MiniFS: A Technical Report on the Design and Implementation of a Miniature File System

0.0 Quick Start

1 Compile

`make`

2 Create Disk Image

- Linux: `./mkfs disk.img 102400`
- Windows: `mkfs.exe disk.img 102400`

3 Start Shell

- Linux: `./shell disk.img`
- Windows: `shell.exe disk.img`

Using test script:

- Linux: `./shell disk.img < fulltest`
- Windows: `shell.exe disk.img < fulltest`

4 View content

1. Start from 10240 byte

```
# Linux/Git Bash
xxd -c 16 -g 4 -s 10240 disk.img | less

# Windows PowerShell (view bytes from offset 10240)
$bytes = [System.IO.File]::ReadAllBytes("disk.img")
$bytes[10240..11263] | Format-Hex

# Or simply view the entire file
Format-Hex disk.img | more
```

1.0 Introduction

The MiniFS project was chartered to design and implement a functional, miniature file system from first principles. This report provides a comprehensive technical overview of the system's

architecture, its operational capabilities, key implementation challenges surmounted during development, and the collaborative effort that brought the project to fruition. The document is structured to cover the following key areas:

- Structure Design
- File Operation in File System
- Implementation Details
- Demonstration
- Contributions

The report first establishes the foundational on-disk architecture of MiniFS, which dictates all subsequent data management operations.

The source code of the project in on Github:
https://github.com/5teveLAN/os_minifs

2.0 Core Architecture of MiniFS

A well-defined on-disk structure is of strategic importance to any file system. This architecture serves as the blueprint for data organization, integrity, and overall functionality. The MiniFS design specifies a clear and logical layout for all control structures and user data, ensuring consistent and reliable operation.

1 The Virtual Disk Foundation

The base storage medium for MiniFS is a virtual disk image named `disk.img`. This file has a fixed size of 100 Kb and is logically segmented into 100 blocks, with each block uniformly sized at 1 Kb (1024 bytes).

2 The Volume Control Block (VCB)

The Volume Control Block (VCB) is the central metadata repository for the entire file system volume, located in the first block of the disk. It contains critical information about the volume's layout and configuration, analogous to a Superblock in the UNIX File System (UFS) or a Master File Table in NTFS. The specific structure of the MiniFS VCB is detailed below.

field	calculation	value
block_size		1024
block_count		100
fcb_size		32
bmap_start_block		1

field	calculation	value
bmap_end_block		1
fmap_start_block		2
fmap_end_block		2
fcb_start_block		3
fcb_end_block		9
fcb_count	(fcb_end_block - fcb_start_block + 1) * (block_size / fcb_size)	224
dbp_count		4
dentry_size		64
dentry_count	block_size * dbp_count / dentry_size	64

3 The File Control Block (FCB/inode)

The File Control Block (FCB), also known as an inode, is a data structure that stores the essential metadata for a single file or directory. This includes its unique identifier, size, and pointers to the data blocks that hold its content.

Field	Value/Description (Example)
Identifier Number	0-224 (from `fcb_count`)
File Size	0-4 KB (4096 bytes)
4 direct block pointer	Block n1, n2, n3, n4

4 File and Directory Representation

Following established UNIX principles, MiniFS treats a directory as a special type of file. The data blocks of a directory file contain a series of directory entries (Dentries), where each entry records a file name and its corresponding inode number. The system implementation is focused on managing plain text files.

The directory system is organized in a tree structure. To support this hierarchy, every directory contains two mandatory dentries:

- `.` : A reference to the directory itself.
- `..` : A reference to the parent directory (except in the root directory).

5 Metadata Management: Block and FCB Maps

To manage the allocation of resources, MiniFS utilizes two bitmaps:

- **Block Map**: Tracks the allocation status (used or free) of each of the 100 data blocks on the virtual disk.
- **FCB Map**: Tracks the allocation status of each of the 224 available FCBs (inodes).

6 Synthesized On-Disk Layout

Synthesizing the architectural components described above results in the following clear on-disk layout for the MiniFS volume.

Block(s)	Type	Rationale
Block 0	Superblock (VCB)	Volume-wide metadata.
Block 1	Free block	Tracks usage of the 100 blocks.
Block 2	Free inode map	Tracks usage of the 224 FCBs/inodes.
Block 3 to 10	**FCB / Inode Table**	**One block stores MANY FCBs.**
Block 11 to 99	Data Blocks	Stores the file and directory contents. (90/4=22 files)

With the static on-disk architecture defined, the subsequent section details the dynamic system operations that interact with this structure.

3.0 System Operations and User Interface

The static on-disk architecture is brought to life through a defined set of system operations. User interaction with the MiniFS file system is facilitated by a custom command shell, which provides a familiar interface for file and directory manipulation.

1 System Initialization via `mkfs` Utility

Before the MiniFS volume can be used, it must be formatted using the `mkfs` utility. This critical tool prepares the `disk.img` file by performing the following initialization tasks:

- Writing the Volume Control Block (VCB) with the volume's parameters to Block 0.
- Initializing the Block and FCB bitmaps, marking all resources as 'free'.
- Initializing the FCB Table, marking all FCB slots as 'unused'.
- Creating the root directory by allocating the first available FCB and its first data block.

2 The MiniFS Command Shell

A custom C program was developed to emulate a command-line shell. This program serves as the primary interface for users, allowing them to execute commands to create, access, and manage files and directories within the MiniFS volume.

3 Supported Shell Commands

The MiniFS shell supports a range of standard file system commands. The table below documents the available commands, their usage syntax, and their functional descriptions.

Command	Usage	Description
mkdir	`mkdir <path>`	Creates a new directory at the specified path.
touch	`touch <path>`	Creates a new empty file at the specified path.
ls	`ls [path]`	Lists the contents of a directory.
cd	`cd <path>`	Changes the current working directory to the specified path.
pwd	`pwd`	Prints the absolute path of the current working directory.
rm	`rm [-r | -f | -rf] <path>`	Removes files or directories. Options: `-r` for recursive, `-f` for force (no prompt), `-rf` for forced recursive.
cat	`cat <path>`	Displays the content of a file.
>>	`echo "..." >> <path>`	**Append**: Adds text to the end of the specified file.
>	`echo "..." > <path>`	**Override**: Replaces the entire content of the specified file with new text.
stat	`stat <path>`	Displays inode information (size, inode number, blocks used).
exit	`exit`	Terminates the current shell session or program.

Executing these high-level commands required solving several low-level programming challenges, which are detailed in the following section.

4.0 Key Implementation Details

This section delves into the specific programming techniques and technical challenges that were addressed to translate the file system's abstract design into functional C code.

1 Core Data Structures and Functions

To represent the VCB, FCB, and Dentry in memory, C `structs` were defined. To guarantee data type consistency and portability across different compiler environments, fixed-size integer types such as `uint32_t` were mandated for all on-disk structures, with memory alignment controlled via `#pragma pack`. A key operational detail is the loading of the VCB and the entire FCB table into memory upon program initialization to accelerate metadata access. This strategy

was complemented by the creation of common, reusable functions to handle core operations, which simplified development and helped avoid code duplication.

2 Handling Endianness

A significant technical challenge was managing the byte order (endianness) when writing data from memory to the disk image. Most modern CPUs use a Little-Endian byte order, while network protocols and human-readable binary dumps (e.g., via `xxd`) often use Big-Endian. To ensure data written to the disk was in a consistent, network-standard Big-Endian format, the implementation leverages the `ntohl()` and `htonl()` functions, available via `winsock2.h` on Windows and `<arpa/inet.h>` on UNIX-like systems, to convert 32-bit integers between host and network byte order.

3 Command and Path Resolution

User input from the shell is received as a single string. To parse this input, the C library function `strtok()` was employed. It was used to split the command string into tokens based on spaces (e.g., `rm -r /dir1/file`) and to resolve file paths by splitting them into components based on the `/` delimiter.

4 Dentry Management and Traversal

Locating a specific file or subdirectory within a parent directory requires a linear search through all dentries stored in the parent's data blocks. Initial logic for this was encapsulated in the `find_file()` helper function. As development progressed, a newer function, `find_file_name_by_id()`, was introduced to simplify the lookup logic. While this traversal approach is functional, it could become a performance bottleneck in directories with a large number of entries; a potential future improvement would be to implement a memory buffer or cache to accelerate lookups.

The successful implementation of these technical details was the result of a coordinated team effort, as outlined next.

5 Demonstration

```
ggd@ggd-pc:~/Desktop/os_minifs$ make
gcc -Wall -g -c minifs_ops.c -o minifs_ops.o
gcc -Wall -g mkfs.c minifs_ops.o  -o mkfs
gcc -Wall -g shell.c minifs_ops.o  -o shell
ggd@ggd-pc:~/Desktop/os_minifs$ ./mkfs disk.img 102400

Successfully created 100 KB disk image.
ggd@ggd-pc:~/Desktop/os_minifs$ ./shell disk.img < fulltest
minishell > # 1. 基本目錄操作
minishell > mkdir /system
minishell > mkdir /user
minishell > ls /
Contents of directory (ID: 0):
ID       Name
--------------------------------------------------------------------
0        .
1        system
2        user
--------------------------------------------------------------------
Total: 3 entries
minishell > pwd
/
minishell >
minishell > # 2. 深度路徑測試 (測試 resolve_path 的遞迴能力)
minishell > mkdir /user/documents
minishell > mkdir /user/documents/lab
minishell > cd /user/documents/lab
Changed to directory 4
minishell > pwd
/user/documents/lab
minishell > ls ..
Contents of directory (ID: 3):
ID       Name
--------------------------------------------------------------------
3        .
2        ..
4        lab
--------------------------------------------------------------------
Total: 3 entries
```

```
minishell > # 3. 檔案建立與狀態檢查
minishell > touch /user/documents/notes.txt
minishell > stat /user/documents/notes.txt
======= FCB Structure Info (ID: 5) =======
Type:       File
File Size:  0 bytes
Data Blocks:
  [DBP 0]:   31
  [DBP 1]:   32
  [DBP 2]:   33
  [DBP 3]:   34
==========================================
minishell > ls /user/documents
Contents of directory (ID: 3):
ID       Name
--------------------------------------------------------------------
3        .
2        ..
4        lab
5        notes.txt
--------------------------------------------------------------------
Total: 4 entries
minishell >
minishell > # 4. 內容寫入與覆蓋 (測試 > 運算子)
minishell > echo "First line of data" > /user/documents/notes.txt
Wrote to notes.txt
minishell > cat /user/documents/notes.txt
First line of data
minishell >
minishell > # 5. 內容追加 (測試 >> 運算子)
minishell > echo "Second line appended" >> /user/documents/notes.txt
Appended to notes.txt
minishell > cat /user/documents/notes.txt
First line of data
Second line appended
minishell >
minishell > # 6. 覆蓋測試 (確認原本內容被清空)
minishell > echo "Clean start" > /user/documents/notes.txt
Wrote to notes.txt
minishell > cat /user/documents/notes.txt
```

```
minishell > # 7. 邊界測試：在不同目錄下建立同名檔案
minishell > mkdir /system/bin
minishell > touch /system/bin/notes.txt
minishell > ls /system/bin
Contents of directory (ID: 6):
ID       Name
--------------------------------------------------------------------
6        .
1        ..
7        notes.txt
--------------------------------------------------------------------
Total: 3 entries
minishell > ls /user/documents
Contents of directory (ID: 3):
ID       Name
--------------------------------------------------------------------
3        .
2        ..
4        lab
5        notes.txt
--------------------------------------------------------------------
Total: 4 entries
minishell >
minishell > # 8. 刪除測試 (rm)
minishell > # 先刪除單一檔案
minishell > rm /system/bin/notes.txt
Removing id=7 name=notes.txt
minishell > ls /system/bin
Contents of directory (ID: 6):
ID       Name
--------------------------------------------------------------------
6        .
1        ..
--------------------------------------------------------------------
Total: 2 entries
minishell >
minishell > # 9. 遞迴刪除測試 (rm -rf)
minishell > # 刪除包含內容的目錄
minishell > rm -rf /user
Removing id=2 name=user
minishell > ls /
Contents of directory (ID: 0):
ID       Name
--------------------------------------------------------------------
0        .
1        system
--------------------------------------------------------------------
Total: 2 entries
minishell >
minishell > # 10. 回到根目錄並最終檢查
minishell > cd /
Changed to directory 0
minishell > ls
Contents of directory (ID: 0):
ID       Name
--------------------------------------------------------------------
0        .
1        system
--------------------------------------------------------------------
Total: 2 entries
minishell > exit
ggd@ggd-pc:~/Desktop/os_minifs$
```

6.0 Project Contributions and Task Allocation

The MiniFS project was a collaborative effort. This section details the division of labor and task allocation among the team members who contributed to its successful completion.

1 Team Members and Roles

- **蕭宏均 陳泰頤** : 重點開發主力
- **鄒昱宸 李冠緯 黃婧絜** : 協力
- **陳裕勛** : 完全沒出現，寄信不回上課也找不到人

2 Detailed Task Breakdown

1. 虛擬磁碟結構設計，完成 mkfs (✅:完成 ☑️:分配中 ❌:未完成)

Items	黃婧絜	鄒昱宸	蕭宏均	陳泰頤	李冠緯	Description
Design Disk Structure			✅			
mkfs mkVCB()	✅	✅				
mkfs mkBitmap()	✅	✅				
mkfs mkFCB()			✅			
mkfs mkRoot()					✅	Create first FCB to root dirctory.
mkfs.exe			✅			Merge these functions to an executable file for initializing the file system.

2. 實作 inode 與檔案讀寫，完成 touch / append / cat

Items	黃婧絜	鄒昱宸	蕭宏均	陳泰頤	Description
touch			✅		1. id = mkFCB() 2. add dentry(file name and id) to root directory(block10)
append and override	✅	✅		✅	Append text to the file. 1. find the file name in the directory and get its inode id 2. get the DBPs from the inode 3. write content to these blocks
cat	✅	✅		✅	Show the content of the file.

3. 實作目錄與路徑解析，完成 mkdir / ls / stat / rm / cd / pwd

Items	蕭宏均	陳泰頤	Description
mkdir	✅		
ls		✅	
pwd	✅		
rm		✅	
cd		✅	
stat	✅		
path resolve	✅	✅	
shell	✅		

4. 整合成 shell，設計測試流程，繳交程式與說明文件

Items	蕭宏均	陳泰頤	Description
Documentation	✅		
Presentation	✅		
Demo		✅	
Test Script		✅	