CS350: Assignment 3 – File Systems

Tavian Barnes, Emil Tsalapatis

1 Introduction

For this assignment, we will improve CastorOS's file system (called O2FS) to handle large files and directories. The current implementation does not support files larger than a few megabytes. Raising this limit will require changes to the kernel's file system implementation as well as the userspace `newfs_o2fs` tool which builds the disk image when you run `scons`.

2 Background: File Systems

File systems are the primary way that users and programmers interact with storage devices (hard drives, SSDs, microSD cards, etc.) File systems organize storage into individual *files* which are grouped together into *directories* (*folders*, on Windows). Directories can be nested within each other to form human-readable *paths* like `/home/user/Downloads/a3.pdf`. The main goal of most file systems is to ensure that the data survives crashes and reboots, a property called *persistence*.

Users can navigate the file system with shell commands like `cd` and `ls`, or with graphical interfaces like Finder (macOS) or Explorer (Windows). Programmers interact with the file system through system calls. On UNIX-like operating systems, these system calls use **file descriptors**. File descriptors tell the kernel which file we'd like to operate on. Internally, the kernel keeps a table that maps each file descriptor to a data structure that keeps track of details like where the file is on disk, and the current read/write offsets inside it.

Listing 1: Using file descriptors to interact with the file system.

```c
// Open a file for reading (RD) and writing (WR)
int fd = open("/usr/etc/example.conf", O_RDWR);
// From now on, all uses of ‘fd’ refer to that file

// Read and write up to 64 bytes of the file
char buf[64];
ssize_t bytes = read(fd, buf, sizeof(buf));
write(fd, buf, bytes);

// Close the file
close(fd);
// From here on, ‘fd’ no longer refers to that file
```

2.1 Offset-to-block translation

Files appear to be a contiguous stream of bytes, but they are not necessarily stored contiguously on disk. Storage devices store data in fixed-size *blocks*, and each block of a file may be stored in a different disk block. For example, the first 512 bytes of a file could be stored in disk block 123, and the next 512 bytes could be stored in disk block 321. When a user reads a range of the file like [504, 520), the file system knows to find the first half of the data in block 123, and the second half in block 321.

Listing 2: Reading a specific range from a file.

```
// Open the file read-only
int fd = open("/usr/etc/example.conf", O_RDONLY);
// Seek to the 504th byte
lseek(fd, 504, SEEK_SET);
// Read 16 bytes (up to byte 520)
char buf[16];
read(fd, buf, sizeof(buf));
```

Every file system uses a different data structure to map file offsets to disk blocks. Modern file systems tend to use (variants of) B-trees, but O2FS uses a simple array. The array has a fixed size of 64, meaning O2FS cannot represent files larger than 64 blocks. With the default O2FS block size of 16 KiB, this limits files to a maximum of 64×16 KiB = 1 MiB. In this assignment, you will change the data structure O2FS uses for block mapping, increasing the maximum file size to 64 MiB.

3 Changing the On-Disk Format

When you build CastorOS with `scons`, a disk image is generated in the path `build/bootdisk.img`. This image contains an O2FS file system that holds all of the files you see from within CastorOS. The image gets generated by the `newfs_o2fs` command, then gets read and written by the CastorOS kernel when you boot it up. For this to work, both `newfs_o2fs` and the kernel must agree on the *on-disk format* of the file system.

The header `sys/fs/o2fs/o2fs.h` defines the on-disk format of O2FS. It is shared between `newfs_o2fs` and the kernel to keep the on-disk format in sync. The structures from that header exactly match the binary layout of parts of an O2FS file system image. Here are some important parts:

Listing 3: O2FS on-disk format.

```
// Super block
typedef struct SuperBlock {
    ...
    uint64_t          blockSize;    /* Block Size in Bytes */
    ...
} SuperBlock;

// Block Pointer: Address raw blocks on the disk
typedef struct BPtr {
    uint8_t           hash[32];
    uint64_t          device;
    uint64_t          offset;
    uint64_t          _rsvd0;
    uint64_t          _rsvd1;
} BPtr;
```

```
// Maximum number of direct blocks
#define O2FS_DIRECT_PTR      (64)

// Block Nodes: Contain pointers to pieces of a file
typedef struct BNode {
    uint8_t          magic[8];
    uint16_t         versionMajor;
    uint16_t         versionMinor;
    uint32_t         flags;
    uint64_t         size;

    BPtr             direct[O2FS_DIRECT_PTR];
} BNode;
```

The *super block* contains global information about the whole file system, including the disk block size. A *block pointer* (`BPtr`) points to a specific disk block. `BPtr::device` specifies which disk device contains the block; since we only have one disk, it is mostly unused. `BPtr::offset` specifies the on-disk offset of the pointed-to block.

A *block node* (`BNode`) represents a single file. The `BNode::direct` array maps *file blocks* to *disk blocks*. These are called *direct pointers* because each pointer directly points to some file contents on disk. Mapping a file offset to a disk offset looks like this:

Listing 4: Mapping direct blocks with O2FS.

```
uint64_t O2FS_MapDirect(SuperBlock *sb, BNode *bn, uint64_t off) {
    uint64_t bs = sb->blockSize;

    // Bytes [0,    bs) are stored at bn->direct[0].offset
    // Bytes [bs, 2*bs) are stored at bn->direct[1].offset
    // etc., so divide by ‘bs’ to get the block number
    uint64_t i = off / bs;

    assert(i < O2FS_DIRECT_PTR);

    // The start of the block that contains the requested offset
    uint64_t block = bn->direct[i].offset;

    // The offset within the block should look like
    // [0, 1, ..., bs-2, bs-1, 0, 1, ..., bs-2, bs-1, 0, 1, ...]
    // so we can use the modulo (%) operator
    uint64_t block_off = off % bs;

    return block + block_off;
}
```

The length of the `BNode::direct` array limits the maximum size of a file. To raise the limit, we could just lengthen this array, but that would waste space for small files. A 64 MiB file needs 4,096 block pointers, meaning the `direct` array would take up 256 KiB by itself. Almost all of it would be unused by small files.

Like all[1] problems in computer science, we can solve this with an additional layer of indirection. Specifically, we will introduce *indirect blocks*: disk blocks containing nothing but an array of block pointers. **Make the following changes to `sys/fs/o2fs/o2fs.h`:**

Listing 5: The new on-disk layout.

```
// New macro: the number of *indirect* blocks in a file
#define O2FS_INDIRECT_PTR         (64)

typedef struct BNode {
    ...
    // Rename this field, and update the array bounds
    BPtr                indirect[O2FS_INDIRECT_PTR];
} BNode;

// New struct: an indirect block
typedef struct BInd {
    BPtr                direct[O2FS_DIRECT_PTR];
} BInd;
```

The new layout works like a multi-dimensional array. Given indices `i` and `j`, we would find the corresponding block like this:

Listing 6: Interpreting indirect blocks.

```
// Read the 'i'th *indirect* block
uint64_t indirect_off = bn->indirect[i].offset;
BInd *indirect = O2FS_ReadBlock(indirect_off);

// Get the 'j'th *direct* block
uint64_t direct_off = indirect->direct[j].offset;
```

Tip: The indices (`i`, `j`) correspond to the `n`th block, at the byte offset `off` within the file, where

```
  n = i * O2FS_INDIRECT_PTR + j,
off = n * sb->blockSize.
```

How would you compute `n`, `i`, and `j`, given the byte offset `off`?

The *type* of the `BNode` block array didn't change, but its *meaning* did: each entry now points to an indirect block. Inside those indirect blocks, another array of direct blocks tells us where the file's contents are on disk. Renaming the array will cause a compiler error on any code using the old meaning, showing us all the places we need to fix. The rest of the assignment will involve fixing those errors.

[1] Almost all

4 The File System Creation Tool

The `newfs_o2fs` tool is implemented in `sbin/newfs_o2fs/newfs_o2fs.c`. It has two important utility functions to be aware of:

Listing 7: `newfs_o2fs.c` utility functions.

```
// Adds a new block to the file system.
// Returns the offset of the new block.
uint64_t AppendBlock(const void *buf, size_t len);

// Overwrites an existing block on the file system.
void FlushBlock(uint64_t offset, const void *buf, size_t len);
```

They can be used like this to add new `BNode`s to the disk image, or overwrite existing ones:

Listing 8: Creating and updating `BNode`s.

```
// Create a zero-initialized BNode in memory
BNode node = {0};
// Add it to the file system
uint64_t offset = AppendBlock(&node, sizeof(node));

// Update the BNode in memory
node.size = 100;
// Update the BNode on disk
FlushBlock(offset, &node, sizeof(node));
```

There are two functions that we will need to update for the new on-disk format: `AddFile` and `AddDirectory`. We will start by updating `AddFile`, which copies an existing file on the host into the O2FS file system image. The current implementation reads one block of the source file at a time, copies it to the image with `AppendBlock`, and appends the new block to the direct block array.

Listing 9: `AddFile` main loop.

```
ObjID *AddFile(const char *file) {
    int i = 0;
    BNode node;
    ...
    while (1) {
        int len = ReadBlock(fd, tempbuf, blockSize);
        // [error checking omitted]
        ...
        node.direct[i].device = 0;
        node.direct[i].offset = AppendBlock(tempbuf, len);
        node.size += (uint64_t)len;
        i += 1;
    }
    ...
    uint64_t offset = AppendBlock(&node, sizeof(node));
    ...
}
```

Change this code to add each new block of file contents to an *indirect* block. When the indirect block is full, add it to the indirect block array, and start working on a new indirect block. This pseudo-code may help get you started:

Listing 10: `AddFile` with indirect blocks.

```
ObjID *AddFile(const char *file) {
    int i = 0, j = 0;
    BInd ind = {0};
    BNode node;
    ...
    while (1) {
        ...
        // [delete the node.direct[i] updates]
        node.size += (uint64_t)len;
        ind.direct[j].device = 0;
        ind.direct[j].offset = AppendBlock(tempbuf, len);
        j += 1;
        if (/* the indirect block is full */) {
            /* append the indirect block to the disk image */
            /* set node.indirect[i] to the indirect block offset */
            /* clear ‘ind’ with zeros to re-use it for the next indirect block */
            i += 1;
            j = 0;
        }
    }
    if (/* the last indirect block is not empty */) {
        /* append that indirect block too */
    }
    ...
}
```

The other function we need to update is `AddDirectory`, which creates directories in the O2FS image. O2FS directories fit in a single block, so the code to create their `BNode`s is simpler:

Listing 11: `AddDirectory` `BNode` creation.

```
ObjID *AddDirectory() {
    ...
    // Write Inode
    memset(&node, 0, sizeof(node));
    memcpy(node.magic, BNODE_MAGIC, 8);
    node.versionMajor = O2FS_VERSION_MAJOR;
    node.versionMinor = O2FS_VERSION_MINOR;
    node.size = size;
    node.direct[0].device = 0;
    node.direct[0].offset = offset;
    uint64_t nodeoff = AppendBlock(&node, sizeof(node));
    ...
}
```

Change this code to create an indirect block with `ind.direct[0]` pointing to `offset`, then point `node.indirect[0]` at the new indirect block. With these modifications the image creation tool should compile and generate a correct CastorOS O2FS image. However, we must also modify the file system itself to properly read and write to the updated image. In the next sections we will be updating O2FS to do just that.

5 The Kernel Read Path

We will now update the CastorOS kernel to be able to read the new O2FS on-disk format. We will be working completely within the `sys/fs/o2fs/o2fs.c` file that holds the kernel's O2FS file system implementation. Functions on the read path like `O2FS_Read` and `O2FS_Lookup` rely on the `O2FSResolveBuf` helper function to read a particular block from a file. Fixing this function is enough to fix the entire read path.

Listing 12: `O2FSResolveBuf` implementation.

```c
// Resolve block number 'b' within a file.
// Returns 0 on success, with *dentp set to the BufCacheEntry.
int O2FSResolveBuf(VNode *vn, uint64_t b, BufCacheEntry **dentp) {
    BufCacheEntry *vnent = (BufCacheEntry *)vn->fsptr;
    BufCacheEntry *dent;
    BNode *bn = vnent->buffer;
    int status;

    status = BufCache_Read(vn->disk, bn->direct[b].offset, &dent);
    if (status < 0)
        return status;

    *dentp = dent;

    return status;
}
```

This function uses the *buffer cache* to read from the disk. The buffer cache exists because disk I/O is slow. Using the buffer cache, we only have to read any given block from disk once. Future reads of the same block will (hopefully) hit the cache and not require I/O. The buffer cache has this API:

Listing 13: Buffer cache API.

```c
// Read a block from a disk.  Returns 0 on success, with *entry set to the new
// buffer cache entry.  (*entry)->buffer contains the actual bytes read from the
// disk.  Once you're done with the entry, call BufCache_Release() to free it.
int BufCache_Read(Disk *disk, uint64_t diskOffset, BufCacheEntry **entry);

// Write a buffer cache entry back to the disk.  Until you call this function,
// any updates to entry->buffer will not be saved.  Returns 0 on success.
int BufCache_Write(BufCacheEntry *entry);

// Release a buffer cache entry.
void BufCache_Release(BufCacheEntry *entry);
```

Modify the `O2FSResolveBuf` implementation to account for indirect blocks. Refer to page 4 for how to compute the appropriate array indices. This pseudo-code may help:

Listing 14: `O2FSResolveBuf` pseudo-code.

```
int O2FSResolveBuf(VNode *vn, uint64_t b, BufCacheEntry **dentp) {
    ...
    size_t i = /* indirect block index for ‘b’ */;
    size_t j = /* direct block index for ‘b’ */;

    BufCacheEntry *ient; // Indirect block bufcache entry
    status = BufCache_Read(vn->disk, /* indirect block ‘i’ */, &ient);
    // [error checking]

    BInd *ind = ient->buffer; // The indirect block itself
    status = BufCache_Read(vn->disk, /* direct block ‘j‘ */, &dent);
    // [error checking]

    // Release ‘ient’

    *dentp = dent;
    return status;
}
```

6 The Kernel Write Path

Finally, we modify the write path to allow CastorOS to write large files to the file system. For this we must modify two calls in the O2FS file system, `O2FS_Write` and `O2FSGrowVNode`. The `O2FS_Write` call moves the data from kernel memory to the buffer cache and initiates the IO. `O2FSGrowVNode` is a helper routine that adds new blocks to the file when a write grows the file enough to allocate more space on the disk. `O2FS_Write` calls `O2FSGrowVNode` when necessary. `O2FS_Write` uses the `O2FSResolveBuf` function to find the right disk block, and we have already adapted it for indirect blocks, so we do not need to directly modify the function.

We only need to adjust `O2FSGrowVNode` that the write call uses to expand the file. The file system allocates a block to a file the first time the block is about to be written to. The function currently assumes all blocks are direct blocks, so we need to adjust it to allocate indirect blocks.

The function is relatively short:

```
    if (filesz > (vfs->blksize * O2FS_DIRECT_PTR))
        return -EINVAL;

    for (int i = blkstart; i < ((filesz + vfs->blksize - 1) / vfs->blksize); i++) {
        if (bn->indirect[i].offset != 0)
                continue;

        uint64_t blkno = O2FSBAlloc(vfs);
        if (blkno == 0) {
                return -ENOSPC;
        }

        bn->indirect[i].offset = blkno * vfs->blksize;

    }
```

The function first tests if the file can be resized to the requested size, or if its beyond the system limit. The existing code assumes a translation array of 64, so a file size larger than 64 blocks is impossible. The code then goes through the translation array for all blocks up to the requested size. For each empty entry the routine allocates a disk block using `O2FSBAlloc` and adjusts the translation array. The function then adjusts the file size of the BNode and writes it out using `BufCache_Write`.

The main change to `O2FSGrowVNode` is in the inner main loop. First, as we iterate we must read the BNode's indirect blocks to check whether there are direct blocks allocated for a given file offset. Every time we allocate a new direct block we must also update the indirect block. We must update indirect blocks to the disk as we modify them to persist the changes we have made. If an indirect block is not allocated, we must create it and attach it to the BNode. Finally, we adjust the file size limit check in the beginning of the function to be `vfs->blksize * 64 * 64`, to reflect the larger number of blocks a file can access.

We provide the pseudocode for the main loop in the next page. With this addition the file system should be able to write files up to 256 MiB.

Algorithm 6.1: Indirect Blocks for `O2FSGrowVNode`

```
for file block offset in [current file end, requested file end] do
    calculate the indirect, direct offsets from the block offset
    if indirect block unallocated then
        allocate indirect block
        BufCache_Read(indirect block number, ient)
        zero out indirect block
        attach indirect block to vnode
    else
        BufCache_Read(indirect block number, ient)
    ind ← ient.buffer
    if direct block allocated then
        BufCache_Release(ient)
        continue
    allocate direct block
    insert direct block number to indirect block
    BufCache_Write(ient)
    BufCache_Release(ient)
```

7 Testing and Submitting Your Work

Submitting Assignment 3 works just like Assignment 0. Use `git commit` to create a single commit with your Assignment 3 solution (on top of your previous commits with your Assignment 0, 1, and 2 solutions). **The commit should only include `sbin/newfs_o2fs/newfs_o2fs.c` and `sys/fs/o2fs/o2fs.[ch]`. If the commit includes any other files, the server will automatically reject the submission.** Next, follow these steps to generate and submit your Assignment 3 patch.

```
$ python client.py patch
$ python client.py submit -a asst3
```

The status of your submission can be monitored using the `client.py status` command:

```
$ python client.py status -a asst3
TOTAL: 9/9
Evaluated at 11/11/2024 12:00
=========END OF SUBMISSION=========
```

We will be evaluating your work using three tests built into the CastorOS image. These tests are `fiotest`, `writetest`, and the newfs command to build images. The source for these tests is in the `tests/` directory in the `castoros` repository.

These three tests evaluate our implementation of the file system IO paths and that the image is generated such that it boots. If your assignment is correct you should be able to support 20MB files.