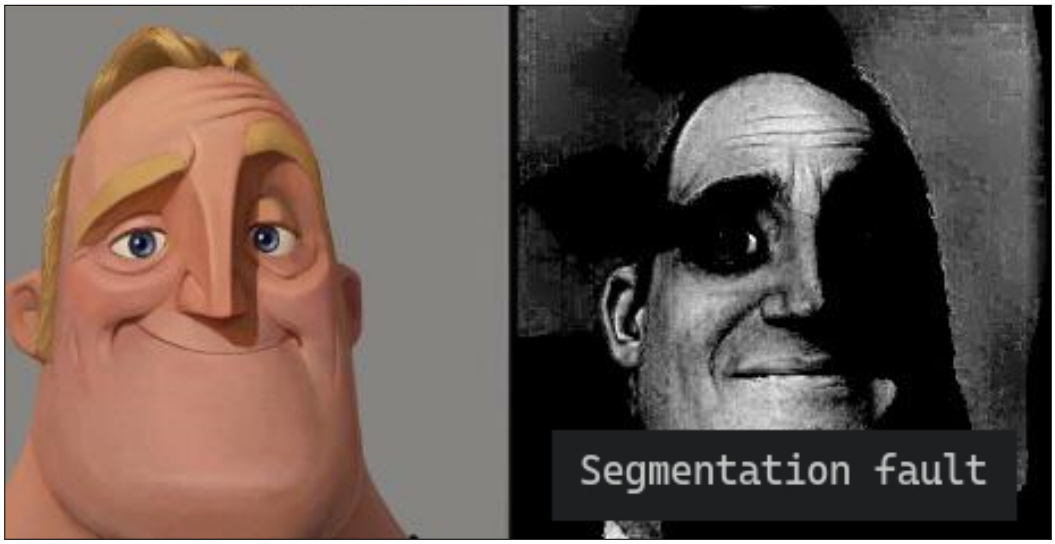
DM510 Operating systems
Assignment 3
File system



Frederik List
frlis21@student.sdu.dk

August 23, 2024

Introduction

When booted, one of the first things operating systems do is mount a block device, which probably has a file system on it. One could hack the kernel to support a new file system, but that's hard. FUSE lets us create a file system in userspace. In this report, I outline how I created the terrible file system (abbreviated TFS) with FUSE.

Design

TFS is very barebones. It can read and write (large) files and directories... and that's it. TFS is neither journalling nor log structured. The focus was on making TFS simple, which one might say TFS fails at, for reasons you will discover.

Layout

The layout of a TFS partition is as ordered:

- TFS Superblock, containing
  - the number of total blocks and nodes.
  - pointers to the heads of the free node and block linked lists.
- Nodes, where each node contains either
  - node metadata, such as node name, blocks, access time, etc.
  - a pointer to the next free node in the free node linked list.
- Blocks, where each block contains either
  - file data or directory entries.
  - indirect block pointers.
  - a single pointer to the next block in the free block linked list.

TFS allocates space for nodes near the beginning of the partition at format time.

Allocation scheme

TFS uses a combined index allocation scheme (exactly the one shown in figure 14.8 of the course textbook). Early in development, TFS used a file allocation table (FAT) for its simplicity, but the author suffers from occasional slight mental degradation and premature optimization syndrome, so he chose to switch to the more complicated and slightly faster indexed allocation scheme. (At least now he can open pictures of his dog a few milliseconds faster on a TFS mount.)

Random access is much faster for the indexed allocation scheme than for a FAT. This is because instead of having to go through the entire file to find a specific block, a formula can be used to calculate the block indices from the file offset.

If the required data cannot be found in the direct blocks, then for a file block offset x (starting at 0 regardless of the number of direct blocks), the indirect block level can be calculated with

$$d = \left\lfloor \log_b \left(x + 1 - \left\lfloor \frac{x+1}{b} \right\rfloor \right) \right\rfloor$$

where b = the number of block pointers that can fit in a block ($\frac{\text{block size}}{\text{pointer size}}$). Block level indices can then be calculated with

$$i_k = \left\lfloor \frac{x - b\frac{b^d - 1}{b-1}}{b^k} \right\rfloor \mod b$$

where k = index level.

If the block size (and block pointer size, and hence max block pointers) is a power of 2, then bit twiddling can be used for fast logarithm and exponent calculation.

Free nodes and blocks

Free nodes contain a single pointer to the next free node, thus forming a linked list of free nodes.

Similarly, free blocks contain a single pointer to the next free block.

Implementation

TFS's code is somewhat littered with comments explaining most of the esoteric bits, this section serves to supplement those comments.

Backing store

TFS's backing store is in the form of a memory-mapped binary file. A TFS image may be created with the provided `mktfs` utility, much like Linux's `mke2fs(8)`. As with `mke2fs`, a file needs to be allocated first, probably with the `fallocate(1)` utility. In the accompanying video a 16 MiB file is allocated with `fallocate -l16M img.tfs`, after which `mktfs img.tfs` is invoked.

Instead of reading and writing to the file with, for example, `pread(2)`, TFS memory maps the entire backing store. The author feels some might not think this to be in the spirit of the project, but argues that without memory mapping, the code would just be polluted with some extra `malloc`s and `pread`s. Memory mapping allows everyone to enjoy the internals of TFS without more cruft than is required!

Looking up entries

Entry lookup is implemented with the standard C library hash table (see `hsearch(3)`). Keys are paths and values are nodes. This makes entry lookup very fast, but the memory usage may be a concern for large file systems, as the hash table size is simply initialized to the number of nodes in the file system.

Reading and writing blocks

An iterator-like structure named the "block cursor" allows for random-access seeking and sequential iteration.

Reading and writing are much the same. A block cursor is simply set to the position to start from and iterated until no more bytes can or should be written or read.

Allocating and freeing blocks and nodes

Nodes are allocated and freed simply by popping from or pushing to the free node linked list.

Blocks are a little more tricky. This is because in addition to blocks storing actual data, index blocks also need to be allocated for sufficiently large files. The cursor iterator

presented earlier is able to take a callback "hook" which will be called for each block it iterates through, including index blocks which are transparent to iterator consumers.

Allocation can be easily implemented using this callback. A cursor is set to the end block of a file and iterated through with a function that pops free blocks from the free block linked list.

Freeing blocks is not so easy. Ideally, a cursor would be set to the end block of a file and iterated through *backwards* while freeing blocks along the way. Instead, a hack was used that makes many assumptions about how the file system works: a cursor is set to the last required block of a file and iterated, adding any blocks along the way to a free block buffer. After an iteration is done, any blocks in the free block buffer are freed and the free block buffer is cleared. This is not so good, as free block and index block pointers are written to the same locations, and as the ancient proverb goes, "don't shit where you eat".

Crash recovery

Crash recovery in TFS is nonexistent; when the kernel decides to page out to the backing store is indeterminate. Disregarding memory mapping, it could be implemented with journalling: A portion of the TFS partition would be alloted to journalling, where dirty blocks (and their checksums) would periodically be sequentially written. After a journal is written, data can be moved to where it is supposed to go with slower random access without fear of losing data.

In case of a crash, the journal can be read and the rearranging of data can be resumed. Checksums are used to ensure only data that was successfully written to the journal are written to the disk.

Journalling can handle any number of crashes while recovering from a crash; the only data needed to recover is written to the nonvolatile disk. Of course, some small amount of data loss is always possible when a crash occurs while writing dirty blocks, the only remedy for which is to write often.

Tests

Mosts of the tests performed in the video are just the author trying to abuse TFS as hard as he can.

Figure 1: Tests with video timestamps

Test	**Time**
Formatting	00:00
File creation and deletion	00:10
Directory creation and deletion	00:24
Access and modification time using `touch` and `cat`	00:47
Large files using `seq`, `head`, and `tail`	01:00
Directories with many entries	01:12
good boi	01:29
File persistence	01:39

In more rigorous testing under development, an ad-hoc analysis concluded that TFS does not produce dead blocks or leak memory.

TFS breaks down a little bit when there is not much space left. Not all cases concerning lack of storage are well tested.

Conclusion

Though the author would not trust it for anything remotely important, it works. TFS is slow and probably riddled with nasty bugs, but it gets the job done.

Appendix A

Source code

Listing A.1: `tfs.h`

```
#ifndef TFS_H
#define TFS_H

#include <stdlib.h>
#include <sys/mman.h>

typedef off_t blkoff_t;
typedef off_t nodoff_t;

// Block size must be a power of 2 for bit twiddlings
#define BLOCK_SIZE 4096
#define BLOCKS_PER_NODE 4
#define DIRECT_BLOCKS 12
#define ILEVELS 3
#define NAME_LIMIT 64
#define BLOCK_MAX_CHILDREN (BLOCK_SIZE / sizeof(nodoff_t))
#define BLOCK_MAX_POINTERS (BLOCK_SIZE / sizeof(blkoff_t))
#define END_BLOCKS -1
#define END_NODES -1

#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

// Absolute node size
#define NODE_SIZE(node) ((node)->mode & S_IFDIR ? (node)->nlink * sizeof(nodoff_t) : (node)->size)
// Number required blocks for data (not including indirect pointer blocks)
#define NODE_NRBLOCKS(node) ((BLOCK_SIZE + NODE_SIZE(node) - 1) / BLOCK_SIZE)

/**
 * TFS superblock:
 * Metadata needed to calculate everything.
 */
struct tfs_header {
        blkoff_t nblocks, free_block_head;
        nodoff_t nnodes, free_node_head;
};

/**
 * The TFS node, as it is represented in the image file.
 */
struct tfs_node {
        union {
                struct {
                        mode_t mode;
                        char name[NAME_LIMIT];
                        // Direct blocks
                        blkoff_t blocks[DIRECT_BLOCKS];
                        // Indirect blocks
```

```
                        blkoff_t iblocks[ILEVELS];
                        // Number of allocated blocks
                        fsblkcnt_t nblocks;
                        // Number of links of directory, file size otherwise
                        union {
                                off_t size;
                                nlink_t nlink;
                        };
                        struct timespec atim, mtim;
                };
                // Used for free node linked list.
                nodoff_t next;
        };
};

/**
 * Collection of pointers to useful places and other info nice to have.
 */
struct tfs_info {
        blkoff_t nblocks;
        nodoff_t nnodes;
        blkoff_t *free_block_head;
        nodoff_t *free_node_head;
        struct tfs_node *nodes;
        char (*data)[BLOCK_SIZE];
        /* no touchy */
        void *base;
        off_t filesize;
};

/**
 * Open a file as a TFS image.
 */
int tfs_open(const char *filename);

/**
 * Open and initialize a TFS image.
 */
int tfs_load(const char *filename);

/**
 * Format a TFS image.
 */
void tfs_format();

/**
 * Calculate pointers and other useful things.
 */
void tfs_init();

/**
 * Write back any "queued" changes.
 */
int tfs_destroy();

/**
 * (De)allocate necessary blocks for a node.
 *
 * Must be called after changing the size or nlink of a node.
 */
int tfs_node_trim(struct tfs_node *node);

/**
 * Get a node from the hash table given the path.
 */
struct tfs_node *get_node(const char *path);

/**
```

```
 * Get the directory of a node from the hash table given the path.
 */
struct tfs_node *get_directory(const char *path);

/**
 * Collect children of a node into one contiguous array.
 *
 * The array must be freed when you are done with it.
 */
struct tfs_node **tfs_node_children(struct tfs_node *node);

/**
 * Read node data.
 */
int tfs_node_read(struct tfs_node *node, char *buf, size_t size, off_t offset);

/**
 * Write node data.
 */
int tfs_node_write(struct tfs_node *node, const char *buf, size_t size, off_t offset);

/**
 * Add a node.
 */
int tfs_add_node(const char *path, mode_t mode);

/**
 * Remove a node.
 */
int tfs_remove_node(const char *path);

#endif // TFS_H
```

Listing A.2: `fuse_tfs.c`

```
#define FUSE_USE_VERSION 29

#include <assert.h>
#include <errno.h>
#include <fuse.h>
#include <libgen.h>
#include <math.h>
#include <search.h> // hsearch
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "tfs.h"

static int fuse_tfs_getattr(const char *path, struct stat *stbuf) {
        fprintf(stderr, "getattr %s\n", path);
        memset(stbuf, 0, sizeof(struct stat));

        struct tfs_node *node = get_node(path);
        if (!node)
                return -ENOENT;

        stbuf->st_mode = node->mode;
        stbuf->st_nlink = (node->mode & S_IFDIR) ? node->nlink + 1 : 1;
        stbuf->st_size = NODE_SIZE(node);
        stbuf->st_atim = node->atim;
        stbuf->st_mtim = node->mtim;

        return 0;
}

```

```c
static int fuse_tfs_mknod(const char *path, mode_t mode, dev_t rdev) {
        fprintf(stderr, "mknod %s\n", path);
        return tfs_add_node(path, mode);
}

static int fuse_tfs_mkdir(const char *path, mode_t mode) {
        fprintf(stderr, "mkdir %s\n", path);
        // Bitwise OR with S_IFDIR because the documentation says to.
        return tfs_add_node(path, mode | S_IFDIR);
}

static int fuse_tfs_unlink(const char *path) {
        fprintf(stderr, "unlink %s\n", path);
        struct tfs_node *node = get_node(path);
        if (!node)
                return -ENOENT;
        if (node->mode & S_IFDIR)
                return -EISDIR;

        return tfs_remove_node(path);
}

static int fuse_tfs_rmdir(const char *path) {
        fprintf(stderr, "rmdir %s\n", path);
        struct tfs_node *node = get_node(path);
        if (!node)
                return -ENOENT;
        if (!(node->mode & S_IFDIR))
                return -ENOTDIR;
        if (node->size > 0)
                return -ENOTEMPTY;

        return tfs_remove_node(path);
}

static int fuse_tfs_truncate(const char *path, off_t size) {
        fprintf(stderr, "truncate %s\n", path);
        struct tfs_node *node = get_node(path);
        if (!node)
                return -ENOENT;
        if (node->mode & S_IFDIR)
                return -EISDIR;

        node->size = size;

        return tfs_node_trim(node);
}

static int fuse_tfs_open(const char *path, struct fuse_file_info *fi) {
        fprintf(stderr, "open %s\n", path);
        if (!get_node(path))
                return -ENOENT;

        return 0;
}

static int fuse_tfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi) {
        fprintf(stderr, "read %s\n", path);
        struct tfs_node *node = get_node(path);
        if (!node)
                return -ENOENT;
        if (node->mode & S_IFDIR)
                return -EISDIR;

        return tfs_node_read(node, buf, size, offset);
}

static int fuse_tfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi) {
```

```
101         fprintf(stderr, "write %s\n", path);
102         struct tfs_node *node = get_node(path);
103         if (!node)
104                 return -ENOENT;
105         if (node->mode & S_IFDIR)
106                 return -EISDIR;
107
108         return tfs_node_write(node, buf, size, offset);
109 }
110
111 static int fuse_tfs_release(const char *path, struct fuse_file_info *fi) {
112         fprintf(stderr, "release %s\n", path);
113         return 0; // no-op
114 }
115
116 static int fuse_tfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
117                             struct fuse_file_info *fi) {
118         fprintf(stderr, "readdir %s\n", path);
119         struct tfs_node *node = get_node(path);
120         if (!node)
121                 return -ENOENT;
122         if (node->mode & S_IFREG)
123                 return -ENOTDIR;
124
125         filler(buf, ".", NULL, 0);
126         filler(buf, "..", NULL, 0);
127
128         struct tfs_node **children = tfs_node_children(node);
129
130         for (int i = 0; i < node->nlink; i++) {
131                 struct tfs_node *child = children[i];
132
133                 filler(buf, child->name, NULL, 0);
134         }
135
136         free(children);
137
138         return 0;
139 }
140
141 static void fuse_tfs_destroy(void *data) {
142         tfs_destroy();
143         hdestroy();
144 }
145
146 static int fuse_tfs_utimens(const char *path, const struct timespec tv[2]) {
147         fprintf(stderr, "utimens %s\n", path);
148         struct tfs_node *node = get_node(path);
149         if (!node)
150                 return -ENOENT;
151
152         node->atim = tv[0];
153         node->mtim = tv[1];
154
155         return 0;
156 }
157
158 struct tfs_config {
159         char *tfs_file_path;
160 };
161
162 enum {
163         KEY_HELP,
164 };
165
166 // We intercept the help flag.
167 static struct fuse_opt tfs_opts[] = {FUSE_OPT_KEY("-h", KEY_HELP), FUSE_OPT_KEY("--help", KEY_HELP), FUSE_OPT_END};
168
```

```c
static int tfs_opt_proc(void *data, const char *arg, int key, struct fuse_args *outargs) {
        struct tfs_config *config = data;

        switch (key) {
        case KEY_HELP:
                fprintf(stderr,
                        "usage: %s file mountpoint [fuse options]\n"
                        "\n"
                        "`file` must exist and must be initialized with `mktfs`."
                        "\n"
                        "See fuse(8) for more options.\n",
                        outargs->argv[0]);
                return -1;
        case FUSE_OPT_KEY_NONOPT:
                if (config->tfs_file_path)
                        return 1;

                config->tfs_file_path = strdup(arg);
                return 0;
        }

        return 1;
}

static struct fuse_operations fuse_tfs_oper = {.getattr = fuse_tfs_getattr,
                                               .mknod = fuse_tfs_mknod,
                                               .mkdir = fuse_tfs_mkdir,
                                               .unlink = fuse_tfs_unlink,
                                               .rmdir = fuse_tfs_rmdir,
                                               .truncate = fuse_tfs_truncate,
                                               .open = fuse_tfs_open,
                                               .read = fuse_tfs_read,
                                               .write = fuse_tfs_write,
                                               .release = fuse_tfs_release,
                                               .readdir = fuse_tfs_readdir,
                                               .destroy = fuse_tfs_destroy,
                                               .utimens = fuse_tfs_utimens};

int main(int argc, char *argv[]) {
        struct fuse_args args = FUSE_ARGS_INIT(argc, argv);
        struct tfs_config config = {.tfs_file_path = NULL};

        if (fuse_opt_parse(&args, &config, tfs_opts, tfs_opt_proc) == -1)
                return 1;

        if (!config.tfs_file_path) {
                fprintf(stderr, "tfs: missing file to mount\n");
                return 1;
        }

        int ret = tfs_load(config.tfs_file_path);
        if (ret)
                return ret;

        return fuse_main(args.argc, args.argv, &fuse_tfs_oper, NULL);
}
```

Listing A.3: `tfs.c`

```c
#include "tfs.h"
#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <math.h>
#include <search.h>
#include <stdio.h>
```

```c
#include <string.h>
#include <time.h>
#include <unistd.h>

static struct tfs_info tfs_info;

// Node number from pointer.
#define NODENO(node) ((node)-tfs_info.nodes)
// Block number from pointer.
#define BLOCKNO(block) ((block)-tfs_info.data)
// Cast block data to an array of node offsets.
#define BLOCK_NODES(block) ((nodoff_t *)(tfs_info.data[block]))
// Cast block data to an array of block offsets.
#define BLOCK_POINTERS(block) ((blkoff_t *)(tfs_info.data[block]))
// Cast block data to the next member in the free block linked list.
#define NEXT_FREE_BLOCK(block) BLOCK_POINTERS(block)[0]

/**
 * Iterating through indirect levels is painful,
 * so the process is abstracted away with the help of this iterator-like thingy.
 */
struct block_cursor {
        struct tfs_node *node;
        blkoff_t i;
        int level;
        blkoff_t pos[ILEVELS];
        blkoff_t block[ILEVELS];
};

// Get what block an iterator is currently on.
#define CURRENT_BLOCK(cursor)                                                                                   \
        ((cursor)->i < DIRECT_BLOCKS ? (cursor)->node->blocks[(cursor)->i]
                                     : BLOCK_POINTERS((cursor)->block[(cursor)->level])[(cursor)->pos[(cursor)->level]])

// Iterator definition helper
#define DEFINE_BLOCK_CURSOR(var, nodeptr)                                                                       \
        struct block_cursor var = {
            .node = nodeptr,
            .level = -1,
        };

// Most significant bit
static int msb(unsigned int n) {
        // I'm pretty sure gcc -O>1 compiles this down to a single BSR instruction.
        // Godbolt says it does, in which case this happens to be fast enough.
        // In any case, this is faster than log2().
        unsigned r = 0;
        while (n >>= 1)
                r++;

        return r;
}

// msb(x) == floor(log2(x))
#define MAX_POINTERS_NBITS msb(BLOCK_MAX_POINTERS)
#define BLOCK_SIZE_NBITS msb(BLOCK_SIZE)
// floor-log base MAX_POINTERS
#define FLOG(x) (msb(x) / MAX_POINTERS_NBITS)
// This bit-twiddling is why BLOCK_SIZE must be a power of 2.
#define MAX_POINTERS_POW(e) (e == 0 ? 1 : BLOCK_MAX_POINTERS << (MAX_POINTERS_NBITS * (e - 1)))

/**
 * Set the position of a cursor for random access.
 */
static blkoff_t block_seek(struct block_cursor *cursor, blkoff_t pos) {
        blkoff_t nblocks = cursor->node->nblocks;
        cursor->i = pos;
        cursor->level = -1;
```

```
77
78          if (pos > nblocks)
79                  return -1;
80          if (pos < DIRECT_BLOCKS)
81                  return cursor->node->blocks[pos];
82
83          // Start from 0
84          pos -= DIRECT_BLOCKS;
85          cursor->level = FLOG(pos + 1 - (pos + 1) / BLOCK_MAX_POINTERS);
86          blkoff_t accum = BLOCK_MAX_POINTERS * (MAX_POINTERS_POW(cursor->level) - 1) / (BLOCK_MAX_POINTERS - 1);
87          blkoff_t offset = pos - accum;
88
89          cursor->block[0] = cursor->node->iblocks[cursor->level];
90
91          for (int i = 0; i < cursor->level; i++) {
92                  blkcnt_t N = MAX_POINTERS_POW(cursor->level - i);
93                  cursor->pos[i] = offset / N;
94                  offset %= N;
95                  cursor->block[i + 1] = BLOCK_POINTERS((cursor)->block[i])[(cursor)->pos[i]];
96          }
97
98          cursor->pos[cursor->level] = offset;
99
100         for (int i = cursor->level + 1; i < ILEVELS; i++)
101                 cursor->pos[i] = 0;
102
103         return CURRENT_BLOCK(cursor);
104 }
105
106 /**
107  * Iterate a cursor through a callback.
108  *
109  * Useful for inspecting all blocks we iterate through, including between levels.
110  */
111 static blkoff_t iter_through(struct block_cursor *cursor,
112                              blkoff_t (*callback)(struct block_cursor *cursor, int level)) {
113         if (++cursor->i < DIRECT_BLOCKS)
114                 return callback(cursor, -1);
115
116         int level = cursor->level;
117         while (level >= 0 && ++cursor->pos[level] >= BLOCK_MAX_POINTERS)
118                 cursor->pos[level--] = 0;
119
120         if (level == -1)
121                 // Done with this level, go to next level.
122                 cursor->level += 1;
123
124         for (; level < cursor->level; level++)
125                 cursor->block[level + 1] = callback(cursor, level);
126
127         return callback(cursor, cursor->level);
128 }
129
130 /**
131  * Callback for simply getting the block a cursor is sitting on.
132  */
133 static blkoff_t _next_block_callback(struct block_cursor *cursor, int level) {
134         if (cursor->i >= cursor->node->nblocks)
135                 return END_BLOCKS;
136         if (cursor->i < DIRECT_BLOCKS)
137                 return cursor->node->blocks[cursor->i];
138         if (level == -1)
139                 return cursor->node->iblocks[cursor->level];
140
141         return BLOCK_POINTERS((cursor)->block[level])[(cursor)->pos[level]];
142 }
143
144 /**
```

```
145  * Sequential access.
146  */
147 #define next_block(cursor) iter_through(cursor, _next_block_callback)
148
149 int tfs_open(const char *filename) {
150         int fd = open(filename, O_RDWR);
151         if (fd == -1)
152                 return -errno;
153
154         // Memory map the file.
155         // This will work for most x86-64 machines, but I'm not so sure about much else...
156         tfs_info.filesize = lseek(fd, 0, SEEK_END);
157         tfs_info.base = mmap(NULL, tfs_info.filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
158         close(fd);
159
160         if (tfs_info.base == MAP_FAILED)
161                 return -errno;
162
163         return 0;
164 }
165
166 void tfs_format() {
167         struct tfs_header *header = tfs_info.base;
168         // Allocate the FAT (implicitly), blocks, and nodes.
169         header->nblocks = tfs_info.filesize / (BLOCK_SIZE + (sizeof(struct tfs_node) / BLOCKS_PER_NODE));
170         header->nnodes = header->nblocks / BLOCKS_PER_NODE;
171         // Root takes 1 node.
172         header->free_node_head = 1;
173         header->free_block_head = 0;
174
175         // Now (re)calculate pointers to FAT n' stuff.
176         tfs_init();
177
178         // Initialize root node:
179         struct tfs_node *root = &tfs_info.nodes[0];
180         root->mode = S_IFDIR | 644;
181         root->name[0] = '\0'; // Root has no name.
182         root->nblocks = 0;
183         root->nlink = 0;
184         clock_gettime(CLOCK_REALTIME, &root->atim);
185         root->mtim = root->atim;
186
187         // Initialize free blocks:
188         for (int i = *tfs_info.free_block_head; i < tfs_info.nblocks - 1; i++)
189                 NEXT_FREE_BLOCK(i) = i + 1;
190         NEXT_FREE_BLOCK(tfs_info.nblocks - 1) = END_BLOCKS;
191
192         // Initialize free nodes:
193         for (int i = *tfs_info.free_node_head; i < tfs_info.nnodes - 1; i++)
194                 tfs_info.nodes[i].next = i + 1;
195         tfs_info.nodes[tfs_info.nnodes - 1].next = END_NODES;
196 }
197
198 /**
199  * Walk the entire filesystem, adding all nodes to the hash table.
200  */
201 static void init_htable(const char *path, struct tfs_node *node) {
202         ENTRY entry;
203
204         if (path) {
205                 entry.key = malloc(strlen(path) + 1 + strlen(node->name) + 1);
206                 sprintf(entry.key, "%s/%s", path, node->name);
207         } else {
208                 // Special case for root
209                 entry.key = strdup("/");
210         }
211
212         entry.data = node;
```

```
213         hsearch(entry, ENTER);
214
215         fprintf(stderr, "found %s\n", entry.key);
216
217         if (!(node->mode & S_IFDIR))
218                 return;
219
220         // Recurse through directory:
221         struct tfs_node **children = tfs_node_children(node);
222
223         for (int i = 0; i < node->nlink; i++)
224                 init_htable(path ? entry.key : "", children[i]);
225
226         free(children);
227 }
228
229 void tfs_init() {
230         struct tfs_header *header = tfs_info.base;
231
232         tfs_info.nblocks = header->nblocks;
233         tfs_info.nnodes = header->nnodes;
234         tfs_info.free_block_head = &header->free_block_head;
235         tfs_info.free_node_head = &header->free_node_head;
236         tfs_info.nodes = tfs_info.base + sizeof(struct tfs_header);
237         tfs_info.data = (void *)tfs_info.nodes + sizeof(struct tfs_node) * tfs_info.nnodes;
238
239         fprintf(stderr, "nblocks: %ld\n", tfs_info.nblocks);
240         fprintf(stderr, "nnodes: %ld\n", tfs_info.nnodes);
241         fprintf(stderr, "free_node_head: %ld\n", *tfs_info.free_node_head);
242         fprintf(stderr, "free_block_head: %ld\n", *tfs_info.free_block_head);
243 }
244
245 int tfs_load(const char *filename) {
246         int ret = tfs_open(filename);
247         if (ret)
248                 return ret;
249
250         tfs_init();
251
252         hcreate(tfs_info.nnodes); // Initialize hash table, see hsearch(3)
253         init_htable(NULL, &tfs_info.nodes[0]);
254
255         return ret;
256 }
257
258 struct tfs_node *get_node(const char *path) {
259         ENTRY entry = {
260             .key = strdup(path),
261         };
262         ENTRY *result = hsearch(entry, FIND);
263         free(entry.key);
264
265         if (!result)
266                 return NULL;
267
268         return result->data;
269 }
270
271 struct tfs_node *get_directory(const char *path) {
272         char *pathc = strdup(path);
273         char *parent_path = dirname(pathc);
274         struct tfs_node *parent_node = get_node(parent_path);
275         free(pathc);
276         return parent_node;
277 }
278
279 /**
280  * Set or update a node give a path.
```

```
 *
 * Especially useful for unsetting a node by passing NULL.
 */
static void set_node(const char *path, struct tfs_node *node) {
        ENTRY entry = {
            .key = strdup(path),
            .data = node,
        };
        ENTRY *result = hsearch(entry, FIND);

        if (!result) {
                hsearch(entry, ENTER);
                return;
        }

        result->data = node;
        free(entry.key);
}

/**
 * Iterator callback for freeing blocks we iterate through.
 *
 * WARNING extreme hacks, detailed in report.
 */
static blkoff_t free_block_buffer[ILEVELS + 1];
static blkoff_t _free_callback(struct block_cursor *cursor, int level) {
        return free_block_buffer[level + 1] = _next_block_callback(cursor, level);
}

/**
 * Iterator callback for allocating and setting blocks.
 *
 * Also kind of hacky...
 */
static blkoff_t _alloc_callback(struct block_cursor *cursor, int level) {
        blkoff_t block = *tfs_info.free_block_head;

        if (block == END_BLOCKS)
                return -1;

        *tfs_info.free_block_head = NEXT_FREE_BLOCK(block);

        if (cursor->i < DIRECT_BLOCKS)
                return cursor->node->blocks[cursor->i] = block;
        if (level == -1)
                return cursor->node->iblocks[cursor->level] = block;

        return BLOCK_POINTERS((cursor)->block[level])[(cursor)->pos[level]] = block;
}

int tfs_node_trim(struct tfs_node *node) {
        blkoff_t nrblocks = NODE_NRBLOCKS(node);
        blkoff_t dblocks = nrblocks - node->nblocks;

        DEFINE_BLOCK_CURSOR(cursor, node);

        if (dblocks < 0) {
                block_seek(&cursor, nrblocks - 1);
                while (dblocks && iter_through(&cursor, _free_callback) != END_BLOCKS) {
                        dblocks += 1;

                        // Clear out free block buffer:
                        for (int i = 0; i <= cursor.level + 1; i++) {
                                if (free_block_buffer[i] < 0)
                                        continue;
                                NEXT_FREE_BLOCK(free_block_buffer[i]) = *tfs_info.free_block_head;
                                *tfs_info.free_block_head = free_block_buffer[i];
                                free_block_buffer[i] = -1;
```

```
349                         }
350                 }
351                 node->nblocks = nrblocks;
352         } else {
353                 block_seek(&cursor, node->nblocks - 1);
354                 while (dblocks && iter_through(&cursor, _alloc_callback) != END_BLOCKS)
355                         dblocks -= 1;
356                 node->nblocks = nrblocks - dblocks;
357         }
358
359         // In case we couldn't allocate enough blocks, set sizes correctly.
360         if (node->mode & S_IFDIR)
361                 node->nlink = MIN(node->nlink, node->nblocks * BLOCK_MAX_CHILDREN);
362         else
363                 node->size = MIN(node->size, node->nblocks * BLOCK_SIZE);
364
365         return dblocks > 0 ? -ENOSPC : 0;
366 }
367
368 int tfs_node_read(struct tfs_node *node, char *buf, size_t size, off_t offset) {
369         DEFINE_BLOCK_CURSOR(cursor, node);
370         size_t chunk, to_read = size;
371         blkoff_t block = block_seek(&cursor, offset / BLOCK_SIZE);
372
373         while (offset < NODE_SIZE(node) && (chunk = MIN(to_read, BLOCK_SIZE - (offset % BLOCK_SIZE)))) {
374                 memcpy(buf, &tfs_info.data[block][offset % BLOCK_SIZE], MIN(chunk, NODE_SIZE(node) - offset));
375                 block = next_block(&cursor);
376                 to_read -= chunk;
377                 offset += chunk;
378                 buf += chunk;
379         }
380
381         clock_gettime(CLOCK_REALTIME, &node->atim);
382
383         return size - to_read;
384 }
385
386 int tfs_node_write(struct tfs_node *node, const char *buf, size_t size, off_t offset) {
387         node->size = MAX(node->size, offset + size);
388         int ret = tfs_node_trim(node);
389
390         DEFINE_BLOCK_CURSOR(cursor, node);
391         size_t chunk, to_write = size;
392         blkoff_t block = block_seek(&cursor, offset / BLOCK_SIZE);
393
394         while (offset < NODE_SIZE(node) && (chunk = MIN(to_write, BLOCK_SIZE - (offset % BLOCK_SIZE)))) {
395                 memcpy(&tfs_info.data[block][offset % BLOCK_SIZE], buf, chunk);
396                 block = next_block(&cursor);
397                 to_write -= chunk;
398                 offset += chunk;
399                 buf += chunk;
400         }
401
402         clock_gettime(CLOCK_REALTIME, &node->mtim);
403
404         return ret < 0 ? ret : size - to_write;
405 }
406
407 struct tfs_node **tfs_node_children(struct tfs_node *node) {
408         nodoff_t *children = malloc(NODE_SIZE(node));
409         if (!children)
410                 return NULL;
411
412         tfs_node_read(node, (void *)children, NODE_SIZE(node), 0);
413
414         struct tfs_node **children_nodes = malloc(node->nlink * sizeof(struct tfs_node *));
415         if (!children_nodes)
416                 return NULL;
```

```
417
418         for (int i = 0; i < node->nlink; i++)
419                 children_nodes[i] = &tfs_info.nodes[children[i]];
420
421         free(children);
422
423         return children_nodes;
424 }
425
426 int tfs_add_node(const char *path, mode_t mode) {
427         if (get_node(path))
428                 return -EEXIST;
429         if (*tfs_info.free_node_head == END_NODES)
430                 return -ENOSPC;
431
432         char *basename = strrchr(path, '/') + 1;
433         if (strlen(basename) + 1 > NAME_LIMIT)
434                 return -ENAMETOOLONG;
435
436         // Allocate node.
437         nodoff_t nodei = *tfs_info.free_node_head;
438         struct tfs_node *node = &tfs_info.nodes[nodei];
439         *tfs_info.free_node_head = node->next;
440         fprintf(stderr, "\tAllocated node %ld...\n", nodei);
441
442         // Initialize node.
443         strcpy(node->name, basename);
444         node->mode = mode;
445         if (node->mode & S_IFDIR)
446                 node->nlink = 0;
447         else
448                 node->size = 0;
449         node->nblocks = 0;
450         clock_gettime(CLOCK_REALTIME, &node->atim);
451         node->mtim = node->atim;
452
453         // Add child to parent.
454         struct tfs_node *parent_node = get_directory(path);
455         parent_node->nlink += 1;
456         tfs_node_trim(parent_node);
457         DEFINE_BLOCK_CURSOR(cursor, parent_node);
458         BLOCK_NODES(block_seek(&cursor, parent_node->nblocks - 1))
459         [(parent_node->nlink - 1) % BLOCK_MAX_CHILDREN] = nodei;
460
461         clock_gettime(CLOCK_REALTIME, &parent_node->mtim);
462
463         // Update hash table.
464         set_node(path, node);
465
466         return 0;
467 }
468
469 int tfs_remove_node(const char *path) {
470         struct tfs_node *node = get_node(path);
471         struct tfs_node *parent_node = get_directory(path);
472
473         // Don't rm -rf / -_-
474         if (!parent_node)
475                 return -ENOTSUP;
476
477         // Remove from parent.
478         DEFINE_BLOCK_CURSOR(cursor, parent_node);
479         nodoff_t last_child =
480             BLOCK_NODES(block_seek(&cursor, parent_node->nblocks - 1))[(parent_node->nlink - 1) % BLOCK_MAX_CHILDREN];
481
482         for (blkoff_t block = block_seek(&cursor, 0); block != END_BLOCKS; block = next_block(&cursor)) {
483                 for (int i = 0; i < BLOCK_MAX_CHILDREN; i++) {
484                         if (BLOCK_NODES(block)[i] == NODENO(node)) {
```

```
                                BLOCK_NODES(block)[i] = last_child;
                                goto outer;
                        }
                }
        }
outer:
        parent_node->nlink -= 1;
        tfs_node_trim(parent_node);
        clock_gettime(CLOCK_REALTIME, &parent_node->mtim);

        // Deallocate blocks.
        node->size = 0;
        tfs_node_trim(node);

        // Deallocate node.
        node->next = *tfs_info.free_node_head;
        *tfs_info.free_node_head = NODENO(node);

        // Remove from hash table.
        set_node(path, NULL);

        return 0;
}

int tfs_destroy() {
        // Write back changes to disk.
        return munmap(tfs_info.base, tfs_info.filesize);
}
```

Listing A.4: `mktfs.c`

```
#include "tfs.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
        int ret = 0;

        if (argc < 2) {
                fprintf(stderr,
                        "usage: %s <file>\n"
                        "\n"
                        "Allocate space to a file using fallocate(1) first.\n",
                        argv[0]);
                return 1;
        }

        ret = tfs_open(argv[1]);
        if (ret)
                return ret;

        tfs_format();
        tfs_destroy();

        return 0;
}
```