The *LARD* file system

Lazy Arrangements, Real Deficiencies

Mikael Heino

July 18, 2019

1 Introduction

LARDfs (Lazy Arrangements, Real Deficiencies) is a barebones file system. The goal is to provide the most basic interface for creating and reading files and directories. To achieve this goal a very naive approach is taken to cut corners when it comes to design. This means performance and efficiency take the back seat.

1.1 Motivation

Originally I wanted to create a simple read-only FAT driver, but since that is mostly just copying existing code I decided to try something more ambitious. The name I naturally wanted to refer to FAT, but worse. Inspiration has been drawn from all over the place, as well some own ideas, which may or may not already be in use somewhere already. I just don't know about them because I haven't done enough research.

1.2 How to use

If you are lucky, you can just issue *make* to build everything, or build the tools and driver manually. The driver sources are under the *fs* directory under the main project directory and the user space tools are under *tools* directory. To compile the driver you must acquire the headers for your current kernel.
Compile the module:

```
$ make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

If you wish to build a debug driver, add the make variable -ODEBUG
Load the module:

```
$ sudo insmod lard.ko
```

Create a filesystem image:

```
$ touch lard_volume.img; truncate -s 128M lard_volume.img; mklardfs
lard_volume.img
```

Mount the volume (ensure available loop devices are available unless using a real block device):

```
$ sudo mount -t lard -o loop ./lard_volume.img /mnt/testrun
```

Errors are reported in dmesg prefixed with '[LARD Error]: message'.

2 Specification

The on-disk layout is intentionally kept as simple and convenient as possible. The disk is split into 4096 byte sized BLOBs of LARD (or clusters as used in other file systems). BLOBs themselves are identified and their allocation status is tracked using a special BLOBMAP. A single bit corresponds to a BLOB number on the disk, with a special zone marked into the boot sector and superblock structures, so we don't try to allocate BLOBs for anything else from there.
The disk has three 'special' zones. The first zone is the boot sector and necessary superblock structures that need to be stored on disk. The second zone is the special BLOBMAP which is used to allocate and track individual BLOBs. The third zone is the BLOB zone, where the files and data is stored.
Each allocated BLOB has a special 'BLOB Header' which specifies what type of (meta)data is associated with that particular BLOB. This allows us to explicitly find any arbitrary BLOB from any BLOB by just pointing to indices in the BLOBMAP.

BLOB types

```
typedef uint8_t BLOB_TYPE
BLOB_TYPE DATA 0x0
BLOB_TYPE FILE 0x1
BLOB_TYPE DIR 0x2
```

BLOB Header

```
BLOB_TYPE type
BMI parent
BMI child
```

Each BLOB must contain as little metadata as possible, so we have as much space for the payload. The payload will normally be mostly file data anyway. Metadata also adds complexity and we want to stay away from that. Sane filesystems would not even consider adding headers to chunks of data. Files themselves have an offset attribute to the first BLOB number where the data starts, and subsequent data BLOBs point to the next data BLOB.
Fragmentation is probably going to be an issue. Possible solutions are:
#1 - Online defragmentation daemon (really complicated to implement)
#2 - Offline defragmentation tool that is run before mounting. (possible to implement)
#3 - Preallocate like a madmachine, then free preallocations if they aren't needed. (feasible to implement)

If you aren't familiar with the severity of fragmentation, a fragmented file system will become almost completely unusable due to the slowness.

2.1 Boot Sector and Superblock

The superblock structure has all the data relevant to the volume: sector size, volume size and root inode.

2.2 Inode handling

Inodes are essentially entries in the BLOBMAP. Each BLOBMAP bit corresponds to an inode, which can be either a FILE inode, DIR inode or DATA inode.

File and directory lookup begins from the root directory. This may change, however bigger file systems with deep directory structures will become almost unbearably slow to use. Countering this could be solved with a cache of recently used directories, since traversing directories works both ways. You can have a reference number to a dir blob located somewhere deep in the file system, seek to it, and start walking the directories deeper.

2.2.1 File blob

BLOB_HEADER
---
char name[MAX_NAMELEN]
uint32_t allocated_space
BLOB_NUMBER data
void data

2.2.2 Directory blob

BLOB_HEADER
---
char name[MAX_NAMELEN]
uint32_t num_entries
BLOB_NUMBER directory_contents[112]

2.2.3 Data blob

BLOB_HEADER
---
void data

2.3 Failsafety strategy

Every BLOB header specifies a parent and a child. This allows us to determine if a BLOB is orphaned, by recursively traversing back and forth using the parent and child entries.