

DESIGN DOCUMENT

1 Graphical User Interface (GUI)

1.1 Objective

The goal of this feature is to implement a fully functional graphical interface for the custom shell using the X11 library. The interface replicates the behavior of a standard terminal such as `bash`, allowing the user to input commands, view outputs, and manage multiple shell instances simultaneously within the same window.

1.2 Design and Implementation

1.2.1 Window Initialization

The X11 window is created using the following functions:

- `XOpenDisplay()` – Opens a connection to the X server.
- `XCreateSimpleWindow()` – Creates the main terminal window.
- `XMapWindow()` – Maps the created window to make it visible on the screen.

The window dimensions, font, and event masks are configured during initialization. Event masks include `KeyPressMask`, `ExposureMask`, and `StructureNotifyMask` to handle keyboard input, redraw events, and window resizing respectively.

1.2.2 Event Handling

All user interactions are processed through `XNextEvent()`. For each event:

- `KeyPress` – Captures keystrokes and maps them to ASCII characters.
- `Expose` – Redraws the screen buffer when the window is uncovered or resized.
- `ClientMessage` – Handles window close events.

A key translation mechanism converts X11 keycodes to characters which are appended to the active tab's text buffer.

1.2.3 Text Buffer and Rendering

Each tab maintains its own circular text buffer, storing all input and output lines. The text buffer is rendered using `XDrawString()` calls. Whenever new data is appended or received from the shell process, the redraw routine updates only the visible region for efficiency. Scrolling is implemented logically by changing the buffer offset rather than re-rendering all lines.

1.2.4 Integration with Shell Processes

Each tab is linked to a child process created using `fork()` and `execvp()`. Two pipes (`to_child` and `from_child`) connect the parent GUI process and the child shell, redirecting standard input and output through the GUI. The parent reads shell output asynchronously and displays it on the window in real-time.

1.2.5 Multi-Tab Management

Tabs are managed through a fixed-size array of `Tab` structures. Each structure stores:

- The shell process PID.
- Input/output pipes.
- Text buffer for terminal contents.
- Current cursor and scroll positions.

Switching between tabs changes which buffer and shell process are active. Each tab operates independently, ensuring isolation of shell sessions.

1.2.6 Keyboard Shortcuts

The following shortcuts are implemented to enhance usability:

- **Ctrl + Shift + T** – Open a new tab (creates a new shell instance).
- **Ctrl + Shift + W** – Close the current tab and terminate its shell process.
- **Ctrl + Tab** – Switch to the next tab cyclically.

1.2.7 Error Handling

All X11 function calls, pipe creation, and process management functions are checked for failure. If initialization fails (e.g., display cannot be opened), the program terminates with an appropriate error message.

1.3 Additional Implementations

1.3.1 Cursor Blinking

- To give the terminal a real GUI feel, the cursor is made to blink.
- Implemented using a timer or a separate thread that toggles the visibility of the cursor at fixed intervals (e.g., 500 ms).
- The drawing routine (`draw_ui()`) checks the cursor state and renders it as either visible (block or underline) or invisible.
- This improves user experience by providing a visual cue for the current typing position.

1.3.2 Scrolling Output

- When output exceeds the visible area of the terminal window, scrolling functionality is needed.
- Implemented by maintaining a circular buffer of lines for each tab and an offset index indicating the first visible line.
- Mouse wheel events are captured to adjust the offset, allowing users to scroll up and down through the terminal history.
- The `draw_ui()` function redraws only the visible lines, ensuring efficient rendering even with large output.
- This feature allows users to view previous command outputs without affecting the current input line.

2 Running External Commands

2.1 Objective

This feature enables the terminal to execute external commands entered by the user within the X11 graphical environment. External commands refer to executable programs present in the system (e.g., `ls`, `gcc`, `./a.out`). They are not built into the terminal itself and must be executed by creating a separate child process.

2.2 Design and Implementation

2.2.1 Command Input and Parsing

When the user types a command and presses the **Enter** key, the text stored in the active tab's input buffer is parsed to separate the command name and its arguments. This parsing is handled by the function `parse_command_line()`, which tokenizes the user input based on spaces and quotes.

Listing 1: Command Parsing

```
int argc = parse_command_line(t->current_line, argv, MAX_ARGS, &
    cmdbuf);
```

The resulting `argv` array is later passed to the process creation routine. The parsed command is also stored in the tab's history buffer for later retrieval.

2.2.2 Process Creation and Execution

To execute the command, the terminal creates a new process using `fork()`. The parent process continues to handle GUI updates, while the child replaces its process image with the requested program using `execvp()`.

Listing 2: Executing External Commands

```
pid_t pid = fork();
if (pid == 0) {
```

```

    dup2(t->to_child[0], STDIN_FILENO);
    dup2(t->from_child[1], STDOUT_FILENO);
    dup2(t->from_child[1], STDERR_FILENO);
    close(t->to_child[1]);
    close(t->from_child[0]);
    execvp(argv[0], argv);
    perror("execvp failed");
    exit(1);
}

```

The parent process saves the child's PID and uses the pipes to read its output asynchronously. This allows the terminal window to display the output from the executed command in real time, similar to a standard bash terminal.

2.2.3 Input and Output Redirection

Two pipes are used to connect the GUI and the child process:

- `to_child[1]` – Used by the GUI to write user input into the child's standard input.
- `from_child[0]` – Used by the GUI to read and display the output produced by the child.

The parent continuously monitors these pipes using the `select()` system call inside the main event loop. Whenever data is available, it is read and appended to the tab's output buffer via the `append_text()` function.

2.2.4 Integration with GUI

The execution of commands is seamlessly integrated into the graphical terminal:

- The user types a command into the X11 window.
- On pressing **Enter**, the command is parsed and executed in a child shell process.
- The output of the command is captured through the pipe and displayed in the GUI via `draw_ui()`.

All user interaction and shell communication occur inside the GUI window, without any console I/O.

2.2.5 Built-in vs External Commands

The terminal distinguishes between built-in and external commands:

- **Built-in commands:** Implemented internally (e.g., `cd`, `clear`, `exit`, `history`).
- **External commands:** Executed via `fork()` and `execvp()`.

If the command is recognized as built-in, the terminal handles it directly. Otherwise, it is treated as an external command and passed to the process execution function.

2.2.6 Example User Commands

```
user@myterm> cd /mnt/c/Users
user@myterm> ls -l
user@myterm> gcc -o myprog myprog.c
user@myterm> ./myprog
```

Each of these commands is processed, executed, and displayed through the same I/O mechanism, ensuring consistent behavior with a real terminal.

2.2.7 Error Handling

If the command does not exist or execution fails, the `execvp()` call returns and prints an error message to the GUI:

```
perror("execvp failed");
```

This message is appended to the output buffer, allowing the user to see the failure directly in the X11 window.

2.3 Internal Built-in Commands

- Unlike external commands, built-in commands are handled directly by the shell without creating a new process. This allows for operations that need to modify the shell's state, such as changing directories.
- The following built-in commands are implemented:
 1. **cd** [**directory**]: Changes the current working directory of the active tab.
 - If no argument is provided, it defaults to the user's home directory.
 - Error handling is performed if the directory does not exist.
 2. **clear**: Clears the terminal window of all previously displayed lines.
 - Resets the line buffer and scroll offset of the active tab.
 3. **exit**: Terminates the shell.
 - Before exiting, it frees all allocated memory and closes open file descriptors.
 4. **history**: Displays the recent commands executed in the shell.
 - Shows up to 1000 of the most recent commands.
 - Integrated with the searchable history feature for quick retrieval.
 5. **multiWatch** ["cmd1", "cmd2", ...]: Executes multiple commands in parallel and continuously prints their outputs.
 - Each command runs in a separate child process.
 - Output is prefixed with the command name and timestamp.
 - Execution stops gracefully on Ctrl+C, terminating all child processes and deleting temporary files.
 6. **echo** [**text**]: Prints the specified text to the terminal, supporting Unicode and multiline input.

- Built-in commands are detected by the shell before attempting to fork a new process. If a command is recognized as built-in, it is executed directly in the main process, otherwise, it is treated as an external command.

3 Take Multiline Unicode Input

3.1 Objective

This feature enables the terminal to accept **multiline input** containing **Unicode characters**. The user can enter commands spanning multiple lines, and characters from different languages and scripts (e.g., Hindi, Marathi, Hawaiian) are displayed correctly in the terminal GUI.

3.2 Design and Implementation

3.2.1 Unicode Support

To support Unicode input and output, the terminal sets the locale at startup:

```
setlocale(LC_ALL, "");
const char *loc = setlocale(LC_CTYPE, NULL);
if (!loc || !strstr(loc, "UTF-8")) {
    setenv("LANG", "en_US.UTF-8", 1);
    setenv("LC_CTYPE", "en_US.UTF-8", 1);
    setlocale(LC_ALL, "");
}
```

This ensures that all input and output operations correctly interpret multibyte UTF-8 characters.

3.2.2 Multiline Input Handling

If the user wants to enter a command over multiple lines, they can use the backslash ‘\’ at the end of a line. The terminal appends the next line to the current input buffer until the command is complete.

```
if (t->current_len > 0 && t->current_line[t->current_len - 1] ==
    '\\') {
    t->current_line[t->current_len++] = '\\n';
    t->current_line[t->current_len] = '\\0';
    t->cursor_pos = t->current_len;
    continue;
}
```

3.2.3 Reading and Writing Unicode

The terminal uses `read()` and `write()` system calls to handle input and output. The input buffer stores raw UTF-8 bytes, and `draw_ui()` renders characters correctly by respecting multibyte sequences.

```
// Example: append Unicode input to the current line buffer
ssize_t n = read(fd, buf, sizeof(buf));
if (n > 0) {
    append_text(t, buf, (int)n); // handles multibyte UTF-8
    sequences
}
```

3.2.4 Integration with GUI

1. The user types a command in the X11 terminal window.
2. If the command is multiline, lines ending with ‘\’ are continued.
3. Unicode characters from multiple scripts are correctly displayed.
4. When the user presses Enter to complete the command, the input is parsed and executed.

3.2.5 Example User Input

```
user@myterm> echo "Hi \n\  
Hello"
```

Output:

```
Hi  
Hello
```

3.2.6 Error Handling

If invalid UTF-8 sequences are entered, they are preserved in the buffer as-is. The terminal avoids crashing and displays as much valid input as possible. The `setlocale()` call ensures correct interpretation and display of multibyte characters.

3.2.7 Note on Input and Output Display

Although the terminal stores and displays the user input as a **single line** in the input area, the **output** correctly renders all Unicode characters and line breaks.

4 Running External Commands with Input Redirection

4.1 Objective

This feature allows the terminal to execute external commands while redirecting their standard input from a file. Instead of taking input from the keyboard, the command reads from a specified file using input redirection.

4.2 Design and Implementation

4.2.1 Command Parsing

The user command is parsed to identify the input redirection symbol < and the file from which input should be read. The parsing function separates the command name, arguments, and the input file path.

```
int argc = parse_command_line(t->current_line, argv, MAX_ARGS, &
    cmdbuf);
char *input_file = NULL;
for (int i = 0; i < argc; ++i) {
    if (strcmp(argv[i], "<") == 0 && i + 1 < argc) {
        input_file = argv[i + 1];
        argv[i] = NULL; // terminate argv before '<'
        break;
    }
}
```

4.2.2 Process Creation and Input Redirection

When an input file is specified, the terminal opens the file and uses `dup2()` to redirect the file descriptor to standard input (`STDIN_FILENO`) in the child process.

```
pid_t pid = fork();
if (pid == 0) {
    if (input_file) {
        int fd = open(input_file, O_RDONLY);
        if (fd >= 0) {
            dup2(fd, STDIN_FILENO);
            close(fd);
        } else {
            perror("Failed to open input file");
            exit(1);
        }
    }
    dup2(t->to_child[0], STDIN_FILENO);
    dup2(t->from_child[1], STDOUT_FILENO);
    dup2(t->from_child[1], STDERR_FILENO);
    close(t->to_child[1]);
    close(t->from_child[0]);
    execvp(argv[0], argv);
    perror("execvp failed");
    exit(1);
}
```

4.2.3 Input and Output Handling

The parent process continues to monitor the pipes using `select()`, reading output from the child process asynchronously. This allows the GUI to display the command output in real time, exactly as it would appear in a normal terminal.

4.2.4 Example User Commands

```
user@myterm> ./a.out < infile.txt
```

4.2.5 Error Handling

If the input file does not exist or cannot be opened, the terminal displays an error message using `perror()` in the X11 window, ensuring the user can immediately identify the problem.

5 Running External Commands with Output Redirection

5.1 Objective

This feature allows the terminal to execute external commands while redirecting their standard output to a file. Instead of displaying output on the screen, the command writes it to the specified file using output redirection.

5.2 Design and Implementation

5.2.1 Command Parsing

The terminal parses the user input to detect the output redirection symbol `>` and the target file for the output. Both input and output redirection symbols can be handled simultaneously to allow full control over I/O.

```
int argc = parse_command_line(t->current_line, argv, MAX_ARGS, &
    cmdbuf);
char *input_file = NULL;
char *output_file = NULL;
for (int i = 0; i < argc; ++i) {
    if (strcmp(argv[i], "<") == 0 && i + 1 < argc) {
        input_file = argv[i + 1];
        argv[i] = NULL; // terminate argv before '<'
        break;
    }
    if (strcmp(argv[i], ">") == 0 && i + 1 < argc) {
        output_file = argv[i + 1];
        argv[i] = NULL; // terminate argv before '>'
        break;
    }
}
```

5.2.2 Process Creation and Output Redirection

In the child process, the terminal uses `dup2()` to redirect standard output (`STDOUT_FILENO`) to the specified output file. Input redirection can also be applied simultaneously if an input file is specified.

```

pid_t pid = fork();
if (pid == 0) {
    if (input_file) {
        int fd_in = open(input_file, O_RDONLY);
        if (fd_in >= 0) {
            dup2(fd_in, STDIN_FILENO);
            close(fd_in);
        } else {
            perror("Failed to open input file");
            exit(1);
        }
    }
    if (output_file) {
        int fd_out = open(output_file, O_WRONLY | O_CREAT |
            O_TRUNC, 0644);
        if (fd_out >= 0) {
            dup2(fd_out, STDOUT_FILENO);
            close(fd_out);
        } else {
            perror("Failed to open output file");
            exit(1);
        }
    }
    dup2(t->to_child[0], STDIN_FILENO);
    dup2(t->from_child[1], STDERR_FILENO);
    close(t->to_child[1]);
    close(t->from_child[0]);
    execvp(argv[0], argv);
    perror("execvp failed");
    exit(1);
}

```

5.2.3 Input and Output Handling

The parent process monitors the pipes using `select()` and reads any output not redirected to the file. When output redirection is applied, the GUI will not display the command output; it will be written entirely to the specified file. Input redirection, if present, is handled before executing the command to provide the correct data to the child process.

5.2.4 Example User Commands

```

user@myterm> ./a.out > outfile.txt
user@myterm> ls > abc
user@myterm> ./a.out < infile.txt > outfile.txt

```

5.2.5 Error Handling

If the output file cannot be created or opened, the terminal displays an error message in the X11 window using `perror()`, informing the user of the failure. Similarly, if the input

file does not exist, the error is reported immediately.

6 Implementing Support for Pipe

6.1 Objective

This feature enables the terminal to execute multiple commands in a pipeline, where the standard output of one command is fed as the standard input to the next. It mimics the behavior of traditional Unix shells and allows chaining commands using the `|` symbol.

6.2 Design and Implementation

6.2.1 Command Parsing

The terminal parses the input line to detect pipe symbols `|` and separates each command in the sequence. Each command is stored along with its arguments to be executed as an independent process.

```
int n_pipes = 0;
char *cmds[MAX_CMDS][MAX_ARGS];
parse_pipe_commands(t->current_line, cmds, &n_pipes);
```

6.2.2 Pipe Creation and Process Execution

For each command in the pipeline, the terminal creates a pipe using `pipe()`. The standard output of the current command is redirected to the write end of the pipe, and the standard input of the next command is redirected to the read end of the pipe. Each command is executed in a child process created using `fork()`.

```
int pipefd[2], in_fd = STDIN_FILENO;

for (int i = 0; i < n_pipes; ++i) {
    if (i < n_pipes - 1) pipe(pipefd);
    pid_t pid = fork();
    if (pid == 0) {
        if (in_fd != STDIN_FILENO) {
            dup2(in_fd, STDIN_FILENO);
            close(in_fd);
        }
        if (i < n_pipes - 1) {
            dup2(pipefd[1], STDOUT_FILENO);
            close(pipefd[0]);
            close(pipefd[1]);
        }
        execvp(cmds[i][0], cmds[i]);
        perror("execvp failed");
        exit(1);
    } else {
        if (in_fd != STDIN_FILENO) close(in_fd);
        if (i < n_pipes - 1) {
```

```

        close(pipefd[1]);
        in_fd = pipefd[0];
    }
}
}

```

6.2.3 Input and Output Handling

The terminal uses the same asynchronous reading mechanism via pipes to capture output from each command in the pipeline. The output of the final command in the sequence is displayed in the X11 GUI window using `append_text()` and `draw_ui()`.

6.2.4 Example User Commands

```

user@myterm> ls *.txt | wc -l
user@myterm> ls *.txt | xargs rm

```

6.2.5 Error Handling

If any command in the pipeline fails to execute, the terminal prints an error message in the GUI using `perror()`. Pipes are closed properly in both parent and child processes to avoid file descriptor leaks.

7 Implementing the multiWatch Command

7.1 Objective

The `multiWatch` command allows the terminal to execute multiple commands in parallel and continuously display their outputs along with a timestamp and the command name. Unlike the traditional `watch` command, which executes commands sequentially, `multiWatch` runs commands simultaneously in separate processes.

7.2 Command Syntax

```
multiWatch ["cmd1", "cmd2", "cmd3", ...]
```

7.3 Design and Implementation

7.3.1 Main Program (Shell) Workflow

1. Parse the `multiWatch` command and extract all individual commands.
2. For each command:
 - Fork a child process.
 - Create a hidden temporary file `.temp.PID.txt` where PID is the process ID of the child process.
 - Redirect the standard output and standard error of the child to this temporary file.

3. The parent process monitors all temporary files using `select()` or `poll()` for asynchronous output.
4. Whenever new output is available, the parent reads from the file and appends it to the terminal GUI using `append_text()` and `draw_ui()` in the specified format:

"cmd_name" , current_time :

Output

7.3.2 Child Process Execution

Each forked child process executes its respective command using `execvp()`:

```
pid_t pid = fork();
if (pid == 0) {
    int fd = open(temp_filename, O_WRONLY | O_CREAT | O_APPEND,
        0644);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);
    close(fd);
    execvp(argv[0], argv);
    perror("execvp failed");
    exit(1);
}
```

7.3.3 Signal Handling and Termination

- The parent process captures `SIGINT` (Ctrl+C) using `signal()`.
- On receiving Ctrl+C, the terminal:
 1. Kills all child processes using `kill()`.
 2. Closes all file descriptors.
 3. Deletes all temporary files created for `multiWatch`.

7.3.4 Input/Output Management

- Each temporary file acts as a pipe-like interface between the child and the terminal.
- The parent process maintains a mapping of PID to file offsets to read only the new output since the last read.
- The output is displayed in real time in the X11 GUI with the timestamp formatted using `strftime()`.

7.3.5 Example User Command and Output

```
user@myterm> multiWatch ["cmd1", "cmd2", "cmd3"]
```

```
"cmd1" , 2025-10-25 10:15:00 :
```

```
-----  
Output1  
-----
```

```
"cmd2" , 2025-10-25 10:15:00 :
```

```
-----  
Output2  
-----
```

```
"cmd1" , 2025-10-25 10:15:05 :
```

```
-----  
Output1  
-----
```

7.4 Error Handling

Error handling in the `multiWatch` feature is implemented as follows:

- **Process creation errors:** If forking a child process fails or the execution of a command fails, the terminal displays an error message in the GUI, e.g., "Failed to start multiWatch" or `perror("execvp failed")`.
- **File I/O errors:** Temporary files used to capture command output are checked when opening and reading. If opening a file fails, the terminal skips writing output, preventing crashes. Read errors close the file descriptor and mark it as invalid.
- **Signal handling:** Pressing `Ctrl+C` terminates all child processes associated with `multiWatch` and deletes temporary files to prevent leftover artifacts.
- **Memory management:** Dynamically allocated strings for command output are freed after use. While `malloc()` failures are not explicitly checked, the program ensures that a NULL pointer does not get written, avoiding segmentation faults.

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

8.1 Objective

This feature enables command-line editing shortcuts similar to Bash, allowing users to navigate quickly within the current input line. Specifically:

- **`Ctrl+A`** moves the cursor to the start of the line.
- **`Ctrl+E`** moves the cursor to the end of the line.

This makes the terminal input experience more intuitive and closer to standard shell behavior.

8.2 Design and Implementation

8.2.1 Capturing Keypresses

- Set the terminal into **raw mode** using the `termios` library to read keypresses immediately without waiting for a newline.
- Use `read()` to capture individual bytes from the input.
- Detect control key combinations by their ASCII values:
 - `Ctrl+A` = `0x01`
 - `Ctrl+E` = `0x05`

8.2.2 Cursor Management

- Maintain a `cursor_pos` index in the input buffer that tracks the current cursor location.
- When `Ctrl+A` is pressed, set `cursor_pos` = 0.
- When `Ctrl+E` is pressed, set `cursor_pos` = `current_len` (end of input).

8.2.3 Integration with Input Buffer

- The input buffer stores the user-typed line, including multiline Unicode input if applicable.
- Navigation commands only change `cursor_pos` and do not alter the buffer contents.
- After cursor movement, the line is re-rendered using `draw_ui()` in the X11 GUI.

8.2.4 Example Behavior

```
user@myterm> echo "Hello World"
Ctrl+A  # Cursor moves to start
Ctrl+E  # Cursor moves to end
```

8.3 Error Handling

The terminal handles errors and edge cases for line navigation as follows:

- Pressing `Ctrl+A` moves the cursor to the start of the current line, while `Ctrl+E` moves it to the end. The cursor position is bounded by the input buffer length, preventing buffer overflows.
- Unsupported keypresses or control combinations are ignored, ensuring that invalid inputs do not crash the terminal.
- Raw input is read using `read()`. If `read()` fails or returns 0, no changes are applied to the cursor position, preventing unexpected behavior.

9 Interrupting Commands Using Signals

9.1 Objective

This feature allows the user to control running commands using keyboard signals:

- **Ctrl+C:** Immediately terminates the currently running command and returns control to the shell prompt.
- **Ctrl+Z:** Suspends the currently running command, moves it to the background, and returns control to the shell prompt.

9.2 Design and Implementation

9.2.1 Signal Handling

The terminal sets up signal handlers using the `signal()` system call:

- **SIGINT** (Ctrl+C) is captured to terminate the foreground process group without exiting the shell.
- **SIGTSTP** (Ctrl+Z) is captured to suspend the foreground process and place it into the background.

9.2.2 Foreground Process Management

- Each tab keeps track of the currently running process PID.
- On receiving **SIGINT**, the shell sends `kill(-pgid, SIGINT)` to the process group to terminate the command.
- On receiving **SIGTSTP**, the shell sends `kill(-pgid, SIGTSTP)` to suspend the command and updates the tab state to indicate a background process.

9.3 Error Handling

- **Invalid PIDs:** If the process has already terminated, sending a signal will fail silently, preventing shell crashes.
- **Pipe closure:** If the process's pipes are closed, the shell properly closes file descriptors and marks the process as inactive.
- **Multi-process scenarios:** In the case of `multiWatch` or background jobs, signals are sent to all relevant PIDs to ensure consistent behavior.
- **Memory Safety:** Lines added to display signal events are managed properly, freeing previous allocations if necessary to prevent memory leaks.

10 Implementing a Searchable Shell History

10.1 Objective

This feature enables the terminal to maintain and search through a history of previously executed commands, providing Bash-like history search functionality:

- Maintains the last 10,000 commands in a persistent history file.
- Implements a `history` command to display the most recent 1000 commands.
- Supports interactive search using `Ctrl+R` to find commands from history.

10.2 Design and Implementation

10.2.1 History Storage

- Commands entered in each tab are stored in a ring buffer of 10,000 entries.
- On shell startup, the history file is loaded into memory using `load_history_file()`.
- On command execution, the new command is appended to both the in-memory history buffer and the history file.

10.2.2 History Display Command

- The `history` command displays the last 1000 commands.
- Commands are printed in chronological order with indices.
- GUI integration ensures output is displayed within the X11 terminal window.

10.2.3 Interactive Search (`Ctrl+R`)

- When the user presses `Ctrl+R` (ASCII 0x12), the shell enters search mode.
- A prompt "`Enter search term`" is displayed in the GUI.
- User input is captured character-by-character using `read()`, supporting Unicode input.
- Upon pressing Enter, the search algorithm looks for the most recent command matching the search term exactly.
- If no exact match exists, the algorithm computes the longest substring match among all commands with length greater than 2 characters and displays those commands.
- If no suitable match is found, the message "`No match for search term in history`" is displayed.

10.2.4 GUI Integration

- The search prompt and resulting commands are displayed in the active tab of the X11 window.
- Input editing, including backspace handling and cursor movement, is supported during search mode.
- After search completion, the shell returns to normal input mode.

10.3 Error Handling

- **Empty history file:** If the history file cannot be loaded or is empty, the shell initializes an empty buffer and displays an informative message.
- **Invalid input:** Non-printable or excessively long search strings are safely ignored or truncated to prevent buffer overflow.
- **Memory management:** Dynamically allocated strings for search results are freed after display to avoid memory leaks.
- **No matches found:** Proper messages are displayed in the GUI instead of causing errors.

11 Implementing Auto-Complete Feature for File Names

11.1 Objective

This feature enables the shell to assist the user by auto-completing file names in the current working directory:

- Provides real-time suggestions when the user presses the Tab key.
- Supports single matches, multiple matches, and prompts the user to choose if multiple possibilities exist.
- Enhances usability and efficiency while typing commands.

11.2 Design and Implementation

11.2.1 Detection and Trigger

- Auto-complete is triggered when the user presses the Tab key.
- The shell captures the current input line and extracts the partial file name after the last space or command delimiter.
- The file name prefix is then matched against all files in the current working directory.

11.2.2 Single Match

- If exactly one file matches the given prefix, the shell appends the remaining characters of the file name to the current input.
- The cursor and input buffer are updated accordingly, preserving the rest of the command.

11.2.3 Multiple Matches

- If multiple files match the prefix, the shell computes the longest common prefix among all matches.
- The input line is updated with this common prefix.
- If ambiguity still exists, the shell prompts the user with a numbered list of matching files.
- The user selects a file by entering the corresponding number, and the shell completes the input line with the chosen file name.

11.2.4 No Matches

- If no files match the given prefix, the shell does not modify the input line or provide suggestions.

11.2.5 Example Scenarios

- Directory contains "abc.txt", "def.txt", "abcd.txt".
- Input: ./myprog de + Tab → Output: ./myprog def.txt.
- Input: ./myprog def.txt abc + Tab → Output: 1. abc.txt 2. abcd.txt.
User selects 1 → Input: ./myprog def.txt abc.txt.
- Even if the program (e.g., myprog) does not exist, auto-complete works for file names in the current directory.

11.2.6 Integration with GUI

- The auto-completed text is displayed in the active tab of the X11 window.
- The cursor position is updated correctly to reflect the auto-completion.
- The shell waits for further user input after auto-completion.

11.3 Error Handling

- **Directory read errors:** If the current directory cannot be read, no suggestions are shown, and an error is optionally logged.
- **No matching files:** Safely handles the case with no matches by not modifying the input line.

- **Memory management:** Dynamically allocated memory for matched file names is freed after use to avoid memory leaks.
- **Invalid user selection:** If the user selects an invalid number when prompted, the shell ignores the input and returns to normal mode.