**C-Shell**

**1. Display Requirement**

✅ 1. Objective Understanding

Start by stating the specification goal:

“The prompt should dynamically show the username, system name, and current working directory. If the current directory is inside the shell’s home directory (i.e., where the shell was first launched), the prompt should display the relative path with a ~. Otherwise, it should show the absolute path.”

✅ 2. Dynamic Shell Prompt Format

Mention the format and how you adhered to it:

* Format: <Username@SystemName:CurrentDirectory>
* Example: <JohnDoe@SYS:~> when in home directory
* Example: <JohnDoe@SYS:/home/johndoe/sem3> when outside

✅ 3. Key Components Used

📌 a. Getting Current User

getlogin\_r(user\_name, buf\_size);

* Retrieves the current user in a thread-safe way.
* Fallback using getenv("USER") in case getlogin\_r() fails.

📌 b. Getting Hostname

uname(&os);

* uname() fills a struct utsname with system details.
* os.sysname is used to get the system name.

📌 c. Getting Current Directory

getcwd(dir, buf\_size);

* Gets the absolute path of the current working directory.
* This is later compared with home\_dir to decide whether to use ~ or the full path.

✅ 4. Handling Relative and Absolute Path Logic

You should explain this logic clearly:

if (strncmp(dir, home\_dir, strlen(home\_dir)) == 0)

{

if (strlen(dir) == strlen(home\_dir))

strcpy(rel\_dir, "~");

else

sprintf(rel\_dir, "~%s", dir + strlen(home\_dir));

}

else

{

strcpy(rel\_dir, dir); // absolute path if outside

}

This conditional checks whether the current directory starts with the home directory path. If yes, it's inside home — and we replace that portion with ~. Otherwise, we show the full path.

This shows you understand path manipulation and string handling.

✅ 5. Color Formatting (Optional but Valuable)

printf("\033[0;35m<%s@%s", user\_name, os.sysname);

printf("\033[0;33m:%s", rel\_dir);

ANSI escape codes are used to colorize different parts of the prompt (username, system, and directory) for better user experience.

✅ 6. Memory and File Handling

You can highlight:

* Dynamic memory allocation for strings (with malloc).
* Prompt behavior is affected by the contents of test.txt — possibly used for indicating if a previous command took too long.
* File opened safely using fopen() and closed after use.

✅ 7. Edge Case Handling

* If getlogin\_r fails → fallback with environment variable or default to "user".
* Handles scenario when current directory is exactly the home directory.
* Prompts update correctly when directory changes via cd.

✅ 8. Mention Design Principle

"I ensured the prompt is context-aware and reacts dynamically to directory changes, keeping the user informed and the shell user-friendly — just like professional Unix shells do."

**2. Input Requirement**

"This specification outlines how the shell should accept and process user input, focusing on multi-command handling, whitespace normalization, background process execution, and proper error messaging."

**✅ 1. Command Delimiters (; and &):**

* The shell must **support multiple commands** on a single line, separated by:
  + ; → Sequential execution (foreground).
  + & → Concurrent execution (background).

**✅ Example:**

bash

<JohnDoe@SYS:~> sleep 5 ; echo Hello

* sleep 5 executes in the **foreground**.
* Then echo Hello is executed **after it finishes**.

bash

<JohnDoe@SYS:~> sleep 5 & echo Hello

* sleep 5 runs in the **background** (shell continues immediately).
* echo Hello runs **in the foreground**.

**✅ 2. Random Whitespace/Tab Handling:**

* The shell should **gracefully handle irregular spacing** (tabs, spaces).
* Commands like:

bash

ls -l ; pwd

or

bash

sleep 5 & echo "Done"

should be parsed **exactly as valid input** — no crashing or misinterpretation.

✔️ This is handled in code using strtok (with \t delimiters), and further cleaned with trimWhitespaces().

**✅ 3. Background Process Execution (&):**

* When & is used:
  + The **preceding command is executed in the background**.
  + The shell must **immediately print the PID** of the spawned background process.

**Example:**

bash

<JohnDoe@SYS:~> vim &

[1] 35006

* PID 35006 is the background vim process.
* Shell prompt remains responsive for new input.

✔️ Background processes are tracked in a structure (activities) and a separate map (process\_map) for PID-to-command mapping.

**✅ 4. Foreground vs Background Execution Order:**

* Even if multiple commands are combined:
  + & should fork a background process.
  + Remaining commands must execute in the **foreground in order**.

**Example:**

bash

<JohnDoe@SYS:~> sleep 5 & echo "Lorem ipsum"

* sleep 5 → background ([2] 35036)
* echo "Lorem ipsum" → immediate foreground output

Then (after 5s):

bash

sleep with pid 35036 exited normally

**✅ 5. Error Handling for Invalid Commands:**

* If a command does not exist (e.g., sleeeep instead of sleep), the shell should:
  + Print a user-friendly **error message**:

ERROR : 'sleeeep' is not a valid command

✔️ This is achieved by trying to execute the command using execl() or execvp(). On failure, perror() or custom error messages are printed.

**🧠 Key Implementation Notes:**

* Commands are **tokenized** using strtok\_r() to distinguish between ;, &, and the actual commands.
* Each command is cleaned of whitespace with trimWhitespaces() before execution.
* Background and foreground commands are managed separately:
  + Background: background\_process() forks and detaches the process.
  + Foreground: foreground\_process() forks, waits, and tracks duration.
* Signals (SIGINT, SIGTSTP, SIGCHLD) are handled to manage process lifecycles and clean up zombies.

**🔚 Summary:**

“In short, this specification ensures that our shell behaves like a real UNIX shell — handling multiple commands, launching background jobs with PID feedback, dealing with user spacing errors, and gracefully handling invalid inputs.”

**3. Hop:**

The hop command is a custom shell built-in that allows the user to change the current working directory (like cd in bash), while printing the new working directory after each change.

✅ Functional Requirements

1. Basic Directory Change:
   * hop <path> changes the current directory to the specified path.
   * It prints the absolute path of the new directory.
2. Special Flags Supported:
   * hop ~ → Changes to the shell's home directory.
   * hop - → Switches to the previous working directory.
   * hop . → Current directory (no change).
   * hop .. → Parent directory.
3. Multiple Arguments:
   * hop dir1 dir2 executes hop on dir1 first, then on dir2 sequentially.
4. No Arguments:
   * hop alone changes to the home directory.
5. Path Types:
   * Supports absolute paths (like /home/user/folder).
   * Supports relative paths (like ../test).
   * Supports paths relative to home (like ~/project).

🧠 Implementation Overview

📌 File: hop.c

* The command is parsed using strtok() to separate arguments.
* For each argument:
  1. Determine the target path:
     + If ~, set path to home\_dir.
     + If -, set path to prevDir.
     + If ., .., or any relative/absolute path, construct the full path appropriately.
  2. Use chdir() to change the directory.
     + If chdir() fails, perror() prints the error.
     + If it succeeds, update prevDir and print the new working directory with getcwd().

📌 Important buffers used:

* curr\_dir: Stores current working directory before change.
* prevDir: Stores last directory visited (used for - flag).
* new\_path: Dynamically built full path for chdir() based on argument and context.

🚫 What’s NOT Used

* No usage of execvp, system() or external programs — instead, the directory is changed using chdir(), making this a built-in command handled directly within the shell.

📦 Edge Cases Handled

* If the path is invalid → perror() shows the error.
* If no argument → default to home directory.
* Multiple arguments → handled one after another in order.

🧩 Why It's Important in a Shell

* Mimics core shell behavior (cd), but with added flexibility.
* Supports multiple flags and chaining, useful in scripting or interactive use.
* Respects shell state (e.g., remembers previous dir like bash does with cd -).

**4. Reveal:**

"**The reveal command is a custom implementation similar to the ls command in Bash. It lists directory contents with optional flags for detailed info and hidden files, while supporting special path symbols and colored output.**"

**✅ 1. Command Syntax & Argument Structure**

The command follows this format:

bash

Copy code

reveal <flags> <path>

* **Flags** can be any combination of:
  + -a → show **all files**, including hidden ones (. prefix)
  + -l → show **detailed info** similar to ls -l (permissions, size, time, etc.)
* We handle various flag formats:
  + Separate: -a -l
  + Combined: -al, -la
  + Repeated: -lala, -aal, -lllaaa — all valid and interpreted correctly.
* If **no path is given**, default to **current directory**.
* Only **one path** is given at a time — no need for multiple arguments parsing.

**✅ 2. Path Handling**

We support:

* . → current directory
* .. → parent directory
* ~ → home directory
* - → previous working directory (tracked internally)
* Both **absolute and relative** paths

✔️ Internally, we normalize these inputs into an absolute path using functions like realpath() or manual path resolution logic.

**✅ 3. File Listing Behavior**

* Files are listed in **lexicographic (sorted) order**.
* We use system calls like:
  + opendir() and readdir() to list directory entries.
  + stat() to get metadata for -l flag.
* By default, reveal does **not show hidden files**.
  + Unless -a is set → includes entries starting with .

**✅ 4. Long Listing (-l) Output**

When -l flag is passed:

* Output mimics ls -l format:

-rw-r--r-- 1 user group 1234 Jan 01 10:00 filename.txt

drwxr-xr-x 2 user group 4096 Jan 02 12:00 folder/

* For this, we extract:
  + File type and permissions using st\_mode bits
  + Owner/group with getpwuid() and getgrgid()
  + Size and timestamps (st\_size, st\_mtime)
  + Number of hard links (st\_nlink)
* Timestamps formatted using strftime() for readability.

**✅ 5. Color Coding of Output**

We visually distinguish files using ANSI escape codes:

* 🔵 **Directories**: Blue
* ⚪ **Regular files**: White
* 🟢 **Executables**: Green (detected via S\_IXUSR in st\_mode)

**✅ 6. Error Handling & Robustness**

* If a path is invalid, reveal prints a friendly error:

lua

Copy code

ERROR: Invalid path or permission denied.

* Edge cases like:
  + Empty directories
  + No permission to read entries
  + File vs directory differentiation  
    Are handled gracefully using stat() and proper checks.

**✅ 7. Implementation Constraints**

🔒 **Do not use execvp() or system calls like system("ls")**.

* Everything must be done via **direct system calls and library functions**, ensuring:
  + Full control over behavior
  + No dependency on external programs
  + Portable and efficient shell behavior

**🔚 Summary**

“In short, the reveal command is a clean reimplementation of ls with support for flags -l and -a, hidden file visibility, metadata printing, and color-coded output. It's designed to be intuitive, standards-compliant, and efficient — all built using system-level functions without relying on exec family calls.”

**5. Log:**

**Goal of Specification**

Implement a log command like Bash history but with custom rules:

* Max 15 commands stored persistently across sessions.
* Do **not** log repeated commands (if same as previous).
* Do **not** log any command containing "log" (e.g., log, log execute, log purge, or even echo hello ; log).
* Store compound commands like cmd1 ; cmd2 in one line.
* Add support for:
  + log → display
  + log purge → clear log
  + log execute <index> → run command at that index from most recent

**✅ Flow of Implementation**

**1. 📁 Log Storage Design**

* Stored in a file: log.txt (in user's home dir).
* Persistent across sessions.
* Maintains **FIFO** (First In, First Out) behavior when >15.

**2. 🧠 Where and When Logging Happens**

Inside main():

c

Copy code

if (strcmp(input, "\n") != 0)

log\_Write(input, logFileName);

* After reading and trimming user input, log\_Write() is called.

**✨ Key Conditions in log\_Write()**

**(a) Rejecting:**

* Empty input.
* Identical to last command (using strcmp).
* Contains "log" anywhere (like sleep 5 ; log).

**(b) Accepting:**

* Non-empty.
* Different from last command.
* Doesn’t include "log".

**(c) Capacity Handling:**

* If < 15 commands → directly append to file.
* If == 15, remove the oldest (first line) and append new one using logMoreThan15().

**3. 🧹 log purge Flow**

Called via:

c

Copy code

logPurge(logFilename);

* Opens log file in "w" mode — truncates it (clears content).
* Does **not** log the log purge command itself.

**4. 📜 log → Print Log Flow**

* Opens file in read mode and prints line by line using fgets().

**5. ▶️ log execute <index> Flow**

**a. Parse index**

* Tokenizes input string (e.g., log execute 3) → extract index.

**b. Compute Actual Line Number**

c

Copy code

in = numC - index + 1;

* Since we show output **oldest to newest**, but indexing is **from most recent**, we calculate position accordingly.

**c. Fetch and Run Command**

* Fetch that specific line from log.txt.
* Send it through dividingCommands() to re-execute it.
* **Does not** log this execution (per spec, optional logging is not implemented).

**⚠️ Edge Cases Handled**

| **Case** | **Handling** |
| --- | --- |
| log repeated twice | Not stored (duplicate) |
| log ; echo "test" | Entire input skipped (contains "log") |
| >15 commands | Oldest removed, new appended |
| log execute 1000 | Shows "Invalid index" |
| Empty command | Not stored |
| log purge | Clears log and is not stored |

**🧪 Example**

1. sleep 5 ; reveal → stored as one line.
2. log → outputs all commands, top to bottom (oldest to latest).
3. log execute 1 → runs **latest** command (e.g., line 3).
4. log execute 3 → runs **oldest** command (e.g., line 1).
5. log purge → clears log.

**✅ Design Choices Justified**

* File-based storage = persistence across sessions.
* String comparison (strstr) for "log" avoids accidental inclusion.
* Efficient overwrite using temp.txt and re-write strategy.
* Keeps log behavior consistent even with compound commands (;, &).

**🧩 Modular Breakdown**

| **Function** | **Purpose** |
| --- | --- |
| log\_Write() | Main entry for logging user input |
| logMoreThan15() | Removes oldest entry and appends new |
| logPurge() | Clears log file |
| logPrint() | Displays all entries |
| log\_fun() | Handles log, log execute, log purge logic |
| numberOfCommands() | Helper to count current log size |

**📌 How to Conclude in Interview**

"In summary, I built a robust, persistent command history system that respects user intent, avoids redundant or sensitive entries, and supports retrieval and re-execution — all while being memory-efficient and session-persistent through a flat file-based approach."

**6. System Commands:**

This specification ensures that the shell implementation supports executing **external system commands** (like emacs, gedit, sleep, etc.), both in **foreground** and **background** modes — similar to a standard Bash shell.

**🔹 Foreground Process Execution**

* **Behavior**:
  + When a user runs a command **without &**, it's treated as a **foreground process**.
  + The shell **waits** for the process to **complete** before accepting the next command.
  + During this time, **terminal control** is transferred to the child process.
* **Additional Feature**:
  + If a foreground process runs for **more than 2 seconds**, the **time (in seconds)** and the **command name** are printed in the **next prompt**.
    - Example:

ruby

Copy code

<User@Shell:~> sleep 5

<User@Shell:~ sleep : 5s>

* **Implementation Highlights**:
  + We record the **start and end time** of the foreground process using time(NULL).
  + If duration > 2, we write the info to a file like time.txt and read from it in the next prompt display.
  + We handle SIGINT and SIGTSTP gracefully and restore terminal control after the process finishes using tcsetpgrp.

**🔹 Background Process Execution**

* **Behavior**:
  + Any command **ending with &** is executed in the **background**.
  + The shell **does not wait** for it; it continues to take user input immediately.
  + **PID** of the background process is printed when it's started.
  + When the process **finishes**, a message like the following is shown **automatically**:

scss

Copy code

Background process 13027 (sleep) ended normally

* **Implementation Strategy**:
  + We use fork() to create a child process, and in the child, we execl("/bin/sh", "sh", "-c", command, NULL) to run the command.
  + We use sigaction() to register a **signal handler for SIGCHLD**, which is triggered when a child (background) process terminates.
  + In the handler (handle\_sigchld()), we use waitpid(-1, &status, WNOHANG) to reap zombie processes.
  + We then print whether the process exited **normally** (WIFEXITED) or **abnormally** (WIFSIGNALED), along with its **command name** (stored in a PID-to-command map).

**🔸 Special Notes**

* **Custom commands** like hop, log, reveal, etc., are always run in the **foreground only** — no background support is required for them.
* Shell supports **multiple background processes**, and all are handled concurrently with autonomous messages printed upon completion.

**📌 Summary (for closing in interview):**

This spec is about ensuring a real shell-like experience. We distinguish between foreground and background processes, transfer terminal control accordingly, measure execution time for long-running foreground tasks, and handle asynchronous background process completion via signals — all to create a robust, interactive shell environment.

**1. Foreground Process Execution**

**File:** systemCommands.c  
**Function:** void foreground\_process(char \*command, char \*home\_dir)

**✅ Responsibilities:**

* Forks a new process
* Gives **terminal control** to child-tcsetpgrp()
* Waits until child terminates
* Records time taken by process-time\_t.
* Logs if time > 2s for prompt display

**2. Background Process Execution**

**Function:** void background\_process(char \*command)

**✅ Responsibilities:**

* Forks child and runs command in background
* Registers SIGCHLD handler (handle\_sigchld) to detect completion
* Prints message with PID & status on exit

**3. SIGCHLD Handler**

**Function:** handle\_sigchld(int sig)

**✅ Responsibilities:**

* Handles termination of background children
* Checks if exited normally or abnormally
* Uses waitpid(-1, &status, WNOHANG) to reap multiple zombies

| **Concept** | **How You Handled It** |
| --- | --- |
| Background/Foreground distinction | Using & in parser and fork() logic |
| Terminal control (foreground) | tcsetpgrp, SIGTTIN/SIGTTOU |
| Execution timing | time() around waitpid() |
| Asynchronous notifications | sigaction(SIGCHLD, handler) |
| Zombie cleanup | waitpid(-1, ..., WNOHANG) in loop |
| User feedback | Prompt update and background completion message |

**7. proclore:**

Yes, sure. I implemented a custom command called proclore in my shell, inspired by how process introspection works in Linux via the /proc filesystem.

**🔍 Objective of proclore:**

The proclore command is used to **display detailed information about a specific process**. If no PID is provided, it defaults to showing information about the current shell process.

It displays the following attributes:

1. **PID** – Process ID.
2. **Process Status** – State of the process (like Running R, Sleeping S, Zombie Z). We also add a + to indicate if the process is running in the **foreground**.
3. **Process Group ID** – Useful for job control and terminal handling.
4. **Virtual Memory Usage** – Memory usage of the process.
5. **Executable Path** – The absolute path to the binary the process is executing.

**🛠️ How I Implemented It:**

I used the /proc/[pid]/stat and /proc/[pid]/exe files to extract information:

* **/proc/[pid]/stat**:
  + It contains the process status (3rd field) and virtual memory usage (23rd field).
  + I used fscanf() to parse these fields, skipping irrelevant ones using format specifiers.
* **/proc/[pid]/exe**:
  + It's a symbolic link to the actual binary.
  + I used readlink() to resolve the absolute path of the executable.
* **Process group**:
  + Fetched using getpgid(pid).
* If proclore is run without a PID, I call getpid() to get the current shell’s PID and show its details.

**🔐 Edge Handling:**

* If an invalid PID is passed (e.g., process doesn’t exist), I gracefully handle it with an error message using perror() or validation after fopen().
* If readlink() fails, I print "Executable path: Not found".

**🔄 Why This Is Useful:**

* Mimics the behavior of tools like ps, top, or pmap in a simplified form.
* Teaches how /proc can be used for lightweight introspection without using external libraries.
* Helps with debugging or learning more about child/foreground/background process behavior.

**🧠 Additional Insight (if time allows):**

I also took care to:

* Use safe buffers (snprintf, bounds-checked readlink).
* Separate logic into two functions: one for the shell itself (proclore()), and one for arbitrary PIDs (procloreID()).

**8. Seek:**

The seek command is a custom implementation for searching files or directories recursively from a given path (or current directory if unspecified). It supports filtering, exact/prefix matching, and conditional execution.

**🔧 Functionality**

* **Basic Usage:**  
  seek <flags> <filename> [target\_directory]
* **Search Behavior:**
  + Recursively searches the directory tree.
  + Matches files or directories that **exactly match or start with the given name**.
  + Returns **relative paths** from the target directory.

**🏷️ Supported Flags**

| **Flag** | **Function** |
| --- | --- |
| -d | Search **only directories**. |
| -f | Search **only files**. |
| -e | **Execute behavior** if only one match is found: |
|  | - If one file: print contents (if readable). |
|  | - If one directory: chdir() to it (if executable). |

* **Mutually exclusive:** -d and -f cannot be used together — prints "Invalid flags!".

**🧪 Behavior Examples**

* seek file → Search in current directory tree.
* seek -f file ./path → Search only for files in ./path.
* seek -e -f file → If only one match found, print file contents.
* seek -d file ./x → Only search directories.
* If no matches found → "No match found!"

**🔍 Implementation Highlights**

**✅ 1. Parsing the command**

* strtok() is used to extract flags, filename, and optional directory.
* Flags are counted and validated.
* Handles Invalid flags! and missing cases gracefully.

**✅ 2. Resolving Target Directory**

* Builds the full path using home\_dir + target\_directory.
* If no directory is given, current directory is used (home\_dir passed from shell).

**✅ 3. Recursive Search – seek\_fun()**

* **opendir()** opens the directory stream.
* **readdir()** iterates entries.
* **stat()** fetches metadata to check:
  + If it’s a file or directory using S\_ISDIR, S\_ISREG.
  + File permissions using access() for X\_OK (execute), R\_OK (read).
* **Recursion** is used to explore directories.

**✅ 4. Matching Logic**

* If flags aren't set, searches both files and directories.
* compare\_s() helps in relaxed matching by splitting filenames by . and comparing tokens (e.g., test.txt matches test).
* Color output:
  + Files in **green** (\033[0;32m)
  + Directories in **blue** (\033[0;34m)

**🧠 -e Flag Execution Logic**

* **Conditionally triggers behavior** if there's exactly:
  + One **file match** → print its contents (with fopen(), fgets()).
  + One **directory match** → chdir() into it.
* If permissions are missing:
  + Outputs: "Missing permissions for task!"

**🧱 Robust Error Handling**

* Detects inaccessible directories using opendir().
* Gracefully handles stat() failures.
* Validates flag combinations (-d -f).
* Tracks counts (numFiles, numDir) to help decide on -e execution.

**⚙️ System Calls & APIs Used**

* opendir(), readdir(), closedir() – directory traversal
* stat() – get file metadata
* access() – permission check
* chdir() – change working directory
* fopen(), fgets() – file reading

**📌 Design Considerations**

* Modularity: seek() handles parsing, seek\_fun() handles recursion.
* Portability: Works with symbolic paths like ~, ., etc., assuming reveal helper handles expansion.
* UX: Colored output helps distinguish file types.
* Security: Checks permissions before reading or changing directories.

**📘 Conclusion**

The seek command is a robust shell utility for file and directory discovery with intelligent filtering and controlled execution. It demonstrates knowledge of:

* Filesystem traversal
* System-level calls
* Permissions & error handling
* Recursive algorithm design
* Shell UX best practices (colors, relative paths)

**9. myshrc() (aliases and functions):**

**Objective**:  
The .myshrc file in our custom shell acts similarly to .bashrc in Bash. It's designed to enhance the shell experience by allowing:

* Definition of **aliases** (shortcut commands)
* Creation of **custom shell functions** (like mk\_hop, hop\_seek)  
  This helps automate workflows and shorten commonly used commands.

**🔹 Alias Implementation (2 marks)**

**✅ What is an alias?**

An alias maps a short keyword to a longer command.

**✅ Examples we support:**

bash

Copy code

reveall = reveal -l

# or

alias reveala = reveal -a

I chose to **support one format consistently** (e.g., with or without the alias keyword) to simplify parsing logic.

**✅ Implementation logic:**

* We store aliases in a global aliases[] structure with alias and actual\_func.
* During command processing, checkForAliases(input, &modifiedInput) scans this list.
* If a match is found, we **replace the alias** with the actual command before execution.
* If not found, the input proceeds unchanged.

**🔹 Bonus: Custom Shell Functions (mk\_hop, hop\_seek)**

These are **simple, one-parameter shell functions**, parsed from .myshrc and stored in a function[] array.

**✅ mk\_hop function**

bash

Copy code

mk\_hop test

# Executes: mkdir test ; hop test

* Makes a directory
* Then "hops" into it (changes directory)

**✅ hop\_seek function**

bash

Copy code

hop\_seek folder

# Executes: hop folder ; seek folder

* Changes to the directory
* Then searches for files with the same name as the directory

**✅ Handling logic**

In the shell:

c

Copy code

if (strncmp(input, "mk\_hop", 6) == 0 || strncmp(input, "hop\_seek", 8) == 0)

* Detected as a **function call**
* We use checkForFunctions() to:
  + Identify the function
  + Replace $1 with argument
  + Reconstruct the expanded command (e.g., mkdir test ; hop test)
* Then pass it to dividingCommands() to handle it like any regular command (with support for redirection, pipes, etc.)

**🔹 Key Structures and Integration**

* **Alias support** is integrated early in dividingCommands():

c

Copy code

checkForAliases(command, &alias);

* **Function support** is handled in checkForFunctions(), where:
  + We detect $1, substitute the user-supplied argument
  + Concatenate individual commands with ;
* Finally, dividingCommands() handles these commands whether they use:
  + foreground/background
  + pipes, redirection, or system commands

**🔹 Design Choices**

* Supported **only one format** for alias (alias name = command) to keep parsing clean
* Functions are **single-argument only** and **purely positional ($1)** – this simplified implementation while still being useful
* Used modular design – alias and function expansion is done **before** actual command execution so the rest of the shell stays unchanged

**🔹 Result**

This feature brings **user customization** to the shell:

* You can define shortcuts for frequently used commands
* Automate small tasks with shell functions
* Makes the shell feel more like Bash, enhancing usability

**✅ Summary for Interview**

“In my shell, I implemented .myshrc support to allow users to define aliases and simple functions. I parse aliases and replace them with their mapped commands before execution. For functions like mk\_hop, I substitute arguments like $1 and expand them into real commands like mkdir <dir> ; hop <dir>. This design makes the shell more user-friendly and scriptable.”

**10. I/O redirection:**

In this custom shell implementation, I implemented **I/O redirection** for >, >>, and < operators. These symbols allow rerouting output to files or reading input from files instead of using the terminal.

The shell supports:

* >: Redirect stdout to a file (overwrite).
* >>: Redirect stdout to a file (append).
* <: Redirect stdin from a file.
* Combined redirection: e.g., wc < input.txt > output.txt.

**🧭 Explain High-Level Flow:**

The shell command input is parsed by a central function called dividingCommands(), which checks for redirection and pipes. If redirection is detected and not part of a pipeline, the command is passed to processRedirection().

**🔍 Deep Dive into dividingCommands() Flow:**

1. **Input is tokenized** using ; and & to separate multiple commands.
2. **Each command** is checked using processCommands() to see if it contains redirection (>, <, >>) or pipes (|).
3. **If redirection is present**, the command is passed to processRedirection().
4. Depending on whether the command is background or foreground, it chooses the right function to run.

**🧠 Key Function: processRedirection()**

This is the heart of the redirection logic.

**Step-by-step:**

1. **Save original stdin/stdout** using dup() so we can restore later.

int copy\_in = dup(0), copy\_out = dup(1);

1. **Detect operator (>, >>, <)** using strstr():
   * For > → open file with O\_WRONLY | O\_CREAT | O\_TRUNC.
   * For >> → open file with O\_APPEND.
   * For < → open file with O\_RDONLY. With open() system call
2. **Redirect file descriptors**:
   * Use dup2(fd\_out, STDOUT\_FILENO) for output redirection.
   * Use dup2(fd\_in, STDIN\_FILENO) for input redirection.
   * This replaces the default stdin or stdout for the command being executed.

**Example:**

wc < input.txt > output.txt

* input.txt is opened and duped to stdin
* output.txt is opened and duped to stdout
* When wc runs, it reads from input.txt and writes to output.txt

1. **Execute the command**:
   * If it’s a built-in command (hop, seek, log, etc.), call its respective handler.
   * Otherwise, call foreground\_process() or background\_process() as needed.
2. **Restore stdin/stdout** if necessary (not shown explicitly but can be handled).

**⚠️ Edge Case Handling:**

If the file for < is missing, the shell prints:

No such input file found!

This is done by checking if open(filename, O\_RDONLY) returns -1.

**🧪 Test Case to Demonstrate:**

sh

Copy code

echo "Hello" > file.txt # Creates/overwrites file.txt

echo "World" >> file.txt # Appends to file.txt

cat < file.txt # Reads from file.txt and prints to stdout

wc < file.txt > stats.txt # Redirects input and output

**🧩 How It Fits with Pipe & Combined Handling:**

If both | and redirection exist (like cat < a.txt | grep x > b.txt), the dividingCommands() delegates it to both(), which handles redirection + piping together — though that's beyond this specific spec.

**🏁 Wrap-up:**

Overall, I ensured the shell mimics bash-like behavior for redirection using dup2, proper open() flags, and basic error handling. It also integrates seamlessly with our custom and system commands in foreground/background execution.

**11. Pipes:**

**Start with the Concept of Pipes**

In UNIX shells, a pipe (|) allows chaining commands by connecting the **stdout of one process to the stdin of another**.  
For example:

echo "Hello" | wc

The output of echo becomes the input to wc.

**🔁 Behavior Your Shell Implements**

* My shell supports **any number of pipes**.
* It runs all commands **sequentially from left to right**.
* If either side of a pipe is missing, the shell prints:

Invalid use of pipe

**⚙️ Internals: processPipes() Function**

**1. Tokenization**

* I use strtok(input, "|") to split the input command into individual components.
* If there are n pipes, we get n+1 commands.

**2. Creating Pipes**

* For n pipes, I create n file descriptor pairs using pipe(pipe\_fd[i]).

**3. Forking for Each Command**

* For each command, I call fork() to spawn a child.
* Each child handles redirection:
* If **not the first**: dup2(pipe\_fd[i-1][0], STDIN\_FILENO)
* If **not the last**: dup2(pipe\_fd[i][1], STDOUT\_FILENO)
* After setting up redirection, I execute the command using:

execl("/bin/sh", "sh", "-c", token, (char \*)NULL);

**4. Parent Process**

* Parent closes used pipe ends and waits for all children using wait(NULL).

**✅ Flow Example for echo "hi" | grep h | wc**

| **Step** | **Command** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 | echo "hi" | stdin | pipe[0] write end |
| 2 | grep h | pipe[0] read end | pipe[1] write end |
| 3 | wc | pipe[1] read end | stdout |

**🧠 Error Handling**

Before processing:

* I check if the command begins or ends with |, or if any two pipes are consecutive.
* If so, I print:

Invalid use of pipe

**12. Redirection along with pipes:**

I had to implement a custom shell (mysh) that supports:

* Pipelining (|) of **any number of commands**
* I/O redirection (<, >, >>) **in combination** with pipes
* Only **one input** and **one output** redirection allowed in a command (as per specification)

For example:

cat < a.txt | wc | cat > b.txt

**🧠 How I approached the problem:**

I split it into **two main responsibilities**:

**1. both() — 🔗 Pipeline Manager**

This function detects and sets up pipes between commands.

**✅ Steps in both():**

* I split the full input by '|' to get **individual commands** (tokens).
* Created pipes (pipe\_fd[i]) between each command.
* For each command:
  + Used fork() to spawn a child process.
  + Used dup2() to:
    - Redirect **input** from the previous pipe’s read-end
    - Redirect **output** to the next pipe’s write-end
  + Closed all unused pipe ends in the child.
  + Passed the command to processRedirection() — which handles redirection inside each command.

🔁 This repeats for **each stage** of the pipeline.

**2. processRedirection() — 🔁 Handles <, >, >>**

This function detects and sets up redirections **within one command**.

**✅ Steps in processRedirection():**

* Checks for <, >, or >>:
  + Opens the corresponding file using open()
  + Redirects stdin or stdout using dup2() to the file descriptor.
* After redirection is set up, I determine if it's a **user-defined command** like hop, seek, reveal, etc., or a **system command**:
  + If user-defined → I call the respective function (e.g., hop()).
  + Otherwise:
    - I run it as a background or foreground system command using execvp() inside foreground\_process() or background\_process().

**🧪 Example:**

bash

Copy code

cat < input.txt | grep "error" | wc > count.txt

* **cat** gets input from input.txt due to <
* Output is piped to grep "error"
* Output of that is piped to wc
* Final output is redirected to count.txt

✔️ My shell correctly redirects stdin from a file and stdout to another file — while maintaining the pipe chain.

**⚠️ Key Technical Challenges I Solved:**

1. **File descriptor management** — making sure to avoid leaks and proper dup2() calls
2. **Pipe setup across multiple children**
3. **Compatibility** — ensured both redirection and piping work seamlessly together
4. **No race conditions or zombie processes** — used wait() to reap child processes

**✅ Summary:**

In my implementation, I modularized the logic by separating pipe handling and redirection. Each child process correctly receives input/output through pipes or redirection, and processes the command accordingly — whether it's a system command or custom.

**13. Activities:**  
The activities command lists **all processes spawned by your custom shell**, sorted in **lexicographic order of command names**.

* It shows:
  + **PID** of the process
  + **Command name** (e.g., gedit)
  + **Current state**: either Running or Stopped

🧪 Example Output:

221 : emacs new.txt - Running

430 : vim - Stopped

620 : gedit - Stopped

**🧠 Interview Explanation (High-Level):**

"**I implemented an activities command in my custom shell which tracks and displays all the child processes spawned by the shell. Each process's PID, command name, and running state is tracked and printed in lexicographical order based on command name. I maintain a global array of all spawned processes, update it whenever a new command runs, and use the /proc/[pid]/stat file to fetch the runtime state of each process (e.g., running, stopped). This required careful signal handling and process tracking.**"

**🛠️ Technical Explanation**

**✅ 1. Data Structure:**

Activities activities[1000];

int top = 0;

* activities is an array of structs storing:
  + pid – process ID
  + fg\_bg – whether it’s foreground/background (1/0)
  + processName – command run
* top is the current size of the list.

**✅ 2. Adding a New Process to the List**

void appendProcessList(char \*input, int pid, int fgbg)

Whenever a process is started (after fork()), this function is called:

* Stores input (the command), pid, and whether it's foreground/background (fgbg)
* Increments top

**✅ 3. Fetching Process State using /proc/[pid]/stat**

char get\_process\_state(int pid)

* Opens /proc/[pid]/stat
* Reads the **process status character** (3rd field):
  + 'R', 'S', 'Z' → Running
  + 'T', 'X' → Stopped

If file is missing (process might be dead), return 'X'.

**✅ 4. Printing All Tracked Processes**

void printProcessList()

* Sorts the activities array lexicographically by processName using qsort().
* Iterates over the list and prints:

[pid] : [command name] - [Running/Stopped]

**✅ 5. Signal Handling**

signal(SIGINT, signalHandler);

signal(SIGTSTP, signalHandler\_z);

**SIGINT (Ctrl+C):**

* signalHandler() kills all **foreground** processes started by your shell.

**SIGTSTP (Ctrl+Z):**

* signalHandler\_z() just reports the current foreground process’s PID.

**✅ 6. On Exit / Ctrl+D**

cleanupAndExit()

* Kills all still-running foreground processes when the shell is terminated.

**🧑‍💼 What to Say in Interview (Behavioral+Tech):**

**"I implemented a custom activities command to track all the processes spawned by my shell. I used a global array to store command metadata including PID and whether it's running in the foreground. To fetch the current state (like Running or Stopped), I parsed the /proc/[pid]/stat file directly and extracted the process state character. I also wrote custom signal handlers to clean up foreground processes on Ctrl+C or exit. To ensure clean and sorted output, I used qsort() with a lexicographic comparator on command names."**

**📌 Summary of Key Functions**

| **Function** | **Purpose** |
| --- | --- |
| appendProcessList() | Adds a new process to the tracking list |
| get\_process\_state() | Parses /proc/[pid]/stat to get process status |
| printProcessList() | Sorts and displays the tracked processes |
| signalHandler() | Handles SIGINT, kills foreground processes |
| cleanupAndExit() | Called on shell exit to kill and clean up |

**14. Signals:**

"I implemented a custom ping command in my shell that sends Unix signals to other processes by their PID. This is similar to the kill command in Bash, but with the added constraint of using the signal number modulo 32, assuming x86/ARM compatibility. My implementation validates the PID against a tracked list of shell-spawned processes (activities[]), then sends the signal using kill(). If the PID is not found in the list, it prints a 'No such process found' error. I also handle Ctrl+C (SIGINT), Ctrl+D (EOF), and Ctrl+Z (SIGTSTP) as special input behaviors to manage foreground/background processes and shell exit."

**🧠 What You Achieved**

1. **ping <pid> <signal> command**
   * Sends a signal to a tracked process.
   * Signal number is taken modulo 32 (standard POSIX behavior).
   * If the PID is valid and exists, the signal is sent using kill(pid, sig).
   * If SIGKILL (9) is sent, the process is also removed from the activities list.
   * If PID is invalid → error message: **“No such process found”**
2. **Handled signals from keyboard input:**
   * Ctrl + C: sends SIGINT to the **foreground** process.
   * Ctrl + D: triggers cleanup and exits shell.
   * Ctrl + Z: sends SIGTSTP to foreground process and changes its state to “Stopped”.

**🔧 Technical Breakdown of the Code**

**🔹 getSignal()**

void getSignal(char \*input)

* Parses the user input string for ping command.
* Extracts:
  + id (PID)
  + signal number
* Example:

ping 221 9

**🔹 executeSignal(int id, int signal)**

int flag = 0;

for (int i = 0; i < top; i++)

{

if (activities[i].pid == id)

{

flag = 1;

signal = signal % 32; // sanitize signal range

kill(id, signal);

if (signal == 9)

{

activities[i].pid = -1; // remove from list if killed

}

}

}

if (!flag) perror("No such process found");

* Validates PID from the activities list.
* Performs kill(pid, signal % 32).
* If killed with signal 9 (SIGKILL) → removes from activities[].

**🔹 Handling Ctrl+C, Ctrl+D, Ctrl+Z (via signal() and handlers)**

You define signal handlers in signals\_.h or your main shell loop:

| **Signal** | **Key** | **Handler** |
| --- | --- | --- |
| SIGINT | Ctrl + C | Interrupt foreground process |
| SIGTSTP | Ctrl + Z | Stop foreground process (make background) |
| Ctrl + D | EOF | Exit shell gracefully and kill all |

**Handled using:**

signal(SIGINT, signalHandler);

signal(SIGTSTP, signalHandler\_z);

**🔹 check() Function (Special Case for sleep)**

If a sleep process is resumed, the original duration is adjusted by checking time.txt and recalculating remaining sleep time.

Purpose:

* Helps when reviving a sleep command in FG\_BG\_() by updating its time to reflect what’s left.

**🔹 FG\_BG\_() Function**

Used when pushing a process to **foreground or background** after stopping:

FG\_BG\_(pid, fg\_or\_bg, home\_dir);

* Finds the process in activities[]
* Calls:
  + foreground\_process() or
  + background\_process()
* Removes from activities list after execution (top--)

**🧑‍💼 How to Explain in Interview**

**"For the ping command, I parse user input to extract PID and signal number, validate it against a tracked activities[] list, and send the signal using kill(). I restrict the signal to POSIX-safe range by taking modulo 32. I handle edge cases like invalid PIDs and cleanup after SIGKILL. I also integrated signal handling for SIGINT, SIGTSTP, and Ctrl+D to manage foreground processes, backgrounding, and graceful shell exit. This involved a strong understanding of Unix signals and process control."**

**15. fg and bg:**

I’ve implemented foreground (fg <pid>) and background (bg <pid>) process control commands in a custom Linux shell written in C. These allow users to resume previously stopped background processes either in the foreground or background.

**Key Features of fg and bg:**

1. **fg <pid>**:
   * Brings a stopped background process with the given PID to the **foreground**.
   * Transfers terminal control to that process.
   * Waits until the process finishes or is stopped.
   * If the PID does not exist or isn’t tracked, it prints: "No such process found".
2. **bg <pid>**:
   * Resumes a stopped process in the **background**.
   * Also prints "No such process found" for invalid or untracked PIDs.

**How it works (internally):**

**1. Command Parsing (dividingCommands)**

* When the user inputs fg <pid> or bg <pid>, the input is passed to GetID\_FG\_BG(...).
* This function tokenizes the command to identify:
  + Whether it’s fg or bg.
  + The process ID.

**2. Process Verification**

* executeFG\_BG(pid, is\_fg, ...) is called next.
* It first validates:
  + If the PID is valid (pid > 0).
  + If the process actually exists using kill(pid, 0).
  + If the process is tracked in the activities[] array.

**3. Sending Signal to Resume Process**

* If all checks pass, it sends SIGCONT to the PID to **resume the process** using kill(pid, SIGCONT).

**4. If Foreground (fg):**

* Shell gives terminal control to that process using:

tcsetpgrp(STDIN\_FILENO, pid);

* Then waits for the process to finish or stop using:

waitpid(pid, &status, WUNTRACED);

* After it finishes, control is restored to the shell with:

tcsetpgrp(STDIN\_FILENO, getpgrp());

**5. If Background (bg):**

* The process is resumed with SIGCONT, and control stays with the shell (non-blocking).

**6. Activity State Management**

* The activities array keeps track of:
  + Each process’s PID.
  + Its state: foreground (fg\_bg = 1) or background (fg\_bg = 0).
* After each action, the activity is updated accordingly.

**Error Handling:**

* Invalid PID: "Invalid PID"
* Process not found in system or activities[]: "No such process found"
* Invalid usage (e.g., missing PID): "Invalid usage of fg/bg"

**Conclusion:**

This design allows robust management of foreground and background jobs in my shell. It replicates typical shell behavior using system calls like kill, tcsetpgrp, and waitpid, and tracks process metadata using a global activity table (activities[]). The shell handles terminal control elegantly and offers a user-friendly interface for job control.

**16. Neonate:**

Command : **neonate -n [time\_arg]**

**✅ 1. Parsing Input**

Function: neonate(char \*input)

* Extracts the time interval (e.g., 4 from neonate -n 4) using strtok() and passes it to callingProcess(int wait\_time).

**✅ 2. Looping and Execution**

Function: callingProcess(int wait\_time)

* Enables **raw mode input** (non-canonical mode) using enableRawMode().
* In a loop:
  + Calls is\_key\_pressed() → checks if 'x' was pressed.
  + Calls executeNeonate() → prints the newest PID.
  + Sleeps for wait\_time seconds.
  + Exits if 'x' is detected at any step.

**✅ 3. Fetching Most Recent PID**

Function: executeNeonate()

* Reads /proc/loadavg, which looks like:

0.00 0.01 0.05 1/123 12345

* The last field (12345) is the **PID of the most recently created process**.
* Extracts and prints this PID.

**✅ 4. Checking for Key Press (x)**

Function: is\_key\_pressed()

* Uses termios to:
  + Turn off canonical mode and echo.
  + Make input **non-blocking** (so it doesn’t wait for Enter).
* Uses getchar() to detect if 'x' is pressed.
* Restores terminal settings after checking.

**✅ 5. Raw Mode Management**

Functions: enableRawMode() and disableRawMode()

* Temporarily disable line buffering and echo so getchar() reacts to keypress immediately.
* Registers disableRawMode() to auto-reset terminal on exit using atexit().

**🔹 Key System Concepts Demonstrated:**

* **Terminal raw mode** via termios and fcntl.
* Reading from the **/proc pseudo-filesystem**.
* **Non-blocking I/O** to allow real-time user interrupts.
* **Signal-safe loop design**, checking for user input before/after sleep.

**🔹 Possible Follow-ups (Bonus Points):**

* 🛠 Could be extended to monitor other process attributes (name, memory usage, etc.).
* 📦 Uses only **POSIX system calls** and direct /proc parsing – no external dependencies.
* 🧠 Shows deep understanding of **UNIX process and terminal internals**.

**✅ In Short:**

"neonate -n is a custom-built utility that monitors and prints the latest PID every *n* seconds using /proc/loadavg, and stops instantly if the user presses 'x', all without any system binaries. It uses raw terminal I/O for responsive key detection and process monitoring."

**17. iMan:**

**"iMan"** is a custom shell command that mimics the functionality of the Unix man command by fetching man pages **from the internet** instead of local files. It uses **raw sockets** to perform an HTTP GET request to http://man.he.net/ and prints the resulting HTML content to the terminal.

**🔹 Purpose and Expected Behavior**

* When the user enters: iMan <command\_name>  
  It connects to man.he.net and fetches the relevant man page.
* If the command is invalid or not found, it still prints the server's HTML response (no special error-handling is required).
* If extra arguments are passed, only the first is considered: iMan sleep extra → iMan sleep.

**🔹 Key Functional Requirements**

1. Parse input and extract the command.
2. Use **sockets** (TCP, IPv4) to connect to man.he.net on port 80.
3. Send a properly formatted HTTP GET request:

GET /?topic=sleep&section=all HTTP/1.1

Host: man.he.net

Connection: close

1. Read the response from the server.
2. **Ignore the HTTP headers** and print only the **body** (can include HTML tags).
3. Display the output on stdout exactly as received.

**🔹 Technologies & Concepts Used**

1. **Socket Programming**:
   * socket(), connect(), send(), read(), close()
   * gethostbyname() for DNS resolution
   * struct sockaddr\_in for setting server info
2. **HTTP Protocol**:
   * Constructs a basic GET request manually
   * Parses raw HTTP response
3. **String Parsing**:
   * strtok used to tokenize the input and extract command
   * Optional HTML tag-skipping logic (though HTML tags can be printed as per spec)

**🔹 Real-World Relevance**

This spec simulates a minimal HTTP client using low-level system calls — a great showcase of understanding:

* How browsers and curl work under the hood
* HTTP protocol basics
* Network programming with sockets
* Command parsing in shells

**🔹 Code Walkthrough Summary**

Here’s what the implementation does:

1. **Parse Command**:
   * Expects input like "iMan sleep" → extracts "sleep".
2. **Build HTTP Path**:

snprintf(path, sizeof(path), "/?topic=%s&section=all", command);

1. **Create and Connect Socket**:
   * IPv4 + TCP socket via socket()
   * Resolves domain using gethostbyname()
   * Connects to port 80
2. **Send HTTP Request**:
   * Sends the GET request string to the server
3. **Receive and Parse Response**:
   * Reads in chunks using read()
   * Strips headers by finding the first blank line (\n\n or \r\n\r\n)
   * Prints the rest to stdout (can include HTML tags)

**🔹 Edge Cases Handled**

* Only first argument is considered even if more are passed.
* If man page is not found, prints server response as-is (per spec).
* Doesn't crash on malformed input, as strtok and checks are in place.

**🔹 Additional Talking Points**

* Could extend to strip HTML tags properly for better readability.
* Could use getaddrinfo() for IPv6 compatibility.
* No use of HTTP libraries like libcurl, demonstrating low-level understanding.

**✅ Summary for Interviewer**

*"I implemented a custom shell command called iMan that fetches online man pages using low-level socket programming. It connects to man.he.net over HTTP, builds a GET request based on the command name, strips the headers from the HTTP response, and prints the body to the terminal. This showcases my understanding of sockets, TCP/IP networking, basic HTTP, and input parsing — all without relying on external libraries."*