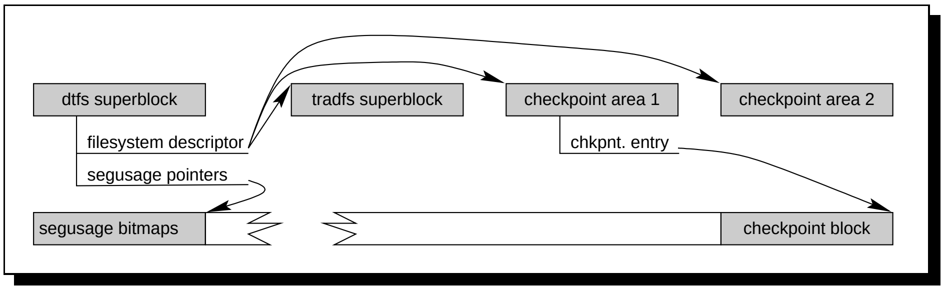


Figure 3.1: An overview of dtfs metadata referencing.

dtfs Super Block

The dtfs super block holds general information about the filesystem like various geometry specifications. Furthermore, the dtfs super block holds a number of filesystem descriptors. Each filesystem descriptor represents a filesystem within a dtfs partition. Such a descriptor contains pointers to the super block and to the checkpoint areas of the respective filesystem.

Filesystem Super Block

Every filesystem within a dtfs partition has its own, traditional super block. This is necessary since many filesystems expect information like the maximum number of inodes in the filesystem to be available.

Checkpoint Area

Each filesystem has two checkpoint areas that are pointed to by its filesystem descriptor in the dtfs super block. The checkpoint area is the only dtfs metadata structure listed so far that is not mapped to the log, but must be updated from time to time during normal filesystem writes. Checkpoint areas contain checkpoint entries that represent the state of a certain version of the filesystem at a given point in time. A checkpoint area must have at least one checkpoint entry. The checkpoint entry in turn points to a checkpoint block in the log that holds the latest write cluster of the respective filesystem version.

Every filesystem has two checkpoint areas that are updated alternately, so that still valid information is available should the write to the checkpoint area fail.

If there are no valid checkpoint areas for a certain filesystem, the checkpoint area information can be reconstructed by a sequential scan through the whole log (which is a rather time-consuming operation).

3.3.3 dtfs Metadata Referencing Summary

Figure 3.1 shows how the different kinds of metadata information are linked together. Starting from the dtfs superblock, the superblocks of all the traditional filesystems, the segment usage bitmaps and the checkpoint areas can be found. So the only block that has to be on a known location when mounting a filesystem from a dtfs partition, is the dtfs superblock.

Chapter 4

Designing dext2

4.1 Introduction

This chapter outlines a few architectural changes that are required for ext2 in order to be able to turn it into *dext2*. This term is used to refer to "dtfs ext2", the dtfs filesystem personality based on the original Linux ext2 kernel sources.

4.2 ext2 Metadata Information

4.2.1 General Considerations

ext2, being a standalone filesystem of its own, maintains several kinds of different metadata information. In this context, metadata information refers to any information that is not mapped to file or directory contents, such as block allocation bitmaps, inodes or indirect blocks.

In order to make ext2 work with the dtfs log, this metadata information must be classified and mapped to mechanisms that are compatible with the dtfs log.

It is important to point out that all the ext2 metadata information required for a dtfs filesystem personality must also be written to the log, just as any other filesystem data, or the append-only nature of the log will be compromised.

4.2.2 ext2 Metadata Classification

ext2 metadata information can be divided into two groups with respect to a dtfs port: Some metadata information is unnecessary in conjunction with dtfs, since its tasks are taken over by the log, while other data structures have to be altered in order to turn ext2 into a dtfs filesystem personality.

Redundant Information

ext2 uses a collection of bitmaps that keep track of free and unused disk blocks and the total amount of free disk space. However, since disk block allocation is now done by the log implementation, the information represented by these bitmaps is now redundant so that they can be discarded.

Information That is Still Required for dext2

Furthermore, ext2 also maintains metadata information that is still required in dext2. Therefore a similar representation of this information must be provided in dext2, too:

- inode information;

  ext2 stores inodes in dedicated disk areas. However, inodes must also be mapped to the log somehow for dext2, so storing them at well-known locations that allow immediate computation of an inode's on-disk location just by knowing its number, is not an option.

- inode allocation bitmap.

  dext2 also needs a way to find a free inode fast. Again, using an inode allocation bitmap at a fixed on-disk location is not an option for a log-based ext2.

4.3 Mapping Metadata to Files

4.3.1 Overview

In order to keep things simple and to get a consistent design, dext2 maps all the metadata structures it requires into regular files. dext2 gets along with the following metadata structures being mapped to files:

1. The .inode file: All inodes are mapped into a single file called .inode.
2. The .iusage file: This file contains information about the inodes' allocation data. It is used to locate a free inode quickly, should the need arise.
3. The .atime file: It contains cleaning hints and the time of the last access for each file. This information is not stored into the inode itself directly because

   (a) every read access to a file needs to update the atime entry of the respective inode. Writing the entire inode structure would result in an unnecessary overhead that can be avoided by keeping the inode data and the time of the last access to the file separated.

   (b) the cleaning hints for the file as used by a log-structured filesystem cannot be found in the inode structure of a traditional filesystem. In order to avoid portability problems, it is a good idea not to add them to the inode of the traditional filesystem since it cannot be expected to conveniently fit into the inode structure.

The only information that dext2 does not map to a file is the inode for the .ifile file itself. However, in section 3.2.2, it is stated that the log allows a filesystem personality to attach information to a write cluster when it is being written out. dext2 uses exactly this mechanism to store the current ifile inode. Once the ifile inode is located, all other metadata information can be accessed by using ordinary file read operations.

4.3.2 The .ifile File

This file contains all the inodes of the traditional filesystem with the exception of the inode for the .ifile file itself. In order to allow quick access to an inode, dtfs relies on the metadata structures of the filesystem personality implementation, since inode accesses are finally translated into simple file accesses, too.

4.3.3 The .iusage File

General Purpose of This File

This file contains metadata information that allows to locate a free inode in the vicinity of another given inode quickly. It holds the inode metabitmap; a concept that is explained in this section.

The Sprite LFS implementation marks free inodes in the inode map file. Locating a free inode is done by a linear search through that file starting from the entry representing the directory inode holding the parent directory of the file to be created. The advantage of this approach is that inodes for files located in the same directory are kept close together resulting in good locality.

Inode Locality vs. Finding a Free Inode Fast

The problem with this approach is that it involves a linear scan through the inode map. Margo Seltzer et.al. [SBMS93] have investigated that problem and have found out that the Sprite LFS has to search an average of 94 bitmap entries (worst case is approx. 120 entries) before it finds a new inode that is available. Therefore this approach is not very efficient.

In order to overcome this problem, the BSD LFS implementation uses a list of free inodes that are available for allocation. However, such an approach makes it harder to get all the inodes of files belonging to a specific directory clustered near by the directory inode itself.

The approach taken by dtfs (using an inode metabitmap) should provide both an acceptable inode location performance and good locality for the inodes of all files within a directory. In addition, it should result in no additional block-writes due to inode-allocation metadata updates in most cases.

dtfs Inode Meta Bitmaps

From the data points found in the 4.4 BSD LFS paper [SBMS93] concerning the Sprite LFS free inode location performance, it can be concluded that a free inode can be found within the first 120 inodes following the parent directory inode in almost all the cases. So a data structure is needed that fulfills the following requirements:

- fast location of a free inode in the vicinity of an other, given inode;
- little overhead (i.e.: only a minimum amount of extra disk I/O for metadata updates).

In order to meet these requirements, dtfs introduces the concept of an inode metabitmap: This is a bitmap marking blocks in the .ifile that contain only allocated blocks rather than marking individual inodes as used or free.

Since Seltzer [SBMS93] has shown that a free inode will be found an average of 94 entries away from the directory inode, this metabitmap can be used to locate a block in the .ifile with at least one free inode.[1] The dext2 implementation can then look for an unused inode within that .ifile block by using a linear scan. Altough this may seem costly at first glance, this option turns out to be not that bad on a closer view: Since this algorithm is aimed at reducing disk I/O it should outperform any other allocation scheme exclusively focussing on computational efficiency.

However, extending the metabitmap to a tree-like data structure has been taken into consideration, but given the fact that a free inode can be found within the next

[1] With an ext2-based traditional filesystem and a logical block size of 4KB, there are 32 inodes within one .ifile block.

120 inodes starting from any given directory inode, the benefits of such a data structure are questionable compared to the additional complexity.

4.3.4 The .atime File

This file contains two 4 byte entries for each inode representing the last time the file was accessed and the LFS version number of the file that is required by the cleaner. There is a third 4 byte entry in every record in the .atime file that allows to extend the access time field to eight bytes, should the need arise.

Please note that the entries in the .atime file are the only data structures in dtfs that are not aligned to an eight byte boundary since the main purpose of the .atime file is to increase I/O efficiency. Padding these entries to an eight byte boundary would cause more time spent doing additional I/O than having to do an unaligned access to an eight byte time field in the future.

4.4 More dext2 Considerations

4.4.1 Maximum Number of Inodes

Since dtfs stores inodes directly in the .ifile, the maximum file size number possible limits the maximum number of inodes in a filesystem too. Since the maximum file size is limited to 2GB in the Linux 2.0.x kernels and an ext2 indoe is 128 bytes in size, approximately 16 million inodes can be in one filesystem.

While this number seems to be comfortably large for most cases, there might be situations in which this limit is too low. However, if such a situation arises, a dext2 implementation could easily circumvent this problem because

- the size of the ifile is not important to user-space programs (with the exception of the cleaner);
- dtfs itself accesses the ifile only by using block-addressing and not by using direct file offsets so that the need to deal with offsets that do not fit within a 32 bit word does not arise; nor is a correct file size representation in the inode required, as long as the indirect block structure can be traversed.

Another issue is that the size of the ifile in bytes as stored in its inode is also limited to a four byte integral value. Again, this is not a real problem for dtfs, since dtfs only uses block addresses for accessing data in the file and does not care about the file size reported in the ifile inode.

4.4.2 Delaying Metadata Block Allocations

An implementation of dext2 can make extensive use of file holes for the metadata information that is mapped to files: After filesystem creation only a small fraction (usually only the first block) of the .ifile, the .iusage and the .atime file are actually allocated. However, this causes no problems since the semantic of reading over a hole in a file is well defined as reading an area of nullbytes.

This allows to minimize the amount of disk space being wasted for currently unneeded inodes.

4.4.3 ext2 Write Cluster Layout

Figure 4.1 shows the layout of an ext2 write cluster.: The data blocks in a dext2 write cluster are grouped together file-by-file. A file's data blocks (ordered by

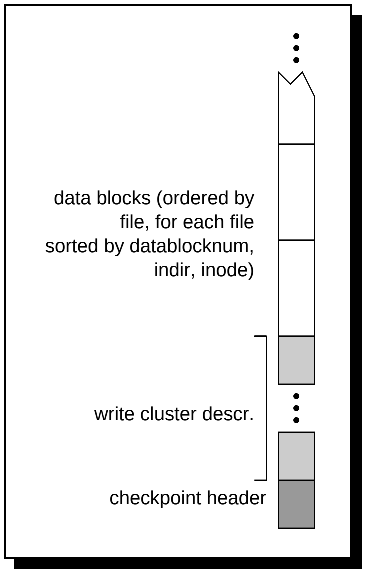


Figure 4.1: Schematic layout of an ext2 write cluster. Block numbers are increasing from top to bottom.

their blocknumber within the file to ensure good read-ahead performance) are *immediately followed* by single indirect blocks, followed by double indirect blocks and triple indirect blocks.

Should the need arise to pack blocks of the .ifile into the write cluster, then these blocks are placed last.

This block ordering scheme allows efficient read-ahead because successive data blocks are stored on successive disk block addresses. On the other hand it simplifies the implementation of dext2 since no block depends on another block that is written after it.

This also facilitates the interface between the log and a filesystem personality as shown in the next chapter.

Chapter 5

Filesystem – Core Interface

5.1 Introduction

Up to now, the tasks to be performed by the core and a filesystem personality, such as dext2, have been discussed, but the interface between these two components is still undefined. However, in order to be able to use more than one filesystem personality together with the core, a clean interface must be defined.

Similar to other kernel interfaces, such as the Virtual Filesystem Switch (VFS), it consists of a set of data structures and a number of calls that must be provided by the filesystem personality implementation.

On the other hand, the core also provides calls that can be used by a filesystem implementation for tasks like reserving disk space or triggering off a commit.

The design of this interface is influenced by the goal to keep log-specific details and actual filesystem issues apart from each other, thereby easing the task of implementing and testing both the core and the filesystem personality implementation independently.

5.2 Basic Interface Description

Besides providing calls for reserving disk blocks, the core also provides a call for filesystem personalities to trigger off a partial segment write. After that call has been made to the core, the log implementation takes over and uses callbacks that must be provided by every filesystem personality implementation in order to perform the write operation.

This is convenient because forming a partial segment by obtaining a number of write clusters from all the currently active filesystem versions and writing them out to disk, can always be done by using the same *sequence* of actions. On the other hand, the actual actions to be carried out will vary from filesystem to filesystem.

5.3 Performing a Partial Segment Write

In order to gain some insight into the way the interface between a filesystem personality and the log is actually working, an overview of how a partial segment is being written to disk is given here. A more in-depth discussion of the interface for filesystem personality implementors can be found in appendix C.

After a filesystem has triggered off a partial segment write, the log will perform the following steps on every filesystem version that is currently active:

1. Ask the filesystem about the number of dirty blocks it wishes to write out.

   This is required because the number of dirty blocks a filesystem wants to be written to disk can depend on the checkpoint type that is to be written out. This will be discussed in more detail in chapter 7.

2. Obtain the dirty blocks from the filesystem.

   In that step, the filesystem version hands over a list of dirty blocks that constitute its write cluster for the currently ongoing write. The log tries to lay out the blocks on disk in the same order as the filesystem has placed them in this list. This makes it possible for dtfs to find good disk layouts for implementing efficient read-ahead operations despite of the fact that the log is totally unaware of any filesystem-specific semantic of the blocks it gets handed over, because a clever block placement can already be chosen by the filesystem personality implementation.

3. Assign a physical address to every dirty block.

   During this step, the log actually decides where each dirty block will finally be placed on the underlying storage device.

4. Inform the filesystem version about the block placement decisions made during the previous step.

   By implementing this callback, the filesystem presonality implementation can perform an address fixups on the dirty blocks it has placed in the write cluster before they are actually written out. This will be necessary for blocks holding metadata that is about to be written out; since the block addresses the metadata information actually has to point to, has been unknown up to now.

   dext2 uses this step to update the block references in inodes and indirect blocks in a way that they represent physical block locations on the underlying device.

5. Ask the filesystem about the information it wants to have attached to its write cluster.

   As already outlined in section 3.2.2, the log can attach filesystem-version specific information to every write cluster that is in a partial segment when the partial segment is being written out. The filesystem personality implementation can use this information as a key to its metadata information. dext2, for example, uses this mechanism for storing and for retrieving the current state of the .ifile inode.

Chapter 6

A Disk Layout for dtfs

6.1 Introduction

As outlined in chapter 3, the only data structure that has to be on a fixed location is the dtfs super block. All other data structures can be located by following references to these blocks made in the dtfs super block. Therefore, there is quite a lot of freedom of choice when it comes to actually laying out dtfs information on the disk. However, a few concepts should be adhered to:

- Replicate vital metadata information.
- Lay out the on-disk data in a way that does not make segment management unnecessarily complicated.

6.2 Replicating Vital Metadata

6.2.1 Classifying Metadata

Replicating vital metadata information makes sure that the filesystem is still accessible at all even in the case of a hardware failure: If the dtfs super block were not replicated, for example, a failure making the block it resides on unreadable would make all the information on the affected dtfs partition unaccessible.

Using the information from section 3.3.2, metadata information can be classified by its importance for accessing filesystem information. Table 6.1 lists this classification using the term "essential" for filesystem data that cannot be reconstructed by any means while "important" refers to data that can be reconstructed by a very time-consuming scan of the entire medium.

Furthermore, disk medium errors tend to cluster together due to the way data is physically laid out on the device. So replicating information in successive blocks to protect it agains harddisk failures is not a good decision.

Metadata Type	Classification
dtfs Super Block	essential
filesystem Super Block	essential
Checkpoint Area	important

Table 6.1: Classification of various dtfs metadata information for accessing filesystem data. The meaning of the classifications is described in the text of this section.

6.2.2 Replication Strategies

The classification of metadata blocks according to table 6.1 has been the basis for finalizing the disk layout. The following conventions are used for that purpose:

Essential Data is placed at the beginning of the device and at the end of the device. Furthermore, essential data is replicated several times across the device.

Important Data is also placed at the beginning of the device. Furthermore, it is replicated once at the end of the device.

Finally, this replication strategy lead to the layout presented in section 6.3.

6.3 Detailed Disk Layout

Figure 6.1 shows an overview of the actual dtfs disk layout. The reserved area at the beginning of the device holds the dtfs super block. This super block is required to start exactly 1024 bytes after the beginning of the device. Furthermore, super blocks of the various filesystems and their checkpoint areas are also stored within this reserved area at the beginning of the device. The dtfs super block and the filesystems' super blocks are replicated in regular intervals. In oder to compensate for that, the respective segments are a bit smaller than the other ones.

Furthermore all the superblocks and the checkpoint areas are replicated in a reserved space at the end of the device.

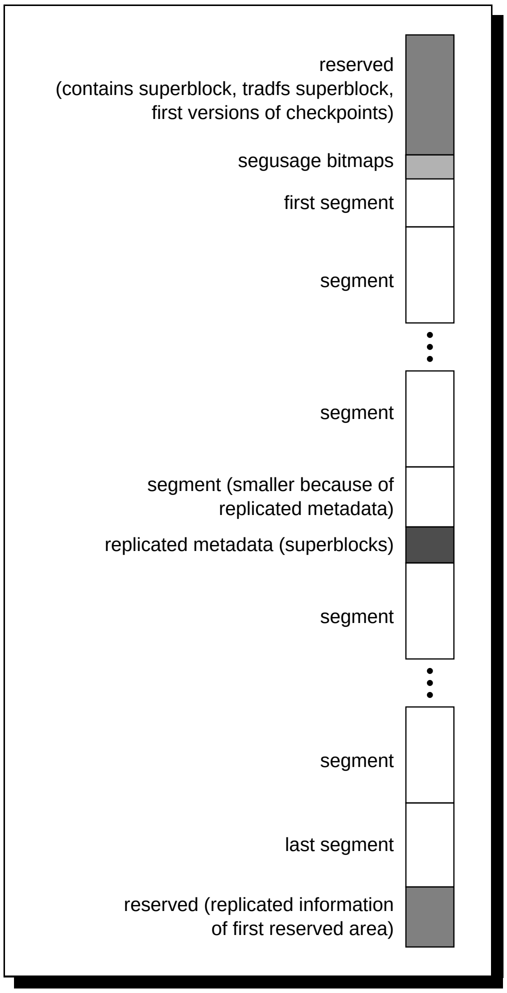


Figure 6.1: dtfs disk layout overview. Device block numbers are increasing from top to bottom.

Chapter 7

dtfs Checkpoints

7.1 The Role of Checkpoints in dtfs

Checkpoints are used to mark a consistent state of a filesystem at a given point in time. However, the problem with "classic" checkpoints as used by Sprite LFS, for example, is that they are rather expensive to create since they require a fair amount of head movement because they involve writes to disk areas that can be spread across the entire physical medium.

On the other hand, writing checkpoints out more frequently minimizes the risk of data lossage and speeds up the recovery of the filesystem after a crash.

In order to find a good tradeoff between these two conflicting goals, I've developed a number of techniques that allow to combine the benefits of infrequent checkpoint writes with those of an efficient crash recovery. However, special care must be taken when dealing with operations that require two basically independent changes to the filesystem that must still be consistent in some way, like the creation of a new file.

In order to deal with this issue, dtfs uses a rather drastic approach to that problem:

Every physical write to the device is committed by a checkpoint. Everything written after the latest checkpoint is ignored. So crash recovery is basically limited to finding the latest checkpoint of a filesystem.

The problem with checkpoints is that they might be rather costly to write degrading overall filesystem performance. Furthermore writing checkpoints often might result in increased cleaner activity since the cleaner also has to free the space occupied by checkpoint-blocks that are no longer live and other meta-data.

In order to deal with that problem dtfs employs different types of checkpoints. Every kind of checkpoint defines a consistent state of the filesystem at a certain point of time that allows to remount the filesystem after a crash. However, the time required to rebuild all the internal dtfs data structures from that checkpoint might differ, depending on its type.

7.2 Filesystem Checkpoint Types

dtfs provides the following different checkpoint types:

7.2.1 Major Checkpoint

A major checkpoint is written out every now and then depending on a tunable parameter. Writing a major checkpoint requires the following steps to be performed:

1. All dirty data and metadata blocks are written out.
2. Writing a checkpoint block holding commits step 1.
3. The current segment usage bitmap is updated.
4. The current checkpoint areas are updated.

If a filesystem is unmounted cleanly, there is always a major checkpoint present at the end of the log that describes the state of the filesystem at the time it was unmounted.

7.2.2 Minor Checkpoint

Writing out a minor checkpoint involves the steps 1 and 2 of writing out a major checkpoint. Writing out a minor checkpoint only involves writes to the tail of the log, so no (big) disk head movements are required. A minor checkpoint can be turned into a major checkpoint by updating the segment usage bitmap and all the checkpoint entries of the filesystems involved. Therefore the checkpoint type information stored in a checkpoint header does not distinguish between a minor checkpoint and a major one.

7.2.3 Data Checkpoint

dtfs introduces another light-weight checkpoint that addresses situations in which frequent syncs are encountered.

NFS writes are particularly challenging in this context because they request syncs after each file block being written. Using minor checkpoints for such a case would result in the indirect information of the affected files being re-written again and again.[1]

In order to overcome that problem, a dtfs implementation can delay the writing of metadata blocks in order to achieve better performance. For this purpose, data checkpoints are introduced in dtfs: Data checkpoints only write filesystem data blocks to disk, but not the changes in inodes and indirect blocks they might cause.

If the system crashes after a data checkpoint has been written, the missing metadata information can be reconstructed by using the block descriptions (that are part of every checkpoint block) in all all the write clusters belonging to that filesystem that have been written to disk since the last minor or major checkpoint.

7.2.4 Log Checkpoint

Since a segment is the unit of cleaning, a partial segment must not be allowed to extend beyond the end of a segment. Therefore the log may decide to add an additional checkpoint block just to end a write operation at a segment boundary. However, this kind of checkpoint does not constitute a commit, but it holds segment description information for its part of the partial segment write that is required by the cleaner. So log checkpoints can be compared to data checkpoints without any commit semantics.

In this way, log checkpoints can avoid that the data describing a block located one segment is written to another segment. Without this convention, cleaning would be overly complicated.

[1] A good example for that would be an ext2-based traditional filesystem writing out a file block that must be accessed by using a triple-indirect block. — A write to such a data block would require at least four metadata blocks for that file to be updated and might require up to three more indirect blocks to be written for the .ifile.

Checkpoint Type	Commits	Flushed	Non-Log Writes Required
Major	yes	yes	yes
Minor	yes	yes	no
Data	yes	no	no
Log	no	no	no

Table 7.1: An overview of the semantics of the various dtfs checkpoint types.

7.3 Filesystem Checkpoint Summary

Table 7.1 is a summary of the various dtfs checkpoint types. In order to understand the notation used in this table, the term *flushed checkpoint* is introduced:

A flushed checkpoint marks a point in time at which all dirty buffers (including metadata information) of all filesystems have been written to disk. This implies that a data checkpoint can never be a flushed checkpoint since the existence of a data checkpoint implies that there is some unwritten metadata present.

Chapter 8

Crucial Filesystem Operations

This chapter is an overview of some crucial filesystem operations in dtfs and shows how they are performed. The descriptions here can help in getting a deeper insight in the way dtfs works.

8.1 Recovering From A Crash

8.1.1 Fast Checkpoint Writes vs. Fast Recovery: A Tradeoff

Unfortunately all the sophisticated checkpointing algorithms outlined in chapter 7 result in additional overhead when it comes to re-constructing the filesystem after a system crash. Re-mounting a filesystem on a major checkpoint is quite straightforward, while reconstructing all the necessary data structures from a data checkpoint can be quite tedious.

In order to be able to guarantee fast crash recovery for a dtfs filesystem that has not been unmounted cleanly, a recommended maximum interval between major checkpoints should be taken into consideration when a dtfs kernel module decides which kind of checkpoint to use. This interval is called the *major checkpoint interval* (MCI) for a given dtfs filesystem.

The MCI is specified in segments. A value of 32 for the MCI means that if the last major checkpoint of a logical filesystem has been written more than 31 segments ago, the next write to that logical filesystem should be committed with a major checkpoint for that logical filesystem.

This does not necessarily mean that there must be a major checkpoint every 32 segments for a given logical filesystem. A logical filesystem can have its latest major checkpoint many segments away from the current tail of the log if there were no writes to it.

The MCI is a filesystem-settable value in order to allow adaption for fast and big storage devices that hold multiple logical filesystems and to user-specific needs for rapid crash recovery.

The recovery is also slowed down if a large amount of indirect information has to be reconstructed: In that case the respective older versions of the indirect blocks have to be read in, resulting in many read operations to blocks that might be spread across the entire physical medium. Because of this effect there is also a recommended *maximum indirect data threshold* (MIDT) specifying the maximum number of logical filesystem blocks containing indirect data that are allowed to be pending.

Recommended values for the MCI and the MIDT are specified in the dtfs super block.

8.1.2 How Crash-Recovery is Performed

Basic Algorithm

If a dtfs filesystem has been unmounted cleanly, there will always be a major checkpoint present at the end of the log allowing the efficient re-construction of all internal dtfs data structures necessary for operation. If a filesystem has not been unmounted cleanly, dtfs starts up by reading in the latest major checkpoint listed in all the checkpoint areas. This unmount checkpoint shall belong to version x of the logical filesystem A, called A_x.

From that starting point a forward scan of the log takes place. The MCI and MIDT value of a filesystem allow a good estimation of the maximum amount of time that is required to reconstruct the filesystem.

The forward scan continues until the end of the log is reached. Problems, such as a cyclic segment list, can be detected by comparing the timestamps of consecutive segments. If the timestamp value decreases from one segment to the next, a problem with cyclic segment lists has been found. However, this should only happen in the case of a dtfs implementation error. In that case, the filesystem can still be recovered by forcing the log to end before the first segment having a smaller timestamp than its predecessor.

If a checkpoint block is found that has a checkpoint for A_x, this newly found checkpoint is turned into a major checkpoint. This step also involves updating the segment usage data according to the segments that have been traversed so far. If there are some segments left at the end of the log that do not have a checkpoint, the log is truncated at the last partial segment with a committing checkpoint.

After that procedure the segment usage data areas are up to date and the current head of the log is known. Furthermore, the logical filesystem A_x is already recovered.

What still needs to be done is the recovery for all other versions of all logical filesystems (including A) on the dtfs partition. This can be achieved by applying a similar procedure as the one just described for the filesystem A_x to all the other filesystems.

A Worst-Case Scenario

If there is no pointer to a valid checkpoint in any checkpoint area to be found, only a sequential scan of the whole device can be used to re-construct the log. But this case should only happen in the case of a bug in the dtfs kernel code or because of a severe disk medium failure that affects more than one block at the beginning as well as at the end of the device since there is a four-fold redundancy as far as checkpointing information is concerned. So this case can be considered as being extremely unlikely.

8.1.3 Kernel vs. User-Space

The algorithm described in 8.1.2 does not necessarily have to be implemented in the kernel. The task of crash recovery could be implemented in an fsck utility that is run automatically on startup-time (as it is done by many Linux distributions for ext2) or before the filesystem is mounted. However, there are good reasons for implementing at least a part of that functionality in the kernel code: A situation may arise in which a filesystem has not been unmounted cleanly and to which no write access is possible at the time of the recovery. — This could be caused by a failing hardware; or maybe the user wants to avoid further writes to a known to be flaky device in order to minimize the risk of further data loss.

In that case a user-space tool cannot be used to reconstruct dtfs major checkpoints from minor ones for the dtfs kernel module because no write is possible. In such a situation it is desirable to have a dtfs kernel module that is able to reconstruct its internal data structures at least to an extent that allows read-only access to the logical filesystems on a dtfs filesystem regardless to the kind of the most recent checkpoints.

8.2 Filesystem Versioning

This section is an outline of how versioning can be implemented in dtfs by using the existing on-disk data structures. The key data structure for versioning is the dtfs checkpoint entry. An array of these data structures form a checkpoint area. Basically, checkpoint entires are pointers to checkpoint blocks holding the latest write cluster for a particular filesystem version.

8.2.1 Adding a Version

Creating a new version of a filesystem can be done by duplicating a checkpoint entry of an existing filesystem version. This filesystem version will then be the predecessor of this newly created version. Therefore it is necessary to specify a predecessor version for every new version to be created.

An implementation of the following algorithm can be used to support versioning within dtfs:

1. Find an empty checkpoint entry in the checkpoint area for the respective filesystem.[1]

2. Write a checkpoint for the old and the newly created version to the next checkpoint block.

3. Make that empty checkpoint entry the checkpoint entry for the new version to be created[2] and let it point to the checkpoint for the new version that has just been written out in step 2. Update the checkpoint entry for the predecessor version, too.

4. Write out the checkpoint area.

Step 4 serves as a commit for all these actions. If the system crashes before step 4 is completed, the filesystem remains in a state that does not have the new version.

8.2.2 Removing a Version

Removing a version is a rather trivial task:

1. Clear the respective checkpoint entry in the checkpoint area (i.e. by setting the timestamp to a reserved NO_TIMESTAMP value).

2. Write out the checkpoint area.

Reclaiming the disk space used up by a removed version is left to the cleaner.

[1] If none can be found, the maximum number of concurrent versions has already been reached and therefore no more versions can be created for this filesystem.

[2] Especially set the logical creation time to the current logical time.

8.3 Cleaning and Versioning

8.3.1 Basic Cleaning Algorithm

The task of the cleaner is to reclaim free disk space by selecting segments that contain no or only a few blocks that are still referenced. The cleaner performs a live-block test on every block in a segment in order to determine the amount of live blocks in it.

If the number of live blocks in a segment is below a certain threshold, the cleaner will re-write the remaining live blocks in the segment to the log.

After this filesystem change has been committed, the segment in question does not contain live blocks anymore and can be removed from the log thereby marking it as free.

8.3.2 A Block-Cleaning Algorithm for Multiple Versions

Minor modifications will be required to the cleaner in order to deal with versioning. The filesystem and the respective version a block belongs to is well known due to the information stored in the block description of each write cluster. Furthermore, the logical time at which the respective block has been written is known too, since it is stored in the checkpoint header of the respective checkpoint.

First of all, a block belonging to any version of *one* filesystem can never be live in *another* filesystem. So only versions of the one filesystem the block belongs to have to be taken into consideration. If the block description states that the block belongs to filesystem version x, and has been written at the logical time t_{block}, then the following live block tests have to be performed:

1. Perform the live block test for filesystem version x.
2. Test the block against every version y of the filesystem that fulfills all of the following requirements:
   (a) x is a predecessor of y;
   (b) The logical time at which y has been created from x is $\geq t_{block}$.

If any of these live tests is successful, then the block must not be cleaned. However, there is even further room for improvement taking the tree-like dependency of filesystem versions into account.

Generally speaking, it must be expected that having many different versions of a filesystem degrades cleaner performance.

8.3.3 Cleaning And Checkpointing

Since dtfs uses a variety of different checkpoint types, care must be taken not to clean away information that could be needed for metadata information reconstruction: This might be the case whenever the current checkpoint of a filesystem is a data checkpoint.

Such unwanted side-effects can simply be avoided by refusing to clean segments that have been written after the latest flushed checkpoint: The flushed checkpoint marks a state of the filesystem where all dirty buffers have been written to disk and no metadata is pending, so the need to reconstruct metadata information can only arise between the latest flushed checkpoint and the current end of the log.

If a dtfs filesystem is unmounted cleanly, there will always be a flushed checkpoint at the end of the log so that no metadata reconstruction is required when the filesystem is remounted. Furthermore, the MCI and MIDT parameters limit the amount of segments between flushed checkpoints.

Part II

The dtfs Module Implementation

Chapter 9

General Considerations

9.1 Implementation Goals

I have also made an implementation of a Linux kernel module following the design concepts outlined here. The goal of this implementation is to show that the ideas discussed in this thesis can actually be turned into a working filesystem implementation based on the current 2.0.x version of the Linux kernel.

Altough achieving good performance has not been one of the primary aims when creating the kernel module, it should be possible to get some hints on the performance impact induced by the introduction of log-structured techniques.

The goal for the implementation of this first kernel module version was to get a working version of dtfs that offers about the same features that can be found in traditional filesystems, too. So this version does not have advanced dtfs features, such as versioning, but it provides almost all the functionality found in the standard Linux ext2 filesystem, for example.

What is still missing from the kernel module implementation is an efficient inode allocation/deallocation policy. This functionality will be added during the implementation of a cleaner for dtfs.

The current dtfs code also lacks two services common to log-structured filesystems:

dtfs does not yet make use of file the cleaning hints in the .atime file that would ease the work of a cleaner.[1] Furthermore, no strategies to deal with the "unlink on close" problem are currently implemented in the dtfs code.[2]

9.2 Overview

One major goal in the implementation of dtfs has been to establish a well-defined interface between dtfs core, that is independent of the traditional filesystem used in a certain implementation of a log-structured filesystem and the "*filesystem personality*" that implements the traditional filesystem that is visible from user-space

[1] Again, this functionality will be added during the implementation of a kernel interface for the cleaner.

[2] This term refers to the fact that files are not explicitely deleted, but "unlinked": A process may unlink a file while still using it. The operating system is responsible for delaying the actual release of allocated resources until the last process accessing the resource, releases it.

Traditional filesystems do not have to address this problem, because any inconsistencies would be resolved during the filsystem check phase anyways after a crash. – A log-structured filesystem could also leave the detection of such problems to the cleaner or use another approach for preventing such problems from happening beforehand; like by linking such unlinked, but open files to a special directory until they can actually be removed.

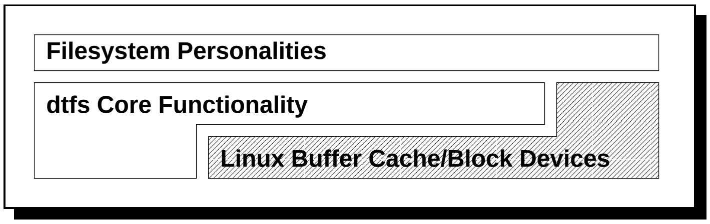


Figure 9.1: An overview of the dtfs implementation architecture.

(through the VFS kernel interface) on top of this core. This separation allows to add future filesystem personalities to a dtfs implementation quite easily. It is even possible to have more than one traditional filesystem residing on one dtfs partition that use different filesystem personalities.

Figure 9.1 gives an overview of the main dtfs implementation architecture.

The filesystem personalities can use Linux buffer cache services for reading while they write to the disk by using the means of the dtfs core. However, the filesystem itself is responsible for caching dirty blocks until they are handed over to the log for writing.

9.3 Implementation Framework

9.3.1 Assumptions Made by the Current dtfs Implementation

The current dtfs implementation makes a few assumptions about the services provided by the underlying Linux block device layer. They are required in order to be able to meet the special needs for LFS write commit semantics.

Currently, dtfs relies on the following conditions:

1. Writes to a certain disk block are either:
   (a) successfully completed as a whole;
   (b) leave the data unchanged;
   (c) return a reproducible error (i.e.: the an incomplete or failed write to a disk block will also cause a read error when trying to read the data back in).
2. When writing consecutive blocks to a block device, the underlying block device driver always writes them to the device in ascending order.

The first assumption is quite a reasonable one when dealing with block devices. On the other hand, the Linux buffer cache offers read and write operations to block devices with blocksizes that are an integral multiple of the "natural" block size of the underlying device: Most harddisks, for example, use a blocksize of 512 bytes internally, while the Linux buffer cache provides blocksizes of 1, 2, and 4KB. So it is up to the implementation of the Linux buffer cache to make sure that these assumptions hold true for such larger disk blocks, too.

According to the Linux kernel Hacker's Guide, the second assumption is consistent with the current implementation of the "elevator algorithm" used in 2.0.x kernels for optimizing block device I/O. However, the dtfs checkpoint blocks do already contain reserved fields for checksums that could be used to ensure the atomicy of a partial segment write, should the Linux write policy change.

9.3.2 Caches

In order to be able to understand some problems that I've faced during implementing dtfs, a short outline of various Linux caching mechanisms is presented here.

Basically, a filesystem will have to deal with four different caches in Linux 2.0.x kernels:

Directory Cache (dcache) This cache is responsible for holding frequently used directory entries. It is used for speeding up the mapping between user-visible file names and inode numbers.

Inode Cache Used for maintaining an in-memory representation of frequently used inodes, thereby speeding up frequent operations, such as updating the access time or obtaining information about the ownership of a file.

Buffer Cache This is a data structure representing physical blocks of a block device. In ext2/2.0.x, write operations are routed through the buffer cache. It is the primary mean of writing data to a block device using asynchronous I/O operations. (Just mark the block as dirty). As for caching, it is also helpful in speeding up access to indirect blocks that are referenced more than once.

Page Cache This is the primary cache in Linux 2.0.x kernels handling filesystem data and pages of executables that are currently running. It is highly integrated with other memory-management issues and is a page-oriented buffer for holding the actual content of a file, as far as filesystem implementations are concerned.

Chapter 10

From ext2 to dext2

10.1 Introduction

This chapter is a short overview presenting the modifications that have been made to the ext2 kernel code to turn it into an ext2 personality for dtfs. The basis for this work has been the ext2 kernel code in the 2.0.32 version of the Linux kernel with the axp-diffs applied.

Turning ext2 into dext2 involved only rather straightforward changes to most parts of the existing ext2 code. Furthermore, additional functionality needed to be implemented that is special to the needs of a log-structured filesystem and is therefore not present in the original ext2 code.

Non-trivial modifications to existing code were only needed in two areas. One of these is the handling of indirect information used for accessing data blocks in large files or directories. Adapting this code to dtfs also provided an easy way to make ext2 interoperable with the disk block allocation scheme of the log.

The other area affected by the move to dtfs is the inode handling code of ext2. Since dtfs keeps all inodes in a dedicated file that is mapped to the filesystem just like any other data, the original ext2 approach of putting inodes into reserved disk blocks that are not visible trough the standard filesystem utilities does not make sense in this context.

Apart from that, two more issues needed to be addressed in a log-structured filesystem implementation that are not present in traditional filesystems: One of them is the task of assembling dirty blocks in the filesystem layer until the filesystem personality implementation decides to flush them to disk.

The need to assemble dirty blocks and to flush them to disk in one partial segment write requires the filesystem personality to deal with another problem that is of only minor importance for traditional filesystem implementations: Maintaining a pool of dirty blocks requires a central data structure (the so-called *dirty pool*) for each filesystem mounted that keeps track of the number of dirty blocks and allows clever placement of them in the filesystem's write cluster for the next partial segment write. This data structure is modified by all write operations going to the filesystem and read by all other filesystem operations in order to locate dirty blocks that are not yet assigned to a physical location on the disk. Since more than one read or write operation can go on concurrently, the need of concurrency control arises to prevent any process from accessing this data structure while it is in an inconsistent state because of an ongoing write operation.

However, other filesystems have to face such problems too, but in a smaller scale since they have fewer common data structures to maintain.

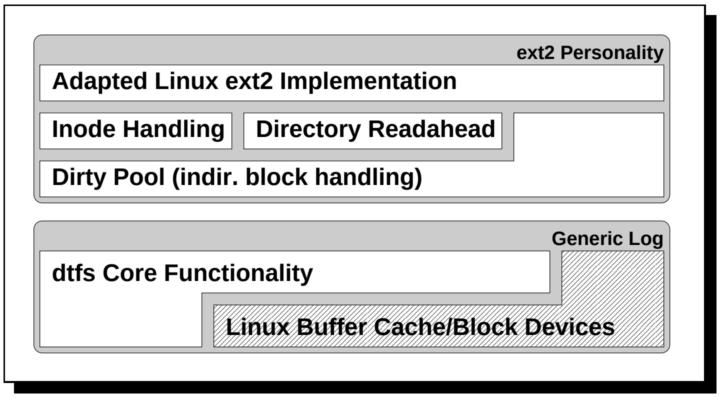


Figure 10.1: An overview of the dtfs ext2 personality implementation and the generic dtfs core.

10.2 ext2 Modifications

10.2.1 Overview

As already outlined in the previous section, the ext2 code required non-trivial modifications in two areas, namely for indirect block handling and for inode allocation. Other areas of the filesystem required mostly trivial changes. Adapting ext2 for dtfs finally resulted in an architecture as shown in figure 10.1.

10.2.2 Indirect Block Handling/Buffer Cache Interface

The ext2 Interface to the Buffer Cache

Porting ext2 to the dtfs log was eased by the fact that all indirect block handling issues are done in just one call that is used throughout the rest of the filesystem implementation to hide indirect addressing issues and to handle disk block allocations, should the need arise. The interface of this `ext2_getblk` function allows the caller to retrieve any data block of any file or directory just by specifying the inode and the logical offset of the block within the file. Furthermore, the `create` flag offered by the function prototype can be used to cause the on-disk allocation of this block if it does not yet exist within the desired file or directory.

The interface to the underlying Linux buffer cache is formed by this call together with a few other ones that directly go to the Linux buffer cache, such as for marking a buffer as dirty or up to date, or releasing a buffer again if it is not in use any longer. So providing a dtfs-aware replacement of this comparatively small set of calls allowed basic filesystem accesses to be performed.

Changes Required

In order to understand why changing this functionality when moving to dtfs was required at all, it is necessary to recall that a log-structured filesystem implementation differs in two major points from the traditional approach: A log-structured filesystem never overwrites old data even in the case of a change being applied to already existing information. Writing is always done in an append-only way to the tail of the log that is provided by the underlying generic dtfs code. Disk writes are clustered together into partial segment writes in order to both achieve better I/O efficiency and to guarantee a transaction-like semantics for disk writes.

This rather different disk layout requires fundamental changes to the ext2_getblk function mentioned earlier. Since this function is also responsible for creating new blocks within a file or directory, it must be made “log-aware”.

Besides this change, it is also important to keep in mind that it is the task of the filesystem personality to assemble dirty blocks into a write cluster for the respective filesystem version. This task is complicated by the fact that the precise on-disk location of a block to be written is unknown until the actual log write takes place requiring the need of a block address fixup at write time as already described in section C.5.

The remainder of the buffer cache interface also required changes in order to be able to implement a functional dirty pool: Dirty blocks that have not yet been placed into a partial segment need to be kept allocated until the respective partial segment has been flushed to disk. Furthermore, if the filesystem implementation changes an already existing block by marking it dirty, the block must be added to the dirty pool, if it is not already there.

10.2.3 Inode Handling

Since the way inodes are laid out on the device is entirely different for ext2 and dtfs, fundamental changes are required in the fields of inode management. Different approaches are taken for finding a free inode and for reading/writing inodes to/from disk in dtfs.

Again the design of the Linux VFS interface provided helpful in minimizing the amount of non-trivial changes required. There are only a few VFS calls that are used for obtaining an inode, flushing a modified inode back to the device and finally discarding the inode if the filesystem entity it represents is not referenced anymore.

All these calls perform their task on a standard, kernel-internal and filesystem-independent representation of an inode. The corresponding ext2-specific inode structure is attached to each of these generic representations.

The basic tasks performed by these calls are

1. reading an ext2 inode from disk;
2. building of the generic kernel-internal representation of an inode from the corresponding ext2 data structure ;
3. propagating changes to the generic inode structure back to the original ext2 data structure;
4. writing the modifications of the ext2 inodes back to disk.

Tasks 1 and 4 can easily be performed by dtfs, since inodes are kept in an ordinary file. So the standard file access mechanisms (basically provided by the modified ext2_getblk call) can also be applied for reading and writing inodes.

Tasks 2 and 3 are complicated by the fact that dtfs maintains the inode access time in a different file, the .atime file. So in order to be able to reconstruct the kernel-internal inode representation from on-disk data, both the .ifile and the .atime entry for the respective inode have to be obtained. Furthermore, some changes (like modification to the block referencing data and the last write time) can be reconstructed from the block description area of a partial segment write, while others, such as permission changes cannot. This has an effect on the decision of whether or not to write out a certain data block in the .ifile for data checkpoints: If the block contains inode changes that cannot be reconstructed from the block

filesystem implements	required for	used by VFS to implement
bmap	swap files	generic readpage
readpage		generic read, generic mmap
file read, file mmap		

Table 10.1: Various levels for read support in the Linux VFS. Of course it is possible for a filesystem implementor to mix these approaches, e.g.: a filesystem may provide a bmap call and still implement its own file read call.

description area of a data checkpoint, then the block has to be written out even for a data checkpoint.

The current implementation of the ext2 personality for dtfs overcomes this problem by comparing the data in the generic inode structure with the still unchanged information in the ext2-specific inode attached. The result of this step is a classification of the changes made to the inode in question leading to an appropriate classification with respect to the checkpoint issue.

10.2.4 The dext2 Dirty Pool

In order to assemble data that has not yet been flushed to disk and has no physical disk block assigned to it yet, the dirty pool is used. This data structure holds all dirty blocks to be placed into future write clusters. Blocks that get marked as dirty must be added to the dirty pool if they are not already there. Furthermore, the dirty pool implementation is responsible for triggering off a partial segment write when the number of blocks it has accumulated exceeds some maximum value.

The dtfs implementation of ext2_getblk also needs to pay special attention to the dirty pool because blocks that must be obtained must be looked up in the dirty pool first before any access to the actual buffer cache is done in order to ensure data consistency.

Maintaining the dirty pool data structures also causes most of the concurrency handling problems that are particular to dtfs. They will be discussed in section 10.3.

10.2.5 Directory Readahead

The ext2 sources contain code that takes care of handling read-aheads when accessing directory structures. Some parts of this code are tied rather closely to the underlying buffer cache calls. This required some changes for dext2 to make directory read-aheads aware of the existence of the dirty pool.

10.2.6 dext2 Read Accesses

In order to ease the task of a filesystem implementor, the Linux VFS layer provides several ways of implementing read and write support to plain files. They are summed up in table 10.1. The most comprehensive support for reads can be used by the filesystem implementor just by providing one simple `bmap` call to the VFS. The task of this function is to return the physical block number on the underlying device for a given data block identified by its inode and the offset of the desired data block.

Implementing this functionality is enough for a filesystem to provide full read support (including memory mapping) to files.

Unfortunately it is not possible to provide a bmap function for dtfs since

- dirty blocks do not have a physical block number assigned to them while they are in the dirty pool;
- a write to a certain block of data causes it to change its physical position on the device due to the append-only nature of log writes.

While the first item could be overcome by making a bmap function return some "logical" block number that allows the VFS to locate the block within the buffer cache even if it does not yet have a physical location on disk, the second issue cannot be overcome easily.

Fortunately, the VFS provides two more mechanisms for filesystem read support: Besides implementing specific calls for file read and mmap, it is also possible to provide a `readpage` call, that obtains one page[1] worth of data from the filesystem. The VFS provides generic implementations for read and mmap calls that can be used once a readpage function is implemented for a filesystem [BBD+97, Rub97].

However, since dtfs does not provide the bmap functionality, it is not possible to create swap files on an ext2/dtfs filesystem since this requires the filesystem to provide a bmap function.

10.3 Addressing Concurrency Issues

10.3.1 Current Approach

Due to the need to maintain a dirty buffer pool as a common data structure, dtfs must take more care of avoiding filesystem race conditions than other approaches that don't require a central data structure. Again, in order to get a stable, working version of dtfs up and running, performance has not been the main issue in designing concurrency support. So a rather restrictive approach is used in the first dtfs version that is described here.

In order to avoid any race conditions between two writers that could occur due to block reservations, the current implementation does not allow filesystem-modifying operations to go on in parallel. This is achieved by serializing filesystem writes using a semaphore.

Furthermore, every filesystem operation is strictly divided into two steps. During the first step, it is made sure that there is enough space available in the log for the filesystem modification operation. The most important part of this task is reserving a sufficiently large number of blocks in the log. An estimation of an upper bound in the number of blocks to be required is enough, since erroneously reserved blocks get freed again in the next partial segment write. Currently, the filesystem implementation is designed in a way that it detects insufficient resource allocation during the next phase and reports an error so that estimations that are too conservative are quite easy to track down.

The next phase involves modifying common filesystem data structures to represent the changes made. Since dtfs does not allow concurrent writes, only race conditions between readers and writers can occur. Such race conditions can occur when a reader has to access multiple blocks of filesystem data (because a data block referenced by indirect blocks is read in, for example) and blocks because it has to wait for data being read from the device. While the reader is blocked, a

[1] The term "page" is used memory-management wise in this context. So one page on i386 refers to 4KB of consecutive data, while the page size used in Linux/Alpha is currently 8KB, for example.

writer might modify the file so that some of the filesystem blocks are now to be found in the dirty pool that have not been there before.[2]

In order to detect such problems, each inode that has dirty blocks in the dirty pool has a version number attached to it. Every time a writer is doing another modification to the file, this version number is incremented. When a reading process starts its operation, it takes a copy of this version number.[3] After having finished the block accessing phase of the read operation, the version number is read in again from the common data structure and compared against the one that has been obtained previously. If the numbers match, no interference between a reader and a writer has taken place. If they do not match, the reader simply repeats its operation.[4]

Since such race conditions have proved to be rather unlikely to happen even during heavy I/O, this approach is considered to be sufficient for the current state of the dtfs implementation.

Unfortunately, there is another race condition that must be taken care of when the actual writing of a partial segment is being performed. The problem is related to the way the current implementation is trying to avoid duplicating buffer cache entries.

Such a duplication could happen during the address fixup phase in a partial segment write. If a dirty block that has not been assigned to any specific location on the physical device is being relocated to block x, it might well be possible that the actual content of this block is currently in the buffer cache. This could be due to some process reading raw device data, such as a disk dump utility.

In that case, a simple relocation of the original block to location x should be avoided, since this would result in two entries for the same block holding different data are present in the buffer cache. Furthermore, it is not possible to distinguish between these two buffer cache entries anymore.

Currently, dtfs handles the problem by copying the data to the buffer cache entry already representing block x and discarding the other one. Due to the way buffer cache entry allocation is currently done in dtfs, this could result in a reader process dying unexpectedly because it is still trying to access the original buffer, that is now freed.

Again, a rather simple approach has been chosen to solve this problem. The following steps are performed by dext2 when it comes to actually writing to the underlying device:

1. Disallow new read requests to the filesystem.
2. Wait for currently ongoing reads to finish.
3. Start the actual address fixup and disk I/O after the last reader has finished.
4. Re-allow new read requests to the filesystem.

This rather restrictive approach is also needed for a second case when it comes to writing out a major checkpoint: This step also involves writing out a checkpoint area for the filesystem. Unfortunately this requires to change the write block size for the underlying block device to 1KB since checkpoint area blocks are currently specified to be only 1KB in size. Unfortunately, changing the blocksize for a certain device forces the current implementation of the Linux buffer cache to discard all blocks for the affected device it currently holds.

[2]Race conditions due to SMP are a non-issue for filesystems in Linux right now since they are addressed in higher-level kernel code that forbids SMP when it comes to filesystem accesses.

[3]Inodes that are not present in the dirty pool are assumed to have a version number of zero.

[4]However, this approach does not allow to specify an upper boundary for process execution time for a given scenario. In general, this problem can be ignored when not designing a real-time system.

However, the performance impact by that is not very severe, since the buffer cache is not the primary cache for Linux 2.0.x kernels. The actual contents of a file will be found in the page cache, and not in the buffer cache. The only things found in the buffer cache are usually blocks holding filesystem metadata information. Mainly indirect blocks will be affected by that since inodes are kept in a dedicated cache, too, once they have been read in.

10.3.2 Lessons Learned From The Current Approach

After having dealt with concurrency issues in the Linux kernel, I have made a lot of experience for improving concurrency handling in future versions of dtfs. Especially the restriction of entirely forbidding filesystem reads while a partial segment is being formed and written out, is too restrictive. In order to overcome the problem, a different way of keeping track of buffer cache entry usage counters must be introduced in dtfs in order to avoid potential problems in the block duplication issue described above. Removing this restriction already provides a significant improvement since concurrent reads and writes only have to be avoided in the case of a major checkpoint write.

The next addition would be to allow more than one writer in parallel for dtfs. The need to restrict the number of concurrent writes to just one, is mainly caused by the way new buffer cache entries needed for I/O are currently requested: Since entries in the buffer cache are identified by the device and the logical block number they represent, dtfs must perform a trick in order to be able to obtain new entries for blocks that do not yet have a physical location on the device. The current implementation solves this problem by using a dedicated block counter: After every checkpoint write, the counter is re-set, so that it points to the first block past the end of the device. Whenever a new buffer cache entry is needed for a new block, this is done by requesting a cache entry for the respective device using the block number that is specified by the counter. After that, this counter is incremented by one so that the next block to be allocated will get a new, unique identifier.

Let us now assume that two concurrent writers, W_1 and W_2 both try to modify data block n for a given file. Furthermore, let us assume that block n is not yet part of the file, either because the file ends before block n or because it has a hole at that location. If the block counter has the value k initially, the following scenario is possible:

1. W_1 requests a new buffer cache entry for n.

   This causes the dext2 implementation to actually allocate a buffer cache entry for the respective device and block number k.

2. W_1 blocks and W_2 is scheduled for execution.

   Such a situation can arise because the allocation of another buffer cache entry might cause the operating system to perform some I/O in order to free up physical memory.

3. W_2 requests a new buffer cache entry for n.

   Again this causes the dext2 implementation to ask for a new buffer cache entry. This time the request will be performed using a block counter value of $k + 1$, since the block counter has been incremented at the end of step 1.

4. As a result of this scheme, an inconsistent view of the buffer cache entry actually representing block n has been established. While W_1 performs the mapping $n \to k$, W_2 uses $n \to k + 1$. So we end up with two different buffer cache entries with different contents that are assumed to represent the same block of data.

This can be avoided by making the dtfs buffer cache entry allocation routines smarter so that they recognize the fact that W_2 is actually trying to access the same block as W_1.

Since writers in the current dtfs implementation already modify the dirty pool in an atomic way (i.e.: it is made sure that the writer never blocks while modifying this data structure),[5] no inconsistencies can arise from this operation with multiple, concurrent writes.

Another issue that must be addressed, however, is a writer trying to trigger off a partial segment write. In oder to ensure data consistency, a similar approach as the one discussed for the readers-writer problem in the previous section could be used.

Furthermore, performance can also be enhanced by improving parallelism in I/O operations. However, this would require changes to the log implementation: By duplicating key log data structures and using them alternately, it would be possible to perform the actual I/O operation in a separate thread with one copy of the data structures, while the other one is already used for providing services to the filesystem personalities.

[5] Adding this to the fact that SMP is currently a non-issue in filesystem implementations and the fact that processes running in kernel-space can never be pre-empted.

Chapter 11

In-Memory Data Structures

This chapter is a brief outline of the in-memory information needed by the current implementation of the dtfs module. The design has been heavily influenced by the ability to place more than one traditional filesystem on one dtfs partition and to support multiple versions of every traditional filesystem. This requires to distinguish between three different levels of information:

1. Information particular for a block device holding a dtfs filesystem.

   This information consists of the per physical device information, such as the segment usage bitmap. (See appendix D.5 for more information.)

2. Information about a certain traditional filesystem.

   This information represents things like the location of the checkpoint areas for the filesystem or the contents of the filesystem's super block. A list of such per-traditional filesystem information chunks is maintained in the respective per-device information associated with the dtfs filesystem this traditional filesystem is residing on.

3. Information particular to a certain version of a traditional filesystem.

   Comprising of things like the physical location of the latest checkpoint for this filesystem version, a list of per-filesystem version information is maintained in the log module associated with the underlying dtfs filesystem. Furthermore, a traditional filesystem personality information can place its own, private data structures here.

Figure 11.1 shows an overview of the relations between the in-memory data structures used by dtfs.

There is one global variable, called `dtfs_filesystems` that is maintained by the dtfs kernel module implementation. It is required for the proper management of all these data structures: When a filesystem is mounted, the user is actually requesting to mount a particular version of a particular traditional filesystem residing on a particular dtfs filesystem. So an entity of type (3) according to the classification that has been given above, must be created.

The global variable is needed to find all the entries necessary that are part of the other two hierarchies when a new version of a filesystem is mounted. It is a list of all the per-device information for all currently active dtfs filesystems. When a new filesystem is to be mounted, the list pointed to by that global variable is searched for an entry representing the device the filesystem to be mounted is located on. If it cannot be found, such an entity is created and inserted into this global list.

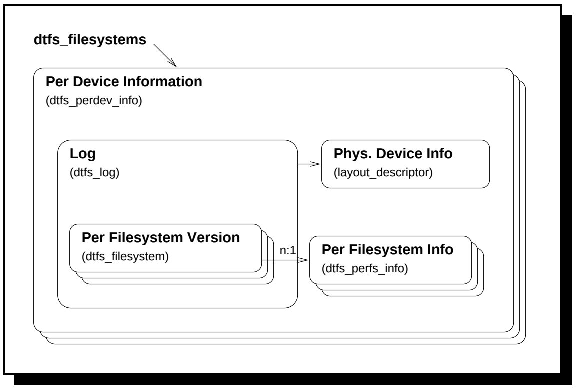


Figure 11.1: In-memory data structures of dtfs. Stacked boxes indicate lists of data structures of the same type. Arrows show additional relations required for proper dtfs operation.

After the per-device information has been located, the list of the per-filesystem information held by that particular per-device information entity is searched for an entry matching the filesystem of the new version of the traditional filesystem that is to be mounted. Again, such an entity is created and inserted into the respective list if it cannot be found.

A usage counter is maintained for the per-device as well as for the per-filesystem information structures: If no filesystem version referencing the respective entry is mounted anymore, its usage counter will drop to zero and the entity gets discarded.

Part III

Testing And Benchmarking

Chapter 12

Testing dtfs

12.1 Modularity Issues

The separation of dtfs into a filesystem independent log and a filesystem personality implementation with a well-defined interface between these two components has also helped with debugging and testing. Furthermore, the decomposition of the log and the filesystem into smaller modules as described in section D.5 also enabled log tests to be performed on a per-module basis.

12.2 Testing the Core

After a first implementation of the core has been finished, I have implemented a "dummy filesystem personality". This filesystem personality has no user-visible functionality and does not interface with the VFS, but it can be used to generate a well-known number of dirty blocks for the log to be written out. I have used this functionality to verify the basic correctness of the log implementation for various special cases, such as

- writing out a partial segment that fills exactly one segment;
- writing out a partial segment that leaves just one unused block in the current segment;
- writing out a partial segment that requires the log to insert one or more log checkpoints in order to prevent the partial segment from crossing a segment boundary:
  - Cause the log to insert exactly one log checkpoint so that only one block has to be written to the next segment.
  - Write so many blocks that more than one log checkpoint has to be inserted.
  - Make the partial segment so big that more than one block has to be placed in the next segment after a log checkpoint.

These first tests that could be done right after the implementation of the log had been completed and before any work on the ext2 personality was started. The "dummy filesystem" provided a valuable tool for a first step of testing and debugging. Assuring that the basic functionality provided by the log is actually working has also helped in finding bugs in dext2.

12.3 Testing the Filesystem Personality

12.3.1 Using a Two-Phase Approach

Implementing the filesystem personality was done in two phases. Starting out with the Linux ext2 filesystem code from kernel 2.0.33, the first goal was to get the read and write access to ordinary files working. This required dtfs to be able to handle all the data addressing issues, such as locating a given inode or finding a data block belonging to a file that is referenced using indirect blocks.

After implementing this basic mechanism, all the other vital filesystem mechanisms, such as creating a new file or altering directory structures could be implemented quite easily since they all rely on the same mechanisms of metadata handling.

12.3.2 Getting File Read/Write Support to Work

The first goal in getting file read/write support to work was to make sure that small files that only use blocks that do not have to be accessed using indirect blocks can be read in correctly. This was eased by the fact that the make filesystem utility for dtfs does already create a few files that are used by dtfs during normal operation, such as the .atime or the .iusage file.

Since the .iusage file (holding the inode meta bitmap) was not yet in use at this point of the implementation, it could be used for testing purposes. Again this eased debugging the kernel code because it was possible to focus on getting read support up and running first by using a file that has already been created by a user-space tool. As a consequence of that, read support for directly addressed blocks was already quite stable when the implementation of the write support was started. Otherwise it would have been necessary to determine whether a bug is caused either by an error in the write routines when the data was put on disk, or by a problem with the read code when the file was read back in after it has been written out. Without any further knowledge about the working state of the read/write implementations the only reliable way to find out which of the two is actually failing, would be to directly examine the data on disk with some low-level tool such as `dd`. However, this can be a very slow and rather tedious process.

The next thing to be addressed was the handling of ext2 indirect blocks. In order to be able to debug this functionality properly, two small utility programs were written that allow the writing of an arbitrary number of blocks to a file starting at an arbitrary block offset. Since the way indirect block addressing is done by ext2 is well known, using these tools allowed efficient testing for single, double, and triple indirect blocks simply by choosing appropriate block address ranges.

One problem that showed up during this stage was the necessity to deal with the Linux inode cache when address fixups were performed during a partial segment write: Since dtfs maps inode modifications to changes in the .ifile and the .atime file, fixing up addresses also requires updates to the inodes of these special files. A first version of this code only altered the inode representations in the dirty .ifile blocks that get written out: This lead to problems with inodes still being in the inode cache, since the cached information did not get updated and was still referring to the in-memory location for the dirty data blocks.

These problems could finally be settled by applying the following strategy when forming a write cluster:

1. Check all dirty inodes in the current write cluster whether they are in the Linux inode cache, too.

2. Flush every inode that has been found during step (1) ensuring that the two images of the inode, one in the Linux inode cache and the other one in the dirty .ifile block actually match.

3. When fixing up block addresses during the partial segment write, make sure that both, the .ifile and the inode cache representation of the inode data structure are updated properly.

12.3.3 Implementing Other Filesystem Functions

Directory Structure Handling

After the issues of inode location and block addressing had been solved, creating hard links was the next goal. This filesystem function allows to focus almost exclusively on directory structure handling. The only thing that is required to be done besides adding a new entry to an existing directory is to increase the link count for the inode the new hard link is pointing to, which is quite a straightforward operation once the basic inode updating support is up and running.

Again, extensive testing was done by using shellscripts that generate directories with a lot of entries in them. After these directories had been created, they were read back in using standard commands, such as ls. However, this phase of testing revealed two subtle bugs: One of them was caused by a mis-interpretation of the meaning of a Linux buffer cache state bit and the other one was caused by an erroneous adaption of the ext2 directory read-ahead code. Altough this second bug only showed up for relatively large directories that will probably not be found in real-world applications, it could be identified and fixed during these tests.

File And Directory Creation

After dtfs had been able to handle directory structures and block addressing correctly, the adaption of the remaining ext2 code was quite straightforward. However, there was one more issue that caused unexpected problems when it came to adding file and directory creation support to dtfs: The original implementation of the dtfs inode location code interfered rather badly with the Linux inode handling routines: The first implementation of this inode location code used the standard Linux `iget` function in order to obtain an inode handle for the .ifile when an inode was requested. The inode handle obtained by this call was then used with the standard indirect block handling routines written for the ext2 personality of dtfs.

Unfortunately this approach did not work quite well resulting in a deadlock whenever an inode request that was routed to the ext2 code through the VFS layer that originated in another `iget` call.

This problem was solved by "faking" an inode structure for the .atime and the .ifile files: "Faking" in this context means to set up a mockup of a standard Linux inode representation that contains just enough information for the ext2 filesystem personality indirect block handling routines, without using any functionality from the inode cache. This approach solved the problem without the need to duplicate any of the indirect block handling routines just for inode location purposes. However, this was achieved at the expense of having to maintain two representations of the .ifile and the .atime inode, since the internal representation of an inode and the ext2 one are different.

Symbolic Links, Pipes, Fifos, etc...

After the inode allocation problems were sorted out, implementing the remaining filesystem functionality turned out the be a relatively easy task, either because the ext2 code could be ported over without much ado, or because the Linux kernel already offers a lot of default functionality, as it is the case for handling device nodes, for example.

12.4 Integration Testing

Integration testing was performed by a combination of the already mentioned techniques with some more tests, such as

- Copying large directory structures to the dtfs filesystem and verifying data integrity by comparing every copied file with the original version. This was done using various means (cp, tar).
- Running several applications directly off a dtfs filesystem.
- Compiling a Linux kernel on the filesystem and using the resulting kernel image for booting the system.
- Running many different processes that were reading or writing to the dtfs filesystem concurrently.

Chapter 13

dtfs Debugging Techniques

13.1 Kernel Debugging Techniques

Handling faults in the kernel code is quite different to addressing problems with user space code. While application programs can be tested using sophisticated tools, such as a runtime debugger, these options are not easily available to the kernel programmer. Furthermore, restarting a crashed application program can be done quite fast, while recovering from a bug in a kernel module might require a reboot albeit slowing down the process of debugging.

Alessandro Rubini [Rub98] describes most of the common kernel debugging techniques available in Linux. However, they have proven to be of varying usefulness. In practice, the following two kernel space debugging techniques have turned out to be the most useful ones:

- Logging crucial system state information (such as the contents of variables of interest) by using the standard Linux syslog mechanisms.
- Obtaining a call stack using the function addresses of all the exported symbols in the module as specified in the `/proc/ksyms` file after a kernel fault. It might even be helpful to deliberately trigger off a kernel fault just to gain insight in the call stack at a certain point in the code during debugging.

13.2 Debugging Code in the Module

The current dtfs kernel module code contains quite a few debugging facilities that can be turned on and off at compile-time. Basically, the kernel debug code is used for four different purposes that are presented here.

13.2.1 Debugging Messages/Error Messages

dtfs contains code for sending information to the system logging facilities. There are several types of messages that are used for different purposes:

Informational messages are used to signal crucial events to the user that do not constitute an error. Currently, this is only done when the module is loaded or unloaded.

Error messages are an indication of a problem being detected in the dtfs code. The cause of such problems can either be a bug in the dtfs kernel module or some external error condition, such as the inability to obtain free memory being needed for dtfs operation or a read error on the device.

Debugging messages are helpful for tracking down a particular problem in the dtfs module implementation. Usually, they are only enabled for one or only a few source files that are suspected to be the cause of a problem.

Call messages indicate whenever a process enters or exits one of the VFS interface functions provided by dtfs. This is very useful for pinning down problems that are due to race conditions when a deadlock occurs due to a bug in the concurrency handling code. Examining the output generated by these call messages shows which processes were actually involved in creating a deadlock and what precise sequence of VFS calls had been causing it.

Call messages are usually disabled during normal filesystem operation, too.

13.2.2 Buffer Leak Debugging

Furthermore, a few routines have been added to the dtfs kernel module that can be used to track down any problems related to I/O buffers being allocated, but never freed. Furthermore, this code can also be used to determine, whether a read or a write request is actually going to a valid buffer that is currently allocated or to an invalid one.

The buffer leak debugging facilities of the dtfs kernel module allow efficient detection of buffer leaks by providing information about the source file and the line number within that file that are the cause of the buffer leak.

Again, this code can be turned off completely at compile time. It is disabled during normal operation since it can add a significant memory overhead to dtfs.

13.2.3 Buffer Usage Debugging

The current kernel patch for dtfs also comes with code that creates an entry in the `/proc` filesystem hierarchy that allows to watch the current state of the Linux buffer cache from user space. This functionality is currently enabled by default since it does not create any overhead unless the user actually touches the /proc filesystem entry created by that feature.

13.3 External Debugging Tools

This section lists tools that have been programmed for the sole purpose of debugging dtfs, as well as a few general-purpose ones that have been very convenient in diagnosing dtfs problems.

13.3.1 The `inspect` Utility

Inspect is the most sophisticated standalone user-space program that has been written so far for debugging dtfs. It allows the experienced user to examine all key dtfs data structures on a device interactively. So inspect can be used to browse through the on-disk data structures and verify their consistency.

By removing the necessity to deal with raw (binary) data when examining a dtfs partition, it has been very helpful in speeding up the debugging process.

However, inspect does not support watching ext2 structures, such as the ext2 directory structures or an ext2 inode within the .ifile file. But this lack of functionality has not been a major problem since having a look at ext2 data structures proved to be necessary only in a few cases since the ext2 code is already very well tested.

13.3.2 `getblk/writeblk`

These tools/shellscripts were used to write or to read a specified number of blocks to a file starting at a certain block number. Furthermore, writeblk fills every block with a different pattern making it quite easy to identify wrongly addressed blocks.

These two tools were useful when testing the indirect block handling code in dtfs.

13.3.3 `dd` and the `emacs` `hexl-mode`

The standard Unix disk-dump utility in conjunction with the hexl-mode of emacs proved to be valuable tools for the cases that required an examination of ext2-specific data structures, such as directories or inodes. The data could be obtained from the disk either by dumping the disk block of interest (that has previously been located using inspect) to a file and loading it into emacs or by using dd to pick out a certain block from the .ifile and inspecting the inode of interest by using emacs again.

Chapter 14

dtfs Performance

14.1 Overview

Finally, I have run a few benchmarks on a dtfs filesystem, both to serve as some more integrative tests, as well as to get some insights in the performance impact caused by the different disk layout of a log-structured filesystem. So the benchmarks were chosen to point out differences between an ext2 personality using the dtfs core and the original ext2 implementation. The benchmarks used are artificial benchmarks that stress one particular aspect of filesystem functionality.

The test equipment was a Linux 2.0.33 system (2.0.33 RAID patch applied) with an AMD K6-200 CPU and 256MB of RAM. However, to avoid strong cache influences, some tests were run with the system configured to use only 64MBb RAM. The harddisk used for testing was an IBM SCSI harddisk model DCAS-34330W connected to an Adaptec 2940W host adapter. Table 14.1 lists some key technical data points of the harddisk as reported by the manufacturer.

In order to get a base line for comparing dtfs performance against, the same tests were run on a dtfs and an ext2 partition. Both filesystems were created with standard filesystem parameters, when not specified otherwise for the respective filesystem test.

14.2 Raw Performance

First of all, the raw data transfer rate of the harddisk was measured by using the dd command for writing out or reading in 2047MB of data with a blocksize of 4KB directly to or from the device. The values obtained by this test are considered to

Interface	SCSI-3 (Ultra)
Capacity (formatted)	4.3GB
Data Buffer (read, look ahead buffer, write cache)	512KB
Rotational Speed	5400RPM
Seek Times (typical read)	
Average/Track to Track/Full Track	8.5ms/1.1ms/15ms
Media Transfer Rate	62.5(inner) 103.5(outer) Mbit/sec

Table 14.1: Some key characteristics of the harddisk used for benchmarking as provided by the manufacturer. What is interesting is the large variation in media transfer rate across the device that is typical for modern drives.

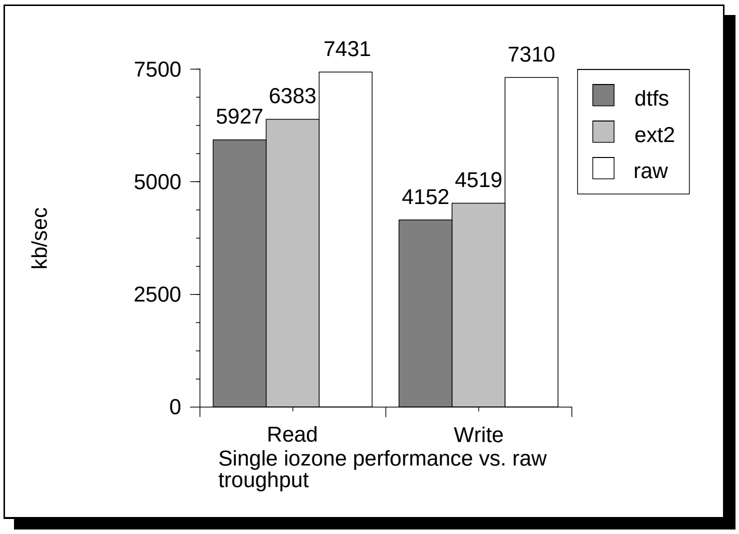


Figure 14.1: Performance as measured by iozone on ext2 and dtfs filesystem compared to raw disk performance measured using dd for reading and writing the same amount of data. While read performace is close to the maximum throughput of the device, write performance is still significantly better for raw device I/O. Larger values indicate better performance.

be reference values pointing out the maximum transfer rate that can be achieved with a harddisk of this type.

For determining the peak large file read and write performance using filesystem mechanisms, the iozone performance test was used. This benchmark writes a file with a specified size to the medium and reads it back in. iozone then reports the measured throughput for read and write operations in KB/sec.

The iozone test was run on a newly formatted partition spanning the whole drive on an otherwise idle system in multiuser mode. The file size specified for iozone was 2047MB, the same as for the dd test.

Figure 14.1 shows the results of these tests as reported by the iozone and the dd test. It is interesting to note that both filesystems maintain a performance close to the maximum throughput of the device for reading the data, while the filesystem write performance is significantly behind the maximum transfer speed for writes as obtained with dd. The good results for the read performance are probably due to the read-ahead algorithms implemented in the Linux kernel and the harddisk itself.

14.3 Filesystem Metadata Access Tests

14.3.1 Test Description

A few basic tests have been run for both dtfs and ext2 regarding metadata access speed. The test involved the following steps:

Create: Creating a directory hierarchy that is 2 levels deep. Every directory contains 20 subdirectories (resulting in 20 directories being created in the first level and a total of $20 * 20 = 400$ directories being created in the second level). Every directory in the second hierarchy level contains 100 files (resulting in a total of $400 * 100 = 40000$ files to be created). The directories were

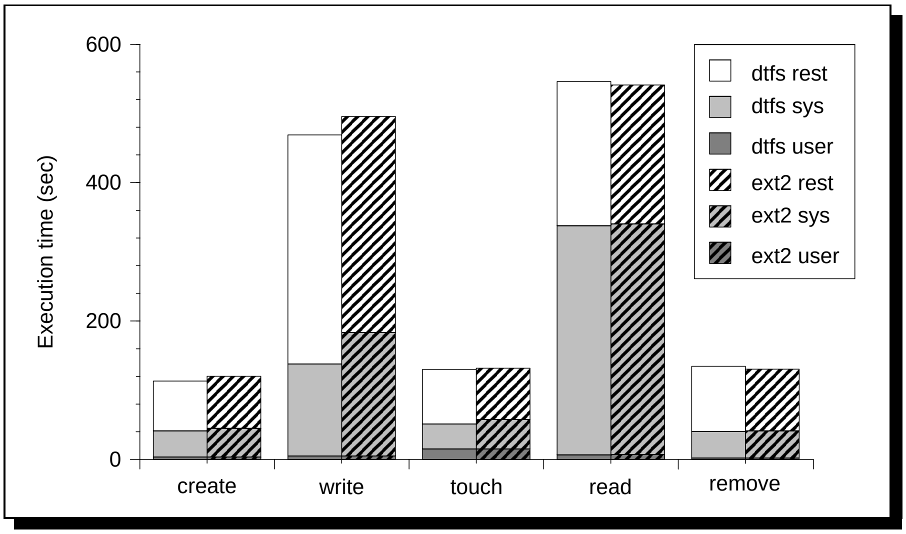


Figure 14.2: Performance figures for the various metadata access tests described in this section. Smaller values indicate better performance.

created using the `mkdir` system call, while the files were created with the `creat` system call.

Write: During this phase, one 4KB block was written to each file that had been created during the first step. Again the time required for this test was measured.

Touch: Next the time needed for accessing all the directory and inode information was measured by timing the execution of an ls command with the "-laiR" option.

Read: This test accessed every file by reading in the 4KB block of every file that has been created during the Write phase of the test.

Remove: During the final step, the directory structure was removed using "rm" with the "-fr" option.

At the beginning of every phase of this test, the system's caches were "cooled" by unmounting the test filesystem and reading in raw data from another harddisk using dd. The amount of data that was read in was equal to the physical size of the used RAM in the system (which was set to 64MB for this test.) Furthermore, to ensure that the inode cache gets flushed, too, an ls command with the "-laR" option specified was run from the root directory with the test partition still unmounted. After that, the test partition was mounted again and the test was performed.

14.3.2 Test Results

All these metadata tests taken together should allow a good estimation of the performance to be achieved when using filesystem tools that stress the filesystem's metadata handling capabilities. However, the tests with dtfs have been performed with a version that does not yet update its inode metabitmap data structure. Figure 14.2 shows the measured results for the tests described above for both dtfs and ext2. Again, the measured values for ext2 and dtfs are rather close.

It is interesting to note that both dtfs and ext2 can perform rather well on the tests that involve metadata accesses only (create, touch, remove). The good results for ext2 are probably due to the fact ext2 that lays out inodes for cylinder groups sequentially, too; thereby reaching the same performance as a log-structured filesystem.

Another thing that is quite interesting in the test data shown above is the fact that small file reads create a significant CPU load on both filesystems. As a matter of fact, the 2.0.x version of the ext2 code (that is also the basis for the dtfs ext2 personality) uses a rather simple directory structure that maps diretories to a linear list. So locating a file entry in a directory requires a sequential scan of the directory. Therefore the handling of very large directories can be a CPU-intensive task in ext2.

However, this effect can hardly be the reason for the excessive CPU usage during the write test, since

- the directory structure used in this test deliberately contains only rather small directories to avoid this effect;
- this problem would show up in all of the tests, since not only the read test requires directory scans.

In order to be able to draw any conclusions from this anomaly, further investigation would be required. A next step in pinning down this problem would involve running this test on various different systems to find out whether this behavior is a common one or peculiar to the test system.

When the excessive CPU usage during the read phase can be observed on other systems, too, the Linux kernel profiling facilities could be used to further track down the problem.

14.4 Concurrent Accesses

14.4.1 Description

Due to the special nature of write accesses for log-structured filesystems as already described in section 1.3, running more than one write operation in parallel has an impact on the way data is actually laid out on disk. A log-structured filesystem can put data into large, sequential chunks when only one writer is active. However, as the number of concurrent writers increases, these chunks get smaller since data from more than one file has to be put into the log.

The tests were run with the default dirty pool setting for dtfs of 256 blocks. Using this configuration, the ext2 personality of dtfs accumulates at most 256 disk blocks of data before it triggers off a partial segment write.

14.4.2 Parallel iozone

As a next test, a number of parallel iozone tests were run on ext2 and dtfs. iozone was used with a filesize of 256MB this time and the standard bash 2.0 `time` utility was used to obtain the total completion time for all iozone operations. Furthermore, the system was rebooted after each test. Figure 14.3 shows the benchmark results normalized to a per-process execution time. It is interesting to note that the per-process execution times for dtfs climbs rather slowly until the number of concurrently running iozone processes exceeds 6. The comparatively large increase when going from 6 to 8 processes is probably due to the early stage of concurrency support in the current dtfs implementation as already described in section 10.3. Furthermore, the values obtained for ext2 feature a slight decrease when going

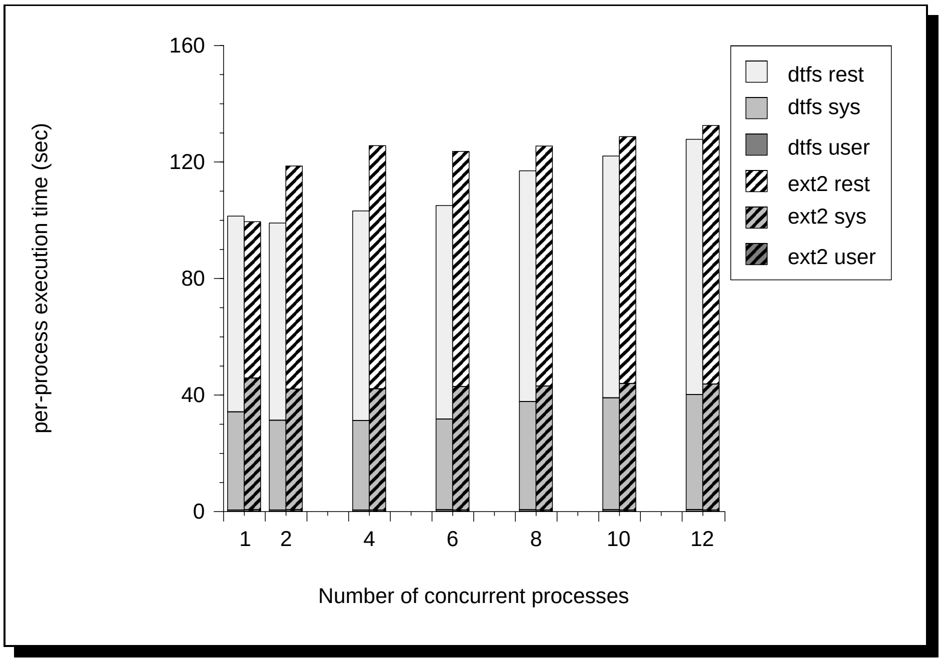


Figure 14.3: Normalized per-process total execution time for running more than one iozone benchmark in parallel. The file size used for the iozone benchmark was 256MB. Smaller values indicate better performance.

from 4 to 6 concurrently running processes. A possible explanation of this effect are differences in the actual media transfer rate of the drive used for testing in relation to the physical location of a block to be accessed. This assumption is supported by the comparatively large variation in actual media transfer rates as specified in table 14.1.

However, due to the append-only nature of dtfs log writes, the overall performance for dtfs is still slightly better than for ext2.

14.4.3 Parallel Writes — Sequential Reads

Description

As a next test, the dd utility was used with a blocksize of 4KB to write files of 100MB in size to both a dtfs and an ext2 partition. While writes were done in parallel, the various files created were then read back in sequentially. An average of the measured read values was then computed.

Again this test was used to examine the effects of different disk block layouts between ext2 and dtfs induced by the log-structured nature of dtfs. While the dtfs disk block layout should help in improving write performance, however, it can cause problems when it comes to reading the data back in because the data is now laid out in comparatively small chunks for each file.

In order to rule out any influence of the Linux file cache the test system was rebooted before the read tests.

Every test run was performed on a newly formatted partition spanning the whole disk.

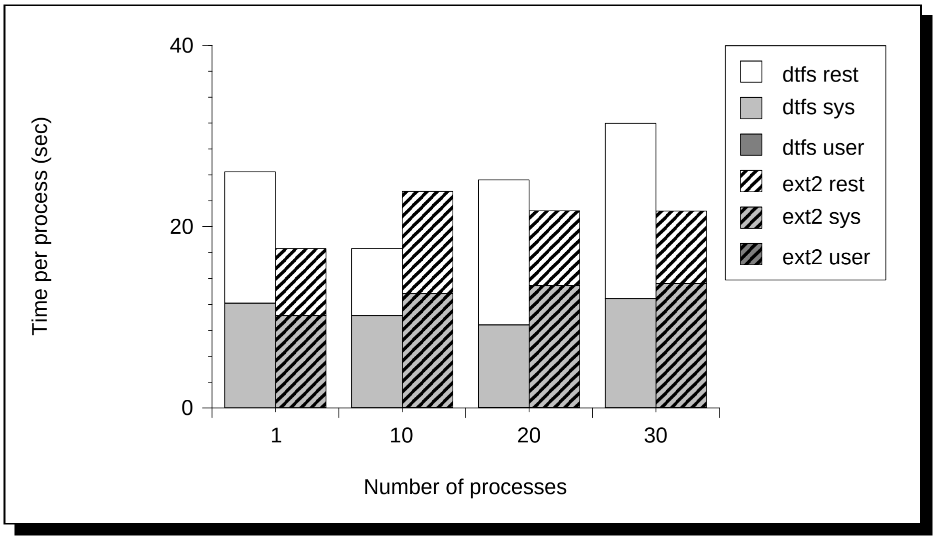


Figure 14.4: Per-process execution time for various numbers of dd processes running in parallel writing 100MB each to an ext2 and a dtfs partition. The performance anomaly for the single-process dtfs write can be attributed to variations in the actual media transfer rates of the underlying device (see text). Smaller values indicate better performance.

Parallel dd Write Tests

Figure 14.4 shows the results for the parallel write tests as described above. Again the results have been obtained using the standard bash 2.0 time utility. Once more the values were normalized to represent a per-process execution time for parallel runs.

The measurement values obtained from this test are quite interesting. First of all, it is noteworthy that the per-process execution time for running only one instance of dd on dtfs is comparable to the per-process time that can be observed when running 20 processes in parallel.

At first sight this seems to be rather counter-intuitive, but on a closer view it can be explained by having a look at the harddisk characteristics as listed in table 14.1 again: The actual media transfer rate of the underlying device varies from 62.5 to 103.5 Mbit/sec for the harddisk used in these tests. Since dtfs writes data in a strictly sequential way to the harddisk starting from very low block numbers on a newly created filesystem, it can be concluded that writing 100MB of data to the beginning of the device is probably taking place at the lowest media transfer rate. Since other write tests performed use bigger data sets, this negative effect cannot be seen since it only shows up for a comparatively small number of blocks located at the beginning of the device. Furthermore, this argument is supported by the relatively bad ratio between total execution time and the time spent waiting for I/O operations to complete ("dtfs rest") indicating a comparatively slow data transfer rate.

Furthermore, an attempt has been made to repeat the dd-write operation on dtfs without formatting the partition before the second write. This results in dtfs writing the data to a different disk location. This immediately decreased the time required to complete the I/O operation by about 25%.

ext2, using a completely different on-disk block layout does not seem to be affected by this anomaly in this test.

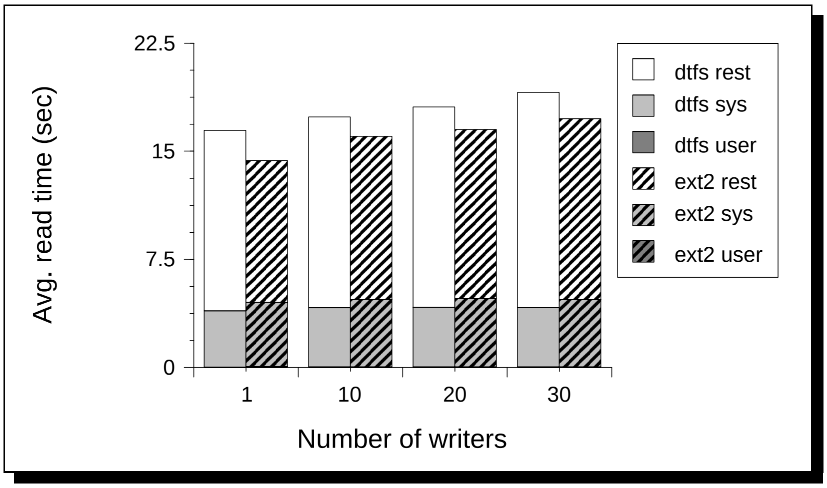


Figure 14.5: Average time required to read 100MB data files back in sequentially that have been written out in parallel. Both filesystems show a slight increase in read time for a bigger number of writers. Smaller values indicate better performance.

Furthermore, dtfs seems to have a linear increase in per-process execution time when multiple instances of the test are run. Again this is can be attributed to the early stage of concurrency support in the current implementation of dtfs.

Reading in the Data Sequentially

The next step in this test involved reading the data back in sequentially in order to determine the performance impact caused by dtfs having to lay out the data in a more inefficient way when many concurrent writers are active. Figure 14.5 shows the result of this test. The values shown in this figure are means of all the sequential file reads (so when 10 files were written concurrently, the read time for every file was measured and the result shown in figure 14.5 is the mean of all these read operations).

Both filesystems perform reasonably well on this test and show only a moderate increase in read time. This results indicates that comparatively small chunks of continuous blocks to be read in are already sufficient to obtain good performance on modern harddisk drives.

This is also illustrated by figure 14.6 that compares the filesystems' performance against raw device throughput as measured in section 14.2.

Again it can be seen that both filesystems perform reasonably well compared to raw device throughput. The comparatively large performance variation in dtfs write is due to the abnormal behavior of dtfs during this test that has already been discussed in this section.

14.5 Conclusions on Performance Measurement

14.5.1 General Observations

Recent developments in hardware technology and improvements in operating system implementations make it rather difficult to obtain reasonable filesystem performance figures.

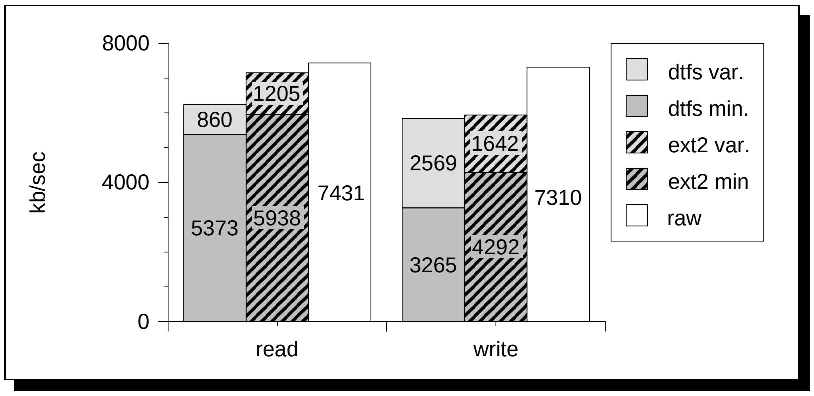


Figure 14.6: Best and worst case performance of dtfs and ext2 in the parallel dd test compared to raw device throughput as obtained in section 14.2. Bigger values indicate better performance.

Modern harddisk drives feature media transfer rates that can vary widely depending on the physical location of the disk block to be accessed. This behavior has a great influence on the actual throughput being measured when comparing filesystems that use a totally different on-disk layout of the data to be written as it is the case for dtfs and ext2. This fact is illustrated best by the fact that the write performance of dtfs is significantly slower when writing data to the first 100MB of the disk compared to the next 100MB.

When the BSD Fast Filesystem was designed, some very sophisticated assumptions about the on-disk layout of the data on the device were made. One key approach was *interleaving* the data: At the time the FFS was designed, harddisk controllers often could not keep up with the speed of the raw device: One of the worst cases that could happen was the necessity to access two successive blocks located on the same track. Since the drive's control logic was not able to immediately start another read request, the penalty of having to wait for the spinning disk drive to complete a full rotation before the second block could finally be read had to be encountered.

The solution taken by the BSD FFS designers was to choose an on-disk layout that kept logically consecutive blocks wide enough apart from each other in order to avoid this problem.

However, today these considerations have been replaced by another paradigm. Today it is considered to be a good practice to view the device as a continuous array of blocks and to map files to a small number of large block intervals. — From the measurements conducted in this chapter, it must be concluded that rather small chunks of consecutive data are enough to achieve reasonable performance with today's hardware.

On the other hand, recent storage device developments have created another challenge that is not yet addressed by today's filesystem designs. This is due to the way the data is actually laid out on a harddisk: Typically, the rotating medium is subdivided into several concentric tracks that are further divided into multiple sectors containing the actual data blocks.

Originally, the division of tracks into sectors was done in a way that all the sectors on all the tracks were of the same *angular* size. As a result of that, the same amount of data could be stored on every track.

However, this approach proved to be rather inefficient for storing data on the rotating medium since the maximum data density that could be achieved on the medium was only reached on the innermost track.

In order to overcome that problem, manufacturers are switching to dividing tracks into sectors of constant *linear* size. While this allows a more efficient exploitation of the total storage capacity, it has also created the (new) phenomenon that reading data from the innermost track is significantly slower than doing so from the outermost one.[1]

Current filesystems do not take this performance variation into account. Enhancing filesystems in a way that they try to keep data that is accessed most often within the fastest area of the device should be one focus in future filesystem developments.

Again, log-structured filesystems lend themselves very well to such modifications, since such a policy could be enforced by collecting access statistics and making them available to the cleaner that could in turn enforce a proper block placement strategy based on that data. Furthermore, the kernel module could favor the allocation of segments located in the faster areas of the disk at the expense of the ones in the slower regions. The division into slow and fast device areas could be done by performing tests during formatting or by using information provided by the manufacturer. This information could then be permanently stored in the filesystem metadata.

14.5.2 dtfs Performance Results

Altough the dtfs kernel module implementation that has been used for obtaining the measurement values is still in a fairly early stage, it is already performing reasonably well. The only severe performance penalty that has been encountered in comparison to ext2 is due to the rather restrictive approach in the handling of concurrent filesystem accesses. It is also worth mentioning that the current dtfs implementation does not cause a significant increase in CPU load caused by filesystem operations.[2]

While performance comparisons between the BSD FFS and BSD LFS have shown that the log-structured filesystem outperforms FFS when it comes to small file writes [SSB+95], this is not true for dtfs and ext2. The reason for that is that FFS performs metadata writes synchronously, while ext2 does asynchronous metadata updates.

Both ext2 and dtfs perform quite well and operate close to the maximum disk throughput for many cases.

[1] This is due to the fact that the medium rotates at a constant angular speed. Since the innermost track contains less blocks than the outermost track, fewer blocks can be read in per disk rotation when reading data located close to the center of the disk.

[2] However, this will change should checksums be used to ensure the atomicity of partial segment writes.

Chapter 15

Related Work

15.1 “Beating the I/O Bottleneck”

The idea to design a log-structured filesystem has already been proposed by John Ousterhout and Fred Douglis at the end of the 80ies. In their paper “Beating the I/O Bottleneck: A Case for Log-Structured Filesystems” [OD88] they suggested a variety of techniques that can be used to close the ever-increasing performance gap between CPU horsepower and I/O performance. However, at that time the main improvement a log-structured filesystem can offer , was considered to be increased throughput by performing all writes sequentially.

Actually, a combination of different ways to increase I/O performance were discussed in that paper, such as:

Large Read Caches

Concerning caches, Ousterhout et.al. have already pointed out that actual disk I/O will be dominated by writes in the future since writes cannot be delayed in the cache indefinitely. Furthermore, they also concluded that disk performance will finally be limited by disk seek times, especially for small files that are only about 3-4KB in size.

Battery Backed-up Caches That can Survive Operating System Crashes

Battery backed-up caches that are maintained in a way that they can survive operating system crashes were suggested as a remedy for increasing write times. Obviosly this proposal has also influenced the design of the RIO filecache [CNC+96].

Cache Logging

By using this term, Ousterhout and Douglis referred to a representation of current operating system file caches on a non-volatile medium, such as a harddisk. Performance improvements can be expected from this approach because writes do not go through a filesystem layer. Furthermore, writing is done in large, sequential chunks.

Cache logging could probably become a new field of interest in the next future: While the original design of a cache log was targetted towards addressing write speed vs. persistent storage issues, it could be used today to overcome problems with large file servers when they have to go down for maintenance:

Since large RAM file caches are volatile, the file server starts up with “cold” caches after the maintenance downtime. As RAM sizes in the range of 1GB and more are not uncommon even today, it would be interesting to explore whether dumping the file caches to a non-volatile medium when the system goes down and reloading their contents when the system is brought up again, would be beneficial.

A Log-Structured Filesystem

In this paper Ousterhout et.al. were also discussing the viability of a log-structured filesystem. Furthermore, several techniques for reclaiming unused disk space and for implementing an append-only log on a standard block-device were discussed. Even more advanced features that a log-structured filesystem can provide, such as versioning and fast crash recovery have already been proposed.

15.2 Other Log-Structured Filesystem Projects

15.2.1 BSD LFS and Sprite LFS

The design of dtfs is heavily influenced by two other log-structured filesystem projects, namely the Sprite LFS [RO90, Ros92] and the 4.4BSD LFS [SBMS93]. Sprite LFS is probably the first implementation of a log-structured filesystem. It was done as a Ph.D. thesis by Mendel Rosenblum. The advisor for this Ph.D. thesis was John Ousterhout, the author of the paper presented in the previous section.

The design of 4.4BSD LFS is also based on the work done for the Sprite LFS implementation, so all three systems (Sprite, BSD and dtfs) share some key characteristics, but use different solutions to address specific problems, such as atomicy of certain filesystem operations.

The remainder of this section presents a short overview of various topics of interest for a log-structured filesystem design and discusses how the problem is being addressed in the three approaches.

Managing Disk Space

All three designs divide the underlying block device into chunks of equal size for putting log information into. Writes are done append-only to the end of the log. All systems use mechanisms similar to the "partial segment write" that has already been outlined for dtfs for actually putting data in the log and for ensuring the all-or-nothing semantics of log writes.

Furthermore, every system maintains a segment usage summary in order to find free segments fast. However, the way in which this information is stored on disk, is different for every system: Sprite LFS writes the segment usage information to the log when writing out a checkpoint, while BSD LFS maps the segment usage bitmap to a user-visible file.

Unfortunately, both approaches are not suitable for dtfs. Writing the segment usage bitmap to the log every time a checkpoint is being written out is not an acceptable behavior for dtfs, since dtfs uses a different approach to checkpointing and filesystem consistency that requires more lightweight checkpoints. Furthermore, placing the segment usage information into a regular file is also not an option for dtfs since it allows more than one filesystem within the log. So the question arises to which filesystem the segment usage information should be mapped to. — Since there is no such a thing like a "master filesystem", the segment usage information would have to be mapped to all filesystems in the log. However, this would add a great deal of complexity to filesystem personality implementations since they would then have to be aware of each other.[1]

[1]One key issue for mapping the segment usage information to a user-visible file was to make this

Therefore, dtfs places the segment usage bitmap in a separate area that is not part of the continuous log. This data structure is duplicated and changes to it go to these copies alternately. This should prevent a complete loss of segment usage summary information in the case of a write error.

Accessing Inodes

All three LFS implementations are facing the problem of locating the current version of an inode on disk, given its inode number. While traditional filesystems can directly compute the on-disk location of an inode from its inode number, the position of inodes is not fixed in a log-structured filesystem.

Sprite LFS maintains a separate data structure, called *inode map*, for performing the translation from inode numbers to physical disk locations. The inode map also holds the current access time of the file and the hints for the cleaner. This helps speeding up inode access times.

BSD LFS places the inode map into a user-visible file, the ifile, that also holds the segment usage information.

dtfs uses another approach for accessing inodes: They are directly placed into the ifile. So the ifile contains the inodes themselves and not just mapping information that is pointing to them. By that, the problem of finding the physical location of an inode from its inode number is translated to the problem of locating a given block of data belonging to a certain file.

dtfs also maintains a second file, the .atime file: It holds the file access time and the cleaner hints, while the .ifile stores the actual inode information.

Since indirect blocks are also mapped into the "block address space" of the file they belong to in dext2, there are no blocks in the log that are not part of any file. This also simplifies the task of implementing a cleaner and performing a live block test.

Finding Free Inodes

All three systems use different approaches for locating free inodes. While Sprite LFS uses a sequential scan of the inode map to find a free inode and chooses random locations for starting this scan when a new directory is to be created while BSD LFS remains a linked list of free inodes in the inode map.

dtfs uses a different approach that is targetted towards minimizing the amount of I/O operations required for getting a new inode. Another file is maintained by dtfs that is called .iusage. Every bit in this file represents one *block* (rather than one *inode*) of the ifile. The corresponding bit is set when all the inodes in the respective block are already allocated. (The motivation and the implementation for this design have already been discussed in depth in section 4.3.3.)

Data Consistency

All three filesystems are facing the challenge of keeping filesystem data consistent. While traditional filesystems heavily rely on a filesystem check utility to achieve data consistency after a crash, log-structured filesystems try to avoid long checking times.

One area of problems arises from directory operations that require more than one change to on-disk data structures. An example of such an operation would be the creation of a file requiring the following steps to be performed:

information accessible to a user-space cleaner. Recent versions of the dtfs implementation achieve this goal by mapping the segment usage bitmap to a /proc dir entry.

1. Create an appropriate inode structure.
2. Mark the inode as used in the filesystem metadata information.
3. Modify the directory information containing the link to the inode created in step 1.

These modifications must either be performed as a whole or none of them is to be done. All other combinations result in an inconsistent filesystem. So when a filesystem is not unmounted cleanly, some measures must be taken in order to ensure the atomicy of this sequence of actions.

All three filesystems use different approaches to guarantee such atomicy.

Sprite LFS uses a *directory operation log* for this purpose. When a filesystem operation is being performed that must be atomic, but consists of more than one single filesystem change (such as the file creation example outlined above), Sprite LFS performs the necessary filesystem modification operations and commits them by writing a directory operation log entry.

When a filesystem is being reconstructed, the filesystem check utility starts reading in the log from the last known checkpoint representing a consistent filesystem check. A roll-forward of the log is then being performed. Together with the commit information in the directory operation log, a consistent filesystem state can be ensured.

BSD LFS uses a different approach to solve that problem, called *segment batching*. This technique makes use of the fact that every write operation is done to an append-only log and that writing out a partial segment always serves as a commit. However, there are cases in which not all dirty blocks can be placed into one partial segment because they do not fit into the current segment anymore. In this case, filesystem operation sequences that must be done as a whole or discarded all together, could be placed in different segments, so that one segment write does not serve as a commit for the overall operation anymore.

When such a situation arises, the affected segment is marked with a set of flags that indicate that a directory operation started in the segment is still incomplete, or that the segment contains the continuation of another directory operation that has started in a previous partial segment.

dtfs uses a solution for this problem that is similar to segment batching, but tries to simplify things even more. The dtfs log implementation hides segment boundary issues from filesystem personalities so that they can actually write partial segments of arbitrary size. The log will transparently prevent partial segment writes from crossing a segment boundary by inserting log checkpoints that contain a block description, but do not have any commit semantics.

De-coupling the writing of partial segments from the actual mapping of data to disk also greatly simplifies crash recovery: The check utility just has to find the latest checkpoint in the log that is not a log checkpoint. From that checkpoint onwards, only indirect block and the segment usage bitmap have to be updated according to the information in the remainder of the log until its end is reached.

So dtfs avoids inconsistencies by ensuring that on-disk filesystem data can never be in an inconsistent state at a commit in the first place.

15.2.2 Other Log-Structured Filesystem Issues

The Spiralog Filesystem

There is another approach that has been taken by the designers of the Spiralog filesystem [JL96, WBW96] for Digital's OpenVMS operating system. This design tries to combine log-structured techniques with an attempt to get rid of block-aligned file boundaries. Spiralog views all the filesystem information as a large tree. The system tries to hold frequently referenced parts of the tree starting from the root node in memory so that no disk accesses are required in order to retrieve this information.

Furthermore, sophisticated tree-balancing algorithms are used to achieve good performance. In order to avoid that a few large files have a negative effect on overall filesystem performance, these tree-layout algorithms take special care of them.

The Spiralog approach adds a lot of complexity to the log-structured filesystem approach. It seems questionable whether the benefits gained by this design really compensate for the added complexity.

Cleaning Policies

Since a log-structured filesystem requires a cleaner in order to reclaim unused disk space, minimizing the overall performance impact caused by cleaning is an issue. Running a cleaner in parallel to normal filesystem activities makes cleaning compete for disk bandwith with other user processes. Therefore it is advisable to do cleaning only when the I/O system is idle otherwise. However, by applying simple heuristics based on recent disk usage patterns, the user-visible overhead of cleaning activities can be kept extremely small for many cases [BHS95]. However, cleaning overhead can very well become an issue for certain environments, such as transaction processing [SSB+95].

15.3 Other Filesystem Developments

15.3.1 Metadata Logging — Journaling Filesystems

Altough journaling and log-structured filesystems are often used as synonyms, they are actually two different concepts. Journaling (or metadata logging) is another way to guarantee fast crash recovery by overcoming the need for time-consuming filesystem metadata consistency checks.

Journaling adds a record of metadata changes to a traditional filesystem. This can simply be done by writing all metadata changes to a dedicated disk file synchronously. When the filesystem is not unmounted cleanly, the information stored in this journal can be used to speed up the task of making metadata information consistent again.

The journal can be cleared whenever the on-disk state of the filesystem data structures are known to be in a consistent state, i.e.: after a sync or when the filesystem gets unmounted cleanly. This prevents the journal from growing without bounds.

Journaling is of particular interest when a filesystem has to cope with small writes and frequent sync requests, as this is the case for an NFS server, for example. Log-structured filesystems such as BSD LFS and Sprite LFS have to encounter a severe overhead in that case. This is due to the fact that writing out a file in small chunks with frequent syncs forces them to write out the file's metadata information to the log again and again because of the append-only nature of log writes. The problem is aggrevated by the fact that log-structured filsystems usually try to perform disk writes in large, consecutive chunks. This is not possible when frequent sync requests are encountered.

Using metadata logging can be helpful here because only the data block and a record to the journal reflecting the metadata changes required need to be commited to disk when a sync is encountered. This can help to improve NFS write performance as it was demonstrated in the Calaveras filesystem project [VGT95].

However, by using data checkpoints, dtfs can achieve the same benefits as the metadata logging approach in the Calaveras filesystem. Furthermore, since dtfs uses a write cluster's description in much the same way the Calaveras metadata log is used as far as data checkpoints are concerned, writing should even be faster, since updating a separate metadata log can still require large disk head movements, while dtfs writes everything to the log sequentially.

Many commercial Unix vendors have also added journaling abilities to their filesystems during the last few years, mainly to improve crash recovery times. An exmple for that is the "journaled filesystem" in IBM's AIX operating system. It uses metadata logging techniques in order to maintain "structural consistency" [IBM93].

While journaling overcomes the problem of excessive filesystem check times, it does not provide any of the more advanced capabilities of a log-structured filesystem, such as versioning or clean integration with a logical volume manager that dynamically resizes filesystem partitions. However, one big advantage of the journaling approach is that it can be added to existing filesystem implementations quite easily [VGT95].

15.3.2 The RIO File Cache

The RIO file cache [CNC+96] takes quite a different approach to the challenge of making filesystem changes persistent. The basic idea behind the RIO file cache is to use the memory-management-unit (MMU) in today's modern CPUs to protect a dedicated area of RAM against operating system errors: Access to this memory area is read-only most of the time; the MMU allows write access only while data is actually being written to the file cache. By using this strategy, the contents of the write cache is even protected against operating system bugs.

After a system crash, the OS simply re-attaches the data structures found in the RIO cache thereby restoring the cache contents as they have been just before the crash.

The persistence of the information stored in the RIO cache can be assured by using NVRAM or an UPS to protect it against blackouts.

The RIO research project has shown that the same degree of reliability can be reached with RIO compared to the traditional approach of considering data to be persistent when it has actually been placed on a non-volatile storage device.

However, some limited hardware support is required in order to be able to implement the RIO file cache: While all contemporary computer architectures feature an MMU, not every hardware is able to preserve RAM contents over a reboot.[2]

Combining the abilities of the RIO file cache with the features of a log-structured filesystem would probably be an excellent platform for handling NFS writes that need to be made persistent before the successful completion of the write is reported to the client [Sun89].

15.3.3 The CODA Filesystem

A very exciting project on implementing a distributed filesystem protocol is currently going on at the Carnegie Mellon University[3] [Bra98]. CODA tries to solve

[2] Just consider a PC doing a memory test after a hard reset. The a memory test will destroy the data in the RIO file cache.

[3] The Linux Journal article [Bra98] provides an excellent overview of CODA and is also available online amongst other CODA papers quoted here from http://www.coda.cs.cmu.edu/.

many problems of today's networked data-sharing approaches. Being the successor of the Andrew Filesystem, the CODA development team tries to incorporate the experiences made at CMU with the Andrew filesystem into their new design. What sets CODA apart from other network filesystem solutions is its ability to support disconnected operation for mobile clients or for clients connected only by weak network links that go down frequently.

The main issues the CODA project tries to address are:

- seamless integration of mobile clients [KS92];
- failure resilience by introducing transparent server replication, and handling of failing network links without interrupting service;
- improved performance: Even on slow links by providing client-side persistent caching and write-back caching;
- improved security by Kerberos-like authentication and the introduction of access control lists (ACL);
- semi-automatic conflict resolution when re-integrating disconnected clients.

Chapter 16

Conclusions

Log-structured filesystems allow to add qualitative enhancements to filesystems that cannot be incorporated into traditional approaches easily. Introducing database-like semantics to disk writes opens up a lot of entirely new ways of improving both filesystem availability (by versioning) and reliability (due to the append-only write semantics).

Furthermore, dtfs simplifies crash recovery and achieves better efficiency for certain filesystem operations, such as NFS writes, by introducing more lightweight checkpoints.

The separation of dtfs into a generic core providing the basic logging functionality and a filesystem specific part allows the implementation of different filesystem personalities using this logging core.

The first dtfs implementation that I have presented in this thesis uses a modified version of the Linux ext2 filesystem as its filesystem personaltiy of choice. This first implementation has shown that the design principles outlined here can actually be turned into a working filesystem implementation that performs approximately on par with the existing ext2 filesystem implementation.

But the "order of magnitude" improvement in I/O throughput mentioned in earlier papers [Ros92] in comparison to traditional approaches cannot be achived, since modern filesystem implementations, such as ext2, already use a fair amount of the maximum I/O bandwidth.

While working on dtfs, I have received some encouraging feedback from Linux users that would greatly appreciate the availability of a filesystem that allows them to define snapshots, just as it can be done with dtfs versioning.

However, a lot of work remains to be done. The filesystem implementation presented in this thesis is a good foundation for future enhancements, but it still lacks some functionality, like a cleaner and a proper fsck utility (or any other way to increase the level of a checkpoint to a major one). In order to implement versioning, a way must be found to allow mounting more than one filesystem for a device; something that cannot be done cleanly with the Linux 2.0.x kernels.

Recent kernel developments, such as the device filesystem and the upcoming logical volume management for Linux will provide additional features that can be exploited by a full-featured implementation of dtfs. Furthermore, basing the filesystem personality on the existing ext2 code will allow to add enhancements that are currently under development for this filesystem, such as btree directory structures for faster access, to dtfs in the near future.

Appendix A

Glossary

checkpoint area The checkpoint area consists of (currently) two disk blocks, one close to the beginning and another one close to the end of the medium. These two blocks should hold identical information. These blocks contain pointers to the current major checkpoints of all filesystems and versions. There are two checkpoint areas in a dtfs filesystem that are used alternately so that there is still a consistent checkpoint area left should the writing to the current checkpoint area fail.

checkpoint block A checkpoint block is a disk block that commits a partial segment write. The checkpoint may actually be bigger than one logical disk block. In that case, writing the checkpoint block holding the checkpoint header serves as a commit. A checkpoint block holds a filesystem description entry for each write cluster in the partial segment it commits.

clone A logical copy of a logical filesystem made at a certain point in time. From that point in time onwards, changes can be applied to any of the copy of the filesystem without being noticeable in any other copy of the filesystem.

commit level This term refers to the kind of commit that is chosen for ending a certain partial segment write. While a data checkpoint is faster to write than a minor checkpoint, it is harder to re-construct the filesystem state when the latest checkpoint has been a data checkpoint. The same holds true for minor checkpoints and major checkpoints. This leads to a classification of checkpoints according to their "commit level". A data checkpoint has a smaller commit level than a minor checkpoint, while a minor checkpoint has a smaller commit level than a major checkpoint.

core A term used to refer to the filesystem-independent part of dtfs. Its main functionality is to provide the abstraction of an append-only log to a filesystem personality implementation.

dext2 This acronym stands for "dtfs ext2" and refers to a version of ext2 that as been turned into a filesystem personality for dtfs.

dirty pool An in-memory data structure for a filesystem version holding dirty blocks that are not yet committed to disk and have no fixed disk location assigned to them. They are assembled to a write cluster to put in the next partial segment write in the dirty pool.

dtfs superblock The super block containing all information necessary for the non-filesystem specific part of dtfs. The dtfs super block also contains pointers to the traditional filesystems' super blocks of all the filesystems that can be found within the respective dtfs filesystem. The dtfs super block is replicated several times throughout the whole disk.

entity An incarnation of a certain data structure is called an entity of that data structure.

ext2 The "second extended filesystem", the current standard filesystem for Linux.

filesystem personality Since the dtfs core is only a generic layer to support a log-structured filesystem, it is necessary to implement high-level filesystem services on top of this generic layer. Such an implementation of filesystem services is called a filesystem personality implementation It will most likely be derived from a non-log-structured filesystem, such as ext2.

flushed checkpoint A checkpoint at which all dirty data of all filesystems has been written out to disk so that no dirty blocks (including blocks with indirect information) are pending.

indirect block Indirect blocks are used by ext2 to reference data blocks in large files. Since only the first twelve data blocks can be accessed from an ext2 inode directly, references to other data blocks are done by a lookup in an indirect block. Indirect blocks can also contain pointers to indirect blocks, too.

log checkpoint A checkpoint that does not have any commit semantics that is inserted by the log to split up one partial segment into smaller pieces in order to avoid writes crossing a segment boundary.

logical blocksize ext2 perceives a device as an array of linearly addressable blocks. The size of these blocks is defined at filesystem creation time and not necessarily identical to the "native" block size of the underlying device. Currently logical blocks sizes of 1, 2, and 4 KB are supported. (In contrast, a typical "native" block size would be 512 bytes for a modern harddisk.) Accesses to the filesystem are performed in logical block-sized chunks that are aligned to their natural boundaries (i.e.: if a logical block size of 4 KB is chosen, I/O requests of 4 KB — or a multiple of it — that are aligned to 4 KB boundaries will be used).

logical filesystem A dtfs partition can hold more than one filesystem. "Logical filesystem" is the term for a filesystem residing on a dtfs partition. Each logical filesystem on a dtfs partition as its own super block and may be implemented by using different traditional filesystems.

logical time The logical time is the way in which dtfs events are timestamped. Every dtfs filesystem has its own logical time. The logical time can be viewed as being generated by a clock that is set to 1 on filesystem creation time and is incremented by one every time a checkpoint header is written out. This approach guarantees that the logical time is steadily increasing in a filesystem's lifetime. The logical time is represented as an eight byte integral value.

MCI value The MCI value is a settable parameter for a dtfs partition. The MCI value is a recommended upper limit for the number of segments that can be written before another major checkpoint is written out. (MCI = Major Checkpoint Interval)

MIDT value The MIDT is a settable parameter for a dtfs partition. It is a recommended upper limit for the number of logical filesystem blocks containing indirect information that are not written to disk, but whose contents can be reconstructed from following the segment description information of a number of data checkpoints. So this value determines the maximum amount of time and memory required for reconstructing metadata information after a dtfs partition has not been unmounted cleanly.

module A part of the dtfs kernel module implementation. A module consists of a data structure and a bunch of functions modifying entities of this data structure. Modules have a well-defined interface to the rest of the implementation and serve as the unit of testing. (MIDT = Maximum Indirect Data Threshold)

natural time A point in time represented according to the usual Unix time format. It is derived from the system's real-time clock. Because of that fact, the natural time cannot be used to timestamp dtfs-specific events since the system clock might be set back by the system administrator resulting in non-monotonically increasing time values.

partial segment write Term for an atomic dtfs write operation. A partial segment consists of one or more write clusters being written to the storage device. A partial segment write is always committed by a checkpoint block immediately following the partial segment. Furthermore, a specific version of a filesystem can have at most one write cluster in a partial segment.

segment A segment is a fraction of the disk that can be used by the log. A segment can either be clear, locked, or marked dirty. A segment is the unit of disk space allocation and the unit of cleaning.

snapshot A read-only version of a logical filesystem created at a certain point in time representing the state of the filesystem at that very point in time.

turn The term turn refers to a complete traversal of the log's state graph as depicted in figure D.2 on page 96 starting from the IDLE state until the log enters the IDLE state again. In a turn filesystems first start to request blocks from the log until one filesystem decides to request the creation of a partial segment write. When the log has committed the partial segment write all the dirty blocks the filesystems wanted to have placed into the partial segment have been handed over to the underlying Linux block device code and the turn is over.

traditional filesystem super block The super block of one traditional filesystem that resides on a dtfs filesystem.

versioning Versioning is the ability of dtfs to support more than one state of a filesystem: Every version of a filesystem has its specific mount point and checkpoint. A version of a filesystem may be read-only (snapshot) or for reading and writing (clone).

write cluster A set of disk blocks belonging to a specific version of a logical filesystem that are written out together.

Appendix B

Implementation Conventions

B.1 Splitting The Implementation Into Modules

In order to get a first working version of dtfs as fast as possible, performance has not been a main issue in the current dtfs implementation. The main focus has been on decomposing the complex functionality required to implement a log-structured filesystem into a subset of small and manageable modules with well-defined interfaces.

Every module consists of one data structure and a set of functions modifying the state of an entity of that data structure. These functions constitute the actual services provided by that module for the rest of the system. They are supposed to transform the data structure they work on from one consistent state to another consistent state.

The specification of well-defined interfaces between the various components of the implementation makes the code more maintainable. Furthermore, the module is also the unit of testing. Again this is possible because every module serves a well-defined purpose so that it is quite easy to identify a reasonable set of test cases. Furthermore, every module communicates with other parts of the implementation only by means of their well-defined interfaces thus avoiding any side effects. The avoidance of side effects should make the system composable and prevent the introduction of new errors when putting the (already tested) various independent modules together.

B.2 Coding Guidelines

In order to make the code more understandable, some basic coding conventions have been introduced that emphasize the modular concept even more:

1. Every module is associated with a prefix

   This should make the code easier to understand and prevent confusion about which module is actually affected by a certain function call: The name of every function defined in a module starts with a prefix that is unique for the respective module. The prefix should end with an underscore. All the functions in the bitmap handling module, for example, start with the prefix `bmap_`.

2. The creation and deletion of data structures is done by dedicated procedures of the respective modules that follow a naming convention.

   Actually, there are two ways of creating a new entity of a data structure :

(a) Small data structures are embedded within other data structures, so it is not necessary to allocate or to free the memory they use when creating/freeing them. The functions initializing such a data structure should always be named `<prefix>create`, and take a pointer to the memory area containing the entity to be initialized as first parameter. The return value for these functions should be char and indicate the success/failure of the initialization. The entity should be released by a call to a void – function named `<prefix>dispose`.

Again the bitmap handling implementation serves as an example: Bitmaps are initialized by a call to `bmap_create` and deleted by a call to `bmap_dispose`.

(b) Larger data structures are created by calls to functions named `<prefix> init` and freed by a call to `<prefix>done`. The init function is expected to return a pointer to the initialized data structure or a NULL pointer in case of any failure. Again, the `done` function should take only one parameter (whenever possible), namely a pointer to the data structure to be freed.

3. Functions should perform sanity checking on their arguments (at least in the debug/test stage).

   This should help locating implementation errors in the testing phase that would otherwise show up far away from the actual faulty code causing a problem in the first place. Furthermore, there is also a good reason in having at least some of these tests in the code even in the production version of a dtfs kernel module. — These tests could catch a flaw in the implementation and prevent at least a kernel panic resulting in a system crash to happen in the case of a severe error.

4. Every function in the module interface should be accompanied with a comment outlining the semantics of the parameters the function takes and of the return value of that function. Furthermore, its purpose should be outlined, too.

   Again this guideline should assist in making the implementation more maintainable.

5. Adhere to the Linux coding styleguides as outlined in the CodingStyle file in the Documentation subdirectory of every Linux kernel source code archive.[1]

[1] Usually this can be found under `/usr/src/linux/Documentation/CodingStyle`.

Appendix C

Filesystem Personality Interface

C.1 Filesystem Personality — Log Interface Summary

The interface between a filesystem personality and the dtfs log consists of three elements:

1. Two common data structures between the log and the filesystem implementation.
2. A set of calls provided by the log.
3. A set of callbacks that must be provided by a filesystem implementation that are used by the log.

This section discusses these three interfaces and shows how the functionality provided by them can be used to implement a log-structured filesystem personality.

C.2 A Remark About The Current Implementation

As already mentioned, dtfs supports different filesystem personalities. A filesystem personality is an implementation of a traditional filesystem that uses the dtfs core described in appendix D. Currently, only an ext2 personality is implemented, but there is a clean interface between the dtfs core and the filesystem personality, so it should be quite straightforward to add support for other traditional filesystems to dtfs in the future.

The interface between the filesystem personalities and the dtfs core is currently hardwired for supporting the ext2 personality only. However, adding support for other filesystem personalities would just require the introduction of callback function structures such as the ones that can be found in the Linux VFS Layer.

C.3 Common Data Structures

The log and the filesystem implementation communicate by using two common data structures. The filesystem personality implementation is of course allowed to extend these structures by adding more data fields to them. This can roughly be compared to the task of adding new data structures to a class in an object-oriented programming language by deriving a new class from a base class.

The following data structures must be provided:

1. The `dtfs_buffer` – struct

   This per-block data structure holds information describing a dirty block in the filesystem personality. It is required for obtaining the block description and for assigning a physical address to the block when it is about to be written out to disk.

2. The `dtfs_filesystem` – struct

   This structure is used to uniquely identify a certain version of a traditional filesystem. The filesystem personality must use this structure for registering and unregistering with the log.

C.4 Log Call Interface

The log provides a few calls for the filesystem personality implementation that allow it to do things like reserving disk space or triggering off a write. This interface consists of just six functions, introduced in this section.

C.4.1 Registering with the log

Before a filesystem is allowed to use any services provided by the log, it must register itself with the log. This is normally done when a filesystem is mounted.

C.4.2 Unregistering with the log

After a filesystem has been unmounted, it must unregister itself with the log indicating, that it will no longer need any of the services provided by the log. Unregistering is important since the log itself can only be discarded if no more filesystems are using it.

C.4.3 Reserving space in the log

Before a traditional filesystem implementation is allowed to acknowledge the successful completion of a filesystem operation, it is required to reserve a sufficient number of blocks in the log. This makes sure that a filesystem operation can actually be committed to disk later and does not fail because the log has run out of free disk space.

Blocks can actually be reserved from two pools; one for "normal" data that is written out on every checkpoint, and one for "meta" data that doesn't get written out when only a data checkpoint is being created.

These block reservations are valid until the next write that flushes the respective type of blocks. So free space reservations for "normal" blocks are only valid until the next write cluster for the respective filesystem is being written out, while metablock reservations stay valid until a minor or a major checkpoint is being written out.

The amount of space reserved in the log does not necessarily be the exact amount of disk blocks required for the filesystem operation. It is sufficient if an estimation that is just an upper boundary for the actual disk space required is used.

This comes handy for filesystem personalities that want to perform some kind of data compression, for example. Such filesystem personalities cannot determine the precise amount of disk space required before the actual write takes place. Furthermore this can also be used to ease the implementation of other filesystem personalities.

C.4.4 Triggering Off A Commit

When a filesystem personality implementation has accumulated a sufficiently large number of dirty blocks, it might decide that it is time to put all these dirty blocks in a write cluster and ask the log to start writing out a partial segment by using this call.

C.4.5 Obtaining/Setting The Current Checkpoint Entry

These two functions are only required if the filesystem wants to update the current number of unused inodes in the filesystem when a major checkpoint is being written.

C.5 Filesystem Personality Callbacks

A filesystem implementation is required to provide a few callbacks for the log. This is actually the largest part of the interface between the log and a filesystem personality implementation. Basically, the log will use these functions to obtain the information to put in the filesystem's write cluster after a partial segment write has been started by any filesystem registered with the log.

Table C.2 lists these callbacks.

C.6 Putting It All Together: Forming A Write Cluster

As already mentioned, the forming of a partial segment can be triggered off by any registered filesystem . The log will then perform the following steps using the filesystem personalities' callbacks:

1. Determination of the actual commit level to be used.
2. For each filesystem registered:
   (a) Determine the number of dirty blocks to be written out.
   (b) Obtain the dirty blocks from the filesystem.
   (c) Assign physical block addresses to every dirty block.
   (d) Build the filesystem descriptor for the filesystem's write cluster by obtaining the ifile inode of the filesystem (if necessary).
3. Write out the partial segment in one call to the underlying Linux block device interface.
4. Closing down (again done for each filesystem registered):
   (a) Release additional data structures.
   (b) Inform the filesystem about its new "latest checkpoint" location.
   (c) Tell the filesystem that the current turn has ended.

Function	Purpose
active_this_turn	True if the filesystem has dirty buffers it wants to be written out
suggest_commitlevel	The filesystem may request a (higher) commit level from the log.
set_commitlevel	The log sets the commit level for the current checkpoint.
get_ifile_inode_size	The size of the ifile inode rounded to an 8 byte boundary. (0 if there is no ifile inode to be written.)
get_ifile_inode	Pointer to the data holding the ifile inode (or NULL if no ifile inode is to be written).
num_dirty_blocks	Number of dirty blocks for current commit level.
get_dirty_blocks	Hands over a list of buffer_heads containing a list of all the dirty blocks to be placed in the write cluster.
resolve_blockaddresses	The log has assigned physical addresses to all dirty blocks. The filesystem can now perform block address resolving.
release_queued_buffers	The dirty buffers handed over to the log have been written out. The filesystem can release the blocks and the corresponding dtfs_buffer structures.
update_checkpoint_pos	The log informs the filesystem about the new position of its current checkpoint.
end_this_turn	Writing out the dirty buffers is finished.
do_sync	Asks the filesystem to start performing a sync.

Table C.2: Callback functions that must be implemented by a dtfs filesystem personality

Please note that the actual algorithm that is performed by the log in order to write out dirty blocks is a bit more complicated since it might be necessary to split one partial segment into multiple ones and to insert log checkpoints because the dirty blocks do not fit in the currently active segment of the log anymore. However, these details are not relevant when designing a filesystem personality since these issues are handled by the log in a transparent way.

Determination Of The Actual Commit Level To Be Used

The first step that is performed by the log is to negotiate the commit level to be used for the requested partial segment. However, the filesystem that has triggered off the commit has already suggested a commit level. The log now starts to ask all traditional filesystems associated with it for a commit level suggestion. This is done by using the `suggest_commitlevel` call provided by the filesystem personality implementation.

A traditional filesystem is only allowed to increase the severity of a commit level, but it must not decrease it. So `suggest_commitlevel` must always return either the same commit level presented to it in the call to `suggest_commitlevel` or a higher one. After all filesystems have agreed on the commit level to be used, the log informs all registered traditional filesystems about the outcome of this negotiation by using the `set_commitlevel` call. This is necessary because the number of dirty blocks a filesystem wants to hand over to the log for writing to the partial segment that is currently in progress might depend on the commit level used.

This is especially true in the case of a data checkpoint being created: No filesystem metadata are written to disk in a data checkpoint, so if a filesystem holds a total of n dirty blocks, out of which k are holding filesystem metadata information, the filesystem will only have $n - k$ dirty blocks to write out.

Per-Filesystem Operations

Determining the number of dirty blocks to be written out. The next thing the log will request from the traditional filesystem is the number of dirty blocks the filesystem wants to have written to the underlying block device in this turn. This is simply done by using the `num_blocks_dirty` call of the respective traditional filesystem. Furthermore, the log will also ask for the size of the ifile inode the traditional filesystem wants to have written out to disk in this turn. If the filesystem does not want any ifile inode information to be placed in its filesystem descriptor for the current turn, it should simply return 0 for any `get_ifile_inode_size` call and a NULL pointer when asked for the ifile inode data by the `get_ifile_inode` callback.

Please note that the size of the memory area pointed to by the pointer returned by `get_ifile_inode` must match the size specified by `get_ifile_inode_size`. The ifile inode size must also be a multiple of eight. Furthermore, a filesystem is not allowed to return a NULL pointer on the `get_ifile_inode` call when it has returned a non-zero value for `get_ifile_inode_size` in the same turn beforehand.

Obtaining dirty blocks from the filesystem. After having determined the number of dirty blocks the filesystem wants to have placed in its write cluster for the current turn, the log gathers these blocks by calling the `get_ditry_blocks` function. This function hands over a pointer to a linked list of `dtfs_buffer` structures constituting the write cluster of the respective filesystem for the current turn.

Once again a filesystem is required to hand over the exact number of dirty blocks that have been returned by a previous call to `num_blocks_dirty` in the current turn.

The filesystem personality can influence the layout of dirty blocks in its current write cluster by arranging them properly within the linked list. The log will try to put the entire write cluster for the filesystem into consecutive physical disk blocks. The dirty block that starts the list, is placed at the lowest block address. The rest of the dirty blocks are then placed in the write cluster in ascending order.

Please note that the log cannot always guarantee that a write cluster is actually committed to disk in one single sequence of disk blocks due to segment geometry constraints.[1] Nevertheless, using a reasonable block placement strategy in a filesystem personality will be beneficial since it allows the efficient implementation of read-ahead strategies.

Assigning physical block addresses to dirty blocks. After the log has decided how to layout the dirty blocks handed over by a traditional filesystem on the un-

[1] This happens every time when a write cluster is split into two separate ones by the log because it would otherwise extend across a segment boundary.

derlying device, it will repeatedly call the function `resolve_blockaddresses`. When this function is called, the `buffer_head` members of all the `dtfs_buffer` structures handed over in the call have already been modified to contain the physical address of the block on the disk. The filesystem personality is expected to perform all its block address translations (from logical addresses to physical addresses) when this function is called.

The filesystem must also fix up the ifile inode data structure if it wants its ifile inode to be written out in this turn. This must be done in the latest call to this function for every turn. The fact that a call is the last one issued in a turn, is indicated by a dedicated flag that is part of the interface of this function.

Obtaining the ifile inode Data After that the dtfs core implementation will use the `get_ifile_inode` call to obtain the data to be written to this turn's filesystem description of this version of the filesystem as it's current ifile inode.

Closing Down

Releasing additional data structures A call to `release_queued_buffers` indicates that the `dtfs_buffer` structures are no longer needed by the log and can be freed. The filesystem must also unlock the `buffer_head` structures associated to the `dtfs_buffers` by calling the Linux kernel function `brelse` on them.

Setting the latest checkpoint. Every version of every filesystem has an entry in its respective checkpoint area. Among other things, this entry also holds a pointer to the checkpoint block holding the latest checkpoint for this filesystem version. Of course, this location must be updated when a new checkpoint for a filesystem has been written out. The filesystem is informed of the new location by a call to `update_checkpoint_pos`.

Ending the current turn. The only thing left to do is to tell the filesystem that the current turn has now ended. This is done by calling `end_this_turn`. The filesystem is expected to retrieve its checkpoint entry, modify it (if required) and hand the modified checkpoint entry back by calling `perfs_update_checkpoint_entry`. Furthermore the filesystem implementation might also do some per-turn cleanup here if this is required.

Appendix D

The dtfs Core

D.1 An Overview Of The dtfs Core

The dtfs core is responsible for providing all services that are not particular for a certain traditional filesystem. From the point of view of a filesystem personality this core realizes an infinite log on top of a block device for writing dirty blocks to it. The dtfs core does all the internal housekeeping required for proper operation, such as

- monitoring free disk space ;

  This involves checking whether a block reservation request issued by a traditional filesystem can actually be fulfilled in order to make sure that there will be a sufficient number of free blocks available when the filesystem requests to commit these blocks later.

- keeping track of allocated/free segments on the underlying device;
- assigning physical block addresses to dirty filesystem blocks as they get written to the log;
- updating the checkpoint areas for the various filesystems after they have changed;
- updating the segment usage bitmaps on disk;
- creating checkpoints.

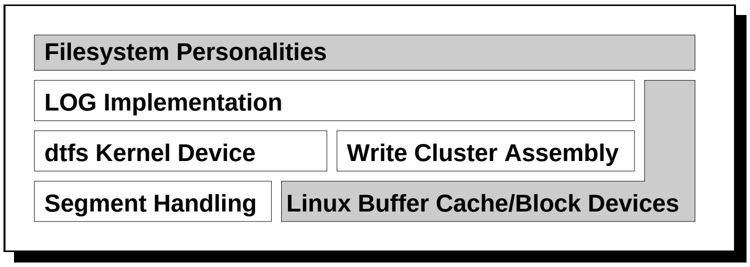
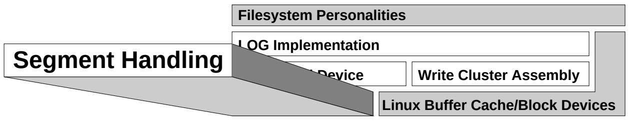


Figure D.1: The various modules of the dtfs core implementation and how they interface each other. (Grayed boxes represent layers that are not part of the dtfs core, but constitute the "world interface" for it.)

In order to be able to accomplish these tasks, the implementation of the dtfs core is split into several modules with a well-defined interface between them. Figure D.1 depicts all the modules of the core implementation and their interfaces. The functionality of these various modules will be discussed briefly in the remaining sections of this chapter.

D.2 Segment Bitmap Handling Routines



Description

This module provides a generic bitmap handling facility. It is used for managing an in-memory representation of the current segment usage bitmap. However, it contains some functions that are specially needed by the log to make good segment allocation decisions.

Services Provided

The following functionality is exported by this module:

- creating/Disposing a bitmap entity for a bitmap with a well-defined size;
- assembling a bitmap from various chunks that are handed over to a bitmap module one after each other;
- testing whether a certain entry in the bitmap is set;
- setting/Unsetting a certain entry in the bitmap;
- returning a pointer to the bitmap data and information about the bitmap's length (needed by the log to write out the bitmap);
- locating an unset bit in the vicinity of an other bit.

The last item is particularly important for the efficient implementation of a log-structured filesystem. In order to achieve good performance, it is desirable that subsequent segments in the log are mapped to disk areas close to each other in order to avoid large head movements when writing out the data or when reading it back in.

When reserving a new segment, the log asks for a free segment in the vicinity of the last segment it has obtained. It is then up to the implementation of the segment bitmap handling routines to find a suitable free segment.

Implementation Notes

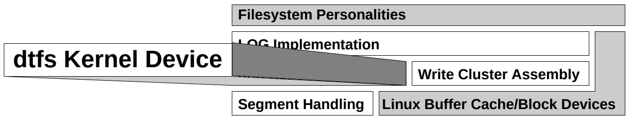
The current implementation of the bitmap handler uses a continuous array of bytes for its internal representation of the segment usage bitmap. However, according to the Linux Kernel Hacker's Guide, some version of the Linux kernel do not allow to allocate a continuous memory area that is bigger than approx. 120KB. This would limit the number of segments that can be on a dtfs partition to about 983000.

Assuming that the filesystem has been created with mkdtfs's default geometry settings where a segment is half a megabyte in size, this would limit the maximum size of a dtfs filesystem to approximately 480GB.

This limitation could be overcome by altering the dtfs segment bitmap handler implementation and by changing the way the dtfs device kernel implementation writes the bitmap back to disk.[1]

Please note that this limit is *not* imposed by dtfs's on-disk data structures, but solely by the current implementation of the dtfs kernel module.

D.3 Kernel Device Code



Description

The kernel device module is based on the device-specific parts of the implementation of dtfslib,[2] so it has already been extensively used and tested before the implementation of the kernel module has been started. It takes care of all dtfs-specific metadata structures that are not mapped into the log, like the segment usage bitmaps, the checkpoint areas and dtfs super blocks that also hold the various dtfs filesystem descriptors for the traditional filesystems.

Services Provided

The services provided by the kernel device module can be separated in four different categories:

Block I/O

- obtaining the logical block size;
- reading/writing random logical blocks to/from the underlying device;
- reading/writing of 1KB blocks to/from the device.

Logical Time Services

As already outlined in the design part, dtfs maintains a "logical time" that is represented by a 64 bit integer. It is incremented every time a checkpoint (and be it only a log checkpoint...) is written. This "logical clock" is integrated in the kernel device services module of the dtfs kernel. It provides the following functionality:

- obtain the current value of the logical clock;
- increment the logical clock by one tick (done by the log implementation).

[1] Of course, another way to get around that limit would be to use bigger segments.

[2] `dtfslib` is a collection of code that is shared by several dtfs utilities running in user-space, such as creating a dtfs filesystem or debugging a dtfs file system.

Segment/Free Space-Related Services

This involves the following functions:

1. Getting information about free blocks.
   - obtaining the total number of blocks that can be used for storing data on them that are currently in unused segments.
2. Obtaining segment geometry information.

   This includes the following services that allow to maintain various layout information about a particular segment on the underlying device:
   - getting the logical block address at which a certain segment starts;
   - getting the length in blocks of a certain segment;
   - getting the number of the segment a certain block is in.
3. Segment allocation information.

   This functionality is used by the log to obtain information about the next segment to be allocated and to perform the actual allocation. This involves calls for the following purposes:
   - allocating a new segment;
   - getting the number of the next segment that is about to be allocated (needed to build the doubly-linked list that is formed by the headers of all the segments that are currently in the log);
   - getting the number of the segment that was allocated last (also needed for the doubly linked list of segment headers).

Commit/Metadata-Related Services

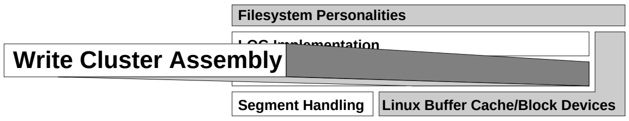
These capabilities are required for obtaining all the necessary information needed for mounting a traditional filesystem. It allows the implementation of various kinds of checkpoints in the log. The following functionality is provided:

- getting the contents of a `dtfs_filesystem_descriptor` for a certain traditional filesystem;
- writing back the contents of a `dtfs_filesystem_descriptor` after it has been modified by the traditional filesystem;
- Committing the segment usage bitmaps and the modified dtfs super block(s) to disk.

Implementation Notes

Writing to the underlying device does not happen synchronously. It is made sure that the write operation is actually triggered off when a write is requested, but the write functions do not wait for the write to complete before they return. So sync semantics have to be ensured by higher level mechanisms.

D.4 Write Cluster Assembly



Description

The write cluster assembly takes dirty filesystem blocks and their block description as they are handed over from the log and groups them into write clusters. These write clusters are then placed into a partial segment. This is done in one write operation to the underlying device. However, the block assembling code does not check whether the partial segment does actually fit into the remaining free space of the segment that is currently active. This has to be ensured by the log. The write cluster assembly is thus tied to the log very closely.

Services Provided

The blockassembly provides the following calls:

- starting/ending of a write cluster;

  The log is required to announce the start of a new write cluster every time it starts processing the dirty blocks of a new traditional filesystem.

- queueing of the ifile inode of a filesystem;

  A call to that function indicates that the traditional filesystem that is currently active wants its ifile inode (that will be placed in the respective filesystem entry) to be written out to disk, too.

- queuing of the next n dirty blocks of the currently active filesystem;

- actually performing the partial segment write;

  After all dirty blocks of all filesystems have been queued, the log will ask the block assembly to actually perform the disk I/O that writes the information that is currently queued by the write cluster assembly to disk. The log may also request a log checkpoint to be written whenever the currently active segment has run out of free space by a call to this function.

- *pausing* a partial segment write.

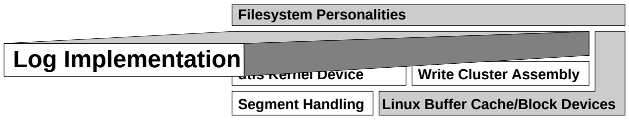
  A partial segment write is paused by the log whenever the free space in the current segment has been exhausted. Such a pausing causes the block assembly to do some internal cleanups after a log checkpoint has been written. These cleanups are required in order to make sure that no dirty block is written out twice.

Implementation Notes

As already mentioned, the write cluster assembly is tied to the log implementation very closely. It relies on the log to correctly calculate the amount of metadata blocks needed for a partial segment. However, it is possible for the write cluster assembly to perform some sanity checks on that when the log requests the actual write to be performed.

The main reason for the existence of the blockassembly was the desire to offload some complexity in the functionality the log has to perform to another module in order to make the log more manageable.

D.5 Log Implementation



Description

The actual log implementation itself has a rather minimalistic interface. Instead of providing a variety of services in the log interface, a filesystem must be *associated* with a log in order to be able to perform disk writes. It can then reserve blocks from the log and ask the log to write all its dirty buffers out to disk later. The log in turn will make callbacks to the filesystem using the filesystem interface. The filesystem interface provided by a filesystem personality to the log is described in section C.5 on page 86.

Services Provided

The log provides the following functionality:

- registering/unregistering a filesystem;
- reserving blocks on the device;

  The filesystem can reserve blocks from two different pools: One is for filesystem meta blocks and the other one is for normal filesystem data blocks. The reason for providing two pools is that there are different kinds of checkpoints available in dtfs: A data checkpoint results in all dirty filesystem data blocks being written out, but the metadata blocks will not be written. This has already been discussed in section 7.2 on page 32.

  Furthermore, a filesystem may reserve more blocks than it actually needs. This is convenient in the case of a filesystem personality for which the exact number of blocks required cannot be exactly determined before an actual write takes place. (This might be the case for a filesystem personality that wants to perform some kind of data compression before committing its dirty blocks to the log, for example.) — The log automatically discards all allocated, but later unused blocks when a commit takes place.[3]

- requesting a commit from the log.

  A filesystem may request a commit from the log it is associated with. The log will then perform the necessary steps by issuing callbacks to all the filesystems that are associated with it and assemble all the dirty blocks to a partial segment.

[3] Of course the respective pool from which the blocks have been allocated, is taken into account: No block reservations from the metadata pool are discarded after a data checkpoint, since a data checkpoint does not involve committing filesystem metadata blocks .

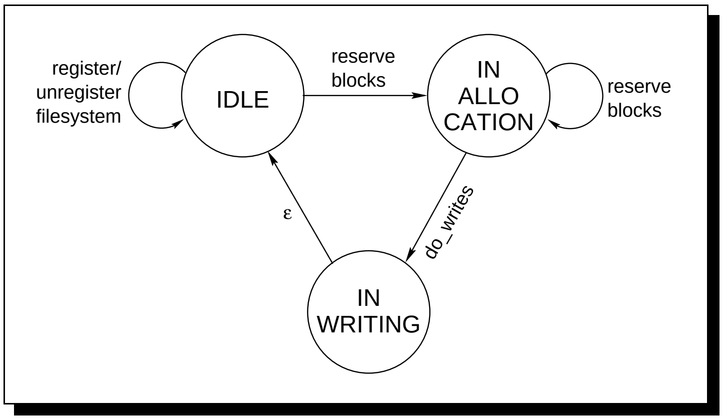


Figure D.2: The three states of a dtfs log and all the valid state transitions.

Implementation Notes

Currently the log does not check whether a filesystem is erroneously trying to write more blocks to it than it has reserved before. The only thing that is checked is that the total number of dirty blocks in a partial segment does not exceed the total number of blocks reserved.

dtfs Log States

A log in a dtfs implementation can be in three different states. The actions that are allowed to be applied to the log depend on the state the log is currently in. Figure D.2 shows a state diagram for a dtfs log. Only state transitions that are shown in this figure are allowed. An attempt to issue a function call in a state in which it is not allowed (like trying to unregister a filesystem from the log when the log is not in the idle state) will be rejected by the log. Furthermore, an error message will be signalled to klogd, the standard Linux kernel error reporting facility.

The transition from the IN_WRITING state back to the IDLE state is not triggered off by any external function called by a registered filesystem, but takes place automatically when the log has finished writing the dirty blocks to the device. However, this transition is still important because the log will tell all its registered filesystems to discard their dirty blocks (since they are now handled by standard Linux kernel mechanisms, just like any dirty block being written to a block device) when the transition takes place.

Again the introduction of log states help to catch erroneous calls by a filesystem personality to the log.

Free Space Management

General Considerations As already mentioned in section D.5, the log is also responsible for ensuring that there is a sufficient number of blocks available for writing all dirty blocks of all filesystems to the underlying device should this be requested. This is important because it must be possible to decide whether a write can actually be committed to disk at the time the application program issues the write request. So it is not an acceptable behavior if the log discovers that there is an over-commitment of free space on the device after the write call issued by the application has already returned successfully.

As already stated, filesystems can reserve blocks from two different pools. Reservations are valid for the turn they are made in only. After a turn is completed, all reservations are discarded with the exception of blocks reserved from the metadata pool if a data checkpoint has been written.

The current dtfs implementation uses a rather simple and very conservative approach to avoid over-commitment situations. This conservative approach might lead to the effect that a write operation that could actually be processed will be rejected in the case of the filesystem running very low on free space. However, measurements in [Ros92] have shown that the performance of a log-structured filesystem starts decreasing when the filesystem reaches a state of 80% full, so this situation is rather unlikely to happen in a typical application scenario.However, it is considered to be an acceptable behavior contrary to allowing over-commitment.

The most important figure for commitment policy is the number of available blocks that are located in currently unused segments.[4]

Block Reservations Every traditional filesystem that wants to write blocks to the log is required to reserve a sufficient number of blocks before the write is actually performed. If the log encounters a request that would result in a possible over-commitment, the log rejects the block reservation. The filesystem personality is then required to deny the write request.

In judging whether a write request might result in an over-commitment, the log uses the following *allocation condition:*

$$\mathit{free\_blocks} >= \mathit{blocks\_allocated} + \mathit{blocks\_requested} + ulim(\mathit{log\_overhead}) + \mathit{segsize}$$

In this equation *free_blocks* is the number of blocks stored in currently unused segments (not counting the segment header). *blocks_allocated* holds the total amount of all blocks that have been reserved so far since the last commit. *blocks_requested* is the number of blocks (from both pools) that the filesystem is trying to reserve and $ulim(\mathit{log\_overhead})$ is an upper boundary for the additional blocks that will be needed by the log for the checkpointing information and for the filesystem and block descriptors. dtfs calculates $ulim(\mathit{log\_overhead})$ as:

$$ulim(\mathit{log\_overhead}) = ulim(\mathit{ifiles}) + ulim(\mathit{blockdescriptions})$$

In this equation $ulim(\mathit{ifiles})$ is the maximum overhead required for writing out the ifile information for all filesystem versions currently registered with the log. $ulim(\mathit{blockdescriptions})$ is an upper limit for the number of blocks required to store the block descriptions that are part of the filesystem descriptions for the various write clusters. These values are bounded by

$$ulim(\mathit{ifiles}) = \mathit{number\_filesystem\_versions\_registered}$$

(Provided that the ifile inode data structure never exceeds one filesystem block in size which is quite a reasonable assumption to make.)

and

$$ulim(\mathit{blockdescriptions}) = \left\lceil \frac{\mathit{blocks\_allocated} + \mathit{blocks\_requested}}{\left\lfloor \frac{sizeof(\mathit{blockdescription})}{\mathit{blocksize}} \right\rfloor} \right\rceil$$

Please note that the allocation condition is not the exact number of blocks that will actually go to the log because of two facts:

[4]The segment header is not taken into account in this calculation altough it is possible to store log metadata information (the filesystem descriptors) there.

1. A filesystem might reserve more blocks than it actually needs.
2. The allocation condition does not represent a precise calculation of the total number of blocks needed rather than a more ore less accurate upper boundary.

Appendix E

Internal Checkpoint Structure

The primary goal in the design of dtfs checkpoint structures has been to minimize the additional number of blocks that must be written to the harddisk containing checkpoint information. As a concession to that requirement, the dtfs checkpoints are a bit more complex than all the other dtfs on-disk data structures. Furthermore it is also possible for a checkpoint to be bigger than one logical disk block. In that case writing the logical disk block containing the checkpoint header serves as a commit.

Figure E.1 shows an overview of dtfs' internal checkpoint structure: The checkpoint header holds the number of filesystem entries in the current checkpoint. Every filesystem entry corresponds to one write cluster. The filesystem entry may also contain the ifile inode for the respective traditional filesystem corresponding to the write cluster described by a particular filesystem entry.

After the ifile inode, the block descriptions for the write cluster are appended. Furthermore the checkpoint header contains information about the total length of the respective checkpoint for convenience.

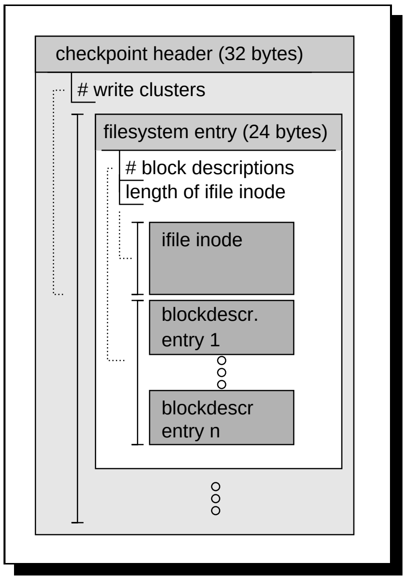


Figure E.1: Internal structure of a checkpoint

Bibliography

[BBD+97] Michael Beck, Harald Boehme, Mirko Dziadzka, Robert Magnus, and Dirk Verworner. *Linux-Kernel-Programmierung*. Addison Wesley, Bonn, 4 edition, 1997.

[BHS95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In USENIX Association, editor, *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 277–288, Berkeley, CA, USA, January 1995. USENIX.

[Bra98] Peter J. Braam. The CODA distributed file system. *Linux Journal*, 50:46–51, June 1998.

[CNC+96] Peter M. Chen, Wee Teck Ng, Subachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[IBM93] IBM. *AIX Version 3.2 System Management Guide*. IBM, 1993. Part.No: SC23-2457-01.

[JL96] James E. Johnson and William A. Laing. Overview of the Spiralog File System. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):5–14, October 1996.

[KS92] James Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transaction on Computer Systems*, 10(1):3–25, February 1992. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[OD88] John Ousterhout and Fred Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Technical Report # UCB/CSD 88/467, Univ. of Calif., Berkeley, October 1988.

[RO90] Mendel Rosenblum and John K. Ousterhout. The LFS storage manager. In USENIX Association, editor, *Proceedings of the Summer 1990 USENIX Conference: June 11–15, 1990, Anaheim, California, USA*, pages 315–324, Berkeley, CA, USA, Summer 1990. USENIX.

[Ros92] Mendel Rosenblum. *The Design and Implementation of a Log-structured File Sytem*. 1992. Ph.D. thesis. Also avaliable as Tech. Report UCB/CSD 88/467.

[Rub97] Alessandro Rubini. The virtual filesystem in Linux. *Linux Journal*, 37:52–58, May 1997.

[Rub98] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, 1998.

[SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 307–326, Berkeley, CA, USA, Winter 1993. USENIX.

[SSB+95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 249–264, Berkeley, CA, USA, January 1995. USENIX.

[Sta92] William Stallings. *Operating Systems*. Maxwell Macmillian, 1992.

[Sun87] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, June 1987.

[Sun89] Sun Microsystems, Inc. NFS: Network file system protocol specification. *Internet Request for Comments*, (1094), March 1989.

[Vah96] Uresh Vahalia. *UNIX Internals*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[VGT95] Uresh Vahalia, Cary G. Gray, and Dennis Ting. Metadata logging in an NFS server. In USENIX Association, editor, *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 265–276, Berkeley, CA, USA, January 1995. USENIX.

[WBW96] Christian Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the server for the Spiralog File System. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):15–31, October 1996.