DESIGNDOC — swagnik@myterm

Project: ***MyTerm — A Custom Terminal with X11 GUI***

Name: Swagnik Ghosh

Roll No.: 25CS60R67

Date: October 2025

Table of Contents

1. Data Structures & Global State

Key data structures (as used across functions):

- `tabState` (structure defined in code): contains per-tab fields:
  - `vector<string> displayBuffer` — screen content.
  - `string input` — current input buffer (multi-line allowed).
  - `int currentCursorPosition` — cursor index within `input`.
  - `bool searchFlag`, `recommFlag`, `multLineFlag` — modes for search/autocomplete/multi-line input.
  - `vector<string> recs` — candidate completions.
  - `string showRec` — formatted suggestion string.
  - `string cwd` — per-tab current working directory.
  - cursor blink state fields: `bool dispCursor`, `chrono::steady_clock::time_point lastBlink`.
  - `int inpIdx`, `int scrlOffset`, `bool userScrolled`, `int count`, `title`.
- Global variables:
  - `Display *disp; int scr; Window root;` — X11 handles (set in `create_window`/`run`).
  - `vector<tabState> tabs` — list of tabs.
  - `int tabActive` — active tab index.
  - `vector<string> inputs` — command history (loaded via `loadInputs`).
  - `std::mutex mwQueueMutex; queue<watchMsg> mwQueue` — queue used by multiWatch threads to push messages to GUI thread.
  - `atomic<bool> mwStopReq` — to stop multiWatch operations.
  - `mutex currPidsMutex; vector<pid_t> currChildPids; atomic<bool> cmdUnderExec` — track running child PIDs for interrupts.
  - `volatile sig_atomic_t sigintReqFlag` — async-signal-safe flag used to notify exec loops of Ctrl+C.

Functions will read and/or update these shared structures; when updating cross-thread state they take the appropriate locks described above.

---

2. File-by-file elaboration and full function workflows

Below are **per-file** descriptions enumerating each function present in the code and the detailed runtime/workflow for each — including inputs, outputs, side-effects, system calls, and how they call or are called by other functions.

`main.cpp`

Function: `main`

Signature: `int main(int argc, char** argv)`

Purpose & role: Program entry point. Responsible for:

- Initializing global state (e.g., reading the parent directory parameter if used to trim prompts).
- Loading history via `loadInputs()` (from `helper.cpp`) or similar initialization.
- Creating and showing the X11 window via `create_window()`.
- Entering the main event loop by calling `run()`.

Detailed workflow:

1. Parse `argc`/`argv`. The project passes a `par_dir` or parent path argument (used to trim displayed prompt path).
2. Initialize `len` or equivalent prompt trimming global, if present (this is used by `getPWD` and `formatPWD` in helpers).
3. Call `loadInputs()` (helper) to load saved history into the `inputs` vector. (`loadInputs()` reads `input_log.txt` and fills `inputs`.)
4. Call `create_window()` to get an X11 `Window` handle and set up basic state.
5. Call `run(win)` which enters the main GUI event loop. After `run` returns, perform cleanup and exit.
6. Return exit code (typically 0 on success).

System calls used indirectly: none directly, but `run()` will use X11 calls and process management.

---

run.cpp

This file contains the main GUI and event loop control.

Function: `create_window`

Signature: `Window create_window(int x, int y, int w, int h, int b)` (exact argument list may vary by your code)

Purpose & role:
Creates and configures the main X11 window.

Detailed workflow:

1. Call `XOpenDisplay(NULL)` (if not already opened) — obtains `disp`.
2. Determine default screen `scr` and root window `root`.
3. Use `XCreateSimpleWindow()` (or `XCreateWindow`) to create a window with the provided position and size.
4. Set event masks (ExposureMask, KeyPressMask, ButtonPressMask, StructureNotifyMask, PointerMotionMask, SelectionNotify, etc.) using `XSelectInput()` so the program receives keyboard and mouse events.
5. Create a `GC` (graphics context) with `XCreateGC()` and set default font (font will be set later in `run()`).
6. Optionally call `XStoreName()` (set window title) and `XMapWindow()` to map it visible.
7. Return the `Window` handle to `main()` / `run()`.

System calls / X11 calls: `XOpenDisplay`, `XCreateSimpleWindow`, `XSelectInput`, `XCreateGC`, `XMapWindow`.

Side-effects: sets global `disp`, `scr`, `root` used by drawing functions.

Function: `run`

Signature: `void run(Window win)`

Purpose & role:
Main event loop. Handles:

- Loading font and XIM/XIC for Unicode input.

- Initializing tabs (`add_tab()` first tab).
- Receiving and processing X events: `Expose`, `ConfigureNotify`, `KeyPress`, `ButtonPress`, `MotionNotify`, `SelectionNotify`.
- Calling drawing routines (`draw_navbar`, `draw_tabs`, `drawScreen`).
- Handling keyboard shortcuts: Enter to run commands, Tab for autocomplete, Ctrl+R for history search, Ctrl+C to interrupt, Ctrl+A/E to move cursor, Up/Down for browsing history, etc.
- Draining and rendering `multiWatch` messages (reads from `mwQueue`) and updating `displayBuffer`.

Detailed workflow and call flow:

Initialization:

1. Load or set a font via `XLoadQueryFont()`. Fallback to a fixed font if needed.
2. Create GC for drawing and set font with `XSetFont`.
3. Initialize XIM/XIC input method (if available) for wide-character input.
4. Call `add_tab("/")` to create the first tab and initialize its prompt.
5. Map the window (`XMapWindow`) if not already mapped.

Main event loop (infinite until exit):

1. `while (true)`: Use `XPending(disp)` and `XNextEvent(disp, &event)` to fetch events.
2. On `Expose` and `ConfigureNotify`:
    - Recalculate window metrics.
    - Call `draw_navbar(win, gc, width)` to render tabs header.
    - Call `draw_tabs(...)` or `draw_navbar` depending on your implementation to draw tab elements (labels, close buttons).
    - Call `drawScreen(win, gc, font, tabs[tabActive])` to draw the current tab content.
3. On `ButtonPress` (mouse click):

- Determine if click is in navbar area by computing positions returned by `draw_tabs` or via `navbar_hit_test`.
- If user clicked "+" → call `add_tab("/")` to create a new tab.
- If user clicked a close `×` on a tab → remove that tab from `tabs` vector, update `tabActive`.
- If user clicked a tab body → set `tabActive` to clicked tab index and redraw.
- If click in content area and mouse wheel (Button4/Button5) → change `tabs[tabActive].scrlOffset` and call `drawScreen`.

4. On `MotionNotify` (pointer move):
   - Update hover state for close buttons and "+" sign (visual hover highlight).
   - Redraw navbar to reflect hover changes.
5. On `KeyPress`:
   - Use XIC (`XwcLookupString`) to decode the multi-byte/unicode character(s) if X Input Methods are available; otherwise decode using keycode.
   - Update `tabs[tabActive].input`, `currentCursorPosition`, and flags (`multLineFlag` toggled on `"` characters).
   - Implement control sequences:
     - **Enter / Return**:
       - If `recommFlag` true: parse numeric selection via `getRecIdx()` (helper) and complete with the choice from `recs`.
       - Else if `searchFlag` true: call `searchHistory()` (helper) with user's search term, then set `input` to the result or show `No match...`.
       - Else if `multLineFlag` true: insert newline into `input` at `currentCursorPosition`.
       - Else: final command execution:
         - `storeInput(input)` to persist history (helper).
         - Call `execCommandInDir(input, tabs[tabActive].cwd)` to execute in the tab's cwd (see `execute.cpp`).
         - Append output lines returned by `execCommandInDir` to `tabs[tabActive].displayBuffer`.

        - Update prompt line via `formatPWD()` or `getPWD()` from helper functions.
    - **Tab key**:
        - Build query via `getQuery(input)` (helper).
        - Call `execCommandInDir("ls", tabs[tabActive].cwd)` to list dir contents (returns vector).
        - Use `getRecomm(query, list)` to get candidate matches (helper).
        - If single match → append completion to `input`.
        - If multiple → format `showRec` and set `recommFlag = true` to allow user to choose (the UI shows numbering).
    - **Ctrl+R**:
        - Set `searchFlag = true`, push a prompt ("Enter search term:") into `displayBuffer`.
    - **Ctrl+C**:
        - Call `notify_sigint_from_ui()` to request pending interrupts (execute functions will act upon this).
        - Add `^C` to `displayBuffer`, reset input prompt.
    - **Ctrl+A/Ctrl+E**:
        - Move `currentCursorPosition` to start/end of the line.
    - **Up/Down**:
        - Navigate history using `inputs` vector (loaded by `loadInputs()`).

6. **SelectionNotify (clipboard paste):**
    - On paste, `XGetWindowProperty` is used to retrieve `PASTE_BUFFER` selection contents and then inserted into `tabs[tabActive].input`. Update `displayBuffer` accordingly.
7. **Periodic tasks inside loop:**
    - Blink cursor logic: toggle `tabs[tabActive].dispCursor` every ~500ms and call `drawScreen` to update cursor visibility.

- Drain `mwQueue`: Acquire `mwQueueMutex`, while queue not empty pop `watchMsg` items and append to appropriate tab `displayBuffer`. If message is for active tab call `drawScreen()`.

8. **Exit conditions:**
    - If user pressed Escape and only one tab exists → return from `run()` to `main()` to exit the application.
    - Clean up XIM/XIC contexts and destroy window.

System calls & libraries used: Xlib (`XNextEvent`, `XMapWindow`, `XCreateGC`, `XLoadQueryFont`, `XDrawString`, `XFillRectangle`, `XTextWidth`, `XOpenIM`, `XCreateIC`), pthreads (`thread` used to spawn multiWatch), and various helpers call `execCommand*` which uses `fork()`/`pipe()`.

Function: `thread`

Signature: `void thread()` (name suggests a helper that may spawn threads — exact signature may vary)

Purpose & role:
This function is present in `run.cpp` (based on extraction); it is used to create detached worker threads where `run()` requires them (e.g., launching `multiWatchThreaded_using_pipes()` as a detached thread). It may wrap `std::thread` launches to ensure consistent API.

Workflow:

- Called by `run()` when a background operation is required (e.g., multiWatch).
- Constructs `std::thread([...])` with captured parameters and calls `.detach()` so the worker runs independently.
- Returns immediately to `run()`.

System calls: None directly — uses C++ thread library.

Side-effects: Spawns asynchronous operations that write to `mwQueue`.

---

drawscreen.cpp

This file contains all drawing logic used by the GUI.

Function: `drawScreen`

Signature: `int drawScreen(Window win, GC gc, XFontStruct *font, tabState &T)` (exact signature may vary)

Purpose & role:
Render the terminal area for the supplied `tabState T`. Handles text wrapping, prompt drawing, color segments, and cursor rendering. Returns the total number of wrapped lines (useful for scroll computations).

Detailed workflow:

1. Query window attributes with `XGetWindowAttributes` to get current width/height.
2. Compute font metrics via `font->ascent` and `font->descent` to get `lineH` (line height).
3. Determine margins and the number of visible rows (seeRows) = floor((height - marginTop) / lineH).
4. Convert `T.displayBuffer` lines into `dispLines` while performing **manual wrapping**:
    - For each `orgLine` in `displayBuffer`:
        - If `orgLine` is empty, add blank `dispLine`.
        - Else iterate characters measuring width via `XTextWidth` and cut string pieces so that each piece fits window width minus margins.
        - If the line begins with the prompt prefix (e.g., `"shre@Term:"`) mark the prompt portion separately to draw it in green.
5. Clip `T.scrlOffset` to valid range (0..alllines - seeRows).
6. For lines in range `[start, end)`:
    - If line begins with `"ERROR:"` draw the rest of string in red.
    - If line begins with `"REC:"` draw it in yellow.
    - If `dl.promptChars > 0` draw prompt prefix in green then remainder in default color.
    - Use `XDrawString()` for each piece.
7. Cursor positioning:

- Compute prompt string for current CWD using `formatPWD()`/`getPWD()` (helper).
- Split `T.input` by `\n` into `lines`.
- Determine `curLine` and `curCol` (which logical line contains the cursor).
- Compute `pxWidth` using `XTextWidth()` for text up to cursor, adding prompt width if cursor is on first line.
- Compute screen coordinates for cursor: `cursorX = marginLeft + pxWidth`; `baselineY = contentYOffset + row*lineH`.
- If cursor line is inside the visible scroll window (cursorLineIdx between `start` and `end`), draw vertical cursor line with `XDrawLine()` using white color.

8. Return total number of lines in `dispLines` (for scrolling calculations).

System calls / X11 calls: `XGetWindowAttributes`, `XTextWidth`, `XDrawString`, `XFillRectangle`, `XDrawLine`, color allocation `XAllocNamedColor`.

Side-effects: None to program logic, but updates UI.

Function: `draw_navbar`

Purpose & role:
Draws the navbar background and a separator line below it. Called whenever the top UI needs repaint or on resize.

Workflow:

1. Fill rectangle at top via `XFillRectangle()` with navbar background color.
2. Draw a bottom border/separator via `XDrawLine()`.
3. Return (no value).

System calls / X11 calls: `XFillRectangle`, `XDrawLine`.

Function: `draw_tabs`

Purpose & role:
Renders each tab (rounded rectangle background, label text, close (`×`) circle) and the "+" add-tab button. Computes and returns positions for tab hit testing.

Workflow:

1. Calculate available width and per-tab width considering the plus button area.
2. For each `tabs[i]`:
    - Compute x position and width `tabW`.
    - Draw filled arc or rounded rectangles to simulate tab shape (`XFillArc`, `XFillRectangle`).
    - Draw active tab indicator by filling a small rectangle at bottom of tab.
    - Draw the label text centered inside the tab using `XDrawString()`.
    - Draw close circular button via `XFillArc` and put an 'x' text in center.
    - Record `tabPosNavbar` for this tab (x, w, xClose, wClose, isPlus flag).
3. Draw the + button at right side similarly.
4. Return the vector of `tabPosNavbar`.

System calls / X11 calls: `XFillArc`, `XFillRectangle`, `XDrawString`, `XDrawRectangle`.

Side-effects: None beyond drawing; returns positions for hit-test.

Function: `navbar_hit_test`

Purpose & role:
Given mouse coordinates relative to the window, determine which navbar element was clicked: a tab body, a close button, or the plus button.

Workflow:

1. Iterate over vector of `tabPosNavbar` (positions computed by `draw_tabs`).
2. For each element:
    - If `isPlus` and mouse over plus area → return special code `-2`.
    - If mouse over `xClose` area → return `-3` and set outIdx to tab index.
    - If mouse over tab body → return tab index and set outIdx.
3. If nothing matches → return `-1`.

Side-effects: none; used by `run()` on `ButtonPress`.

Function: `add_tab`

Purpose & role:
Create a new `tabState` object, initialize its prompt line and CWD, push into `tabs`, and set `tabActive` to new tab index.

Workflow:

1. Construct `tabState t`.
2. Set `t.cwd = initial_cwd` (usually `"/"` or given path).
3. Create prompt using `formatPWD(t.cwd)` / `getPWD()` and push it into `t.displayBuffer`.
4. Set other defaults: `t.inpIdx = inputs.size() - 1`, `t.title = "Tab N"`.
5. `tabs.push_back(std::move(t))`; set `tabActive = tabs.size()-1`.
6. Return.

Side-effects: Global `tabs` and `tabActive` mutated.

---

execute.cpp

This file implements command execution, piping, capturing output, multiWatch, and signals logic.

Function: `getCurrentTime`

Signature: `string getCurrentTime()`

Purpose:
Return a formatted timestamp string like `YYYY-MM-DD HH:MM:SS` used by multiWatch to show a human-readable time in output headers.

Workflow:

1. Call `time()` to obtain current epoch.
2. Use `localtime_r` to convert to `struct tm`.
3. Use `strftime()` to format into a char buffer and return as `string`.

System calls: `time`, `localtime_r`, `strftime`.

Function: `notify_sigint_from_ui`

Signature: `void notify_sigint_from_ui()`

Purpose & role:
Called by the GUI handler when user presses Ctrl+C. It's designed to be async-safe (or called from safe contexts) and sets flags interpreted by execution loops.

Workflow:

1. Set `sigintReqFlag` to 1 (volatile `sig_atomic_t`) to indicate a SIGINT request.
2. Set `mwStopReq.store(true)` to instruct multiWatch threads to stop.
3. Returns immediately.

Side-effects: Signals (via shared flags) to other execution loops and threads.

Function: `pids`

Signature: As found in code (likely a helper that manages child PIDs)

Purpose & role:
Helps record and manage child processes spawned by command execution. Usually called inside `execCommand` and `execCommandInDir`. Maintains `currChildPids` vector and `cmdUnderExec` flag.

Detailed workflow (typical):

1. On start of exec pipeline, `pids` logic sets `currChildPids = pids` under `currPidsMutex` lock and `cmdUnderExec.store(true)`.
2. Execution loops call `handle_pending_sigint()` to check for interrupts.
3. On process termination, `pids` helper clears `currChildPids` and sets `cmdUnderExec.store(false)` under lock.

System calls: none directly but coordinates calls to `kill()` in `handle_pending_sigint()`.

Function: `handle_pending_sigint`

Signature: `void handle_pending_sigint()`

Purpose & role:
Checks the async flag `sigintReqFlag`. If set, clears it and sends `SIGINT` to every PID recorded in `currChildPids`. This allows UI-triggered interrupts (Ctrl+C) to kill the running child processes without killing the GUI.

Detailed workflow:

1. If `sigintReqFlag == 0` return. Otherwise continue.
2. Set `sigintReqFlag = 0`.
3. Acquire `currPidsMutex`.
4. For each pid in `currChildPids` call `kill(pid, SIGINT)`.
5. Release mutex and return.

System calls used: `kill()`.

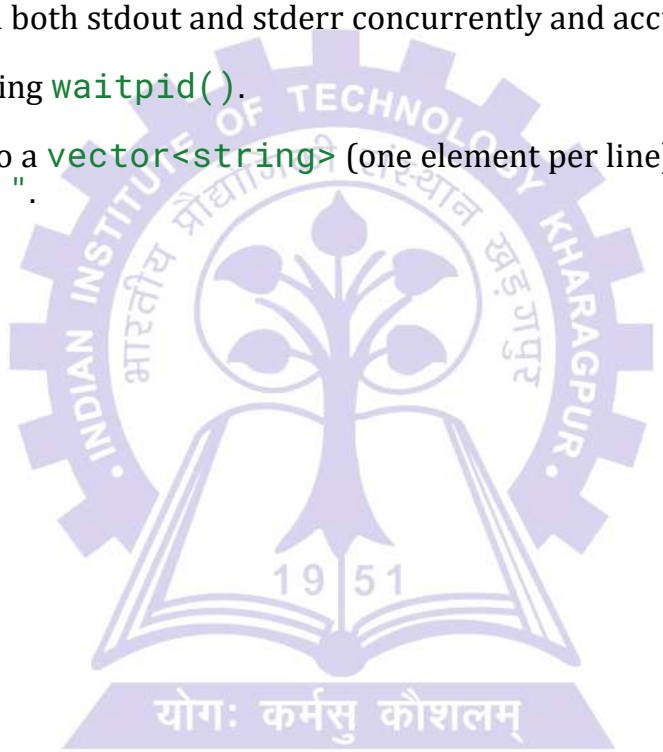
Side-effects: Sends `SIGINT` to child processes. The exec/poll loops should observe killed processes when `waitpid()` or `poll()` returns.

Function: `execCommand`

Signature: `vector<string> execCommand(const string &cmd)`

Purpose:
Execute one pipeline command in the parent process's environment (no per-tab CWD change). This function:

- Parses `cmd` into pipeline components split by `'|'`.
- Spawns child processes for each component using `fork()`.
- Connects pipeline using `pipe()` calls.
- Captures stdout and stderr of the last process via `capture_out` and `capture_err` pipes.
- Uses `poll()` to read both stdout and stderr concurrently and accumulates outputs.
- Waits for children using `waitpid()`.
- Packages outputs into a `vector<string>` (one element per line). If any error, prefixes lines with `"ERROR: "`.



Command Execution Workflow

1. Handle Empty Command

- If cmd is empty →
  Return: [""]

2. Preprocessing

- Trim leading and trailing whitespace from cmd.

3. Handle Built-in cd Command

- If cmd starts with "cd":
  - Attempt to perform chdir() to the specified directory.
  - If success: return [""]
  - If failure: return ["ERROR: cd: no such file or directory"]

4. Parse Pipeline

- Split cmd into parts using the pipe ('|') delimiter → getPipeParts
- Trim whitespace for each part.
- Compute:
- numPipes = getPipeParts.size() - 1

5. Create Pipe Chain

- Initialize:
- vector<int> chainFds(2 * numPipes, -1);
- For each pipe (from 0 to numPipes-1):
  - Call pipe().
  - On failure:
    - Close any already-created FDs.
    - Return an error (e.g., "ERROR: pipe creation failed").

6. Create Capture Pipes

- Create two additional pipes:
- capture_out[2] // for stdout

- capture_err[2] // for stderr

These capture the final command's output and error streams.

7. Fork for Each Command

For each command i in getPipeParts:

Child Process (pid == 0)

1. Redirect stdin/stdout/stderr:
    - stdin:
      If i > 0,
    - dup2(chainFds[(i-1)*2], STDIN_FILENO);
    - stdout:
        - If i < numPipes →
        - dup2(chainFds[i*2 + 1], STDOUT_FILENO);
        - Else (last command) →
        - dup2(capture_out[1], STDOUT_FILENO);
    - stderr:
    - dup2(capture_err[1], STDERR_FILENO);
2. Close unused FDs:
    - Close all chainFds and both capture pipe ends not in use.
3. Execute command:
4. execlp("bash", "bash", "-c", getPipeParts[i].c_str(), NULL);
    - On failure:
      Call perror() and _exit(127).

Parent Process

- Record child PID in a vector<pid_t> pids**.**

8. Post-Fork Cleanup in Parent

- Close all write ends of chainFds and capture_out[1], capture_err[1].
- Acquire currPidsMutex:
    - currChildPids = pids

- cmdUnderExec = true

9. Read from Output & Error Pipes

- Use poll() to monitor:
  - capture_out[0] (stdout)
  - capture_err[0] (stderr)

Loop:

While active streams > 0:

1. Call handle_pending_sigint() to process interrupts quickly.
2. poll() on both fds.
3. When data available:
   - read() into buffer.
   - Append to opBuffer or errBuffer.
4. On EOF or POLLHUP, close FD and decrement active count.

10. Final Cleanup

- Close all remaining file descriptors (chainFds, capture_out[0], capture_err[0]).

11. Wait for Children

- For each pid in pids:
- waitpid(pid, &status, 0);
- If any child exited with non-zero status → set hadError = true.

12. Update Execution State

- Under currPidsMutex:
  - Clear currChildPids
  - Set cmdUnderExec = false

13. Process Output Buffers

- Split opBuffer and errBuffer into lines (\n-delimited):
- vector<string> outLines;

- vector<string> errLines;

14. Handle Errors

- If hadError == true:
    - If errLines not empty →
      return {"ERROR: " + errLine ...}
    - Else if outLines not empty →
      return {"ERROR: " + outLine ...}
    - Else
      return {"ERROR: (process exited with code N)"}
    - **15. Return Normal Output**
- If success
    - If outLines not empty → return outLines
    - Else → return [""]

16. Exit

- End of workflow.**System calls used:** `fork`, `pipe`, `dup2`, `close`, `execlp`, `poll`, `read`, `waitpid`, `kill` (via `handle_pending_sigint`), `perror`, `write`/`read` indirectly.

Notes & side-effects: Updates `currChildPids` to allow GUI interrupt to kill children. Uses `bash -c` to allow redirection (`<`, `>`) and expansions to be handled by shell — thus the program itself does not parse `<` / `>` but relies on `bash` to perform redirection (this is valid and simpler).

Function: `execCommandInDir`

Signature: `vector<string> execCommandInDir(const string &cmd, string &cwd_for_tab)`

Purpose:
Same semantics as `execCommand` but:

- Implements tab-local `cd` built-in (affects `cwd_for_tab` rather than actual process CWD).
- Ensures each child `chdir(cwd_for_tab.c_str())` before `execlp(...)` so commands execute with per-tab working directory.

Detailed workflow:

1. Trim `cmd`.

2. Implement built-in `cd`:
    - If `cmd` starts with `cd` :
        - Compute `target` path using `cwd_for_tab` if relative, or accept absolute path if begins with `/`.
        - Resolve path using `realpath()` into `resolved`.
        - If `stat(resolved, &st) == 0 && S_ISDIR(st.st_mode)` then set `cwd_for_tab = resolved` and return `[""]`.
        - Else return `{"ERROR: cd: no such file or directory: " + path}`.
    - If `cmd == "cd"` or `cd ~` set `cwd_for_tab` to `$HOME`.
3. Split `cmd` into pipeline parts and create pipes and capture pipes similar to `execCommand`.
4. For each child process:
    - Immediately after fork in the child, call `chdir(cwd_for_tab.c_str())` so child runs in the tab-specific directory.
    - Set up duplication of FDs with `dup2()` for pipes and capture pipes.
    - `execlp("bash", "bash", "-c", part)` as before.
5. Parent reads capture pipes using `poll()` into `opBuffer`/`errBuffer`.
6. `handle_pending_sigint()` is periodically invoked in the read loop.
7. Wait for child processes, cleanup `currChildPids`.
8. Return output or error formatted as `vector<string>` like `execCommand`.

System calls used (same as above) plus `chdir`, `stat`, `realpath`.

Side-effects: Mutates `cwd_for_tab` when built-in `cd` is invoked; this results in prompt change and arranges correct working dir for subsequent commands in that tab.

Function: `multiWatchThreaded_using_pipes`

Signature: `void multiWatchThreaded_using_pipes(const vector<string> &cmds, int tabIdx)`

Purpose & role:
Implement `multiWatch ["cmd1","cmd2", ...]`. Runs provided commands in parallel, repeatedly gathers their outputs, and posts timestamped messages to the GUI via the `mwQueue`.

Detailed design & workflow:

Overall semantics: For each command in `cmds`, launch a watcher that:

- Forks one process to run the command (via `bash -c`), redirecting both stdout and stderr into a pipe.
- Parent thread reads non-blockingly from that pipe and collects output lines.
- Parent posts headers and body messages into `mwQueue` with `watchMsg` entries that include `tabIdx` to tell GUI which tab to append messages to.

Step-by-step:

1. Validate `cmds`; if empty push `"multiWatch: no commands provided"` into `mwQueue` and return.
2. `mwStopReq.store(false)` to ensure we start in running state.
3. Push initial `"multiWatch: started :: press Ctrl+C to stop."` entry into `mwQueue`.
4. Enter a while loop `while (!mwStopReq.load())` so multiWatch keeps re-running at interval (code uses `sleep_for(2s)` between cycles):
    - For each `cmd` in `cmds`, spawn a watcher thread (std::thread) that:
        1. Create a pipe `pipefd[2]`.
        2. `pid = fork()`.
            - If `pid == 0` (child):
                - Close `pipefd[0]`.
                - `dup2(pipefd[1], STDOUT_FILENO); dup2(pipefd[1], STDERR_FILENO)` so stdout & stderr go to the same pipe.
                - `chdir(tab_cwd)` if provided (tabIdx valid and `tabs[tabIdx].cwd` available).
                - `execlp("bash","-c", cmd.c_str(), NULL)` to run the command.

        - On exec fail `_exit(127)`.
    - Else if `pid > 0` (parent):
        - Close `pipefd[1]`; set `pipefd[0]` to non-blocking using `fcntl`.
        - Add `pid` to `currChildPids` under `currPidsMutex` and set `cmdUnderExec = true`.
        - Read data from the pipe in a loop with `poll()` (with small timeout e.g., 200ms) collecting into `opBuffer`. If `mwStopReq` becomes true, caller will break out and kill children.
        - After EOF or termination, `waitpid(pid, NULL, 0)`.
        - Remove `pid` from `currChildPids`, and if empty set `cmdUnderExec=false`.
        - Build messages:
            - Header: `"\"<cmd>\" , <getCurrentTime()> :"`.
            - Separator line of `-`.
            - Each output line from `opBuffer` as separate messages.
            - Another separator.
        - Acquire `mwQueueMutex` and push header, separators, and output lines as `watchMsg` objects (with `.tabIdx = tabIdx`).
    3. Thread joins (or detached depending on code). In the code you provided the watchers `join()` at the end of the command cycle.
- Wait for all watcher threads (join); then sleep for 2 seconds; loop again unless `mwStopReq` set.

5. When `mwStopReq` set (UI pressed Ctrl+C or other), watchers will exit loops, child processes will be signaled via `handle_pending_sigint()`/`kill()`, watchers will push any remaining output, and the worker thread will return.
6. Optionally, push `"__MULTIWATCH_DONE__"` sentinel to `mwQueue` so GUI can append final prompt.

System calls used: `fork`, `pipe`, `dup2`, `fcntl` (O_NONBLOCK), `poll`, `read`, `waitpid`, `kill` (via `handle_pending_sigint`), thread functions.

Notes:

- Implementation uses pipes rather than temporary files (efficient, immediate).
- The multiWatch thread communicates via `mwQueue` which `run()` drains and displays.

---

helper.cpp

This file provides many utility functions for prompt handling, history, and autocomplete.

Function: `getPWD`

Signature: `string getPWD()`

Purpose: Return the current working directory as a string, trimmed according to configured prefix length (global `len` in code). Used to create the prompt display.

Workflow:

1. Call `getcwd(cwd, sizeof(cwd))`.
2. Convert to `string currentDir`.
3. If `currentDir.size() < len` return `currentDir`.
4. Else return `currentDir.substr(len)` — i.e., path relative to `par_dir`.

System calls: `getcwd`.

Function: `formatPWD`

Purpose & role: Similar to `getPWD` — returns a prompt-formatted cwd string (may add `~` or trailing `/$` formatting). Used by `drawScreen` to compute prompt width and by `add_tab` to initialize the prompt.

Workflow: minimal string manipulation based on `getPWD()`.

Function: `stripQuotes`

Signature: `string stripQuotes(const string &s)`

Purpose: Remove surrounding or internal quotes (") in input strings, used when storing or parsing multiline strings.

Workflow:

- Iterate over characters and remove `"` (or handle escaped quotes) depending on code logic.

Use: Called before sending commands to execution functions or storing history, to normalize strings.

Function: `commonPrefixLength`

Signature: `int commonPrefixLength(const string &a, const string &b)`

Purpose: Return the length of the common prefix between two strings. Used to compute the best matching prefix when multiple autocomplete candidates exist and also used in history "closest match" computation.

Workflow:

- `len = min(a.size(), b.size())`.
- Iterate `i` from 0 to `len-1` and increment until characters differ.
- Return `i`.

Function: `searchHistory`

Signature: `string searchHistory(const string &input, const string &historyPath = "./input_log.txt")`

Purpose: Search `input_log.txt` for a most recent exact match; if no exact match, return the best prefix match where prefix length >= 2; otherwise return `"No match for search term in history"`.

Workflow:

1. Open `historyPath` for reading via `ifstream`.
2. If not open, return error message.
3. Read lines into vector `history`.
    - For each line, skip leading numbers and spaces, push only command part into `history`.
4. Reverse iterate `history`:
    - If a command equals `input` return it immediately (most recent exact match).
    - Else compute prefixLen = `commonPrefixLength(cmd, input)` and maintain `maxLenPrefix`, storing candidates with the maximum.
5. If full match found return it; else if `maxLenPrefix >= 2` return `allCandidates[0]` (most recent among those); else return `"No match..."`.

Side-effects: none.

Function: `getLastHistoryNumber`

Signature: `int getLastHistoryNumber()`

Purpose: Read `input_log.txt` and parse numeric index at start of each line to find the largest index (so that `storeInput` can append with `last + 1`).

Workflow:

1. Open `input_log.txt`.
2. Iterate each line, `istringstream iss(line)`, attempt to `iss >> num`.
3. Track `lastIdx = max(lastIdx, num)`.
4. Return `lastIdx`.

Side-effects: none.

Function: `storeInput`

Signature: `void storeInput(const string &input, const string &historyPath = "./input_log.txt")`

Purpose: Append `input` string to `input_log.txt` with an incremented index.

Workflow:

1. Determine `histNum = getLastHistoryNumber() + 1`.
2. Open `historyPath` in append mode using `ofstream`.
3. Write formatted line `" " << histNum << " " << input << endl`.
4. Close file.

System calls: `open`/`write` indirectly via fstreams.

Side-effects: modifies `input_log.txt`.

Function: `loadInputs`

Signature: `vector<string> loadInputs(const string &historyPath = "./input_log.txt")`

Purpose: Read entire history file and return the list of commands (without numeric prefixes). Called once at startup to populate `inputs` used during runtime.

Workflow:

1. `ifstream in(historyPath)`.
2. `while (getline(in, line))` parse and strip leading numbers/whitespace before pushing into `vector<string> inputs`.
3. Return `inputs`.

Side-effects: none (returns vector).

Function: `getQuery`

Signature: `string getQuery(const string &input)`

Purpose: Extract the last token (after the last space) from `input` for use during Tab completion. It handles possible `./` cases and returns the last fragment.

Workflow:

- Iterate characters of `input`. When a space is encountered reset `query` to empty, otherwise append.
- At end return `query`. (This is a simple heuristic: returns last space-separated token).

Function: `getRecIdx`

Signature: `int getRecIdx(const string &inp)`

Purpose: Parse integer digits from a string `inp` (e.g., user typed "1" to choose suggestion) and return selection index (default 1 if none).

Workflow:

- Build `recIdx` string by iterating digits; if empty return 1; else convert via `stoi()` and return.

Function: `getRecomm`

Signature: `vector<string> getRecomm(const string &query, const vector<string> &list)`

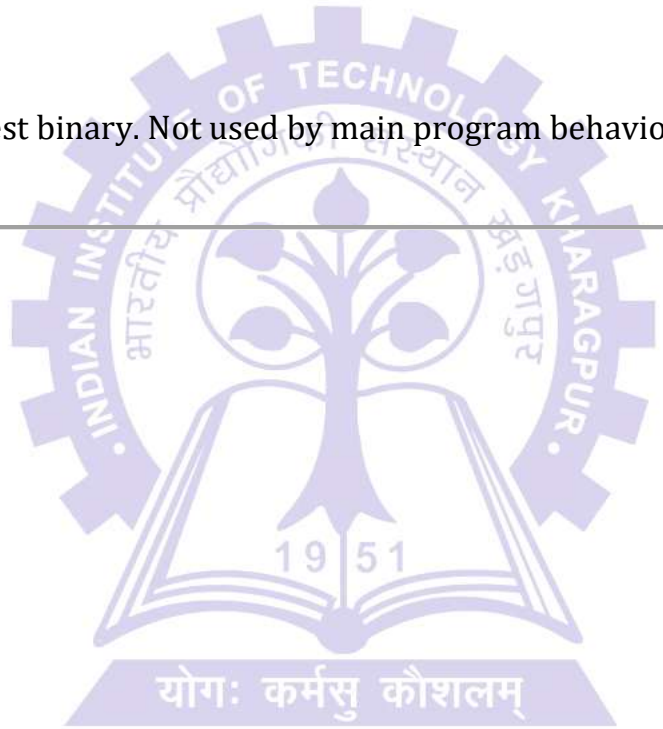
Purpose: Return the subset of `list` entries that start with `query`. Used for Tab auto-completion.

Workflow:

- Iterate `list`, if `ele.rfind(query,0) == 0` then `push_back(ele)`.
- Return `recs`.

`a.cpp`

Function: `main` — small test binary. Not used by main program behavior; often included as test or stub.



3.MyTerm (swagnik@myterm) — Detailed Feature Implementation Summary

---

1. Graphical User Interface (GUI with X11)

Goal: Build a custom terminal GUI using Xlib that mimics bash behavior, handles input/output via an X11 window, and supports multiple tabs.

Implementation:

- Files: main.cpp, run.cpp, drawscreen.cpp
- Core Functions:
    - create_window() — initializes display using XOpenDisplay(), creates the main window using XCreateSimpleWindow(), and maps it with XMapWindow().
    - run() — main event loop capturing KeyPress, ButtonPress, and Expose events via XNextEvent().
    - drawScreen() — renders terminal content, prompt, and cursor using XDrawString() and XTextWidth().
    - draw_navbar(), draw_tabs() — render tab headers and “+”/“×” buttons.
    - add_tab() — creates new tabState objects, each storing input/output buffers and its own cwd.

How it works:
Each tab corresponds to an independent shell instance, stored in a vector<tabState>. When the user types in the active tab, the key events append characters to that tab’s buffer. When Enter is pressed, commands are executed, and output lines are drawn in the X11 window via drawScreen().

---

2. Run External Commands

Goal: Execute external binaries or shell commands (ls, gcc, ./a.out, etc.) through child processes.

Implementation:

- Files: execute.cpp
- Functions: execCommand(), execCommandInDir()
- System Calls: fork(), pipe(), dup2(), execlp(), waitpid()

How it works:

1. The command string is parsed to detect pipelines.
2. pipe() creates communication channels.
3. For each pipeline part, a child process is forked:
    - The child redirects STDIN/STDOUT using dup2().
    - Executes via execlp("bash", "bash", "-c", cmd).
4. The parent polls the pipe using poll() and reads the child's output.
5. Output is appended to the current tab's display buffer and redrawn in the GUI.

---

3. Multiline Unicode Input

Goal: Support multi-line and multi-language input/output.

Implementation:

- Files: run.cpp
- Key Functions: run() event loop; uses XwcLookupString() for wide-character input.
- Libraries: <locale.h>, <X11/Xlib.h>

How it works:

- Locale set via setlocale(LC_ALL, "en_US.UTF-8").
- KeyPress events are converted to Unicode using X Input Methods (XIM/XIC).
- If the user presses Enter while inside a quoted string, multLineFlag keeps input in the same buffer and inserts \n.
- The display buffer supports wide characters, rendered directly via Xlib's text functions.

---

4. Input Redirection (“<”)

Goal: Allow commands like ./a.out < infile.txt to read input from files.

Implementation:

- Files: execute.cpp
- Handled by: Shell via bash -c inside execlp()

How it works:
The shell (bash) interprets < infile.txt automatically inside the forked child process. Since the program runs commands via bash -c, there’s no need for manual dup2(open()).

---

5. Output Redirection (“>”, “< >”)

Goal: Redirect output to files (ls > out.txt, ./prog < in.txt > out.txt).

Implementation:

- Files: execute.cpp
- Handled by: bash -c within child processes.

How it works:
When execlp("bash", "bash", "-c", cmd) executes, bash handles file opening and dup2() internally. The MyTerm code does not need to manually duplicate descriptors.

---

6. Pipe (“|”) Support

Goal: Chain multiple commands so that one command’s output becomes another’s input.

Implementation:

- Files: execute.cpp
- Functions: execCommand(), execCommandInDir()
- System Calls: pipe(), dup2(), fork(), close()

How it works:

- The input command is split by '|'.
- For each stage, pipe() creates two FDs: read/write.
- The first process writes to the pipe, the next reads from it.
- STDIN/STDOUT redirection via dup2().
- Each process executes through bash -c.
- The parent process polls all outputs and merges results for display.

---

7. multiWatch Command

Goal: Execute multiple commands concurrently and display their live output with timestamps.

Implementation:

- Files: execute.cpp
- Function: multiWatchThreaded_using_pipes(), getCurrentTime()
- System Calls: fork(), pipe(), poll(), dup2(), kill()
- Concurrency: std::thread, std::mutex, std::atomic

How it works:

- For each command, fork() creates a child process, redirecting both stdout/stderr to a pipe.
- Parent process polls multiple pipes concurrently using poll().
- Each line read is timestamped with getCurrentTime() and tagged with the originating command name.
- Output is queued into mwQueue (shared queue) and displayed in real time by the GUI loop.
- When Ctrl+C is pressed, notify_sigint_from_ui() sets mwStopReq = true and kills all children.

---

8. Line Navigation (Ctrl+A / Ctrl+E)

Goal: Provide in-line cursor navigation similar to bash.

Implementation:

- Files: run.cpp
- Detected in: KeyPress event handler
- Logic:
    - Ctrl+A (ASCII 0x01): move cursor to start of input (currentCursorPosition = 0)
    - Ctrl+E (ASCII 0x05): move cursor to end (currentCursorPosition = input.length())
- Effect:
    - Updates tabState.currentCursorPosition
    - Redraws cursor using drawScreen()

---

9. Searchable Shell History (Ctrl+R)

Goal: Maintain and search through 10,000 past commands.

Implementation:

- Files: helper.cpp
- Functions:
    - storeInput() — appends new command to input_log.txt with serial number.
    - loadInputs() — loads entire file into inputs vector at startup.
    - searchHistory() — finds exact or prefix match.
    - commonPrefixLength() — helps fuzzy match if no exact match found.

How it works:

- History stored in ./input_log.txt as <index> <command>.
- Pressing Ctrl+R sets searchFlag = true and shows a search prompt.
- User input passed to searchHistory().
- If exact match found → loads it to input.
- If no exact match → finds commands with longest prefix match (>2 chars).
- Prints "No match for search term in history" if none found.

---

10. Auto-Complete for File Names (Tab)

Goal: Provide filename completion when typing commands.

Implementation:

- Files: run.cpp, helper.cpp
- Functions:
    - getQuery() — extracts last token typed.
    - getRecomm() — lists files matching prefix.
    - getRecIdx() — reads user's numeric selection when multiple matches.
- Execution flow:
    - Pressing Tab → run() calls execCommandInDir("ls", cwd) to list current directory.
    - getRecomm() filters list for names starting with the query prefix.
    - One match → auto-complete directly.
    - Multiple matches → print numbered list, store in tabState.recs, set recommFlag = true.
    - User can select via number key and press Enter to confirm.

5.MyTerm (swagnik@myterm) — Bonus Functionalities

---

1. Left / Right Arrow — Move Across Input

Purpose:
Allow the user to move the text cursor horizontally across the current command line while editing input.

Implementation Details:

- **File:** run.cpp
- **Logic:**
  When a KeyPress event corresponds to the left (XK_Left) or right (XK_Right) arrow keys:
  - If **Left Arrow** is pressed and currentCursorPosition > 0, it decrements the cursor index by one.
  - If **Right Arrow** is pressed and currentCursorPosition < input.size(), it increments by one.
- The visual cursor is redrawn by calling drawScreen() which recalculates the cursor's x-position using XTextWidth() for text up to that point.

Effect:
The user can freely move the cursor inside a partially typed command, edit text at any position, or insert characters anywhere along the line — just like in Bash.

---

2. Up / Down Arrows — Command History Navigation

Purpose:
Navigate through previously executed commands stored in the history file.

Implementation Details:

- **File:** run.cpp (with history data from helper.cpp)
- **Data:** inputs (loaded via loadInputs()), tabState.inpIdx
- **Logic:**
    - **Up Arrow:** Moves one step backward in the history (inpIdx--) and loads the previous command into input.
    - **Down Arrow:** Moves forward (inpIdx++) to newer commands or clears the line if at the most recent one.
- After each navigation, drawScreen() redraws the line showing the new command text.

Effect:
The user can quickly recall or edit previously executed commands directly from within the GUI terminal.

---

3. Ctrl+V— Paste Clipboard Text

Purpose:
Allow the user to paste text from the system clipboard into the terminal's input area.

Implementation Details:

- **File:** run.cpp
- **X11 Mechanism:**
    - On Ctrl+V or middle mouse button click, the program requests clipboard content via XConvertSelection().
    - When a SelectionNotify event arrives, the data is retrieved using XGetWindowProperty() and appended to tabState.input.
    - The GUI updates immediately via drawScreen().

Effect:
Users can paste text (including multi-line or Unicode) directly into the terminal input buffer — identical to typical GUI terminals.

---

4. Ctrl+Tab / Ctrl+Shift+Tab — Switch Between Tabs

Purpose:
Enable quick switching between open terminal tabs using keyboard shortcuts.

Implementation Details:

- **File:** run.cpp
- **Data:** vector<tabState> tabs, int tabActive
- **Logic:**
    - Detects ControlMask in combination with XK_Tab.
    - If ShiftMask is also pressed → switch to the **previous** tab (tabActive = (tabActive - 1 + tabs.size()) % tabs.size();).
    - If only Ctrl+Tab → switch to the **next** tab (tabActive = (tabActive + 1) % tabs.size();).
    - Calls drawScreen() to refresh the newly active tab display.

Effect:
Allows tab navigation entirely via keyboard, cycling forward or backward between open terminal instances.

---

5. Mouse Click — Add, Close, or Switch Tabs

Purpose:
Provide mouse-based tab control similar to browser interfaces.

Implementation Details:

- **Files:** drawscreen.cpp, run.cpp
- **Functions:** draw_tabs(), navbar_hit_test(), add_tab()
- **Logic:**
    - draw_tabs() draws all tabs, their labels, close ("×") icons, and the "+" add button.
    - When a ButtonPress event occurs:
        - If click is on "+" → add_tab() is called to open a new tab.
        - If click is on a close button → removes that tab from tabs.
        - If click is on a tab body → switches tabActive to that index.
    - drawScreen() updates the active tab content.

Effect:
Fully interactive tab management using the mouse — open, close, or switch tabs dynamically.

---

6. Blinking Cursor — Realistic Typing Experience

Purpose:
Visually replicate a real terminal's blinking cursor effect.

Implementation Details:

- **Files:** run.cpp, drawscreen.cpp
- **Variables:** tabState.dispCursor, tabState.lastBlink
- **Logic:**
    - In the main loop of run(), a timer checks if more than 500ms have passed since lastBlink.
    - Toggles dispCursor between true/false and calls drawScreen() to update visibility.
    - Cursor position is calculated precisely using text width via XTextWidth().

Effect:
The cursor blinks smoothly while typing, enhancing realism and UX.

---

8. Backspace / Delete — Edit Text Inline

Purpose:
Provide text-editing control similar to real shells.

Implementation Details:

- **File:** run.cpp
- **Variables:** tabState.input, tabState.currentCursorPosition
- **Logic:**
    - **Backspace:** Removes the character before the cursor and shifts cursor one step left.
    - **Delete:** Removes the character at the cursor without shifting left.
    - Buffer updates and redraw handled by drawScreen().

Effect:
Allows in-line editing of commands — deleting characters anywhere, not just at the end.

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
भारतीय प्रौद्योगिकी संस्थान खड़गपुर
1951
योगः कर्मसु कौशलम्