Design Document

MyTerm – A Custom Terminal with X11 GUI

Developed by: Harsh Jain

Platform: macOS with XQuartz

Course: Computing Lab (CS69201)

October 25, 2025

Contents

1. Overview

MyTerm is a C-based graphical shell that uses X11 for rendering a custom terminal interface. It supports multiple tabs, command history, I/O redirection, background jobs, and multi-process monitoring through threads.

The project was implemented and tested on macOS using XQuartz for X11 display.

2. Core Data Structures

Each terminal tab, command job, and text buffer are stored using the following key structures:

```
typedef struct {
    char *lines[MAX_LINES];
    int line_count;
} TextBuffer;

typedef struct {
    pid_t pid;
    int master_fd;
    int active;
    char cmd[256];
} Job;

typedef struct {
    TextBuffer tb;
    char input[INPUT_MAX];
    char *history[MAX_HISTORY];
    Job jobs[MAX_JOBS];
    int job_count, hist_count;
    int cursor_pos, scroll_offset;
    int search_mode;
    char cwd[PATH_MAX];
} Tab;
```

Listing 1: Key Data Structures

—

3. Feature Design and Implementation

3.1. 1. X11 GUI Initialization

The GUI is rendered using the X11 library. The window, graphics context, and input event loop are initialized as follows:

```
Display *dpy = XOpenDisplay(NULL);
Window win = XCreateSimpleWindow(dpy, RootWindow(dpy, 0),
    40, 40, WIN_W, WIN_H, 1, BlackPixel(dpy,0), WhitePixel(dpy,0)
       );
```

```
XSelectInput(dpy, win, ExposureMask | KeyPressMask |
   ButtonPressMask);
XMapWindow(dpy, win);
```

Listing 2: Creating X11 Window

Each tab is drawn manually using `XDrawString()` and `XFillRectangle()` in the `draw_ui()` function.

—

3.2. 2. Command Execution with Fork and Exec

Commands are parsed and executed in `run_command()`. The shell uses `fork()` + `execvp()` to execute commands.

```
pid_t pid = fork();
if (pid == 0) {
    // In child process
    if (infile) dup2(fd_in, STDIN_FILENO);
    if (outfile) dup2(fd_out, STDOUT_FILENO);
    execvp(argv[0], argv);
    _exit(127);
} else {
    // In parent
    waitpid(pid, &status, 0);
}
```

Listing 3: Core of run$_c ommand()$

—

3.3. 3. Multiline Unicode Input

Unicode input is enabled by setting locale and handling newline escapes:

```
setlocale(LC_CTYPE, "");
if (t->input[t->input_len - 1] == '\\') {
    t->input_len--;
    t->input[t->input_len++] = '\n';
    t->multiline_mode = 1;
}
```

The display loop detects `multiline_mode` and renders each input line separately.

—

3.4. 4. Input / Output Redirection

When parsing tokens, symbols <, >, and >> are detected, and file descriptors are reassigned.

```
if (strcmp(tok, "<") == 0) infile = strtok(NULL, " ");
if (strcmp(tok, ">") == 0) { outfile = strtok(NULL, " ");
   append_mode = 0; }
```

```
if (strcmp(tok, ">>") == 0) { outfile = strtok(NULL, " ");
   append_mode = 1; }

int fd = open(outfile, O_WRONLY | O_CREAT |
             (append_mode ? O_APPEND : O_TRUNC), 0644);
dup2(fd, STDOUT_FILENO);
```

Listing 4: I/O Redirection Example

—

3.5. 5. Pipe Implementation

Supports Unix-style pipes using `pipe()` between multiple child processes.

```
pipe(pipes[i]);
if (fork() == 0) {
    dup2(pipes[i-1][0], STDIN_FILENO);
    dup2(pipes[i][1], STDOUT_FILENO);
    execvp(argv[0], argv);
}
```

Listing 5: Pipe Chain Execution

—

3.6. 6. MultiWatch Command

Runs multiple shell commands concurrently in a dedicated thread.

```
void *multiwatch_thread(void *arg) {
    while (multiwatch_active) {
        for (each cmd) {
            pid_t pid = fork();
            if (pid == 0)
                execlp("sh", "sh", "-c", mw->cmds[i], NULL);
            else {
                read(pipefd[0], buf, sizeof(buf));
                tb_append(&t->tb, buf);
            }
        }
        sleep(2);
    }
}
```

Listing 6: MultiWatch Thread Function

Each cycle displays time-stamped outputs from all running commands.

—

3.7. 7. Signal Handling (Ctrl+C, Ctrl+Z)

Foreground processes are interrupted or suspended using `SIGINT` and `SIGTSTP`.

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

void handle_sigtstp(int sig) {
    if (fg_pid > 0) {
        kill(fg_pid, SIGTSTP);
        add_job(&tabs[active], fg_pid, -1, "Suspended job");
    }
}
```

Listing 7: Signal Handlers

—

3.8. 8. Searchable Command History

Persistent history is saved in ˜/.myterm_history and supports interactive search.

```
if (strcmp(t->history[i], term) == 0)
    return i;  // exact match
if (strstr(t->history[i], sub))
    best_idx = i;  // longest substring
```

Listing 8: History Search (Ctrl+R)

```
FILE *fp = fopen(HISTORY_FILE, "w");
for (int i = 0; i < t->hist_count; i++)
    fprintf(fp, "%s\n", t->history[i]);
fclose(fp);
```

Listing 9: History Persistence

—

3.9. 9. Line Navigation (Ctrl+A / Ctrl+E)

Cursor management inside input buffer:

```
if (c == 1) t->cursor_pos = 0;          // Ctrl+A
if (c == 5) t->cursor_pos = t->input_len; // Ctrl+E
```

Listing 10: Cursor Movement

Cursor is drawn visually in draw_ui() using XDrawLine().

—

3.10. 10. File Auto-Completion (Tab Key)

Implements file completion using directory scanning.

```
DIR *d = opendir(t->cwd);
```

```
while ((de = readdir(d)) != NULL)
    if (strncmp(de->d_name, prefix, len) == 0)
        matches[mcount++] = strdup(de->d_name);
```

Listing 11: Auto-Completion Logic

If multiple matches:

```
tb_append(&t->tb, "Multiple matches:");
for (int i=0;i<mcount;i++)
    tb_append(&t->tb, matches[i]);
```

—

3.11. 11. Background Jobs and Non-blocking I/O

Each job's output is monitored asynchronously using `fcntl(O_NONBLOCK)` and polled periodically.

```
while ((r = read(t->jobs[i].master_fd, buf, sizeof(buf)-1)) > 0)
   {
    buf[r] = '\0';
    tb_append(&t->tb, buf);
}
waitpid(t->jobs[i].pid, &st, WNOHANG);
```

Listing 12: Background Job Read

—

3.12. 12. Multi-Tab Management

Tabs are dynamically created and closed through mouse clicks:

```
int create_tab(Tab *tabs, int *tab_count, int *active) {
    Tab *t = &tabs[*tab_count];
    tb_init(&t->tb);
    getcwd(t->cwd, sizeof(t->cwd));
    snprintf(t->title, sizeof(t->title), "tab %d", *tab_count+1);
    load_history(t);
    (*tab_count)++;
    return *tab_count - 1;
}
```

Listing 13: Tab Creation

—

4. Threading and Non-blocking Architecture

- The GUI event loop runs on the main thread.
- The `multiWatch` feature runs on a detached thread via `pthread_create()`.

- Non-blocking reads prevent the UI from freezing when background jobs write data.

—

5. Cleanup and Exit

When the user exits:

```
for (int i=0;i<tab_count;i++) {
    save_history(&tabs[i]);
    for (int j=0;j<tabs[i].job_count;j++)
        if (tabs[i].jobs[j].active)
            kill(tabs[i].jobs[j].pid, SIGKILL);
    tb_free(&tabs[i].tb);
}
```

Ensures all jobs are terminated and memory freed.
—

6. Conclusion

This design demonstrates practical integration of:

- Process creation and inter-process communication
- Asynchronous job handling
- Signal control and GUI synchronization
- File operations, tabbed UI, and persistent storage

The modular structure of `MyTerm` ensures maintainability and extensibility for future enhancements.