MyTerm shell - Design Document

CS69201 Computing Lab Project

Sumit - 25CS60R63

October 2025

Table of Contents

# Executive Summary

MyTerm is a custom terminal emulator built using X11, designed to replicate and extend bash functionality. The application features a GUI with multi-tab support, Unicode input, command history with fuzzy search, pipes, redirections, signal handling, and innovative features like MultiWatch for parallel process monitoring.

**Key Statistics:** - Total Lines of Code: ~5,000+ - Source Files: 30+ (C source and headers) - Features Implemented: 11 core + enhancements - System Calls Used: 40+ distinct calls - Maximum Tabs: 10 independent shells - History Capacity: 10,000 commands

# 1. System Architecture

## 1.1 High-Level Overview

┌──────────────────────────────────────────────────┐  
│ MyTerm Application │  
│ - Event loop coordinator │  
│ - X11 event processing │  
│ - Global state management │  
└────────┬─────────────────────────────────────────┘  
 │  
 ┌────┴────┬──────────┬──────────┐  
 ▼ ▼ ▼ ▼  
┌────────┐ ┌──────┐ ┌────────┐ ┌────────┐  
│ GUI │ │Input │ │ Shell │ │ Utils │  
│ Layer │ │Layer │ │ Layer │ │ Layer │  
└────────┘ └──────┘ └────────┘ └────────┘

## 1.2 Module Dependencies

* **GUI Layer**: X11 window management, rendering, tab system
* **Input Layer**: XIM/XIC for Unicode, line editing, autocomplete
* **Shell Layer**: Command parsing, execution, pipes, redirections, signals, history, processes
* **Utils Layer**: Unicode handling, escape sequences

## 1.3 Data Flow

Keyboard → X11 Event → XIM/XIC → LineEdit → Command Parser  
 │  
 ┌───────────────┴────────────┐  
 ▼ ▼  
 External Command Built-in Command  
 │ │  
 ▼ ▼  
 Command Executor Direct Execution  
 │ │  
 ┌───────────────┼────────────┐ │  
 ▼ ▼ ▼ │  
 Redirects Pipes Normal │  
 │ │ │ │  
 └───────────────┴────────────┴──────────────┘  
 │  
 ▼  
 Process Manager → Text Buffer → GUI

# 2. Feature 1: X11 Graphical User Interface

## 2.1 Architecture

**Components**: X11Context, TabManager, TextBuffer, Renderer

**Implementation**:

// Core structure  
typedef struct {  
 Display \*display;  
 Window window;  
 GC gc;  
 XFontStruct \*font;  
 int width, height;  
} X11Context;

**Key Functions**: - XOpenDisplay(): Connect to X server - XCreateSimpleWindow(): Create 800x600 window - XLoadQueryFont(): Load monospace font - XMapWindow(): Display window

**Tab Management**: Up to 10 independent tabs, each with: - TextBuffer (10,000 lines × 256 chars) - LineEdit state - ProcessManager - Working directory - History search state

**Rendering Pipeline**: 1. Tab bar (30px height) 2. Text buffer (scrollable) 3. Input line with cursor 4. XFlush() to display

**System Calls**: XOpenDisplay, XCreateSimpleWindow, XSelectInput, XMapWindow, XNextEvent, XDrawString, XFillRectangle

# 3. Feature 2: External Command Execution

## 3.2 Process Flow

Command → Parser → Fork → Child Process  
 │  
 ┌──────────┴─────────┐  
 ▼ ▼  
 Setup (setpgid, Execute (execvp)  
 signals, I/O)  
 │  
 ▼  
 Parent waits (waitpid) + reads output (pipe)  
 │  
 ▼  
 Display in GUI

## 3.3 Implementation

**Parsing**:

int parse\_command(char \*cmd\_str, Command \*cmd);  
// Handles: quotes, spaces, arguments  
// Output: args[] array, argc count

**Built-in Commands**: - cd [dir]: Change directory using chdir() - echo [-e] [-n] text: Print with escape sequences - history: Display command history

**Execution with Signal Handling**:

char\* execute\_command\_with\_signals(Command \*cmd,   
 RedirectInfo \*redir\_info,  
 ProcessManager \*pm,   
 const char \*cmd\_str) {  
 pipe(output\_pipe);  
 fcntl(output\_pipe[0], F\_SETFL, O\_NONBLOCK);  
   
 pid = fork();  
 if (pid == 0) { // Child  
 setpgid(0, 0);  
 signal\_handler\_setup\_child();  
 dup2(output\_pipe[1], STDOUT\_FILENO);  
 execvp(cmd->args[0], cmd->args);  
 exit(127);  
 }  
   
 // Parent: wait loop with event processing  
 while (!process\_exited) {  
 g\_event\_processor\_callback(); // Keep GUI responsive  
 read(output\_pipe[0], buffer); // Non-blocking  
 waitpid(pid, &status, WNOHANG | WUNTRACED);  
 usleep(10000);  
 }  
   
 return output;  
}

**System Calls**: fork, pipe, execvp, dup2, setpgid, waitpid, fcntl, read, close

# 4. Feature 3: Multiline Unicode Input

## 4.1 UTF-8 Support

**Character Encoding**: - 1 byte: ASCII (0xxxxxxx) - 2 bytes: 110xxxxx 10xxxxxx - 3 bytes: 1110xxxx 10xxxxxx 10xxxxxx - 4 bytes: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

**XIM/XIC Setup**:

InputState\* input\_state\_init(Display \*display, Window window) {  
 state->xim = XOpenIM(display, NULL, NULL, NULL);  
 state->xic = XCreateIC(state->xim,  
 XNInputStyle, XIMPreeditNothing | XIMStatusNothing,  
 XNClientWindow, window,  
 NULL);  
 return state;  
}

**Key Processing**:

int len = Xutf8LookupString(xic, &event.xkey,   
 buffer, sizeof(buffer)-1,   
 &keysym, &status);  
// Converts KeyPress to UTF-8 string

## 4.2 Line Editing Operations

**Insert**: Handle multi-byte characters, memmove for space **Delete**: Find UTF-8 character start, remove complete character **Move Left/Right**: Skip continuation bytes (10xxxxxx)

**Escape Sequences**: \n, \t, \r, \\ processed by process\_escape\_sequences()

**System Calls**: setlocale, XOpenIM, XCreateIC, Xutf8LookupString

# 5. Feature 4: Input Redirection

## 5.1 Implementation

**Parsing**: Extract < filename from command string

void parse\_redirections(char \*cmd\_str, RedirectInfo \*info);  
// Finds '<', extracts filename, cleans command

**Execution**:

// In child, before execvp:  
int in\_fd = open(filename, O\_RDONLY);  
dup2(in\_fd, STDIN\_FILENO);  
close(in\_fd);  
// Now stdin reads from file

**File Descriptor Evolution**:

Initial: FD 0 → stdin (terminal)  
After open: FD 3 → input.txt  
After dup2(3,0): FD 0 → input.txt ✓  
After close(3): FD 0 → input.txt (only reference)

**System Calls**: open(O\_RDONLY), dup2, close

# 6. Feature 5: Output Redirection

## 6.1 Implementation

**Execution**:

int out\_fd = open(filename, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);  
dup2(out\_fd, STDOUT\_FILENO);  
close(out\_fd);

**Flags**: - O\_WRONLY: Write-only mode - O\_CREAT: Create if doesn’t exist - O\_TRUNC: Truncate to 0 bytes - 0644: rw-r–r– permissions

**Combined Redirection**: sort < input.txt > output.txt - Apply input redirection first (stdin → file) - Apply output redirection second (stdout → file) - Command reads/writes files exclusively

**System Calls**: open(O\_WRONLY|O\_CREAT|O\_TRUNC), dup2, close

# 7. Feature 6: Pipe Support

## 7.1 Pipeline Architecture

cat file.txt | grep error | wc -l  
  
┌──────┐ pipe1 ┌──────┐ pipe2 ┌──────┐ capture  
│ cat │─────────→│ grep │─────────→│ wc │─────────→ Parent  
└──────┘ └──────┘ └──────┘

## 7.2 Execution Algorithm

char\* execute\_pipeline\_with\_signals(Pipeline \*pipeline, ...) {  
 pipe(capture\_pipe);  
 pid\_t pipeline\_pgid = 0;  
 int input\_fd = STDIN\_FILENO;  
   
 for (int i = 0; i < num\_commands; i++) {  
 if (i < num\_commands-1) pipe(pipe\_fds);  
   
 pids[i] = fork();  
 if (pids[i] == 0) { // Child  
 if (i == 0) setpgid(0, 0); // Create group  
 else setpgid(0, pipeline\_pgid); // Join group  
   
 signal\_handler\_setup\_child();  
   
 if (input\_fd != STDIN\_FILENO) {  
 dup2(input\_fd, STDIN\_FILENO);  
 }  
   
 if (i < num\_commands-1) {  
 dup2(pipe\_fds[1], STDOUT\_FILENO);  
 } else {  
 dup2(capture\_pipe[1], STDOUT\_FILENO);  
 }  
   
 execvp(cmd->args[0], cmd->args);  
 exit(127);  
 }  
   
 // Parent: manage process group, close unused fds  
 if (i == 0) pipeline\_pgid = pids[0];  
 setpgid(pids[i], pipeline\_pgid);  
   
 if (input\_fd != STDIN\_FILENO) close(input\_fd);  
 if (i < num\_commands-1) {  
 close(pipe\_fds[1]);  
 input\_fd = pipe\_fds[0];  
 }  
 }  
   
 // Read output, wait for all processes  
 return output;  
}

**Process Group**: All pipeline processes share PGID for unified signal handling

**System Calls**: pipe (N times), fork (N times), dup2 (2N times), setpgid, execvp, waitpid

# 8. Feature 7: MultiWatch Command

## 8.1 Architecture

multiWatch ["date", "uptime", "who"]  
  
Shell  
 ├─ Watcher 1 (date)  
 │ └─ Loop: fork → exec date → write .temp file → sleep(1)  
 ├─ Watcher 2 (uptime)  
 │ └─ Loop: fork → exec uptime → write .temp file → sleep(1)  
 └─ Watcher 3 (who)  
 └─ Loop: fork → exec who → write .temp file → sleep(1)  
  
Shell polls temp files with poll() every 100ms

## 8.2 Implementation

**Watcher Process Loop**:

static void watch\_process\_loop(const char \*command,   
 const char \*temp\_file) {  
 signal(SIGINT, SIG\_IGN); // Ignore Ctrl+C  
 while (1) {  
 pid\_t grandchild = fork();  
 if (grandchild == 0) {  
 int fd = open(temp\_file, O\_WRONLY|O\_CREAT|O\_TRUNC, 0600);  
 dup2(fd, STDOUT\_FILENO);  
 dup2(fd, STDERR\_FILENO);  
 execlp("/bin/sh", "sh", "-c", command, NULL);  
 exit(127);  
 }  
 waitpid(grandchild, NULL, 0);  
 sleep(1);  
 }  
}

**Polling**:

void multiwatch\_poll\_output(MultiWatch \*mw,   
 void (\*output\_callback)(const char \*)) {  
 int ready = poll(mw->poll\_fds, mw->num\_commands, 100);  
   
 for (int i = 0; i < mw->num\_commands; i++) {  
 if (mw->poll\_fds[i].revents & POLLIN) {  
 lseek(mw->commands[i].fd, 0, SEEK\_SET);  
 bytes = read(mw->commands[i].fd, buffer, size);  
   
 time\_t now = time(NULL);  
 sprintf(output, "\"%s\", current\_time: %ld\n%s\n",  
 mw->commands[i].command, now, buffer);  
 output\_callback(output);  
 }  
 }  
}

**Cleanup**: kill(SIGTERM) all watchers, close() fds, unlink() temp files, waitpid() to reap

**System Calls**: fork (2× per command per iteration), execlp, open, dup2, sleep, poll, lseek, read, kill, waitpid, unlink, signal

# 9. Feature 8: Line Navigation (Ctrl+A/Ctrl+E)

## 9.1 Implementation

void process\_keypress(XEvent \*event, ...) {  
 if (event->xkey.state & ControlMask) {  
 switch (keysym) {  
 case XK\_a: line\_edit\_move\_to\_start(le); break;  
 case XK\_e: line\_edit\_move\_to\_end(le); break;  
 }  
 }  
}  
  
int line\_edit\_move\_to\_start(LineEdit \*le) {  
 le->cursor\_pos = 0;  
 return 0;  
}  
  
int line\_edit\_move\_to\_end(LineEdit \*le) {  
 le->cursor\_pos = le->length;  
 return 0;  
}

**Cursor Rendering**: Calculate X position using XTextWidth(), draw block with XFillRectangle()

# 10. Feature 9: Signal Handling (Ctrl+C/Ctrl+Z)

## 10.1 Signal Handler Setup

int signal\_handler\_init(void) {  
 shell\_pgid = getpid();  
 setpgid(0, shell\_pgid); // Create shell's process group  
   
 signal(SIGINT, SIG\_IGN); // Shell ignores Ctrl+C  
 signal(SIGTSTP, SIG\_IGN); // Shell ignores Ctrl+Z  
   
 sigaction(SIGCHLD, &sa\_chld, NULL); // Handle child exits  
   
 return 0;  
}  
  
void signal\_handler\_setup\_child(void) {  
 signal(SIGINT, SIG\_DFL); // Child handles signals  
 signal(SIGTSTP, SIG\_DFL);  
}

## 10.2 Terminal Control

int signal\_handler\_give\_terminal\_to(pid\_t pgid) {  
 tcsetpgrp(shell\_terminal, pgid); // Give control  
 return 0;  
}  
  
int signal\_handler\_take\_terminal\_back(void) {  
 tcsetpgrp(shell\_terminal, shell\_pgid); // Take control  
 tcsetattr(shell\_terminal, TCSADRAIN, &shell\_tmodes);  
 return 0;  
}

## 10.3 Signal Flow

**Ctrl+C (SIGINT)**: 1. Terminal sends SIGINT to foreground process group 2. Shell (different PGID) ignores signal 3. Command process group receives signal 4. Parent detects exit via waitpid() 5. Clean up, take terminal back

**Ctrl+Z (SIGTSTP)**: 1. Terminal sends SIGTSTP to foreground process group 2. Process stops (WIFSTOPPED status) 3. Parent moves to background jobs array 4. Display: [1]+ Stopped command 5. Take terminal back, shell ready for input

**System Calls**: setpgid, tcgetpgrp, tcsetpgrp, tcgetattr, tcsetattr, sigaction, signal, kill, waitpid

# 11. Feature 10: Searchable Shell History

## 11.1 Circular Buffer Implementation

typedef struct {  
 char commands[MAX\_HISTORY\_SIZE][MAX\_COMMAND\_LENGTH]; // 10000 × 512  
 int start\_index; // Ring buffer start  
 int count; // Current size  
 char history\_file[PATH\_MAX]; // ~/.myterm\_history  
} HistoryManager;

**Add Command**:

int history\_manager\_add\_command(HistoryManager \*hm, const char \*command) {  
 sanitize\_command(cmd\_copy); // Trim whitespace  
   
 // Skip duplicates  
 if (strcmp(last\_command, cmd\_copy) == 0) return 0;  
   
 if (hm->count < MAX\_HISTORY\_SIZE) {  
 idx = (start\_index + count) % MAX\_HISTORY\_SIZE;  
 strcpy(commands[idx], cmd\_copy);  
 count++;  
 } else {  
 strcpy(commands[start\_index], cmd\_copy);  
 start\_index = (start\_index + 1) % MAX\_HISTORY\_SIZE; // Wrap  
 }  
   
 return 0;  
}

## 11.2 Fuzzy Search with LCS Algorithm

**Longest Common Substring (LCS)**:

Dynamic Programming approach:  
 For each position i,j:  
 if str1[i] == str2[j]:  
 dp[i][j] = dp[i-1][j-1] + 1  
 else:  
 dp[i][j] = 0  
   
 max(dp[i][j]) = LCS length

**Implementation**:

int calculate\_lcs\_length(const char \*str1, const char \*str2) {  
 int \*\*dp = malloc((len1+1) \* sizeof(int\*));  
 // ... allocate 2D array  
   
 int max\_length = 0;  
 for (int i = 1; i <= len1; i++) {  
 for (int j = 1; j <= len2; j++) {  
 if (str1[i-1] == str2[j-1]) {  
 dp[i][j] = dp[i-1][j-1] + 1;  
 max\_length = max(max\_length, dp[i][j]);  
 }  
 }  
 }  
   
 return max\_length;  
}

**Search Flow**: 1. User presses Ctrl+R 2. Enter search term 3. Try exact match first 4. If no match, fuzzy search using LCS (threshold > 2) 5. Sort by LCS length (descending) 6. Display results

**Persistence**: Save to ~/.myterm\_history on every command, load on startup

**System Calls**: fopen, fgets, fprintf, fclose, rename, chmod

# 12. Feature 11: Filename Auto-completion

## 12.1 Implementation

**Match Finding**:

int autocomplete\_find\_matches(const char \*prefix,   
 AutocompleteResult \*result) {  
 DIR \*dir = opendir(".");  
 struct dirent \*entry;  
   
 while ((entry = readdir(dir)) != NULL) {  
 if (entry->d\_name[0] == '.') continue; // Skip hidden  
   
 if (strncmp(entry->d\_name, prefix, prefix\_len) == 0) {  
 strcpy(result->matches[result->num\_matches], entry->d\_name);  
 result->num\_matches++;  
 }  
 }  
   
 closedir(dir);  
   
 // Calculate longest common prefix  
 result->prefix\_length = autocomplete\_longest\_common\_prefix(...);  
   
 return 0;  
}

**Tab Key Handler**:

int tab\_manager\_handle\_autocomplete(TabManager \*mgr) {  
 // Extract last token from command line  
 autocomplete\_extract\_last\_token(command\_line, &token\_start, &token\_end);  
   
 // Find matches  
 autocomplete\_find\_matches(prefix, &result);  
   
 if (result.num\_matches == 0) {  
 return 0; // Nothing  
 } else if (result.num\_matches == 1) {  
 // Auto-complete immediately  
 autocomplete\_replace\_last\_token(command\_line,   
 result.matches[0],  
 new\_command);  
 line\_edit\_clear(le);  
 line\_edit\_insert\_string(le, new\_command);  
 } else {  
 // Show menu  
 sprintf(menu, "1. %s 2. %s ...", match1, match2);  
 text\_buffer\_append(buffer, menu);  
 tab->in\_autocomplete\_mode = 1;  
 }  
   
 return 0;  
}

**Selection**: User types number, file inserted into command line

**System Calls**: opendir, readdir, closedir

# 13. Additional Features

## 13.1 Scrolling Support

**Implementation**: - Page Up/Down: Scroll full page - Shift+Up/Down: Scroll one line - Ctrl+Home/End: Jump to top/bottom - Mouse wheel: Scroll 3 lines

void text\_buffer\_scroll\_up(TextBuffer \*buf, int lines) {  
 buf->scroll\_offset += lines;  
 if (buf->scroll\_offset > buf->line\_count - 1) {  
 buf->scroll\_offset = buf->line\_count - 1;  
 }  
}

**Rendering**: Calculate visible range based on scroll\_offset, render only visible lines

## 13.2 Copy/Paste

**Copy (Ctrl+C when no foreground process)**:

handle\_copy\_to\_clipboard(ctx, line\_edit\_get\_line(le));  
// Sets clipboard ownership, stores content

**Paste (Ctrl+Shift+V)**:

handle\_paste\_from\_clipboard(ctx);  
// Requests clipboard via XConvertSelection  
// Inserts received text into line edit

# 14. Data Structures Summary

| Structure | Size | Purpose |
| --- | --- | --- |
| TextBuffer | ~2.5MB | 10000 lines × 256 chars per tab |
| HistoryManager | ~5MB | 10000 × 512 chars shared |
| LineEdit | 4KB | Input buffer per tab |
| ProcessManager | ~50KB | Job control per tab |
| Pipeline | Variable | Temporary during pipe execution |

# 15. Error Handling Strategy

**Principle**: Fail gracefully, inform user, clean up resources

**Techniques**: - Check all malloc() returns - Validate file operations - Handle EINTR, EAGAIN, ECHILD - Cleanup on error paths - Display error messages in GUI - Exit child processes on exec failure

**Example**:

if ((fd = open(file, flags)) == -1) {  
 perror("open"); // Display error  
 exit(1); // Exit child  
}  
// Parent sees exit status, displays error

# 16. Testing Strategy

## 16.1 Unit Testing

* Command Parser: quotes, spaces, special characters
* UTF-8 Handling: multi-byte operations
* Circular Buffer: wrap-around, overflow
* LCS Algorithm: various string pairs

## 16.2 Integration Testing

* Pipe chains: 2-stage, 3-stage, error handling
* Combined redirections: < input > output
* Signal handling: Ctrl+C during various operations
* Tab switching: state preservation

## 16.3 System Testing

* Long-running commands with output
* MultiWatch with various update rates
* History with 10,000+ commands
* Memory leaks (valgrind)
* Thread safety (race conditions)

# 17. Performance Considerations

**Optimizations**: - Non-blocking I/O prevents GUI freezing - Event callback during command execution - Circular buffer avoids reallocation - Lazy rendering (only visible lines) - Font/GC reuse - Process group for efficient signaling

**Bottlenecks**: - X11 rendering (mitigated by XFlush batching) - History search (mitigated by early termination) - Large output (8KB buffer limit)

# 18. Future Enhancements

1. **Command substitution**: $(command) or backticks
2. **Background jobs**: command & syntax
3. **Environment variables**: export, $VAR expansion
4. **Aliases**: User-defined command shortcuts
5. **Shell scripting**: Conditionals, loops
6. **Output append**: >> operator
7. **Here documents**: << syntax
8. **Command chaining**: &&, ||, ;
9. **Wildcard expansion**: \*, ?, [...]
10. **Job control**: fg, bg, jobs commands

# Conclusion

MyTerm demonstrates comprehensive understanding of: - Process management (fork, exec, wait, signals) - Inter-process communication (pipes, file descriptors) - System-level I/O (redirection, non-blocking) - GUI programming (X11, event loops) - Data structures (circular buffers, dynamic arrays) - Algorithms (LCS for fuzzy search)

The implementation showcases clean code architecture, proper error handling, and efficient resource management suitable for a production-quality terminal emulator.

**Document Version**: 1.0  
**Last Updated**: October 2025  
**Total Pages**: 14