Implementing `dtfs`

Christian Czezatke

March 26, 1998

Abstract

This paper is a first implementation outline for `dtfs`, a log-structured filesystem implementation for Linux. It is based on *LFS for Linux – An Outline*. This document is a proposal on how an LFS for Linux can be implemented, but it does not (yet) define LFS blocks and data structures in all detail.

Contents

1 Basic Design

1.1 Why There Are Differences To My Previous Approach

After having worked on the design of an LFS for some time, I've arrived at a slightly different design for an LFS implementation than in my conclusion in [LFSOUT]. The main reasons for the design overhaul were the following:

- I've now access to more adequate documentation on the Linux 2.0.x kernel internals.
- Having a closer look at the Linux kernel implementation revieled that some assumptions made in [LFSOUT] are not appropriate.
- Too much emphasis has been given to memory management and flushing issues

In [LFSOUT] It́ve outlined three different implementation scenarios for an LFS for Linux:

1. Adding callbacks for writes to the VFS
2. Implementing a separate cache for LFS writes
3. Adding LFS pseudo-Block devices

In BLAH, the conclusion was made that the addition of LFS pseudo-block devices would be the easiest way to integrate LFS into the Linux kernel. This decision was influenced by the fact that many considerations in [LFSOUT] are centered around the issues of interfacing with the Linux memory management and the bdflush daemon. However, having a closer look at current Linux kernel implementations reveals that these interfaces are both rather small:

1.1.1 Dirty Buffers and Memory Shortage

The key function for Linux memory management issues (as far as memory shortage is concerned) is `try_to_free_page`. This function attempts to either discard buffer cache entries, pages from processes or shared memory. Examining the code of `try_to_free_buffer` (which is responsible for freeing a certain buffer cache entry) shows that only unchanged buffer cache entries may get discarded in case of memory shortage anyway. So dirty buffer cache entries are never touched in case of a memory shortage. Therefore it is useless to deal with complex algorithms in order to integrate dirty LFS blocks into the standard buffer cache just to integrate cleanly into the buffer cache-related memory management issues of the Linux kernel.

1.1.2 bdflush

Similar to the dirty buffer management in Linux, the interface of bdflush to the buffer cache is a rather minimalistic one, too. The bdflush process basically traverses the buffers in the buffer cache and writes out every buffer that is dirty. Specia treatment must be given to buffers belonging to a loopback device.

Similar to the special treatment that is granted to the loopback device blocks, there would have to be special treatment for blocks belonging to LFS pseudo block devices too. So implementing LFS pseudo block devices would not be possible without altering other kernel code anyhow.[1] Furhtermore, having the building up of a segment being triggered off by a single block write operation would result in a rather inefficient implementation (gathering all blocks that should go into a partial segment from the buffer cache) of LFS writes.

1.1.3 sync_device

Similar to bdflush, a sync request for a certain device is implemented as a traversal of the buffer cache data structures that writes out all dirty blocks of a certain device. Again this can also be achieved by calling a function that flushes out all unwritten partial segments for a LFS residing on the respective block device.

1.2 Current Implementation Outline

Figure 1 shows an outline of the current implementation outline for LFS/Linux. Most of the functinality that has been in the LFS pseudo block devices is now in the Address Translation Layer, ATL.

1.2.1 FAL – The Filesystem Adaption Layer

Unfortunately traditional file systems cannot be integrated into a log-strucutred framework without some changes to the file system implementation. The following parts of a conventional file system will have to be altered:

free space management There is no need for a traditional file system to maintain bitmaps of free blocks or bitmaps of free inode numbers, since the file system can only deal with logical block numbers, but not with

[1] Since bdflush has to be changed anyhow, it is easier and more convenient to add a one-liner at the end of this function that causes all partial segments being older than x seconds to be flushed out.

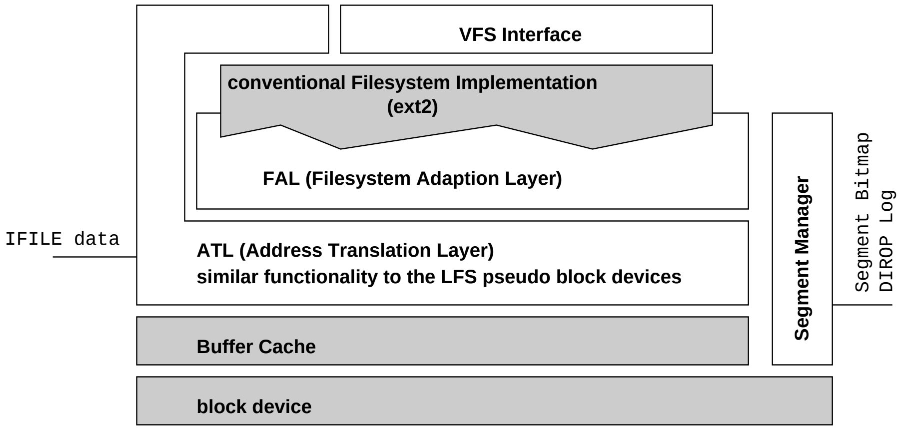

Figure 1: Architecture for the `dtfs` implementation.

physical block numbers. The traditional file system must request free inode numbers and free blocks by using functions provided by the LFS framework.

write operations Write operations need to be performed by means of the LFS framework because the LFS framework is responsible for gathering write operations and placing them into partial segments. Furthermore, block number fixups will be required before writing partial segments to the block device.

directory operation logs In oder to meet the database-like guarantees that an LFS can provide, it is necessary to maintain a directory operations log since directory altering operations are not atomic. — When a new file has to be added to a directory, for example, the inode block that holds the inode representing the new file and the directory data of the directory containing the file have both to be altered. The directory operations log is updated and written to a partial segment after both operations have already been written to the device. Writing out the directory log information serves as a commit of such a non-atomic directory-altering operation.

Since conventional file systems have no such transaction-like semantics, calls that update the directory operations log must be added to the filesystem implementation when a directory has been altered.

In oder to fulfill these requirements, it is necessary to alter the implementation of the conventional file system in order to get it to work within a log-structured environment.

1.2.2 ATL – The Address Translation Layer

The ATL is an abstraction layer that decouples the file system from the physical location of a block on the device by mapping logical block addresses (as used by the conventional filesystem and the FAL) to physical block addresses. This is necessary because of the special way that LFS uses to assign physical block locations to a data block. The ATL uses standard buffer cache mechanisms in order to retrieve blocks from the block device.[2] The ATL performs writes by using services provided by the Segment Manager. Furthermore, the ATL is also responsible for managing inodes and for maintaining the `IFILE`.

The main task of the ATL is to provide a "virtual block address space". – This virtual block address space represents an address space with fixed data block locations on which traditional file system operations can be performed. Furthermore, this allows a partition of the block address space that does not have to take the physical locations of blocks into account, but facilitates certain tasks, such as inode-to-block translation. The virtual block addressing scheme is outlined in 1.2.4.

1.2.3 SEGMGR – The Segment Manager

The SEGMGR gathers dirty blocks and arranges them into partial segments. Furthermore, the SEGMGR is also responsible for the actual physical block allocation: When conventional file systems consult their block usage bitmaps in oder to find a new block on the device that is not yet in use, an LFS framework uses the SEGMGR to obtain a free block. The SEGMGR returns a free block of the requested type within the virtual block address space, but it also assures that the block can actually be written to the device (i.e. the device is not already full) should the need arise.

The Segment Manager also maintains a directory log that records all the non-atomic changes to directory structures within the respective segment.

In addition to that, the Segment Manager holds a segment bitmap that indicates segments on the file system that are clean and can be used for writing partial segments into them.

Furthermore the segment manager provides calls to write some or all partial segments that are currently in memory to the device. This functionality is needed in oder to be able to implement the various sync_ calls and to provide the required functionality for the bdflush system call implementation.

1.2.4 The Virtual Block Address Space

Since the LFS framework delays the actual assignment of a data block to a physical block on the device, a virtual address space has to be provided that allows the conventional file system residing on top of an LFS layer to address blocks that have not yet been written to the device and therefore have not been

[2] So reads from the device are handled in the same way as reads performed by a conventional file system: They are done by using the Linux buffer cache.

assigned a to physical block address. Furthermore, the introduction of such a virtual block address space allows to distinguish between the various block types that a file system has to deal with[3] by just having a look at the virtual address of the respective block.

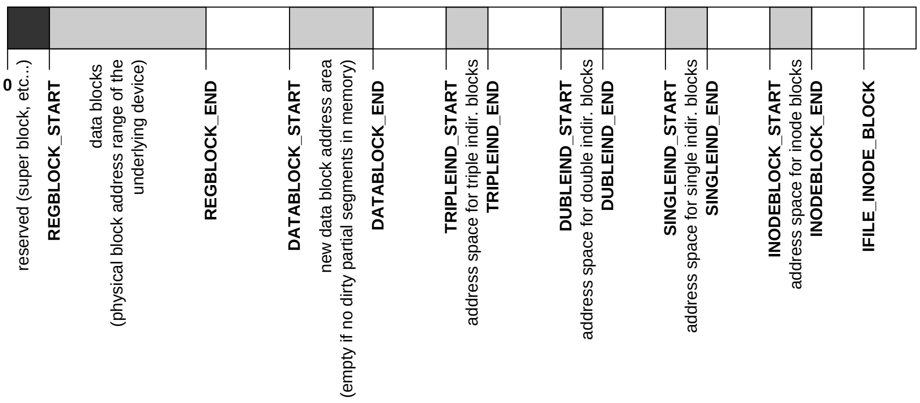

Figure 2: The partitioning of the `dtfs` Virtual Block Address Space

Figure 2 shows the partitioning of the virtual block address space.This figure shows that each type of block has its own, distinct range within the address space. The way in which the block types are ordered makes it easier to build up a partial segment: The segment handler just has to order the blocks to be written in a partial segment in ascending logical block number.[4][5]

2 Interfaces Provided by the `dtfs` Components

2.1 FAL (Filesystem Adaption Layer) Interface

The code that can be found in the FAL is specific to the file system that resides on top of the LFS-layer. Therefore it is not possible to specify a generic interface that is provided to the conventional file system implementation by the FAL.

2.2 ATL (Address Translation Layer) Interface

The following interfaces must be provided by the ATL:

[3] single, double, triple indirect blocks, inode blocks, data blocks

[4] However, additional treatment of blocks within the valid physical block numbers should be added in oder to ensure that contignuous data blocks of a file are written contignuousely.

[5] Additional address space partitioning could be added in oder to be able to differentiate between blocks that have been re-written and blocks that have not been altered.

2.2.1 ATL Interfaces for Use With the FAL (Filesystem Adaption Layer)

struct dtfs_super_block *atl_read_super(...) This call instructs the address translation layer to read in its super block. The parameters are similar to the ones provided to a read_super call from the VFS and must be specified in more detail in the future.

inode_id atl_get_free_inode(inode_id parent) This function returns a free inode number. For optimisation reasons, the inode number of the directory that should hold the new file or directory for which an inode is required is passed as an argument. The ATL should try to get a free inode in the vicinity of other inodes in this directory in oder to achieve a locality of reference for `IFILE` accesses.

void atl_free_inode(inode_id inode) Indicates that the inode with the numeric id identified by the parameter is not in use anymore. (Some sanity checks could be added in the implementation of this function for debugging reasons. For example, if the inode being freed belongs to a directory, the directory must be empty).

struct dtfs_block *atl_read_block(ulong blocknum) Performs a block read. The block to be read will first be searched in the Segment Manager (it could be dirty). – If this search fails, the block will be obtained by using a standard buffer cache `bread` call. — It might be appropriate to allow more than one block to be read at once, as it can be done with the `bread` call too.

several sync calls The ATL must also support sync calls arriving through the VFS. The detailled list of calls to be provided is not yet specified. But it can be expected that these functions will mainly be a wrapper around some Segment Manager functions.

2.2.2 ATL Interfaces for Usw With The SEGMGR

void atl_update_ifile(ulong log_blknum, ulong phys_blknum) This call is used by the Segment Manager in oder to keep the IFILE up-to-date: When the SEGMGR performs a segment write, it also fixes up block refrences in indirect nodes and assigns phisical block numbers to inode blocks. The translation between logical inode block numbers (derived from the inode number itself) and the phyiscal location on the device is done by a lookup in the IFILE. Therefore the IFILE must be updated in order to represent the correct physical location of an inode block after it has been written out.

Furthermore, this call also informs the ATL that the respective inode block has eventually been assigned to a physical location on the device and has been written out. – This is important since the IFILE itself can only be written to disk if all inode blocks within the IFILE are assigned to physical blocks on the device or inconsistencies will occur should the system crash.

char atl_ifile_ready() Returns true if the IFILE is in a consistent state so that it can be written out to the device. (It might be necessary to lock the IFILE too…)

2.3 SEGMGR (Segment Manager) Interface

2.3.1 SEGMGR Interfaces for Use With the FAL

void segmgr_dirop_commit(char type, inode_id parent, inode_id child) Informs the SEGMGR of a completion of a directory manipulation operation that has not been atomic. This document does not yet give a complete listing of these operations, but they can be found in [ROS92]. (Maye we will need a dirop_start operation too?)

struct dtfs_block *segmgr_get_free_block(int type) Returns a new, unused block of the requested type. Valid types are `DATABLOCK, SINGLE_IND, DOUBLE_IND, TRIPLE_IND, INODE_BLOCK`. A return value of NULL indicates that the request cannot be fulfilled due to insufficient free storage on the block device. On the other hand, a return value != NULL can be interpreted as a guarantee that the block will actually fit on the underlying device.

void segmgr_release_block(ulong blocknum) Indicates that a dirty block can be released and does not have to be written out to the device. This could happen if a file is deleted or truncated before it has been written out to the device. The Segment Manager will then discard the block and update its free blocks statistics accordingly.

2.3.2 SEGMGR Interfaces for Use With the ATL

void segmgr_init_segment_map(char *seguse_bmap, ulong segcount) This function initialises the Segment Manager. The parameters specify the segment usage bitmap and the total number of segments within the file system.[6]

struct dtfs_block *segmgr_read_block(ulong blocknum) Searches for the block specified with the logical block number `blocknum` in the Segment Manager. – If the search is succesful, this indicates that the block is dirty (since only dirty blocks will be in the Segment Manager).

void segmgr_write_block(struct dtfs_block *block) Queues the specified block for writing out in a partial segment. When implementing this function care must be taken: The block could already be in the Segment Manager (if it has been written, but not yet been flushed to the device and is altered again).

struct dtfs_block *segmgr_new_inode_block(inode_id inode) *This function is optional!* This function would allow to increase the size of the IFILE at run-time by adding new inode blocks and thus increasing the total number of inodes in the file system. Since the inode number determines the virtual address of the block it resides in, the segmgr_get_free_block call cannot be used since this call does not allow to control which logical block number will actually be assigned to the block to be allocated.

2.3.3 SEGMGR Interfaces for Various Tasks

void segmgr_sync(...) Call to cause the Segment Manager to perform the actual writes to the device. Various options should be supported. The following sync–semantics should be possible:

- Flush out just one segment (the oldest segment?) – `try_to_free_*` semantics
- Flush out all segments that are older than a certain threshold (and possibly create a checkpoint) – bdflush semantics
- Make sure that all unsaved blocks of a file are actually written ot the device – `sync_file` semantics
- Write all segments to disk and create a checkpoint – `flush_device` semantics.
- Synchronous and Asynchronous syncs should be possible...

calls to manipulate the segusage bitmap This is necessary for the implementation of a cleaner (and might be turned into a syscall?)

3 `dtfs` Scenarios

This section will discuss a few common filesystem operations and show how these tasks will be performed within a log-structured file-sytem environment. The goal of this section is to get an idea of how the modules the dtfs implementation consists of actually work together in oder to get things done.

3.1 Creating a `dtfs`: `mkedtfs`

Creating a dtfs on an empty partition requires the following tasks to be performed:

[6] When taking into account that kmalloc can obtain contignuous memory areas <122KB only, this limits the size of the file sytem to approx. 450GB when using a segment size of 0.5MB. – However, there are kernel patches for allocating bigger chunks of memory floating around, should the need arise...

- Create the dtfs superblock(s)
- Create the traditinal filesystem's superblock(s)
- Partition the device into segments and mark them as empty
- Create the root directory entries[7]
- Create the IFILE and the SEGUSAGE file in the root directory
- CREATE the lost+found directory in the root directory.

3.2 Mounting a `dtfs` File Syste

Mounting adtfs file system will result in the VFS call `read_super` to be issued by the VFS layer. Of course it is not sufficient to simply call read_super as it has been implemented by the traditional file system residing on top of the log-structured framework. Mounting a dtfs File System will require the following steps:[8]

At first, the dtfs super block must be located. This can easily be done, for example, by making a fixed physical block on the device be the super block. The dtfs super block will hold the following information:

- blocksize, size of filesystem in blocks, size of segment in blocks
- location of checkpoint areas
- location of the traditional filesystem's super block

At first, some general data structures are initialized holding the number of segments etc... Then the checkpoint areas listed in the super block are examined in search for the checkpoint with the highest timestamp (This value will also be the highest timestamp in use on the file system.)[9]

The checkpoint area will basically hold the following information:

- The location of the last partial segment that holds a checkpoint.

Since we have now got a pointer to a valid segment (it must be valid since the checkpoint area will be updated after the checkpoint has been written. Furthermore we must have a matching timestamp and a correct header for this segment) that holds a checkpoint information, we can now obtain the physical location of the IFILE inode block from this partial segment.[10] Furthermore, the

[7]A special inode number will be assigned to the root directory

[8]implemented in `atl_read_super`

[9]Actually, it must not really be a timestamp that is used; some kind of a "unique key" semantics is sufficient, but the unique keys must be assigned in ascending order.

[10]If a segment contains a checkpoint, the IFILE inode block will be at a fixed location within the partial segment's header.

segment usage bitmap can be obtained, too and will be passed to the Segment Manager.[11]

After having found the IFILE inode block, it is possible to perform read requests issued by the FAL, so the next step is to call the read_super routine of the traditional file system and passing the location of the traditional filesystem's superblock along. This routine will (by using the fixed inode number of the root directory) then in turn read in the root directory inode as required by the VFS `read_super` specification.

3.3 Reading a File

The log-structured layer does not interfere very much with read operations. The only peculiarity that is specific for LFS implementations is the process of locating the device block that holds a specific inode. The traditional file system can easily calculate the virtual block address of the inode block. The mapping to the actual device block is done by the FAL using the IFILE.[12]

3.4 Adding a File to a Directory or Making a Hard Link

This operation requires two steps to be performed:

1. Update the directory information in the parent directory
2. Add or modify the inode representing the file

In both cases the traditional file system will determine a directory data block into which the new file information will be written. This is done by one or more calls to `atl_read_block`. The modified buffer is then written to the current partial segment by using `segmgr_write_block`. After that the block containing the inode will be read in and modified in the same way (in the case of adding a hard link).

When a new file has to be created, writing the directory information block out with `segmgr_write_block` has to be delayed until a calll to `atl_get_free_inode` has successfully returned a new inode number for the new file. After that the block containing this newly allocated inode number will be read in, modified and written out in just the same way as this has been done with the directory data block.

After the successful completion of these operations, a call to `segmgr_dirop_commit` is required in oder to indicate that both changes required to the filesystem to make the requested change have been completed.

[11] be it implemented as a bitmap or as a special file, like the IFILE.

[12] So a read access might cause another read access to the IFILE before it can actually be completed.

4 Starting a dtfs implementation

These are the next steps planned in the implementation of dtfs:

1. Write a `mkedtfs` tool
2. Get the `dtfs` to read its own root directory (requires parts of the FAL and the ATL, but not the SEGMGR to be implemented)

References

[ROS92] The Design and Implementation of a Log-structured File System, Mendel Rosenblum

[LFSOUT] LFS for Linux – An Outline, Christian Czezatke