LFS for Linux – An Outline

Christian Czezatke

March 11, 1998

Contents

1 Overview

This is just a summary of my thoughts about how to implement a log-structured filesystem for Linux. Future versions of this document should present a more in depth-discussion of Linux-related LFS implementation issues.

Since there are LFS implementations already existing that can at least be used as a basis for Linux/LFS,[1] I haven't focussed too much on LFS design issues for now, rather on how to integrate an LFS file system into the existing Linux 2.0 kernel implementations at all.

[1] Sprite and 4.4BSD

2 Linux VFS and the Buffer Cache

2.1 The role of the Buffer Cache in Linux

Current implementations of Linux file systems are embedded in the framework that is outlined in figure 1.In order to avoid performance problems when ac-

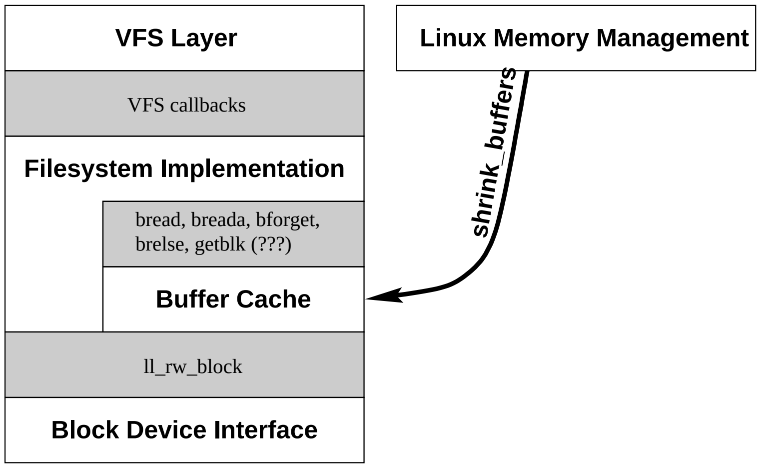

Figure 1: Linux File System interfaces

cessing the data of the block device a filesystem actually resides on, accesses to blocks are done by using the buffer cache mechanism. The buffer cache tries to keep frequently referenced blocks from block devices in the main memory in order to speed up access. Current Linux file systems perform all their read accesses to a device by using the buffer cache interface.

The functions provided by the buffer cache for these means are mainly `bread, breada, bforget`. When the file system tries to read a block (by using the `bread`–call, for example) and the block has already been read in, no device I/O needs to be performed. If the block is not resident in main memory, the buffer cache uses the block device function `ll_rw_block` in order to actually read in the requested data. After that, the newly read in block is kept in main memory just like the other blocks currently stored in the buffer cache.

2.2 Writes to Block Devices

2.2.1 Filesystem-Induced Writes

Generally, writes to a block device are also done asynchronuosly by using the buffer cache. A block in the buffer cache can be changed in-memory and marked as "dirty" afterwards. However, current file systems are often implemented in a way that requires them to update metadata information synchronousely for consistency reasons. In order to achieve this, the filesystem implementation has the following two opportunities:

- Write the data "by hand", i.e: use `ll_rw_block` directly in order to perform the write. (And do not mark the block as dirty in the buffer cache, of course.)
- Do a `sync_buffers` for the device the file system resides on. This results in all dirty blocks of the device being written out.

Since the latter approach cannot assure that metadata blocks are actually written out before the data blocks, it is not really feasible for a file system implementation that needs the meta data information being written out to the device first for consistency reasons. Furthermore, performance would probably suffer greatly because `sync_buffers` flushes out *every* buffer that is currently marked dirty for the respective device.

2.2.2 Writes Triggered off by the Memory Manager

Writes to the block device may also occur when the buffer cache must be shrunk. This might happen when the memory manager calls `shrink_buffers` in order to free some main memory. This case is handled entirely by the Linux buffer cache implementation and the file system cannot interfere with such writes at all.

2.3 What's the Problem With Writes At All?

2.3.1 Changing Data Immediately Before a Write

An LFS file system needs to be able to investigate and change the data to be written to the device immediately before the write to the device occurs. This is necessary because in an LFS implementation the assignment of a data block to a physical block on the device can generally only be done at the time the write to the hardware device actually occurs. Furthermore, this results in the necessity of fixing up block addresses in inode and indirect blocks at that time.

2.3.2 Controlling the Device Access Pattern

One key point in the LFS approach is to do disk transfers in large chunks so that seeks are virtually eliminated. Therefore an LFS must have some control over the sequence of writes that a device actually performs. The current Linux buffer cache implementation does not take care of the sequence of device accesses. It relies on the device driver's ability to do scather/gather I/O.[2] For an LFS implementation it would be beneficial if the LFS could assemble a segment in

[2] At least according to what I've found out so far.

memory and write it out with just one I/O operation that should ideally be atomic.

3 Implementing an LFS for Linux

In order to be able to implement an LFS for Linux, the general framework as shown in figure 1 has to be altered in order to meet the special needs of a log-structured file system. As already mentioned in this paper, the actual problem is the behavior of the buffer cache for writes since there is no way for the file system to get involved when the actual write to the device takes place.[3] Furthermore, another caching policy could maybe also be beneficial since LFS tries to do I/O in large chunks only.

One of the solutions given in this chapter could be chosen in order to overcome problems like that.

3.1 Solution 1: Filesystem Callbacks for Writes

One approach to overcome the problem of the filesystem not being able to interfere with actual device writes would be to provide callbacks for each file system that do the actual device I/O. For conventional file systems these callbacks would simply be forwarded to `ll_rw_block`. An LFS, however, could catch such a request and do the necessary computations to form a partial segment to be written out. This approach seems to have been taken by the `VOP_` operations in the 4.4BSD log-structured file system implementation.

In order to implement such a callback in Linux, an appropriate pointer to a callback function for a file system could be added to `struct file_system_type`. When the buffer cache needs to write a modified block to a device that contains an LFS, it does not directly call `ll_rw_block`, but uses the callback to invoke a file system specific method. Traditional file systems may choose to simply call `ll_rw_block` (which could be provided as a default implementation). LFS will in that case start forming a partial segment and write that partial segment to the harddisk.

Since every buffer has an entry that identifies the device it belongs to, it should be possible to use that information to find out which file system resides on the block device. If the block device is not associated to a file system at all, a call to `ll_rw_block` will be made.[4]

An approach like this would lead to the architecture depicted in figure 2.Please note that this does not mean that writes are generally uncached. It

[3]This is at least true for writes due to `shrink_buffers` or maybe by `sync_device` calls. Sync–Calls caused by an application calling `fsync` can be handled by the LFS because these requests are passed on by means of the VFS interface.

[4]Will blocks of block devices that are not accessed via file system routines be in the buffer cache at all?

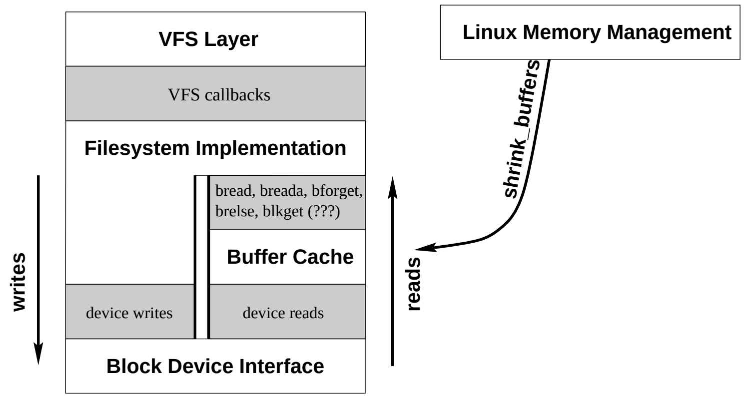

Figure 2: Buffer Cache and VFS with callback for writes

just means that writes to the device always go through the file system implementation.

However, this approach has a few disadvantages:

- Minimal changes to all file system implementations are required.

- Write operations could probably have a considerable overhead. (Buffer Cache determining the file system responsible for a buffer, invoking a file system method, etc. . .)

- Possible kernel deadlocks: `try_to_free_page` calling `get_free_page`, `get_free_page` invokes `shrink_buffers` which causes a file system callback to be activated. If that file system callback tries to allocate memory, but none is available, `try_to_free_page` will be called, which will call `get_free_page etc`. . . — Therefore such file system callbacks must not be allowed to allocate memory.

- kernel code that is known to be stable has to be changed resulting in a possible loss of reliability even if no LFS is used.

3.2 Solution 2: Special pseudo block-devices for LFS

In this approach the problem of LFS writes would be addressed by adding special-purpose pseudo-block devices to the Linux kernel. There are some special purpose pseudo-block devices in the kernel already, like the “multiple devices” device and the loopback device.

In this scenario LFS file systems are only allowed to reside on such LFS pseudo-block devices. Each LFS pseudo-block device will in turn be associated with a "real" device, like a hard disk partition.[5]

The buffer cache interface will remain unchanged in this implementation variant and remain as depicted in figure 1. But when the buffer cache tries to write a block to a device that holds an LFS file system, the buffer cache will end up forwarding this request to one of the LFS pseudo block-devices.

This LFS pseudo block-device will now take care of building up a partial segment for writing to the hard disk. This involves all these things like re-ordering blocks, assigning physical block addresses to data blocks, fixing up inodes and indirect blocks, updating the `ilist`, etc...

This approach seems to have the following disadvantages:

- LFS pseudo block-devices could make it a bit trickier to actually set up an LFS file system.

- LFS pseudo block-devices could need some overhaul in case the Linux core-developers decide to re-design the "physical device to logical device relationship" (see physical vs. logical volumes on other systems, for example). — But on the other hand the introduction of such LFS pseudo block-devices could also mark the start of a rework in that area...

- The fact that one `ll_rw_block` call results in more than one block being actually written to disk could cause some problems in kernel code that does not expect things like this to happen. (But it should probably be rather easy to locate such areas of possible problems.)

However, this approach also has some interesting advantages over other approaches:

- No changes to existing (and known to be stable) code are required.

- Performance impacts (if there are some at all) are only limited to LFS, since other file systems do not use these pseudo block-devices.

- Implementing the LFS functionality would be split up in two distinct parts with a well-defined interface easing the (parallel) implementation of LFS for Linux.

- The introduction of a more flexible disk management for Linux (as found in other Unix systems) could easily be started by extending such LFS pseudo block-devices.

[5]Maybe this could also be extended in the future: like a pseudo LFS block device spanning over more than one hard disk partition, or even into some physical/logical volume management as it can be found in AIX, for example.

Also some more exotic things would be possible, like placing an LFS on a DAT drive...

3.3 Solution 3: A Separate Cache for LFS

In order to satisfy the LFS–specific block device write demands, the Linux buffer cache architecture could be changed to reflect the differences between traditional filesystems' needs and the requirements of an LFS implementation. This would result in a buffer-cache architecture that looks like the scenario in figure 3.

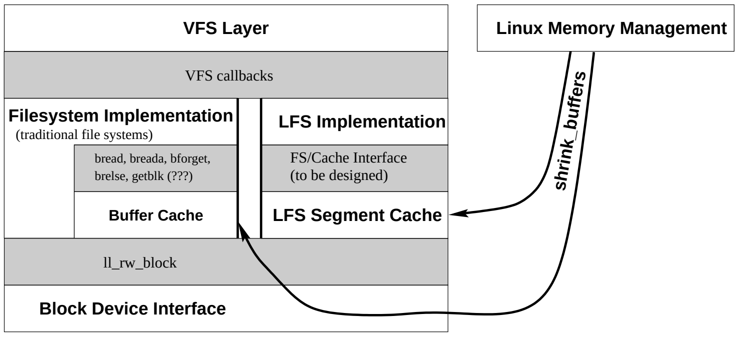

Figure 3: Separate Caches for traditional and log-structured file systems

In this scenario there are two separate caches: One for log-structured file systems and another one for traditional file systems. The implementation of the "segment cache" takes care of caching read and write operations for log-structured file systems. This would require the modification of the `shrink_buffers` function so that it deals with both the buffer cache and the segment cache.

Again, this approach has some drawbacks:

- This scenario requires a vital change to the memory management of the Linux kernel. These routines are generally very well tuned in order to achieve high performance. A change in this area would probably degrade overall system throughput.
- The implementation of `shrink_buffers` could become substantially more complicated when having to consider two different block device caches.
- There might be scenarios in which both the buffer cache and the segment cache are competing for main memory resulting in poor cache hit rates for both the block cache and the segment cache.
- Implementing two separate caches would probably result in a lot of duplicated code.
- Improvements to the buffer cache management (such as improved policies for deciding which buffers to discard in case of main memory shortage) are

not immediately available to the LFS implementation. This would generate additional maintenance costs for porting buffer cache improvements over to the LFS segment cache.

An advantage of this approach would be:

- The segment cache could be fine-tuned to the different read/write and caching strategies required for an efficient implementation of an LFS.

4 Conclusion

In order to implement an LFS within the Linux kernel, the underlying mechanisms for the file system implementations have to be altered. From the three different implementation scenarios outlined in this document, I'm currently favoring Scenario 2 (which uses the LFS pseudo block-devices) because it requires no changes to existing source code, allows structuring the implementation in two different modules and allows the future extension of Linux's physical to logical device mapping.