CS 343 Operating Systems, Winter 2026
Filesystem Lab: Implementing an Inode-based Filesystem

Contents

1 Introduction

Motivation The purpose of this lab is to familiarize you with the inner workings of a file system by implementing parts of one yourself. Filesystems are heavily dependent on indirection and metadata management in order to govern data in an organized manner.

In this lab, you will implement parts of the `WTFS` filesystem in the Nautilus Kernel (NK). This filesystem is capable of creating and deleting files and directories, reading and modifying files, as well as retrieving metadata about those files. It is split into logical layers in order to provide structure for easier implementation of complex disk operations. By the end of the lab, you will have implemented portions of nearly every layer, giving you a solid understanding of inode-based filesystems.

You may work in a group of up to three people in this lab. Clarifications and revisions will be posted to the course discussion group. This is ***NOT*** a trivial lab. Our own solution represents a diff (`git diff`) of about 1200 lines compared to the starter code. It is **recommended that you start early.**

Winter 2026 This lab was initially developed by Branden Ghena and staff for the Fall 2025 iteration of the class. In the Winter 2026 iteration, we are adopting the lab and extending it in the following ways:

- The Filesystem Lab previously used only in-memory disk images ("RAM disks"). It has been extended to include persistent disk images as well. Part of the challenge with filesystems design is that filesystems contain stored data structures that must be maintained in a consistent manner (remain sane) over the very long term.
- Persistent disk images will be provided using an NVMe block device driver. NVMe is how high performance SSDs are interfaced. Our goal here is to make it possible for you to follow filesystem operation through the block layer and all the way down to the level of the device drivers that operate the block storage devices that ultimately provide persistence. Additional block device drivers (ATA, virtio-block) may also be used.
- We will trial various other enhancements to the Filesytem Lab.

Because of potential new features, bug fixes, and the like, we may send you patches or pull requests over the course of the lab.

We also plan to extend the Paging Lab (the last lab in the class) so that you will implement user-level processes that are based on executable files stored in persistent disk images accessed via your filesystem.

2 Setup

You can work on this lab on any modern Linux system, but our class server, Moore, has been specifically set up for it, and this is also where we will evaluate it. We will describe the details of how to access the lab repo via Github Classroom on Piazza. You will use this information to clone the assignment repo using a command like this:

```
server> git clone [url]
```

This will give you the entire codebase of the Nautilus kernel framework ("NK"), just as in the Getting Started Lab. As before, you may want to use `chmod` to control access to your directory.

Important! You will need to make sure you have a valid display for NK to run. You can get that through FastX or with `ssh -Y`. See the Getting Started Lab for more details. You also learned how to use the gdb with NK in the Getting Started Lab. You'll want to leverage that knowledge in this lab!

You can now boot your kernel: (this will also build the kernel if needed, so don't worry about running `make` every time)

```
server> ./run
```

The `run` command will execute the emulator (QEMU) with a set of options that are appropriate for the lab. The emulated machine will boot NK, and if all is successful, you will see a blue screen with a red prompt, just as in the Getting Started Lab. Remember that the shell you are talking to is within the kernel itself.

3 What the File System? Overview

`What the File System?` is a pedagogical filesystem built within the Nautilus kernel (NK). Being an inode-based filesystem, all files (including directories) have one inode that contains metadata for the file/directory. The system is made up of a variety of **disk blocks** which we go over below. The layout of a `WTFS`-formatted disk is as follows:

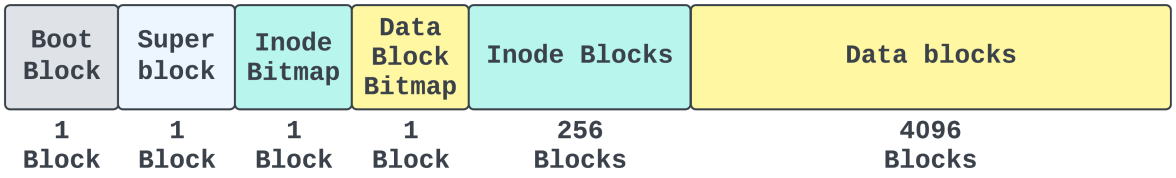

Figure 1: `WTFS`-formatted Disk Layout

3.1 Disk Layout

With all of the blocks added up and a block size of 512 bytes, the total size of the disk is only about 2.2 MB. This is tiny for a filesystem, but suits our purposes for a lab.[1] The following paragraphs go over every type of **disk block** found in the system. For a more detailed and complete diagram, see Appendix B. [2]

Boot Block The boot block contains an identifier for the filesystem, to be used to check the disk is formatted correctly on boot. You do not need to do anything with the boot block in this assignment.

Superblock The superblock contains metadata describing the filesystem. One useful field is **s_block_size**, which dictates the block size used throughout the filesystem (512 bytes). This is a field you will likely want to use in your implementations.

Bitmaps Bitmaps are responsible for tracking which inodes/data blocks are in use in the filesystem, with each bit representing a free (0) or taken (1) inode/data block. `WTFS` has two of these bitmaps, one tracking inodes and one tracking data blocks. Each bitmap occupies a full block, which means that the inode bitmap can track 4096 inodes and the data block bitmap can track 4096 data blocks. These values impose a limit on the total size of the filesystem.

Inode Blocks An inode contains metadata for a file on disk, such as size, permissions, and references to the file's data blocks. Each Inode is 32 bytes in size. Since each disk block is 512 Bytes in size and the inode bitmap tracks 4096 inodes, there are 256 blocks set aside to hold the system's inodes (16 inodes per block).

[1] Typically, a filesystem can be *parameterized* when a block device is *formatted* using the filesystem. Parameters include the block size, the number of blocks, the number of inodes, the version of the filesystem, etc. To put this into historical context, the FAT filesystem commonly used today for gigantic SD cards, thumb drives and the like, started as "FAT12", with support for 2^{12} = 4096 blocks, just like WTFS. The current version ("FAT32/exFAT") uses 2^{32} =about four billion blocks.

[2] Some conceptual questions to think about :) What is the maximum possible size in bytes of a file in WTFS? What's the maximum number of files that could go in a single directory? How many bytes must a WTFS disk image be in size?

Data Blocks The remainder of the filesystem is made up of 4096 data blocks. These data blocks hold the data for files. However, they can **ALSO** be used for blocks of directory entries, indirect data blocks, or doubly indirect data blocks. Doubly indirect blocks contain block numbers that point to indirect blocks and indirect blocks contain block numbers that point to either file data blocks or directory entry blocks. An indirect block will not contain block numbers that point to another indirect block. Figure 4 and Figure 5 go over this in more detail. A list of the possible data blocks are as follows:

- File Data Block
- Directory Entries Block
- Indirect Block
- Doubly Indirect Block

3.2 Layer Architecture

This filesystem is designed with a layered architecture to provide a structured and modular breakdown of functionality, making it easier for you to understand the inner workings of the system. `WTFS` (and every other filesystem in NK) sits between two interfaces: at the top, the Nautilus File System Abstraction Layer provides a unified interface that filesystem implementations must conform to, ensuring compatibility with the broader Nautilus environment. At the bottom, the concrete filesystem implementation interacts with the Nautilus Block Device Abstraction Layer, using basic read and write operations to access disk blocks. Each intermediate layer serves a distinct purpose, handling paths, directories, files, inodes, and disk operations, allowing you to focus on individual components without being overwhelmed by the entire system at once. Our hope is that this design will enable incremental implementation and testing, which should help when debugging. Below is a visual of the system layers.

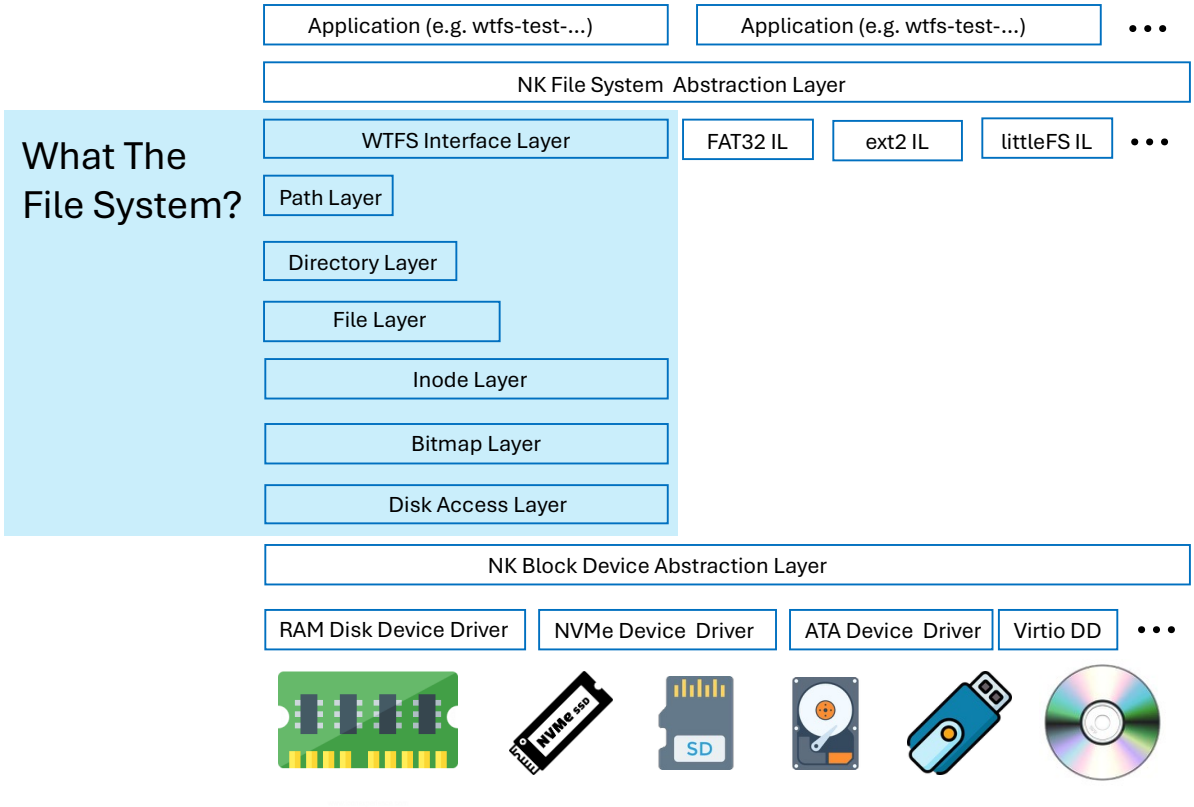

Figure 2: `WTFS` Layered Architecture

To give some extra context, a concrete file system implementation for NK is expected to adhere to the `nk_fs_int` interface, which provides essential UNIX-based file system operations such as creating and deleting files and directories, checking file existence, reading and writing file contents, truncating files, and listing directory contents. The Nautilus file system (Nautilus FS) abstracts these operations within a modular structure that allows different file systems to be registered, mounted, and accessed in a unified manner. If you find this layering interesting, take a look at the ext2 or fat32 implementations in Nautilus

for additional file systems. In the paging lab, you'll also see the LFS filesystem. All these, and WTFS, all adhere to the same interface.

Similarly, WTFS is layered on top of NK's block device interface, which abstracts the interface to device drivers for devices such as hard drives and SSDs. In this lab, you will being using this layer to access RAM disks (ephemeral storage in main memory), and persistent NVMe SSDs. In this lab, we will call anything accessed via the block device interface a "disk".

For the interested, the abstract interfaces noted above are defined in `include/nautilus/{fs.h,blockdev.h}`. Filesystem implementations live in `src/fs`, and device drivers live in `src/dev`.

3.3 Inodes

In `WTFS`, inodes can exist in many different forms. The `i_mode` field encodes multiple attributes of an inode such as its type, mode, permissions, and more. The only attributes you will have to worry about in this lab are an Inode's **Type** and **Mode**. You can find out more about the available `i_mode` flags in the `wtfs_define.h` file.

Something you should be aware of is that the inode numbering is **1-indexed**, while all other indexing in `WTFS` starts from 0. For example, if you wanted to get the first/root inode, you would pass in the inode number `1` as an argument to a function. The reason behind this is for error handling. When dealing with corrupted disk images, it is common to encounter zeroed-out sections of the disk. If an inode number of 0 is read in, it is a clear indication that something has gone wrong!

3.3.1 Inode Types

All inodes share the same format, but the type of the inode dictates the contents of the inode's data blocks.

File Type File type inodes contain disk block numbers of data blocks containing file data, which could be anything – it's just bytes of data that belong in that file.

Directory Type Directory type inodes contain disk block numbers of data blocks containing directory entries. A directory entry contains the name of a file or directory (up to 14 characters in length) and the corresponding inode number for that file. This makes 1 directory entry 16 bytes in size, so 32 of them can fit in 1 data block. Any attempt to create a file with a name longer than 14 characters will result in an error.

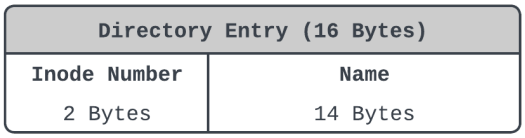

Figure 3: Directory Entry Structure

3.3.2 Inode Modes

Small mode In small mode, an inode has space to hold 8 disk block numbers (16 bits each), referencing up to 8 data blocks that hold the contents of the file. However, if the file grows beyond 8 blocks (4096 Bytes), this is no longer enough and we will need to shift the inode into large mode.

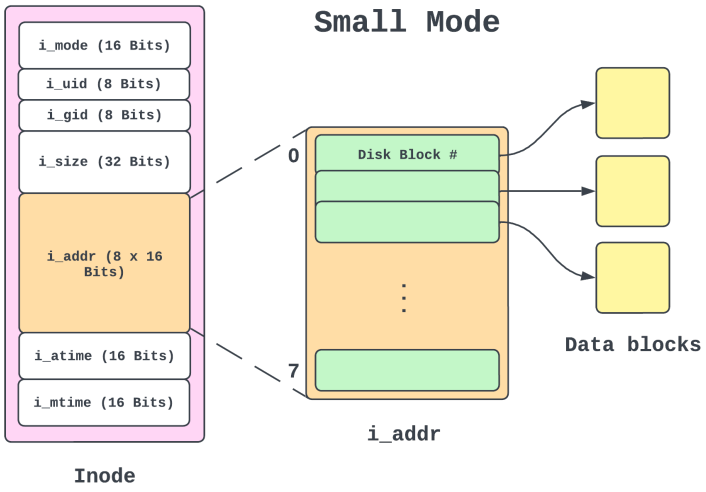

Figure 4: Inode Small Mode

Large mode In large mode, an inode still has the same amount of space to hold 8 disk block numbers, however, the first 7 disk block numbers in the inode reference indirect blocks and the 8th references a doubly indirect block. An indirect block is still a data block, but rather than holding the contents of the file directly, it holds an additional 256 disk block numbers that reference data blocks containing the file data. A doubly indirect block follows a similar principle, and holds 256 indirect block numbers. This is a significant increase in the amount of data the inode can reference, to the point that it becomes impossible to reach the maximum size of a file in large mode in `WTFS` due to the size limitations from other factors such as the size of the bitmaps.

Note: Both File and Directory type inodes can be in either small or large mode. Regardless of mode, any unused slots for data block numbers are zeroed out.

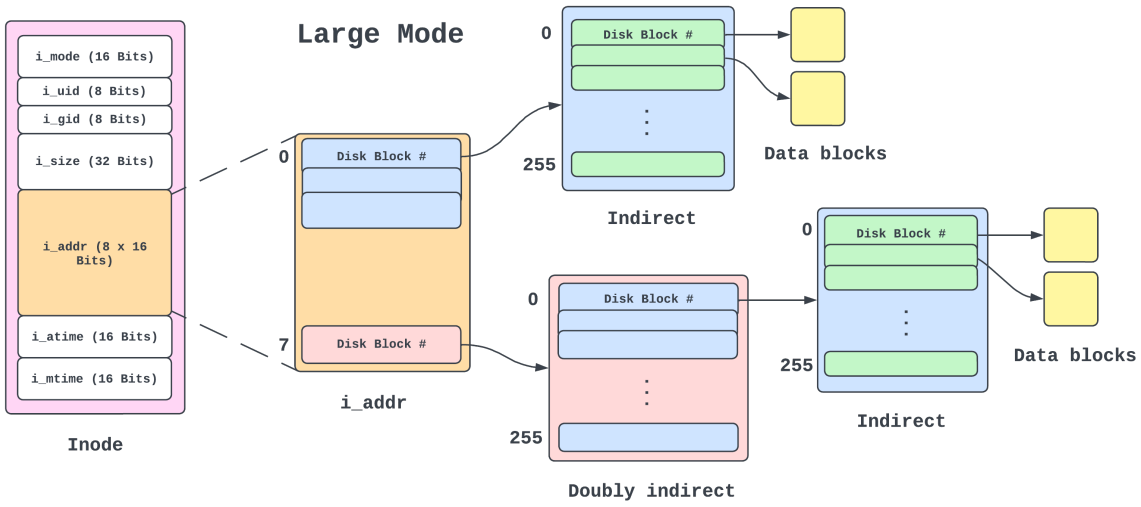

Figure 5: Inode Large Mode

Why does small mode exist? This is an example of optimizing for workload and device characteristics. Many files and directories are small. Small mode allows accessing them with fewer block reads/writes. This logic is then repeated in large mode, where the first 256 blocks of a file or directory are singly indirect, and double indirection only appears beyond this. While the "big-O" of the access does not change when using small mode or single indirection, in practice block reads/writes are very expensive ("big constant") and this optimization in the design makes the filesystem operate considerably more quickly for the user.

4 Important Files

Nautilus contains hundreds of thousands of lines of code across more than a thousand files, but thankfully for this assignment we'll only have to worry about a relatively small set of these. Below are the important files for this lab, with any that you'll have to make changes to highlighted in red. If this looks like a lot of files to you – don't panic! You'll usually only be responsible for implementing one function in each of those files.

The files for this lab are all located within `src/fs/wtfs/` in Nautilus.

- `wtfs_define.h` This file contains all of the struct definitions and global constants that define `WTFS`. **You will not change this**, but you should definitely read through it and understand what's in here as you'll be using these things frequently.
- `wtfs_access.[hc]` This contains the functions that connect our filesystem to Nautilus's block device interface, allowing us to read and write blocks on disk. **You will not change this.**
- `wtfs_bitmap_layer.[hc]` This implements the bitmap layer of `WTFS` and contains functions for allocating/freeing data blocks and inodes using their respective bitmaps. It includes the starter code for `reserve_bit_in_bitmap()`. **You will add to this.**

- `wtfs_inode_layer.[hc]` This implements the inode layer of `WTFS` and contains functions for reading inodes from disk, modifying them in memory, and writing them back to disk. It includes the starter code for `add_block_to_inode()`. **You will add to this.**
- `wtfs_file_layer.[hc]` This implements the file layer of `WTFS` and contains functions for reading/writing data blocks of files and changing the size of files. It includes the starter code for `truncate_file()`. **You will add to this.**
- `wtfs_directory_layer.[hc]` This implements the directory layer of `WTFS` and contains functions for reading, adding, and removing directory entries. It includes the starter code for `get_remove_dirent()`. **You will add to this.**
- `wtfs_path_layer.[hc]` This implements the path layer of `WTFS` and contains functions for accessing inodes based on a given path. It also includes the implementation of a few interface functions. **You will not change this.**
- `wtfs_checkdisk.[hc]` This includes the starter code for your own implementation of the checkdisk utility as well as several helper functions for your implementation. **You will add to this.**
- `wtfs_bind.[hc]` This file provides wrapper functions for `WTFS`'s implementations of the Nautilus FS interface functions, and exports those functions as the interface to which the Nautilus kernel has access. It also implements functions for attaching/detaching a device formatted with `WTFS`. **You will not change this**.
- `wtfs_util.[hc]` This includes utility functions for testing and helpful printing functions for debugging – for example, you could call `print_inodes()` to print all valid inodes on disk in a readable format. You do not have to change this, but you will definitely want to use functions from it!
- `wtfs_integration_tests.[hc]` This file includes the integration tests that verify the successful implementation of the filesystem by calling the interface functions. Feel free to add additional tests as needed.
- `wtfs_unit_tests.[hc]` This file includes the unit tests for each function you will implement. These tests are meant to be layer independent and aid you in testing each function as you go. Feel free to add additional tests as needed.
- `wtfs_shell_cmds.c` This file handles registering the various commands you're able to run in the Nautilus shell, including the debug utilities, layer-independent unit tests as well as the integration tests. You do not have to change this, but feel free to add to it if you have additional utilities you want to register for use in the shell. **It is highly recommended you read through this file**.

5 Testing

You can test your implementation in the Nautilus shell using commands that we have set up for you (see Section 6). Each task described in the following sections explains which unit tests are expected to pass with a working implementation of that layer. Once you have completed all of the tasks and have all unit tests passing, you should test your implementation with the provided integration tests (which are much more fine-grained and are likely to catch any remaining bugs).

To do so, first boot the Nautilus shell by running `$./run` from the root directory of the assignment (this runs a script that compiles and initializes the operating system, so you do not need to run `$make` separately). In the blue QEMU window that appears, run `>wtfs-test-integration` to run the integration tests. Omit shell prompts (`$`, `>`) when typing your commands.

Don't be alarmed by error messages if you are surviving the unit/integration tests! The tests are designed to check not only valid cases but also impossible or invalid operations. Some tests intentionally trigger errors to ensure that the system handles them correctly, so seeing errors can actually mean things are working as expected.

In both integration and unit tests, you will notice that many utility functions such as `check_inode()` have a `print` parameter. You can toggle on printing to help you debug by editing the test code to pass in a `1` as an argument.

Note that while normal filesystems write to a persistent disk, the WTFS implementation used for this lab actually writes to non-persistent RAM disks stored in memory. This means each time you reboot Nautilus you get fresh disks without any modifications (or any mistakes!) from prior runs of Nautilus. Once you can pass all the tests, you will go on to use your WTFS implementation on a persistant disk where the user's data and your filesystem's data structures must be able to survive a reboot (Section 12).

6 Task 0: Understanding the Implementation Details

In this layer-based system, you must start from the bottom layer[3] and provide a correct implementation of each layer as you move up the stack. Since the layers above are dependent on the layers below, any missing implementations or bugs early on will propagate up the layers and make it extremely challenging for you to debug. This means that you should think about the edge cases in each layer thoroughly. When working through each layer, we highly recommend sketching out diagrams and thinking about a possible solution before coding it up.

Macros

Macros in C are a tool to instruct the compiler to replace one piece of code with another during the pre-compilation phase. They are similar to functions in that they allow you to reuse code, but unlike functions, macros perform text substitution directly, meaning that the code is expanded inline wherever the macro is used. This can make code cleaner and more readable, especially when the same code needs to be used in multiple places. There are some other benefits of using macros such as performance optimizations during compile time and conditional compilation, but we won't get into them here.

In this assignment, you can think of them similar to a wrapper functions that simply calls another function and passes some arguments to it. This is one of several examples of how macros are used in `WTFS`:

```
int read_write_inode(struct wtfs_state *fs, int inumber, struct inode *inode, bool write)

#define read_inode(fs, inumber, inode) read_write_inode(fs, inumber, inode, 0)
#define write_inode(fs, inumber, inode) read_write_inode(fs, inumber, inode, 1)
```

In this case, you can see that the macros are defined with the `#define` preprocessor directive. The reason why we use them here is because, logically, reading and writing inodes use a lot of the same code, so

[3] Just above the block layer, though you are welcome to observe the operations of the layers below—we have turned on their debugging output.

it makes sense to have one function (`read_write_inode()`) that handles both operations with a boolean parameter to indicate which to perform. However, from a programmer's perspective, it's cleaner to have a separate function for each operation. So, to differentiate the read and write operations without having to set the write parameter manually (and figuring out what the value of the flag should be in the first place), you can simply call either the `read_inode()` or `write_inode()` macro with the parameters shown above. In this case, you no longer have to pass in the write flag parameter. One thing to be aware of is that the types should match those in the `read_write_inode()` function header.

Bitmaps

In a filesystem implementation, bitmaps are used to efficiently track resource allocation, such as free and occupied blocks or inodes. Each bit in a bitmap corresponds to a specific resource, where a value of **0** typically indicates availability, and **1** signifies usage. You will find that bitmaps are frequently updated as files and directories are created, modified, or deleted. Unlike the heap allocation setting (i.e. as `malloc`), here, all allocations are of the same size, either an inode or a block. This greatly simplifies the allocation problem.

Logical vs Physical Blocks

In a filesystem, inodes store metadata about files, including references to the data blocks where file contents are stored. A file can span multiple data blocks, which are not necessarily stored contiguously on disk. Instead, they can be scattered across different physical locations. To track a file's structure, we use logical block numbers and physical block numbers. The **logical block number** represents a data block's position within a file. The corresponding **physical block number** represents the number of that block within the disk.

For example, in a 2048-byte file with 512-byte blocks, the file consists of four logical blocks (0 to 3). If we need to read the second block of the file, we refer to logical block 1. However, this logical position must be mapped to a physical block number, which represents the actual location of the block on disk (which is arbitrary and may not be contiguous with other blocks in the file). When reading a file, we translate logical block numbers to their corresponding physical block numbers and pass them to `read_block` or `write_block` to retrieve or alter the data.

Helper Functions

The purpose of this lab is not for you to implement an entire filesystem on your own (although that could be a fun project). You can think of this as jumping into an unfamiliar codebase and iterating on the work of previous developers. Sprinkled throughout the code are helper functions that have been pre-written to help speed up your development. We realize that they might be hard to find at times, so we have included a toolbox for each task that lists the relevant helpers that you might want to use. You may use the helpers provided at your discretion. If you would like to develop a deeper understanding of the material, you can feel free to write your own, but be warned that debugging might become quite difficult.

Error Checking

To help you with debugging, we recommend frequent error checking following function calls. In C programming, error handling differs from exception-based approaches in other languages. A common paradigm is to check for error codes, such as **-1** or **0**, to detect failures when calling functions. Most WTFS functions follow this convention, returning -1 when an operation fails and 0 on success. When debugging, these return values provide useful checkpoints for identifying issues. Adding debug and print statements at

these failure points can help you trace errors and understand the flow of execution in your program. A common way to approach this is by wrapping function calls in an if statement and checking if the functions return an error code. Examples of this can be seen throughout the `WTFS` code base.

Debugging Utilities

In writing this lab, we have encountered many nightmarish bugs and, therefore, have written many printing utilities to help us debug. Normally, it would be up to you to write your own debugging utilities, however, we thought it would be nice to share them since this is likely a challenging lab. There are a number of functions available in `wtfs_util.h` and `wtfs_util.c` for debugging. Most of these functions have been wrapped in shell command functions and registered as shell commands in `wtfs_shell_cmds.c`, meaning they can be run directly in the NK terminal. Some of these debug commands can take in a number of different arguments. It will be up to you to figure out how to use them if you so desire. Of course, we also encourage you to write your own if you'd like additional ones. The following functions are the print utilities provided to you for you to call within your own code when debugging:

- `void print_inodes(struct wtfs_state *fs)`
  `//Prints all allocated inodes in the filesystem`
- `void print_inode(struct wtfs_state *fs, int inode_num)`
  `// Prints a specific inode`
- `void print_bitmap(struct wtfs_state *fs, int bitmap_block_num)`
  `// Prints a bitmap in a readable binary format`
- `void print_file(struct wtfs_state *fs, char *path, int n)`
  `// Prints the first n bytes of a file`
- `void print_directory_block(struct wtfs_state *fs, int block_num)`
  `// Prints a data block's contents formatted as directory entries`
- `void print_indirect_block(struct wtfs_state *fs, int block_num)`
  `// Prints a data block's contents formatted as an indirect block`
- `void print_directory_structure(struct wtfs_state *fs)`
  `// Prints a visual of the filesystem hierarchy (all directories and files)`

Shell Commands

We have provided a number of commands you can run within Nautilus to test and understand the filesystem. Each is prefaced with `wtfs-` to signify that it's part of this lab. You can see a full list of shell commands by running `>help` in Nautilus.

`>wtfs-load-disk-image image_name`	Load WTFS disk image by name
`>wtfs-print-bitmaps`	Print WTFS bitmaps
`>wtfs-print-directory-block block_num`	Print given WTFS indirect block
`>wtfs-print-directory-structure`	Print WTFS directory structure
`>wtfs-print-file path num_bytes`	Print bytes from WTFS file at path
`>wtfs-print-indirect-block block_num`	Print given WTFS indirect block
`>wtfs-print-inodes`	Print WTFS inodes
`>wtfs-run-checkdisk`	Run WTFS Checkdisk on loaded image
`>wtfs-test-checkdisk`	Run WTFS Checkdisk Tests
`>wtfs-test-integration`	Run WTFS Integration Tests
`>wtfs-test-unit-bitmap`	WTFS Bitmap Layer Unit Test
`>wtfs-test-unit-directory`	WTFS Directory Layer Unit Test
`>wtfs-test-unit-file`	WTFS File Layer Unit Test
`>wtfs-test-unit-inode`	WTFS Inode Layer Unit Test

7 Task 1: Implement CheckDisk

A CheckDisk utility scans a filesystem for errors, inconsistencies, and corruption, verifying the integrity of metadata structures like inodes, directories, and bitmaps. It detects issues such as lost data blocks, orphaned inodes, and bitmap corruption.[4]

Our version of CheckDisk accomplishes this by making a blank copy of each bitmap (inode and data block), and traversing the filesystem to mark all reachable inodes and data blocks as allocated. It then compares these bitmaps to the actual bitmaps on disk, checking for discrepancies. If the bitmaps match, then the disk format is valid. Otherwise, data blocks and/or inodes have been incorrectly allocated or freed, indicating a problem in the system modifying the disk.

You will implement two functions as helpers for the checkdisk test, both of which are heavily commented and can be found in `wtfs_checkdisk.c`.

`mark_blocks_in_file()` This function will iterate through all the data block numbers in a given inode, marking each one as allocated in a copy of the data block bitmap. Be careful - remember that all inodes (both file and directory types) can be in either small or large mode, meaning there might be indirect and even a doubly indirect block in use. This may affect how you choose to iterate through the contents of the inode to find all of the data block numbers it contains. Additionally, recall that indirect and doubly indirect blocks are data blocks.

`traverse_inode_contents()` This function should perform a recursive depth first search on the filesystem tree, marking each inode it encounters as allocated in a copy of the inode bitmap. It should also call `mark_blocks_in_file()` for each inode in order to account for all the data blocks in use. A diagram to help you visualize what the filesystem tree might look like can be found below in Figure 6.

Toolbox This is a toolbox of helper functions, macros, and content you should familiarize yourself when working on this task.

[4]This process is sometimes called "sanity checking"—is the data structure in a legitimate state? On Unix-style systems, the `fsck` utility is usually used to do sanity checking (and repair), while on Windows, the `chkdsk` utility serves the same purpose. Having a good sanity checker is very helpful while building a complex piece of code. You can run it before and after each operation you implement and thus quickly pinpoint where things go wrong, even if it doesn't cause a crash.

Functions

- `mark_spot_in_bitmap()` in `wtfs_checkdisk.c`
- `get_physical_block_num()` in `wtfs_inode_layer.[hc]`
- `read_file_block()` in `wtfs_file_layer.h`

Macros

- `BLOCK_SIZE` in `wtfs_util.h`
- `read_block()` in `access.h`
- `read_inode()` in `wtfs_inode_layer.h`

Content

- `dirent` struct in `wtfs_define.h`
- `inode` struct in `wtfs_define.h`
- Filesystem tree (Figure 6)

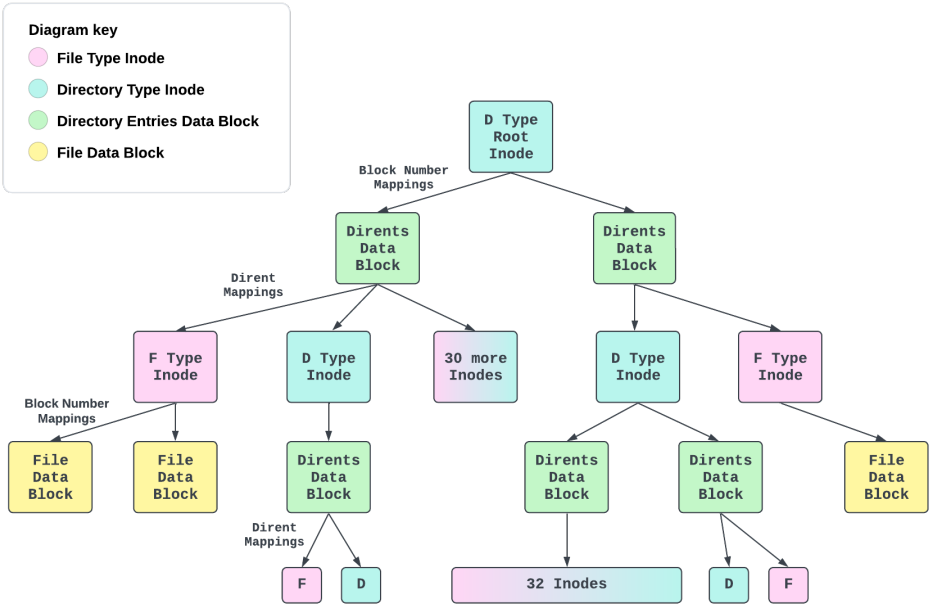

Figure 6: A **Simple** Example of the `WTFS` Tree Structure

Testing We have provided many disk images on which you can test your CheckDisk implementation. In `configs/boot-extra/wtfs/checkdisk_images` you will find the disk image files and a text file `README.md`, which contains details on the filesystem contained within each disk image. These disk images have two categories: *valid* disk images should pass when CheckDisk is run on them, while *corrupt* disk images are expected to fail. So errors being printed in terminal are not necessarily concerning as some of them are expected.

To run your CheckDisk on all disk images, run `>wtfs-test-checkdisk` in the Nautilus shell and check that all tests have correct results.

You can also run CheckDisk on an individual disk image by first loading the specified disk image. For example, to run CheckDisk on the first test image, run `>wtfs-load-disk-image checkdisk_valid1` and then `>wtfs-run-checkdisk` in the NK shell.

8 Task 2: Bitmap Layer

The bitmap layer of `WTFS` is responsible for handling all interactions with the **inode** and **datablock** bitmaps, including both allocating/freeing inodes and data blocks. In this task, you will implement a helper function used to find an available spot in a bitmap and allocate it.

`reserve_bit_in_bitmap()` In this function, you will read a given bitmap from disk and iterate through each bit from the beginning until a 0 is found. That bit should then be set to 1 to indicate that the inode/data block has been allocated, and the index of the newly allocated bit should be returned. Be sure to also write the modified bitmap back to disk. This function can be found in `wtfs_bitmap_layer.c`.

Toolbox This is a toolbox of helper functions, macros, and concepts with which you should familiarize yourself when working on this task.

Functions

- `get_bit_from_byte()` in `wtfs_bitmap_layer.c`
- `set_bit_in_byte()` in `wtfs_bitmap_layer.c`

Macros

- `read_block()` in `wtfs_access.[hc]`
- `write_block()` in `wtfs_access.[hc]`

Content

- Bitmap blocks
- `wtfs_state` struct in `wtfs_define.h`
- `s_block_size` field of `superblock` struct

Testing Run `>wtfs-test-unit-bitmap` in the Nautilus shell.

9 Task 3: Inode Layer

Inodes are responsible for storing metadata for files, including data block numbers, file type and permissions flags, the size of the file, and access and modification timestamps. In `WTFS`, each file (recall, this includes directories too) has one corresponding inode that manages the disk block numbers for each of the file's data blocks. The inode has space to store 8 data block numbers within an array, but if the file grows beyond this size it becomes necessary to use indirection (indirect/doubly indirect blocks) to store the data block numbers in a compact manner. Each time a new data block is added to a file, the inode is responsible for managing any resulting mode changes (between small and large) and indirection. In this task, you will implement a function that handles adding a new data block to a file.

`add_block_to_inode()` In this function, you are given a pointer to a file's inode and a data block number to add to that file.

Upon the addition of a new block, you will need to determine whether any mode changes (small/large) are necessary and allocate any indirect and/or doubly indirect blocks as necessary based on the current file size. Once you have handled this and stored the new block number, you should increment the size of the file according to the `num_valid_bytes` parameter.

This is the most complex function we are asking you to implement, so we have given you a lot of structure and comments explaining what to do in each case in the starter code. We also recommend working on one case at a time and testing as you go rather than attempting to write the entire function at once and hoping it works on the first try—it is a long function, which means there will be a significant amount of code to sift through looking for mistakes if something goes wrong.

The code is split into seven total cases based on what state the file is initially in, and what state it should be in after the block is added:

- Case 1: File is currently in small mode.
  - Case 1a: File will be in large mode, using one singly-indirect block.
  - Case 1b: Block fits in inode, still in small mode.
- Case 2: File is currently in large mode, but doesn't use the doubly-indirect block.
  - Case 2a: File will use the doubly-indirect block.
  - Case 2b: File will use an additional singly-indirect block.
  - Case 2c: Block fits in current singly-indirect block.
- Case 3: File is current in large mode, and does use the doubly-indirect block.
  - Case 3a: File will use a new singly-indirect block added to the doubly-indirect block.
  - Case 3b: Block fits in the current singly-indirect block.

Toolbox This is a toolbox of helper functions, macros, and concepts with which you should familiarize yourself when working on this task.

Functions

- `create_indirect_block()` **This helper can save you from a lot of duplicate code**, so we strongly recommend using it. Make sure you read the comments to understand how it works.

Macros

- `read_block()` in `wtfs_access.[hc]`
- `write_block()` in `wtfs_access.[hc]`

Content

- `inode` struct in `wtfs_define.h`
- Inode modes (small vs. large)
- Indirect/doubly indirect blocks: implemented as arrays of 256 `uint16_t` block numbers

Testing Run `>wtfs-test-unit-inode` in the Nautilus shell. Each unit test corresponds to one of the cases as described above. If you fail a test, look at the print outs in terminal and look through the code in the unit test to figure out what went wrong.

10 Task 4: File Layer

The file layer of `WTFS` provides operations for interacting with files in the system, including reading/writing individual blocks within a file, reading/writing bytes to a file, and changing the size of a file. This is where we can really start to see the benefits of the layered structure of `WTFS`, because the file layer is able to make use of the abstraction of inodes provided by the inode layer below it. More specifically, the work we've done in the inode layer means that in the file layer we no longer have to concern ourselves with what mode an inode is in or how to translate a logical block number to a physical one for a given file. In this task, you will implement a function that handles an essential filesystem operation: truncation, the shrinkage or expansion of a file.

`truncate_file()` For this function, you are given an inode for a file and a desired length of that file. You will first need to determine whether to shrink or expand the file according to the current size of the file and the target size.

If you are expanding the file, you may need to allocate new data blocks. They should each be added to the inode using the helper you wrote in Task 3.

If you are shrinking the file, you may need to free data blocks starting from the end of the file. You should then zero out any newly invalid bytes in the last block of the file. Make sure you write any changes you make to data blocks back to disk. This is a reasonably complex function (although not as bad as `add_block_to_inode`), so we have again provided detailed comments in the starter code to help guide your implementation.

Toolbox This is a toolbox of helper functions, macros, and concepts with which you should familiarize yourself when working on this task.

Functions

- `add_block_to_inode()` in `wtfs_inode_layer.[hc]` (Hey! This one should be familiar already!)
- `remove_block_from_inode()` in `wtfs_inode_layer.[hc]`
- `get_physical_block_num()` in `wtfs_inode_layer.[hc]`

- `inode_set_size()` in `wtfs_inode_layer.[hc]`
- `read_file_block()` in `wtfs_file_layer.[hc]`
- `write_file_block()` in `wtfs_file_layer.[hc]`

Macros

- `alloc_block()` in `wtfs_bitmap_layer.h`
- `free_block()` in `wtfs_bitmap_layer.h`
- `write_inode()` in `wtfs_inode_layer.h`

Testing Run `>wtfs-test-unit-file` in the Nautilus shell.

Note that these tests require a fully working version of `add_block_to_inode()` in the inode layer.

11 Task 5: Directory Layer

In `WTFS` and similar UNIX-based filesystems, directories serve as a structured way to organize files and other directories. Rather than storing file metadata directly, directories contain directory entries (dirents), which map file names to inode numbers–unique identifiers for files and directories. The inode itself stores critical metadata, such as file permissions, ownership, timestamps, and disk block locations, but it does not contain the file name. When a user accesses a file by name, the filesystem traverses directory entries to find the corresponding inode and retrieve the file's metadata and contents. This approach promotes space efficiency for naming, supports hard links, and maintains a clean separation of concerns. In this final layer implementation task, you will write an essential function that handles the retrieval or removal of a directory entry from a directory file.

`get_remove_dirent()` For this function, you are given a filename, an inode number corresponding to a directory's inode, and a pointer to a pre-allocated directory entry struct. Note that this function is called by two macros, which each pass in a different `is_get` flag and determine if the function is either getting or removing a dirent. In your implementation, you will apply a simple linear search algorithm to find the target dirent and either get or remove it from the directory.

For the get functionality, once you find the matching directory entry, you should copy its contents into the memory location pointed to by `dir_entry`. This allows the caller to access the retrieved directory entry after the function returns.

For the remove functionality, once you find the matching directory entry, you should overwrite it with the last directory entry in the directory. This ensures that directory entries remain contiguous, preventing gaps. Remember to propagate your changes back to disk. Finally, truncate (shrink) the directory file to remove the duplicate last entry, ensuring the directory's size reflects the removal.

If no matching entry is found, return a failure code.

Toolbox This is a toolbox of helper functions, macros, and concepts with which you should familiarize yourself when working on this task.

Functions

- `inode_get_size()` in `wtfs_inode_layer.[hc]`
- `read_file_block()` in `wtfs_file_layer.[hc]`
- `write_file_block()` in `wtfs_file_layer.[hc]`
- `truncate_file()` in `wtfs_file_layer.[hc]`
- `get_last_dirent()` in `wtfs_directory_layer.c`
- `strncmp()` (C library function)

Macros

- `read_inode()` in `wtfs_inode_layer.h`

Content

- `dirent` struct in `wtfs_define.h`
- `s_block_size` field of `superblock` struct

Testing Run `>wtfs-test-unit-directory` in the Nautilus shell.

Note that these tests require at least a partially working version of `truncate_file()` in the file layer.

12 Task 6: Persistence and Observing Device Interaction

So far, all the tests you have done have used RAM disks. In this part of the lab, you will see how well your filesystem operates across reboots, and use it to copy data to/from the host (Moore).

For the following tasks, when you run NK, you will generally want to capture all of the output that NK is printing to the serial port - this is what appears in the terminal window where you issue the `run.sh` command. You can capture output via redirection. The following will capture output in the file `serial.out`:

```
server> ./run > serial.out
```

Note that this will *overwrite* the `serial.out` file!

We have provided several tools in `~cs343/HANDOUT/` for use in this task:

- `mkwtfs` creates a WTFS filesystem in an image file on the host (Moore).
- `cptowtfs` copies a host file into a WTFS image
- `cpfromwtfs` copies a file from a WTFS image to a host file.

Additionally, the following NK commands are available:

- `attach` mounts a filesystem located in a block device into the namespace.
- `fses` lists the attached filesystems.
- `ls` lists the contents of a directory.

- `cat` prints out the contents of a file.
- `dog` creates and fills a file with data.

If you have disabled debugging output in NK, turn it back on now. At minimum you should have debugging output for the filesystem abstraction layer, your filesystem, the block device abstraction layer, and the NVMe block device driver.

If you would like to see how our solution operates, you can use the `~cs343/HANDOUT/run-fslab-solution-{debug,nodebug}` commands instead of the `./run` command.

T6a: Create persistent image, copy host to guest

Starting in the top-level directory of the lab, create a directory that will contain files and directories to be placed on a persistent image, and then create at least one file within it:

```
server> mkdir stuff
server> echo "Hello world!" > stuff/myfile.txt
```

Create an NVMe WTFS disk image named `nvmsmall.img` that is based on your directory:

```
server> ~cs343/HANDOUT/mkwtfs stuff nvmsmall.img
```

You can copy additional files to the image like this:

```
server> ~cs343/HANDOUT/cptowtfs README.md nvmsmall.img /readme.md
```

Now run NK as usual with `./run > serial-T6a.out`. The script will make `nvmsmall.img` appear as an NVMe drive attached to an NVMe controller. In NK, the corresponding block device is `nvme-0`. Reads and writes to `nvme-0` in the guest (NK) will ultimately turn into read and writes to `nvmsmall.img` in the host (Moore).

You can attach `nvme-0` with WTFS using the command

```
rootshell> attach nvme-0 wtfs stuff
```

This means that we want to access a WTFS filesystem resident on the block device `nvme-0` and give it the name `stuff`.

Now, list the contents of your volume and print out one of the files you created on it:

```
rootshell> ls stuff:/
rootshell> cat stuff:/myfile.txt
rootshell> cat stuff:/readme.md
```

You can now shutdown NK for this task. Keep the `serial-T6a.out` file., add it to your repo, and push it.

The `serial-T6a.out` file (and the shell for Moore) contains all your commands and their output, in addition to the debugging output. Put a copy of the commands and the responses in your `STATUS` file.

T6b: Create files on NK, copy guest to host

Start with the `nvmsmall.img` you have from T6a. Boot it using `./run > serial-T6b.out`. and attach the volume as before:

```
rootshell> attach nvme-0 wtfs stuff
```

Now, create a file on on the volume:

```
rootshell> dog stuff:/test.txt 893
```

This will create the file `test.txt` and fill it with 893 bytes of data consisting of letters and digits. You should be able to print it outL:

```
rootshell> cat stuff:/test.txt
```

You can now shut down NK.

Now, on Moore, copy your file out of the `nvmsmall.img` image file:

```
server> ~cs343/HANDOUT/cpfromwtfs nvmsmall.img /test.txt test.txt
```

Verify that `test.txt` contains 893 bytes of data consisting of letters and digits, the same as you saw from within NK.

Keep the `serial-T6b.out` file., add it to your repo, and push it.

The `serial-T6b.out` file (and the shell for Moore) contains all your commands and their output, in addition to the debugging output. Copy the relevant commands and their output to your `STATUS` file

Reflection Note that the filesystem in `nvmsmall.img` has now survived two boots of NK—it is persistent. It should be able to survive indefinitely.

T6c: Analyze T6a and T6b debugging output

Using the `serial-T6a.out`, using `gdb`, trace what happens from an invocation of an `nk_fs_read()` from the NVMe in T6a. Document the behavior across the filesystem abstraction layer, your filesystem, the block device abstraction layer, and the NVMe block device driver, both in terms of how data flows down through the layers, and how the completion of an NVMe read (signaled by polling or interrupts) flows up through the layers. Add this documentation to your `STATUS` file.

Using the `serial-T6b.out`, using `gdb`, trace what happens from the invocation of the `nk_fs_open()` to create a new file on the NVMe in T6b. Document the behavior across the filesystem abstraction layer, your filesystem, the block device abstraction layer, and the NVMe block device driver, both in terms of how data flows down through the layers, and how the completion of an NVMe write (signaled polling or interrupts) flows up through the layers. Add this documentation to your `STATUS` file.

Be sure to commit and push your `STATUS` file.

Reflection We hope that this has helped you form a stronger intuition of how the layers interact.

13 Grading

Your group should regularly push commits to Github. You also should create a file named `STATUS` (no file extension, please) in which you regularly document (and push) what is going on, todos, what is working, etc. Your commits are visible to us, but not to anyone else outside of your group. The commits that we see up to deadline will constitute your hand-in of the code. The `STATUS` file should, at that point, clearly document that state of your lab (what works, what doesn't, etc). Task 6 output/analysis will also go here.

In addition to your `STATUS` file, you should regularly push your work within `src/fs/wtfs/*` and any other files you are changing.

You will also need to submit your files to Gradescope. To do so, in the root of the repository run:

```
$ make submit
```

That will upload your `STATUS` and `src/fs/wtfs/*` files to Gradescope. After you submit, you'll need to mark your group members on your submission. Unfortunately, you will need to do this each time you submit.

We will run your code against all the provided unit tests. Correct code will survive all tests. We will also run the integration test `>wtfs-test-integration` to ensure that the entire filesystem works correctly. Functions should properly fail when expected to and succeed when able to. The entire filesystem should always remain consistent. We will also check out your `STATUS` file carefully for Task 6.

For this lab, we will not have futher "hidden" tests. We will grade you by running the same tests you can run yourself.

The breakdown in score will be as follows:

- 20% Task 1—Completed CheckDisk with working `mark_blocks_in_file()` and `traverse_inode_contents()` functions.
- 10% Task 2—Completed Bitmap Layer `reserve_bit_in_bitmap()` function.
- 25% Task 3—Completed Inode Layer `add_block_to_inode()` function.
- 25% Task 4—Completed File Layer `truncate_file()` function.
- 10% Task 5—Completed Directory Layer `get_remove_dirent()` function.
- 10% Task 6—Completed Persistence and Observing Device Interaction `STATUS` file information.

14 Extra Credit

Please contact us first if you are interested in doing extra credit for this lab. We are at the very early stages of adding extra credit.

Here are some potential ideas:

- Enable a new inode mode with data resident in the inode itself for sufficiently small files. One possible use of this would be to create "device" (major+minor number), "named pipe", and "lock" inodes.
- Modify inodes and indirect blocks to use extents rather than individual inode pointers. An extent is run-length encoding. Suppose the physical blocks $(5, 6, 7, 8, 9, 10, 11, 12, 19, 20)$ were used. This could be encoded as two extents $((5, 8), (19, 2))$.
- Add support for arbitrary-length file names in directory entries (or at least considerably bigger than the current limit). In a typical implementation of this the file names are of variable length.
- Add support for hard links to the filesystem. Hard links allow you to have multiple names (and paths) for the same inode.

A Acknowledgements

WTFS was initially designed and implemented by Northwestern students Natalie Hill and Jason Lu in Fall 2024. They further developed it into a course assignment in Winter 2025. Many thanks to John Ousterhout and Nick Troccoli at Stanford, who developed the read-only educational filesystem that WTFS is based on and generously gave us access to it.

B What the File System? Architecture

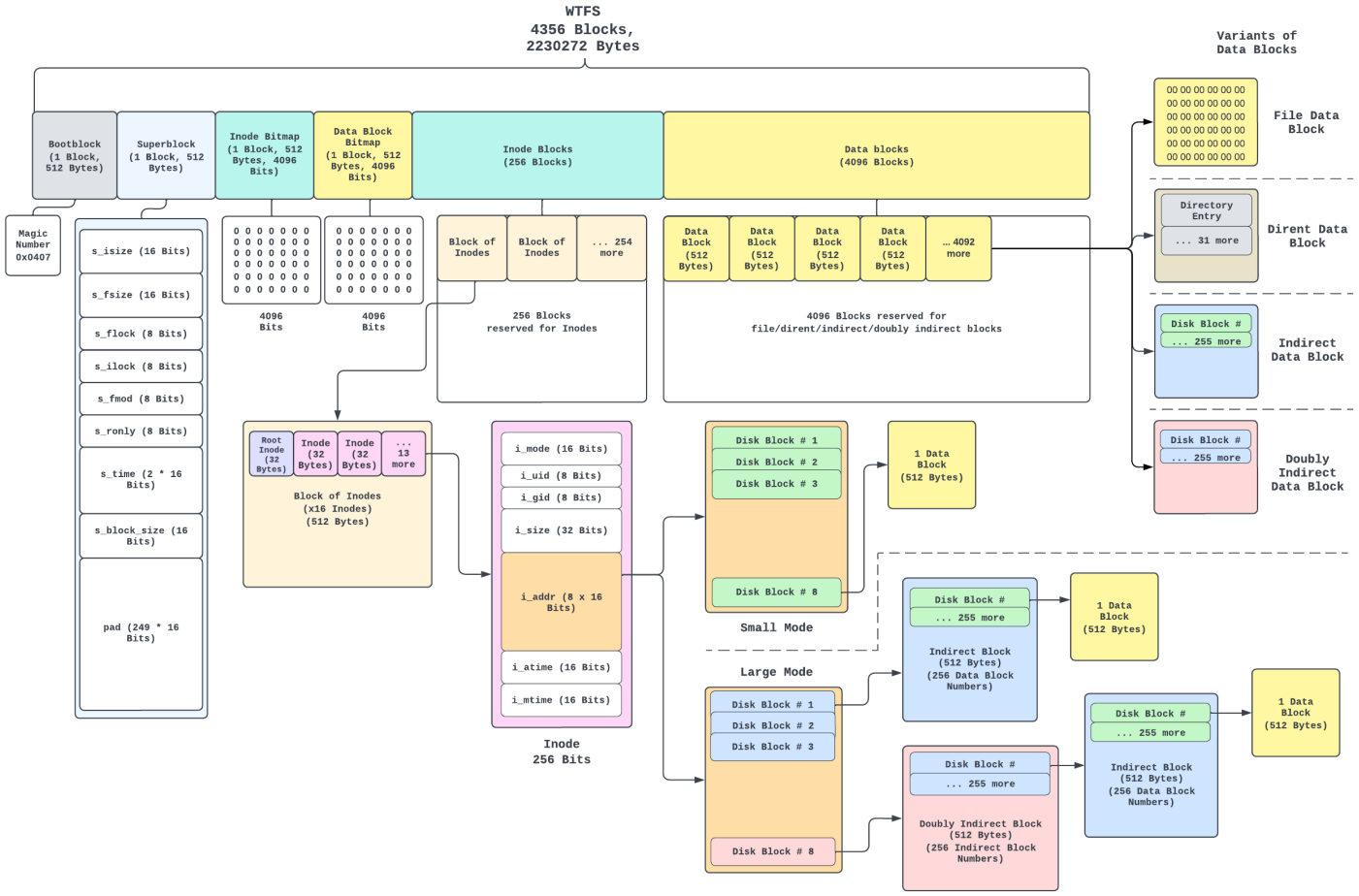

Figure 7: WTFS Architecture Overview