`dtfs`
A Log-Structured Filesystem For Linux

Christian Czezatke

Revision: 1.6
August 31, 1998

Abstract

This paper discusses the design and implementation of dtfs, a log-structured filesystem for Linux. dtfs features a generic core providing logging facilities that is 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.

Contents

Part I

Design

Chapter 1

The Design Of dtfs

1.1 General Considerations

1.1.1 Design Goals

dtfs' on-disk data structures have been designed with future extensibility in mind. It should be possible to add a number of new features to dtfs without having to alter any of the on-disk data structures. The dtfs on-disk data structures already incorporate hooks for the following additional functionality that goes beyond a mere implementation of a log-structured file system:

- multiple logical file systems (Up to 255 file systems would be possible without altering the dtfs data structures).
- dtfs Versioning (Up to approx. 65000 versions would be possible without altering the dtfs data structures).
- Platform-independence (i.e: a dtfs filesystem created on one particular platform should be accessible by all platform supporting dtfs regardless to different word-sizes or endianness.)
- 64 bit wide data structures for various crucial filesystem parameters.
- efficient checkpointing
- Handling of fragments
- Block Sharing
- Files $> 2\,GB$ for the ext2 tradfs.
- A simple hierarchical storage management

The design has been made with storage efficiency in mind, too, but the primary focus was on extensibility.

1.1.2 Explanation of Planned Features

Multiple Logical File Systems

Since dtfs is separated into generic code and code that is specific to one traditional file system making use of the generic dtfs part, it should be possible to place more than one file system (and even different traditional file systems) within one dtfs file system. From a user's point of view this means that a partition containing a dtfs file system could export more than one root directory. Each root directory would then be the starting point for an independent file system tree, a *logical file system*.

dtfs Versioning

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

Having the ability to define read-only snapshots of a file system while users can still access the file system in read/write mode is quite convenient for backup purposes. A backup is meant to represent the status of a file system at a certain point of time.[1] In reality the problem with backups is that it takes a considerable amount of time (minutes to hours) to complete the backup since the speed of current backup media is rather limited.

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

In order to avoid that problem, access to a file system is turned off (or switched to read-only) for ordinary users to the file system during backup time. However, this may result in some important services being 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. In many environments such a "denial of service approach" can still be unacceptable.

A dtfs implementation with versioning support could circumvent that problem by defining a read-only version of the file system to be backed up at a certain point of time t. Creating such a snapshot should not last much longer than creating any other dtfs checkpoint, so the delay should be within a limit that is expected for normal file system operations. This snapshot could then be used as a starting point for a backup that is guaranteed to match the filesystem status as it was at the time t, while users can still continue to access the filesystem in read/write mode while the backup is created.

Platform-Independence

Since Linux is an OS that runs on a variety of different hardware platforms, a filesystem for Linux should be designed in a way that it eases interplatform operability: A filesystem created on one particular hardware running Linux should be accessible from any other hardware platform supported by Linux, 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 on all platforms. Therefore a dtfs file system should be readable on any platform regardless to the platform on which it has been created.

[1] This is required for consistency reasons.

However, this causes a problem with traditional file systems that are not specified in a way that supports this platform-independence.[2] In order to cope with that problem, the filesystem descriptor for each traditional file system has a flag that can be used to indicate that the filesystem has been created on a little-endian platform and does therefore not conform to the RFC 1014 XDR standard.

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

64 bit wide data structures

dtfs incorporates 64 bit wide data items into crucial data structures of the dtfs core in order to avoid the potential problems of running out of address space in the near future (as this is currently the case on 32 bit Linux platforms with the 2 GB 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".[3] An eight byte representation has been chosen in dtfs for the following data:

- "Logical" dtfs time stamps
- disk block addresses and disk sizes
- segment numbers

A four byte representation has been chosen for "natural" time values. However, when any of these data types appear in a dtfs data structure, they are always preceded by four reserved bytes that are reserved and 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 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.[4]

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 like the creation of a file.

BSD LFS uses segment batching while Sprite LFS maintains a directory operations log in order to avoid such unwanted effects. Both LFS implementations use

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

[3] 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 file system, according to [SELTZER].

[4] GCC eases the implementation of 64 bit integral data types by 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.

sophisticated recovery algorithms based on that additional information in order to be able to do a roll-forward after a crash.

dtfs tries to avoid all these problems by writing out checkpoints more often. 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 as described in section 1.4.

However, since the on-disk data structures of dtfs are designed in a way that using a unified area for segment description and directory operation logging is possible, adding a directory operation log should be quite straightforward, should the need arise.Segment batching could be added to dtfs, too, since it does not require any additional on-disk data structures.

Handling Of Fragments

Different strategies can be used in order to address the problem of efficient handling of small files. One approach is to de-couple the allocation of disk space from a block-oriented strategy at all.[5] Since the first implementation of dtfs will be based on ext2 data structures, it seems natural to keep the block-oriented allocation strategy that is used by ext2 in oder to keep things simple.

ext2 is already prepared for the addition of fragments. Fragments are sub-block units of disk space allocation that help to reduce the waste of disk space caused by even small files allocating a full disk block.

Fragments can be supported by dtfs by adding additional entries in the segment description area that describe the contents of a fragment block (i.e: which fragment belongs to which file).

Block Sharing

Disk space could also be preserved by applying a similar techniques as used by [MERGEMEM] to disk blocks: When file A as well as file B contain a block with identical data in it, it would be possible to allow the meta-data of both files to point to the same physical block and to employ a copy-on-write strategy if such a block should be changed in one of the two files.

A user-space program running at low priority could scan the files on the file system for blocks that can be shared and present suggestions to the kernel implementation of dtfs concerning such merges of blocks.

Again, this could be covered by allowing more than one entry in the segment description area for each block in a checkpoint.

A Simple Hierarchical Storage Management

Generally speaking, a log-structured file system writes its data to disk 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 would allow the reconstruction of the file system's state at any checkpoint at any given time by using the information on this tertiary storage medium.

[5] This approach has been taken by the designers of the Spiralog file system, for example.

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 file system at any checkpoint.

The knowledge of the original location of the segment is required since the metadata structures of the traditional file system are generally unaware of such issues specific to a log-structured file system 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 file system.

1.2 The dtfs Disk Layout

1.2.1 Introduction

Basically, a dtfs partition consists of a reserved area at the beginning of the device holding some dtfs-specific metadata information, a number of segments and another reserved area at the end of the device that replicates most of the metadata found in the first reserved area.

Segments are all of the same size (with the exception of smaller segments in periodic intervals due to metadata replication) and are either marked as in use by the log or as free. All the segments in the log form a doubly linked list so that the log can be traversed easily.

1.2.2 Overview of dtfs Metadata Structures

General Remarks

dtfs uses five different kinds of metadata sections:

1. dtfs super block
2. traditional filesystem super block
3. checkpoint area
4. checkpoint block
5. segment usage bitmap

dtfs super block

The dtfs super block holds some general information about the file system like various geometry specifications and some flags indicating whether the file system has been unmounted cleanly the last time it was used. Furthermore the dtfs super block holds a number of filesystem descriptors. Each file system descriptor represents a traditional file system within the dtfs file system. It contains pointers to the super block and to the checkpoint areas of the traditional file system. The dtfs super block is expected to reside at byte offset 1024 from the start of the device.

Furthermore, since the dtfs super block contains vital information about the file system that is absolutely essential for accessing the file system, the super block is replicated at the end of the device. In addition to that, a dtfs implementation is expected to replicate the dtfs super block several times on the device.

traditional filesystem super block

Every traditional file system within a dtfs file system has its own, traditional super block. This is necessary since many file system expect information like the maximum number of inodes in the file system to be available. Again, the traditional filesystem super block could be absolutely essential in order to be able to retrieve the information within that traditional file system. Therefore every traditional filesystem super block is replicated et the end of the device. Furthermore a dtfs implementation is encouraged to replicate the traditional filesystems' super blocks several times on the device. (A reasonable approach could be to replicate the traditional filesystems' super blocks whenever the dtfs super block is replicated).

dtfs super block copies are encouraged to point to different copies of the traditional file system super blocks.

checkpoint area

Each traditional file system has two checkpoint areas that are pointed to by the traditional filesystem's filesystem descriptor in the dtfs super block. Every checkpoint area is replicated once at the end of the device.[6] Checkpoint areas contain checkpoint entries that represent the state of a certain version of the file system 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 actual checkpointing information.

Checkpoint areas are very important for accessing the file system. The checkpoint areas are used alternately by a dtfs implementation, so there is always one valid checkpoint area left, should the writing to one checkpoint area fail.

Checkpoint areas consist of one filesystem block in the current dtfs implementations. However, future dtfs implementations might use more than one filesystem block for checkpoint areas in order to support more versions of a certain traditional file system.[7]

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

checkpoint block

A checkpoint block is written out every time a cluster write operation is finished (see chapter 1.4 on further information about checkpointing in dtfs). In general, checkpoints mark a consistent state of one or more filesystems and on or more versions of a file system at a certain point of time. Checkpoints contain the block description for the blocks in the last write operation and the inode(s) for the ifile(s) for the traditional file system(s) they represent. Furthermore, there are special checkpoint blocks at the end of each segment. These special segment blocks are used to store some information about the respective segment, i.e: the segment number

[6]There are pointers to these copies of the checkpoint areas in the filesystem descriptor, too.

[7]Since every checkpoint area has a number of entries field, this number of entries field can be used to determine the number of filesystem blocks in therespective checkpoint area.

and pointers to the segment checkpoints of the previous and the next segment in the log.

If there is more than one checkpoint in a segment, these checkpoints form a linked list, too.

segment usage bitmap

A segment usage bitmap consists of one or more consecutive filesystem blocks. Every bit in the segment usage bitmap corresponds to one segment in the file system. If the bit is set, the corresponding segment is in use. The dtfs kernel implementation sets bits in the segment usage bitmap as it writes to the log fill the corresponding segments, while the cleaner will re-set bits to 0 as it frees the corresponding segments.

There are two segment usage bitmaps in a dtfs file system that are used alternately so that there is still one valid segment usage bitmap left, should the writing to one segment usage bitmap fail. Furthermore, each segment usage bitmap is prefixed with a small header. This header contains summary information about the state of segment allocation such as the total number of free segments and the number of blocks in them. Furthermore, the segment usage header also contains the number of the segment that was allocated as the latest one and the number of the next segment that will be allocated in the case the log requests another segment. This information should help in providing a fast crash recovery for the segment usage bitmap.

In order to verify that the segment usage bitmap's integrity, the header also contains two checksum fields which help to detect an only partially written bitmap. Such a situation could arise in the case of a dtfs file system whose segment usage bitmap does not fit into one block anymore.[8] In that case it could happen that not all blocks of the segment usage bitmap have been updated properly resulting in an inconsistent state of this data structure after a crash.

A more sophisticated cleaner might require additional information about a segment than just a used/unused bit. However, information about the contents of a segment and its approximate time of writing can already be obtained from the block descriptions in the checkpoints within the segment, so this data structure is considered to be sufficient for the first version of dtfs.

1.2.3 Cluster Writes — Write Clusters

In order to achieve good I/O throughput, a log-structured file system clusters block write operations together and performs them as one write to succeeding blocks. (This can be done due to the append-only nature of the log.) Such a write operation is called a cluster write.

One cluster write consists of one or more write clusters followed by a checkpoint. The checkpoint commits the cluster write and makes the changes to the affected file systems permanent (with the exception of a log checkpoint. See section 1.4 for an in-depth discussion of log checkpoints).

A write cluster consists of a number of blocks belonging to a particular version of a particular traditional file system. Every version of a traditional file system is allowed to have at most one write cluster in a cluster write. A dtfs implementation is encouraged to arrange blocks within a write cluster in a way that allows efficient read-ahead strategies to be employed.

[8]When mkdtfs is called without any special parameters, it defaults to creating a filesystem with 4kb blocksize and a segment size of 128 blocks per segment (i.e. segments that are half a megabyte in size). In that case the segment usage bitmap needs more than one filesystem block when the total size of the partition the filesystem is created on exceeds approx. 2GB.

1.2.4 The Role of the Segment

The segment is the unit of disk space allocation and the unit of cleaning. A segment can either be marked as clean or as occupied by the log. Furthermore, a segment can be locked (i.e: no cleaning is allowed to take place) or as being bad (it will not be used for storing log information in it). In general, segments are all of the same size. However, there are two cases in which a segment may be smaller than expected:

1. The first and the last segment on the disk might be smaller: The last segment could be smaller because there is not enough space left at the end of the device for another full-sized segment. The first segment could be smaller in order to enforce some kind of alignment for further segments: For example, it could be beneficial to start a 512kb sized segment at a 512kb boundary. Since there is some reserved space for super blocks and the segment usage tables at the beginning of the device, the first segment will most likely be a little bit smaller than the average.
2. Smaller segments are used in periodic intervals to allow super block replication: The dtfs super block (and maybe the tradfs super blocks) are replicated throughout the device in periodic intervals. In order to make room for this replications, segments that are a bit smaller must be inserted at periodic intervals.[9] Furthermore a segment can hold one or more cluster writes.

Furthermore, cluster writes are not allowed to extend across a segment boundary. In order to enforce this condition, a cluster write can be split into two cluster writes by inserting a log checkpoint at the end of the segment. dtfs checkpointing will be discussed in more detail in section 1.4.

Apart from the items described above (which are only related to free space management and cleaning) the segment does not have any special semantic for dtfs.

1.2.5 Detailed Disk Layout

Figure 1.1 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 the traditional filesystems' super block(s) and the checkpoint areas(s) are also stored within this reserved area at the beginning of the device. The super block and the traditional filesystems' super block(s) are replicated in regular intervals. In oder to compensate for that, the respective segment will be a bit smaller than the other segments.

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

1.2.6 Metadata References

However, all the disk-layout specifications that have been given in section 1.2.5 are actually just conventions that need not be adhered to from a dtfs implementation's point of view: All the dtfs metadata information can be located starting from the dtfs super block.[10]

[9] The current mkdtfs implementation replicates the dtfs super block and the tradfs super block in every 64th segment by default. So every 64th segment will be 1–2 blocks smaller (depending on the logical blocksize used).

[10] So the only requirement that must be met is that a copy of the dtfs super block must start at offset 1024 from the beginning of the device.

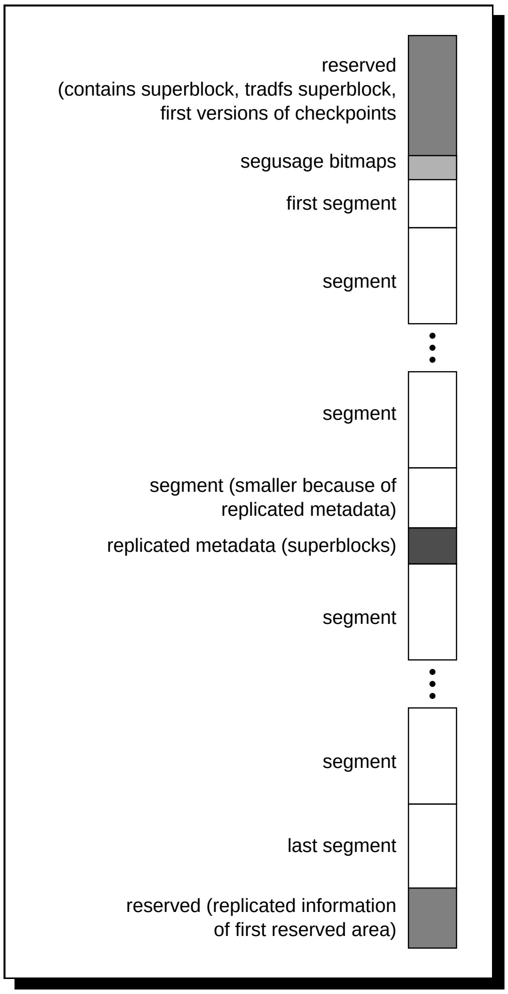


Figure 1.1: dtfs disk layout overview. Device block numbers are increasing from left to right.

Figure 1.2 shows how the different kinds of metadata information are linked together. Starting from the dtfs superblock, the superblocks of all the traditional file systems, the segment usage bitmaps and the checkpoint areas can be found.

1.2.7 Description of the dtfs Metadata Blocks

The current design for dtfs uses the following metadata blocks in order to define its disk layout:[11] (Reserved entries in the metadata structures and entries used for padding should always be set to 0 if no other default value for them is specified in this document.)

`dtfs_super_block` A super block to define general key items of a dtfs filesystem.

`dtfs_segusage` One or more blocks holding the segment usage bitmap.

`dtfs_checkpoint_area` Blocks that hold information about checkpoints and versions of a logical file system.

[11] Apart from the blocks listed below, dtfs also requires a superblock for the traditional file system for each logical file system in the dtfs file system.

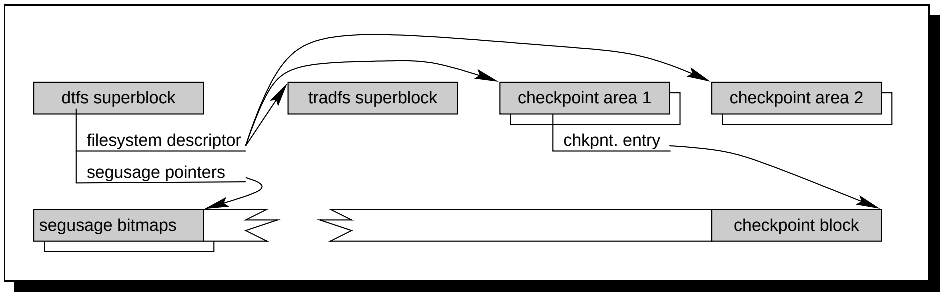


Figure 1.2: An overview of dtfs metadata referencing. Device block numbers increase from left to right and from top to bottom.

`dtfs_checkpoint_header` One or more block describing the blocks in a cluster write. It can also serve as a commit for the cluster write. Every cluster write is ended by such a data structure.

`dtfs_segdatabase_block` Blocks for storing directory log and segment content description information.

dtfs_super_block

This data structure is placed on a well-defined block of the device and is replicated several times on the device. It contains vital information about the filesystem layout on the device. Furthermore, it contains information about the logical file systems found in the respective dtfs filesystem. Currently, up to twelve filesystem descriptors can be stored in the super block directly.[12] In order to allow the support of more than twelve traditional file systems in the future, the dtfs_super_block structure contains a pointer to another device block that can hold additional filesystem descriptors.

The super block also stores the size of a filesystem descriptor. This makes it possible to add extensions to the filesystem descriptors in later versions without the risk of loosing compatibility.

The super block also contains 12 unused 4 byte entries at the moment that could be used for future enhancements. These unused entries should be set to 0 for now.

It is also expected that a dtfs implementation replicates the filesystem descriptor blocks and the traditional filesystems' super blocks whenever it replicates the dtfs superblock.

The size of this data structure is 1024 bytes.

dtfs_checkpoint_area

This data structure holds all information about checkpoints for a special logical file system. Each logical file system has two different checkpoint areas that are used by dtfs alternately. A timestamp within each checkpoint area indicates which one is current. Every checkpoint is replicated once at the end of the device to avoid single point of failures.

A checkpoint area contains a list of checkpoint entries. Currently up to 31 checkpoint entries can be in a checkpoint block. Future dtfs implementations are expected to add additional blocks that contain a list of checkpoint entries only, immediately after the respective checkpoint block.

Checkpoint entries are described later in chapter 1.2.8.

The size of this data structure is 1024 bytes.

[12] filesystem descriptors will be explained in section 1.2.8.

dtfs_checkpoint_header

This data structure serves a twofold purpose: First of all it commits a cluster write. Every cluster write is ended by a checkpoint block that starts with a checkpoint header. Since cluster writes are not allowed to cross segment boundaries, there is always one dtfs_checkpoint_header data structure at the end of each segment. So these structures can also be used to form a doubly linked list of segments that are currently in use by the log.

Furthermore a checkpoint header located at the end of a segment also holds the unique number of the segment that as been assigned to it by the cleaner at the time of writing and a special segment flags field that can indicate whether the segment

- is clean (i.e: is not currently used by the log)
- is locked (i.e: must not be cleaned)
- has bad blocks (segment will never be used)

The checkpoint header at the start of the segment also contains information about the size of the particular segment in logical filesystem blocks.

Every checkpoint header contains the offset in logical filesystem blocks to the next checkpoint header in the same segment and the offset in logical filesystem blocks to the checkpoint header ending this particular segment. Furthermore it has a logical timestamp indicating the time of writing for the cluster write it commits. There are also two 4 byte checksums that can be used to ensure the commit semantic of writing out the checkpoint header.

Genreally, the checksums should not be needed. However, for the implementation of 4.4BSD LFS checksums were added to the partial segment block. The reason for this design decision was that in BSD the order in which block writes in multiblock write operations are actually performed, cannot be determined. — According to the Linux Kernel Hacker's Guide, Linux uses an implementation of the elevator algorithm to re-order I/O requests. This approach guarantees that Linux will do block writes in ascending block number order. However, reserved entries for checksums are still there in the dtfs segment header since it might be possible that either a re-ordering of writes is again done in the harddisk controller and to avoid the necessity of an overhaul of the checkpoint header structure should Linux' write policy be changed.

The first implementation of dtfs will not use these checksum areas and rely on the elevator algorithm to guarantee that the checkpoint block is actually written out last.

Furthermore a checkpoint header holds a field indicating the type of checkpoint that is represented by the particular checkpoint header and an overview of the write cluster descriptions following immediately.

1.2.8 Additional Data Structures

The metadata blocks described in the previous section will also make use of a bunch of other data structures:

dtfs_filesystem_descriptor A data structure used within the super block that holds all the required information for a logical file system.

dtfs_checkpoint_entry Used within the checkpoint blocks to describe a certain checkpoint for a logical file system.

dtfs_fsys_entry Starts the description of a write cluster in a cluster write.

dtfs_dirlog_entry A proposed 24 byte data structure describing an entry in the directory operation log. However, dtfs does not use a directory operations log in the first implementation (and probably never will).

dtfs_blkdesc_entry Another 24 byte structure that contains the description of a certain block in a write cluster.

dtfs_segdatabase_entry Either a dtfs_dirlog_entry or a dtfs_blkdesc_entry.

dtfs_segusage A header for the segment usage bitmap.

dtfs_filesystem_descriptor

A dtfs superblock holds a number of filesystem descriptors. Every filesystem descriptor can be used to describe one traditional file system within a dtfs file system.[13] The filesystem descriptor contains the addresses of the checkpoint areas for the respective filesystem, the inode IDs for some crucial inodes (ifile, root inode) and a flags field that is currently used for marking the filesystem as "unmounted cleanly" and as "readonly".[14]

dtfs_checkpoint_entry

A checkpoint block is made up of several checkpoint entries. A checkpoint entry holds the timestamp at which the checkpoint was made and the address of the corresponding checkpoint block. A flags field indicates whether the checkpoint is current, a version (r/w ⇒clone, r/o ⇒snapshot) or read-only. Furthermore the version number of the predecessor and the time at which the version was created from the predecessor is also held in a checkpoint entry.[15]

dtfs_fsys_entry

This data structure holds some basic information about a write cluster, like the id and the version number of the file system owning the blocks in the write cluster. This data structure can also include the ifile inode for the particular filesystem/version number in the checkpoint it belongs to.

A dtfs_fsys_entry also specifies the number of blocks in the write cluster and the number of block descriptions belonging to that write cluster. The number of these block descriptions (that are immediately following their respective `dtfs_fsys_entry`) should usually match the number of blocks in the respective write cluster with the exception of a dtfs implementation that uses a directory operation log or supports fragments.[16]

dtfs_dirlog_entry

An entry describing a directory operations log. It consists of an operation type indicator, the inode of the parent directory and the file (or directory) affected. Furthermore it contains the number of the data block of the parent directory that has been changed by the directory operation.

[13] The first implementation of dtfs will only use the first of these entries since multiple logical file systems will not be supported in the first dtfs implementation.

[14] Altough this will probably be handled by the traditional file system.

[15] If a version does not have a predecessor, the predecessor version number and the time at which it was derived from the predecessor should be set to 0.

[16] In such a case the number of block descriptions can exceed the number of blocks in the write cluster.

Contrary to Sprite LFS, dtfs does not store the name of the child file or directory affected by the change since this would use additional amount of storage and complicate parsing of the directory log/segment database. Furthermore, the benefits from storing the filename of the child in the directory operation log are questionable anyway.

Please note that dirlog entries are defined by dtfs, but not used, since dtfs uses lightweight checkpoints instead of directory operation logs. (See section 1.4 for more information on that).

dtfs_blkdesc_entry

A data structure describing the contents of a certain block in the current write cluster. It contains the inode number it belongs to and a logical block offset within the file. It is worth mentioning that dtfs does not distinguish between traditional filesystem metadata blocks and actual data blocks here. The traditional filesystem implementation must be able to map metadata information for a file into the logical block address space of the respective file.

In the case of the ext2 traditional filesystem implementation this is done by mapping indirect blocks to very high block numbers that cannot be used by ext2 anyways because these blocks cannot be addressed with the current ext2 block addressing scheme.

However, if such an approach should not be possible for any traditional file system, it would still be possible to use a reserved inode number and assign this inode number to all metadata blocks thus introducing a separate block number name space for all metadata blocks.

Furthermore the id and version of the filesystem the block belongs to is also stored in the block description.[17] Since only a 4 byte entry is used for the version number there is the remote possibility of a version counter overflow for an inode. However, this is not a severe problem for dtfs: In order to deal with the problem dtfs implementation is expected to simply disable the cleaning version number optimization for this particular file before the counter overflows.

Furthermore the block description does also contain the natural time at which the block was written. This is necessary in order to be able to support data checkpoints properly. (See section 1.4.2 for more information about data checkpoints.)

dtfs_segdatabase_entry

Actually a union of directory log entry and block description entry. Whether an entry is either a block description or a directory log entry can still be determined by the type field.

dtfs_segusage

This header is used at the start of the two segment usage bitmaps. They have an entry for a logical time (that is used to determine which of the two segusage bitmaps is the current one) and two checksum entries that are unused by the current dtfs implementation. Furthermore the segment usage header also contains the number of the last segment allocated and the number of the next segment that is to be allocated should the log request another segment. The information about the next segment to be allocated is required in order to be able to correctly establish the doubly linked list of allocated segments that is formed by the segment headers of all currently used segments.

[17] This additional information should greatly ease cleaning when multiple filesystems/versions are present.

Furthermore this information also helps in speeding up the reconstruction of a valid segment usage bitmap after a crash should the need arise.

It is not necessary to store the number of segments within that header since this information can be obtained from the dtfs super block.

1.3 Filesystem-Mapped Metadata Information

1.3.1 Overview

dtfs tries to map as many metadata information as possible into regular files. This helps to avoid having to deal with separate metada information types for the traditional file system used together with dtfs and for dtfs itself. dtfs gets along with only using the metadata structures provided by the respective traditional file system. Therefore an improvement to the filesystem metadata structures in the traditional file system will also be available for dtfs metadata handling as well.

The following dtfs data structures are mapped into regular 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 the current version number 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 version number of the file as used by a log-structured file system cannot be found in the inode structure of a traditional file system. In order to avoid portability problems, it is a good idea not to add the version field to the inode of the traditional file system (since it cannot be expected to conveniently fit into the inode structure that is already used by the traditional file system).

1.3.2 The .inode File

This file contains all the inodes of the traditional file system with the exception of the inode for the .inode file itself.[18] In order to allow quick access to a special inode, dtfs relies on the metadata structures of the traditional file system, since inode accesses are finally translated into simple file accesses, too.

1.3.3 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: Every block of the .ifile is represented by one bit in the .iusage file. The bit is set when all the inodes in the corresponding .ifile block are already in use.

[18] However, the current ext2 implementation of dtfs reserves space for the ifile inode in the ifile itself, but does not use it.

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. [SELTZER] investigated that problem and has 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 these problems, 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 [SELTZER] 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] 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.[19] The traditional file system 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 120 inodes starting from any given directory inode, the benefits of such a data structure would be questionable compared to the added complexity.

[19] With an ext2-based traditional file system and a logical block size of 4 kb, there are 32 inodes within one .ifile block.

1.3.4 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.

1.4 dtfs Checkpoint

1.4.1 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 "classical" 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, a number of sophisticated techniques have been developed 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:

This operation requires the allocation of a new inode for the file and the modification of a directory file. Altough each of these two operations could be performed independent of each other when reconstructing a filesystem after a crash, it must be assured that either both of such two matching operations are re-done or none of them is re-done or an inconsistency in the filesystem will arise.

In assure this atomicity, Sprite LFS uses a technique called directory operation log while the BSD LFS uses special write order conventions named "segment batching".

dtfs does not use any of these techniques in order to keep track of such interdependent operations when recovering a file system. The approach taken by the dtfs implementation is to completely rely on checkpoints when it comes to filesystem recovery after a crash. This is achieved by adhering to the following principle:

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

The problem with checkpoints is that they might be rather costly to write thus degrading overall filesystem performance. Furthermore writing checkpoints rather 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 (filesystem mapped) 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 file system at a certain point of time that allows to remount the file system after a crash. However, the time required to rebuild all the internal dtfs data structures from that checkpoint might differ:

1.4.2 Filesystem Checkpoint Types

dtfs provides the following different checkpoint types:

Unmount Checkpoint

An unmount checkpoint is written only when a filesystem mounted for read/write access is being unmounted. Writing an unmount checkpoint requires the following steps:

1. All dirty data and metadata blocks are written out.
2. A checkpoint block holding the ifile inode (and maybe some hints for the cleaner) is created.
3. The current segment usage block is updated.
4. The current checkpoint areas are updated.
5. All the dtfs super blocks are updated and re-written.

If a file system is unmounted cleanly, there is always an unmount checkpoint present that describes the state of the file system at the time it was unmounted.

Major Checkpoint

A major checkpoint is written out by dtfs every now and then (implementation dependend). Writing a major checkpoint involves the same steps as writing out an unmount checkpoint with the exception of step 5 (writing out the super blocks).

Minor Checkpoint

Writing out a minor checkpoint involves the steps 1 and 2 of writing out an unmount checkpoint, i.e: writing out a minor checkpoint only involves writes to the tail of the log, so no (big) disk head movements should be required to write out a minor checkpoint.[20]

Data Checkpoint

There might be a situation in which dtfs has to write out a checkpoint, but decides to rather not write out all the pending metadata changes in that very checkpoint for efficiency reasons. This could be the case if an application requests an explicit sync for a given file or in the case of an NFS write that requests syncs after each file block written resulting in much overhead since the indirect information for that file would be re-written again and again.[21] In order to overcome that problem, a dtfs implementation might decide to delay the writing of metadata blocks in order to achieve better performance. If the system crashes after a data checkpoint has been written, the missing metadata information can be reconstructed by using the block descriptions of all the write clusters belonging to that file system that have been written to disk since the last minor, major or unmount checkpoint.

[20] However, since a minor checkpoint only has advantages over a data checkpoint when it comes to crash recovery (no reconstruction of indirect data required), it is questionable whether they make sense at all. This is still an open issue.

[21] A good example for that would be an ext2-based traditional file system 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.

Incremental Checkpoint (not implemented in first dtfs version)

Write operations in a log-structured filesystem get particularly inefficient when files are written in many small blocks and a sync request is issued after each small block that has been written (as it is the case for NFS writes with the default, small block sizes). Data checkpoints already help a lot in reducing that overhead. However, in order to minimize the overhead of writing out many checkpoints that are live for only a short period of time, another optimization can be employed.[22]

Writes to a single block in a file followed by a sync to that file can already be reduced to writing just a single block immediately followed by a checkpoint block by applying the techniques described in section 1.4.2. However, this approach still requires every data block written to be placed in its own write cluster resulting in 50% overhead since every data block written has to be succeeded by a checkpoint block.

An incremental checkpoint could be used in order to avoid that overhead. Such a checkpoint would start out with a write cluster consisting of just a single block followed by *two* checkpoint blocks:

1. the first data block
2. the checkpoint block
3. a copy of the checkpoint block (but marking block 2 as unused)

If another block arrives followed by a sync request, the first checkpoint blockwill be overwritten by that data block and a new checkpoint will be appended resulting in the following blocks being in the segment:

1. first data block
2. second data block
3. old checkpoint block (pointing only to the 1st data block marking block 2 as unused)
4. an updated checkpoint block (marking block 3 as unused)

This approach is consistent with the transaction-like write semantics of a log-structured file system since the appending of the new checkpoint block at the end of the current cluster write serves as a commit. Furthermore a write operation does not affect data that has been written by an earlier write since there are two checkpoint blocks that are used alternately so that if an error occurs in the n^{th} block write all $n-1$ writes that have taken place so far can be re-constructed by using the older of the two checkpoint blocks that is still present.[23]

Log Checkpoint

Since a segment is the unit of cleaning, a cluster write 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 cluster write at a segment boundary.

[22] Again, this kind of checkpoint would help to optimize NFS-like access patterns even further.

[23] In order to be able to determine whether a checkpoint block is valid, the checksum for the checkpoint block could be used, should the need arise.

Flushed Checkpoint

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.

1.4.3 Internal Checkpoint Structure

The primary goal in the design of dtfs checkpoint structures was 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.

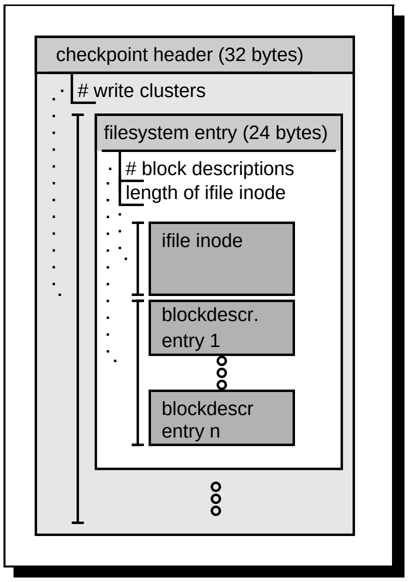


Figure 1.3: Internal structure of a checkpoint

Figure 1.3 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 file system 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.

1.5 Recovering From A Crash

1.5.1 Fast Checkpoint Writes vs. Fast Recovery: A Tradeoff

Unfortunately all the sophisticated checkpointing algorithms outlined in section 1.4 result in additional overhead when it comes to re-constructing the file system after a system crash. Re-mounting a file system on a major or an unmount checkpoint is quite straightforward, while reconstructing all the necessary data structures from an incremental checkpoint can be quite tedious.

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

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

This does not necessarily mean that there must be a major checkpoint every 32 segments for a given logical file system. A logical file system can have its latest unmount checkpoint many segments away from the current head of the log.

The MCI should be a filesystem-settable value in order to allow adaption for fast and big storage devices that hold multiple logical file systems.

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.

1.5.2 How Crash-Recovery Is Performed

Basic Algorithm

If a dtfs file system has been unmounted cleanly, there will always be an unmount checkpoint present allowing the efficient re-construction of all internal dtfs data structures that are necessary for operation. If a file system has not been unmounted cleanly, dtfs starts up by reading in the last unmount 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 file system.

The forward scan continues until the end of the log is reached or until the maximum number of segments have been scanned (if the log continues beyond that number of segments, this is an error that would most likely have its cause in an erroneous dtfs implementation causing a cyclic segment list). 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 or if the log continues beyond the MCI boundary given above, the log is truncated at the last cluster write with a checkpoint for A_x.

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

What still needs to be done is the recovery for all other versions of all logical file systems (including A) on the dtfs partition. This can be achieved by applying a similar procedure as the one just described for the file system A_x to all the other file systems: All that is required to achieve that task is to start a scan of at most MCI

segments[24] starting from the location of the respective file system's latest major checkpoint in order to find all the minor (data, incremental. . .) checkpoints of that file system that might exist.[25]

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 blocks 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.

1.5.3 Kernel vs. User-Space

The algorithm described in 1.5.2 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 file system has not been unmounted cleanly and to which no write access is possible at the time of the recovery, for whatever reason.

In that case a user-space tool cannot be used to reconstruct dtfs major checkpoints from minor checkpoints for the dtfs kernel module because no write is possible. In such a case 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 file systems on a dtfs file system regardless to the kind of their most recent checkpoints.

1.6 Cleaning and Versioning

1.6.1 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 file system can never be live in another file system. So only versions of the one file system 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 file system that fulfills all of the following requirements:

[24] Scan MCI segments or until the end of the log is reached, whichever comes first.

[25] There cannot be another minor checkpoint more than MCI segments away from the last major checkpoint of that file system since a major checkpoint is written every time a write to a logical file system takes place and the last major checkpoint is more than MCI segments away.

(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.

But generally speaking, it must be expected that having many different versions of a file system degrades cleaner performance.

1.6.2 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 file system is a data checkpoint or an incremental checkpoint.

Such unwanted side-effects can simply be avoided by refusing to clean segments that have been written after the last flushed checkpoint: The flushed checkpoint marks a state of the file system 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 last 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.

1.7 Filesystem Versioning

This section is an outline of how versioning could be implemented in dtfs by using the existing on-disk data structures. The key data structure for versioning is the dtfs_checkpoint_entry.

1.7.1 Adding a Version

Creating a new version of a file system 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 could be used to support versioning within dtfs:

1. Find an empty checkpoint_entry in the checkpoint block for the respective filesystem.[26]

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. (Especially set the logical creation time to the current logical time) and let it point to the checkpoint for the new version that has just been written out in step 2 and update the checkpoint_entry for the predecessor version, too.

4. Write out the checkpoint areas.

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.

1.7.2 Removing a Version

Removing a version is a rather trivial task:

1. Clear the respective entry in the checkpoint area. (i.e. by setting the timestamp to the NO_TIMESTAMP value)
2. Write out the checkpoint entries.

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

[26] 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 file system.

Chapter 2

An ext2 Personality For dtfs

2.1 ext2 Personality-Specific Features of dtfs

2.1.1 The 2GB File-Size Limitation

Why There Is A 2 GB Limitation At All

Currently Linux can handle only files with a maximum size of 2 GB. This limitation is caused by two reasons:

1. The Linux VFS layer cannot deal with file offsets $> 2^{31}$ bytes on 32 bit platforms.[1]
2. The ext2 on-disk inode data structure does not allow to store file sizes that do not fit into a 32 bit word which would result in a maximum file size of 4 GB. Due to other constraints, the current ext2 implementation in the Linux kernel deliberately limits the file size to 2 GB. However, the indirect block structure of ext2 would allow file sizes of up to 4 Tb on a filesystem with a block size of 4 kb.

While it is impossible to overcome the first reason without major changes to the Linux VFS layer, it is possible to address the second issue in a filesystem implementation.

Avoiding the 2 GB ext2 Limitation in dtfs

The current dtfs traditional file system implementation is based on ext2 and it is desirable to keep the dtfs tradfs implementation close to the ext2 implementation in order to be able to incorporate future ext2 enhancements into dtfs effortlessly. So using one of the currently unused entries in the ext2 inode structure to represent file sizes $> 2\,GB$ is not acceptable since the ext2 maintainers might decide to use these entries for other purposes in the future.

Since there is no direct compatibility between dtfs using an ext2 tradfs implementation and ext2 itself[2], it does not harm compatibility to move around entries in the given ext2 inode data structure. So the 4 byte ext2 inode entry for the atime can be used by dtfs for its own purposes since dtfs stores the atime of the file separately. So the atime entry can be moved around within the ext2 inode structure

[1] However, at least the Linux/Alpha 2.0.33 kernel together with Jay Estabrook's axp-diffs does not seem to have that problem.

[2] i.e: you cannot (read-only) mount a dtfs partition as an ext2 partition and vice versa.

forming an eight byte entry together with the file size attribute that is already there thus allowing file sizes $> 2\,GB$.

Altough it would be possible to overcome the 2 GB file size limit in ext2 with this approach, this should not be implemented in a dtfs kernel module before consulting the ext2 maintainers.

Benefits for 64 bit Linux platforms

It is already possible to get the VFS layer support large file offsets as it is done by the axp-diffs for Linux/Alpha. On these platforms support for these large files by dtfs should be available without any other changes to either the kernel, the C library or to application programs.

However, 32 bit platforms will not be able to take advantage of dtfs big file support in the forseeable future.

2.1.2 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 file system too. Since the ext2 inode structure is 128 bytes in size and the maximum file size in ext2 is limited to 2 GB, approximately 16 million inodes could be in one file system.

While this number seems to be comfortably large for most cases there might be situations in which this limit is too low. Fortunately, this limit can be overcome by applying the change to the ext2 inode structure described above.

The only problem remaining is the fact that 32 bit platforms cannot pass file sizes $> 2\,GB$ to user space. However, this is not too hard a restriction to live with in the case of the ifile since

- The size of the ifile is not important to user-space programs (as long as there are other means to figure out the maximum number of nodes that is currently allowed, at least for 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.

2.1.3 Delaying Metadata Block Allocations

The current implementation of dtfs makes 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.

2.1.4 ext2 Write Cluster Layout

Figure 2.1 shows the layout of an ext2 write cluster. The following write cluster layout is suggested for an ext2-based traditional file system based on dtfs:

The data blocks in an ext2 write cluster are grouped together file-by-file. A file's data blocks (ordered by 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.

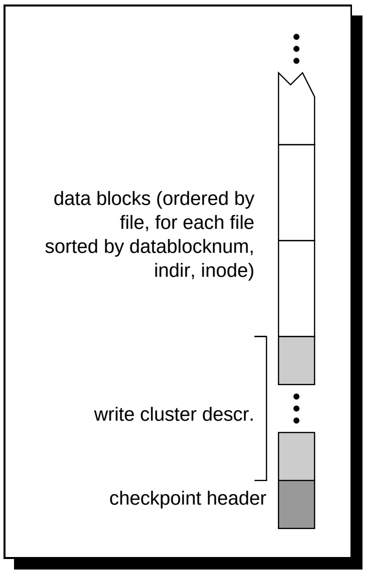


Figure 2.1: Schematic layout of an ext2 write cluster

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

This block ordering scheme should allow efficient read-ahead as well as simple implementation since no block depends on another block that is written after it.

Part II

The dtfs Module Implementation

Chapter 3

General Considerations

3.1 Overview

3.1.1 dtfs Main Architecture

One major goal in the implementation of dtfs has been to establish a well-defined interface between a "*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 on top of this core. This separation should allow to add future filesystem personalities to a dtfs implementation quite easily. It should even be possible to have more than one traditional file system residing on one dtfs partition that use different filesystem personalities.

Figure 3.1 gives an overview of the main dtfs implementation architecture. The

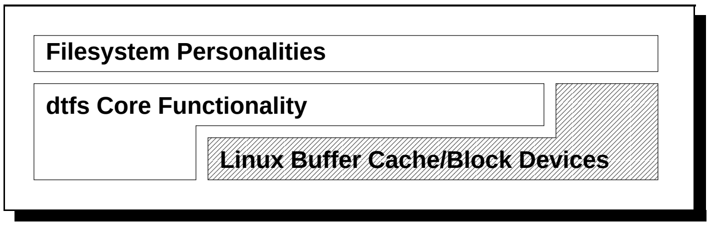


Figure 3.1: An overview of the dtfs implementation architecture.

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. (This will be discussed in more detail in chapter 5.)

3.2 Conventions

3.2.1 Splitting The Implementation Into Modules

In order to get a first working version of dtfs as fast as possible, performance is not a main issue in the current dtfs implementation. The main focus is 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 avoid the introduction of new errors when putting the (already tested) various independent modules together.

3.2.2 Coding Guidelines

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

1. Every module is associated with a prefix

   This should make the code more understandable 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 how a new entity of a data structure can be created:

   (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 de-constructed 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 the purpose of the function 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`.

Chapter 4

The dtfs Core

4.1 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

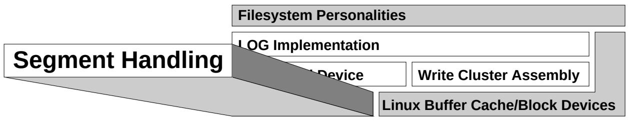
- monitoring free disk space

  This involves checking whether a block reservation request issued by a traditional file system 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 lateron.

- 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
- writing out the dtfs super block(s)
- creating checkpoints

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 4.1 depicts all the modules of the core implementation and their interfaces. The functionality of these various modules will be discussed in the remaining sections of this chapter.

4.2 Segment Bitmap Handling Routines



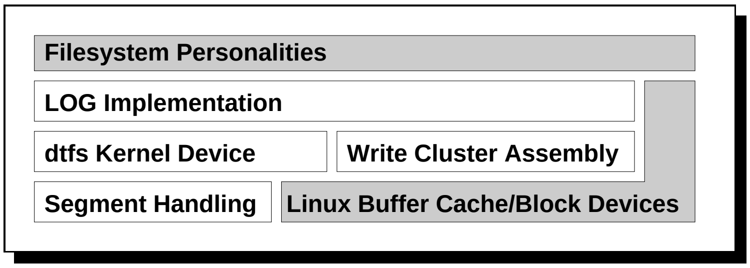


Figure 4.1: The various modules of the dtfs core implementation and how they interface each other. (greyed boxes represent layers that are not part of the dtfs core, but constitute the "world interface" for it)

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 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.

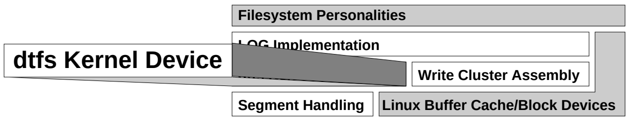
Implementation Notes

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 [KHG], 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 file system 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.

4.3 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 utilites 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 file system. Furthermore 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 file system

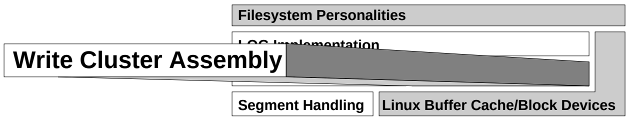
- writing back the contents of a `dtfs_filesystem_descriptor` after it has been modified by the traditional file system.

- Committing the segment usage bitmaps and the modified dtfs super block(s) to disk. — Whether the super blocks shall be committed too, can be controlled by a parameter of the respective call.

Implementation Notes

The current implementation of the log updates only one dtfs super block, namely the first one in block 1. This will change in the final version of dtfs. Furthermore, 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.

4.4 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 cluster write that is performed with one write operation to the underlying device. However, the block assembling code does not check whether cluster write 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 file system .

- queueing of the ifile inode of a file system

  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 file system entry) to be written out to disk, too.

- queuing of the next n dirty blocks of the currently active file system.

- actually performing the cluster write

  After all dirty blocks of all file systems 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 cluster write

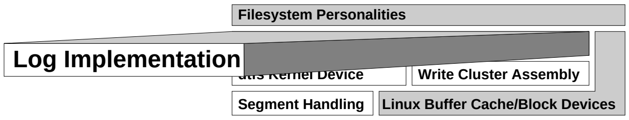
  A cluster write is paused by the log whenever the free space in the current segment has been exhausted. Such a pausing causes the block assembly to do some internal cleanups after a log checkpoint has been written. These cleanups are required in order to make sure that no dirty block is written out twice

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 cluster write. 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 off-load some complexity in the functionality the log has to perform to another module in order to make the log more manageable.

4.5 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 by the log and ask the log to write all its dirty buffers out to disk lateron. 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 5.2.3 on page 44.

Services Provided

The log provides the following functionality: the following functionality:

- registering/unregistering a file system.
- 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 1.4.2 on page 19.

  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 cluster write.

[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 .

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 cluster write 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 4.2 shows a state diagram for a dtfs log. Only state transitions that are shown

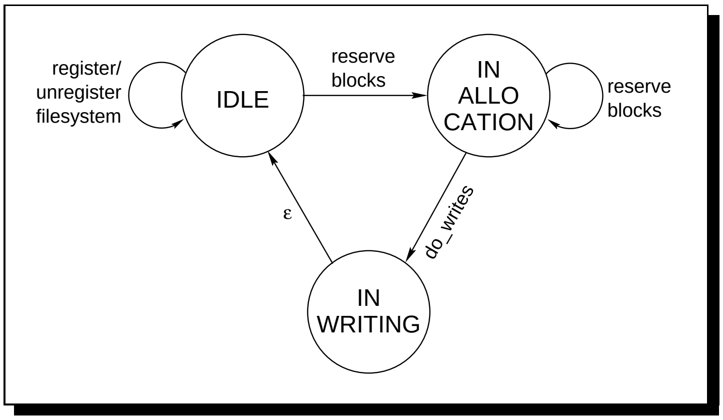


Figure 4.2: The three states of a dtfs log and all the valid state transitions.

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 4.5 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 by the filesystem. 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 overcommittment 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 [ROSENBLUM] 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. Furthermore 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:*

$$free\_blocks >= blocks\_allocated + blocks\_requested + ulim(log\_overhead)$$

In this equation $free\_blocks$ is the number of blocks stored in currently unused segments (not counting the segment header). $blocks\_requested$ is the number of blocks (from both pools) that are currently reserved by traditional filesystems and $ulim(log\_overhead)$ 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 $ulim(log\_overhead)$ as:

$$ulim(log\_overhead) = ulim(ifiles) + ulim(blockdescriptions)$$

In this equation $ulim(ifiles)$ is the maximum overhead required for writing out the ifile information for all filesystem versions currently registered with the log. $ulim(blockdescriptions)$ 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

$$ulim(ifiles) = number\_filesystem\_versions\_registered$$

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

and

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

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.

Chapter 5

Filesystem Personalities

5.1 Overview

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 chapter 4. 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.

5.2 Interfacing the dtfs Core

5.2.1 A Remark About The Current Implementation

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.

5.2.2 Interface Components

The interface between the dtfs core and a filesystem personality consists of four different parts:

- The interface of the dtfs log module.

  This interface has already been discussed in section 4.5, so it won't be described in this section anymore.

- The callback interface of the filesystem personality implementation

- Two data structures that must have some common fields:

  1. The `dtfs_buffer` – struct
     This structure holds some information about a certain dirty block in the filesystem personality. It is required for obtaining the block description for this block 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.

- Two additional functions to obtain and to modify the checkpoint entry of the respective version of the traditional file system.

  The filesystem will use this function to update the number of used/unused inodes.

5.2.3 The Filesystem Personality Callbacks

Callbacks required

A filesystem personality must provide eleven callback functions for the log as listed in table 5.2. These callbacks allow the log to perform a cluster write when one

Function	Purpose
`active_this_turn`	True if the filesystem has dirty buffes 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`	# of dirty blocks for current commit level.
`get_next_block`	Handing over the next dirty block to the log.
`resolve_blockaddresses`	The log has assinged 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 filesytem 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.

Table 5.2: Callback functions that must be implemented by a dtfs filesystem personality

traditional filesystem that is registered with the respective log requests a commit. A filesystem requests a commit by calling the `log_do_write` function provided by the dtfs core implementation. The log will then start making use of the filesystem callback functions in order to gather all the information needed to form a cluster write from the registered filesystem.

How A Cluster Write is Being Assembled

As already mentioned, the forming of a cluster write can be triggered off by any registered filesystem whenever the log is in the IN_ALLOCATION state. 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 cluster write 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.

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 cluster write into multiple cluster writes 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 way that is transparent for the filesystem personality.

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 cluster write. However, the filesystem that has triggered off the commit has already suggested a commit level to be used. 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 file systems 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 cluster write 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 file system 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 aligned to an eight byte boundary. 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 repeatedly calling the `get_next_block` function. This function hands over a pointer to the `dtfs_buffer` structure for the next dirty block. In order to facilitate the implementation of `get_next_block` for the filesystem personality implementor, the function also takes a pointer to a `dtfs_buffer` structure as an argument. This structure represents the `dtfs_buffer` structure of the dirty block that was obtained by the previous call to get_next_block[1].

Once again a filesystem is required to provide the exact number of dirty blocks that have been returned by a previous call to `num_blocks_dirty` in the current turn. If the log calls `get_next_block` after `num_blocks_dirty` blocks have already been obtained in the current turn, the filesystem personality is required to return NULL.

The filesystem personality can influence the layout of dirty blocks in its current write cluster. The log will try to put the entire write cluster for the filesystem into consecutive physical disk blocks. The dirty block that is handed over by the filesystem first, 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[2]. But 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 underlying device, it will call the function `resolve_blockaddresses`. When this function is called, the `buffer_head` members of all the `dtfs_buffer` structures that have been handed over to the log in this turn by the respective filesystem version have been modified to contain the physical address of the block on the disk.

[1] So when `get_next_block` is invoked for the first time for a traditional filesystem in a turn, it is called with a NULL parameter

[2] 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.

The filesystem personality is expected to perform all its block address translations (from logical addresses to physical addresses) when this function is called.

Furthermore the filesystem can rely on the log calling this function exactly once in every turn for each filesystem version that has reported dirty blocks to be written out in this turn. The filesystem must also fix up the ifile inode data structure if it wants its ifile inode to be written out in this turn. When `resolve_blockaddresses` is called, all the dirty blocks that have been handed over to the log by the filesystem version in this turn are now accessibly by their physical block address using standard Linux buffer cache methods.

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.

Performing the actual write There is no callback being issued to the filesystem personalities in this stage.

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 file system 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.

5.2.4 Common Data Structures

`struct dtfs_buffer`

This data structure is required to share information about a dirty block between the filesystem personality and the dtfs core. It consists of the following data members:

1. `bh`, a pointer to a `buffer_head` structure representing the contents of the dirty block.
2. `blkdesc`, the block description structure for the dirty block that will be placed in the filesystem description by the log representing this block.
3. `wq_next`, a pointer to a `dtfs_buffer` for internal use by the dtfs core implementation.

`struct dtfs_filesystem`

This data structure represents a certain version of a dtfs traditional file system. It must start with the following three data members:

1. `fsys_index`, the filesystem version number of the respective filesystem version (specified at mount time)
2. `pfi`, a pointer to a per-filesystem structure of the dtfs core for the filesystem this entity is a version of. (The filesystem version itself will need this entry in order to be able to modify its filesystem version entry in the filesystem's checkpoint area.)
3. `log_next`, a pointer to a `dtfs_filesystem` structure for internal use with the dtfs core implementation.

Extending The Common Data Structures

A filesystem personality will most likely need additional information to be stored in these data structures. However, this can be accomplished quite easily. The filesystem implementation can encapsulate these data structures into its own. Even casting pointers would be save if the data structures are placed first in the filesystem's data structures. An example of an extension of the `dtfs_buffer` structure might look like this:

```
struct dummy_buffer {
     struct dtfs_buffer b;
     blkaddr_l log_addr;
     void *data;
     struct dummy_buffer *next;
     struct dummy_buffer *hash;
}
```

Chapter 6

In-Memory Data Structures

The design of dtfs's in-memory data structures has been heavily influenced by its ability to place more than one traditional filesystem on one dtfs filesystem and to support multiple versions of every traditional filesystem. This allows the distinction of three different levels of information:

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

   This information consists of the physical device information stored in a kdevice (that also holds a segment usage bitmap displayed as separate entity in figure 4.1 showing the functional dependencies) for the device entity and of more dtfs-specific information stored in a log module.

2. Information about a certain traditional filesystem.

   This information represents things like the location of the checkpoint areas for the file system or the contents of the filesystem's super block. A list of this per-traditional filesystem information 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 file system.

   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 the traditional filesystem version resides on.

Figure 6.1 shows an overview of the relations between the in-memory data structures used by dtfs.

Furthermore 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.

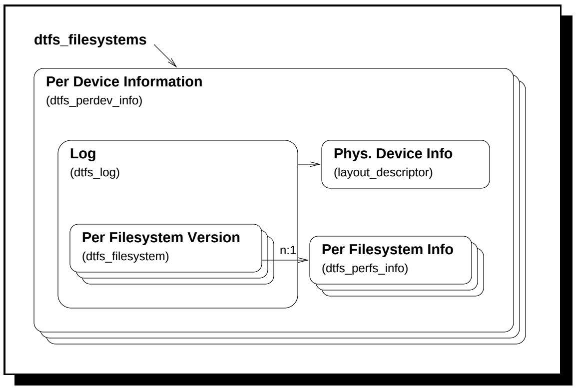


Figure 6.1: In-memory data structures of dtfs. Stacked boxes indicate lists of data strucutres of the same type. Arrows show additional relations required for proper dtfs operation.

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 that list if it cannot be found.

A usage counter is used 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 will be discarded.

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

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

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

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 block of all the file systems 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.

filesystem personality see traditional filesystem

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 of a file. 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.

logical blocksize ext2 perceives a device as an array of linearly addresable 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 file system 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 file system residing on a dtfs partition. Each logical file system on a dtfs partition as its own super block and may be implemented by using different traditional file systems.

logical time The logical time is the way in which dtfs events are timestamped. Every dtfs file system 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.

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 modifiying entities of this data structure. Modules have a well-defined interface to the rest of the implementation and serve as the unit of testing.

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.

segment A segment is a fraction of the disk that can be used for 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 file system created at a certain point in time representing the state of the file system 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 4.2 on page 40 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 *cluster write*. When the log has committed the cluster write all the dirty blocks the filesystems wanted to have placed into the cluster write have been handed over to the underlying Linux block device code and the turn is over.

traditional filesystem Since dtfs is only a generic layer to support a log-structured file system, it is necessary to implement high-level filesystem services on top of this generic layer. Such an implementation of filesystem services is called a traditional file system (since it will most likely be derived from a non-log-structured file system, such as ext2).[1]

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

versioning Versioning is the ability of dtfs to support more than one state of a file system: Every version of a file system has it specific mount point and checkpoint. A version of a file system may be read-only (snapshot) or read/write (clone).

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

[1]Sometimes the term "traditional file system" is also used for any non-log-structured file system. The current ext2 implementation is an example of a traditional file system in that context. However, the meaning should be evident from the context.

Appendix B

Changes

$Log: dtfs-disklayout.lyx,v $Revision 1.6 1998/08/27 15:57:43 ceci

+) Changed LaTeX class, polished up layout

+) added next_segment and last_segment su to segusage header

+) added part about implementation

+) run chktex on it

+) Namechange of documentRevision

1.5 1998/07/13 08:40:31 ceci:

Added MIDT value to super_block

num_inodes and num_free_inodes added to checkpoint entry (fixed a design flaw)

Introduction of flushed checkpoint

rotated figures

lots of typos fixed (mw)

Revision 1.4 1998/06/04 10:03:12 ceci:

Converted document in a design specification

another major overhaul of the on-disk data structures resulting in an eradication of partial segment headers, re-worked checkpoint blocks

introduced 64bit data types...

Revision 1.3 1998/05/23 16:33:15 ceci:

major update adding chapters on checkpoints, ext2 personality and filesystem mapped meta-data. General overhaul of the document

Revision 1.2 1998/04/01 09:16:54 ceci: Updated chapter on versioning (removed version_usage bitmap)

Revision 1.1 1998/03/30 17:45:59 ceci: Initial revision

Bibliography

[SELTZER] Margo Seltzer et. al: An Implementation of a Log-Structured File System for UNIX. Proceedings of the 1993 Winter USENIX Technical Conference

[ROSENBLUM] Mendel Rosenblum: The Design and Implementation of a Log-Structured File System. Diss. 1992

[MERGEMEM] Philipp Richter, Philipp Reisner: The mergemem Project. Practial Study at the TU Wien: http://www.complang.tuwien.ac.at/ulrich/mergemem/

[SIPRALOG1] James E. Johnson, William A. Laing: Overview of the Spiralog File System. Digital Technical Journal Volume 8, Issue 2 (Oct. 1996)

[SPIRALOG2] Christopher Whitaker, J. Stuart Bayley, Rod D. W. Widdowson : Design of the Server for the Spiralog File System. Digital Technical Journal Volume 8, Issuce 2 (Oct. 1996)

[RFC1014] Sun Microsystems Network Working Group: Request for Comments #1014, External Data Representation Standard (XDR),. June 1987

[KHG] The Linux Kernel Hacker's Guide

[RIO] Peter M. Cheng, Wee Teck Ng, et. al: The Rio File Cache: Surviving Operating System Crashes (ACM — Communications?)

[RUBINI] Alessandro Rubini: Linux Device Drivers. O'Reilly & Associates Inc. 1st. edition, 1998. ISBN 1-56592-292-1

[LKP] Michael Beck, Martin Böhme, Mirko Dziadzka et.al: Linux-Kernel-Programmierung Algorithmen und Strukturen der Version 2.0. Addison-Wesley, 4. aktualisierte und erweiterte Auflage, 1997. ISBN: 3-8273-1144-6

— bibliography is still incomplete and not BibTEX'ed —