RV College of Engineering®

Experiential Learning Phase -I :

CS235AI

Operating Systems

Topic : Simple File System

Hanisha R

1RV22CS244

PRESENTATION CONTENTS

- PROBLEM STATEMENT
- REFERENCE TO OS
- METHODOLOGY
- REFERENCE PAPERS

RV College of Engineering®

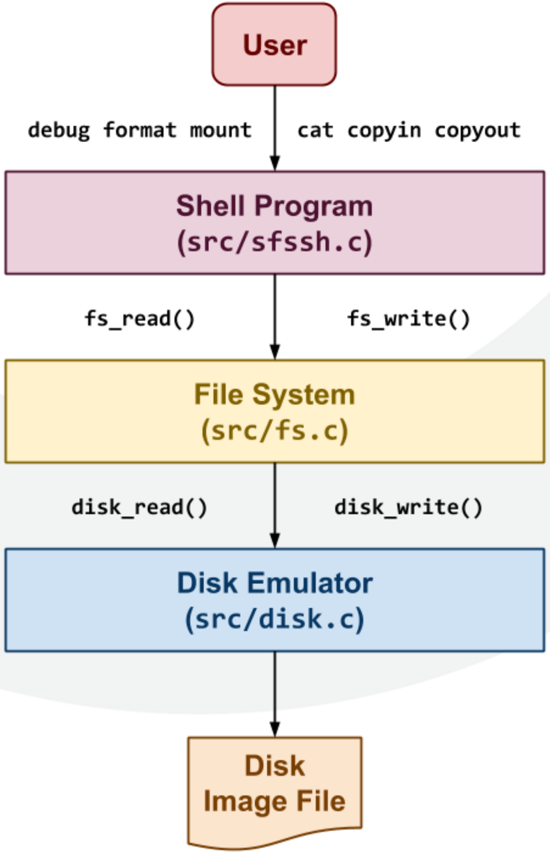
Go, change the world®

Problem Statement

Create SimpleFS, a simplified Unix-like File System, consisting of a shell application, file system component, and disk emulator. Develop in C to manage on-disk data structures, handle disk operations, and provide a user-friendly interface for filesystem tasks. Deliverables include source code, documentation, and a demonstration video, aiming to deepen understanding of file systems and low-level system programming.

Relevance of the project to the course

This project is crucial for understanding operating systems as it involves practical work on filesystem design, disk management, and system-level programming. By implementing SimpleFS, students gain hands-on experience in building filesystem structures, managing disk operations, and interacting with hardware resources. This project enhances understanding of filesystem architecture, storage optimization, and system-level programming, making it highly relevant to the study of operating systems.



RV College of Engineering®

Summary of the Phase I report

Go, change the world®

METHODOLOGY

Design: Create a detailed blueprint of the SimpleFS filesystem structure, including superblock, inode layout, data blocks, and free block bitmap.

Implement: Write code to create the disk emulator, filesystem operations (e.g., fs_debug, fs_format, fs_create), and data structures

Test: Develop test cases to ensure the correct functionality of each filesystem operation and data structure

Debug: Identify and fix any errors or issues discovered during testing to ensure the system operates as expected.

Optimize: Refine the implementation to improve performance, efficiency, and resource utilization

Document : upload on GitHub

fs.h

```
#ifndef FS_H
#define FS_H

void fs_debug();
int  fs_format();
int  fs_mount();

int  fs_create();
int  fs_delete( int inumber );
int  fs_getsize( int inumber );

int  fs_read( int inumber, char *data, int length, int
offset );
int  fs_write( int inumber, const char *data, int
length, int offset );
int  fs_defrag();

#endif
```

disk.h

```
#ifndef DISK_H
#define DISK_H

#define DISK_BLOCK_SIZE 4096

int  disk_init( const char *filename, int nblocks );
int  disk_size();
void disk_read( int blocknum, char *data );
void disk_write( int blocknum, const char *data );
void disk_close();
```

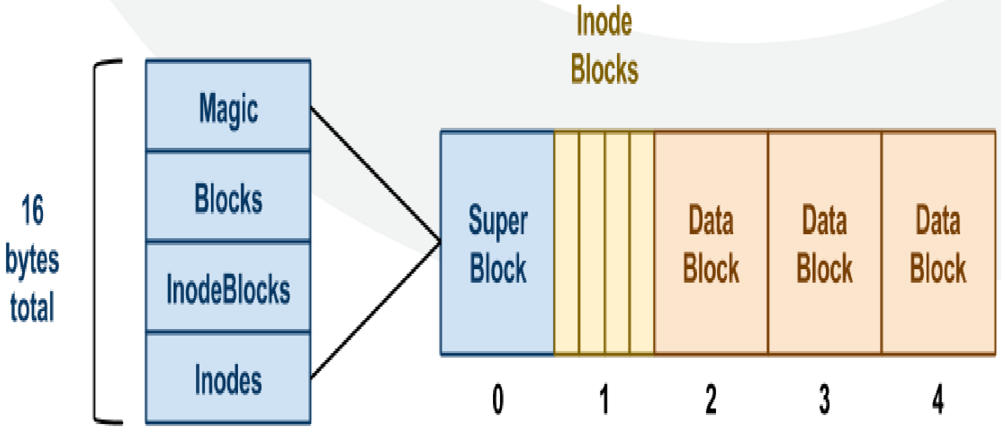
Summary of the Phase I report

- This **superblock** consists of four fields:

1.Magic: The first field is always the MAGIC_NUMBER or 0xf0f03410. The format routine places this number into the very first bytes of the **superblock** as a sort of filesystem "signature". When the filesystem is mounted, the OS looks for this magic number. If it is correct, then the disk is assumed to contain a valid filesystem. If some other number is present, then the mount fails, perhaps because the disk is not formatted or contains some other kind of data.
2.Blocks: The second field is the total number of blocks, which should be the same as the number of blocks on the disk.
3.InodeBlocks: The third field is the number of blocks set aside for storing **inodes**. The format routine is responsible for choosing this value, which should always be 10% of the **Blocks**, rounding up.
4.Inodes: The fourth field is the total number of **inodes** in those **inode blocks**.

```
typedef unsigned char* bitmap_t;

struct fs_superblock {
    int magic;
    int nblocks;
    int ninodeblocks;
    int ninodes;
};
```

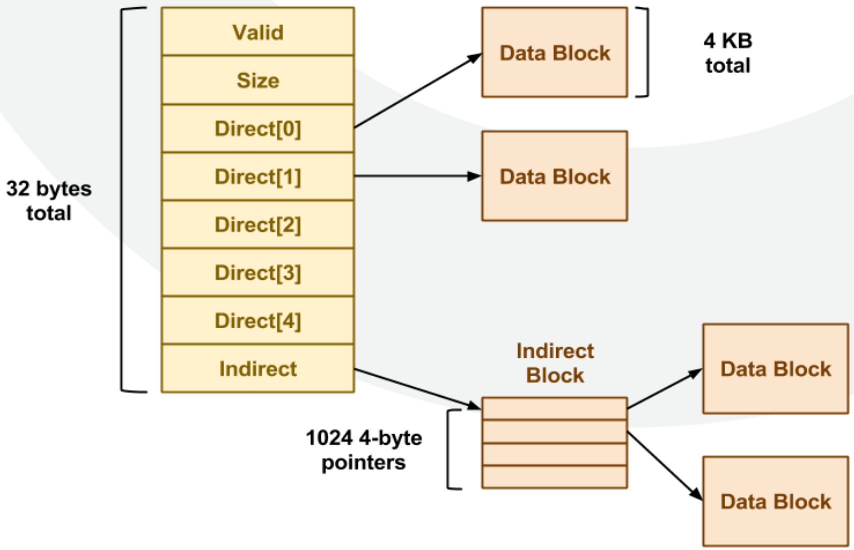


Summary of the Phase I report

- Each field of the **inode** is a 4-byte (32-bit) integer. The Valid field is 1 if the **inode** is valid (i.e. has been created) and is 0 otherwise. The Size field contains the logical size of the **inode** data in bytes. There are 5 direct pointers to data blocks, and one pointer to an **indirect** data block. In this context, "pointer" simply means the number of a block where data may be found. A value of 0 may be used to indicate a *null* block pointer. Each inode occupies 32 bytes, so there are 128 inodes in each 4KB inode block.

```
struct fs_inode {
    int isvalid;
    int size;
    int direct[DATA_POINTERS_PER_INODE];
    int indirect;
};

union fs_block {
    struct fs_superblock super;
    struct fs_inode
inodes[INODES_PER_BLOCK];
    int
pointers[DATA_POINTERS_PER_BLOCK];
char data[DISK_BLOCK_SIZE];
};
```

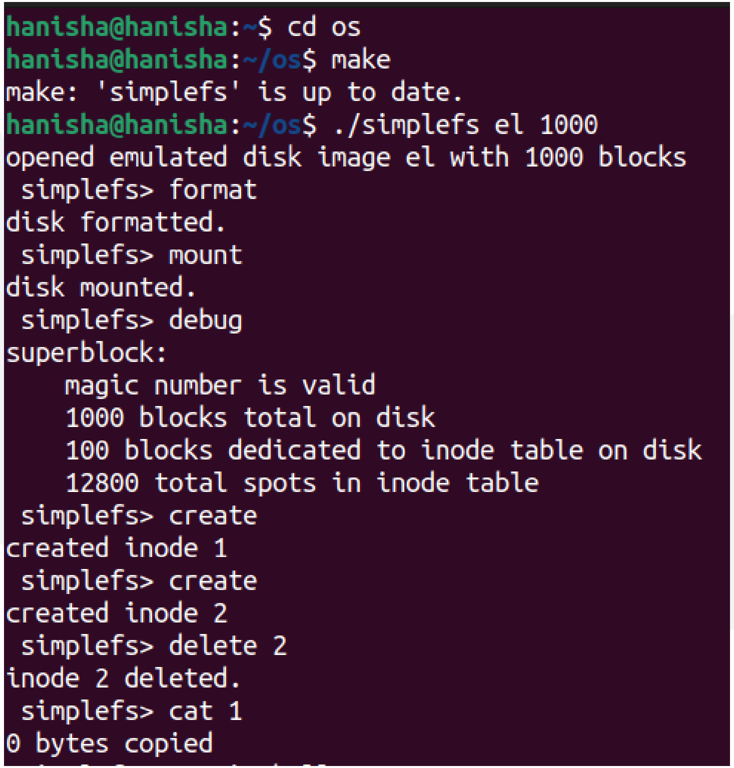


RV College of Engineering®

Go, change the world®

Implementation

1. *format*: Formats the disk to prepare it for use with SimpleFS. This command erases all existing data on the disk.
2. *mount*: Mounts the file system. This command prepares SimpleFS to interact with the disk.
3. *debug*: Displays debug information about the file system, including the superblock, inode information, and data block information.
4. *create*: Creates a new file in the file system and returns the inode number of the newly created file.
5. *delete*: Deletes a file from the file system given its inode number. delete <inode>
6. *cat*: Displays the contents of a file given its inode number. cat <inode>



```
hanisha@hanisha:~$ cd os
hanisha@hanisha:~/os$ make
make: 'simplefs' is up to date.
hanisha@hanisha:~/os$ ./simplefs el 1000
opened emulated disk image el with 1000 blocks
 simplefs> format
disk formatted.
 simplefs> mount
disk mounted.
 simplefs> debug
superblock:
    magic number is valid
    1000 blocks total on disk
    100 blocks dedicated to inode table on disk
    12800 total spots in inode table
 simplefs> create
created inode 1
 simplefs> create
created inode 2
 simplefs> delete 2
inode 2 deleted.
 simplefs> cat 1
0 bytes copied
```

Implementation

1. *copyin*: Copies a file from the host system into the file system, associating it with the specified inode number. copyin <filename> <inode>
2. *copyout*: Copies a file from the file system to the host system, given its inode number. copyout <inode> <filename>
3. *getsize*: Retrieves the size of a file given its inode number. getsize <inode>
4. *defrag*: Defragments the file system to optimize disk space and improve performance. defrag
5. *help*: Displays a list of available commands and their descriptions. help
6. *quit* or *exit*: Exits the SimpleFS shell and closes the disk. quit or exit

These are the commands you can use in the SimpleFS shell to perform various file system operations. You can type any of these commands followed by the required arguments to execute them. Additionally, you can use the help command to get a summary of available commands at any time.

```
 simplefs> copyin hello.txt 1
33 bytes copied
copied file hello.txt to inode 1
 simplefs> copyout 1 hello.txt
33 bytes copied
copied inode 1 to file hello.txt
 simplefs> getsize 1
inode 1 has size 33
 simplefs> defrag
disk defragged.
```

```
 simplefs> help
Commands are:
    format
    mount
    debug
    create
    delete  <inode>
    cat     <inode>
    copyin  <file> <inode>
    copyout <inode> <file>
    help
    quit
    exit
```

TOOLS/APIs USED

- Assembly Language: Low-level programming language for tasks like bootloader development and interacting with hardware.
- C Programming Language: Widely used for kernel development due to its efficiency and proximity to hardware.
- GNU Compiler Collection (GCC): Compiles code for the target architecture, generating executable binaries.
- Linker (LD): Links compiled code and libraries to create the final kernel image.
- Make: Automates the build process, managing dependencies and compiling source code efficiently

APPLICATIONS

1. **Filesystem Understanding**: SimpleFS can be used as a teaching tool to help students understand the fundamental concepts of filesystems, such as inodes, directory structures, file operations, and data storage.
2. **Operating System Courses**: In courses related to operating systems, SimpleFS can be used to demonstrate filesystem implementation principles, disk management, file I/O operations, and system calls related to filesystem interactions.
3. **Prototyping and Testing**: Developers can use SimpleFS as a starting point for prototyping new filesystem features or experimenting with different algorithms for disk layout, file allocation, directory organization, etc.
4. **Research and Experimentation**: Researchers in the field of filesystems or storage systems can use SimpleFS to conduct experiments, evaluate performance, and compare different filesystem designs in a controlled environment.
5. **Embedded Systems Development**: SimpleFS can be adapted for use in embedded systems where lightweight filesystems are required. It provides a simple yet functional filesystem implementation suitable for resource-constrained environments.
6. **Backup and Recovery Tools**: SimpleFS can be integrated into backup and recovery tools for creating disk images, copying files to and from disk images, and testing data recovery mechanisms.

REFERENCES (*As per IEEE format and must be Numbered consecutively in order of first mention*) & Annexures / Appendix

- **Time Administration of Virtual File System Operations**
Saloni Parekh;Arnav Deshpande;N. Narayanan Prasanth

- **A New Design of In-Memory File System Based on File Virtual Address Framework**
Edwin H.-M. Sha;Xianzhang Chen;Qingfeng Zhuge;Liang Shi;Weiwen Jiang
IEEE Transactions on ComputersYear: 2016 | Volume: 65, Issue: 10 |

- **Overtmpfs: A virtual memory file system based on tmpfs**
Hao Li;Yongping Xiong;Jian Ma
Proceedings of 2011 International Conference on Computer Science and Network Technology

- **Virtual file system for scalable media formats: Architecture proposal for managing and handling scalable media files**
Heiko Sparenberg;Alexander Schmitt;Robert Scheler;Siegfried Foessel;Karlheinz Brandenburg
2011 14th ITG Conference on Electronic Media Technology