CS350: Assignment 3 – File Systems

Emil Tsalapatis

1 Introduction

In this assignment, we will improve CastorOS's file system to handle large files and directories. The current implementation does not support files larger than a few MiBs.

This document begins with a design overview of file systems. The overview explains how file systems are used by users and programmers. We then present CastorOS's built-in file system, called O2FS. It also lists O2FS's current file size limitations and outlines the work required to fix them.

O2FS does not support large files, so we cannot write more than 1 MiB into a single file. We will change the file format in the OS to allow it to store large files. Since we will be modifying the file format of the file system we will also need to change the tool we use to create the `.img` file at compilation time.

2 Background: File Systems

File systems are kernel subsystems that store data on the disk. File systems use files and directories to organize our data and let us access it using human-readable file paths. The main goal of most file systems is to ensure that the data survives crashes and reboots, a property called persistence.

File systems implement most system calls that use **file descriptors**. File descriptors are handles, integers to be exact, that the kernel gives to userspace applications when they open a file using a system call like `open`. The application uses file descriptors to inspect and modify files using system calls like `read` and `write` that take a file descriptor as arguments. The application releases the file descriptor when it is done reading and writing to it.

Listing 1: Using `open` and `close` to create and destroy a file descriptor.

```
int file_descriptor_example()
{
   /* Open the file for reading and writing. */
   int fd = open("/usr/etc/example.conf", O_RDWR);

   /* Operate on the file and exit. */
   read(...);
   write(...);
   close(fd);
}
```

2.1 Offset-to-block translation

The file API lets us address our data using a combination of file paths and numerical offsets that transparently translate these coordinates into locations on the disk. For example, let's consider a hypothetical configuration file named `/usr/etc/example.conf` on our local disk. File systems allow us to read data without knowing its on-disk location. We instead use `open()` to open it, then make a `read()` system call to read the file from the beginning. We can also skip the beginning of the file and directly access any location by specifying a **file offset** that we want to read from our program. We can either do this by seeking to the position using `lseek` or `pread`.

Listing 2: Using the file API to read a file starting from a specific offset.

```c
int read_100th_char()
{
   char c;

   /* Open a file as read only. */
   int fd = open("/usr/etc/example.conf", O_RDONLY);
   pread(fd, &c, sizeof(c), /* offset */100);
   close(fd);
}
```

File systems automatically translate file names and offsets to on-disk data structures by tracking where on disk each file's data lies. All file data is stored as a collection of **disk blocks**, contiguous fixed-sized regions on the disk that hold file data. The data structure that the file system uses to track each file's disk blocks is called an `inode`. Each file's inode includes a mapping between file offsets and on-disk blocks and uses this mapping to read and write at the right disk locations.

O2FS currently implements the mapping using an array, but every file system uses different data structures. The choice of data structure influences the file system's semantics and performance. In this assignment, we will be replacing the array used by O2FS with a more efficient data structure.

3 Objective 1: Adding Large File Support

Our first task is to add support for large files to our system. O2FS currently handles file sizes up to 1 MiB because it uses a fixed-size array to translate between file offsets and disk blocks. The size of the array limits the maximum size of the file itself, so we must replace it with a more flexible data structure. Increasing the array size would allow us to have larger files, but would be wasteful for smaller files that do not need the whole array. We will support large files without wasting space by replacing the array with using **indirect blocks**.

To implement large file support for O2FS we must first understand how it is currently implemented. Every file inode has a fixed-size array of integers where it stores file-to-block translations. The offset of each array element in the array models its offset in the file, while the value of the element is the on-disk location of the block.

For example, the 0th element of the array is `0xbaba00`, so the on-disk location of the file data corresponding to offsets 0–16383 bytes are located on the disk block `0xbaba00`. The disk blocks that the file uses on the disk need not be contiguous; for example, the same file may have a value of `0xcdcd00` for the 1st array element, meaning that bytes 16384–32767 of the file are stored in the disk block with offset `0xcdcd00`.

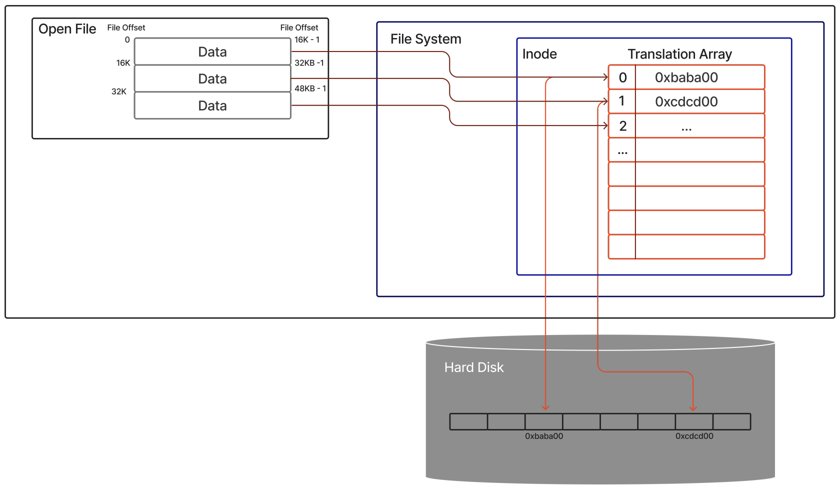

Figure 1: The translation process from file offsets to disk blocks. Each file organizes its data in the kernel into chunks of 16 KiB bytes. The file system keeps an inode for the file that it uses to match a file chunk to an on-disk block. The file system divides the file offset of an IO operation by the size of the block to find the right index within the translation table, then reads the disk block address held in that element of the array. The file's disk blocks on the disk may not be contiguous.

The size of a file is limited by the size of the translation array because the size of the array limits the maximum file offset that it can translate. For example, if the array has a length of 64 and its block has 16 KiB bytes in it, then the maximum file size is $64 * 16\,\mathrm{KiB} = 1\,\mathrm{MiB}$.

Increasing the array size would be wasteful because most files are small and do not need the full array. For example, let's assume that the disk block size is 512 bytes. If we increased the size of the array from 64 to 128 entries we would need an additional sector for every file since the array is of type `long[]` and would need $sizeof(long) * 64 = 512\,\mathrm{B}$ more space. Most file data is small enough to fit into a single block, so they currently take up 2 blocks on the disk - one for the inode and one for the data. With the new array they would take up 3 blocks, a 50% increase in size. In the worst case, we would reduce the amount of data we can store on the disk by 33%.

Indirect blocks allow us to increase the maximum file size without increasing the size of the file metadata of most files. Indirect blocks are self-contained disk blocks that only offset-to-block translations. We change the array translation table so that its entries point to indirect blocks instead of directly storing the on-disk address of a block. This layer of indirection lets us store much larger files without expanding the inode itself. Each indirect block is 512 bytes in size and can thus hold 64 entries, while the inode points to an indirect block per array entry. This in turn means that we can now store files 64 times larger than we could before.

Switching to indirect blocks means we must adjust the offset to block translation process. The file system still divides the file offset by the block size to find the block number. The system then must find both which indirect block the address translation belongs to, and the offset within the indirect block it must look into. We find the indirect block by dividing the block number by the number of disk block entries in each indirect block, 128 in this example. We then find the offset within the indirect block by taking the remainder of the block number with the number of disk block entries in each block. In short:

$$\begin{aligned} \text{blkno} &= \text{offset} \,/\, \text{bytes\_per\_block} && (1)\\ \text{indirect\_block} &= \text{blkno} \,/\, \text{entries\_per\_indirect\_block} && (2)\\ \text{index} &= \text{blkno} \bmod \text{entries\_per\_indirect\_block} && (3)\\ \text{blkaddr} &= \text{inode\_array}[\text{indirect\_block}][\text{index}] && (4) \end{aligned}$$

Using indirect blocks as described relaxes the limits on file size but is not space-efficient. If we use indirect blocks we must allocate one block for each entry in the array that we use. For example, when we write to the first bytes of a file we must first allocate an indirect block for the first entry of the indirect block array, in addition to the block where we store the file data. Even the smallest files therefore take up three blocks on the disk, one data block, one indirect block, and one block for the inode itself. However, the implementation of this scheme is simple so we will be using it for this exercise.

The solution in real file systems is to directly store the disk block address of the first few blocks directly in the translation array the same way as the initial O2FS. Block addresses corresponding to higher file offsets are stored in indirect blocks to enable larger file sizes. This scheme combines the advantages of both direct and indirect block addressing by keeping small files compact, while also keeping the file limit larger at a small storage cost. This scheme is however also slightly more difficult to implement, so we will not be using it here.

Code Additions

Switching to indirect blocks requires that we modify both the file system itself and the file system generation tool used during compilation (see Section 1). We will start with the file generation tool, whose code we can find in sbin/newfs_o2fs/newfs_o2fs.c. We will then modify the file system to store the addresses of indirect blocks in the inode instead of the direct disk block addresses. The file system is entirely contained in sys/fs/o2fs/o2fs.c.

Expanding the type definitions First, we must change the layout of O2FS's inode-related data structures in sys/fs/o2fs/o2fs.h. The O2FS file system calls its inodes struct BNode, so we will be referring to them as such from now on. These BNodes currently hold the block translation array and various other inode metadata. The pointers are of type BPtr. This type includes a device field that represents which disk the pointer's data is in, and an offset field that holds the logical block address within the disk for the data. For this assignment, we will be ignoring all fields apart from those two, and will always set the device field to 0.

Listing 3: lstlistingThe initial O2FS disk pointer (BPtr) and inode (BNode) definitions .

```
/*
 * 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;

/*
 * Block Nodes: Contain indirect 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[64];
} BNode;
```

We will be making two modifications to the file. First, we will be renaming the direct field in BNode to indirect. This change is non-functional but documents the fact that the pointers now refer to indirect blocks. Renaming the field also turns all references to this field from other files into compiler errors, so it helps us track down and fix all the code we will be invalidating when we change the

BNode layout. The second change is to add a struct that represents the indirect node.

After we change the prototype we change the file creation tool to create a file system that uses indirect nodes. The new structure will be called `BInd` (Block Indirect) and will have a single field, a 64-element array of `BPtr` called `direct`. We will be using this struct to represent the indirect blocks in memory.

A small note regarding the types: The on-disk addresses of the indirect block pointers are *still* type `BPtr`, even though we just made a new type for them. The reason is that `BPtr` is an on-disk address, while `BInd` is an in-memory data type. We will see later in this assignment how we read data from the disk and turn it into an in-memory data structure.

The FS creation tool: `AddFile` Next, we update the FS creation tool in `sbin/newfs_o2fs/newfs_o2fs.c` to generate a file system that uses indirect blocks. The tool generates the hard disk image that holds the CastorOS kernel, programs, and files that we see when we run QEMU. The tool uses O2FS to format the image and has the code required to directly read and write core O2FS data structures into the disk. The tool currently creates BNode instances that use direct nodes, so we will be changing it to generate indirect nodes instead.

We will be modifying the two functions that create BNode instances, `AddFile` and `AddDirectory`. These functions create files and directories, both of which the file system represents as BNode instances. We will be starting with the `AddFile` function which is slightly simpler, going through it step by step before attempting to modify it. We will skip some lines that mostly hold boilerplate.

```
ObjID *AddFile(const char *file)
{
```

The function takes a filename in our local machine (in the CastorOS source directory to be exact), and inserts it into the O2FS disk image. The function returns an `ObjID` that is used to create the image's directories, but it is of no concern for us.

```
    memset(&node, 0, sizeof(node));
    memcpy(node.magic, BNODE_MAGIC, 8);
    node.versionMajor = O2FS_VERSION_MAJOR;
    node.versionMinor = O2FS_VERSION_MINOR;
```

This part of the code initializes a BNode that will hold the file data. We will be writing out this BNode directly to the disk.

```
    // Copy file
    fd = open(file, O_RDONLY);
    if (fd < 0) {
        perror("Cannot open file");
        exit(1);
    }
    while (1) {
        int len = read(fd, tempbuf, blockSize);
        if (len < 0) {
            perror("File read error");
            exit(1);
        }
        if (len == 0) {
            break;
        }
```

To copy a file from our local source tree to the CastorOS image we simply open it and read it into a buffer one block at a time. We continue until there is no more data left to read, at which point we have copied the entire file to the disk.

```
        uint64_t offset = AppendBlock(tempbuf, len);

        // Update node
        node.direct[i].device = 0;
        node.direct[i].offset = offset;
        node.size += (uint64_t)len;
        i++;
    }
```

Every time we read a block-sized data chunk into our buffer we use `AppendBlock` to allocate a block on the disk image and write the chunk to it. The function returns the disk block address. For each such address we append the block address to the translation array and adjust the size of the BNode. **This is the part of the function that we will be modifying.**

```
    // Construct BNode
    uint64_t offset = AppendBlock(&node, sizeof(node));

    // Construct ObjID
    id->device = 0;
    id->offset = offset;

    return id;
}
```

At the end of the function, after we are done writing out all the data blocks, we write out the BNode itself. We do so by passing it to `AppendBlock` that simply treats it as a byte buffer that it writes to the disk. We only modify the code snippet above that write out the data to the disk and updates the BNode array. We will replacing it with code based on Algorithm 3.1 on the next page. When implementing the algorithm make sure to allocate a new indirect block during the **first** iteration.

Algorithm 3.1: Modified `AddFile` code

```
1  $BIndbind$ is the indirect block data structure
2  **if** *an indirect block must be allocated* **then**
3      use $AppendEmpty()$ to get a new indirect block
4      zero out $bind$
5      append the indirect block to $node.indirect$
6  $AppendBlock(tempbuf, len)$
7  append the new direct block to $bind.direct$
8  update $node.size$ with the length of the new data
9  use $FlushBlock$ to write $bind$ into the indirect block
10 **if** *The bind.direct is full* **then**
11     allocate an indirect block in the next iteration
```

The FS creation tool: AddDirectory The `AddDirectory` code is for our purposes very similar to `AddFile`, because they write their data to the disk in a similar way. We only need to adjust the later half of the function:

```
// Write Directory
uint64_t size = entry * sizeof(BDirEntry);
uint64_t offset = AppendBlock(entries, entry * sizeof(BDirEntry));
```

This code writes out to the disk the directory data in the `entries` variable. This is equivalent to the chunk writing operation in `AddFile`, only here we are writing out a specific data structure we have constructed in the first half of the function. The code assumes that the data structure's length is smaller than the block size, so it only allocates a single block for it.

```
memset(id, 0, sizeof(*id));
memset(&node, 0, sizeof(node));
memcpy(node.magic, BNODE_MAGIC, 8);
node.versionMajor = O2FS_VERSION_MAJOR;
node.versionMinor = O2FS_VERSION_MINOR;
node.size = dirLength;
```

This snippet sets up the inode state of the directory and is almost identical to the beginning of the `AddFile` function. We will not be modifying it.

```
node.direct[0].device = 0;
node.direct[0].offset = offset;
```

Finally, this snippet updates the BNode translation array with the location of the block we wrote `entry` in. The code only sets up the first entry of the table because the code assumes that all directory data fits in a single block.

We will be replacing the last snippets to adjust for indirect blocks. For our code we also make the simplifying assumption that all the directory contents fit into a single block. We can then assume that we need a single data block and a single indirect block. We collectively replace all of the code above with the following:

Algorithm 3.2: Modified `AddDirectory` code

```
1 BInd bind
2 zero out bind
3 add to bind.direct the disk location of entries
4 write out bind using AppendBlock
5 update node.indirect with the disk location of bind
```

The BNode itself will be written out by the already present `AppendBlock` call at the end of the function.

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 section we will be updating O2FS to do just that.

4 Objective 2: Updating the Read Path

We will now update the read path to handle indirect block pointers. We will be working completely within the `sys/fs/o2fs/o2fs.c` file that holds the implementation for the O2FS file system used by the kernel to read and write to the disk. We will implement the code used to read from the file system first to make sure that we can boot our system. Without fixing the read path the kernel cannot read any data from the file system, so it cannot load up binaries or even mount the root directory, since it cannot read it.

For the read path, we must fix two functions, `O2FS_Read` and `O2FS_Lookup` that reads from a file and resolves filenames, respectively. `O2FS_Read` reads data from the disk by translating a file offset into a disk offset and performs the disk IO operation. `O2FS_Lookup` translates file paths like `/etc/example.conf` into file system inode. This call is used by the `open()` system call that underpins all file IO, and even `spawn()`.

We do not modify the functions themselves, but rather update the helper function `O2FSResolveBuf` that they use to traverse the translation array. This function take an inode and a file block offset and returns the on-disk block that holds the data.

```c
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;
}
```

To modify the existing code we need to understand how it works, which includes how we use the *buffer cache* to cache disk blocks. The buffer cache is a kernel subsystem that caches accesses to and from the disk to minimize IO. For example, when we read a block from the disk the buffer cache keeps a copy of it. If we try to access the same block at a later time the buffer cache will return the copy of the data already in memory, avoiding the cost of doing an IO. Similarly, when writing a disk block to a file the buffer cache keeps a copy to speed up subsequent reads. All IO from the file system is done by interacting with the buffer cache.

The buffer cache has a simple API with three calls, `BufCache_Read`, `BufCache_Write`, and `BufCache_Release`. The file system accesses the data in the buffer cache using `BufCache_Read`. The call takes a disk offset and returns the data held on the disk in that offset. The call will perform a disk read if the data is not currently in the cache. The call returns a `BufCacheEntry` instance that holds the data in its `buffer` field. The caller now holds a reference to the `BufCacheEntry` and must manually release it when done interacting with the data using `BufCache_Release`.

`BufCache_Write` writes modified buffer cache blocks out to the disk. This call must be called on a buffer cache entry that we previously read using `BufCache_Read`.

We will use the buffer cache API to first read the indirect block, then access it as a `BInd` instance to find the direct disk pointer itself. We replace the code in the listing above with the following pseudocode:

Algorithm 4.1: Indirect Blocks for `O2FSResolveBuf`

```
1 calculate the direct and indirect indices from $b$
2 BufCache_Read($indirectblockno., ient$)
3 $bind \leftarrow ient.buffer$
4 BufCache_Read($directno., dent$)
5 Release $ient$
```

5 Objective 3: Updating the 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 `O2FS_GrowVNode`. The `O2FS_Write` call moves the data from kernel memory to the buffer cache and initiates the IO. `O2FS_GrowVNode` 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 `O2_GrowVNode` 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 `O2FS_GrowVNode` 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 `O2FS_GrowVNode` 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 5.1: Indirect Blocks for `O2FS_GrowVNode`

```
for file block offset in [current file end, requested file end] do
    calculate the indirect, direct offsets from the block offset
    if indirect block unalllocated then
        allocate indirect block
        BufCache_Read(indirectblocknumber, ient)
        zero out indirect block
        attach indirect block to vnode
    else
        BufCache_Read(indirectblockno., ient)
    bind ← 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)
```

6 Testing and Submitting Your Work

We will be submitting to the test server the same way as we did for assignment one. We use the `client.py` program with the same username and magic number. We only change the `ASST` variable from `"asst1"` to `"asst3"`. Changing this variable notifies the server that any new submissions will be run against the Assignment 3 tests. For more details on how to submit please refer to Assignment 1's Section 5.

NOTICE: The patch to be submitted to the server should include the changes made to the kernel for assignments 1 and 2. Newer versions of the `client.py` utility include all commited changes to the patch, but some older versions required all changes to be included in a a single commit. Please inspect the generated patch and ensure it includes all your code before submitting, including the code for assignments 1 and 2.

We will be evaluating our 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.