

Project 3 Final Technical Report

Andy Lim & Ifunanya Okafor

Department of Computer Science, College of ECST, Cal State LA

CS 4440: Introduction to Operating Systems

Dr. Zilong Ye

03 November 2025

Table of Contents

Introduction.....	3
Part 1: User's Manual.....	4
Overall Build - Two Options.....	5
Question 1: Basic Client-Server.....	6
Question 2: Directory Listing Server.....	7
Question 3: Basic Disk Storage System.....	8
Question 4: File System Server.....	9
Question 5: Directory Structure.....	10
Part 2: Technical Manual.....	11
Disk Design.....	12
File System Design.....	14
Directory Support.....	17
Concurrency + Networking.....	19
Testing Tips.....	20
References.....	21

Introduction

This project is a culmination of our work in CS 4440: Operating Systems; from the basic principles in network programming to the working, network-accessible filesystem we have today. We completed the five guidelines as outlined:

1. A basic, multi-threaded client-server that reverses strings and is also resistant against denial-of-service attacks (DoS).
2. An extension of the client-server that provides directory/file listing, layering a remote ls service over sockets.
3. An extension of the client-server that upgrades it to a simulated physical disk-storage system (addressed sectorially and cyclically in 128 bytes) by exposing a “block device” over TCP
4. An extension of the client-server that builds a flat filesystem on the “block device”
5. An extension of the client-server that extends the filesystem with a hierarchical directory structure (MKDIR, CD, PWD, and RMDIR)

Note: All components are written in C using POSIX sockets and pthreads.

Part 1: User's Manual

Overall Build - Two Options

Option 1: Using the Makefile provided in zip file



Option 2: Manually compiles files (see below)

A screenshot of a terminal window titled "bash" containing a series of manual compilation commands. The commands involve using gcc with various flags (-O2, -std=c17, -Wall, -Wextra, -pedantic) and pthreads to compile source files into executables. The commands are organized into five sections, each starting with a comment like "# Q1", "# Q2", "# Q3", "# Q4", or "# Q5".

```
# Q1
gcc -O2 -std=c17 -Wall -Wextra -pedantic -pthread server.c -o q1_server
gcc -O2 -std=c17 -Wall -Wextra -pedantic           client.c -o q1_client

# Q2
gcc -O2 -std=c17 -Wall -Wextra -pedantic -pthread ls_server.c -o q2_ls_server
gcc -O2 -std=c17 -Wall -Wextra -pedantic           ls_client.c -o q2_ls_client

# Q3
gcc -O2 -std=c17 -Wall -Wextra -pedantic -pthread disk_server.c   -o disk_server
gcc -O2 -std=c17 -Wall -Wextra -pedantic           command_client.c -o
disk_client_cli
gcc -O2 -std=c17 -Wall -Wextra -pedantic           random_client.c -o
disk_client_rand

# Q4
gcc -O2 -std=c17 -Wall -Wextra -pedantic -pthread file_system_server.c -o fs_server
gcc -O2 -std=c17 -Wall -Wextra -pedantic           file_system_client.c -o fs_client

# Q5
gcc -O2 -std=c17 -Wall -Wextra -pedantic -pthread "file_system_server+directory.c" -
o fs_server_dirs
gcc -O2 -std=c17 -Wall -Wextra -pedantic           file_system_directory_client.c - -
o fs_client_dirs
```

Question 1: Basic Client-Server

Function:

- Server listens on TCP port 8080, spawns a thread per connection, reads one string, reverses it, and replies.
- Client connects to server-ip 8080, sends a single argument string, prints the reversed response.

How to Run:



```
# Terminal 1
./server
# -> "Server listening on port 8080"

# Terminal 2
./client 127.0.0.1 "hello world"
# -> prints IP, sent string, and the reversed response
```

Additional Notes:

- Server prints each connection and uses a mutex for clean console output
- Client usage: ./q1_client <server-ip> <string>; validates args, errors gracefully on bad address/connection

Question 2: Directory Listing Server

Function:

- Server accepts a connection on TCP port 8083, executes ls with flags provided by the client, and then streams the output back.
- Client connects to <server-ip>:8083 and sends the space-joined ls parameters, then prints the response

How to Run:



```
# Terminal 1
./ls_server
# -> server listens on port 8083

# Terminal 2
./ls_client 127.0.0.1 -lh ~
./ls_client 127.0.0.1 -la /tmp
```

Additional Notes:

- Ensure the server process has permission to exec ls and read target directories.
- Client usage: ./q2_ls_client <server-ip> [ls options...]. It connects and sends exactly the options/paths you type; output is printed verbatim.

Question 3: Basic Disk Storage System

Function:

- I = server replies with <cylinders> <sectors>\n
- R c s = server returns 1<128 bytes> on success, 0 if out of range
- W c s l <data> with $0 \leq l \leq 128$ = returns 1 on a valid write, 0 if otherwise
- Seek time simulated by a track delay (μ s). Storage is a regular file (aka image).

How to Run:

```

bash

# Terminal 1
./disk_server 9090 4 8 100 ./disk.img --sync=after
#     ^port  ^cyl ^sec ^track_us    ^image

# Terminal 2 (CLI tester)
./command_client 127.0.0.1 9090
I
R 0 0
W 0 0 5
hello
R 0 0
exit

# Random workload (N=50 ops, seed=42)
./random_client 127.0.0.1 9090 50 42

```

Additional Notes:

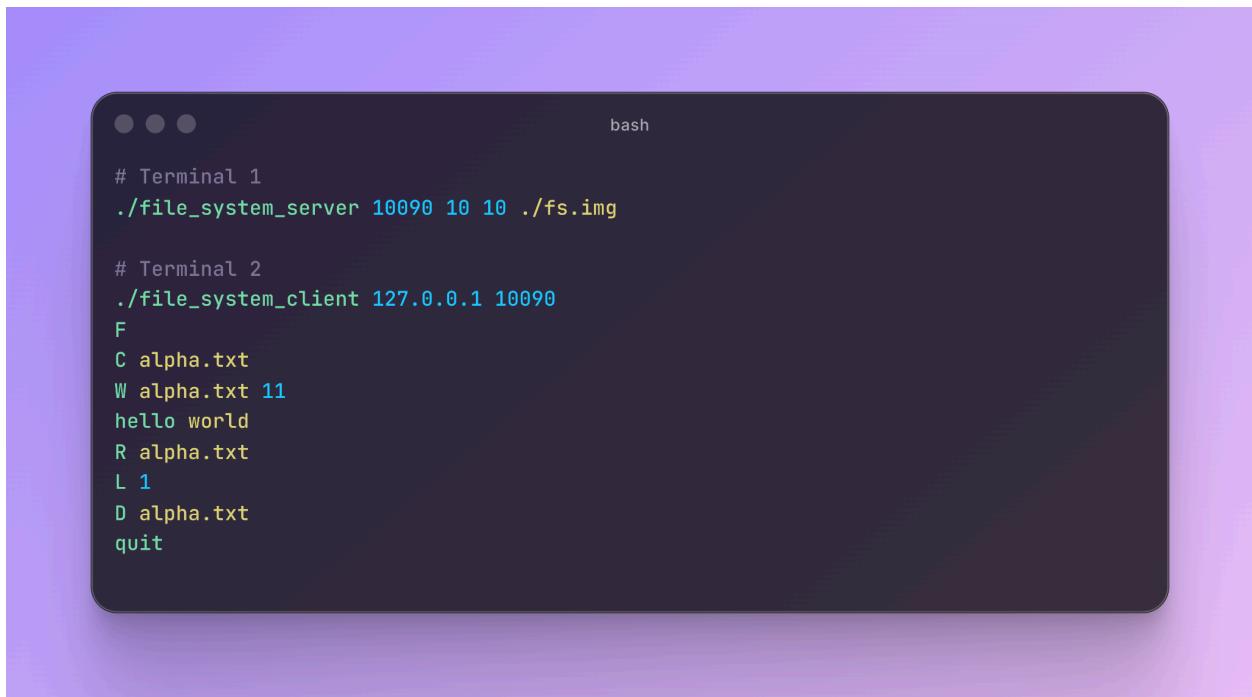
- Out-of-range cylinders/sectors and invalid lengths (ex: L=129) are rejected.
- Random client prints a compact progress stream (R/W characters).

Question 4: File System Server

Function:

- F = format; initializes superblock, FAT, and directory table in the backing image.
- C f = create; 0 ok, 1 exists, 2 errors.
- D f = delete; 0 ok, 1 not found, 2 errors.
- L b = list names (b=0) or name size (b=1), ends with a blank line.
- R f = read; replies code len <raw-bytes>.
- W f1 <bytes> = overwrite file with 1 bytes.

How to Run:



```
bash

# Terminal 1
./file_system_server 10090 10 10 ./fs.img

# Terminal 2
./file_system_client 127.0.0.1 10090
F
C alpha.txt
W alpha.txt 11
hello world
R alpha.txt
L 1
D alpha.txt
quit
```

Additional Notes:

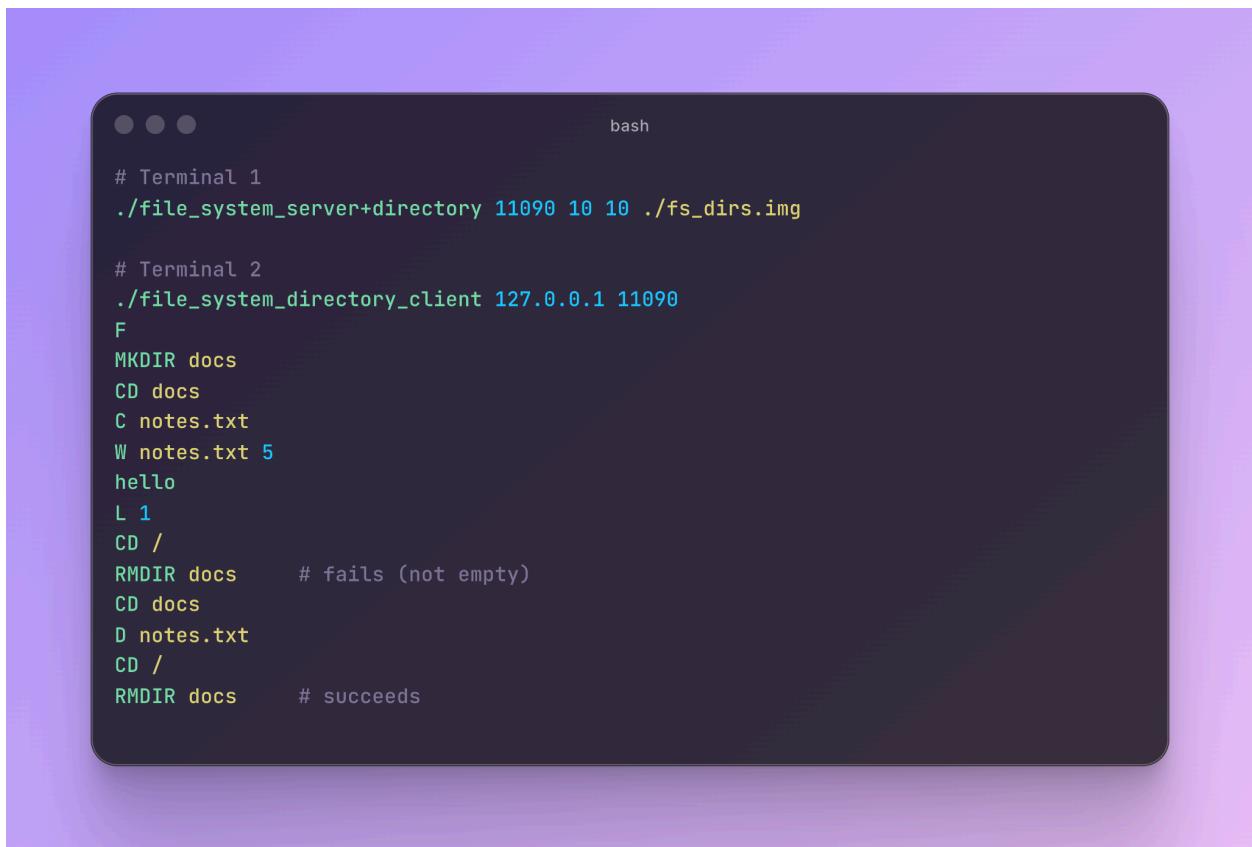
- N/A

Question 5: Directory Structure

Function:

- MKDIR name = 0/1/2 (ok/exist/error)
- CD name|...| / = 0 ok, 1 not found, 2 not a directory
- PWD = absolute path + newline
- RMDIR name = 0 ok, 1 not found, 2 not empty/other
- L now lists current directory (types shown when b=1: d for dirs, - for files)

How to Run:



```
bash

# Terminal 1
./file_system_server+directory 11090 10 10 ./fs_dirs.img

# Terminal 2
./file_system_directory_client 127.0.0.1 11090
F
MKDIR docs
CD docs
C notes.txt
W notes.txt 5
hello
L 1
CD /
RMDIR docs      # fails (not empty)
CD docs
D notes.txt
CD /
RMDIR docs      # succeeds
```

Additional Notes:

- N/A

Part B: Technical Manual

Disk Design

Geometry + Block Size

- The disk is a rectangular grid: cylinders \times sectorsPerCylinder. Each sector is 128 bytes.
- The server persists to an image file and simulates track-to-track latency via a sleep parameter (μs).

Protocol

- I = ASCII: <cyl> <sec>\n.
- R c s = returns '1' followed by 128 raw bytes on success; '0' if out of range.
- W c s l data = l $\in [0,128]$. Returns '1' for valid c,s,l (data stored, and any unwritten tail bytes of the 128-byte sector may be left as-is or zero-filled), '0' otherwise.

I/O Model

- Sector offset = (c * sectors + s) * 128. Pread/pwrite (or seek+read/write) to the image file.
- Optional mmap is acceptable in our disk; correctness is independent of the API.

Concurrency

- Thread per client
- Each request is atomic at sector granularity; the server serializes conflicting writes

Error Handling

- Parameter validation: $0 \leq c < \text{cylinders}$, $0 \leq s < \text{sectors}$, $0 \leq l \leq 128$.
- Connection errors close that session without affecting others.

File System Design

Block Size + Addressing

- Uses the same 128-byte block size. Blocks are addressed by absolute block index.
- Layout (Note: The indices are block indices):

```
[0] Superblock
[1 .. fat_start+fat_blocks-1] FAT (int32 entries; one per block)
[dir_start .. dir_start+dir_blocks-1] Directory entries (fixed-size)
[data_start .. end] File data blocks (payload)
```

Superblock

- magic ('FSL1'), cylinders, sectors, block_size, total_blocks
- fat_start, fat_blocks, dir_start, dir_blocks, data_start
- max_files (derived from directory region size)
- All fields are 32-bit except the padding since the server initializes and validates these.

FAT

- Values: FREE = -1, EOC = -2, otherwise next block index in the chain.
- First-fit allocation by scanning for free entries; chains are singly linked.

Directory Entries

- used (1B), padding
- first_block (int32 head of FAT chain, or -1 if empty)
- size_bytes (uint32)
- name[48] (NUL-terminated, truncated if needed)
- Two entries fit in one 128-byte block; the directory region has a fixed capacity.

Main Algorithms

- Format: compute FAT/dir region sizes; mark metadata blocks as reserved (EOC), clear FAT and dir.
- Create: ensure unique name; claim a free directory slot; first_block = -1, size = 0.
- Delete: walk the FAT chain (if any at all) and mark all blocks FREE; clear the directory entry.
- Write W f1 data:
 1. Compute required blocks ceil (l/128)
 2. Grow/shrink FAT chain via ensure_capacity (extend or free tail)
 3. Copy bytes block-by-block; zero any partial tail
- Append A f1 data: grow chain to fit oldSize + 1; write starting at tail offset (handle partial first block, then whole blocks).
- Read R f: iterate chain, copying up to size_bytes.
- List L b: iterate directory table; skip used == 0.
- Thread Safety: one global FS mutex guarding metadata and data accesses.

Limits + Complexity

- File count limited by `dir_blocks * (128 / 64)`. With 8 dir blocks: 16 entries.
- Allocation and deallocation are $O(\text{totalBlocks})$ worst-case scans of the FAT; average case is small.

Directory Support

Extended Directory Entry

Add two fields to the previous entry to support a tree and per-connection CWD:

- `is_dir` (bool/byte)
- `parent` (int32, index of parent directory entry; root has parent = -1)

Your hierarchical build also bumps the on-disk magic to distinguish formats.

Root and CWD

- Root directory is entry 0, name '/', `is_dir` = 1, parent = -1.
- Each connection tracks its own CWD index (defaults to root). All file operations are interpreted relative to that CWD.

New commands

- `MKDIR name` = create a child directory of CWD; reject names containing / ; return 1 if exists.
- `CD name|..| /` = update CWD to child, parent, or root; reject non-directory targets.
- `PWD` = compute absolute path by walking parent pointers to root and joining names.
- `RMDIR name` = only succeeds for empty directories (no entries with parent == `thatDir`).

Main Algorithms

- Lookup in CWD: linear scan of directory table for used `&& parent == cwd && name == input...`
- Emptiness Check: scan for any used `&& parent == target`.

- Path build (PWD): collect component names following parent pointers to root, then print in reverse.
- Listing (L): filter by parent == cwd; with b = 1, prefix each line with type (d or -) and include size for files.

Error Handling

- Return codes mirror Question 4 (0/1/2), with 1 meaning “not found” or “name conflict,” and 2 for “not directory,” “not empty,” or no space.
- Names are sanitized to forbid / so that path resolution is unambiguous within this flat table implementation.

Concurrency + Networking

- All servers use TCP, accept() loop, and spawn a thread per request/connection; Question 1's multi-threading is explicit in server.c.
- Each client is a simple single-connection program for test/interaction. Question 1's client shows straightforward connect/send/recv.
- For Q3–Q5, a single global mutex around FS/disk state is sufficient for correctness.

Testing Tips

Quick Routine Checks:

```
bash

# Q3
./disk_client_cli 127.0.0.1 9090 <<EOF
I
W 0 0 5
hello
R 0 0
W 0 0 129
exit
EOF

# Q4
./fs_client 127.0.0.1 10090 <<EOF
F
C a
W a 5
hello
R a
D a
R a
quit
EOF

# Q5
./fs_client_dirs 127.0.0.1 11090 <<EOF
F
MKDIR d
CD d
C n
W n 3
hey
CD /
RMDIR d
CD d
D n
CD /
RMDIR d
quit
EOF
```

References

- Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5), 29–43. <https://doi.org/10.1145/945445.945450>
- Ye, Z. (2025). *Intro to Operating Systems CS 4440-03 (92232)*. [https://calstatela.instructure.com/courses/119862]. Canvas, Cal State LA.
<https://calstatela.instructure.com/>