CSE134 Assignment 4 Design Doc

Group ABAJ – Alexander Sung, Benson Ho, Aidan Sojourner, James Quiaoit

June 5, 2021

1 Introduction

This is the design document for Assignment 4 (`ddfs`) for CSE 134, Spring 2021, at the University of California, Santa Cruz.

1.1 Goals

`ddfs` is a deduplicating in-kernel file system for FreeBSD that operates similarly to the Berkeley Fast Filesystem (FFS/UFS2).

In particular, `ddfs` has the following goals:

- Deduplication means that there's *exactly one* copy of any given data block in the file system:
  - Each 4KiB data block in `ddfs` is referred to by exactly one 160-bit hash value (created using `sha-1`).
  - Files which have identical contents will always refer to the same block.
  - Usage of the blocks is tracked with a reference count. Blocks are not freed until the reference count reaches 0.
- `ddfs` supports the following functionality:
  - Read, write, truncate, stat of files
  - Create and unlink of files
  - Permissions
  - Timestamps
  - Directories
  - `fsync`
- `ddfs` block pointers are 20 bytes (160 bits) instead of 8 bytes, and limiting the total number of block pointers to 6. (3 direct, 1 single indirect, 1 double indirect, 1 triple indirect).
  - A `ddfs` file is limited to a maximum of 8531487 blocks, or approximately 32 GiB per file (when using 4KiB blocks).

Additionally, a tool `newfs-ddfs` was created which allows the user to format a disk device with the `ddfs` filesystem.

Finally, a tool `statddfs` was also created. When given a `ddfs` partition on a disk, `statddfs` will compute the total amount of space saved from the deduplication.

2 Design

One of our goals with `ddfs` was to leverage as much existing code from FreeBSD FFS as possible. To that end, our group started with a copy of the FFS/UFS2 source code, and made changes in specific places in order to add deduplication. We tried hard to modify as little of the complex FFS codebase as possible.

For more info on the consequences of this design decision, see the Modifying FFS section.

2.1 Disk Layout

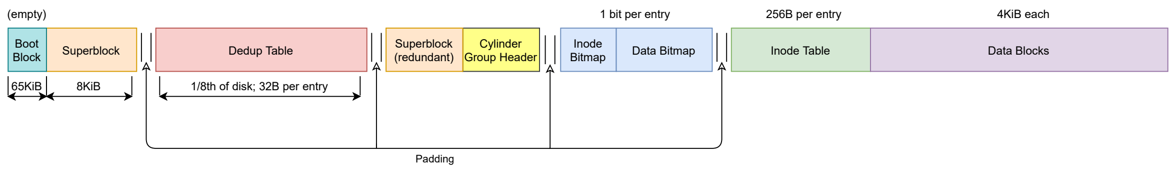

Figure 1: `ddfs` Disk Layout

The on-disk layout of `ddfs` is very similar to FFS/UFS2. There is an empty 65KiB boot block in the beginning of the disk, followed by an 8KiB superblock. We then insert a deduplication table (more detail in the Deduplication Table section). Finally, we have the standard FFS cylinder group header, inode and free block bitmaps, and inode/block data allocation making up the rest of the disk.

For simplicity, we have decided to allocate only one cylinder group, and force the data block size to match the deduplication size of 4KiB. This is to ensure that there is no distinction between fragments and blocks, as otherwise our deduplication strategy becomes much more complex.

2.1.1 Initializing the Filesystem – `newfs-ddfs`

We slightly modified existing FFS `newfs` to lay out the disk space for `ddfs`. The only major change is shifting over the start of the first cylinder group to make room for the deduplication table.

We calculate the block offset and size for the deduplication table and write these to the superblock.

Note that in order to store these two values in the superblock, we use unused fields in the FFS superblock, since we did not want to modify `libufs` (which is used by `newfs`) in order to add new fields to the superblock. Luckily, there are two unused fields right at the start of the superblock with the correct `uint32_t` type. We add `#define`s to alias these names for code clarity.

We also write a custom `FS_DDFS_MAGIC` magic number to the superblock, to avoid someone mounting a `ddfs` partition as a UFS2/FFS partition.

2.2 Deduplication

The basic strategy for deduplication is to keep one and only one block pointer per unique block hash. Entries in a deduplication table containing block hash, block pointer, and ref count are used to track this.

When a file is written, the existing file block is read in, and any modifications from the write are applied to the block. Then the block is then hashed with `sha-1`. If this hash matches an entry in the deduplication table, the block pointer is swapped to the one from the table, the reference count on the table is increased, and the new block that was allocated for this file is released. Otherwise, a new entry is created in the deduplication table.

2.2.1 Deduplication Table

The deduplication table is simply a linear array of "deduplication entries", each containing a key, block pointer, and reference count. While similar to a `kvfs` inode, these are not identical, as our `kvfs` inodes had a

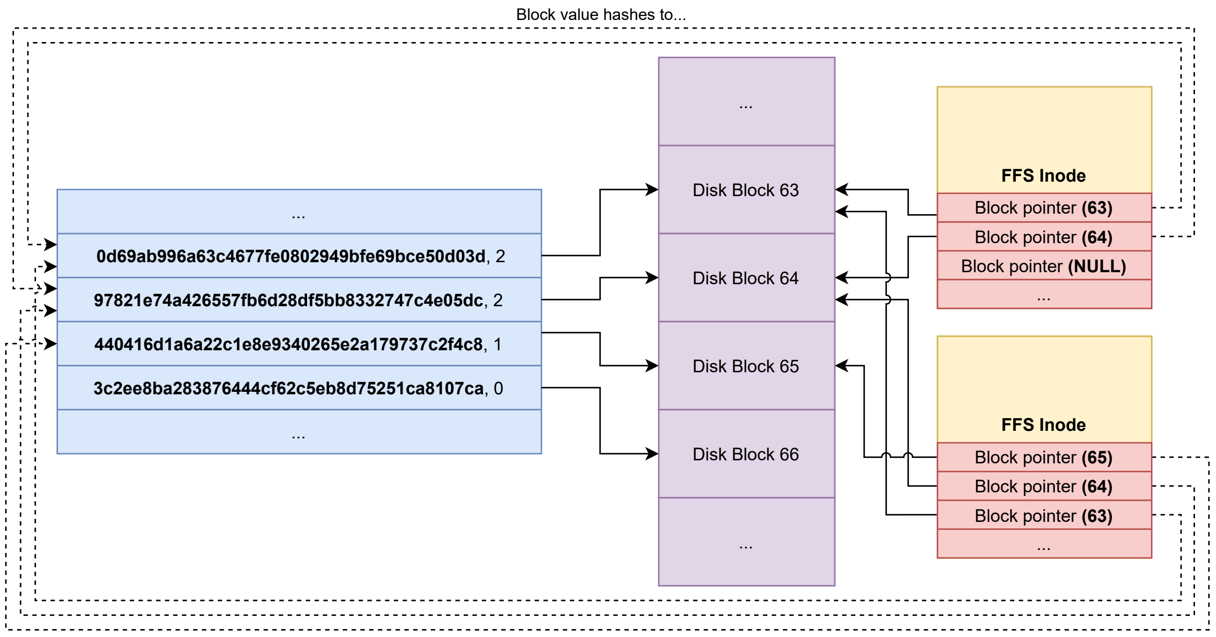

Figure 2: Deduplication Strategy

static 1:1 mapping to a particular block. In contrast, our deduplication table entries are meant to represent any one block, so each contains a 64-bit UFS2 logical block pointer.

It was mentioned earlier that `ddfs` disk layout was changed to always use the same fragment and block size. If this was not done, we would need to include an extra "fragment offset" in each entry, as each block could contain multiple 4KiB fragments which are actually deduplicated separately. By making fragments and blocks the same size, we can always use the inode's block pointer directly when deduplicating.

```
/* DDFS dedup entry flags */
#define DDFS_DEDUP_FREE 0x0001
#define DDFS_DEDUP_ACTIVE 0x0010

struct __attribute__((packed)) ddfs_dedup {
    uint8_t key[20];     /* 160 bit key */
    uint16_t flags;      /* flags. one of FREE | ACTIVE */
    uint16_t ref_count; /* reference count*/
    daddr_t blockptr;    /* block pointer for this key-value pair */
};
```

A table was chosen for simplicity. If given more time, we would have liked to store deduplication entries in a B-Tree (keyed by hash value) or some other type of persistent map. This would have greatly improved lookup efficiency, at the cost of a large amount of complexity.

We reserve 12.5% (1/8) of the disk for these deduplication entries. Performance and usage testing ideally should be used to fine-tune this value, likely to something smaller. This amount was chosen completely arbitrarily, the reasons essentially being "close to 10%, but still a power of 2".

2.2.1.1 Interface

The deduplication table presents the following interface to the filesystem:

```
/*
 * Allocate a free space in the ddtable, or increment an existing key if found.
 * Sets *out_block to the block pointer that was deduplicated for this block.
 * If in_block != out_block, the caller should free the in_block, since it was deduplicated.
 */
int ddtable_alloc(struct ufsmount *mnt, uint8_t key[20], daddr_t in_block, daddr_t *out_block);

/*
 * Decrement a key-value pair in the ddtable.
 * Removes the entry from the table if refcount == 0.
 * Returns the updated refcount of the block, or -1 if not found.
 * If the refcount is 0, the caller is responsible for removing the block.
 */
int ddtable_unref(struct ufsmount *mnt, daddr_t blocknum);
```

This interface is used by the filesystem code in the File Operations section.

2.2.1.2 Lookup

The two external interfaces share an internal helper function `ddtable_lookup()`, that will try to look up an entry in the deduplication table.

In order to reduce duplicate code, this helper function will also return a pointer to a buffer containing the entry that was searched for. The caller can then use this preallocated buffer to read or modify the table. `ddtable_lookup()` will also keep track of the first free space found, if the caller wishes to allocate a new entry.

The lookup algorithm works as follows:

```
for block in dedup_table:
    buf = bread(block)
    for entry in buf:
        if entry is free and free space pointer is NULL:
            free space pointer = idx
        else if entry is free and entry.hash == hash_lookup:
            return idx, buf
    brelse(buf)

# entry was not found
reutrn -1
```

2.3 File Operations

There are two places where we must heavily modify existing FFS file operations: `ffs_write` and `ffs_blkfree`.

- `write` is called whenever a file content is modified.
    - Write is the only entrypoint to modifying a data block.
- `blkfree` is called in many circumstances, but typically whenever file size is changed or when a file is deleted.
    - We modify `blkfree` directly due to the large number of code paths that will deallocate and reallocate blocks. The goal is to change as little code in FFS as possible.

We increment the reference count and swap the block pointers in `write`, and handle decrementing the reference counts in `blkfree`.

2.3.1 Write

We utilize most of the existing `ffs_write` function, including the block allocation strategy in `ffs_balloc`. `write` can always assume that the incoming data blocks are "fresh", since the existing FFS code will re-allocate a block when a file grows or shrinks in size.

Where in FFS the data is simply `uiomove`'d from the user to the data block, we instead hash the block after updating it, and if the hash matches another block, we deduplicate it by swapping the block pointer with the previously allocated block stored in the dedup table with that hash.

If we "deduplicated" the block pointer, we must deallocate the block which was freshly allocated earlier in `write`. This is done in order to minimize the surface area of our changes, as changing the block allocation strategy instead would be much more complex.

The diagram below shows the algorithm used for `write`, including the reallocations done in `ffs_balloc`.

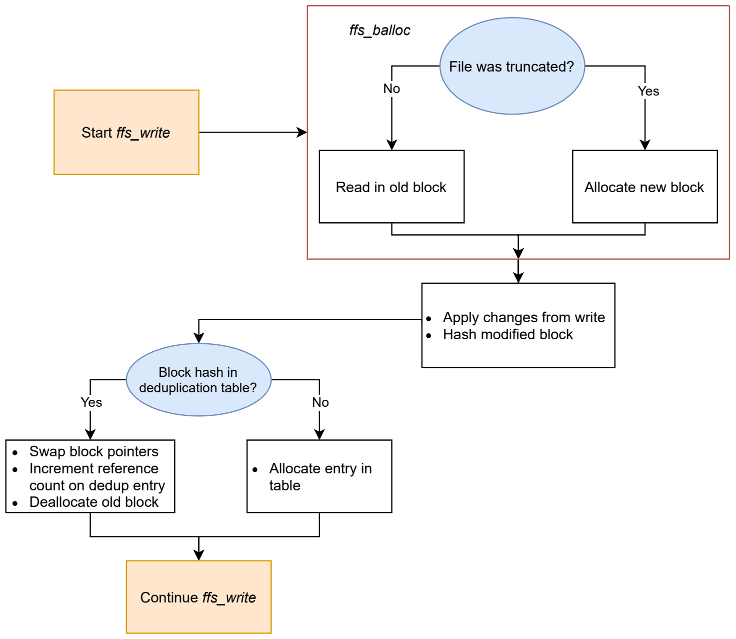

Figure 3: Deduplication Strategy – Write

One point of complexity is that `ffs_write` supports either buffer-mapped or page-mapped allocation for the file blocks. It was not clear how to read in a pagemapped block in order to hash it, whereas hashing the buffer's `b_data` pointer is trivial.

For this reason, `ffs_balloc` was modified to ensure that the backing store is always buffer-mapped.

Finally, the `vop_reallocblks` handler was disabled. `reallocblks` was causing some issues when copying files, as the system will try to grow a fragment up to a full block, which would un-do our deduplication. Because block size == fragment size in `ddfs`, we can just disable `reallocblks` and everything will work as intended.

2.3.2 Blkfree

Modifying `blkfree` is simple. We look up the block being removed in the deduplication table. If the entry is found, we decrement the reference count. If the reference count is 0, we continue with the normal `ffs_blkfree` to properly free the block. We also remove the block if the entry was *not found*, since we do not deduplicate inodes or indirect block pointer blocks or other metadata, so it is important to ensure that these blocks will be freed as expected.

For every block pointer that was to be removed, no matter if we actually freed the blocks or not, we must make sure to overwrite the block pointer in that inode to 0.

2.4 Extra Credit – `statddfs`

We can easily scan through the deduplication table on-disk and calculate how much space was saved by reading the reference counts for each entry. If an entry has a reference count of 3, that means two *additional* files reference this block which would have had their own copy had deduplication not been used. Therefore, 2 blocks are saved from just this entry.

The pseudocode for this program follows.

```
sb = read superblock
start = sb.dedup_table_offset
size = sb.dedup_block_count

blocks_saved = 0
for block in (start to start+size):
    table_entries = read(block)
    for entry in table_entries:
        if entry is not free:
            blocks_saved = entry.refcount - 1

space_saved = blocks_saved * 4KiB
```

3 Modifying FFS

While starting with an existing FFS filesystem and modifying it seemed like a good idea initially, it quickly became clear that there is a lot of complexity in FFS that is unnecessary for our this assignment. This extra complexity made changing the existing code in any major way very difficult. Because it was deemed too difficult to change existing FFS assumptions without severely breaking the many interconnected parts that relied on these assumptions, we decided that it was best to change as little of the existing code as possible in order to get a working deduplicating filesystem.

3.1 Divergence from Stated Goals

Because of this choice, our design diverges from the intended design and stated goals in several notable ways:

- Instead of replacing 64-bit block pointers in a file's inode with 160-bit keys, block pointers from FFS are preserved, and simply swapped out for existing block pointers when doing deduplication.
- Because we retain 15 64-bit block pointers, the maximum file size is instead limited to what it was in Berkeley FFS, which is approximately 1TiB when using 4KiB blocks.
- Although not explicitly stated in the assignment specification, our disk layout is different from intended, reserving 12.5% of the disk for deduplication metadata, with the rest of the disk acting as a normal FFS filesystem.

We feel that our resulting `ddfs` is functionally similar to the design stated in the `ddfs` assignment specification, even though the internal details are different.

3.2 Files and Modifications

The `src/` directory contains the `ddfs` filesystem kernel module. The majority of code in this directory is copied directly from FFS or UFS and slightly modified. The following files were created specifically for `ddfs`:

- `src/ddfs.h`
- `src/ddfs_util.c`

Additionally, the `newfs-ddfs` program located in `tools/` is also copied from `/usr/src/sbin/newfs` and modified.

Any changes made in the copied files are marked with an `/* XXX(ddfs) */` comment. Unless otherwise commented, all other code in these directories is not written by our group, and is unmodified from FFS.

Finally, the `statddfs` extra-credit program in `tools/` is written from scratch, using some existing code from asgn3.

4 Testing

Basic functionality was tested with user-space tools like `cat`, `touch`, `rm`, `mv`, `stat`, and `ls`. In addition, `hexdump` was used extensively to verify the contents of the deduplication table after I/O operations.

Additionally, a suite of functional and integration tests are located in the `tests/` subdirectory. These tests are mostly for basic functionality, and include a number of randomized tests in order to try to break the filesystem.

5 Limitations and Known Issues

- Because our deduplication strategy uses a simple table, all lookups are $O(n)$. We do deduplication lookups on every file I/O operation except `read()`.
  - If given more time, we would have liked to implement caching for deduplication table lookups.
- We do not currently implement deduplication on indirect block pointers, and do not support them.
  - Unfortunately our `test_max` would always fail due to some weirdness with indirect block pointers. This test is currently disabled.
  - Because we utilize block allocation and read/write and all other code from FFS, files with indirect blocks *should* still work. However, large files using indirect blocks have not been tested and debugged due to time constraints, so we just say we don't support them.
- Occasionally when unloading or reloading the module, `dmesg` will get filled with warnings about re-using sysctl leafs. These sysctls are unmodified from FFS code, and were not removed due to time constraints.
- Would like to clean up and remove unused code related to UFS1, soft updates, etc.
- The deduplication table is always synchronously written to disk with `bwrite()`.

References

As stated in the Modifying FFS section, much of the code was taken directly from the FreeBSD source tree. This includes:

- `/usr/src/sys/ufs/ffs/`
- `/usr/src/sys/ufs/ufs/`
- `/usr/src/sbin/newfs/`
- `/usr/src/sys/modules/ufs/Makefile`

The only external resource used was some conceptual discussions between our group and Matthew Boisvert's group. Aidan and Matthew discussed high-level concepts for the assignment, and both groups ended up using a similar design.

The filesystem layout diagram, deduplication diagram, and write flow diagram were created using https://diagrams.net.