Computing Lab (CS69201)

Rahul.k

25CS60R16

Design Document: MyTerm

A Custom Terminal with X11GUI with some basic and some advanced functionalities.

1. Overview

The goal of the project is to build our custom shell(MyTerm) that will have its own GUI with some basic and some advanced functionalities and run as a standalone application program. The shell will accept user commands (one at a time), and execute the same.

This project is entirely written in c++ using x11 library for GUI support and supports a wide range of unicode characters for I/O.
It has the essential features of a modern terminal, like tabs, a good command history search, and smart autocompletion that helps in typing faster and an upgraded version of the watch command where commands are executed in parallel.

Core Features:

- ★ Tabbed Interface : Support for multiple independent shell tabs.
- ★ Basic External commands stored as executables and some advanced commands.
- ★ Multiline Command Editing **:** Support for commands that span multiple lines (unicode characters too).
- ★ Stdio redirection into and from a file,combination is also supported.
- ★ Support for pipe : standard output of one command is redirected to the standard input of the next command, in sequence.
- ★ **MultiWatch :** Parallel version of the watch command with unix timestamp and command name which generated the output.
- ★ Line Navigation : Ctrl+A moves the cursor to the start of the current line, Ctrl+E moves to the end mimicking the behavior as in real terminal.
- ★ Interrupts : Ctrl+C stops the current command in execution whereas ctrl+Z pushes the execution to background and the shell prompt reappears.
- ★ Searchable History : The system keeps a record of the last 10,000 commands executed. This history can be viewed using the **history** command.
  - ➢ Ctrl+R helps in searching through the shell history by displaying the exact matches or the ones with longest substring match of length greater than 2
- ★ Auto Complete : Multi-stage completion for filenames and directories which helps us select in case of common prefixes.

2.System Architecture

2.1 Process Model: The Parent-Child Architecture

The most critical architectural decision was to separate the application into a multi-process system. For each tab, a **parent process** manages the UI and a **child process** handles shell command execution.

Parent Process (UI Manager):

- **Responsibilities:** Manages the X11 window, handles all user input (keyboard, mouse), renders the UI for all tabs, and maintains the overall state.
- **Function:** This ensures the UI remains responsive and never freezes, even when a command in one of the tabs occupies the shell for a relatively longer time.

Child Process (Shell Executor):

- **Responsibilities:** Executes shell commands, manages the current working directory, creates process groups for job control, and communicates output and state changes back to the parent.
- **Function:** This isolates the potentially blocking shell operations from the main event loop, creating a stable and robust user experience.

2.2 Inter-Process Communication (IPC)

Communication between the parent and child processes is achieved using **unidirectional pipes (pipe())**. Each tab has two pipes: one for parent-to-child communication (sending commands) and one for child-to-parent communication (receiving output).

Model: A simple, text-based protocol was established with prefixes to denote the message type:

- ➢ OUT:<line>: A line of standard output/error from the command.
- ➢ CWD:<path>: The current working directory has changed.
- ➢ PGID:<id>: The process group ID of the newly launched command.
- ➢ CMD_DONE:: The command has finished executing.
- ➢ CLEAR:: A request to clear the screen.

Challenge: Ensuring non-blocking reads from the child. The parent process cannot afford to wait for data.

Solution: The read end of the child-to-parent pipe was set to non-blocking mode (O_NONBLOCK) using fcntl(). This allows the parent to poll for data on every iteration of its main loop without halting.

2.3 Core Data Structure: ShellTab

The entire state for a single terminal session is encapsulated within the ShellTab struct. A vector<ShellTab> in the parent process manages all open tabs.

- **Key Fields:** pid, parent_to_child_fd, child_to_parent_fd, input, outputLines, cursorPos, scrollOffset, jobs, completionMode, searchMode.
- **Significance:** This centralized data structure was crucial for cleanly managing the state of multiple concurrent sessions and ensuring that user events were applied to the correct active tab.

3. Feature Modeling and Implementation

3.1 Graphical user Interface with MultiTab support

Tab creation in the terminal is handled by a multi-step process that creates a new, independent shell session while keeping the user interface responsive.

Fork + exec

First, the application sets up two communication channels **(pipes)**. This is how the main UI will send commands to the new tab and how the new tab will send its output back using pipe().

parent_to_child_fd: For sending user commands *to* the new shell.

child_to_parent_fd: For receiving command output and status updates *from* the new shell.

1. Next, the main application uses the fork( ) command to create an exact duplicate of itself. Instantly, there are two processes running instead of one.
2. The two processes are immediately given different roles:
    a. The **new process** becomes the dedicated shell for the new tab. Its only job is to run the commands.
    b. The **original process** stays as the UI manager. It adds the new tab to its list on-screen and listens for output from its new "worker" without ever freezing.

now on, the child's life consists of reading commands from the parent, executing them, and writing the results back.

Parent uses fcntl() to set the reading pipe from the child to non-blocking mode. This is critical it ensures the UI never freezes while waiting for output from a command. The parent can "peek" for data without getting stuck.

Control Commands:

Ctrl + T - create a new Tab.

Ctrl + W - Delete the current Tab(closing the last tab left closes the terminal).

Ctrl + Left/Right - Switch between tabs.

Ctrl + Mouse Scroll - Tab_bar scroll if it exceeds the Window's view.

3.2 Run an External Command

The child process (shell/current tab) doesn't run the external command itself. If it did, it would be replaced by that command and the tab's shell would die.

Instead, it acts like a manager that launches a temporary, disposable process for each command. Here's the simple breakdown of how it works.

1.Parse the Command : First, the shell process receives the command string from the main UI (e.g., ls -l /home). It can't use this string directly hence it is passed into the **ParseCommand** function implemented where it breaks down the command into the program name (ls) and a list of arguments (["ls", "-l", "/home"]).

2.Fork again : The shell process duplicates itself by calling fork( ) again. This creates a *new* child process (a "grandchild" from the main UI's perspective).

- The Shell Process: Its job is to wait for the grandchild to finish.
- The Grandchild Process: Its only purpose is to run this one single command.

This is crucial because it allows the shell to stay running and ready to accept another command after the current one is done.

3.Redirect the Output : The grandchild process needs to send its output back to the main UI.

Before running the command, it uses a command called dup2() to redirect its standard output and standard error to the pipe that leads back to the parent UI.

This ensures that the grand child is now ready to execute the command (ls) and redirect the output to our terminal window to be displayed .

4.Execvp( ):

Now the grandchild process is ready. It calls execvp().

What it does: The execvp command completely replaces the grandchild process with the program it's told to run (e.g., ls). The grandchild process stops being a copy of the shell and becomes the ls program.

The Search: execvp is smart, it automatically searches the system's PATH to find the ls executable file unless execl or execv where we have to pass the whole path of the binary of the command.

No Return: If successful, this command never returns. The ls program runs, prints its output (which is redirected by the pipe), and then exits.

5.Clean up

The shell process, which was waiting, detects that its grandchild has finished. It then sends a CMD_DONE(as mentioned in 2.2) message back to the main UI to let it know the prompt can be displayed again. The temporary grandchild process is now gone, and the shell is ready for the next command.

3.3 Multiline Unicode Input:

The logic for multiline commands is handled by buffering lines until the command is deemed complete**.**

Model: Instead of executing a command immediately upon pressing Enter, the line of text is first added to a temporary multilineBuffer.

The Check: A simple helper function, commandComplete(), then inspects the *entire* buffer. In our implementation, it uses a basic but effective heuristic: it checks if the number of double quotation marks (") is even. If it's odd, it assumes the user is in the middle of a string and waits for more input.

Execution Flow:

1. User types the first line (e.g., echo "hello) and presses Enter.
2. The commandComplete check fails (one quote is an odd number).
3. The terminal displays a continuation prompt (>) and waits for the next line.
4. User types the second line (e.g., world") and presses Enter.
5. The check now passes (two quotes is an even number).
6. The lines in the buffer are joined together into a single command string and executed.

Unicode Support

The most basic way to handle keyboard input in X11 is with functions like XLookupString.This approach is fundamentally broken for Unicode. It's designed for ASCII and cannot process characters outside that limited set.

Any attempt to use international characters, symbols (like ₹), or even simple accented letters would fail, resulting in incorrect or missing text.

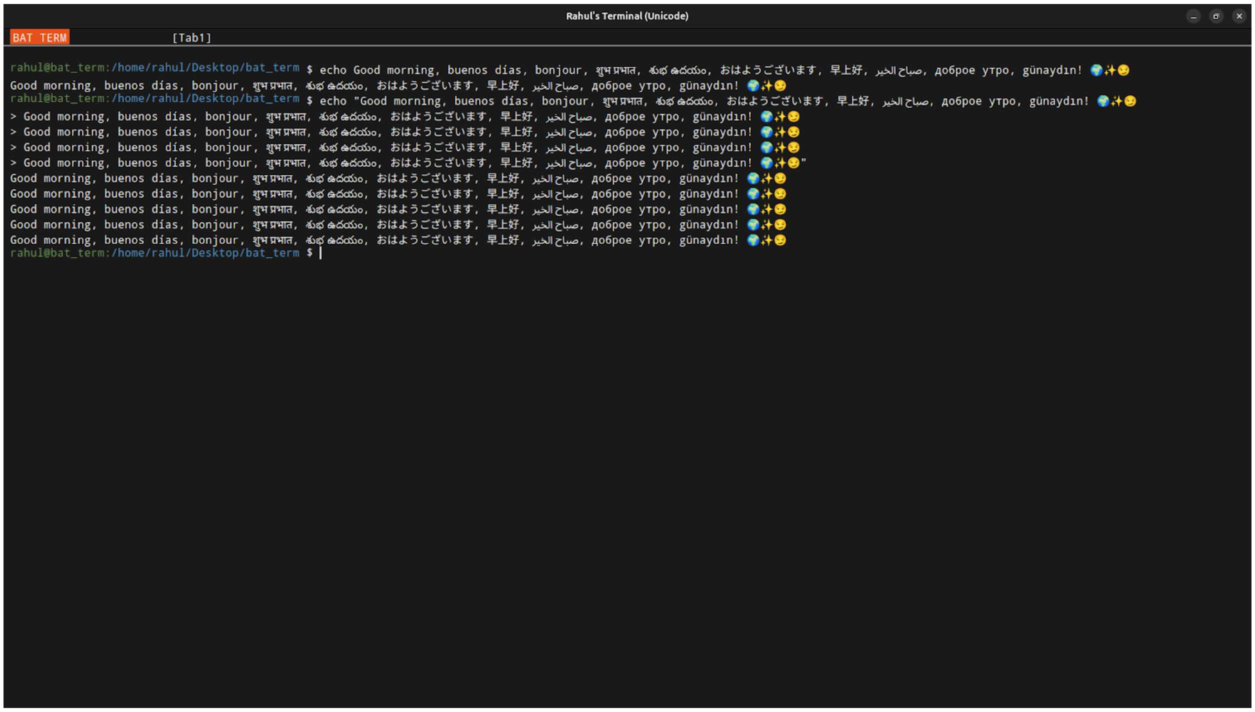
We use the Xutf8LookupString function. This correctly interprets keypresses and provides the corresponding characters as a **UTF-8 encoded string**. This was a major improvement, as it allowed us to at least **capture and store** Unicode characters correctly.

we faced immense difficulty in *displaying* them correctly with basic X11 drawing functions (XDrawString) facing challenges like text looked blocky and aliased, with poor support for modern fonts (TTF/OTF)

This was the biggest problem. Many languages, like Hindi or Arabic, require "shaping," where characters change form or combine based on their neighbors (e.g., क् + ष = क्ष). Basic X11 cannot do this; it would just draw the characters separately and incorrectly.

The clear and final solution was to adopt the standard libraries used by virtually all modern Linux applications: **Cairo and Pango**.

By moving to this combination, we offload all the incredibly complex aspects of text rendering to libraries that are expertly designed for that exact purpose. This allows us to support virtually any language in the world or any unicode character(emojis) correctly and with high-quality presentation.



4 & 5. Input and Output Redirection

When the command is parsed by the shell, it sees “>” / ”<” and filename (out/in).txt.
It opens the file in write/read mode and the os gives back a file descriptor of the corresponding file.

Before the fork and exec step in the shell(child process) we make a crucial system call which connectsThe standard output/input to the given file

dup2(file_descriptor_for_outfile, STDOUT_FILENO);

The child process then calls exec, it runs and tries to print its output, the OS automatically sends it to out.txt because of the rewiring.

Combination of Both:

For ./a.out < infile.txt > outfile.txt, just do both steps.
Before calling exec, the child process makes two dup2 calls: one to rewire input from infile.txt and another to rewire output to outfile.txt.

6. Implementing support for pipe

Before running any commands, the shell creates a communication channel using pipe( ).

1. Create the First Child

The shell then calls fork( ) to create the first child process. Inside this child, before it executes, it performs two key actions: Rewire Output: It redirects its standard output to the write end of the pipe using dup2(). Thismeans anything the command tries to print to the screen will go into the pipe instead.

Execute: It calls exec to run the command.

2.Create the Second Child (for wc)

The shell calls fork( ) again to create a second child process. This child does the opposite rewiring:

Rewire Input: It redirects its standard input to the read end of the pipe using dup2(). This means when Second command asks for input from the keyboard, it will get the data coming out of the pipe instead.

Execute: It calls exec to run the wc command.

The parent shell's job is now to clean up and wait.

Close the Pipe: The parent shell must close both ends of the pipe for itself. This is a critical step that signals to the child processes that no more data is coming.

Wait for Finish: The shell waits for both child processes (ls and wc) to complete their execution.

7. MultiWatch

This feature runs multiple commands at a regular interval and displays all their outputs on a single, continuously updating screen.

The entire process relies on **temporary files** to capture the output of each command before displaying it. Here is the complete flow.

1. Setup and Parsing

When a command like multiwatch "ls -l" "date" is run,

the shell Parses the Arguments: It recognizes the command and separates the arguments into

list of commands to be monitored (["ls -l", "date"]).

2. Execution Loop (repeats infinitely with an interval of 2 seconds between successive runs)

For each command (executed in parallel)
2.1 Create a **Temporary File** a unique, temporary filename (e.g., .temp.PID1..txt). This file acts as a temporary notepad.

2.2 It calls **fork()** to create a child process that will be responsible for running just this one command.

2.3 **Redirect and Execute**: Inside this new child process:
It opens the temporary file.
It uses dup2() to redirect its standard output to this temporary file.
It calls execvp to run the command (e.g., ls -l). The command runs, but instead of printing to the screen, its entire output is written into the temporary file.

2.4 **Wait for Completion:** The main MultiWatch process waits for the child to finish. At this point, it knows the temporary file is complete.

2.5 Delete the temp files and kill the child processes after execution (either stopped by Ctrl + C or killed after a Ctrl + Z).
mw_mul.webm

8.Line Navigation with Ctrl+A and Ctrl+E

support for command-line editing shortcuts similar to Bash. When typing a command in the shell:
a. Pressing Ctrl+A should move the cursor to the start of the current line.
b. Pressing Ctrl+E should move the cursor to the end of the current line.

instead of **termios** , x11 already provides graphical equivalent, as it captures individual KeyPress events instantly without waiting for the user to press Enter. It serves the same purpose as raw mode
but in a graphical context.
A **cursor index** is maintained which will updated on either of the above commands and the line is Redrawn.

9.Interrupting commands running in the shell

The design relies on fundamental Unix concept: **process groups**. Instead of managing individual processes, the shell treats each command (even complex pipelines like cat file | grep word) as

a single entity called a process group.

The main UI process never runs commands directly. It captures the user's intent (e.g., Ctrl+C) and sends the appropriate **signal** to the target command's process group, leaving the shell itself unharmed.

a. Ctrl+C: Halting a Running Command (SIGINT)

1.Launching the Command in a New Process Group

When fork() is called to create a process for the command, the very first thing the new child process does is call setpgid(0, 0).

- This system call creates a **new process group** and sets the child process as its leader. The ID of this new group (the PGID) will be the same as the child's process ID (PID).
- Any further processes created by this command (e.g., in a pipeline) will inherit this process group ID.

2.Storing the Target Process Group ID (PGID)

1. The main UI process needs to know the PGID of the command it needs to signal.
2. After launch_command gets the new PGID, the tab_main process sends it back to the main UI process through a pipe with the message format "PGID:<id>\n".
3. In app_logic.cpp, the handle_child_message function receives this message.
4. The PGID is stored in the active tab's state:
5. tabs[activeTab].running_command_pgid = stoi(msg.substr(5))

3.Detecting the Keystroke and Sending the Signal

The code checks for the key combination ControlMask + XK_c. When detected, it executes the kill system call.

kill(-currentTab.running_command_pgid, SIGINT);

Key Detail: The first argument to kill is negative. kill(-pgid, SIGINT) tells the operating system to send the SIGINT signal to every process in that process group. This is what allows Ctrl+C to correctly terminate an entire pipeline.

4.Restoring the Prompt

The UI sends an __INTERRUPT__ message to its tab process. The tab_main function in commands.cpp receives this, knows the command was killed, cleans up the I/O pipe it was

reading from, and sends CMD_DONE: back to the UI. This sets is_busy = false, and the prompt is drawn on the next frame.

b. Ctrl+Z: Moving a Command to the Background (SIGSTP + SIGCONT)

The mechanism for Ctrl+Z is similar but involves different signals and adds the concept of **job control**.

1.When **Ctrl+Z** is pressed in the terminal, our application code does not send the pause signal. Instead, the Operating System's terminal driver intercepts this key combination.

Signal: SIGTSTP (Signal: Terminal Stop)

Sender: The OS kernel

Receiver: The current foreground process group (i.e., the running command, like sleep 100).

Effect: The command is instantly frozen. Its execution is paused, but its state (memory, variables, etc.) is preserved perfectly.

2: The **Resume** (sent by our Shell)
At the same time, our X11 event loop in app_logic.cpp detects the Ctrl+Z keypress. Our code knows that the OS has just frozen the command, so its job is to resume it in the background.

Signal: SIGCONT (Signal: Continue)

Sender: Our shell application (specifically the kill(-pgid, SIGCONT) line).

Receiver: The same process group that was just frozen.

Effect: The command is instantly un-frozen and continues running exactly where it left off.

As we see a question arises here, why is the os terminal driver sending the ctrl Z , we have to do it right?

It is technically possible to handle Ctrl+Z entirely on our own, but it's complex and goes against the standard design of Unix-like systems. To do it, we would have to tell the OS kernel to stop helping us.

This involves changing the terminal's operating mode from its default "canonical mode" to "raw mode".

If we were to put the terminal in raw mode to handle `Ctrl+Z` ourselves, we would suddenly become responsible for everything the OS driver used to do for free.

Brittleness: If our application crashes while the terminal is in raw mode, the user's terminal is left in a "broken" state—it won't respond correctly to input. They would have to manually type `reset` or close the terminal to fix it.

The current design is the standard and most robust solution. We use a hybrid approach:

The OS: Handles the low-level, universal task of seeing Ctrl+Z and sending the initial SIGTSTP signal.

Our Shell: Handles the high-level, custom logic of what to do next—sending SIGCONT and managing the process as a background job.

10.Searchable shell history

The history functionality is primarily managed by three files: history.cpp for storage, commands.cpp for the history command, and app_logic.cpp for the Ctrl+R search feature.

Storing and Loading History

Data structure: `vector<string> commandHistory`.

At the start of the shell, the loadHistory() function in history.cpp reads commands line-by-line from a file named .bat_term_history into the vector.

Whenever a new command is run, the saveHistory() function is called. It appends the new command to the commandHistory vector and then overwrites the .bat_term_history file with the updated contents.

Limits : MAX_HISTORY_FILE (10000) & MAX_HISTORY_DISPLAY(1000).

History command :
Mechanism: When a history `command is run` , the code checks for this specific command string. It then calculates the correct starting index to show the last 1000 commands and iterates through the `commandHistory` vector, writing each formatted line to the pipe that sends output back to the main terminal window.

Ctrl+R Search Functionality

After trying Suffix Tree and Best Fuzzy Match Algorithms for longest prefix match, We moved to Inverted Index based algorithm as it is observed to be faster when the length of history grows large(**This is the same technique used by search engines and is a great balance of performance and simplicity).**

How it Works: pre-processing the history by breaking every command into small, overlapping chunks of characters called n-grams (e.g., 3-character chunks, or "trigrams"). then an Inverted Index is built, which is a map where each key is a trigram and the value is a list of all commands that contain it.

Example: ls -l -> trigrams {"ls ", "s -", " -l"}.
Search Process:

1.Break the search term into the same trigrams.
2.Look up each trigram in the index to get lists of "candidate" commands.
3.Combine these lists to get a small set of potential matches.
4.Run the current (slow) Longest Common Substring algorithm only on this much smaller set of candidates, instead of all 10,000 commands.

One-Time Setup

When the shell loads its history, it now also builds an Inverted Index. This is a map where the keys are small, 3-character chunks of text (trigrams) and the values are lists of which commands contain them.

Example: The command git status would be broken into trigrams {"git", "it ", "t s", " st", "sta", ...}. The index would map each of these trigrams to the ID of the git status command.

When Ctrl + R is pressed followed by Enter, the performHistorySearch() function now uses the index to execute a much faster search:

Exact Match: The code first checks for a simple, exact match for maximum speed and relevance.

Candidate Filtering (The Index Search): If no exact match is found, the search term is broken into the same trigrams. The index is used to instantly find all commands in the history that share at least one trigram with the search term. This creates a small "candidate list," likely containing just a few dozen commands instead of 10,000.

Final Ranking (Longest Common Substring): The original, more intensive Longest Common Substring algorithm is then run only on this small candidate list. This final step accurately finds the best match(es) from the most relevant commands with minimal delay.

Pros:

- Massive Speedup: Drastically reduces the number of commands needed to check in detail.

- Easy to Implement: Can be built easily using a C++ std::unordered_map.
- Good Memory Profile: Memory usage is manageable.

Cons:

- Not as theoretically fast as a suffix tree, but more than fast enough for this purpose.

Alternative Ranking Algorithms
Instead of using Longest Common Substring to rank the candidates from the inverted index, a lighter-weight scoring metrics can be used.

Example: **Jaccard Index**: This metric measures similarity by comparing the sets of trigrams from the search term and a history command. The formula goes like (size of intersection) / (size of union). It's much faster to compute than the dynamic programming approach and often gives very good "best guess" results for relevance.

11. Auto-Complete Feature for File Names

Recursive file and directory path completion. The logic is primarily located in the handleTabCompletion function within app_logic.cpp

The process begins when the `Tab` key is pressed, which is detected in **`handle_x11_event`**. This calls **`handleTabCompletion`**, which intelligently parses the word currently under the cursor. It correctly separates a path like `test1/ab` into the directory part (`test1/`) and the prefix to be matched (`ab`).

Finding Matches: The function then scans the contents of the relevant directory (either the current directory or one specified in the path) using the `opendir()` and `readdir()` system calls. It builds a list of all files and directories whose names begin with the typed prefix.

Handling Results:

Single Match: If only one item matches, the shell completes it automatically. It intelligently adds a / for directories to allow for further recursive completion, or a space for files to prepare for the next argument.

Multiple Matches: If there are several possibilities, the shell calculates the Longest Common Prefix (LCP) of all matches.

If the LCP is longer than what is typed it completes the input up to that common point.

If already the entire LCP is typed, the shell enters "completion mode," displaying a numbered list of all options to choose from.

Why simple LCP and not a trie based approach when we have prefix match involved?

1. Command Completion (e.g., git, grep)
This is for completing the name of the program to run.

Why a Trie is **PERFECT** here: The list of available commands on the system (in directories like /bin and /usr/bin) changes very rarely only when a software is installed/uninstalled.

Implementation: The Trie can be built once when the shell starts up. There's no need to worry about updating it because the command list is stable. This gives incredibly fast command searches with almost no maintenance overhead during the session.

concerns like (rm, etc.) do not apply here.

2. File Path Completion (e.g., cd my_project/)
This is for completing file and directory names.

Why a Trie is **PROBLEMATIC** here: As correctly pointed out, the filesystem changes constantly.

If a Trie is used as a cache for directory contents, the difficult problem of **cache invalidation** might occur**.**

rm *.txt example is perfect. To update the cache, the shell would need to either constantly monitor the filesystem for changes (which is complex and resource-intensive) or re-scan the directory frequently, which defeats the purpose of the cache.

So , Our LCP Approach guarantees

Performance : For the vast majority of directories, the performance of reading from the disk is perfectly fast. Modern filesystems and disk caching by the OS make this operation very quick. The overhead of a complex Trie caching and validation system would likely be slower in the average case.

Correctness : the method of reading the directory directly with `readdir()` guarantees that the completion suggestions are **always 100% up-to-date**. It can never offer a file that was just deleted or miss a file that was just created. Correctness is more important than raw speed here.

And also simplicity.

After correctly identifying the trade-offs. Sticking with the current LCP and direct-directory-scan method for file paths. It is the right design choice.
A Trie is the ideal solution for the *separate* feature of completing command names.