MyTerm Design Document

CS69201 — CL Lab Project

Developed by: Rounak Neogy (25CS60R55)
Department of Computer Science & Engineering, IIT Kharagpur

October 10, 2025

Contents

1 System Overview

1.1 Objective

The objective of the **MyTerm** project is to design and implement a fully functional **graphical terminal emulator** from scratch using the C programming language and the **X11** windowing system.

MyTerm provides a custom-built GUI that closely mimics the behavior of a Unix shell while offering several enhanced interactive features. It integrates low-level **system programming concepts** such as `fork()`, `execvp()`, `pipe()`, `dup2()`, and signal handling with advanced GUI rendering and event handling through X11.

The terminal supports multiple independent tabs, interactive command editing, history search, Unicode input, and concurrent process execution through `multiWatch`. The design emphasizes modularity, concurrency, and user interactivity—bridging the gap between traditional terminal functionality and graphical usability.

1.2 Key Features

- **Graphical terminal interface using X11:** A custom windowed environment built entirely with Xlib, handling keyboard and mouse events via functions such as `XNextEvent()` and rendering text using `XDrawString()`. The GUI replicates a standard terminal while providing visual feedback and multi-tab management.

- **Execution of external commands:** Supports standard Unix commands executed using the `fork()`–`execvp()` model. Command output is captured through pipes and displayed directly inside the X11-rendered window.

- **Multiline Unicode input:** Allows multi-line input and Unicode character support through `setlocale()` and UTF-8 safe cursor navigation. Enables multilingual text and proper alignment across multi-byte characters.

- **Input redirection (<):** Redirects standard input from files using `dup2()` and `open(O_RDONLY)`, allowing commands to read data from files rather than keyboard input.

- **Output redirection (>) and combined redirection:** Redirects command output to files using `dup2()` with `O_WRONLY | O_CREAT | O_TRUNC`. Supports both input and output redirection simultaneously (e.g., `./a.out < in.txt > out.txt`).

- **Pipe support (`|`):** Implements Unix-style command chaining where the output of one process serves as the input of another. Uses multiple `pipe()` calls and proper descriptor management to handle commands like `ls | grep .c | wc -l`.
- **Parallel command monitoring with `multiWatch`:** Executes multiple commands concurrently, each in its own child process. Displays timestamped outputs in real-time within the GUI, providing continuous system monitoring similar to an enhanced `watch` command.
- **Command-line navigation (Ctrl+A, Ctrl+E):** Implements Bash-like navigation shortcuts for editing commands. `Ctrl+A` moves the cursor to the start of the input line, while `Ctrl+E` moves it to the end.
- **Command interruption and suspension (Ctrl+C, Ctrl+Z):** Provides robust signal handling to manage process states. `Ctrl+C` interrupts the active foreground command, while `Ctrl+Z` suspends it and sends it to the background — keeping the terminal alive and interactive.
- **Searchable shell history (Ctrl+R):** Maintains a persistent history of the last 10,000 commands across sessions in `~/.myterm_history`. Pressing `Ctrl+R` opens an interactive search mode for exact or closest substring matches.
- **Auto-complete for filenames:** Implements intelligent filename autocompletion triggered by the Tab key. Scans the current directory using `opendir()` and `readdir()` for matches, supports prefix completion, and prompts for user choice when multiple candidates exist.
- **Extra feature — Clipboard paste support:** Allows users to paste full commands (via mouse or keyboard shortcuts like Ctrl+V) directly into the terminal input buffer. Automatically detects pasted input sequences and integrates them seamlessly into the current command line.

1.3 Technology Stack

- **Programming Language:** C (C11 Standard)
- **Core Libraries:** X11 (for GUI rendering and event handling), pthreads (for concurrency), POSIX semaphores (for synchronization), and termios (for input mode control)
- **System Calls Used:** `fork()`, `execvp()`, `dup2()`, `pipe()`, `poll()`, `waitpid()`, `kill()`
- **Build System:** GNU Make
- **Executable Output:** `./myterm`
- **Operating System Compatibility:** Linux / macOS / Unix-based systems with X11 server support

2 System Architecture

2.1 High-Level Architecture

The **MyTerm** system follows a modular, event-driven design that integrates low-level process control with a graphical interface. The architecture separates the GUI, command execution, and data management layers for maintainability and concurrency safety.
The architecture is divided into four major layers:

- **GUI Layer (X11 Event Handling):** Responsible for rendering text, handling keyboard and mouse inputs, and managing terminal tabs. Events are processed through the Xlib interface using functions such as `XNextEvent()`, `XMapWindow()`, and `XDrawString()`. The GUI runs the main event loop, ensuring responsiveness and asynchronous updates.
- **Command Execution Layer:** Manages execution of commands entered by the user. It supports forking new processes using `fork()` and executing programs with `execvp()`. Standard input/output redirection, pipe creation, and synchronization with the GUI are handled in this layer.
- **History and Autocomplete Subsystem:** Maintains command history across sessions and implements the Tab-based autocomplete mechanism. History is persistently stored in a file ($\tilde{/}$`.myterm_history`), while autocomplete dynamically scans the working directory for possible matches.
- **Process Control and Signal Handling:** Handles signal forwarding (`SIGINT`, `SIGTSTP`) to child processes, allowing users to pause or terminate running commands. The shell itself remains unaffected, ensuring continued usability and stability.

Each component communicates through shared data structures and pipes, maintaining clear separation between the GUI event loop and the process execution subsystems.

2.2 Major Source Modules

The implementation of MyTerm is distributed across several source files, each dedicated to a specific subsystem. These modules collectively enable GUI rendering, process management, history maintenance, and user interaction.

- **main.c:** The entry point of the program. Initializes the X11 display and window, sets up event listeners, and handles tab management (create, switch, close). It dispatches keyboard input and signals to appropriate modules.
- **shell_tab.c:** Implements the logic for managing multiple tabs within the terminal. Each tab is an independent shell environment maintaining its own buffers, history, and process states.
- **cmd_exec.c:** Executes user-entered shell commands using the `fork()`–`execvp()` model. Supports input/output redirection (`<`, `>`) and pipes (`|`). Handles background processes and integrates with the signal manager for `Ctrl+C` and `Ctrl+Z` operations.
- **multiwatch.c:** Implements the `multiWatch` command for concurrent monitoring of multiple commands. Manages multiple subprocesses and uses `poll()` to capture and timestamp outputs dynamically.
- **history.c:** Handles persistent command history. Maintains a circular buffer of the most recent 10,000 commands and performs both exact and approximate search (via longest-common-substring) for `Ctrl+R` reverse lookup.
- **autocomplete.c:** Provides file name auto-completion using the Tab key. Lists matching candidates and, in the case of multiple matches, prompts the user for selection.
- **line_edit.c:** Manages user input editing and cursor navigation. Implements text insertion, deletion, and shortcuts like `Ctrl+A` (jump to start) and `Ctrl+E` (jump to end). Handles multibyte Unicode input via UTF-8-safe cursor positioning.

- **multiwatch.c:** Manages concurrent command execution and output synchronization across multiple child processes using semaphores and non-blocking I/O.

- **multiwatch.c:** Manages concurrent command execution and output synchronization across multiple child processes using semaphores and non-blocking I/O.

2.3 Header Files

Header files define shared data structures, macros, and function prototypes that enable inter-module communication and maintain modularity. Below is the overview of all header files in the project.

- **main.h:** Declares global constants, includes system headers, and defines function prototypes for initialization and signal handling.
- **shell_tab.h:** Declares the `Tab` structure and associated functions for creating, switching, and closing tabs. It also defines interfaces for rendering tab outputs and managing tab-local state.
- **cmd_exec.h:** Contains function declarations for command parsing, execution, piping, and redirection. Also defines macros for buffer size limits and error handling.
- **multiwatch.h:** Defines the `MultiWatchState` structure, containing process IDs, file descriptors, and semaphores used for concurrent monitoring. Declares entry points for starting and stopping the multiWatch feature.
- **history.h:** Defines the `HistoryBuffer` structure and exposes APIs for initializing, saving, loading, and searching command history. Manages persistent storage integration.
- **line_edit.h:** Declares the `LineEditor` structure for managing the input buffer, cursor position, and editing state. Provides function prototypes for editing actions and input manipulation.
- **autocomplete.h:** Provides declarations for file scanning and prefix matching functions used by the autocomplete system. Defines interfaces for Tab-based input completion and suggestion rendering.
- **multiwatch.h:** Declares data structures and helper functions for multi-command polling, file descriptor management, and output formatting.
- **common.h:** Contains cross-module constants, macros, and shared type definitions used by multiple modules (e.g., buffer limits, key mappings, ANSI color codes).
- **utils.h:** Defines generic utility functions used throughout the system, including string manipulation, timestamp generation, longest common substring computation, and ANSI escape stripping.

2.4 Data Structures

The project uses several key data structures that form the backbone of the terminal emulator's functionality.

- `Tab` Represents a single shell tab, containing:
  - Input and output buffers
  - Process IDs for active commands

  - History index and list
  - File descriptors for I/O pipes
  - Synchronization flags for rendering and event handling
- `LineEditor` Manages user input line with:
  - UTF-8 safe character buffer
  - Cursor position and editing mode
  - Flags for terminal redraws
- `MultiWatchState` Handles parallel monitoring of commands with:
  - Array of process IDs
  - File descriptors for temporary files
  - Mutexes and semaphores for synchronization
  - Stop and pause flags for signal-based control
- `HistoryBuffer` Maintains command history using:
  - Circular buffer of up to 10,000 entries
  - File handle for persistent storage
  - Reverse search cursor position for `Ctrl+R`

These data structures collectively ensure that input handling, command execution, and output rendering operate independently yet cohesively, preserving performance and responsiveness across concurrent terminal sessions.

3 Design Decisions and Rationale

The design of **MyTerm** was guided by three key goals: *responsiveness*, *modularity*, and *extensibility*. Each major subsystem was implemented with these principles in mind to create a robust, interactive terminal emulator capable of running concurrent processes while maintaining a fluid graphical interface.

3.1 Event-Driven GUI Architecture

A core decision was to adopt an **event-driven architecture** built on the X11 event loop. Instead of relying on blocking I/O or multi-threaded polling for GUI operations, MyTerm listens for X events using `XNextEvent()` and responds asynchronously to user inputs (keyboard, mouse, or tab actions).

This design ensures:

- High responsiveness even when executing background commands.
- Clear separation between input handling, rendering, and command execution.
- Minimal CPU overhead, since the GUI reacts only to actual user or system events.

By maintaining the main event loop in a single thread and offloading process execution to child processes, MyTerm avoids race conditions and deadlocks common in multi-threaded GUI systems.

3.2 Per-Tab Isolation and Independent Buffers

Each terminal tab in MyTerm is implemented as a logically isolated shell instance with its own:

- Input and output buffers
- Process ID list
- Command history stack
- Line editor state

This architectural choice prevents interference between parallel sessions and improves fault tolerance—an error or process crash in one tab does not affect others. The per-tab design also simplifies concurrency management, since synchronization is localized to tab-level mutexes rather than global locks. This mirrors the modularity of modern multi-tab terminals like GNOME Terminal, but implemented from scratch in C with Xlib primitives.

3.3 Polling-Based Non-Blocking I/O for MultiWatch

For the `multiWatch` feature, which runs multiple commands concurrently, non-blocking reads were essential to achieving real-time updates. Instead of using separate threads for each subprocess, MyTerm uses the `poll()` system call to monitor all output file descriptors simultaneously.
Rationale:

- Polling provides deterministic control over I/O readiness without thread scheduling overhead.
- It scales efficiently with multiple processes.
- It allows precise timing and formatted output updates with timestamps.

This design minimizes synchronization complexity, making `multiWatch` both efficient and safe. The use of semaphores and mutexes ensures atomic access to shared buffers during concurrent output rendering.

3.4 Process Control and Signal Safety

Another major decision was to handle process control via UNIX signals (`SIGINT`, `SIGTSTP`, etc.) rather than ad-hoc flag-based termination. The shell intercepts signals and forwards them only to the currently running foreground child, ensuring:

- `Ctrl+C` interrupts the running command but not the shell itself.
- `Ctrl+Z` suspends the command to the background cleanly.
- The shell remains active and ready for new input after signal delivery.

This separation of control flow between shell and child processes mirrors the philosophy of traditional Unix shells like Bash, implemented at the system-call level in C.

3.5 Persistence Through Minimalism

The project intentionally avoids high-level frameworks or external GUI libraries beyond X11. This decision was driven by two primary motivations:

- To maintain low-level control over rendering, input capture, and process management.

- To ensure portability across Unix systems without dependency conflicts.

All features—tab management, autocomplete, history, and multiWatch—were implemented using standard POSIX system calls and C data structures, ensuring full transparency of how the terminal operates internally.

3.6 Simplicity and Extensibility

Throughout the design, simplicity and future extensibility were prioritized:

- The codebase is modular, with one module per logical subsystem (GUI, history, autocomplete, etc.).
- Each module exposes a clean header interface, allowing independent modification or extension.
- The system supports adding new features (e.g., mouse scroll, theme customization) without altering the event core.

This modular approach made debugging, integration, and incremental feature addition significantly easier, ensuring that new functionalities (like scrollable history or clickable buttons) can be integrated in future iterations with minimal changes to existing logic.

4 Feature-Wise Design Details

Each major feature of **MyTerm** was designed with modularity, responsiveness, and accuracy in mind. This section explains the design and implementation of all functional components, including algorithms, synchronization mechanisms, and GUI integrations.

4.1 Graphical User Interface (X11)

The graphical interface forms the backbone of **MyTerm**, implemented using the **X11** library. It provides a custom windowed terminal that mimics the behavior of a traditional Bash shell while allowing multiple concurrent tabbed sessions.

Key Components:

- **Display Initialization:** The X server is accessed using `XOpenDisplay()`, and a window is created with `XCreateSimpleWindow()`.
- **Font and Text Rendering:** Output text is drawn using `XDrawString()` for ASCII and `Xutf8DrawString()` for Unicode characters.
- **Event Loop:** Input and exposure events are continuously handled via `XNextEvent()`.
- **Tab Management:** Tabs are drawn as rectangular regions at the top. The active tab is highlighted with a distinct color for easy identification.

Pseudocode:

Listing 1: Simplified Event Loop

```
while (1) {
    XNextEvent(display, &event);
    switch (event.type) {
        case Expose: redraw_all_tabs(); break;
        case KeyPress: handle_keypress(event.xkey); break;
        case ButtonPress: handle_mouse_click(event.xbutton); break;
        }
}
```

User Interaction:

- **F1** — Open a new tab.
- **F2** — Switch between existing tabs.
- **F3** — Close the current tab.

Each tab represents an isolated shell instance, ensuring parallel command execution without interference.

4.2 Run External Commands

All user commands are executed through the `fork()`–`execvp()` model, replicating native Linux shell behavior.

Implementation Steps:

1. Tokenize user input using `strtok()`.
2. Call `fork()` to create a new child process.
3. In the child:
   - Handle redirection and piping if necessary.
   - Execute the command using `execvp()`.
4. In the parent:
   - Wait for the child process with `waitpid()`.
   - Capture and render output in the GUI.

Rationale: Using `fork()` and `execvp()` preserves UNIX shell semantics while providing controlled execution and easy signal handling.

4.3 Multiline Unicode Input

MyTerm supports multiline input and full Unicode text handling to enable multi-language command entry.

Design Details:

- Unicode enabled with `setlocale(LC_ALL, "en_US.UTF-8")`.
- Cursor navigation functions — `nextcharoffset()` and `prevcharoffset()` — prevent breaking multibyte UTF-8 characters.
- Multiline input is recognized when a line ends with a trailing backslash (\), buffering the input until completion.

Example:

```
echo "We say Hello in English \n\
      \n\
Aloha mākou ma ka ōlelo Hawaii \n\
  "
```

The terminal correctly prints all multilingual text and line breaks as expected.

4.4 Input Redirection (¡)

Input redirection allows a command to read input from a file instead of the keyboard.
Algorithm:

1. Detect the `<` operator and extract the filename.
2. Open the file using `open(..., O_RDONLY)`.
3. Use `dup2()` to replace `STDIN_FILENO` with the file descriptor.
4. Execute the command via `execvp()`.

Example:

```
int fd = open("input.txt", O_RDONLY);
dup2(fd, STDIN_FILENO);
close(fd);
execvp(args[0], args);
```

This mechanism ensures the command reads data directly from the specified file.

4.5 Output Redirection (¿)

Output redirection sends a program's output to a file rather than to the terminal.
Implementation:

1. Detect `>` and extract the filename.
2. Open the file using `open(..., O_WRONLY | O_CREAT | O_TRUNC, 0644)`.
3. Redirect `STDOUT_FILENO` using `dup2()`.

Example:

```
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(fd, STDOUT_FILENO);
execvp(args[0], args);
```

Combination: Input and output redirection can coexist:

```
./a.out < infile.txt > outfile.txt
```

4.6 Pipe Support (—)

MyTerm supports pipe chaining, where the output of one command acts as the input to the next.
Design:

1. Parse the command by splitting at each `|`.
2. Create pipes between adjacent commands.
3. Redirect `stdin` and `stdout` accordingly using `dup2()`.
4. Close all unused pipe ends.

Example Workflow:

```
ls *.txt | grep log | wc -l
```

The data flows sequentially: `ls` → `grep` → `wc`, ensuring UNIX-style pipe execution.

4.7 MultiWatch Command

The `multiWatch` feature enables concurrent monitoring of multiple commands, showing real-time output tagged with timestamps and command identifiers.

Implementation Steps:

1. Parse command list: `multiWatch ["cmd1", "cmd2", ...]`.
2. Fork a process for each command.
3. Redirect output to temporary files (`.temp.PID.txt`).
4. Use `poll()` to monitor all file descriptors simultaneously.
5. Periodically read outputs and display them with timestamps.

Termination: `Ctrl+C` gracefully stops all child processes and deletes the temporary files.

Synchronization: `poll()` ensures non-blocking reads, while mutexes synchronize output rendering across threads.

4.8 Line Navigation with Ctrl+A and Ctrl+E

Implemented bash-like shortcuts for quick cursor navigation within the input line.

Functionality:

- `Ctrl+A` — Move cursor to the start of the line.
- `Ctrl+E` — Move cursor to the end of the line.

Implementation Snippet:

```
if (key == CTRL_A)
    le->cursor = 0;
else if (key == CTRL_E)
    le->cursor = le->len;
redraw_line(le);
```

This provides a familiar editing experience and improves typing efficiency.

4.9 Command Interruption and Suspension

Signal handling ensures smooth user control over running processes.

Key Bindings:

- `Ctrl+C` — Sends `SIGINT` to terminate the current process.
- `Ctrl+Z` — Sends `SIGTSTP` to suspend the process.

Implementation:

```
void handle_sigint(int sig) {
    if (active_pid > 0) kill(active_pid, SIGINT);
}
void handle_sigtstp(int sig) {
    if (active_pid > 0) kill(active_pid, SIGTSTP);
}
```

This ensures that the parent shell remains alive while managing background and interrupted tasks gracefully.

4.10 Searchable Shell History

A persistent history system stores up to 10,000 commands and allows real-time reverse search.
Design Overview:

- History stored in a circular buffer for efficient access.
- Persisted across sessions in ~/.myterm_history.
- Accessible using the `history` command or `Ctrl+R`.

Search Algorithm:

1. On pressing `Ctrl+R`, a search prompt appears.
2. The terminal checks for exact matches first.
3. If not found, it searches for the longest substring match.

Example Output:

```
No match for search term in history
```

Persistence Implementation:

```
void save_history() {
    FILE *f = fopen(history_path, "w");
    for (int i = 0; i < hist_count; i++)
        fprintf(f, "%s\n", hist_buf[i]);
    fclose(f);
}
```

4.11 Auto-Complete for File Names

The Tab key triggers dynamic filename auto-completion within the current working directory.
Steps:

1. Detect the partial filename under the cursor.
2. Scan directory entries using `opendir()` and `readdir()`.
3. Match all filenames starting with the prefix.
4. If a single match is found, auto-complete it.
5. If multiple matches exist, display an indexed list for user selection.

Example:

```
./myprog de [Tab]
1. def.txt  2. demo.txt
```

Selecting 1 completes the command as:

```
./myprog def.txt
```

Supports both absolute and relative path completions.

4.12 Extra Feature — Paste Command Support

The terminal supports command pasting from the clipboard, allowing users to insert multi-line commands effortlessly.

Design:

- Rapid multi-character input is recognized as a paste event.
- Input buffer expands dynamically to accommodate long text.
- Cursor automatically moves to the end after insertion.

Benefit: Users can paste long commands, scripts, or configuration blocks directly into MyTerm without losing formatting or encountering input lag.

5 Synchronization and Concurrency

The design of **MyTerm** carefully integrates concurrency control mechanisms to ensure smooth multitasking between command execution, GUI updates, and inter-process communication. Since multiple threads and processes operate simultaneously—especially during multi-tab execution, background jobs, and the `multiWatch` feature—proper synchronization was essential to avoid data corruption and ensure responsiveness.

5.1 Overview

Concurrency in MyTerm arises from:

- Concurrent reading and writing between parent and child processes through pipes.
- Parallel execution of multiple shell commands via `multiWatch`.
- Simultaneous user input handling and GUI event rendering in the main thread.

To achieve correct coordination, MyTerm employs:

- POSIX semaphores for synchronizing access to shared I/O buffers.
- Pthread mutex locks to guard tab-level data structures.
- Dedicated threads for non-blocking I/O and GUI updates.

—

5.2 Semaphores for Shared Buffer Synchronization

Shared buffers are used to hold data received from child process pipes before being rendered to the X11 GUI. Because multiple threads (e.g., reader threads and GUI event handlers) can attempt to access the buffer simultaneously, semaphores ensure controlled access.

Implementation:

- A binary semaphore (`sem_t io_sem`) is initialized with value 1.
- Each reader thread performs:

```
sem_wait(&io_sem);
// Write output to display buffer
sem_post(&io_sem);
```

- The semaphore ensures mutual exclusion while writing to the shared buffer, preventing partial writes or corrupted output.

Rationale: Semaphores provide lightweight synchronization for I/O operations, allowing safe concurrent writes from multiple subprocesses (e.g., during `multiWatch`) without blocking the entire terminal.

—

5.3 Mutex Protection for Tab Structures

Each tab in MyTerm maintains independent state — including input buffers, display buffers, process IDs, and history lists. To prevent data inconsistencies when multiple threads access these structures, mutexes are employed.

Implementation Details:

- A `pthread_mutex_t tab_lock` is associated with each tab.
- Before updating tab state (e.g., writing new command output or switching tabs), the mutex is locked.
- After the operation completes, the mutex is released.

Code Snippet:

```
pthread_mutex_lock(&tab->lock);
update_display_buffer(tab, data);
pthread_mutex_unlock(&tab->lock);
```

Purpose: This ensures:

- Only one thread modifies tab data at a time.
- GUI thread can safely read buffer data while I/O threads write to it.
- Tab switching (F1/F2/F3 events) never leads to undefined behavior.

—

5.4 Threaded I/O for Non-Blocking GUI Rendering

A major challenge in terminal design is avoiding GUI freezes during command execution or heavy I/O operations. To solve this, MyTerm offloads all blocking reads and process monitoring to separate threads.

Thread Responsibilities:

- Each tab has a dedicated **reader thread** that continuously reads from its child process's `stdout` pipe.
- The main thread remains dedicated to handling X11 events (keyboard, tab switching, redraw).
- Reader threads write data into the shared display buffer (protected by semaphores and mutexes).

Workflow Example:

1. A new command is executed.
2. The main thread spawns a reader thread:

```
pthread_create(&tab->reader_thread, NULL, reader_func, tab);
```

3. The thread reads from the pipe using:

```
while (read(pipe_fd, buf, sizeof(buf)) > 0) {
    sem_wait(&io_sem);
    append_to_display(tab, buf);
    sem_post(&io_sem);
    request_gui_redraw();
}
```

4. The GUI thread remains responsive, processing events in real time.

Outcome: Even when long-running processes (e.g., `ping` or `multiWatch`) produce continuous output, the terminal remains responsive and interactive.

—

5.5 Synchronization in MultiWatch Execution

The `multiWatch` feature executes several commands concurrently. Each command's output is monitored and merged into the terminal's display buffer without interleaving or race conditions.

Design Highlights:

- A separate reader thread is spawned for each command.
- Each thread writes tagged output (command name + timestamp) to a common buffer.
- Access to this buffer is controlled using a semaphore.
- A central polling loop monitors thread states and triggers periodic screen refreshes.

Example:

```
for (i = 0; i < cmd_count; i++) {
    pthread_create(&threads[i], NULL, mw_reader, &cmds[i]);
}
for (i = 0; i < cmd_count; i++) {
    pthread_join(threads[i], NULL);
}
```

This ensures that outputs from different commands (e.g., `ping`, `date`, `ls`) are merged consistently with accurate timestamps.

—

5.6 Avoiding Deadlocks and Race Conditions

Challenges: Concurrent GUI updates, I/O buffering, and tab operations risked race conditions when multiple threads accessed shared data.

Solutions:

- All shared structures (buffers, tab state) are protected by mutexes.
- Semaphores regulate access order during simultaneous output appends.
- The GUI thread performs `trylock()` operations to avoid blocking on mutexes during rendering.

- Threads periodically yield via `usleep()` to maintain fairness and prevent starvation.

Result: The final architecture achieves stable concurrency where multiple commands, tabs, and outputs can coexist without data races or GUI stalling.

—

6 Challenges and Solutions

During the development of **MyTerm**, several system-level and architectural challenges were encountered. This section outlines the key issues faced and the design strategies adopted to overcome them.

6.1 Concurrent I/O and Synchronization

Challenge: When multiple commands or background processes were producing output simultaneously (especially during `multiWatch`), inconsistent rendering and partial data writes occurred in the shared display buffer.

Cause: Concurrent access from multiple reader threads led to race conditions and interleaved writes to the same buffer region, occasionally corrupting displayed text.

Solution: Introduced fine-grained synchronization using `pthread_mutex_t` and POSIX semaphores. Each thread locks the buffer before appending data, ensuring atomic writes:

```
pthread_mutex_lock(&tab->lock);
append_output(tab, data);
pthread_mutex_unlock(&tab->lock);
```

This eliminated overlapping writes and ensured deterministic, thread-safe rendering of concurrent process outputs.

6.2 Signal Propagation and Process Isolation

Challenge: Pressing `Ctrl+C` or `Ctrl+Z` initially caused the entire terminal to terminate or freeze, instead of only affecting the foreground process.

Cause: The terminal process itself inherited the same signal handlers as its child processes, resulting in unintended signal propagation.

Solution: Explicit signal handling logic was added in the parent shell to isolate child process signals:

```
void handle_sigint(int sig) {
    if (active_pid > 0)
        kill(active_pid, SIGINT);
}
```

This ensured that only the currently running command received the interrupt or suspend signal, while the parent terminal remained active and ready for further input. Additionally, a confirmation message was displayed when a process was paused, improving user feedback.

6.3 Redraw Flicker and GUI Responsiveness

Challenge: Frequent full-screen redraws during continuous output (e.g., `ping` or `multiWatch`) led to visible flickering and degraded rendering performance.

Cause: Each incoming line triggered a full X11 window refresh via `XClearWindow()` and `XDrawString()`, which caused redundant repainting of static regions.

Solution: Optimized GUI rendering using **partial redraws** and region-specific updates. Only the modified sections of the text buffer were redrawn during each cycle, minimizing flicker and improving throughput. Redraw frequency was further regulated using small controlled delays:

```
usleep(10000);   // Prevent X11 overload during heavy output
```

This resulted in smooth, real-time updates while maintaining responsive keyboard and tab-switching performance.

6.4 History Persistence and File Locking

Challenge: Concurrent writes to the history file (`.myterm_history`) during rapid session switches caused occasional truncation or corrupted entries.

Solution: Introduced a file-level lock during history save operations using `flock()` to ensure atomic writes:

```
int fd = open(history_path, O_WRONLY | O_CREAT, 0644);
flock(fd, LOCK_EX);
write(fd, data, strlen(data));
flock(fd, LOCK_UN);
```

This guarantees safe persistence of command history across sessions even under concurrent access.

6.5 Thread Lifecycle Management

Challenge: Reader threads occasionally persisted after tab closure or process termination, leading to resource leaks and stray file descriptors.

Solution: Each thread checks a global termination flag and gracefully exits when its associated tab or process is closed. `pthread_join()` ensures cleanup:

```
tab->active = false;
pthread_join(tab->reader_thread, NULL);
```

This provided clean resource deallocation and stable long-term performance during extended use.

—

7 Conclusion

The **MyTerm** project successfully combines the functionality of a Unix shell with an interactive graphical interface built using X11. It integrates key features such as process management, I/O redirection, history tracking, auto-completion, and signal handling into a unified, responsive environment.

The modular architecture—separating GUI, shell logic, and concurrency control—ensures smooth execution and maintainability. By implementing these features entirely in C using POSIX and X11, **MyTerm** demonstrates an efficient and extensible approach to modern terminal design while retaining the simplicity and power of traditional Unix systems.