Deliverables

You should deliver a **compressed file** containing all the files originally handed in the skeleton code, completed according to the specifications below.

1 Introduction

In this project, you will implement an inode-based filesystem in C, providing functions to format the filesystem, and read/write based on an inode number.

The module `storage.c/storage.h` emulates a 64MB disk, divided into blocks of 4K. The function `storage_read_block()` reads a given block, copying the data **into** a buffer provided by the user. The function `storage_write_block()` writes a given block, copying the data **from** a buffer provided by the user.

The module `files.c/files.h` has a `format()` function that initializes the following blocks in the disk, creating a primitive filesystem. Please refer to Fig. 1.

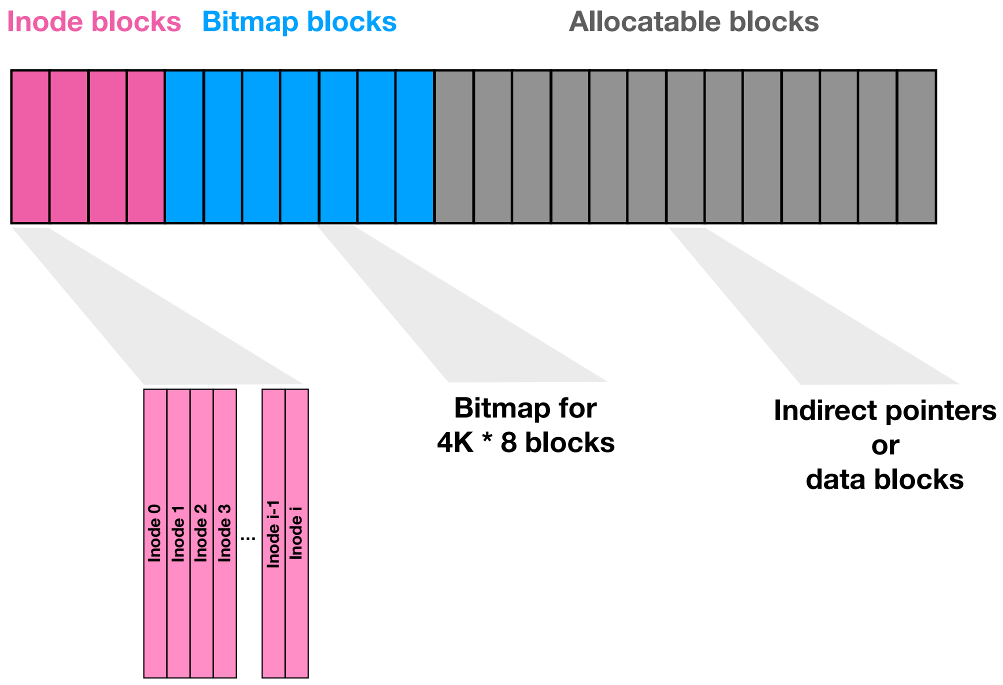

Figure 1: File system overall structure.

Each of the inode blocks contains $i =$ `BLOCK_SIZE / sizeof(inode_t)` inodes, and we will have $I =$ `MAX_FILES / i` inode blocks in total (rounded up). Each of the bitmap blocks contains $b =$ `BLOCK_SIZE * 8` bits, and we will have $B =$ `NUM_BLOCKS / 8` bitmap blocks in total (rounded up).

In this assignment, each inode has three usable fields:

1. `flags_used` (1 bit). Indicates whether the inode is in use or not.
2. `size` (64 bit). Indicates the file size.
3. `head_pointer_block` (64 bit). The number of the (unique) indirect block that each file has associated with itself. Fig. 2.

   This indirect block is just like the indirect block in the UNIX FFS. Indirect blocks are allocated together with the data in the *allocatable blocks* part of the disk.

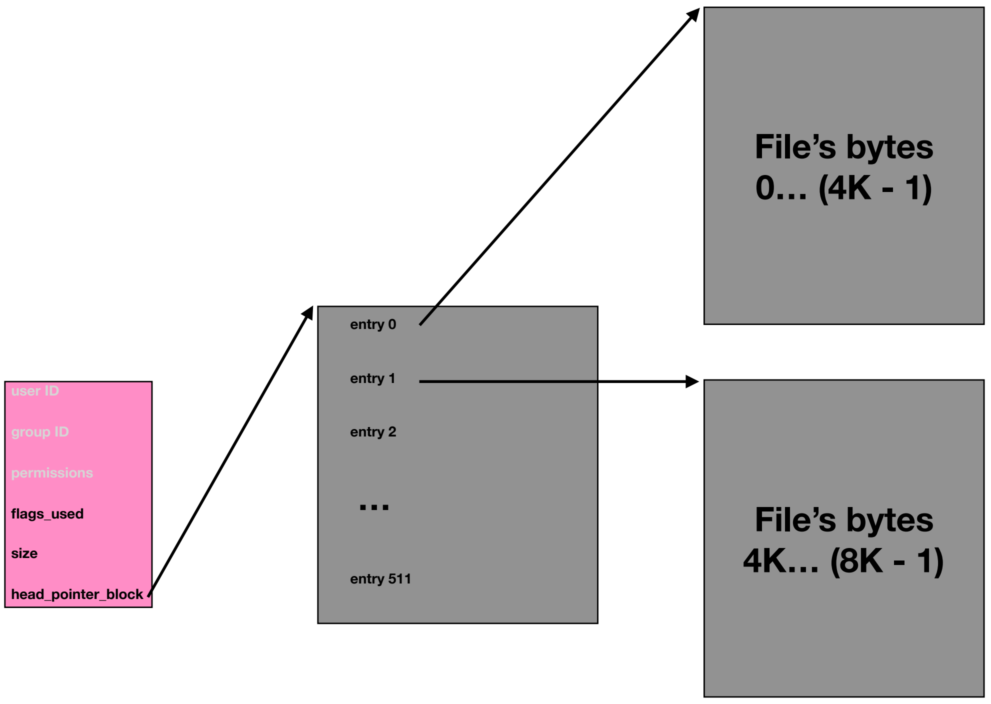

Figure 2: Inodes, indirect blocks, and data blocks.

The module `bitmap.c/bitmap.h` implements a bitmap to allocate or deallocate blocks in the disk. It only allocates blocks form the "Allocatable blocks" area (see Fig. 1). The function `bitmap_allocate_block()` obtains a block among the free blocks available, and accounts for its usage by setting the appropriate bitmap entry. Symmetrically, the function `bitmap_deallocate_block()` releases a block from usage, marking it suitably in the bitmap.

2 Setup

You can implement this homework in any UNIX-like system (including FreeBSD, OpenBSD, Linux, macOS). The easiest programming/debugging setup is using Eclipse and DDD, but you can use any other preferred method.

Create a new, empty C project, and drag the skeleton files into the project's top-level directory. Then, right-click in the project, and choose "Properties". In the window that appears, go in the "Dialect" section, and make sure to tell Eclipse that you are using ISO C11, the recent 2011 revision of the C standard.

3 (25 pts) File Creation

Complete the `ifile_create()` function in `files.c`. This function creates a new file based on a user-provided inode number. The inode is initialized, the inode's (unique) indirect block is allocated and initialized, and a first data block for the file is allocated. More specifically:

1. Read the block storing the `inode_number` provided as parameter. You should start by calculating which inode block contains the `inode_number`, and the offset of the inode within its block.

2. Update the inode: set the file size to zero; set the used flags to true; and finally allocate a new block using `bitmap_allocate_block()`, and set the inode's `head_pointer_block` to point to it. The allocated block will be the (unique) indirect block containing all the file's disk blocks, as indicated in Fig. 2.

3. Write the block storing the `inode_number` back to disk, since we made changes to one of its inodes.

4. Read the indirect block of the inode (the block pointed by `head_pointer_block`), and zeroes all entries, except the first, which should point to a newly allocated **data** block. That block will store the first 4K bytes of the file. You have to allocate the first data block for the file also using `bitmap_allocate_block()`, just as you did to allocate the indirect block itself.

5. Write the indirect block of the inode (the block pointed by `head_pointer_block`), since we made changes to one of its entries.

Please look for the function docstrings `frame.h`, and additional implementation details in `frame.c`, as comments in the body of the function.

4 (25 pts) File Reading

Complete the `ifile_read()` function in `files.c`. This function reads a specified number of bytes (the `how_many` parameter) from the file referenced by the `inode_number parameter`, starting the read from a specified offset (the `from` parameter), copying the data into the buffer referenced by the `buffer` parameter. Specifically, you should:

1. Read the block storing the `inode_number` provided as parameter. You should start by calculating which inode block contains the `inode_number`, and the offset of the inode within its block.

2. Check if the user is requesting to read past the end of the file, in which case you should return -1 to indicate the error. You can check this condition by comparing `(from + how_many)` to the inode's file size.

3. If the read is valid, read the inode's (unique) indirect block, and call the `pointers_read ()` function.

   The `pointers_read()` function will iterate through the file's data blocks (pointed by the indirect block), and read the contents into the user-supplied buffer. You should understand what this function is doing because you are going to write a similar writing counterpart later on in the assignment.

Please look for the function docstrings `frame.h`, and additional implementation details in `frame.c`, as comments in the body of the function.

5 (25 pts) File Writing

Complete the `ifile_write()` function in `files.c`. This function is **very similar** to the `ifile_read()` function that you just implemented. It writes a specified number of bytes (the `how_many` parameter) into the file referenced by the `inode_number parameter`, starting the write at a specified offset (the `to` parameter), copying the data pointed by the `buffer` parameter into the file. Specifically, you should:

1. Read the block storing the `inode_number` provided as parameter. You should start by calculating which inode block contains the `inode_number`, and the offset of the inode within its block.

2. Check if the user is requesting to read past the end of the file, in which case you should call `ifile_grow()` function to extend the size of the file. You can check this condition

by comparing `(from + how_many)` to the inode's file size.

The `ifile_grow()` function, which is conceptually very simple, and has been implemented for you, allocates as much data blocks as needed in order to reach the necessary size for the `ifile_write()` function. The allocated data blocks are not initialized. So, if the file size is 25, and you write to offset 100, bytes between offset 26 and 99 will contain arbitrary data.

3. Since the `ifile_grow()` function changes the inode's size, write the block storing the `inode_number` back to disk.

4. Read the inode's (unique) indirect block, and call the `pointers_write()` function, which is symmetric to the `pointers_read()` function. Implementing the `pointers_write()` function is part of the next part of the assignment.

Please look for the function docstrings `frame.h`, and additional implementation details in `frame.c`, as comments in the body of the function.

6 (25 pts) Implement the Actual Data Writing

Complete the `pointers_write()` function in `files.c`. The `pointers_write()` function will iterate through the file's data blocks (pointed by the indirect block), and write the contents of the user-supplied buffer into them.

Please look and understand completely the `pointers_read()` function, and implement the `pointers_write()` function as a symmetric operation. Make sure to write the data blocks back to disk after you updated their contents.

7 The Success String

In `tester.c`, a program formats a filesystem and creates two files, updating one of them twice. You pass the test case if your program prints:

```
READ 0: Hello filesystem world!
READ 1: It's me again! Hello, hello, hello!
READ 1 (after overwrite): LALA me again! Hello, hello, hello!
READ 1 (from high offset): HIGH_OFFSET Hello!
```

8 Style Deductions

I may deduct up to 15% of every grade item to account for bad style, which includes:

1. Poor indentation;
2. Cryptic variable names;
3. Poor error treatment;
4. Naming style inconsistencies (adopt one style with your partner and stick to it);
5. Overly-complicated code.

Good luck,
- Hammurabi