Final Project

32172917 Dabin Lee / 32190984 Isu Kim

December 17, 2022

Left Free Days : 5

1 Index

1 Virtual Memory Concepts (Project 2)

1.1 Virtual Memory

As we saw in the previous project, in order to run a program, the program on disk must be in a "process" state loaded into memory. That is, *the instructions currently executing must exist in physical memory.* The easiest way to satisfy this requirement is to put the entire process in memory. However, this method is not good because it limits the size of the program to the size of physical memory. In fact, in many cases, the entire program doesn't need to be in memory all at once.

So, what are the benefits of running only a portion of a program in memory? First, programs are no longer constrained by the size of physical memory. Second, each program takes up less memory, allowing more programs to run at the same time. As a result, the response time does not increase, but CPU utilization and throughput increase. Finally, programs run faster because the number of I/Os required to put and swap programs into memory is reduced.

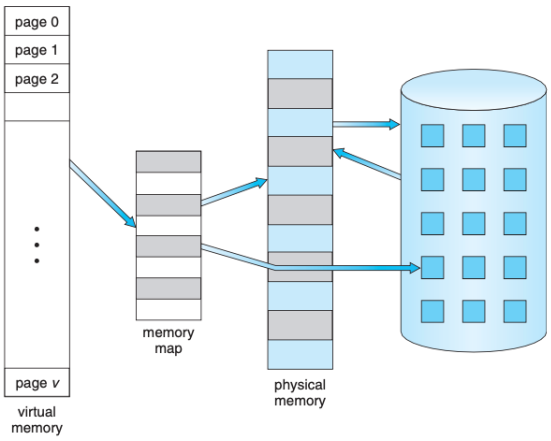

Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

Figure 1: Virtual Memory Diagram

Virtual memory is a separation between the actual physical memory concept and the developer's concept of virtual memory. The advantage of it is that a small amount of memory can provide the programmer with as much virtual address space. Virtual memory makes it easy for programmers to write programs without having to worry about memory size issues. In addition, virtual memory allows files and memory to be shared by two or more processes through page sharing.

1.2 Virtual address

The virtual address space of a process is the logical (or virtual) appearance of how a process is stored in memory. In general, it starts at a specific virtual address (usually address zero) and occupies a *contiguous space*, as shown in the figure above. The virtual address space of each process is divided. A 32-bit process can have 4 GB of virtual address space. However, users don't have access to all 4GB of virtual address space, only about 2GB. The reason is that the 32-bit address space is divided as follows:

- Null-pointer allocation partition: 0x00000000 0x0000FFFF

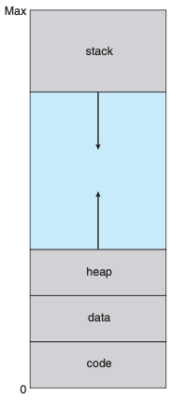

Figure 9.2 Virtual address space.

Figure 2: Virtual Address Space

- User Mode Partition: `0x00010000  0x7FFEFFFF`
- 64KB no-access partition: `0x7FFF0000 0x7FFFFFF`
- Kernel mode partition: `0x80000000 0xFFFFFFFF`

1.3 Paging

Paging is a *non-contiguous allocation* method. As a method of solving the inefficiencies of the compaction of external fragmentation, breaking physical memory into fixed-sized blocks called *frames* and breaking logical memory into blocks of the same size called *pages*. In this way, we'll be able to store a page in one frame.

The space used by a process is divided into several pages, and each page is mapped and stored in a frame in memory in any order. Since processes are not stored in memory in order, we need to know which frame the page is in in order to execute the process. This information is stored in a table called *page table*, which is used to convert logical addresses to physical addresses. Now that the concept of page tables is complete, it is time to map the physical page frame to the virtual page depends on the MMU. In other words, to access memory, it must be accessed by converting a logical address to a physical address through a memory management unit (MMU).

This paging technique has the advantages of being able to solve the problem of external fragmentation, fast allocation and deallocation, and simple swap out, but the disadvantages still do not solve internal fragmentation, additional memory for storing the page table is consumed, and since the page table resides in memory, the access operation is performed twice, which slows down memory access. Therefore, it uses a high-speed hardware cache called *TLB* (Translation Look-aside Buffer) to improve speed.

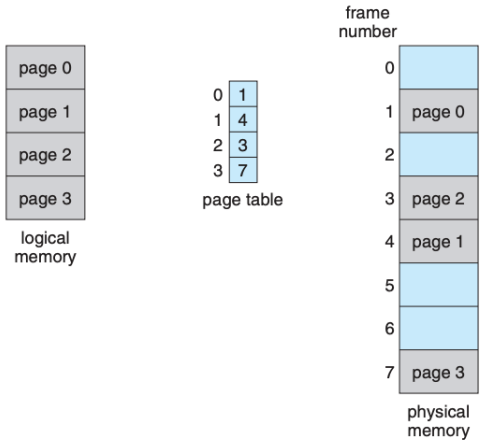

Figure 8.11 Paging model of logical and physical memory.

Figure 3: Paging Model

1.4 Demand Paging

Given the nature of paging described above, it seems that there is no need to load all pages into memory. If only the pages that are essential to run the process are loaded into memory at the right time, there will be no big problem in executing the process. This technique is called demand paging. This increases CPU utilization and throughput, and allows for more users. Demand paging uses a valid-invalid bit in the page table that indicates whether the page is in memory. If bit is invalid, the page is not in physical memory. Therefore, if all page entries are initialized to invalid, and if bit is invalid during address translation, a page fault error occurs. The specific process of address conversion is as follows:

1. The hardware checks the TLB.
2. If it is a TLB hit, immediately convert the address, and if it is a TLB miss, check the page table ($\rightarrow$3).
3. If the valid-invalid bit of the page table is valid, convert the address and add page to the TLB. If invalid, a page fault occurs ($\rightarrow$4).
4. When a page fault occurs, the MMU traps the operating system and enters kernel mode, causing the page fault handler to be invoked.
5. If the reference is invalid, terminate the process, otherwise get an empty page frame. If there is no empty frame, select a victim page from memory to replace it.
6. The operating system loads the referenced page from disk into memory, and this process deprives the CPU until disk I/O ends.
7. When disk I/O is finished, the page table is updated and the valid-invalid bit is changed to valid. Then put the process in the ready queue.
8. When the process catches the CPU, it continues again.

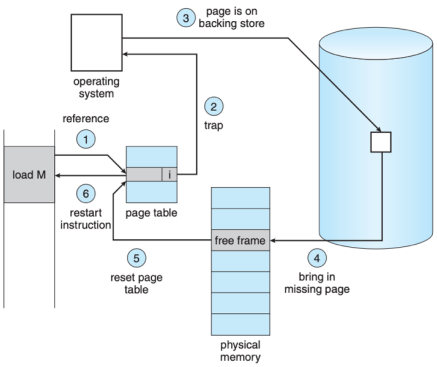

Figure 9.6 Steps in handling a page fault.

Figure 4: Demand Paging

1.5 Page Replacement Algorithm

As in step 5 of address conversion described above, what if the physical memory is full? One of the existing pages must be stored from physical memory to disk and a new page must be placed in that physical memory. At this time, the page replacement policy is about which page to be demoted from physical memory to the disk, that is, to find the victim page.

- **FIFO (First-In-First-Out)**: FIFO algorithms are a way to kick out what comes in first. You can use it even if you don't know the future. The advantage is that all pages live in the frame equally, and it is easy to implement. However, some pages may always be needed, and even in that case, there is a disadvantage that it is replaced.
- **LRU (Least-Recent-Used)**: The LRU algorithm is a way to erase the oldest references. It is the most optimized page replacement algorithm, but it is difficult to implement, and each time a process accesses main memory, it must record the time for the referenced page. Therefore, large overhead can be incurred.

2 File System Background

2.1 Project Introduction

This project implements a file system, which reads and mounts the *disk.img* file after all implementations are completed. After that, 10 file names are randomly assigned, and the output is the data of the file name according to the correct structure. In detail, the goal of this project is to print out all the information of the mounted super block and the inode table, and check whether the desired file is loaded correctly. In addition, the goal of this project is to add a write function so that the data in the file can be modified with the data desired by the user.

To implement the project, we must understand the concept, structure, features, and role of the file and the file system and think about why a file system is necessary. Also, by studying the representative types of file systems we understand the overall theory of file systems. Based on this, we can implement open, read, and write that works on the actual file system, and implements the file system based on thinking about how to create additional directories and use shared files by multiple users. Based on this, we will implement open, read, and write that run on the actual file system, and think about how to create additional directories and use files shared by multiple users. Through these efforts, let's implement a complete file system.

2.2 Concepts

2.2.1 File and File System

File is a logical storage unit that gives a name to a set of related information materials (data or programs). It provides a uniform logical perspective of information storage for convenient use of computer systems. It is generally stored in nonvolatile disk memory in record or block units.

A file system is a system that stores or organizes files stored on such disks so that they can be easily found and accessed. To use a simple analogy, a hard disk can be thought of as a library, a file system as a book search table, files as books, and data as the contents of books. So, in order to find the file containing the data you want, there is a file system inside the hard disk.

2.2.2 File Attributes and Operations

The attributes of the file vary by operating system, but are usually organized as follows:

- **Name**: Only information in human-readable form.
- **Identifier**: Unique tag that identifies the file within the file system.
- **Location**: Pointer to file location.
- **ETC**: Type, Size, Time, date, and user identification.

This information about all files is kept in the directory structure that also resides on secondary storage. A directory entry usually consists of the file's name and its unique identifier. This identifier is in turn used to find other file attributes.

A file is an abstract data type. So, to define a file we need to know its operations. File performs *create, write, read, reposition within a file, delete, truncate.* These six basic operations consist of

the minimal set of required file operations.

2.2.3 Directory and Disk Structure

Typically, thousands to billions of files are stored on random access devices such as hard disks. Since there are too many files to be saved, it is necessary to build a system for saving files. This entails the use of *directories*. The directory determines the overall structure of the file system by associating the name of each file with the file itself.

- **Volume**: storage area on disk formatted with a filesystem, disks are segmented into one or more volumes, each containing a file system or left raw(unformatted disk space).
- **Device directory**: records information such as name, location, size, and type for all files on that volume.
- **Partition**: a protocol defined so that contiguous storage space can be divided into one or more contiguous and independent areas. It's common to have multiple partitions on a single physical disk.

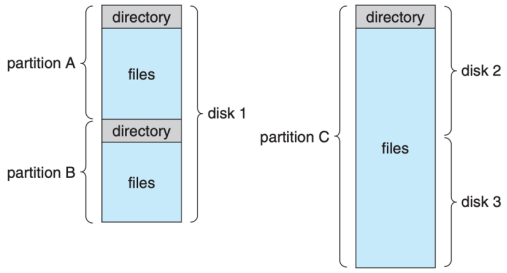

Figure 11.7 A typical file-system organization.

Figure 5: File System Organization

- **Tree-Structured Directories**: the tree structure allows users to create their own subdirectories and organize files accordingly. It has one root directory and several subdirectories. Also, every file on the system has its own path. This allows for efficient navigation and grouping. Since a directory is a type of file, it is necessary to distinguish whether it is a file or a directory. Each entry in the directory uses one bit to distinguish whether it is a file (0) or a directory file (1). And all directories have the same internal format.

2.2.4 File-System Mounting

The operation performed before the file is used, the file system must be mounted before it can be used by processes. That is, the operating system is given the name and file location of the device. Typically, a mount point is an empty directory where the file system will be attached. The operating system verifies that the device contains a valid file system. The process is done by requesting that the device driver read the device directory and verify that the directory has a valid format. The operating system records in the directory structure that the file system is attached on the specified mount point. This technique allows the operating system to traverse the directory structure and replace the file system appropriately.

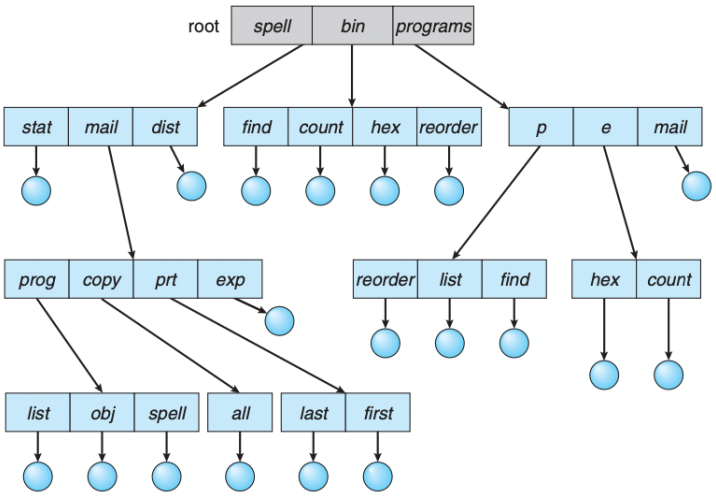

Figure 11.11 Tree-structured directory structure.

Figure 6: Tree Structured Directory

2.2.5 File-System Structure and Implementation

File system resides on disks. It is organized into the following layers: Application programs → $logical file system$ → $file-organization module$ → $basic file system$ → $I/O control$ → *devices*

To implement the file system, several on-disk and in-memory structures are used. This structure varies depending on the operating system and file system but follows some general principles. The file system that exists on the disk has the following structure: the operating system stored on the disk, the total number of blocks, the number of blocks that can be used, the directory structure, and information about the files.

- **Boot Control Block**: It contains information the system needs to boot an operating system from that volume. This block may be empty if there is no operating system on the disk. It is usually the first block of a volume. In UFS, it is called *boot blocks*. In NTFS, it is the partition *boot sector*.
- **Volume Control Block**: It contains volume (or partition) details such as number of blocks in the partition, block size, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a *superblock*. In NTFS, it is stored in the *master file table*.
- **Directory Structure**: It is located per file system and is used to organize files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.
- **File Control Block**: It contains many details about the file. It has a unique identifier number to allow association with a directory entry.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. Then, the system reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. Now that a file has been created, it can be used for I/O.

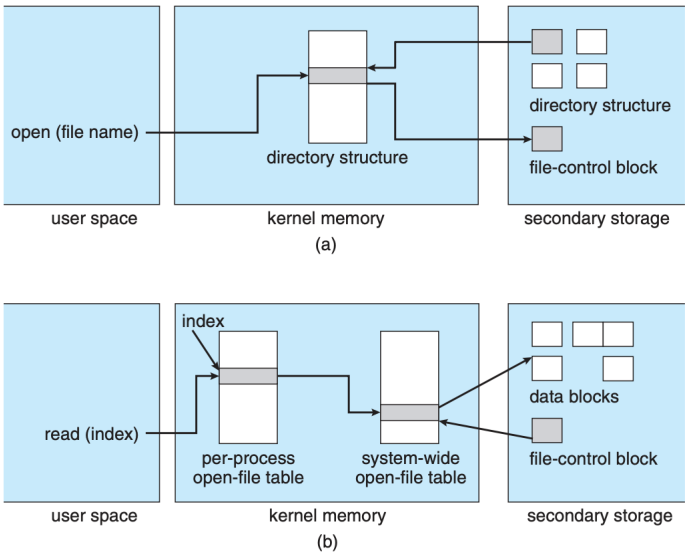

Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

Figure 7: File Open and File Read

2.2.6 Directory Implementation

- **Linear List**: The simplest way to implement a directory is to use a linear list of file names with pointers to data blocks. To create a new file, we must first search the directory to make sure that there are no existing files with the same name. Then, add a new entry to the end of the directory. To delete a file, search for the named file in the directory, and then release the allocated space. However, the disadvantage of a linear list is that *we need a linear search to find the file*. Therefore, this method is simple to program but time-consuming to execute. Directory information is used frequently, and users may notice when access is slow. In fact, many operating systems implement a software cache that stores the most recently used directory information. A cache hit does not require a constant re-reading of information from disk.
- **Hash Table**: Another data structure is hash tables. In this method, linear list stores directory entries, but a hash data structure is also used. The hash table takes the calculated value from the file name and returns a pointer to the file name in the linear list. This can significantly reduce directory browsing time. The main difficulty is usually a fixed size and the dependency of the hash function on that size. For example, if an existing hash table is full and we want to create a new file, we need to increase the size of the hash table, a new hash function that needs to remap the file names accordingly, and reconfigure the existing directory entries to reflect the new hash function values.

2.2.7 Hard Disk Structure

- **Platter**: It is a magnetic disc that records actual data. There are multiple platters as shown in the figure, and they can be used back and forth. A platter consists of multiple tracks.
- **Track**: It is a region that forms a concentric circle of the platter.
- **Sector**: The area where a track is divided into several is called a sector. The sector size

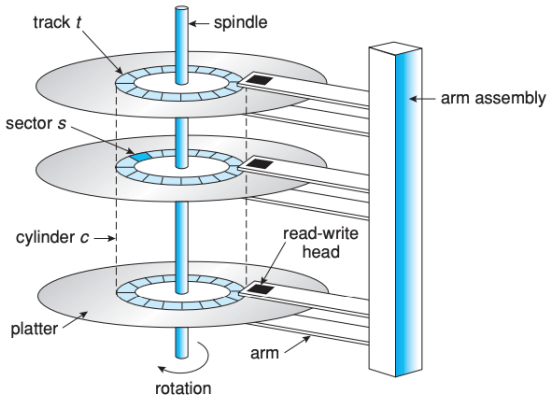

Figure 10.1 Moving-head disk mechanism.

Figure 8: Inside the Hard Disk

is typically 512 bytes, and it is usually used in groups.

- **Cylinder**: A cylinder is the set of the same track positions in all platters.

Earlier, it was said that sector is used in groups, which is called a block. Hard disks are also called block devices because they read and write in blocks. A simple way to check that your hard disk is reading and writing block by block is to write down the alphabet 'a' in a notepad program and save it. 'a' has a size of 1 byte as a character, and if you check the properties of the actual saved text file, you can see that 4KB (one block size) is allocated to the disk. (The actual disk allocation size varies by operating system. The following tests were performed in a MacBook OS environment.) Therefore, a disk can be seen as a collection of empty blocks. So *how does the operating system allocate a free block for each file?*

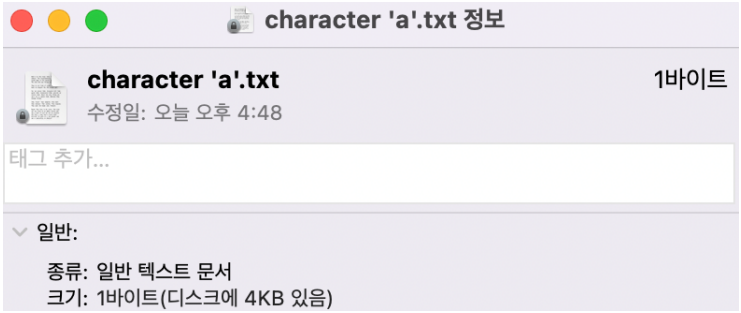

Figure 9: Example File

2.2.8 Allocation Methods

In almost every case, many files are stored on the same disk. The main problem is how to allocate space for these files so that they can effectively utilize disk space and access files quickly. The three main ways to allocate disk space are popular: continuous, linked, and indexed.

• *Contiguous Allocation*: Continuous allocation is literally allocating a file in contiguous blocks. The advantage of continuous allocation is that the movement of the disk header can be minimized, thereby increasing I/O performance. This method is used by IBM in the past, and is mainly suitable for video, music, VOD, etc. Contiguous allocation is stored sequentially, so the

operating system can access the desired block directly with the starting block number obtained from the directory. However, this method has major drawbacks and is rarely used today. When allocating and erasing files repeatedly, there is an empty space (hole) in the middle, and since continuous allocation must find contiguous space, external fragmentation problems occur. External fragmentation results in a very high waste of disk space. Another problem is that the actual size is unknown when saving the file. In particular, files that are used continuously can continue to grow in size, so it is very inappropriate to allocate them continuously.

• *Linked Allocation*: With linked allocation, each file is a list of disk blocks and the disk blocks can be scattered anywhere on the disk. To create a new file, select any empty block as the first block, and if the file grows, we only need to assign another block and link it to the existing block. Linked allocation can be assigned regardless of location, so there is no external fragmentation problem (= no disk waste). However, there are also various problems with linked allocation:

- Sequential access is possible, but direct access is **not**. Because the blocks of the file are all scattered, so we cannot directly access the block at the desired location with the starting block number.
- More than 4 bytes of damage is caused to store the pointer.
- *Low reliability*: if the pointer of the middle block breaks, all subsequent blocks are not accessible.
- *Slow speed*: Since the blocks are all scattered, there is just as much movement of the disk header.

One of the improvements to the above problems is the FAT (File Allocation Table) system, which is the same connection allocation method. The FAT system collects only the pointers to the next block, creates a table, and stores it in one block.

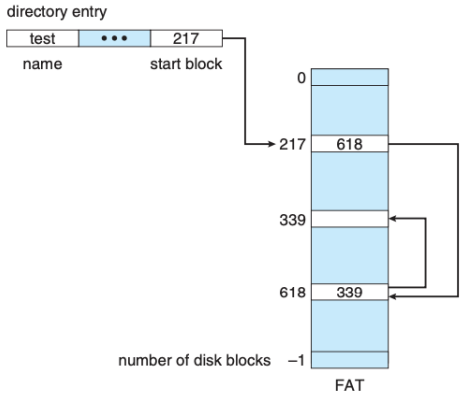

Figure 12.7 File-allocation table.

Figure 10: FAT System Table

If we look at the FAT stored in block 0, the index of the table is the block number of the entire disk, and *each index stores the next block number*. The table entry in the last block has a special value (-1) indicating the end of the file.

• Indexed Allocation: Index allocation, like connection allocation, assigns data to random block numbers, but stores the allocated block pointers separately in a block. These blocks are called index blocks, and there is one index block per file. Index allocation differs from other

allocations in directory information, which stores the index block number rather than the starting block number.

Index allocation can be accessed directly and since there is no need to allocate continuously, there is no external fragmentation problem. Index allocation is primarily used by Unix/Linux. The disadvantage of index allocation is that even small files use one block as an index block, so storage space is lost. And we can't store a large file with a single index block. Let's take a look at the combined scheme that Unix uses to solve this.

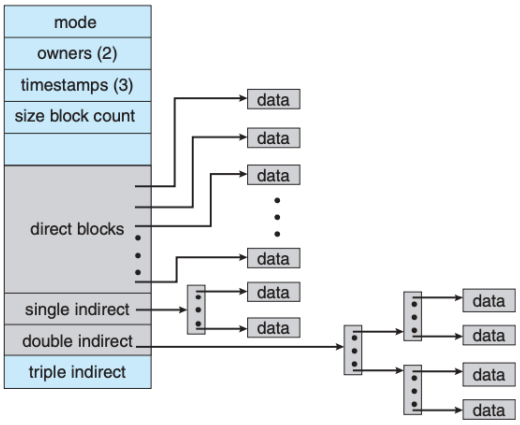

Figure 12.9 The UNIX inode.

Figure 11: UNIX Inode

It keeps the first 15 pointers of the index block at the index of the file, and the first 12 pointers point to the direct block. The direct block has the address of the block containing the data of the file. Therefore, there is no need for an index block for data in small files that do not exceed 12 blocks. And the next three pointers point to indirect blocks. The three pointers point to a single indirect block, a double indirect block, and a triple indirect block, respectively, which is an index block that stores the address of the block rather than the block that stores the data.

2.3 References

For some figures, we borrowed figures from Avraham Silberschatz, Operating System Concepts 10^{th} edition. The copyrights of each figures belongs to their respective authors.

3 Compiling and Building

There are 1 directory in this `git`.

- `MyFS`: Simple file system implementation

`MyFS` supports both `CMake` and `Makefile`. By using the `Makefile`, you can compile the project. `Makefile` includes following optional recipes:

- `clean`: For cleaning up object files and compiled program.
- `debug`: For debugging purpose. This will enable `-DDEBUG` option.

For example, if you were to compile and build `MyFS`, you can do it so by following command:

```
$ cd ./MyFS
$ make
```

Code Snippet 1: Using `Makefile` for compiling MyFS

This will automatically build project and generate an executable file named `MyFS`. If you would like to use `CMake` instead, you can do it so by following commands:

```
$ cd ./MyFS
$ mkdir build & cd build
$ cmake .. & make .
```

Code Snippet 2: Using `CMake` for compiling MyFS

Please place at least one valid `.img` file in the same directory as `MyFS` executable. Otherwise, the program will consider that there is no disk images and will quit execution right away. For the best practice, please place `disk.img` that professor offered us in the same directory as `MyFS`. No other `.img`s were checked working.

For executing `MyFS` you can do it so by following command:

```
$ ./MyFS
...
            Simple File System Simulator
[INFO] Successfully mounted disk 0: disk.img
MyFS@root >>
```

Code Snippet 3: Executing `MyFS`

Each commands will be explained in later pages.

4 Design and Sketch

4.1 Choosing Topic

There were too many choices for this project. Meaning that not only we were allowed to chose project 2 or project 3, we also had choice of implementing project 3 in our way. So, the fundamental reason that we chose project 3 over project 3 was the fact that it is more 'realistic' than the virtual memory. That is, virtual memory implementation will live in a virtual world that is quite different from our real computers. Of course, we could have implemented page tables, swapping, dynamic address, shared memory and etc. However those things were not that much interesting compared to our own file system in terms of 'realism'. However, with file system implementation, we can kind of mimic OS operations and can literally store and read data into our own file system. Therefore we chose project 3 over project 2.

Within project 3, there were also lots of choices that we can make. Since in class we talked about file system after virtual memory, project 3 was kind of running short on time. There were some brilliant ideas on project 3, however was not able to implement due to time being:

- **Scheduling**: Since we have implemented a scheduling simulation in project 1, we were thinking of combining our file system into the program. Since the scheduling were performing CPU burst and IO burst sequentially, we thought of implementing the IO burst into our own file system.
- **RAID 0**: In our mid term exam, the last question was about having a single file across many disks. That kind of gave us the inspiration and motivation for making a simple RAID system with RAID 0. Implementing a real RAID 0 system would have been a tough subject, however implementing a simple strafing have seemed to be feasible. Also, we were planning to make some comparisons about reading and writing operations to one disk versus reading and writing operations to multiple RAID 0 disks.
- **RAID 3 and 4**: As we have mentioned before, RAID seemed to be a good topic. Also since we were thinking about implementing strafing, we also wanted to implement parity checking in a single disk. We also planned to have some comparison between RAID and non raid disks in terms of failure and data loss as well.

Those three subjects seemed to be very interesting. However, since we were running short on time, we had select a topic that we can implement in time for sure. Therefore those three subjects were not selected. This does not mean those 3 topics were discarded forever. We will be implementing those topics if we have some free time in the future. After lots of consideration, we came to conclusion that we would like to make a simple file system that works like `bash`. Therefore, we decided to make a simple (so called) terminal that can interact between user and perform disk operations: *write, read, move, copy and etc.*

4.2 Requirements

In order for us to implement those features, we need to have some basic bootstrapping on disk file system as well as implementing advanced features. The basic features that we must implement were as it follows:

- **Mounting disk**: In order for us to use a disk, we need to mount it. This sounds easy, however needs some internal operations. Those operations require: reading superblock, reading inode table.

- **Directory**: For our program to correctly print out the files and its contents, we need to implement directory. Also the directory must indicate a specific inode that a specific directory entry is using.
- **File**: Including directory, we need to make our program read and write into specific disk blocks in order to read contents from a specific file. The actions that were required were: read, write, append.

Those three elements were the very basic elements that were required. On top of those basic requirements, we wanted to put more advanced features. Those were:

- **Directory Navigation**: We need to make our program traverse throughout the directories in the file system just like Linux environments. For example, we wanted `cd` command from Linux.
- **Creation**: The program needed to create a new empty file when user requested. This requires assigning empty inode and assigning empty blocks. We wanted `touch` and mkdir command from Linux.
- **Deletion**: Since we have feature of creating a new file, we also needed to have deletion features. Not only just deleting a single regular file, we also wanted to implement recursive file deletion when deleting a directory. That is, we wanted `rm` and `rm -rf`.
- **Read**: Since basic requirements supported file reading, we wanted to implement a simple interface for file open and read combined. That is, we wanted to mimic `cat` command from Linux.
- **Write and Appending**: Writing into a file and appending into a file is necessary as well. We wanted a simple interface that supports easy writing and easy reading that internally performs file open and write (or append). This was to mimic `echo "foo" > bar.txt` or `echo "foo" » bar.txt`.
- **Copy**: Modern file system supports copying file. Of course, we could have just copied a duplicate file. However, since our program runs in a small disk, creating a new file whenever a file was being copied was not a good idea. Therefore, we wanted "**Copy on Write**" action implemented into our program.
- **Permissions**: Linux offers some file permissions. We also wanted to implement the feature of file permission editing just like `chmod`. We also considered about having multiple users and making each user have some permissions as well. (The multiple user part was not implemented due to the time being)
- **Move and Rename**: Just like Linux, we wanted to make our program rename a file or directory when user requested. Also we wanted to make files 'movable'. This was intended to mimic mv command in Linux.
- **Stats and Listing**: In order for us to get information on each files and directories, we needed something like `stat` and `ls`. Those were implemented as well.
- **Caching**: This is not 100% caching, however the intention was to have the directory structure cached in the memory. By this, we wanted to make our program access to a file faster.

4.3 Designing Directory

The most important thing that we needed to implement was the directory structure. There are lots of ways to implement directory structures. However, the one that we chose to implement our directory structure was "**Left Child Right Sibling**" tree. The reason of choosing left child right sibling tree for storing directory structure was that this the fact that it very easy for modification. In file system, we need some operations that remove link from a single entry, such as delete and move. Also we need some operations that add links to the entire directory structure, such as creation and move. Therefore, due to ease of modifying the tree structure, we selected left child right sibling tree for storing our directory structure. Let's assume that we have following directory stored in our file system:

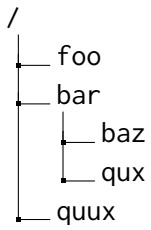

Figure 12: Example file system structure

As its name indicates, child element goes into the left side of the node. The for siblings, the node goes into the left side. With the directory structure, we can make following understanding:

- `/` has child `foo`, but has no sibling
- `foo` has no child, but has a sibling `bar`
- `bar` has a child `baz`, and also has a sibling `quux`
- `baz` has no child, but has a sibling `qux`.
- `quux` has no child and no sibling.

With left child right sibling tree, the structure will be represented as following tree:

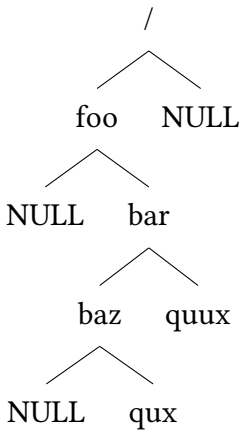

Figure 13: Example Left Child Right Sibling Tree

Please be aware that for leaf nodes, the sibling and child is not denoted. .Huge thanks to `forest` and `tikz` for this simple yet beautiful tree. Now, let's suppose that we wanted to delete

/bar/baz from the directory, the tree will be transformed into the following tree: Now let's

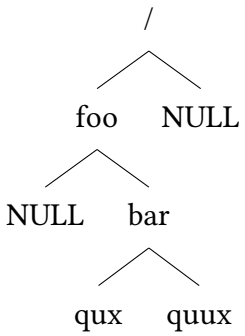

Figure 14: Deleted /bar/baz

suppose that we would like to create a file /corge. Then the tree will become something like below:

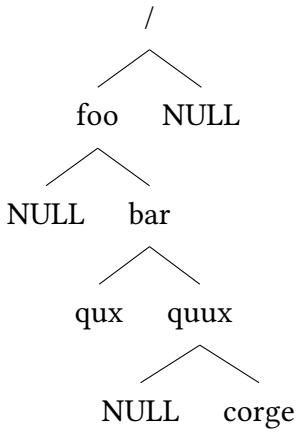

Figure 15: Created /corge

As you can see, this tree is very easy to modify as well as easy to maintain. As we all know, implementing a tree structure in C is also easy with a little help of pointer operations. We made a struct data type named entry_t that will help us represent the tree. struct entry_t is defined in disktree.h.

```c
struct entry_t {
    char name[16];
    unsigned int disk_index;
    unsigned int inode_index;
    struct entry_t *sibling;
    struct entry_t *child;
    struct entry_t *parent;
};
```

Code Snippet 4: Definition of struct entry_t

For easy operations in the future, such as `cd ..` and `mv ...`, there is extra field that points to the parent element. With `struct entry_t`, the pseudo-code for adding an entry into a directory tree will be like below:

Algorithm 1 Adding Entry to Directory

```
procedure INSERT(new, directory)
    if directory.child = NULL then                    ▷ The directory was empty
        directory.child ← new
    else
        peer ← directory.child
        while peer.sibling ≠ NULL do                  ▷ Find the last sibling
            peer ← peer.sibling
        end while
        peer.sibling ← new                            ▷ Add sibling to last sibling
    end if
```

The pseudo-code will insert a new entry into the directory. Since the algorithm needs to find the last sibling, the time complexity will become $O(n)$ if there are n entries in the target directory. Yes, there will be a better solution with better time complexity, however the best we can think of is this one until now. Since we have checked inserting a new entry into a directory, let's check removing an entry from the directory. The pseudo-code will become something like below:

```
procedure REMOVE(target, directory)
    peer ← directory.child
    while peer.sibling ≠ NULL do                      ▷ Find sibling connected to target
        if peer.sibling = target then
            Break loop
        end if
        peer ← peer.sibling
    end while
    peer.sibling ← target.sibling                     ▷ Un-link target
```

The pseudo-code will remove entry from the directory if found. For simple explanation, this pseudo-code assumes that the file exists in the directory. This also is an $O(n)$ operation when there is n entries in the target directory. In real implementations, to avoid `SIGSEV`, there are some protection measures added as well.

Those two pseudo-codes were implemented under various functions in `diskutil.c`. To name a few functions that are using this pseudo-code, the following list are the functions:

- `create_file`: When creating a new file.
- `delete_file`: When deleting a file.
- `r_delete_directory`: When deleting a directory.
- `copy_file`: When copying a file.
- `handle_cow`: When *Copy on Write* was triggered.
- `move_file`: When moving a file.

5 Implementation and Usage

Since there are lots of features implemented in this program, we decided to introduce each implementations and usages at the same time. This program offers a list of commands and each commands were designed to mimic actions from the original commands in Linux. Those commands are as it follows:

Command	Description	Original Linux Command
`ls`	Lists element in this directory	`ls`
`cat`	Shows file content	`cat`
`chmod`	Changes file permission	`chmod`
`stat`	Shows file stats	`stat`
`touch`	Creates empty file	`touch`
`mkdir`	Creates empty directory	`mkdir`
`rm`	Removes a file	`rm`
`rmdir`	Removes a directory recursively	`rm -rf`
`vstat`	Prints out volume stats	N/A
`write`	Writes data to file	`echo "foo" > bar`
`append`	Appends data to file	`echo "foo" » bar`
`cd`	Changes current working directory	`cd`
`cp`	Copies a file	`cp`
`rename`	Renames a file or directory	`mv`
`mv`	Moves a file into other directory	`mv`
`cwd`	Prints current working directory	`cwd`
`xmas`	Prints Christmas tree	N/A
`cxmas`	Prints Christmas tree in color	N/A

Table 1: Table of Implemented Commands

All commands were confirmed it working on a private server with following environments:

- **OS**: Ubuntu 22.04 LTS
- **CPU**: Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz
- **RAM**: 32GB
- **Endian**: Little Endian

Also, the program was checked it working with the disk image `disk.img` that professor gave us. With other `.img` files, this program was not checked it working.

5.1 ls

5.1.1 Feature

This command prints out all the entries in current working directory.

5.1.2 Usage

```
ls <args>
```

This program offers following arguments

- `-al`: Shows permission, file size and name of all files.
- `-ald`: Shows detailed stats of each files, including inode and disk information

5.1.3 Example Outputs

With default `ls`, it will show entries and its names like following:

```
MyFS@root >> ls
file_1  file_2  file_3  file_4  file_5  file_6  file_7  file_8  file_9  ...
file_72  file_73  file_74  file_75  file_76  file_77  file_78  file_79  ...
file_82  file_83  file_84  file_85  file_86  file_87  file_88  file_89  ...
file_92  file_93  file_94  file_95  file_96  file_97  file_98  file_99  ...
```

With `ls -al`, it will show entries and its names with permission and size like following:

```
MyFS@root >> ls -al
-rwxrwxrwx         37 file_1
-rwxrwxrwx         40 file_2
...
-rwxrwxrwx         38 file_99
-rwxrwxrwx         42 file_100
```

With `ls -ald`, it will show inode information and disk index of each files like following:

```
MyFS@root >> ls -ald
-rwxrwxrwx         37 file_1(Inode 3 @ disk 0)
-rwxrwxrwx         40 file_2(Inode 4 @ disk 0)
...
-rwxrwxrwx         38 file_99(Inode 101 @ disk 0)
-rwxrwxrwx         42 file_100(Inode 102 @ disk 0)
MyFS@root >>
```

5.1.4 Implementation

This was implemented by traversing all the siblings of current directory. The pseudo code is like below:

Algorithm 2 Directory Listing

```
procedure LS(directory)
    if directory.child = NULL then                ▷ The directory was empty
        Return
    else
        peer ← directory.child
        while peer.sibling ≠ NULL do              ▷ Print all names of entries
            Print peer.sibling.name
            peer ← peer.sibling
        end while
    end if
```

The command is implemented with following functions:

- `ls`: Under `ui.c` for wrapping `impl_ls`
- `impl_ls`: Under `diskutil.c` for printing out all the entries.

5.1.5 ETC

By default, this program sets a line break in every 10 elements. If you would like to how more files in a single line, you can modify the length by defined as `LS_SPLIT_COUNT` under `diskutil.h`.

5.2 cat

5.2.1 Feature

This command will not read data from a directory file

This command prints out the content in a file.

5.2.2 Usage

```
cat [file]
```

5.2.3 Example Outputs

When executing `cat` to a single file it will be shown like following:

```
MyFS@root >> cat file_1
file 1 741474268 913453413 395175819
```

When executing `cat` to a directory file it will be shown like following:

```
MyFS@root >> cat directory
cat: directory: is a directory
```

When executing `cat` to a non-existing file it will be shown like following:

```
MyFS@root >> cat unknown
cat: unknown: No such file or directory
```

5.2.4 Implementation

When this command is called, following operations are held in sequence:

1. Open file in `R` mode and lock file.
2. Retrieve block information from inode.
3. Read block information and store it to buffer.
4. Close file and unlock file, then print it out.

The command is implemented with following functions:

- `cat`: Under `ui.c` for wrapping `impl_cat`
- `impl_cat`: Under `diskutil.c` for opening and requesting read then printing buffer.
- `read_file_data`: Under `diskutil.c` for reading actual data from disk in binary then storing it to buffer.

5.3 chmod

5.3.1 Feature

This command changes permission of a file, including a directory file.

5.3.2 Usage

```
chmod [permission] [file]
```

For permission, this command uses the same permission as the Linux uses. Consider a example like below:

- User has all `RWX` permission.
- Group has only `R` permission.
- Others have only `R` permission.

This will make permission as following:

Target	R	W	X
User	1	1	1
Group	1	0	0
Others	1	0	0

Table 2: Example Permission

Therefore, this will make the total permission as `744`. So, example execution will be `chmod 744 foo`.

5.3.3 Example Outputs

When executing `chmod` to a file named `foo` with original permission of `777` and changing it to `744`:

```
MyFS@root >> stat foo
   File: foo    Empty file: 1
   Size: 3    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0777/-rwxrwxrwx)
MyFS@root >> chmod 744 foo
MyFS@root >> stat foo
   File: foo    Empty file: 1
   Size: 3    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0744/-rwxr--r--)
```

When executing `chmod` with invalid permission `999`

```
MyFS@root >> chmod 999 foo
chmod: invalid mode: ‘999’
```

When executing `chmod` to an unknown file.

```
MyFS@root >> chmod 777 bar
chmod: cannot access 'bar': No such file or directory
```

Code Snippet 5: Example Execution of `cat` Unknown File

5.3.4 Implementation

When this command is called, following operations are held in sequence:

1. Lock file and check if copy on write is required.
2. Retrieve inode from `struct entry_t`.
3. Change permission to the designated one.
4. Write inode back to the disk.
5. Unlock file

The command is implemented with following functions:

- `chmod`: Under `ui.c` for wrapping `impl_chmod`
- `impl_cat`: Under `diskutil.c` for finding inode from `struct entry_t` and changing the information then writing back to the disk.

5.3.5

ETC This program modified the definitions of file permissions defined in `fs.h` that professor gave us. For ease of use and for better permission control, we have changed values like following:

MACRO	Original	New
INODE_MODE_AC_USER_R	0x001	0x400
INODE_MODE_AC_USER_W	0x002	0x200
INODE_MODE_AC_USER_X	0x004	0x100
INODE_MODE_AC_OTHER_R	0x010	0x004
INODE_MODE_AC_OTHER_W	0x020	0x002
INODE_MODE_AC_OTHER_X	0x040	0x001
INODE_MODE_AC_GRP_R	0x100	0x040
INODE_MODE_AC_GRP_W	0x200	0x020
INODE_MODE_AC_GRP_X	0x400	0x010

Table 3: Example Permission

5.4 stat

5.4.1 Feature

This command prints the statistics in a file

5.4.2 Usage

```
stat [file]
```

5.4.3 Example Outputs

If we had a file that is an original file without any indirection to other inode, this will be shown:

```
MyFS@root >> stat file_1
   File: file_1    Empty file: 0
   Size: 37    IO Block: 1
   Disk: disk.img (0)    Inode: 3
   Access: (0777/-rwxrwxrwx)
   Indirection(s) from: 105,
```

If we were performing `stat` to a copied file, this will be shown. In this case `copied` was a file copied from `file_1` and was not triggered copy on write operation yet, this will be shown:

```
MyFS@root >> stat copied
   File: copied    Empty file: 0
   Size: 37    IO Block: 1
   Disk: disk.img (0)    Inode: 105
   Access: (0777/-rwxrwxrwx)
   Indirection to: 3
```

5.4.4 Implementation

When this command is called, following operations are held in sequence:

1. Retrieve inode from `struct entry_t`.
2. Check in-going and out-going indirection for this inode.
3. Print out the information.

The command is implemented with following functions:

- `stat`: Under `ui.c` for wrapping `impl_stat`
- `impl_stat`: Under `diskutil.c` for reading and printing out the inode information.

5.5 touch

5.5.1 Feature

This command creates an empty regular file.

5.5.2 Usage

```
touch [file]
```

5.5.3 Example Outputs

We wanted to create a file named new, then `cat` the new file:

```
MyFS@root >> touch new
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100  new
MyFS@root >> cat new

MyFS@root >>
```

5.5.4 Implementation

When this command is called, following operations are held in sequence:

1. Allocate an empty inode. Initial size is 3bytes for marking that this inode is assigned. Also, the initial permission is set to 777.
2. Allocate an empty block and set first bit to .
3. Write inode and empty block to the disk.
4. Update entry in the left child right sibling tree.

The command is implemented with following functions:

- `touch`: Under `ui.c` for wrapping `impl_touch`.
- `impl_touch`: Under `diskutil.c` for creating an empty regular file.
- `create_file`: Under `diskutil.c` for actually creating an empty file. This function allocates free inode and free data blocks as well.
- `dir_add_child`: Under `diskutil.c` for adding an child element to parent directory.

This program does not manage fragmentation. That is, if the new file acquired disk block number 3, and the next block 4 was already allocated, this will get fragmented. The program does not take care of fragmentation and has no way of relaxing fragmentation. Also, the initial size of the file is set to 3 bytes to mark free inode vector that this one is assigned and is not ready to be used. Therefore, the actual byte written on the disk and the size that inode indicates will differ initially.

5.6 mkdir

5.6.1 Feature

This command creates an empty directory file.

5.6.2 Usage

```
mkdir [file]
```

5.6.3 Example Outputs

We wanted to create a directory named new_dir, then cd into the new directory:

```
MyFS@root >> mkdir new_dir
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100  new_dir
MyFS@root >> cd new_dir
MyFS@root/new_dir >> ls

MyFS@root/new_dir >>
```

5.6.4 Implementation

When this command is called, following operations are held in sequence:

1. Allocate an empty inode. Initial size is 20bytes for marking that this inode is assigned. Also, the initial permission is set to 777.
2. Allocate an empty block and set two disk entries. Those are . and ..
3. Write inode and empty block to the disk.
4. Update entry in the left child right sibling tree.

The command is implemented with following functions:

- `mkdir`: Under `ui.c` for wrapping `impl_mkdir`.
- `impl_mkdir`: Under `diskutil.c` for creating an empty directory file.
- `create_file`: Under `diskutil.c` for actually creating an empty file. This function allocates free inode and free data blocks as well.
- `dir_add_child`: Under `diskutil.c` for adding an child element to parent directory.

This program does not manage fragmentation. Like `touch`, this also does not manage fragmentation. That is, if the new file acquired disk block number 3, and the next block 4 was already allocated, this will get fragmented. The program does not take care of fragmentation and has no way of relaxing fragmentation.

5.7 `rm`

5.7.1 Feature

This command deletes a regular file

5.7.2 Usage

```
rm [file]
```

5.7.3 Example Outputs

We wanted to delete a file named `foo`:

```
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100  foo
MyFS@root >> rm foo
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100
```

5.7.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Check if copy on write operation is required or not. If copy on write is required, this will automatically trigger copy on write.
3. Remove entry from the parent directory's file.
4. Remove link from the left child right sibling tree.
5. Remove inode and blocks. Then mark those inodes and blocks as not being used.
6. Physically change all changes to disk, then unlock file.

The command is implemented with following functions:

- `rm`: Under `ui.c` for wrapping `impl_rm`.
- `impl_rm`: Under `diskutil.c` for removing an regular file.
- `handle_cow`: Under `diskutil.c` for checking of copy on write is required or not. If it is required, this function will automatically perform copy on write.
- `delete_file`: Under `diskutil.c` for deleting a regular file from file system.

When `rm` command was issued, this program will check whether or not to perform copy on write. If there were inodes that were directing this target, all those inodes will be performed copy on write. Therefore using n time more disk space than before.

5.8 rmdir

5.8.1 Feature

This command deletes a directory file

5.8.2 Usage

```
rmdir [file]
```

5.8.3 Example Outputs

We wanted to delete a directory named `foo`:

```
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100  directory
MyFS@root >> rmdir directory
MyFS@root >> ls
...
...  file_95  file_96  file_97  file_98  file_99  file_100
```

5.8.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Delete all child entries recursively, this includes all subdirectories and normal regular files.
3. Remove entry from the parent directory's file.
4. Remove link from the left child right sibling tree.
5. Remove inode and blocks. Then mark those inodes and blocks as not being used.
6. Physically change all changes to disk, then unlock file.

The command is implemented with following functions:

- `rmdir_`: Under `ui.c` for wrapping `impl_rmdir`. This is named `rmdir_` since function `rmdir` is already defined in `unistd.h` sadly.
- `impl_rmdir`: Under `diskutil.c` for removing an directory file.
- `delete_directory`: Under `diskutil.c` for deleting a directory file from file system recursively.

Calling `rmdir` will call recursive function to delete all its child entries. Therefore, if there are too many files under a directory, it can exceed the hard limit of the recursion limit that is defined in the OS. This is highly unlikely to happen since there are not that many files and the maximum directory recursion will be the size of total inodes.

For deleting all the files recursively, this function uses following pseudo-code to delete all its sub directories and its files.

Algorithm 3 Recursive Directory Removal

```
procedure R_DELETE_DIRECTORY(current_directory, child_directory)
    if child_directory.child = NULL then                ▷ This was just a regular file
        Delete file child_directory
    else                                                ▷ This was a sub directory
        peer ← child_directory.child
        while peer ≠ NULL do                            ▷ Iterate through all peers
            if peer.is_directory then
                r_delete_directory(child_directory, peer
                Delete inode, blocks and un-link entry
            end if
            Delete file peer
            peer ← peer.sibling
        end while
        Delete file child_directory
    end if
```

This pseudo-code is implemented as `r_delete_directory` under `diskutil.c`. This function has a bit more of protection in edge cases to prevent `SEGFAULT`. The function `r_delete_directory` will be called by `delete_directory` which gives more abstraction over deleting a entire directory recursively.

5.9 `write`

5.9.1 Feature

This command writes to a specific file

5.9.2 Usage

If content is not enclosed by ", there is a potential of program failing due to `*** buffer overflow detected ***: terminated`. Also, by using `write`, this will overwrite everything that was in the file.

```
write [file] [content]
```

The <content> must be enclosed by ". For example, if we were to use `Hello world`, the example command will be `write foo "Hello world"`.

5.9.3 Example Outputs

We wanted to write into a file named `foo` with content `Hello world`:

```
MyFS@root >> touch foo
MyFS@root >> cat foo

MyFS@root >> write foo "Hello world"
MyFS@root >> cat foo
Hello world
MyFS@root >>
```

Then we wanted to write another data into `foo` with content `Bye world`:

```
MyFS@root >> cat foo
Hello world
MyFS@root >> write foo "Bye world"
MyFS@root >> cat foo
Bye world
MyFS@root >>
```

5.9.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Check if copy on write operation is required or not. If copy on write is required, this will automatically trigger copy on write.
3. Open file as `W` mode and write blocks with the data that came from buffer.

4. Update size of the file in inode then unlock file.

The command is implemented with following functions:

- `write_`: Under `ui.c` for wrapping `impl_write`. This is named `write_` since the name `write` is already defined in `unistd.h`.
- `impl_write`: Under `diskutil.c` for writing into a regular file.
- `handle_cow`: Under `diskutil.c` for checking of copy on write is required or not. If it is required, this function will automatically perform copy on write.
- `write_file_data`: Under `diskutil.c` for physically emitting write operation to the disk.

5.9.5 Copy on Write

When `write` command was called, the program will internally check if copy on write is required. There are two cases that copy on write will be triggered with `write` command. Those are:

- The target being written is directing to another inode. Meaning that this target was a copy of another file, however was not triggered copy on write. Since with the write command, the copied file and the original file starts to become separate. Therefore, this will trigger copy on write and copy target as another independent file.
- The target being written has other inodes directing to it. Meaning that this target is a original data and some other copy files are directing this target. In this case, for all inodes that are directing to this file will become all independent with each other. All inodes will be performed a copy on write.

For example, let's assume that we had file `a` and that was copied by `b` and `c`. The example output will be like below.

```
MyFS@root >> write a "hello world"
MyFS@root >> cp a b
MyFS@root >> cp a c
MyFS@root >> stat a
   File: a    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 103
   Access: (0777/-rwxrwxrwx)
   Indirection(s) from: 104, 105,
MyFS@root >> stat b
   File: b    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0777/-rwxrwxrwx)
   Indirection to: 103
MyFS@root >> stat c
   File: c    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 105
```

```
   Access: (0777/-rwxrwxrwx)
   Indirection to: 103
```

Then we had written data into the `a`. In this case, `b` and `c` will become separate independent files. Example output is like below:

```
MyFS@root >> write a "bye world"
MyFS@root >> cat a
bye world
MyFS@root >> cat b
hello world
MyFS@root >> cat c
hello world
MyFS@root >> stat b
   File: b    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0777/-rwxrwxrwx)
MyFS@root >> stat c
   File: c    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 105
   Access: (0777/-rwxrwxrwx)
```

As you can see, both files `b` and `c` no longer have any indirection to `a`. From now on, each files will be treated as a different file.

For implementation of copy on write, please refer to following functions in the source code provided:

- `handle_cow`: A function that checks whether or not to perform copy on write.
- `process_cow`: A function that actually performs copy on write. This will separate the original file from the copied file. Those two files will become two different files without any relationships.

5.10 append

5.10.1 Feature

This command appends to a specific file

5.10.2 Usage

If content is not enclosed by ", there is a potential of program failing due to `*** buffer overflow detected ***: terminated`.

```
append [file] [content]
```

The `[content]` must be enclosed by ". For example, if we were to use `Hello world`, the example command will be `append foo "Hello world"`.

5.10.3 Example Outputs

We wanted to append into a file named `foo`. `foo` already has `Hello world` and we also would like to append `Hello world` into `foo`:

```
MyFS@root >> touch foo
MyFS@root >> write  foo "Hello world"
MyFS@root >> cat foo
Hello world
MyFS@root >> append foo "Bye world"
MyFS@root >> cat foo
Hello world
Bye world
MyFS@root >>
```

5.10.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Check if copy on write operation is required or not. If copy on write is required, this will automatically trigger copy on write.
3. Open file as `R` mode and copy the content from the disk into a buffer then close file.
4. Append user data into the buffer that came from the disk.
5. Open file as `W` mode and write data from the buffer into the disk then close the file.
6. Update size of the file in inode then unlock file.

The command is implemented with following functions:

- `append`: Under `ui.c` for wrapping `impl_append`.
- `impl_append`: Under `diskutil.c` for appending into a regular file.

- `handle_cow`: Under `diskutil.c` for checking of copy on write is required or not. If it is required, this function will automatically perform copy on write.
- `append_file_data`: Under `diskutil.c` for physically taking append operation to the disk.

5.10.5 Copy on Write

This will trigger copy on write just like it did with `write` operation. Therefore, for better explanation, please check section 4.9.5 for more information. Following example is a simple example that is for copy and write demonstration.

```
MyFS@root >> write a "Hello world"
MyFS@root >> cp a b
MyFS@root >> append a "Bye world"
MyFS@root >> cat a
Hello world
Bye world
MyFS@root >> cat b
Hello world
```

Since from this point, file `a` and file `b` is an independent file. If we check the stats of each file, the results are like it follows:

```
MyFS@root >> stat a
   File: a    Empty file: 0
   Size: 22    IO Block: 1
   Disk: disk.img (0)    Inode: 103
   Access: (0777/-rwxrwxrwx)
MyFS@root >> stat b
   File: b    Empty file: 0
   Size: 12    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0777/-rwxrwxrwx)
MyFS@root >>
```

5.11 cp

5.11.1 Feature

This command copies a regular file. This works in a *"Copy on Write"* way.

5.11.2 Usage

If user copied a file into a name that exists, this command will overwrite discard everything in the original file and then store the copied file

```
cp [source] [destination]
```

5.11.3 Example Outputs

We wanted to copy a file named `foo` into `bar`:

```
MyFS@root >> touch foo
MyFS@root >> write foo "I am foo"
MyFS@root >> cp foo bar
MyFS@root >> cat bar
I am foo
MyFS@root >>
```

5.11.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Allocate a free inode and copy everything from the source's inode. Then emit change to disk.
3. Update the parent directory file to include the copied file then unlock file.

The command is implemented with following functions:

- `cp`: Under `ui.c` for wrapping `impl_cp`.
- `impl_cp`: Under `diskutil.c` for copying a regular file.
- `copy_file`: Under `diskutil.c` for actually copying a file and then adding copied entry to the parent directory.

5.11.5 Copy on Write

To save resources, when a file was copied, the new file will have just the inode. It will direct to another inode. Once there is a change in the original file or the copied file, the copy on write will be triggered. Then those two files will become separate files. For marking whether if an inode is a indirection inode or an original inode, we use `struct inode`'s `indirect_inode`. By checking the `struct inode`'s `indirect_inode`, we can retrieve following information:

- `indirect_inode` is -1: This means that this inode is non copied inode.
- `indirect_inode` is not 1: The value of `indirect_inode` is the place where we shall refer.

This program supports following case as well with recursive inode direction:

1. Original file is a.
2. Copy a as b.
3. Copy b as c.

In this case, c will have indirection to the original file a. Let' see an example:

```
MyFS@root >> touch a
MyFS@root >> cp a b
MyFS@root >> cp b c
MyFS@root >> stat a
   File: a    Empty file: 1
   Size: 3    IO Block: 1
   Disk: disk.img (0)    Inode: 103
   Access: (0777/-rwxrwxrwx)
   Indirection(s) from: 104, 105,
MyFS@root >> stat b
   File: b    Empty file: 1
   Size: 3    IO Block: 1
   Disk: disk.img (0)    Inode: 104
   Access: (0777/-rwxrwxrwx)
   Indirection to: 103
MyFS@root >> stat c
   File: c    Empty file: 1
   Size: 3    IO Block: 1
   Disk: disk.img (0)    Inode: 105
   Access: (0777/-rwxrwxrwx)
   Indirection to: 103
MyFS@root >>
```

This feature was implemented by `find_original_inode` defined in `diskutil.h` The function works in a recursive way and will find the original data.

5.12 `rename`

5.12.1 Feature

This command renames a file or directory

5.12.2 Usage

```
rename [target] [name]
```

5.12.3 Example Outputs

We had a file named `foo` and had content of `I am foo`. Then we wanted to change the name of `foo` into `bar`.

```
MyFS@root >> touch foo
MyFS@root >> write foo "I am foo"
MyFS@root >> cat foo
I am foo
MyFS@root >> rename foo bar
MyFS@root >> ls
...
... file_95  file_96  file_97  file_98  file_99  file_100  bar
MyFS@root >> cat bar
I am foo
```

5.12.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Change name in the parent directory file.
3. Update the name stored in `struct entry_t` then unlock file.

The command is implemented with following functions:

- `rename_`: Under `ui.c` for wrapping `impl_rename`. This is named `rename_` since `rename` is already defined in `stdio.h` sadly.
- `impl_rename`: Under `diskutil.c` for renaming a file.
- `rename_file`: Under `diskutil.c` for actually renaming a file and then changing entry in the parent directory.

Since this is just changing names, this will not trigger copy on write.

5.13 mv

5.13.1 Feature

This command moves a file or directory into another directory

5.13.2 Usage

```
mv [target] [destination]
```

5.13.3 Example Outputs

We had a file named `foo` and had content of `I am foo`. Also we had a directory named `new_directory`. Then we wanted to move `foo` into the `new_directory`.

```
MyFS@root >> touch foo
MyFS@root >> write foo "I am foo"
MyFS@root >> mkdir new_directory
MyFS@root >> mv foo new_directory
MyFS@root >> ls
... file_95  file_96  file_97  file_98  file_99  file_100  new_directory
MyFS@root >> cd new_directory
MyFS@root/new_directory >> ls
foo
MyFS@root/new_directory >> cat foo
I am foo
```

`foo` and its contents were successfully moved. Now, lets make another directory named `bar` in the root directory. Then move `new_directory` into `bar`.

```
MyFS@root/new_directory >> cd ..
MyFS@root >> ls
... file_95  file_96  file_97  file_98  file_99  file_100  new_directory
MyFS@root >> mkdir bar
MyFS@root >> mv new_directory bar
MyFS@root >> ls
... file_95  file_96  file_97  file_98  file_99  file_100  bar
MyFS@root >> cd bar
MyFS@root/bar >> ls
new_directory
MyFS@root/bar >> cd new_directory
MyFS@root/bar/new_directory >> ls
foo
MyFS@root/bar/new_directory >> cat foo
I am foo
MyFS@root/bar/new_directory >>
```

As you can see, foo is in `/bar/new_directory` with all its contents. `cd` supports `..` directory as

well.

```
MyFS@root >> mkdir foo
MyFS@root >> cd foo
MyFS@root/foo >> ls

MyFS@root/foo >> touch bar
MyFS@root/foo >> write bar "I am bar"
MyFS@root/foo >> ls
bar
MyFS@root/foo >> mv bar ..
MyFS@root/foo >> cd ..
MyFS@root >> ls
...  file_95  file_96  file_97  file_98  file_99  file_100  foo
bar
MyFS@root >>
```

We can move files across the directories freely like the demonstration.

5.13.4 Implementation

When this command is called, following operations are held in sequence:

1. Find the target from `struct entry_t`, then lock inode.
2. Find the destination entry.
3. From the directory file where target was originally at, delete entry for target. Also update inode for the directory.
4. Then add file entry to the destination entry, also update inode for the directory.
5. Update left child right sibling tree accordingly then unlock file.

The command is implemented with following functions:

- `mv`: Under `ui.c` for wrapping `impl_rename`.
- `impl_move`: Under `diskutil.c` for moving a file or directory.
- `move_file`: Under `diskutil.c` for actually moving a file and then changing entry in the parent directory.

5.14 cd

5.14.1 Feature

This command changes current working directory.

5.14.2 Usage

```
cd [directory]
```

5.14.3 Example Outputs

When we were moving into a directory named `directory`:

```
MyFS@root >> mkdir directory
MyFS@root >> cd directory
MyFS@root/directory >> ls

MyFS@root/directory >>
```

When we perform `cd`, the prompt also changes as well. `cd` also supports moving into upper directory:

```
MyFS@root/directory >> cd ..
MyFS@root >> ls
...  file_95  file_96  file_97  file_98  file_99  file_100  directory
MyFS@root >>
```

5.14.4 Implementation

When this command is called, following operations are held in sequence:

1. Retrieves target directory's entry
2. Set current directory as the target directory's entry.

The command is implemented with following functions:

- `cd`: Under `ui.c` for changing directory.

5.15 cwd

5.15.1 Feature

This command prints out the current working directory.

5.15.2 Usage

```
cwd
```

5.15.3 Example Outputs

When we were inside a directory named `directory`:

```
MyFS@root >> cd directory
MyFS@root/directory >> cwd
root/directory
MyFS@root/directory >>
```

5.15.4 Implementation

When this command is called, following operations are held in sequence:

1. Print out current working directory's path.

5.16 vstat

5.16.1 Feature

This command prints out the information of volume.

5.16.2 Usage

```
vstat [index]
```

5.16.3 Example Outputs

When we had a volume and the index was `0`:

```
MyFS@root/directory >> vstat 0
   Volume Name: Simple_partition_volume
   Used Inodes: 104   Free Inodes: 120
   Used Blocks: 104   Free Blocks: 3984
MyFS@root/directory >>
```

5.16.4 Implementation

When this command is called, following operations are held in sequence:

1. Print out super block's data

5.17 xmas

For your best experience, please use PuTTY if you are using Windows. Since Windows prints out backslash terribly, the ASCII art might get ruined. Also please set your terminal size big enough to print the whole tree out.

5.17.1 Feature

This command prints out Christmas tree in ASCII art.

5.17.2 Usage

```
xmas
```

5.17.3 Example Outputs

On Christmas day, please use this command.

```
MyFS@root >> xmas
         |
        -+-
         A
        /=\                /\  /\    ___  _ __  _ __ __    __
      i/ 0 \i             /  \/  \  / _ \| '__|| '__|\ \  / /
      /=====\            / /\  /\ \|  __/| |   | |    \ \/ /
      /  i  \            \ \ \/ / / \___/|_|   |_|     \  /
    i/ 0 * 0 \i                                        / /
    /=========\        __  __                         /_/    _
    /  *   *  \        \ \/ /        /\  /\    __ _  ____  | |
  i/ 0   i   0 \i       \  /   __   /  \/  \  / _` |/ ___\ |_|
  /=============\       /  \  |__| / /\  /\ \| (_| |\__ \  _
  /  0   i   0  \      /_/\_\      \ \ \/ / / \__,_|\____/ |_|
i/ *   0   0   * \i
/=================\
       |___|
                          2022 Fall Semester
                          Yay! I am now free!!!
```

Since we were working around the December, we thought that it would have been better if we add some nice Christmas tree and celebrate Christmas. Therefore we added this `xmas` command. Please stay tuned for another command: `cxmas`.

5.18 cxmas

For your best experience, please use PuTTY if you are using Windows. Since Windows prints out backslash terribly, the ASCII art might get ruined. Also please set your terminal size big enough to print the whole tree out, this tree is quite big!

5.18.1 Feature

This command prints out Christmas tree in ASCII art with ANSI colors.

5.18.2 Usage

```
cxmas
```

5.18.3 Example Outputs

On Christmas day, please use this command.

Figure 16: Colored Christmas Tree

6 Future Idea & Conclusion

„What I cannot create, I do not understand."

– Richard Feynman

We learned file system in the class. In the class time, the concept such as *FAT* seemed to be quite straight forward. Therefore, we thought that project 3 would have been a quite simple implementation. However, that turned out to be completely wrong. This was the longest code throughout the whole semester. As Richard Feynman said, if we could not implement a simple file system, we do not understand the concept completely. Therefore we tried hard to implement the code. To close this report, those are the conclusions and future plans for further topics.

6.1 Conclusion

Implementing `MyFS` was a lot of fun. Since we actually managed to build a simple file system that supports some basic commands, it was satisfying to see them working properly as they should. There were some difficulties with implementations. The following list are the things which we had the most challenges but with the most fun in it.

6.1.1 Directory Tree

Since we decided to use "Left Child Right Sibling" tree in order to store the directory structure and for caching the structure, implementing the tree was one of a big huddle. The concept of the tree was quite straight forward. However, with implementing create, delete and moving files, making the directory tree track everything was quite challenging. By implementing this tree based directory system, we were able to learn how to use tree data structure properly.

6.1.2 Writing Disk

Since this project were to implement a file system that actually works, we needed to store data into the physical file in our PC. In order to make this work, we had to use lots of `memcpy`, `memset`, `strcpy` and `malloc`. Not to forget the fact that we actually had to kind of trick `C` to think `struct blocks` as an array of `unsigned char` for offset calculation. Or treat `unsigned int` as an array of 4 `unsigned char` to perform `memcpy`. Yes, this was super fun. One of the most challenging part when calculating the offset was following code:

```
...
while (offset <= buffer_size) {
    // Seek throughout the directory file and look for the file name
    char file_name[0x10] = {0};
    memcpy(file_name, buffer + offset + 0x10, sizeof(unsigned char) * 0x10);
    if (!strcmp(file_name, target)) { // Meaning that we have found the name.
        found_target = 1;
        memset(buffer + offset, 0, sizeof(unsigned char) * 0x20);
        memcpy(buffer + offset, buffer + offset + 0x20,
               (int) (in->size - offset - 0x20) > 0 ?
```

```
                in->size - offset - 0x20 : 0);
        break;
    }
    offset = offset + 0x20;
}
...
```

This is the code used when we had to rearrange directory file's entries when an entry was removed. Let's assume that we had following files:

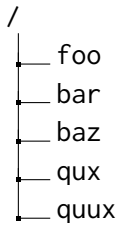

Figure 17: Example rearranging

Suppose that we deleted `bar`. When we delete `bar`, there will be an `0x20` sized offset in the directory file. To avoid having those gaps, we had to shift all entries after `bar`. As you can see, we have bunch of pointer arithmetic operations to calculate offset and bytes size. Therefore, physically writing disks and calculating those offsets were one of the most challenging parts. After finishing the implementation of this file system project, we were able to calculate offsets and perform memory copies far better than we ever could.

6.2 Future Idea

There were lots of great ideas within this `MyFS` project. Some were successfully implemented, meanwhile some could not. These were the things that we could not implement due to constrains:

- **Mount**: We wanted to implement mounting feature. By treating each `.img` file as a separate disk, we wanted to make the program mount a specific volume into a directory just like Linux.
- **Device Files**: We wanted to implement a simple device files using file system. For example, we wanted to perform some actions like `write /dev/screen "hello world"` print out `hello world` into the screen. Or perform give some other device files to interact with.
- **Strict Permissions**: We wanted to make a multi-user environment. Since our file system supports permissions, we wanted to put strict permissions on each files. That is, some user has access to a specific file, when some does not.
- **Real Indirection**: Since the original file system had size limit of 6 disk blocks, we wanted to create an indirection table that can make a single file much bigger.

These ideas were the things we tried to implement in `MyFS` at first, however was not able to. Beside those ideas, there are some other good topics we also wanted to implement. To list a few:

- **RAID**: As we have mentioned before, creating an RAID file system was one of our first goals. As inspired by the mid-term's final question (the 50 points one), we wanted to implement the RAID 0 system and compare the performance between just using one disk. Then make a statistical report on how better it performed. Not only the RAID 0, but also those RAID 3 and 4 that supports parity. Also with RAID 0 and RAID 3 combined for better reliability as well.
- **Disk Generator**: As suggested in the original guide line that professor gave us, we wanted to create a disk generator that generates disk images randomly and correctly. Not only we wanted the disk images, we also wanted to implement some bigger file system generator. Since our file system has limitations of 4 megabytes, we wanted to make a program that actually writes bigger disk images and works across the bigger disk.

Those future ideas are still in our mind and would like to implement more features in the future when we have spare time.

6.3 Final Conclusion

Throughout this semester, we had learned following things:

- **Introduction to OS**: About threads, concurrency, multi-processes.
- **Scheduling**: About scheduling techniques and its metrics, such as round robin, CFS, and etc.
- **Virtual Memory**: About virtual addresses and its usages, such as shared memory, page table, swapping and etc.
- **File System**: About storing files and its applications.

We learned a lot. Throughout this semester, as we have implemented various projects and assignments, we learned following things:

- **Trade-off**: In operating systems, there is no such thing like "answers". For example, if foo is better than bar in terms of a, in the most cases bar is better than foo in terms of b. Round robin gives us fast response time compared to FIFO. However, FIFO gives us faster turnaround time than round robin. We learned that OS is balancing in the middle and selecting reasonable trade-offs that will work the best. Since there was no such thing like "answers", OS was an interesting subject to study.
- **Insight**: Before learning operating systems, we were very short sighted and saw a single tree, not the forest. After learning operating systems, we kind of got knowledge about how things work in a overall system. For example, multi-threading is good. Before getting insights about operating systems, we all thought making a multi-threaded program will always perform better. However, after the class, instead of blindly saying that something is better than something, we started to question ourselves on "is this really better?". For example, with poorly designed multi-threaded program, it will perform worse than single threaded.

It was a great joy of implementing almost everything that we had learned throughout the class. By projects, we were able to understand topics clearer. This was a great class. We will be closing our report here.