project_understanding& System calls

🔹 Project Title

Low-Latency Linux Command Line (Linux_CML)

High-Performance Linux CLI File Explorer in C++

📌 Overview

This project is a **high-performance Linux command-line file explorer and mini shell** implemented in **modern C++** using **POSIX/Linux system calls**.
It is engineered with a focus on:

✔ Clean Modular Architecture
✔ Direct OS interactions (no abstraction overhead)
✔ Low memory allocation overhead
✔ Deterministic, high-speed operations
✔ Real systems engineering practices

Tool	Purpose
WSL (Ubuntu 22.04)	Linux environment on Windows
g++	C++ compiler
CMake	Build system
Git	Version control
VS Code Remote-WSL	IDE with Linux environment integration
perf / Valgrind	Profiling and memory tools

Memory Type	Speed	Allocation	Lifetime
Stack	Very Fast	Automatic	Local Scope
Heap	Slower	Through new/malloc	Manual Control

Example (Stack buffer)

```
char buffer[4096];
```

Fast, no malloc, ideal for predictable latency.

🔹 System Call — getcwd()

C function signature:

```
char* getcwd(char* buf, size_t size);
```

Meaning:

✔ Ask the kernel for current working directory
✔ Kernel writes directory path into provided buffer
✔ Returns pointer on success, NULL on failure

Typical usage:

```
char buffer[PATH_MAX];
if(getcwd(buffer, sizeof(buffer)) != nullptr) {
    std::cout << buffer << "\n";
} else {
    perror("getcwd");
}
```

getcwd(buffer, sizeof(buffer))
This is core.

What is getcwd?
C library function.

Full form:

👉 get current working directory

Signature:

char* getcwd(char* buf, size_t size);
Parameters
1️⃣ buffer
Where to store result

2️⃣ size
How big is buffer

So:

OS writes path inside buffer
🧠 Under the hood (important)
When you call:

getcwd()
Flow:

Your program
↓
libc
↓
kernel syscall
↓
kernel gets cwd
↓
copies into your buffer
↓
returns pointer
So:

Kernel → user space copy.

Low level.
==

Let's visualize memory:

Suppose path:

```
/home/kshivam
```

Memory:

```
buffer:

| / | h | o | m | e | / | k | s | h | i | v | a | m | \0 | ... |
```

Note:

`\0` = null terminator

Marks end of string.

C style strings always end with \0

"Explain this function"

"It uses a fixed-size stack buffer to avoid heap allocations and calls the POSIX getcwd system call to retrieve the current working directory. The kernel writes the path directly into the provided buffer. We check for failure using the returned pointer and use perror for error reporting. This approach is faster and more deterministic than using std::string, which is better suited for low-latency systems."

🧠 Now quick intuition version

Think like:

```
char buffer[4096];   → empty box
getcwd()             → OS fills box
cout                 → print box
```

🔹 System Call — getcwd()

C function signature:

```
char* getcwd(char* buf, size_t size);
```

Meaning:

✔ Ask the kernel for current working directory
✔ Kernel writes directory path into provided buffer
✔ Returns pointer on success, NULL on failure

📌 Common Interview Questions You Can Answer

❓ Why use stack buffers over std::string?
✅ Stack avoids heap, offers deterministic performance.

❓ What is getcwd and how does it work?
✅ System call to retrieve current working directory into user buffer.

❓ Why use CMake instead of g++ directly?
✅ Scalability, reproducible builds, multi-file support.

❓ What is PATH_MAX?
✅ Maximum path length constant in Linux.

❓ Why do we avoid binaries in Git?
✅ Binaries are machine-specific and clutter repos.

CD..Command Notes

Linux File Explorer (CML) – cd Command Deep Notes

1. Problem Statement

Implement Linux shell command:

```
cd <path>
```

System Call Used

```
chdir()
```

Prototype:

```
int chdir(const char* path);
```

Return:

value	meaning
0	success
-1	failure

4. Final Code

```
void Shell::changeDirectory(const string& path){
    if(chdir(path.c_str()) != 0){
        perror("cd");
    }
}
```

void Shell::changeDirectory(const string& path)

Meaning

• Member function of Shell class
• Takes path as input

chdir(path.c_str())

Problem:

chdir() expects:
⇒const char*

But we have:
⇒std::string

Solution

Use:

`path.c_str()`

What is `c_str()` ?

Returns:

`const char*`

pointer to internal buffer of string.

Example:

```
string path = ".."
```

Memory:

```
Heap: ['.' '.' '\0']
```

`c_str()` → pointer to this memory

Interview point:

"POSIX APIs use C-style strings, so we convert std::string to char* using c_str()"

input ".."
↓
stored in std::string
↓
c_str() converts to char*
↓
chdir("..")
↓
kernel changes cwd
↓
returns 0
↓
no error printed

symbol	meaning
.	current
..	parent
/	root

Q1

Why use const string& ?

→ avoids copy, faster

Q2

Why c_str() ?

→ convert to char* for POSIX API

Q3

Where is current directory stored?

→ kernel (process structure)

Q4

Why not change directory manually?

→ only kernel can change cwd

Q5

What happens on failure?

→ returns -1 and sets errno

Q6

Stack vs heap here?

→ string on stack, char buffer on heap internally

Why is std::filesystem slower than chdir?

Answer:

filesystem:

- allocations

- exceptions
- abstraction
- extra checks

chdir:

- direct syscall
- minimal overhead

What happens if two threads call chdir() simultaneously?

🔥 VERY GOOD QUESTION

Answer:

cwd is **process-wide**, not thread-specific.

So:

Thread A:

```
chdir(/home)
```

Thread B:

```
chdir(/tmp)
```

Race condition.

Both affect each other.

Result:

UNSAFE in multi-threaded programs.

Why is `char buffer[PATH_MAX]` preferred over `vector<char>`?

Answer:

vector:

- heap allocation
- slower
- allocator overhead

array:

- stack
- fixed size
- zero allocation

HFT rule:

> Stack allocation preferred over heap to reduce latency.

Design question (very common)

❓

How would you design a high-performance shell for millions of file ops?

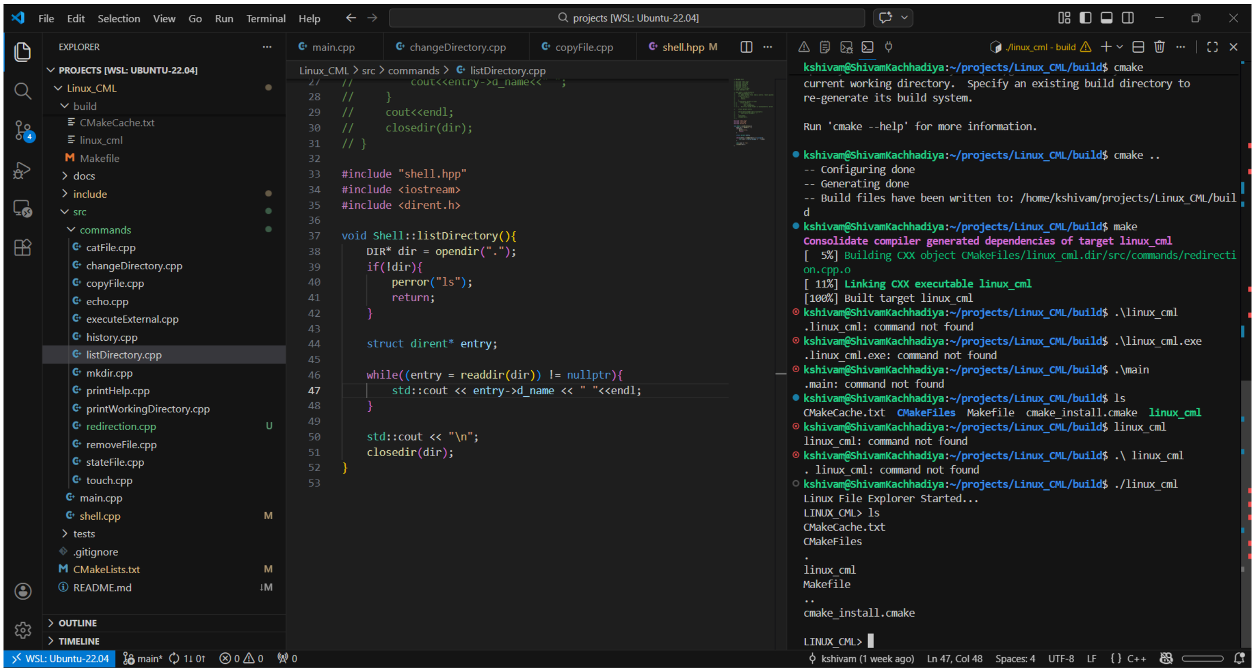
Good answer:

- avoid chdir frequently
- use absolute paths
- minimize syscalls
- avoid heap allocations
- use buffers
- cache metadata
- async IO if needed

LS Command Notes

LS Command Notes:

```
#include "include.hpp"
using namespace std;

void Shell::listDirectory(){
    DIR* dir=opendir(".");
    //. means current file. take a control. return
pointer. storee in DIR*
    if(dir==nullptr){
        perror("ls");
        return;
    }
    //internally dirent in linux
//      struct dirent {
//          ino_t d_ino;
//          char d_name[256];
// };   Each file inside folder is represented by: dirent

    struct dirent* entry;

    while((entry=readdir(dir))!=nullptr){
        cout<<entry->d_name<<"  ";
    }
    cout<<endl;
    closedir(dir);
}
```



🧠 FIRST — Big picture (what is ls actually?)

When you run:

```
ls
```

What happens internally?

Linux:

👉 A directory is just a file
👉 containing list of entries

Each entry:

```
filename + inode number
```

DIR* dir = opendir(".");

Think	Type
file	FILE*
directory	DIR*

Step 2 — What is opendir() ?

Prototype:

```
DIR* opendir(const char* path);
```

Meaning:

👉 "Kernel, open this folder and give me handle"

Step 3 — Why "." ?

```
"."
```

means:

👉 current directory

Same like:

pwd

Step 4 — What happens internally?

Kernel does:

1. find current directory inode
2. open it
3. allocate file descriptor
4. create DIR structure
5. return pointer

IMPORTANT:

struct dirent* entry;

What is dirent?

Very important.

Linux defines:

```
struct dirent {
    ino_t d_ino;
    char d_name[256];
};
```

Meaning:

Each file inside folder is represented by:

```
dirent
```

Example:

Directory:

```
src
main.cpp
build
```

Kernel returns:

```
dirent("src")
```

```
dirent("main.cpp")
dirent("build")
```

Why pointer?

```
struct dirent*
```

Not object.

Because:

- avoid copying
- faster
- kernel gives pointer to internal buffer

Low latency design.

Flow:

```
opendir(".")
↓
get DIR*
```

↓

readdir()

↓

print name

↓

repeat

↓

nullptr

↓

closedir()

SastaRivision

1️⃣ What is Linux (from zero)

Definition

Linux is a **Unix-like operating system**.

It provides:

- Process management
- Memory management
- File system
- System calls
- Device drivers

Architecture

Draw this diagram in notebook:

```
User Programs (Shell, Apps)
        ↓
System Calls (POSIX APIs)
        ↓
Linux Kernel
        ↓
Hardware (CPU, Disk, RAM)
```

Meaning

Layer	Role
App	your shell

Syscall	getcwd(), chdir(), fork()
Kernel	handles real work
Hardware	executes instructions

2️⃣ What is POSIX?

Definition

POSIX = **Portable Operating System Interface**

It is a **standard API for Unix systems**.

Why needed?

Without POSIX:

- Windows → different API
- Linux → different API
- Mac → different API

Code won’t be portable.

With POSIX:
Same code runs on:

- Linux
- Mac
- Unix
- WSL

Examples

POSIX Function	Use
getcwd()	get current dir
chdir()	change dir
opendir()	open directory
readdir()	list files
stat()	file info
fork()	create process
exec()	run program
pipe()	interprocess communication

---

3 What is Shell?

Definition

Shell = **command interpreter**

It:

1. reads command
2. parses command
3. executes action

Example

```
pwd
```

Flow:

```
User → Shell → getcwd() → Kernel → Path → Shell prints
```

4️⃣ What we built (big picture)

Draw this:

```
main()
   ↓
Shell.run()
   ↓
Read line
   ↓
Tokenize
   ↓
Match command
   ↓
Call function
```

5️⃣ Current Project Architecture

```
Linux_CML
│
├── include/shell.hpp
├── src/shell.cpp
├── src/commands/
│       ├── pwd
│       ├── cd
│       ├── ls
│       ├── stat
│       ├── echo
│       ├── help
```

Why modular?

Interview gold point:

> "Each command is implemented independently to follow single responsibility and scalability principles."

6️⃣ Deep Understanding of Each Command

🔹 PWD

Code concept

```
char buffer[PATH_MAX];
getcwd(buffer, sizeof(buffer));
```

How it works

1. buffer created on stack
2. getcwd asks kernel
3. kernel copies path into buffer
4. printed

Memory

```
Stack:
buffer[4096 bytes]
```

Interview Q

Q: Why buffer needed?
A: Kernel writes result into user memory.

🔷 CD

Code

```
chdir(path.c_str())
```

Flow

1. string → char*
2. kernel updates process working directory

Important

Working directory stored in:

```
task_struct (kernel)
```

Interview gold

Q: Why cd cannot be separate program?
A: Because directory is property of process.

🔷 LS

Code

```
DIR* dir = opendir(".");
readdir(dir);
```

Flow

```
opendir → open directory file
readdir → read entries
closedir → release
```

Memory

```
DIR* → heap object created by libc
```

Interview

Q: Why must close?
A: FD leak.

🔷 STAT

Code

```
stat(path, &info)
```

Gives

- size
- inode

- permissions
- type

Kernel

Reads inode metadata from filesystem.

Interview

Q: stat vs lstat?
A: lstat doesn't follow symlink.

🔹 ECHO

Code

```
vector<string> args
```

Why vector?

Modern C++ dynamic container.

Better than C arrays.

7️⃣ Core Engine (MOST IMPORTANT PART)

Your run() function:

Steps

Step 1

`getline`

Take full command

Step 2

`stringstream`

Tokenize

Step 3

`vector<string> tokens`

Store arguments

Step 4

Dispatch

`if(command=="pwd")`

Interview Question

Q: Why getline not cin >> ?
A: To support multi-argument commands.

8️⃣ Memory Concepts Used

You must know this for interviews:

Stack

- char buffer[PATH_MAX]
- local variables

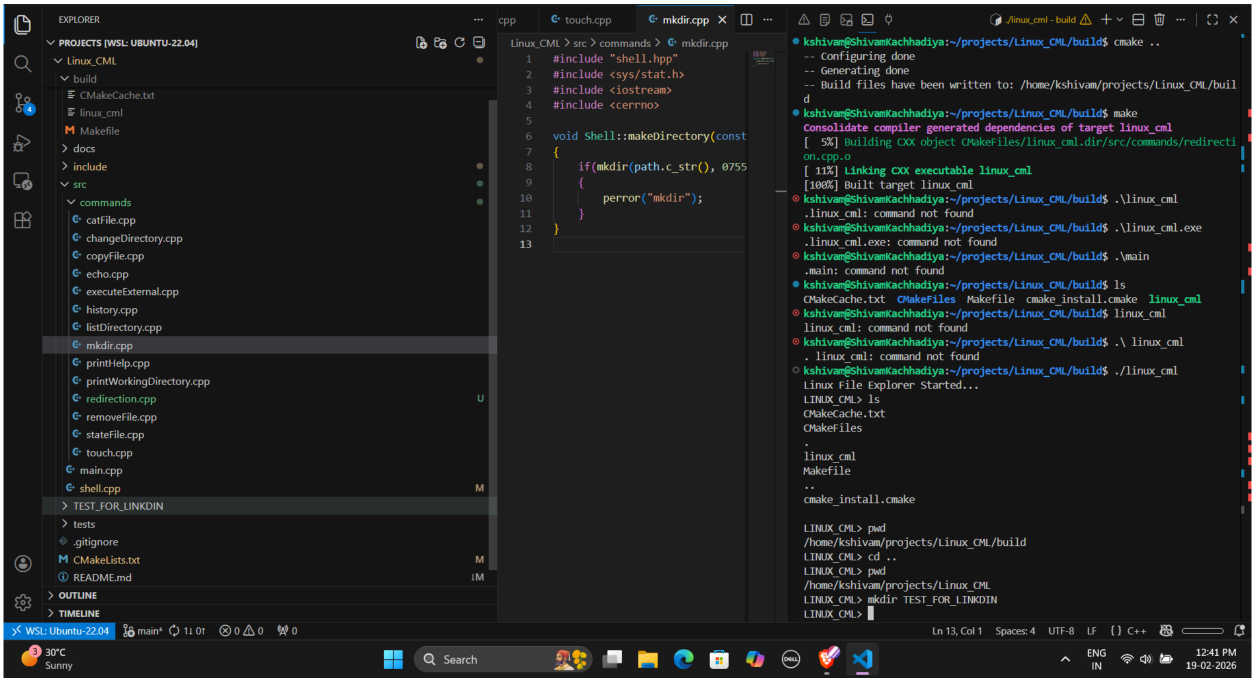
Heap

- vector
- string
- DIR*

Kernel memory

- directory info
- file metadata

mkdir command notes



✅ What is mkdir?

Definition

`mkdir` = **create a new directory entry in filesystem**

It creates:

- a new inode
- a directory file
- adds entry in parent directory

✅ POSIX system call

```
int mkdir(const char* path, mode_t mode);
```

✅ Full code

```
#include "shell.hpp"
#include <sys/stat.h>
```

```cpp
#include <cerrno>
#include <cstring>
#include <iostream>

void Shell::makeDirectory(const std::string& path)
{
    // call POSIX mkdir
    if (mkdir(path.c_str(), 0755) != 0)
    {
        // error handling
        std::cerr << "mkdir: " << std::strerror(errno) << "\n";
    }
}
```

Gives:

```
strerror()
```

Convert error number → message

Function

```cpp
void Shell::makeDirectory(const std::string& path)
```

Means:

- member function of Shell
- takes modern C++ string

```cpp
    if (mkdir(path.c_str(), 0755) != 0){}
```

---

path.c_str()

`path.c_str()`

WHY?

Because:

POSIX expects:

`const char*`

But we have:

`std::string`

So:

`string → char*`

conversion

---

mkdir call

`mkdir(path.c_str(), 0755)`

Create directory with permission:

`rwx r-x r-x`

---

check failure

`!= 0`

Because:

`0  → success`

```
-1 → failure
```

---

error printing

```
std::strerror(errno)
```

Example output:

```
mkdir: File exists
mkdir: Permission denied
```

Example:

```
0755
```

Meaning:

```
Owner  → rwx (7)
Group  → r-x (5)
Others → r-x (5)
```

Q1

Difference between mkdir and open?

Answer:

open():

- creates regular file

mkdir():

- creates directory inode
- adds "." and ".."

Q2

Why mode 0755 not 755?

Answer:

0755 = octal
permissions stored in octal

Q3

What is umask?

Answer:

Default permission mask applied by kernel:

```
final = mode & ~umask
```

Q4

What happens internally when mkdir called?

Answer:

- inode allocation
- directory entry creation
- update parent directory
- permission set

Q5

Why use string.c_str()?

Answer:

POSIX APIs use C style char*

Q6

Difference between mkdir and mkdir -p?

Answer:

mkdir → single level
mkdir -p → recursive parent creation

Q7

Is directory special?

Answer:

No. It’s a file storing name → inode mappings.

Q8

What happens if directory exists?

Answer:

mkdir returns -1
errno = EEXIST

Q9

Thread safety?

Answer:

mkdir is atomic at kernel level

Q10 (advanced)

Which syscall used internally?

Answer:

Linux uses:

```
sys_mkdir
sys_mkdirat
```

rmdir command notes

🔥 Code

```
#include "shell.hpp"
#include <iostream>
#include <sys/stat.h>
#include <unistd.h>
using namespace std;
void Shell::removeFile(const string& path){

    struct stat st;
    if (lstat(path.c_str(), &st) == -1) {
        perror("rm");
        return;
    }

    if (S_ISDIR(st.st_mode))
        rmdir(path.c_str());
    else
        unlink(path.c_str());

}
```

```
void Shell::removeFile(const string& path){
```

👉 Shell class ka function
👉 input: file/directory ka path
👉 kaam: delete karna

🔹 1️⃣ struct stat info;

```
struct stat info;
```

Kya hai?

Kernel structure jisme **file metadata store hota hai**

Contains:

```
st_mode   → file type + permissions
st_size   → size
st_uid    → owner
st_gid
st_mtime  → modified time
st_ino    → inode number
```

👉 Basically **inode ka data**

🔹 2️⃣ stat()

```
if(stat(path.c_str(),&info)!=0){
```

Syscall:

```
stat()
```

Kya karta hai?

👉 "File ka metadata lao"

Internally:

```
path → directory lookup → inode → fill struct stat
```

Return value

```
0  → success
-1 → fail
```

So:

```
!= 0  → error
```

Fail kab hoga?

Case	errno
file not exist	ENOENT
permission denied	EACCES

🔹 3️⃣ perror()

```
perror("rm");
```

kya karta hai?

Automatically prints:

```
rm: <error message>
```

Example:

```
rm: No such file or directory
```

👉 uses errno internally

Better than manually:

```
strerror(errno)
```

🔹 4️⃣ return

```
return;
```

Agar file exist hi nahi karti → delete ka sense nahi
👉 exit function early

🔥 Next: file type check

🔹 5️⃣ S_ISREG

```
if(S_ISREG(info.st_mode)){
```

Kya hai?

Macro

Checks:

```
Is this a regular file?
```

Internally:

```
(info.st_mode & S_IFMT) == S_IFREG
```

Regular file examples

```
.txt
.cpp
.log
```

🔹 6️⃣ unlink()

```
if(unlink(path.c_str())!=0){
```

Syscall

`unlink()`

Important concept

👉 File delete **actually means removing directory entry**

Linux me:

```
filename → inode
```

unlink:

```
remove filename link
decrement link count
if count=0 → free inode + blocks
```

Interview gold line:

👉 **unlink removes name, not immediately the data**

If file open hai:

```
data stays until last fd closed
```

🔥 Directory case

🔹 7️⃣ S_ISDIR

```
else if(S_ISDIR(info.st_mode)){
```

Check:
👉 directory hai ya nahi

🔹 8️⃣ rmdir()

```
if(rmdir(path.c_str())!=0){
```

Syscall

```
rmdir()
```

Kya karta hai?

👉 empty directory delete

Important:

```
rmdir only works if directory empty
```

Otherwise:

```
ENOTEMPTY
```

Why separate from unlink?

Because:

- directories special structure hoti hain
- contains . and ..
- safety reasons

So:

```
files → unlink
dirs → rmdir
```

🔷 9️⃣ else

```
cout<<"Unsupport file type\n";
```

Other types:

- symlink
- socket
- FIFO
- device file

Abhi handle nahi kiya

🔥 Full flow (simple)

Program logic:

```
stat(path)

if not exist → error

if regular → unlink
if directory → rmdir
else → unsupported
```

👉 basically mini `rm`

🔥 Important OS Concepts hidden here

This small code tests:

✅ **inode**

✅ **stat**

✅ **file types**

✅ **unlink vs rmdir**

✅ **errno**

✅ **system calls**

Interviewers love this.

🔥 Interview Questions (Intermediate → Advanced)

🟡 Intermediate

Q1

Why use stat before unlink?

👉 To check file type and existence.

Q2

Difference between unlink and rmdir?

👉 unlink → files
👉 rmdir → empty directories

Q3

What happens if you unlink an open file?

👉 file removed from directory
👉 data stays until fd closed

Q4

Why stat returns metadata not filename?

👉 filename stored in directory entry, not inode

🟠 Advanced

Q5

Why stat + unlink causes race condition?

Between:

```
stat()
unlink()
```

file might change (TOCTOU bug)

Safer:
👉 try unlink directly and handle error

Q6

Difference between stat and lstat?

👉 stat follows symlink
👉 lstat gives symlink info

For rm:

```
should use lstat
```

otherwise symlink target delete ho sakta hai

Q7

How does rm -r work internally?

👉 recursion:

```
opendir
readdir
stat
unlink/rmdir
```

Q8

What is link count?

inode field:

```
st_nlink
```

file deleted only when:

```
link count == 0 AND no open fds
```

Q9

Why directories need execute permission?

Without x:

```
cannot cd or traverse
```

Q10 (very common)

How would you improve this code?

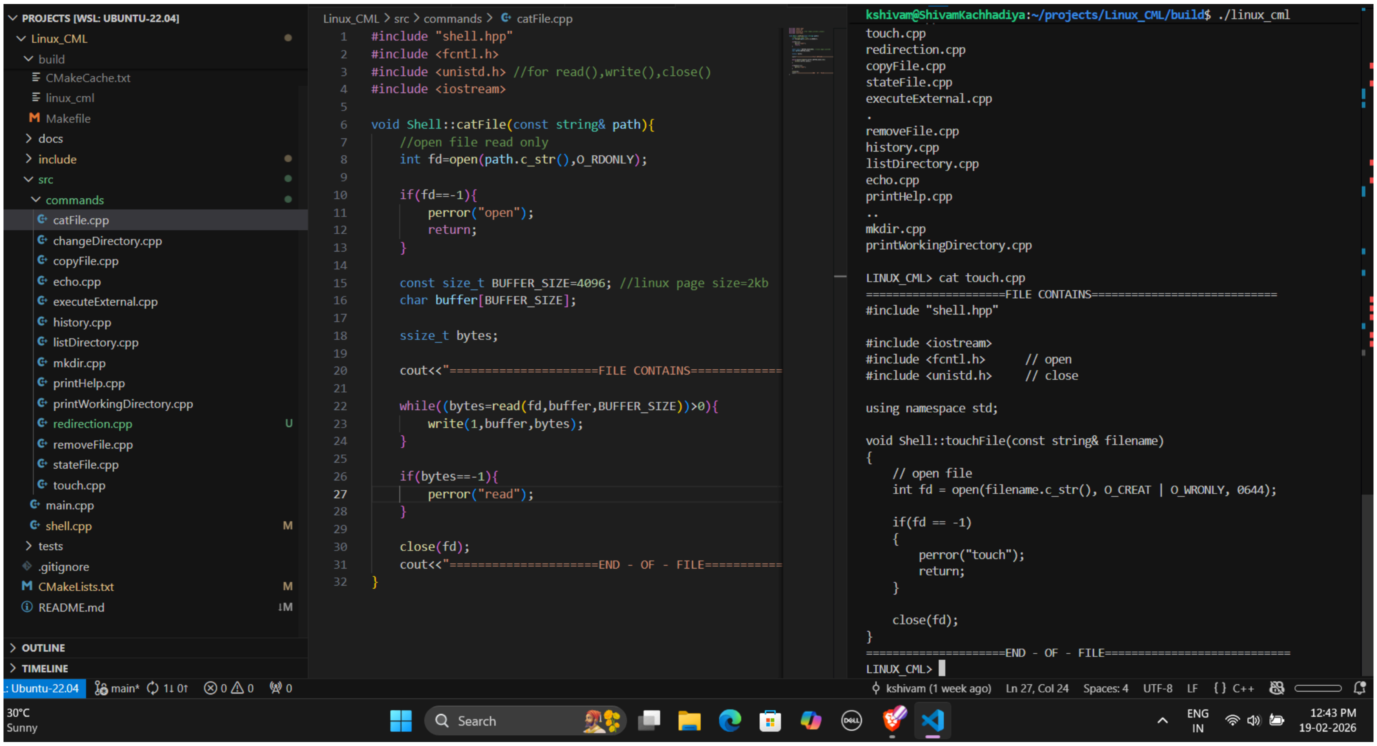
Expected answer:

- use lstat
- handle symlinks
- recursive delete
- better error handling
- avoid TOCTOU
- use unlinkat()

catFile command

CODE: catFIle.cpp

```
#include "shell.hpp"
#include <fcntl.h>
#include <unistd.h> //for read(),write(),close()
#include <iostream>

void Shell::catFile(const string& path){
    //open file read only
    int fd=open(path.c_str(),O_RDONLY);

    if(fd==-1){
        perror("open");
        return;
    }

    const size_t BUFFER_SIZE=4096; //linux page size=2kb
    char buffer[BUFFER_SIZE];

    ssize_t bytes;

    cout<<"=====================FILE
CONTAINS============================"<<endl;

    while((bytes=read(fd,buffer,BUFFER_SIZE))>0){
        write(1,buffer,bytes);
    }

    if(bytes==-1){
        perror("read");
    }

    close(fd);
    cout<<"=====================END - OF -
FILE============================"<<endl;
}
```



Example

```
int fd = open("a.txt", O_RDONLY);
```

Suppose:

```
fd = 3
```

Means:

👉 "Kernel ne file a.txt ko index 3 pe map kar diya"

Process ke paas FD table hota hai

```
FD table (per process)

0 → stdin
1 → stdout
2 → stderr
3 → a.txt
```

```
4 → pipe read
5 → socket
```

👉 FD is just index into this table

🔹 Line 1

```
int fd = open(path.c_str(), O_RDONLY);
```

Syscall:

```
open()
```

What happens internally?

Kernel:

1. find file inode
2. permission check
3. create open file object
4. add entry in FD table
5. return FD number

Return:

```
>=0 → fd
-1 → error
```

🔹 Line

```
const size_t BUFFER_SIZE = 4096;
```

Why 4096?

👉 4KB = Linux page size

Benefits:

- fewer syscalls
- faster
- page aligned

Important:

More read calls = slower

Bad:

```
read 1 byte each time ❌
```

Good:

```
read 4KB chunks ✔
```

Interview question

Why bigger buffer improves performance?

👉 fewer syscalls → fewer kernel switches

🔹 Line

```
char buffer[BUFFER_SIZE];
```

Memory to store read data.

🔹 Line

```
while((bytes = read(fd, buffer, BUFFER_SIZE)) > 0)
```

Syscall:

```
read()
```

What it does:

👉 copies data:

```
kernel → user buffer
```

Returns:

value	meaning
>0	bytes read
0	EOF
-1	error

Example:

File size = 10KB

Flow:

```
4096
4096
1808
0 (EOF)
```

🔹 Line

```
write(1, buffer, bytes);
```

Syscall:

```
write()
```

What is 1??

👉 FD 1 = stdout

Default mapping:

```
0 → stdin
1 → stdout
2 → stderr
```

So:

```
write(1,...) → print on screen
```

Example:

```
write(1, "hello", 5)
```

prints:

```
hello
```

🔥 Visual diagram (very important)

During execution

```
Process FD table

0 → stdin
1 → stdout
2 → stderr
3 → file.txt  ← fd returned by open
```

Then:

```
read(3, buffer)
write(1, buffer)
```

🔥 Entire flow summary

```
open → get fd
loop:
    read(fd)
    write(stdout)
close(fd)
```

👉 Exactly how cat works internally

🔥 One-line memory tricks

Remember:

FD basics

```
0 stdin
1 stdout
2 stderr
```

cat logic

```
open → read → write → close
```

performance

```
big buffer = fewer syscalls = faster
```

✅ Q1. What is a File Descriptor (FD)?

Answer

File descriptor is an **integer handle** that refers to an **open file object in the kernel**.

Each process has an **FD table**:

```
0 → stdin
1 → stdout
2 → stderr
3+ → files/sockets/pipes
```

`open()` returns this index.

One-liner (interview)

👉 FD is an index into the per-process file descriptor table.

✅ Q2. What happens internally when open() is called?

Answer (steps)

Kernel:

1. path lookup → inode
2. permission check
3. create open file description
4. add entry to FD table
5. return fd

Diagram

```
fd → file table → inode → disk blocks
```

One-liner

👉 open maps file to kernel object and returns fd.

✅ Q3. Why read()/write() instead of printf()/cout?

Answer

read/write	printf/cout
syscall	library
unbuffered	buffered
faster, low-level	convenient

System tools use read/write for:

- performance
- control
- binary safety

One-liner

👉 read/write are raw syscalls; printf is buffered user-space.

✅ Q4. Why use large buffer (4096)?

Answer

Because:

- 4096 = page size

- fewer syscalls
- fewer kernel switches
- faster

Bad:

```
read 1 byte → slow
```

Good:

```
read 4KB → fast
```

One-liner

👉 Larger buffer reduces syscall overhead.

✅ Q5. What does read() return and why ssize_t?

Answer

```
>0 → bytes read
0 → EOF
-1 → error
```

ssize_t is signed so it can return -1.

One-liner

👉 ssize_t allows negative error value.

✅ Q6. How does `cat file > out.txt` work internally?

Answer

Shell does:

```
open(out.txt)
dup2(fd, 1)
exec(cat)
```

Now:

```
write(1,...) → goes to file
```

Key concept

👉 stdout is just FD 1

One-liner

👉 redirection uses dup2 to replace stdout.

✅ Q7. What happens if you don't close(fd)?

Answer

FD leak:

- kernel resources not freed
- eventually:

```
Too many open files
```

Limit:

```
ulimit -n
```

One-liner

👉 not closing causes file descriptor leak.

✅ Q8. Difference between text file and binary file in Linux?

Answer

Linux:

```
NO difference
```

Everything is bytes.

`read/write` treat both same.

Only terminal interprets ASCII.

One-liner

👉 Linux has no text/binary distinction.

✅ Q9. Why is cat slow for huge files? How to optimize?

Problem

Too many read/write syscalls.

Solutions

- bigger buffer
- use `mmap`

- use `sendfile`
- async I/O

mmap approach

```
file mapped directly to memory
no copy
faster
```

One-liner

👉 use mmap or sendfile to reduce copies.

✅ Q10. Explain full data path of read().

Deep (very common advanced question)

When you do:

```
read(fd, buf, 4096);
```

Flow:

```
disk → kernel page cache → copy to user buffer
```

So 2 copies:

1. disk → kernel
2. kernel → user

mmap removes second copy.

One-liner

👉 read copies kernel buffer to user space.

🔥 Rapid 30-sec Revision (before interview)

Remember only this:

```
FD = file handle
open → get fd
read → bytes from kernel
write → bytes to stdout(1)
4096 buffer → fewer syscalls
close → free resources
Linux files = bytes only
dup2 → redirection
mmap/sendfile → faster
```

copyFile command

```cpp
#include "shell.hpp"
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <cstring>

void Shell::copyFile(const string&src,const string& dest){
    //open file
    int src_fd=open(src.c_str(),O_RDONLY);
    if(src_fd<0){
        perror("open source");
        return;
    }

    //open/create destination file

    int dest_fd=open(dest.c_str(),
                O_WRONLY|O_CREAT|O_TRUNC,
                0644);


    if(dest_fd <0){
        perror("open destination");
        close(src_fd);
        return;
    }

    const int BUFFER_SIZE=4096;
    char buffer[BUFFER_SIZE];
    ssize_t bytes;

    //read -> write loop
    while((bytes=read(src_fd,buffer,BUFFER_SIZE))>0){
        if(write(dest_fd,buffer,bytes)!=bytes){
```

```
        perror("write");
        break;
      }
  }
  if(bytes<0)
      perror("read");
  close(src_fd);
  close(dest_fd);
}
```

🔹 STEP 1 — Open source file

```
int src_fd = open(src.c_str(), O_RDONLY);
```

What happens internally (kernel steps)

When you call `open()`:

Kernel:

1. Path lookup

```
/home/a.txt → inode
```

2. Permission check

Read allowed?

3. Create open file object

Stores:

- offset

- flags
- inode pointer

4. Add to FD table

```
fd → open file object
```

5. return fd

Example:

```
src_fd = 3
```

Why int?

FD = integer index.

Interview Q

Q: What does open return internally?

👉 index into per-process file descriptor table.

```
if(src_fd < 0){
    perror("open source");
    return;
}
```

Why check?

Because:

```
-1 → failure
```

Possible reasons:

- file not exist
- permission denied
- too many open files

Why perror?

Prints:

```
open source: No such file or directory
```

Uses errno.

Interview Q

Q: Why must every syscall be checked?

👉 kernel can fail anytime → never assume success.

🔹 STEP 2 — Open destination file

```
int dest_fd = open(dest.c_str(),
                   O_WRONLY|O_CREAT|O_TRUNC,
```

```
                  0644);
```

Flags breakdown (very important)

O_WRONLY

write only

O_CREAT

Create file if not exists

O_TRUNC

If file exists:

```
size = 0
```

overwrite

Combined meaning

```
open for writing,
create if needed,
clear old content
```

Exactly cp behavior.

0644 (permissions)

Octal:

```
owner 6 → rw-
group 4 → r--
other 4 → r--
```

Meaning:

```
rw-r--r--
```

Owner can write, others read only.

Default safe permission for files.

Interview Q

Q: Why cp uses 0644 not 0755?

👉 normal files shouldn't be executable.

```
if(dest_fd <0){
    perror("open destination");
    close(src_fd);
    return;
}
```

Why close(src_fd)?

If dest open fails:

👉 avoid FD leak

VERY important.

Interview Q

Q: What happens if we don't close?

👉 FD leak → eventually “Too many open files”.

🔹 STEP 3 — Buffer

```
const int BUFFER_SIZE=4096;
char buffer[BUFFER_SIZE];
```

Why 4096?

Because:

```
4KB = Linux page size
```

Benefits:

- page aligned
- optimal disk I/O
- fewer syscalls
- faster

Performance logic

Bad:

```
read 1 byte → 1M syscalls
```

Good:

```
read 4KB → 250 syscalls
```

Interview Q

Q: Why large buffers improve performance?

👉 reduces syscall overhead.

🔹 STEP 4 — read → write loop

```
while((bytes=read(src_fd,buffer,BUFFER_SIZE))>0){
```

---

read() internals

Flow:

```
disk → kernel page cache → copy to buffer
```

Return:

```
>0 bytes
0 EOF
-1 error
```

Why ssize_t?

Signed because:

```
-1 needed for error
```

```
if(write(dest_fd,buffer,bytes)!=bytes){
```

---

write() internals

Flow:

```
user buffer → kernel → disk
```

Why check != bytes ?

Because:

write may:

- partial write
- interrupted

So:

```
write < bytes → incomplete
```

Important correctness check.

Interview Q (advanced)

Q: Can write write fewer bytes than requested?

👉 Yes (partial write). Must loop until complete.

Most candidates miss this.

```
perror("write");
break;
```

Stop copying on error.

🔹 STEP 5 — read error check

```
if(bytes<0)
    perror("read");
```

If read returned -1 → error.

🔹 STEP 6 — close

```
close(src_fd);
close(dest_fd);
```

---

close() internals

Kernel:

- remove FD entry
- decrement ref count
- flush buffers
- free resources

Why mandatory?

Otherwise:

- memory leak
- FD leak
- data may not flush

Interview Q

Q: When is data actually written to disk?

👉 usually on close or flush.

🔥 FULL FLOW DIAGRAM

```
open(src)  → fd 3
open(dest) → fd 4
```

```
loop:
    read(3)  → buffer
    write(4) → disk

close both
```

🔥 Deep Kernel Picture (advanced)

```
disk
 ↓
page cache
 ↓ copy
user buffer
 ↓ copy
page cache
 ↓
disk
```

2 copies.

Better:

```
sendfile/mmap → zero copy
```

🔥 High ROI Interview Questions

🔥 Q1

Difference between file descriptor and FILE* ?

👉 fd = kernel
👉 FILE* = user buffered

🔥 Q2

How to make copy faster?

👉 mmap
👉 sendfile
👉 splice
👉 bigger buffer

🔥 Q3

What happens if process crashes before close?

👉 kernel auto closes all FDs

🔥 Q4

What if src and dest are same file?

👉 data corruption
(real cp checks inode equality)

🔥 Q5

Why open dest before reading?

👉 fail fast, avoid partial copy

🔥 Q6

What happens after fork?

👉 child inherits same FDs

🔥 Q7

How many copies happen in read/write?

👉 two copies

🔥 Q8

Why O_TRUNC needed?

👉 clear old content

🔥 Q9

What is partial write?

👉 write returns less than requested

🔥 Q10

How would you implement cp -r ?

👉 recursion + opendir + readdir

🔥 One-line memory summary

```
open → get fd
read → bytes in
write → bytes out
buffer 4KB
check partial write
close always
0644 safe permission
```

👉 **FD = integer number that represents an open file**

Linux rule:

```
Everything is a file
```

So FD can represent:

- regular file
- directory
- pipe
- socket
- terminal

Example

```
int fd = open("a.txt", O_RDONLY);
```

Suppose return:

```
fd = 3
```

Means:

👉 “Kernel ne a.txt ko FD table ke index 3 pe map kar diya”

🔥 PART 2 — FD Table kya hota hai?

Har process ke paas hota hai:

👉 **File Descriptor Table (per process)**

Example table

```
Process FD Table

FD   Points to
----------------------
0    stdin (keyboard)
1    stdout (screen)
2    stderr (screen)
3    a.txt (source)
4    b.txt (destination)
```

👉 FD sirf index hai
👉 actual file kernel object me hoti hai

🔥 PART 3 — src_fd and dest_fd kya represent karte hain?

Tumhara code:

```
int src_fd = open(src.c_str(), O_RDONLY);
```

```
int dest_fd = open(dest.c_str(), O_WRONLY|O_CREAT|O_TRUNC,
0644);
```

Meaning

src_fd

👉 source file handle
👉 read karne ke liye

Example:

```
src_fd = 3
```

---

dest_fd

👉 destination file handle
👉 write karne ke liye

Example:

```
dest_fd = 4
```

So:

```
read(3)  → from a.txt
write(4) → into b.txt
```

🔥 PART 5 — DRY RUN (Full flow)

Initial state

a.txt

```
hello world
(11 bytes)
```

b.txt

```
(empty)
```

STEP 1 — open source

```
src_fd = open("a.txt", O_RDONLY);
```

Kernel:

```
find inode
check permission
create open file object
add FD entry
```

FD table:

```
0 stdin
1 stdout
2 stderr
3 a.txt
```

STEP 2 — open destination

```
dest_fd = open("b.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Kernel:

- create file if missing
- truncate to 0
- open for writing

FD table:

```
4 b.txt
```

🔥 Now copy loop starts

STEP 3 — read()

```
bytes = read(3, buffer, 4096);
```

Kernel:

```
disk → page cache → copy → buffer
```

buffer contains:

```
"hello world"
```

bytes = 11

STEP 4 — write()

```
write(4, buffer, 11);
```

Kernel:

```
buffer → page cache → disk
```

b.txt becomes:

```
hello world
```

STEP 5 — next read

```
read(3)
```

returns:

```
0 (EOF)
```

Loop ends.

STEP 6 — close

```
close(3);
close(4);
```

Kernel:

- flush
- free FD entries

🔥 Full Visual Flow

Kernel + User space diagram

```
a.txt (disk)
     ↓
kernel page cache
     ↓ copy
user buffer
     ↓ copy
kernel page cache
     ↓
b.txt (disk)
```

🔥 Super simple mental model

```
fd 3 → read
fd 4 → write
```

Loop:

```
copy bytes from 3 → 4
```

🔥 Step-by-step animation

```
[read]
buffer = "hello world"

[write]
b.txt = "hello world"

[next read]
EOF
```

Done.

🔥 Common Interview Questions (High ROI)

Q1

What is file descriptor?

👉 index into per-process FD table

Q2

Why two FDs needed?

👉 one for reading, one for writing

Q3

Why O_TRUNC needed?

👉 clear old content before writing

Q4

What happens if O_CREAT not used?

👉 open fails if file missing

Q5

Why close important?

👉 release resources + flush data

Q6

How many copies happen here?

👉 two copies (kernel↔user)

Q7

How to optimize?

👉 mmap/sendfile (zero copy)

Q8

Can write return partial?

👉 yes, must handle

Q9

Where is file offset stored?

👉 kernel open file description

Q10 (advanced)

After fork, what happens to FDs?

👉 child inherits same FD table

🔥 Final Memory Cheat

Remember:

```
FD = handle
open → get fd
read(fd1)
write(fd2)
close
flags control behavior
```

FD table

🔥 PART 1 — Sabse pehle reality check

Linux rule

👉 **Everything is a file**

Linux ke liye:

```
file
socket
pipe
terminal
keyboard
stdout
```

sab = FILE

So OS needs **one uniform handle**

That handle =

🔥 File Descriptor (FD)

🔥 PART 2 — FD kya hota hai?

Simple definition

👉 FD = small integer number

Example:

```
3
4
```

```
5
```

Why integer?

Because:

```
index into table
```

Fast lookup.

🔥 PART 3 — FD table kya hota hai?

Definition

👉 **Per-process array of open files**

Har process ke paas:

```
FD table
```

Imagine like this

```
Process
   |
   +--- FD Table (array)
```

🔥 REAL MEMORY DIAGRAM

Example program

```
open("a.txt");
open("b.txt");
```

FD table becomes

```
FD   Points to
----------------------
0    stdin
1    stdout
2    stderr
3    a.txt
4    b.txt
```

👉 FD number = index
👉 inside = pointer to kernel file object

🔥 Golden Rule (remember forever)

```
FD is NOT file
FD is pointer to file
```

🔥 PART 4 — Default FD

Every process automatically has:

```
0 → stdin  (keyboard)
1 → stdout (screen)
2 → stderr (screen)
```

Always.

So:

```
printf() → write(1,...)
```

🔥 PART 5 — open() kya karta hai?

```
int fd = open("a.txt", O_RDONLY);
```

Kernel:

1. find inode
2. create open file object
3. add entry to FD table
4. return index

Example:

```
fd = 3
```

So:

```
read(3,...)
```

means:

👉 read from a.txt

🔥 PART 6 — read() ka magic

```
read(fd, buffer, 100);
```

Kernel:

```
FD table[fd]
     ↓
file object
     ↓
disk
```

Copy data to buffer.

🔥 PART 7 — dup2 ka asli game

```
dup2(3,1);
```

Means:

```
FD 1 → same file as FD 3
```

Before

```
1 → terminal
3 → file.txt
```

After

```
1 → file.txt
3 → file.txt
```

So:

```
printf() → goes to file
```

👉 This is HOW redirection works.

🔥 PART 8 — fork() ka effect on FD table

```
fork();
```

Child gets:

👉 copy of FD table

Example:

Before:

```
Parent
3 → a.txt
```

After:

```
Parent  3 → a.txt
Child   3 → a.txt
```

Both share same file object.

So:

👉 offset shared

Interview favorite question 🔥

🔥 PART 9 — exec() ka effect

---

exec:

```
memory replaced
BUT FD table remains
```

So:

👉 redirection still works after exec

This is WHY:

```
dup2 before exec
```

works.

🔥 PART 10 — FULL FLOW example

Command:

```
ls > out.txt
```

Step 1

open:

```
fd=3
```

Step 2

dup2:

```
1 → out.txt
```

Step 3

exec(ls)

Step 4

ls writes:

```
write(1,...)
```

→ file

🔥 Final mental model

Remember:

```
FD = index
Table = mapping
dup2 = change mapping
```

🔥 NOW your 2nd question

🔥 What is size_t ?

Definition

```
unsigned integer for sizes
```

Used for:

```
memory size
array length
buffer length
```

Why unsigned?

Because:

```
size can't be negative
```

Example

```
size_t n = read(fd, buf, 4096);
```

Actual type?

Depends system:

system	size
32 bit	unsigned int
64 bit	unsigned long

Interview answer

👉 size_t = type for sizes returned by sizeof, malloc, read lengths

---

🔥 What is pid_t ?

Definition

```
type for process ID
```

Used for:

```
fork()
waitpid()
getpid()
```

Example

```
pid_t pid = fork();
```

Why not int?

Because:

OS may change size.

So portable typedef.

Usually:

```
typedef int pid_t;
```

But abstracted.

🔥 Quick cheat table

Type	Meaning	Used for
int	general	numbers
size_t	size	memory length
ssize_t	signed size	read/write return
pid_t	process id	fork/wait
off_t	file offset	lseek

🔥 Top Interview Questions (FD Table)

Q1

What is file descriptor?

👉 index into per-process FD table

Q2

Why 0,1,2 reserved?

👉 stdin, stdout, stderr

Q3

What does dup2 do?

👉 duplicate FD mapping

Q4

Does exec clear FD?

👉 No

Q5

Does fork copy FD table?

👉 Yes (shared objects)

Q6

How redirect stdout?

👉 dup2(fd,1)

Q7

Difference between size_t and int?

👉 unsigned, platform dependent

Q8

Why read returns ssize_t?

👉 can return -1 (error)

Q9

Where is FD table stored?

👉 kernel space, per process

Q10

What happens if not close FD?

👉 leak resources

🔥 Final ultra simple memory rule

Remember 4 lines only

```
FD = number
table = mapping
dup2 = change mapping
fork copies mapping
```

EXEC MOST IMP ENTIRE LINUX

```cpp
#include "shell.hpp"
#include <unistd.h>
#include <sys/wait.h>
#include <iostream>
void Shell::execCommand(const vector<string>& tokens)
{
   pid_t pid = fork();

   if(pid == 0) // child
   {
      vector<char*> argv;

      for(const auto& t : tokens)
         argv.push_back(const_cast<char*>(t.c_str()));

      argv.push_back(nullptr);   // ⭐ VERY IMPORTANT

      execvp(argv[0], argv.data());

      perror("exec failed");
      exit(1);
   }
   else if(pid > 0) // parent
   {
      waitpid(pid, nullptr, 0);
   }
   else
   {
      perror("fork failed");
   }
}
```

A.cpp contains

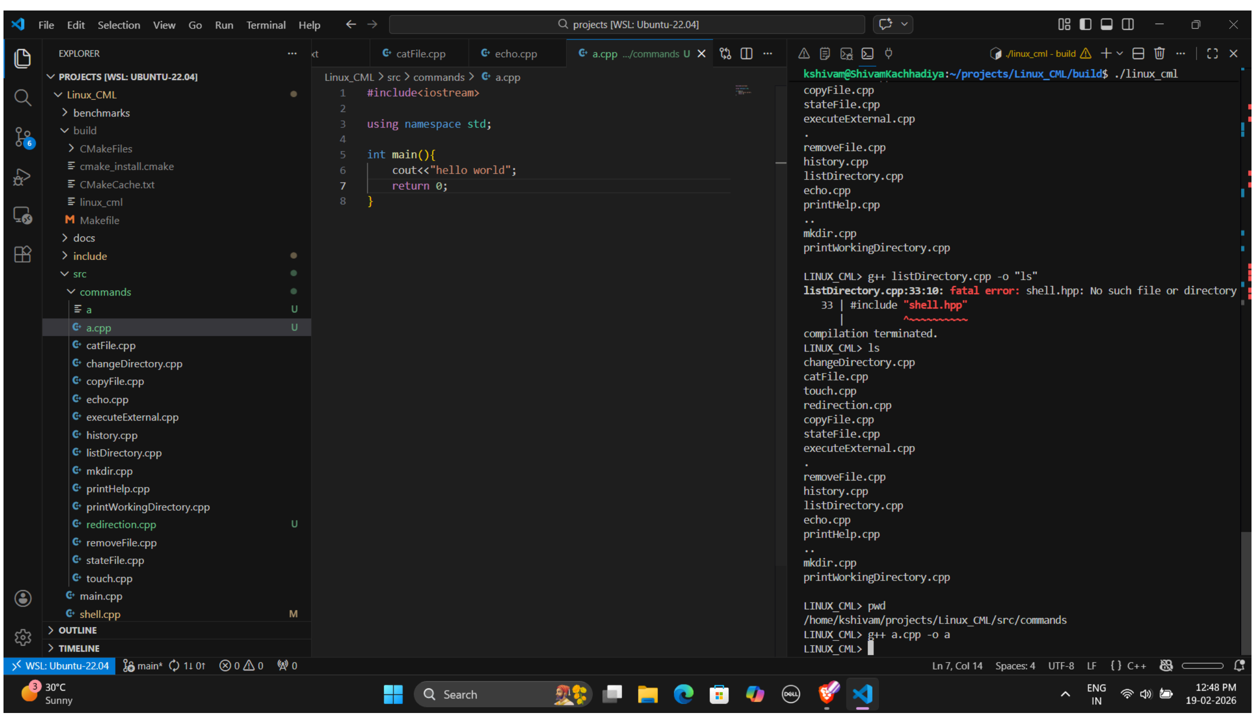
```
#include<iostream>

using namespace std;

int main(){
    cout<<"hello world";
    return 0;
}
```

Making .exe file

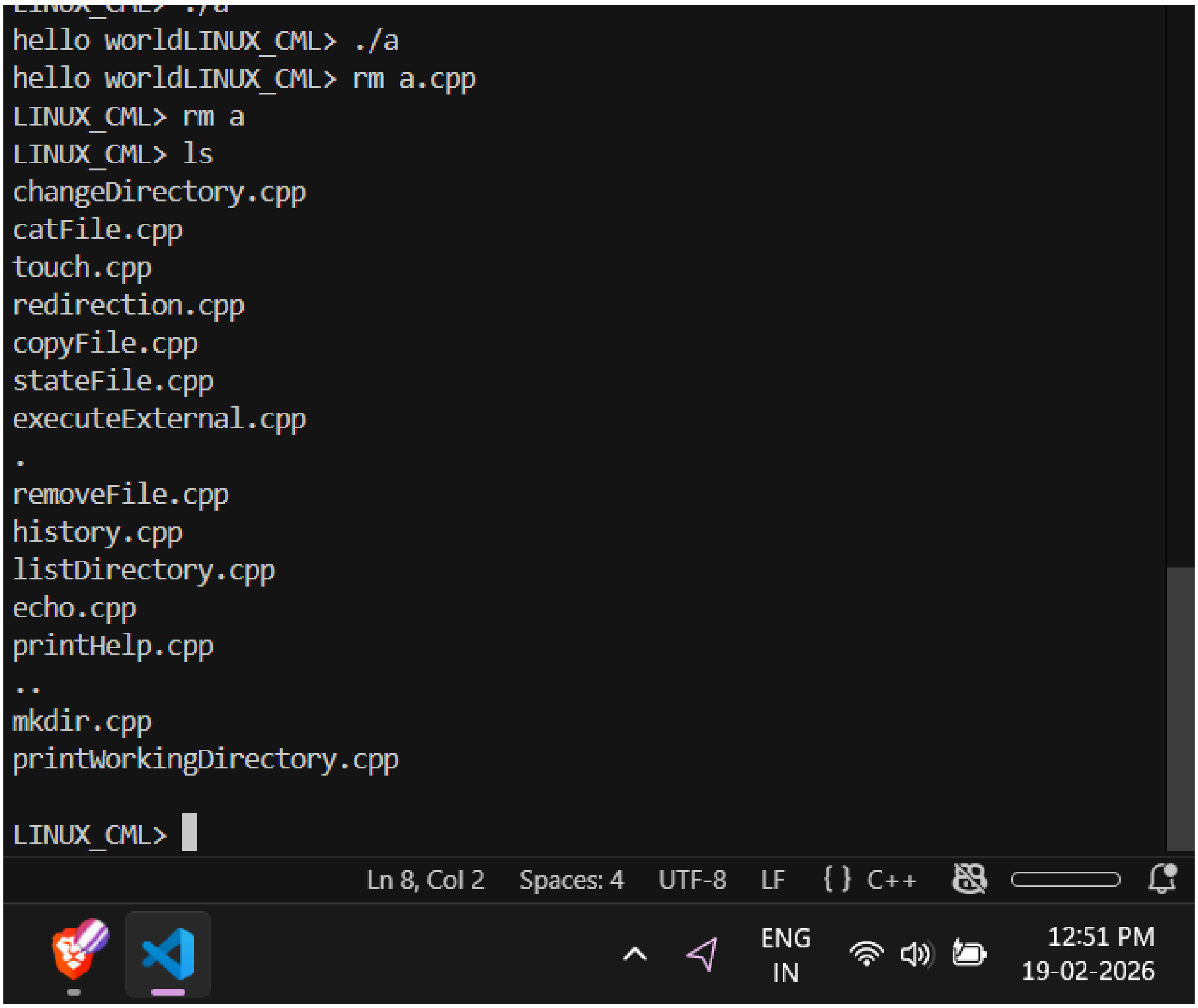
```
LINUX_CML> pwd
/home/kshivam/projects/Linux_CML/src/commands
LINUX_CML> g++ a.cpp -o a
LINUX_CML> 
```

Print output

```
LINUX_CML> ./a
hello worldLINUX_CML> ./a
hello worldLINUX_CML> 
```



Remove a.cpp & a.exe file via rm



🔥 FINAL PROCESS DIAGRAM

```
Shell
   |
   |  fork
   |
   +---- Parent ---- wait ---- continue
   |
   +---- Child ---- exec(ls) ---- run ls ---- exit
```

🔥 Kernel Level Diagram

```
fork → duplicate process
exec → replace memory
wait → synchronize
```

🔥 Interview Ready Notes (Memorize)

Shell execution pattern

```
fork → exec → wait
```

fork

creates child

exec

replaces program

wait

prevents zombie

exec never returns on success

argv must end with NULL

🔥 STEP 1 — Compile a.cpp

You type

```
LINUX_CML> g++ a.cpp -o a
```

Step 1

```
pid_t pid = fork();
```

Kernel:

```
create child process
```

Now:

```
Parent (shell)
Child  (copy of shell)
```

Step 2 (child)

Build argv:

```
{"g++","a.cpp","-o","a", NULL}
```

Step 3

```
execvp("g++", argv)
```

Kernel:

1. search PATH

```
/usr/bin/g++
```

2. load binary
3. replace child memory

Now child becomes:

```
g++ process
```

Shell code gone.

Step 4

g++ runs:

Internally:

```
open("a.cpp")
read()
compile
write("a")
```

Creates:

```
a  (executable)
```

Step 5

Child exits

Parent:

```
waitpid()
```

Shell resumes.

🔥 Now STEP 2 — Run program

You type

```
LINUX_CML> ./a
```

🔥 Again → mapping to YOUR code

Shell sees:

```
cmd = "./a"
```

Not builtin → goes to:

```
execCommand()
```

🔥 FULL INTERNAL FLOW

---

tokens

```
{"./a"}
```

---

fork()

Creates:

```
parent shell
child copy
```

---

child

execvp("./a")

Kernel:

1. open file `./a`
2. check executable permission
3. load ELF binary
4. replace memory
5. jump to main()

Now child is YOUR PROGRAM

```
main() of a.cpp runs
```

Example:

If a.cpp:

```
int main(){
    cout<<"Hello\n";
}
```

Output:

```
Hello
```

---

child exits

Parent:

```
waitpid()
```

Shell prompt returns.

🔥 Visual Diagram (VERY CLEAR)

Compile

```
shell
  └─ fork
       └─ exec(g++)
            └─ creates "a"
```

Run

```
shell
  └─ fork
       └─ exec(./a)
            └─ runs main()
```

🔥 Syscalls happening (real kernel view)

When you run:

```
./a
```

Kernel sequence:

```
fork
execve
open binary
mmap code
set stack
start main
write(stdout)
exit
waitpid
```

🔥 Tier-1 (MUST KNOW – asked everywhere)

✅ Q1 — How does a shell execute `ls -l` internally?

Answer

```
1. fork()
2. child → execvp("ls", argv)
3. parent → waitpid()
```

fork creates child, exec replaces child with ls, parent waits.

✅ Q2 — Difference between fork() and exec()?

Answer

fork	exec
creates new process	replaces current process
new PID	same PID
copies memory	loads new program

fork → duplicate
exec → replace

✅ Q3 — Does exec create a new process?

Answer

❌ No

It only **replaces memory of current process**.
PID remains same.

✅ Q4 — Why fork before exec?

Answer

If shell directly calls exec, shell itself gets replaced.

fork allows:

- child runs program
- parent shell survives

✅ Q5 — Why argv must end with NULL in exec?

Answer

exec doesn't know argument count.

NULL marks end of arguments.

Without it → undefined behavior.

✅ Q6 — What does execvp do?

Answer

- v → vector args
- p → search PATH

Searches command in PATH and executes.

✅ Q7 — What happens if exec succeeds?

Answer

It **never returns**.

Process memory replaced.
Next instruction is new program's `main()`.

✅ Q8 — Why waitpid() needed?

Answer

Prevents zombie processes.

Collects child exit status.

✅ Q9 — What is zombie process?

Answer

Child finished but parent didn't wait.

Kernel keeps entry in process table.

State: Z

✅ Q10 — What is returned by fork()?

Answer

return	meaning
0	child
>0	parent (child pid)
-1	error

🔥 Tier-2 (Intermediate / System design interviews)

✅ Q11 — What is copy-on-write in fork?

Answer

Memory is NOT copied immediately.

Parent & child share pages.

Copy happens only when one writes.

Improves performance.

✅ Q12 — What happens to file descriptors after fork?

Answer

Child inherits all FDs.

They point to same open file description.

Offsets are shared.

✅ Q13 — Why is exec fast even for large programs?

Answer

Because it replaces memory instead of copying.

Loads only needed pages (demand paging).

✅ Q14 — Difference between execv and execvp?

Answer

execv	execvp

needs full path	searches PATH

✅ Q15 — Why use vector + data() for exec?

Answer

exec needs `char**`.

vector is C++ container.

`.data()` gives raw pointer.

✅ Q16 — Can parent and child run simultaneously after fork?

Answer

Yes.

Scheduler decides order.

No guaranteed sequence.

✅ Q17 — What happens if parent exits before child?

Answer

Child becomes orphan.

Adopted by init/systemd.

✅ Q18 — Why is fork expensive?

Answer

- page table copy
- kernel structures
- TLB flush

But mitigated by copy-on-write.

redirection command imp

```cpp
#include "shell.hpp"
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <iostream>
#include <vector>
using namespace std;
void Shell::execRedirect(const vector<string>& tokens)
{
    int pos = -1;
    bool append = false;

    // find > or >>
    for(int i = 0; i < tokens.size(); ++i)
    {
        if(tokens[i] == ">" || tokens[i] == ">>")
        {
            pos = i;
            append = (tokens[i] == ">>");
            break;
        }
    }

    if(pos == -1 || pos + 1 >= tokens.size())
    {
        cout << "syntax error\n";
        return;
    }

    string filename = tokens[pos + 1];  //ls > ab.txt

    pid_t pid = fork();

    if(pid == 0)
    {
        int fd = open(
            filename.c_str(),
            append ? (O_WRONLY | O_CREAT | O_APPEND)
                   : (O_WRONLY | O_CREAT | O_TRUNC),
            0644
```

```
        );

        if(fd < 0)
        {
            perror("open");
            exit(1);
        }

        dup2(fd, STDOUT_FILENO);
        close(fd);

        vector<char*> argv;

        for(int i = 0; i < pos; ++i)
            argv.push_back(const_cast<char*>(tokens[i].c_str()));

        argv.push_back(nullptr);

        execvp(argv[0], argv.data());

        perror("exec failed");
        exit(1);
    }
    else
    {
        wait(nullptr);
    }
}
```

What does `>` actually mean?

When you type:

```
ls > out.txt
```

You are NOT telling `ls` to write to file.

👉 You are telling **shell**:

```
"Before running ls, redirect stdout to file"
```

So:

```
shell changes FD 1 → file
then runs ls normally
```

🔥 Core trick (remember forever)

```
stdout = FD 1
dup2(file_fd, 1)
```

Now anything printed → file

🔥 Let's walk your code line-by-line

🔹 Function header

```
void Shell::execRedirect(const vector<string>& tokens)
```

Input example:

```
ls > out.txt
```

tokens:

```
{"ls", ">", "out.txt"}
```

🔥 Step 1 — find redirection operator

```
int pos = -1;
bool append = false;
```

Meaning:

```
pos    → index of > or >>
append → whether >>
```

```
for(int i = 0; i < tokens.size(); ++i)
```

Loop through tokens.

```
if(tokens[i] == ">" || tokens[i] == ">>")
```

Check:

```
>   overwrite
>>  append
```

```
pos = i;
append = (tokens[i] == ">>");
break;
```

Store:

Example:

```
tokens = {"ls", ">", "out.txt"}

pos = 1
append = false
```

🔥 Step 2 — syntax check

```
if(pos == -1 || pos + 1 >= tokens.size())
```

Cases:

❌ no >
❌ missing filename

Examples:

```
ls >
```

```
cout << "syntax error\n";
return;
```

Stops execution.

🔥 Step 3 — filename

```
string filename = tokens[pos + 1];
```

Example:

```
"out.txt"
```

🔥 Step 4 — fork()

```
pid_t pid = fork();
```

Why fork?

Same as always:

👉 child executes command
👉 parent shell survives

🔥 Child process (IMPORTANT PART)

```
if(pid == 0)
```

Everything inside:

👉 child only

🔥 Step 5 — open file

```
int fd = open(
    filename.c_str(),
    append ? (O_WRONLY | O_CREAT | O_APPEND)
           : (O_WRONLY | O_CREAT | O_TRUNC),
```

```
    0644
);
```

Two modes

>

`O_TRUNC`

clear file

>>

`O_APPEND`

add to end

Permissions

`0644 → rw-r--r--`

Now:

`fd = 3`

🔥 Step 6 — dup2 (MOST IMPORTANT LINE)

```
dup2(fd, STDOUT_FILENO);
```

STDOUT_FILENO = 1

So:

```
dup2(3, 1)
```

What dup2 does?

```
FD 1 now points to same file as FD 3
```

🔥 FD table BEFORE dup2

```
0 → stdin
1 → terminal
2 → terminal
3 → out.txt
```

🔥 AFTER dup2

```
0 → stdin
1 → out.txt  ⭐⭐⭐
2 → terminal
3 → out.txt
```

Now:

👉 printf / cout / write(1) → file

```
close(fd);
```

Why?

Because:

```
1 already points to file
3 not needed
```

🔥 Step 7 — build argv

```
vector<char*> argv;
```

```
for(int i = 0; i < pos; ++i)
    argv.push_back(...)
```

Only before >.

Example:

```
{"ls"}
```

Not including > or filename.

```
argv.push_back(nullptr);
```

Mandatory for exec.

🔥 Step 8 — exec

```
execvp(argv[0], argv.data());
```

Child becomes:

```
ls
```

But:

👉 stdout already redirected

So:

```
ls → prints → FD 1 → file
```

🔥 Step 9 — parent waits

```
wait(nullptr);
```

Shell waits.

🔥 FULL DRY RUN

Command:

```
ls > out.txt
```

Step-by-step

1. parse

`{"ls", ">", "out.txt"}`

2. fork

`parent + child`

3. child open file

`fd=3`

4. dup2

`1 → out.txt`

5. exec ls

`ls prints → goes to file`

6. file content:

```
file1
file2
file3
```

🔥 Visual Diagram

Normal

```
ls → stdout → terminal
```

With redirection

```
ls → stdout → file
```

🔥 Kernel level flow

```
fork
open
dup2
execve
write(1,...)
```

🔥 SUPER IMPORTANT Concepts

---

stdout = 1

stderr = 2

stdin = 0

---

dup2 rule

```
dup2(old, new)
```

means:

```
new → same file as old
```

🔥 Interview Questions (High ROI)

Q1

How does `>` work internally?

👉 open file + dup2(fd,1) + exec

Q2

Why dup2 used?

👉 replace stdout

Q3

Difference between > and >>?

👉 truncate vs append

Q4

Why fork before redirect?

👉 so shell’s stdout not changed

Q5

What happens if you dup2 before fork?

👉 shell output redirected too (wrong)

Q6

What is STDOUT_FILENO?

👉 1

Q7

What if close(fd) removed?

👉 still works but FD leak

Q8

How implement `2>` ?

👉 dup2(fd, STDERR_FILENO)

Q9

How implement `< input.txt` ?

👉 dup2(fd, STDIN_FILENO)

Q10

How implement `ls > out 2>&1` ?

👉 dup2(1,2)