Project 3 Report, Team 3

Design and Implementation of Secondary Storage System

Joshua Encinas, Cindrella Shine, and Emily Weimer

CS 4440, Introduction to Operating Systems

November 22, 2025

Table of Contents

User Manual

This section explains how to build and run the programs for Problems 1 through 5 of the project.

Problem 1: Basic Client-Server

1.1 Overview

The system consists of two programs:

• server: A multi-threaded TCP server that accepts connections on a specified port, receives a line of text from each client, reverses the characters in the line, and sends the reversed line back to the client.

• client: A TCP client that connects to the server, sends a line of text constructed from its command-line arguments, and prints the reversed string received from the server.

1.2 Build Instructions

The programs are built using the provided Makefile. From the project directory:

1. Ensure server.c, client.c, and the Makefile are present.
2. Run the command: make

   This produces two executables: server and client.

1.3 Running the Server

Usage:

```
./server <port>
```

• <port> must be a positive integer (e.g., 5555).

Example:

```
./server 5555
```

Behavior:

• The server listens on the given port for incoming TCP connections.

• For every accepted client connection, the server creates a new detached thread.

• Each thread handles exactly one request on its connection, then closes the socket.

1.4 Running the Client

Usage:

```
./client <host> <port> <string ...>
```

Arguments:

• <host> – IPv4 address of the server (e.g., 127.0.0.1).

• <port> – Port number on which the server is listening.

• <string ...> – One or more words to be sent to the server. All remaining arguments are concatenated with spaces into a single line.

Example:

```
./client 127.0.0.1 5555 hello world from client
```

In this example, the client sends the string "hello world from client" to the server. The server responds with the reversed string, which the client prints to standard output.

1.5 Expected Output

Given the example above, with a running server, the client should print:

tneilc morf dlrow olleh

1.6 Error Handling and Exit Behavior

Both programs validate their command-line arguments and system call results. If a fatal error occurs, the program prints an error message and exits with a non-zero status.

Client-side errors include:

- Missing or too few arguments – prints a usage message and exits with status 1.
- Invalid port (<= 0) – prints "Invalid port" and exits with status 1.
- Failed socket(), inet_pton(), connect(), send(), or recv() – uses perror() to print an informative error message and exits with status 1.

Server-side errors include:

- Invalid or missing port – prints a usage or "Invalid port" message and exits.
- Failed socket(), setsockopt(), bind(), listen(), or malloc() – uses perror() and exits.
- Failed accept() – prints an error message and continues to accept new connections.
- Failed recv() or send() in a worker thread – prints an error message (for r < 0 or s < 0) and closes the client connection.

In all cases, sockets are closed before exit, so both programs terminate gracefully.

1.7 Testing Scripts and Typescripts

Two Bash scripts are provided to test the programs:

- test_server.sh – exercises server behavior, including invalid arguments, normal requests, a large input line, and multiple concurrent clients.
- test_client.sh – exercises client behavior, including invalid arguments, connection failures, and normal requests.

The script utility is used to capture terminal sessions into typescript files:

- server_tests.typescript – record of running test_server.sh.
- client_tests.typescript – record of running test_client.sh.

These logs demonstrate that the programs handle all tested inputs and error cases correctly.

Problem 2: Directory Listing Server

2.1 Overview

This problem extends the basic client-server system to provide a directory listing service. Instead of reversing a string, the server executes the standard ls command and returns its output to the client over a TCP connection.

The programs are:
• ls_server: A TCP server that forks a child process for each client connection. The child reads a single line of text specifying ls arguments, redirects its standard output and standard error to the client socket, and then execs ls with the requested arguments.
• ls_client: A TCP client that connects to ls_server, sends a line containing ls-style arguments, and prints everything the server sends back (both normal ls output and error messages).

2.2 Build Instructions

The same Makefile used for Problem 1 also builds the Problem 2 programs. From the project directory:
1. Ensure ls_server.c, ls_client.c, and the Makefile are present.
2. Run the command: make
   This produces two additional executables: ls_server and ls_client.

2.3 Running ls_server

Usage:
   ./ls_server <port>

• <port> must be a positive integer (for example, 5560).

Example:
   ./ls_server 5560

Behavior:
• The server listens on the given port for incoming TCP connections.
• For every accepted client connection, the server forks a child process.
• The child process closes the listening socket, reads a single line of text from the client, interprets it as the argument string for ls, redirects its standard output and standard error to the client socket using dup2, and calls execvp("ls", ...).
• The parent process closes the connected client socket and continues to accept new connections.
• A SIGCHLD handler is installed so that terminated child processes are reaped promptly and do not remain as zombies.

2.4 Running ls_client

Usage:
   ./ls_client <host> <port> <ls-args...>

Arguments:
• <host> – IPv4 address of the server (for example, 127.0.0.1).
• <port> – Port number on which ls_server is listening.
• <ls-args...> – One or more arguments that will be passed to ls. These may include options (such as -l or -a) and pathnames (such as . or /tmp).

Example (basic listing):
./ls_client 127.0.0.1 5560 .

Example (long listing with options):
./ls_client 127.0.0.1 5560 -l -a .

In each case, the client sends a single line containing the ls arguments to the server, then receives and prints all of the output produced by ls on the server side.

2.5 Expected Output

For the command:
./ls_client 127.0.0.1 5560 .

the client will display the same directory listing that would appear if you ran:
ls .

directly in the shell on the server machine. For the command:
./ls_client 127.0.0.1 5560 -l .

the output will match a local ls -l . (long listing) of the directory the server is running in.
If an invalid option or nonexistent path is provided, ls prints its usual error message, which the client also displays.

2.6 Error Handling and Exit Behavior

Both ls_server and ls_client validate their command-line arguments and check system call return values. If a fatal error occurs, the program prints an error message and exits with a non-zero status.

ls_client error cases include:
• Missing or too few arguments – prints a usage message and exits with status 1.
• Invalid port (<= 0) – prints "Invalid port" and exits with status 1.
• Failed socket(), inet_pton(), connect(), send(), or recv() – uses perror() to print an informative error message and exits with status 1.

ls_server error cases include:
• Missing or invalid port – prints a usage or "Invalid port" message and exits.
• Failed socket(), setsockopt(), bind(), or listen() – uses perror() via a fatal() helper and

exits.
• Failed accept() – prints an error and continues accepting new connections.
• Failed recv(), dup2(), or execvp() in a child process – prints an error message and exits that child with a non-zero status.

In all cases, open sockets are closed before exit, so both programs terminate cleanly.

2.7 Testing Scripts and Typescripts

Two Bash scripts are used to demonstrate correct behavior of the Problem 2 programs:
• test_ls_server.sh – compiles ls_server and ls_client, then exercises ls_server with invalid ports, simple listings, long listings, multiple paths, an invalid option, and several concurrent clients.
• test_ls_client.sh – compiles ls_server and ls_client, then focuses on ls_client behavior, including argument validation, connection failures, and valid requests with different ls arguments.

The script utility is used to capture terminal sessions:
• ls_server_tests.typescript – record of running test_ls_server.sh.
• ls_client_tests.typescript – record of running test_ls_client.sh.

These logs show that ls_server and ls_client handle valid and invalid inputs correctly and exit gracefully.

Problem 3: Basic Disk-Storage System

3.1 Overview

Problem 3 extends the earlier client-server work to implement a simple simulated disk service.

The system consists of three programs:

• disk_server: A TCP server that simulates a disk with a fixed geometry (cylinders, sectors, and block size).
It stores its data in a backing file using mmap() and implements commands to read and write 128-byte blocks.

• disk_cli: An interactive command-line client that connects to disk_server, sends disk commands (I, R, W),
and displays the responses.

• disk_rand: A workload generator that connects to disk_server and issues a sequence of random read and write
requests, printing only a compact progress indicator.

3.2 Build Instructions

The same Makefile used for the earlier problems also builds the disk programs. From the project directory:

1. Ensure disk_server.c, disk_cli.c, disk_rand.c, and the Makefile are present.
2. Run the command: make
This produces three additional executables: disk_server, disk_cli, and disk_rand.

3.3 Running disk_server

Usage:

./disk_server <port> <cylinders> <sectors> <track_us> <backing_file>

Arguments:

• <port> – TCP port number on which the server listens.
• <cylinders> – Number of cylinders in the simulated disk (must be > 0).
• <sectors> – Number of sectors per cylinder (must be > 0).
• <track_us> – Track-to-track seek time in microseconds.
• <backing_file> – Path to the file used to store the disk image. This file is created or resized as needed to hold <cylinders> × <sectors> × 128 bytes.

Example:

./disk_server 5570 10 20 1000 disk.img

Behavior:

• The server opens or creates the backing file and uses ftruncate() and mmap() to map the entire disk
image into memory.

• It then creates a TCP listening socket and waits for client connections.
• For each accepted client, the server creates a new thread that handles all requests from that client.
• All threads share a single simulated disk arm position; a mutex is used so that seeks are serialized
and the configured track-to-track delay is applied consistently.

3.4 Running disk_cli

Usage:
./disk_cli <host> <port>

Arguments:
• <host> – IPv4 address of the disk_server (for example, 127.0.0.1).
• <port> – TCP port on which disk_server is listening.

Once started, disk_cli reads commands from standard input and sends them to the server.
Supported commands are:
• I
Asks the server for its geometry. The server replies with a line containing two integers:
<cylinders> <sectors>
disk_cli prints this line directly.

• R c s
Requests that the block at cylinder c, sector s be read. The server replies with a single character code
followed by up to 128 bytes:
- '0' (zero) indicates that the requested block is invalid (out of range).
- '1' indicates that the block is valid and is followed by 128 data bytes.
disk_cli prints a summary: it displays '1' and a short hex dump of the beginning of the block.

• W c s l
Requests a write of l bytes to the block at cylinder c, sector s. After the command line, disk_cli reads
exactly l raw bytes from stdin and sends them to the server. The server replies with a single character:
- '1' for a successful write.
- '0' if the cylinder/sector or length is invalid.
disk_cli prints this status character.

3.5 Running disk_rand

Usage:
./disk_rand <host> <port> <N> <seed>

Arguments:
• <host> – IPv4 address of the disk_server (for example, 127.0.0.1).
• <port> – TCP port on which disk_server is listening.
• <N> – Number of random requests to send.
• <seed> – Seed for the pseudo-random number generator so that runs are repeatable.

Behavior:
• disk_rand first sends an I command to discover the disk geometry, then parses the returned numbers.
• It then issues N random requests. Each request is randomly chosen to be either a read (R) or a write (W).
• Cylinders and sectors are chosen uniformly from the valid ranges.
• Writes always send a full 128-byte block of random data.
• For each request, disk_rand prints a single character ('r' for read, 'w' for write) so that progress
is visible without printing the actual block contents.

3.6 Error Handling and Exit Behavior

All three programs check their command-line arguments and important system calls and exit with a non-zero status
if a fatal error occurs.

disk_server:
• Validates that the number of cylinders and sectors is greater than zero.
• Checks the results of open(), ftruncate(), mmap(), socket(), bind(), listen(), and pthread_create().
Failures are reported with perror(), and the server exits with a non-zero status.
• Uses a SIGINT handler to allow clean shutdown when interrupted.

disk_cli:
• Validates its argument count and ensures the port number is positive.
• Checks socket(), inet_pton(), connect(), and uses helper functions that check the results of send() and recv().
• On protocol or I/O errors, it prints an error message and terminates cleanly.

disk_rand:
• Validates its argument count and ensures both the port and N are positive.
• Checks socket(), inet_pton(), connect(), and all network I/O.
• If any fatal error occurs (for example, connection failure or unexpected EOF), it reports the issue and exits
with a non-zero status.

3.7 Testing Scripts and Typescripts

Three shell scripts are used to test the Problem 3 programs:

• test_disk_server.sh – Compiles disk_server, disk_cli, and disk_rand. It runs negative tests on disk_server
(such as missing or invalid geometry arguments) and functional tests using disk_cli and disk_rand that exercise
normal reads and writes as well as out-of-range accesses.

• test_disk_cli.sh – Compiles disk_server and disk_cli. It focuses on disk_cli behavior, including invalid
arguments, connection failures when the server is not running, and valid sequences of I, W, and R commands.

• test_disk_rand.sh – Compiles disk_server and disk_rand. It checks argument validation and connection failures,
then runs short and longer random workloads to exercise the disk server under a mix of reads and writes.

Each test script is run inside the script utility to capture its terminal output:

• disk_server_tests.typescript – Record of running test_disk_server.sh.

• disk_cli_tests.typescript – Record of running test_disk_cli.sh.

• disk_rand_tests.typescript – Record of running test_disk_rand.sh.

These logs demonstrate that the disk programs behave correctly for both valid and invalid inputs and that they
terminate gracefully.

Problem 4: File System Server

4.1 Overview

Problem 4 introduces a flat file system built on top of the simulated disk server. The filesystem keeps track of files in a single directory table and exposes a small command set through fs_server (server) and fs_cli (client):

- F – format the filesystem on the disk.
- C f – create a file named f.
- D f – delete a file named f.
- L b – list directory contents (names only when b=0, names and sizes when b=1).
- R f – read the entire contents of file f.
- W f l – overwrite file f with l bytes of data.

The fs_server program talks to the lower-level disk_server over TCP to read and write 128-byte sectors, while fs_cli connects to fs_server and lets the user type filesystem commands interactively.

4.2 Build Instructions

The programs are built using the same Makefile as the earlier problems. From the project directory:

1. Ensure disk_server.c, fs_server.c, fs_cli.c, and the Makefile are present.
2. Run: make

   This produces the executables disk_server, fs_server, and fs_cli.

4.3 Running the Disk and File System Servers

First start the disk server, which simulates a physical disk file. Example:

```
./disk_server 5600 32 32 1000 disk.img
```

Here 5600 is the TCP port, 32 cylinders and 32 sectors per cylinder define the geometry, 1000 is the track-to-track delay in microseconds, and disk.img is the backing file used to store the raw 128-byte sectors.

Next start the filesystem server, pointing it at the disk server:

```
./fs_server 5601 127.0.0.1 5600
```

where 5601 is the port on which fs_server listens for filesystem clients, and 127.0.0.1:5600 is the host:port of the running disk_server. The programs print clear error messages and exit with a non-zero status if they cannot bind or connect to the requested ports.

4.4 Running the File System Client

Once fs_server is running, the client can connect with:

```
./fs_cli 127.0.0.1 5601
```

fs_cli enters a simple REPL-style loop. Each line typed by the user is treated as a filesystem command and sent to fs_server. For example:

```
F
C foo
W foo 18
```

Hello, filesystem!
R foo
L 1

fs_cli prints the exact status codes and data returned by the server. The client ignores blank lines, reports socket errors via perror(), and exits gracefully when EOF is reached on standard input or when the server closes the connection.

4.5 Expected Behavior and Error Handling

Typical interactions look like this (assuming a fresh disk image):

F
0
C foo
0
W foo 18
Hello, filesystem!
0
R foo
0 18 Hello, filesystem!
R bar
1 0

Return code 0 indicates success, 1 indicates a missing file, and 2 indicates some other failure such as a disk error. Both server and client check all system calls for failure, use perror() to display descriptive messages, and return non-zero exit codes when unrecoverable errors occur.

Problem 5: Directory Structure Client

5.1 Overview

Problem 5 adds a directory abstraction on top of the flat filesystem implemented for Problem 4. The underlying fs_server still manages a single table of files, but the fs_dirs client interprets certain file names as directories and maintains a current working directory (CWD). From the user's perspective, the following commands are available:

- mkdir dirname – create a directory named dirname.
- cd dirname – change the current directory to dirname.
- pwd – print the current working directory.
- rmdir dirname – remove the directory dirname (if it is empty).

The program communicates with fs_server using the existing filesystem protocol (C, D, L, and R commands) so no changes to fs_server are required.

5.2 Build Instructions

fs_dirs is compiled by the same Makefile as the rest of the project. From the project directory:

```
make fs_dirs
```

This produces an executable named fs_dirs in the current directory. Running make without arguments also builds fs_dirs along with the other programs.

5.3 Running the Directory Client

Before using fs_dirs, both disk_server and fs_server must already be running, as in Problem 4. For example:

```
./disk_server 5600 32 32 1000 disk.img
./fs_server 5601 127.0.0.1 5600
```

Then start the directory client and connect it to fs_server:

```
./fs_dirs 127.0.0.1 5601
```

The client presents a simple prompt of the form:

```
fs:/current/path$
```

At this prompt the user can type the commands mkdir, cd, pwd, rmdir, help, and exit/quit. The root directory is represented as '/', and the initial CWD is '/'. cd accepts either relative names (e.g., cd foo) or absolute paths starting with '/' (e.g., cd /foo/bar).

5.4 Examples and Expected Behavior

Example session (assuming a freshly formatted filesystem):

```
fs:/$ mkdir foo
fs:/$ cd foo
fs:/foo$ pwd
/foo
fs:/foo$ mkdir bar
fs:/foo$ cd bar
fs:/foo/bar$ pwd
```

/foo/bar

If mkdir is invoked on an existing directory, the client prints an error. cd into a non-existent directory prints a clear error but leaves the CWD unchanged. rmdir refuses to remove non-empty directories and reports the reason, matching the specification that an error should be thrown when a directory does not exist or cannot be removed.

Technical Manual

This section describes the internal design of each of the programs for Problems 1 through 5, including data structures, algorithms, and use of the socket API.

Basic Client-Server
client.c and *server.c*

Overall Architecture

The system follows a traditional TCP client-server architecture:
• The server listens on a fixed TCP port and accepts incoming connections.
• For each connection, the server spawns a dedicated thread that handles exactly one request.
• The client program creates a TCP connection to the server, sends a single line of text, receives the reversed line, prints it, and closes the connection.

This design isolates the per-request work into independent threads, allowing concurrent processing of multiple client requests.

Data Structures

The server code defines a small structure used to pass arguments into the worker threads:

```
typedef struct {
    int client_fd;          // connected client socket descriptor
    struct sockaddr_in addr;    // client address for logging
} client_arg_t;
```

For each accepted connection, a client_arg_t is allocated on the heap and populated with the new file descriptor and address. A pointer to this structure is passed to pthread_create(). The worker thread frees the structure as soon as it extracts the file descriptor.

Both client and server also use fixed-size character buffers for I/O:
• Client: out[8192] and in[8192]
• Server: buf[4096]

These buffers are large enough for the assignment requirements and are always null-terminated before being treated as C strings.

Threading Model

The server uses POSIX threads (pthreads) to handle concurrency:
1. The main thread calls socket(), setsockopt(), bind(), and listen() to set up the listening socket.
2. In an infinite loop, the main thread calls accept(). For each successful connection:
   • A client_arg_t structure is allocated and filled.
   • pthread_create() is called with thread_main() as the entry point.
   • The new thread is immediately detached using pthread_detach(), so the main thread does not need to join it.

3. thread_main() handles all communication with that client, then closes the socket and returns.

The worker threads include a call to sleep(2) to simulate processing delay and to make denial-of-service effects observable when many clients connect at once.

Client Algorithm

The client program follows this algorithm:
1. Validate arguments: at least three user arguments are required (host, port, and one word).
2. Convert the port string to an integer and ensure it is positive.
3. Build the output line by concatenating argv[3..] with single spaces, then appending a newline.
4. Create a TCP socket using socket(AF_INET, SOCK_STREAM, 0).
5. Fill a sockaddr_in structure with the server IP address and port, using inet_pton() to convert the address.
6. Call connect() to establish the TCP connection.
7. Send the entire request string to the server, looping on send() to handle partial sends.
8. Call recv() once to read the server's response into a buffer.
9. If recv() returns > 0, null-terminate the buffer and print it to standard output.
10. If recv() returns 0, report that the server closed the connection; if it returns < 0, use perror() to report an error.
11. Close the socket and exit with status 0 on success or 1 on error.

Server Algorithm

The server program follows this algorithm:
1. Validate command-line arguments (exactly one port argument) and ensure the port is positive.
2. Create a TCP socket using socket(AF_INET, SOCK_STREAM, 0).
3. Set SO_REUSEADDR on the socket using setsockopt() to allow rapid restart.
4. Bind the socket to INADDR_ANY and the chosen port using bind().
5. Start listening with a small backlog (e.g., 16) using listen().
6. Enter an infinite loop:
   a. Allocate a client_arg_t.
   b. Call accept() to wait for an incoming connection.
   c. On success, fill the client_arg_t with the new file descriptor and client address.
   d. Create a new thread with pthread_create(), passing the client_arg_t pointer.
   e. Detach the thread with pthread_detach().
   f. On accept() failure, log the error and continue.

7. The worker thread (thread_main()) executes the following steps:
   a. Free the client_arg_t structure and keep a local copy of the socket descriptor.
   b. Call sleep(2) to simulate processing time.
   c. Call recv() to read up to BUF_SIZE-1 bytes from the socket.

d. If recv() returns <= 0, log an error (if < 0), close the socket, and exit the thread.
e. Strip a trailing newline from the received data, if present.
f. Null-terminate the string.
g. Reverse the characters in place using reverse_inplace().
h. Append a newline to the reversed string.
i. Loop on send() until all bytes of the reversed string have been transmitted or send() fails.
j. On send() failure, log the error and break out of the loop.
k. Close the client socket and return from the thread.

String Reversal Algorithm

The core transformation is implemented by reverse_inplace(char *s, ssize_t n), which takes a pointer to a character array and its length n. The function uses a standard two-index swap algorithm:

- Initialize i = 0 and j = n-1.
- While i < j, swap s[i] and s[j], then increment i and decrement j.

This is an in-place, O(n) time, O(1) extra-space algorithm.

Directory Listing Server

ls_client.c and *ls_server.c*

Problem 2 replaces the simple string-reversal service with a server that executes the standard ls command. The server and client together implement a remote directory listing utility using TCP sockets and a fork/exec process model instead of threads.

Overall Architecture

The architecture is still based on a traditional TCP client-server model:

• ls_server listens on a TCP port, accepts incoming client connections, and forks a child process for each.

• Each child process reads one line of text containing ls-style arguments from the client, then runs ls with those arguments and sends its output back through the same socket.

• ls_client connects to the server, sends a single line containing the desired ls arguments, and prints whatever output the server sends (both normal output and error messages).

Process Model and Signal Handling

Unlike Problem 1, which used POSIX threads, ls_server uses a process-based concurrency model:

• The main process creates a listening socket and repeatedly calls accept().

• For each new connection, it forks a child process. The child is responsible for serving exactly one request.

• The child closes the listening socket, communicates with the client, and then terminates.

• The parent closes the connected client socket and returns to the accept loop.

To avoid zombie child processes, ls_server installs a SIGCHLD handler that calls waitpid(-1, &status, WNOHANG) in a loop until no more dead children remain. This ensures that the operating system can reclaim all child resources.

Data Structures

The server and client use simple buffers and argument vectors rather than complex custom structures:

• A fixed-size buffer (for example, char buf[4096]) is used in the child process to receive the line of ls arguments from the client.

• The received line is tokenized on whitespace to form an argv-style array for execvp(). The first element of that array is the string "ls", and the remaining elements are the tokens sent by the client.

• The client uses a buffer (for example, char payload[8192]) to construct the line of ls arguments to send, and another buffer (char buf[8192]) to receive ls output.

ls_client Algorithm

The ls_client program follows these steps:

1. Validate that at least three arguments are provided (host, port, and at least one ls

argument).
2. Convert the port string to an integer and ensure it is positive.
3. Build a single line of text by concatenating all ls arguments (argv[3..]) with spaces in between, then appending a newline character.
4. Create a TCP socket using socket(AF_INET, SOCK_STREAM, 0) and fill a sockaddr_in structure with the server address and port.
5. Call connect() to establish the TCP connection.
6. Send the entire payload to the server using a loop around send(), to correctly handle partial writes.
7. Repeatedly call recv() to read data from the server. Each chunk of data is immediately written to standard output using fwrite().
8. When recv() returns 0, treat that as end-of-stream and close the socket.
9. If recv() ever returns a negative value, report the error via perror("recv") and exit with a non-zero status.
10. On success, exit with status 0 after closing the socket.

ls_server Algorithm

The ls_server program follows this algorithm:
1. Validate that exactly one port argument is provided and that it is a positive integer.
2. Create a TCP socket and enable SO_REUSEADDR with setsockopt() to make restarting the server easier.
3. Bind the socket to INADDR_ANY and the requested port using bind().
4. Call listen() to start listening for incoming connections.
5. Install a SIGCHLD handler that reaps terminated children using waitpid(-1, &status, WNOHANG) in a loop.
6. Enter an infinite loop that calls accept() to wait for an incoming connection.
   - On accept() error, log the error with perror("accept") and continue the loop.
   - On success, fork a child process.
7. In the parent process:
   - Close the newly accepted client socket and return to the accept loop.
8. In the child process:
   - Close the listening socket, since the child does not accept new connections.
   - Call recv() to read a line of ls arguments into a buffer.
   - If recv() returns <= 0, log an error if appropriate, close the client socket, and exit.
   - Strip any trailing newline from the received line.
   - Tokenize the line into whitespace-separated arguments and build an array argv_exec such that:
     argv_exec[0] = "ls";
     argv_exec[1..n] = the tokens from the client;
     argv_exec[n+1] = NULL.
   - Use dup2() to redirect STDOUT_FILENO and STDERR_FILENO to the client socket descriptor.
   - Call execvp("ls", argv_exec) to replace the child with the ls program. If execvp() fails,

print an error and exit with a non-zero status.
9. After execvp(), the child process becomes ls, and any output or error messages from ls are sent directly over the socket to ls_client.

Basic Disk-Storage System
disk_cli.c, disk_rand.c and *disk_server.c*

Overall Architecture

The Problem 3 programs implement a simple block-based disk service on top of the TCP socket interface.
disk_server simulates a disk with a configurable geometry (cylinders, sectors, and a fixed 128-byte block size)
stored in a backing file. Clients speak a small text-based protocol to read and write blocks. disk_cli provides
an interactive front-end for issuing commands, while disk_rand generates automated random workloads for testing performance and robustness.

All communication occurs over TCP: disk_server listens on a port, and both disk_cli and disk_rand establish connections as clients.

Disk Geometry and Backing Store

disk_server stores the disk image in a regular file whose size is:
<cylinders> × <sectors> × 128 bytes.
This file is opened and resized with ftruncate(), then mapped into memory with mmap(). The mmap()'d region is
treated as an array of fixed-size blocks. The sector at cylinder c, sector s is located by computing a linear
index and adding BLKSZ bytes per sector:
index = c × sectors + s
pointer = base + index × 128

Because the disk is mapped into memory, reading or writing a block is simply a memcpy() between the buffer used
for network I/O and the appropriate part of the mapped region.

Process/Thread Model and Seek Time Simulation

disk_server uses a thread-per-connection model:
• The main thread creates a listening socket and repeatedly calls accept() to accept client connections.
• For each connection, it allocates a small client_arg_t structure, spawns a new thread, and passes the client
socket descriptor to that thread.
• Each worker thread handles all commands from its client until the client closes the connection or an error
occurs.

All threads share a single simulated disk arm position, represented by a global cylinder index protected by a
mutex. When a thread needs to access a block on cylinder c, it:
1. Locks the mutex.
2. Computes the distance in tracks from the current head cylinder to c.
3. Sleeps for distance × track_us microseconds using nanosleep().
4. Updates the global head position to c.
5. Performs the read or write on the mapped block.
6. Unlocks the mutex.

This ensures that even though multiple clients may be issuing requests concurrently, all disk accesses pass
through a single serialized "arm" that experiences the configured seek delay.

Data Structures

disk_server Command Handling Algorithm

The disk protocol is implemented in a loop that reads lines from the client and dispatches to command handlers:

• I
  The server sends a line containing two integers:
    <cylinders> <sectors>\n
  This allows clients to discover the disk geometry dynamically.

• R c s
  The server validates that c and s are within the configured geometry. If not, it sends a single byte '0'.
  If the block is valid, the server:
  - Acquires the disk arm mutex and simulates the seek from the current cylinder to c.
  - Copies 128 bytes from the mapped block at (c,s) into a temporary buffer.
  - Releases the mutex.
  - Sends a single byte '1' followed by the 128 data bytes.

• W c s l
  The server parses c, s, and l, and verifies that the cylinder/sector are valid and 0 ≤ l ≤ 128. If the
  parameters are invalid, it sends '0' and does not modify the disk. Otherwise it:
  - Reads exactly l bytes of raw data from the client.
  - Zero-fills a 128-byte buffer and copies the l bytes into the front.
  - Acquires the disk arm mutex, simulates a seek to cylinder c, and writes the buffer into the mapped block
    at (c,s).
  - Releases the mutex.
  - Sends a single byte '1' to indicate success.

If malformed commands or network errors are encountered, the worker thread closes the client socket and exits.

disk_cli Algorithm

disk_cli provides a straightforward command-line interface to the disk server:
1. It parses its command-line arguments, creates a TCP socket, and connects to disk_server.
2. It enters a loop reading lines from stdin using fgets(). Each line is sent to the server using a helper that
loops on send() until the entire line has been transmitted.
3. For an I command, disk_cli reads a single line response, null-terminates it, and prints it directly.
4. For an R c s command, it reads a single-byte status code. If the code is '0', it prints "0". If the code
is '1', it reads exactly 128 data bytes and prints a one-line summary that includes '1' and a hex dump of the
first part of the block.
5. For a W c s l command, disk_cli parses l from the command line, then reads exactly l raw bytes from stdin and
sends them to the server. It then reads back a single status byte and prints it.
6. Any unknown command type is reported to stderr, but the program continues reading further input until EOF or
an error occurs.

All network I/O uses helper functions that handle partial reads/writes and EINTR, and that treat unexpected
EOF or errors as fatal for the current session.

disk_rand Algorithm

disk_rand implements an automated test client that drives the disk server with random workloads:
1. It parses the host, port, number of operations N, and a random seed from its arguments, validating that N and
the port are positive.
2. It connects to disk_server and sends a single I command to retrieve the disk geometry. The returned cylinder
and sector counts are parsed from the response line.
3. For each of N iterations:
- A random cylinder in [0, cylinders-1] and random sector in [0, sectors-1] is selected.
- A random bit chooses between a read and a write operation.
- For writes, a 128-byte buffer is filled with random bytes, and a W c s 128 command is sent followed by the
128-byte block. disk_rand reads a status byte and prints 'w'.
- For reads, an R c s command is sent. The client reads a status byte and, if it is '1', reads

and discards
the 128 data bytes. It prints 'r' to indicate a completed read.
4. Periodically it prints newlines to keep the progress output readable, and at the end it prints a final newline
and closes the connection.

Because cylinder and sector indices are derived from the geometry returned by disk_server, all requests issued by
disk_rand are valid by construction.

File System Server
fs_cli.c and *fs_server.c*

Overall Architecture

The file system server implements a flat filesystem on top of the simulated disk. Clients speak a high-level filesystem protocol with fs_server over TCP. Internally, fs_server translates each filesystem command into one or more low-level disk requests issued to disk_server.

The major components in fs_server are:
• A TCP listener that accepts client connections and spawns a thread per client.
• A set of in-memory metadata structures describing the disk layout, directory, and file allocation table (FAT).
• Per-command handlers for F, C, D, L, R, and W that implement the filesystem semantics using the metadata and the underlying disk interface.

On-Disk Layout and Data Structures

The disk is organized into fixed-size 128-byte blocks. The layout is:
• Superblock: stored in block 0 and contains magic values and layout parameters.
• FAT region: an array of 32-bit entries, one per block, describing the allocation state (FAT_FREE, FAT_EOF, or index of next block in a file).
• Directory region: a fixed array of directory entries, each storing a file name, file length in bytes, and the index of the first data block.
• Data region: the remaining blocks that hold file contents.

fs_server caches the layout and FAT in memory in a global structure guarded by a pthread mutex to ensure thread-safe updates across concurrent clients.

Filesystem Operations

Each filesystem command is implemented as a dedicated function:
• F (format): initializes the superblock, clears the FAT to FAT_FREE, and marks all directory entries as unused. The global formatted flag is set to true.
• C f (create): scans the directory for an existing file named f. If a matching entry is found, the server returns code 1. Otherwise it finds a free directory slot, sets length = 0 and first = FAT_EOF, and returns code 0.
• D f (delete): locates the directory entry for f. If not found, the server returns 1. If found, it walks the FAT chain starting at first, marking each block FAT_FREE, then clears the directory entry and returns 0.
• L b (list): iterates over all directory entries whose name field is non-empty. When b = 0 it sends one file name per line; when b = 1 it also includes the file length and any additional metadata.
• W f l data (write): finds the directory entry for f. If missing, the server returns code 1.

Otherwise it frees any existing FAT chain, allocates enough free blocks to hold l bytes, writes the data into those blocks, updates the length and first fields, and responds with code 0 or 2 on failure.
• R f (read): finds the directory entry and walks its FAT chain, streaming the file data back to the client. The response format is "code len " followed by len bytes of file data and a trailing newline. Code 0 indicates success, 1 indicates a missing file, and 2 indicates some internal or disk error.

All metadata updates (FAT and directory) occur under the global mutex to ensure that concurrent clients see a consistent filesystem state.

Directory Structure

fs_dirs.c

Design and Representation

The underlying filesystem provided by fs_server is flat: each entry is identified only by a name and length, with no built-in notion of nested directories. fs_dirs implements directories purely at the client level by mapping canonical paths to special filesystem entries:

• The current working directory (CWD) is tracked as a canonical string that always begins with '/', with no trailing slash except for root.

• A directory with canonical path '/a/b' is represented in the flat filesystem by a zero-length marker file named 'a/b/'. In other words, directory names are stored as regular files whose names end with '/'.

• The root directory '/' is implicit and does not require a marker file.

This design allows the client to add directory semantics on top of the existing Problem 4 implementation without modifying fs_server.

Data Structures

Path Handling

Helper functions normalize and translate between user paths and internal filesystem names:

• join_path(cwd, name) – combines the current directory and a user argument (relative or absolute) into a canonical path string.

• path_to_fsname(path) – converts a canonical path like '/a/b' into the corresponding marker file name 'a/b/' used in the flat filesystem.

These routines ensure that operations such as mkdir, cd, and rmdir all use a consistent naming convention, and that redundant trailing slashes are removed from user input.

Implementation of mkdir, cd, pwd, rmdir

Each Problem 5 command is implemented in terms of the Problem 4 protocol:

• mkdir dirname:

1. Compute the canonical path P from cwd and dirname.
2. If P is '/', report an error (root cannot be created).
3. Map P to a marker file name F using path_to_fsname().
4. Send "C F" to fs_server. Return code 0 indicates success, 1 means the directory already exists, and 2 indicates another failure.

• cd dirname:

1. Compute the canonical path P. If P is '/', simply set cwd to '/'.
2. Otherwise map P to F and issue an "R F" request.
3. Only when the server returns code 0 (marker exists) is cwd updated; otherwise an error is printed.

• pwd:
  Simply prints the current canonical path string.
• rmdir dirname:
  1. Compute P and corresponding marker F.
  2. Use "R F" to verify that the directory exists.
  3. Issue "L 0" to obtain the complete file listing and scan it; if any name begins with F and is not equal to F itself, the directory is considered non-empty and the removal is refused.
  4. If the directory is empty, send "D F" to delete the marker file.

These steps ensure that rmdir only succeeds on empty directories, matching the expected semantics.