

Terminal Multiplexer: Design Document

Overview

A terminal multiplexer that manages multiple shell sessions through a single terminal interface, handling pseudo-terminal allocation, terminal emulation, and window management. The key architectural challenge is coordinating PTY I/O, escape sequence parsing, and multi-pane rendering while maintaining session isolation and terminal compatibility.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): All milestones (1-4) - this section establishes the foundational understanding needed across the entire project

Mental Model: The Terminal Switchboard

Imagine a busy telephone switchboard operator in the 1950s. Dozens of callers want to make phone calls simultaneously, but there's only one operator sitting at the central console. The operator's job is to connect each caller to their desired destination, manage multiple conversations at once, and switch between them seamlessly. When Caller A wants to talk, the operator plugs their line into the main console. When Caller B needs attention, the operator switches the connection. From each caller's perspective, they have the operator's full attention, even though the operator is actually juggling many conversations.

A **terminal multiplexer** works exactly like this switchboard operator, but instead of routing phone calls, it routes terminal input and output between multiple shell sessions. You have one physical terminal (your keyboard and screen), but you want to run multiple shell sessions - perhaps editing code in one, running tests in another, monitoring logs in a third, and debugging in a fourth. The multiplexer sits between your physical terminal and these multiple shell processes, creating the illusion that each shell has its own dedicated terminal.

When you type commands, the multiplexer routes your keystrokes to whichever shell session currently has "focus" - just like the operator connecting your voice to the right phone line. When shells produce output (listing files, showing error messages, displaying program results), the multiplexer captures that output and displays it on your screen, switching between different sessions as you request. Each shell process believes it has a real terminal attached, complete with the ability to move the cursor, change colors, and clear the screen, but it's actually communicating with a "fake" terminal that the multiplexer creates and manages.

The technical term for these fake terminals is **pseudo-terminals** or **PTYs**. Think of a PTY as a telephone extension - it looks and behaves exactly like a real phone from the caller's perspective, but it's actually connected

to the central switchboard system. The multiplexer creates one PTY for each shell session, managing the illusion that each has its own dedicated terminal hardware.

Just as the switchboard operator needs to understand telephone protocols (how to establish connections, handle busy signals, manage conference calls), the multiplexer needs to understand terminal protocols - the special codes that programs send to control cursor position, change text colors, clear the screen, and handle window resizing. This is where **terminal emulation** comes in: the multiplexer must speak the same language as both the physical terminal and the shell processes.

The switchboard analogy extends to window management as well. Imagine the operator has a large control panel that can be divided into sections, with each section showing the status of different phone conversations simultaneously. Similarly, a terminal multiplexer can split your screen into multiple **panes**, each showing a different shell session. You can see multiple conversations at once, resize the sections to give more space to important ones, and switch focus between them with keyboard shortcuts.

This mental model helps explain why building a terminal multiplexer is technically challenging: you're essentially building a sophisticated communications system that must maintain the illusion of multiple dedicated terminals while actually sharing a single physical terminal. Every keystroke, escape sequence, and terminal capability must be correctly routed and translated between the different abstraction layers.

Existing Approaches Comparison

The terminal multiplexing space has several mature solutions, each representing different architectural approaches and feature sets. Understanding their design decisions helps illuminate the core challenges and trade-offs in this problem domain.

GNU Screen represents the original approach to terminal multiplexing, developed in 1987 when terminal hardware was expensive and remote connections were unreliable. Screen's primary innovation was **session persistence** - the ability to detach from a terminal session and reattach later, potentially from a different machine. This was revolutionary for users connecting over slow dial-up or unreliable network connections.

Aspect	GNU Screen Approach	Strengths	Limitations
Architecture	Single-process session manager	Simple, stable, well-tested	Limited window management
Window Model	Linear window list (Ctrl-a n/p)	Easy to understand and navigate	No panes or splits within windows
Terminal Emulation	Basic ANSI/VT100 support	Compatible with most applications	Limited color and Unicode support
Configuration	Simple RC file format	Easy to configure	Limited customization options
Session Management	Strong detach/reattach support	Survives network disconnections	Sessions tied to single user
Resource Usage	Low memory footprint	Runs on minimal hardware	Single-threaded performance limits

Screen's architecture is fundamentally a **session container** - it creates a persistent environment that outlives individual terminal connections. When you run Screen, it forks into a daemon process that manages all the PTYs and shell processes. Your terminal connects to this daemon as a client. If the connection breaks, the daemon continues running, keeping all your shells alive. This design priority explains why Screen has limited window management features - it was built for persistence first, convenience second.

tmux (Terminal Multiplexer) emerged in 2007 as a modern reimplementation of Screen's concepts with a focus on **advanced window management** and a cleaner architecture. tmux's key innovation was treating terminal multiplexing as primarily a layout and presentation problem rather than just a session persistence problem.

Aspect	tmux Approach	Strengths	Limitations
Architecture	Client-server with multiple sessions	Clean separation, multiple clients	More complex setup
Window Model	Sessions → Windows → Panes hierarchy	Very flexible layouts	Steeper learning curve
Terminal Emulation	Modern terminal support (256 colors, UTF-8)	Excellent compatibility	Higher resource usage
Configuration	Rich scripting and customization	Extremely configurable	Configuration complexity
Session Management	Multiple named sessions	Better organization	Overkill for simple use cases
Layout Engine	Dynamic pane splitting and resizing	Professional workspace feel	Can be overwhelming

tmux's architecture centers around a **three-level hierarchy**: sessions contain windows, windows contain panes. This allows for much more sophisticated organization - you might have a "development" session with windows for "editing", "testing", and "debugging", where each window has multiple panes showing different tools. The tmux server can host multiple sessions simultaneously, and multiple clients can connect to different sessions or even the same session from different terminals.

The layout engine in tmux is particularly sophisticated. Instead of Screen's simple linear window switching, tmux implements a **tree-based layout algorithm** where panes can be split recursively in both horizontal and vertical directions. Each pane maintains its own PTY and screen buffer, but they share screen real estate dynamically. When you resize the terminal, tmux recalculates the entire layout tree and adjusts all panes proportionally.

Terminal Multiplexer in IDE Context represents a third approach seen in modern development environments. Tools like VS Code's integrated terminal, iTerm2's tmux integration, and terminal emulators with built-in multiplexing take a **native integration** approach rather than running as separate processes.

Aspect	IDE Integration Approach	Strengths	Limitations
Architecture	Embedded in larger application	Seamless user experience	Tied to specific environment
Window Model	Tabs and splits native to UI	Familiar interface patterns	Less keyboard-driven
Terminal Emulation	Full terminal emulator built-in	Perfect rendering compatibility	High development complexity
Configuration	GUI preferences integration	User-friendly setup	Less scriptable
Session Management	Application lifecycle tied	Simple mental model	No detach/reattach capability
Performance	Native code integration	Fast rendering and input	Memory usage of full application

This approach treats terminal multiplexing as a **user interface problem** rather than a systems programming problem. Instead of creating PTYs and managing processes directly, these solutions embed full terminal emulators within larger graphical applications. The benefit is seamless integration with other development tools, but the cost is loss of the lightweight, universal nature of traditional multiplexers.

Modern Alternatives like Zellij, Wezterm, and other recent multiplexers represent a fourth wave focused on **user experience modernization** while maintaining the core architectural patterns established by tmux.

Feature Category	Screen (1987)	tmux (2007)	Modern Tools (2020+)
Session Persistence	✓ Strong	✓ Strong	✓ Strong
Pane Management	✗ Windows only	✓ Hierarchical	✓ Advanced layouts
Visual Polish	✗ Basic	~ Good	✓ Excellent
Configuration	~ Simple	~ Powerful but complex	✓ User-friendly
Plugin Ecosystem	✗ Limited	✓ Rich	✓ Growing
Learning Curve	✓ Gentle	✗ Steep	~ Moderate
Resource Usage	✓ Minimal	~ Moderate	~ Higher
Maturity	✓ Battle-tested	✓ Mature	✗ Newer, less proven

Key Architectural Insight: The evolution from Screen to tmux to modern multiplexers reflects a fundamental tension between **simplicity and capability**. Screen prioritized reliability and minimal resource usage. tmux prioritized flexibility and power. Modern tools prioritize user experience and discoverability. Each represents valid engineering trade-offs for different use cases and user populations.

Core Technical Challenges Across All Approaches

Regardless of architectural approach, all terminal multiplexers must solve the same fundamental technical challenges:

PTY Management Complexity: Every multiplexer must create and manage pseudo-terminal pairs, which involves intricate systems programming. The PTY subsystem has many subtle requirements around process groups, session leadership, controlling terminals, and signal handling. A single mistake in PTY setup can cause shells to behave incorrectly or not start at all.

Terminal Protocol Compatibility: Modern terminal applications use hundreds of different escape sequences for cursor control, color management, window manipulation, and feature negotiation. The multiplexer must parse these sequences accurately and either handle them directly (for screen management commands) or pass them through to the underlying terminal (for display commands). Incorrect parsing leads to garbled output or broken application behavior.

Input/Output Performance: A multiplexer sits in the critical path between user input and program output. Any latency introduced by the multiplexer directly impacts the user experience. High-performance applications like text editors, games, or real-time monitoring tools are particularly sensitive to input lag or output buffering delays.

Layout and Rendering Coordination: When managing multiple panes, the multiplexer must solve complex geometric problems: how to divide available screen space, handle terminal resizing, manage scrolling and scrollback buffers, and composite multiple data streams into a single output stream without visual artifacts.

Process Lifecycle Management: Each shell session represents a separate process tree that must be properly managed. This includes handling process creation, signal forwarding, exit status collection, and cleanup of

resources when processes terminate. The multiplexer must also handle edge cases like orphaned processes, zombie processes, and cascading failures.

Design Principle: The fundamental challenge of terminal multiplexing is **maintaining multiple illusions simultaneously**. Each shell process must believe it has a dedicated terminal, each user must feel they're interacting directly with their programs, and the underlying terminal must receive a coherent stream of properly formatted output. The multiplexer succeeds when these illusions are invisible to all parties.

This context establishes why building a terminal multiplexer is an advanced systems programming project: it requires deep understanding of Unix process management, terminal protocols, event-driven programming, and user interface design, all while maintaining high performance and rock-solid reliability. The implementation must get low-level details exactly right while presenting a clean, intuitive interface to users.

Implementation Guidance

This implementation guidance focuses on establishing the foundational understanding and development environment needed to tackle the terminal multiplexer project.

A. Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
PTY Management	<code>posix_openpty()</code> + manual setup	<code>forkpty()</code> library function	Manual approach teaches fundamentals
Event Loop	<code>select()</code> system call	<code>epoll()</code> or <code>kqueue()</code>	<code>select()</code> is portable and sufficient
Terminal Control	Raw <code>termios</code> manipulation	<code>ncurses</code> library	Raw control teaches terminal protocols
Configuration	Hard-coded key bindings	Config file parsing	Focus on core functionality first
Logging	<code>fprintf(stderr, ...)</code>	syslog integration	Simple debugging adequate initially

For this learning-focused implementation, we recommend the simpler options that expose the underlying mechanisms rather than hiding them behind convenience libraries.

B. Recommended Project Structure

```
terminal-multiplexer/
├── src/
│   ├── main.c           ← Entry point and main event loop
│   ├── pty_manager.c    ← PTY creation and process management
│   ├── pty_manager.h
│   ├── terminal_emulator.c  ← Escape sequence parsing and screen buffers
│   ├── terminal_emulator.h
│   ├── window_manager.c  ← Pane layout and rendering
│   ├── window_manager.h
│   ├── input_handler.c   ← Key binding and command processing
│   ├── input_handler.h
│   └── common.h          ← Shared constants and data structures
├── tests/
│   ├── test_pty_manager.c
│   ├── test_terminal_emulator.c
│   └── test_integration.c
└── docs/
    └── escape_sequences.md   ← Reference for ANSI codes
└── Makefile
```

This structure separates concerns cleanly while keeping related functionality together. Each component gets its own source file with a matching header, making the codebase navigable as it grows.

C. Infrastructure Starter Code

Terminal State Management (complete implementation):

```
// common.h - Core data structures and constants
```

C

```
#ifndef COMMON_H
```

```
#define COMMON_H
```

```
#include <sys/types.h>
```

```
#include <termios.h>
```

```
#include <stdbool.h>
```

```
#define MAX_PANES 16
```

```
#define MAX_SCREEN_WIDTH 1024
```

```
#define MAX_SCREEN_HEIGHT 256
```

```
#define MAX_SCROLLBACK 1000
```

```
#define ESC_SEQ_BUFFER_SIZE 64
```

```
// Terminal cell representing one character position
```

```
typedef struct {
```

```
    char ch;
```

```
    unsigned char attributes; // Bold, underline, etc.
```

```
    unsigned char fg_color;
```

```
    unsigned char bg_color;
```

```
} terminal_cell_t;
```

```
// Screen buffer for one pane
```

```
typedef struct {
```

```
    terminal_cell_t cells[MAX_SCREEN_HEIGHT][MAX_SCREEN_WIDTH];
```

```
    int width, height;
```

```
    int cursor_x, cursor_y;
```

```
    int scroll_offset;
```

```
    terminal_cell_t scrollback[MAX_SCROLLBACK][MAX_SCREEN_WIDTH];
```

```
    int scrollback_lines;
```

```
    unsigned char current_attributes;

    unsigned char current_fg_color;

    unsigned char current_bg_color;

} screen_buffer_t;

// Forward declarations for main data structures

typedef struct pane pane_t;

typedef struct session session_t;

// Error handling utilities

typedef enum {

    TMUX_OK = 0,

    TMUX_ERROR_PTY_ALLOCATION,

    TMUX_ERROR_PROCESS_SPAWN,

    TMUX_ERROR_TERMINAL_SETUP,

    TMUX_ERROR_MEMORY_ALLOCATION

} tmux_error_t;

// Initialize a screen buffer with default values

void screen_buffer_init(screen_buffer_t* buffer, int width, int height);

// Terminal attribute constants

#define ATTR_BOLD      (1 << 0)

#define ATTR_UNDERLINE (1 << 1)

#define ATTR_REVERSE   (1 << 2)

// Color constants

#define COLOR_BLACK    0

#define COLOR_RED      1

#define COLOR_GREEN    2
```

```
#define COLOR_YELLOW 3

#define COLOR_BLUE     4

#define COLOR_MAGENTA 5

#define COLOR_CYAN    6

#define COLOR_WHITE   7

#define COLOR_DEFAULT 9

#endif // COMMON_H
```

```
// common.c - Implementation of utility functions

#include "common.h"

#include <string.h>

void screen_buffer_init(screen_buffer_t* buffer, int width, int height) {

    buffer->width = width;

    buffer->height = height;

    buffer->cursor_x = 0;

    buffer->cursor_y = 0;

    buffer->scroll_offset = 0;

    buffer->scrollback_lines = 0;

    buffer->current_attributes = 0;

    buffer->current_fg_color = COLOR_DEFAULT;

    buffer->current_bg_color = COLOR_DEFAULT;

    // Clear the screen with spaces

    for (int y = 0; y < height; y++) {

        for (int x = 0; x < width; x++) {

            buffer->cells[y][x].ch = ' ';

            buffer->cells[y][x].attributes = 0;

            buffer->cells[y][x].fg_color = COLOR_DEFAULT;

            buffer->cells[y][x].bg_color = COLOR_DEFAULT;

        }

    }

    // Initialize scrollback buffer

    memset(buffer->scrollback, 0, sizeof(buffer->scrollback));

}
```

C

Terminal Control Utilities (complete implementation):

```
// terminal_control.h - Raw terminal mode management C

#ifndef TERMINAL_CONTROL_H

#define TERMINAL_CONTROL_H

#include <termios.h>

typedef struct {

    struct termios original_termios;

    bool raw_mode_active;

} terminal_state_t;

// Enter raw terminal mode, saving original state

tmux_error_t terminal_enter_raw_mode(terminal_state_t* state);

// Restore original terminal mode

void terminal_restore_mode(terminal_state_t* state);

// Get current terminal window size

tmux_error_t terminal_get_size(int* width, int* height);

#endif
```

```
// terminal_control.c

#include "terminal_control.h"

#include "common.h"

#include <unistd.h>

#include <sys/ioctl.h>

tmux_error_t terminal_enter_raw_mode(terminal_state_t* state) {

    // Save original terminal attributes

    if (tcgetattr(STDIN_FILENO, &state->original_termios) != 0) {

        return TMUX_ERROR_TERMINAL_SETUP;

    }

    struct termios raw = state->original_termios;

    // Disable canonical mode and echo

    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    // Disable input processing

    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    // Disable output processing

    raw.c_oflag &= ~(OPOST);

    // Set character size to 8 bits

    raw.c_cflag |= CS8;

    // Set read timeout

    raw.c_cc[VMIN] = 0;    // Minimum characters to read

    raw.c_cc[VTIME] = 1;   // Timeout in deciseconds

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) != 0) {
```

C

```

    return TMUX_ERROR_TERMINAL_SETUP;

}

state->raw_mode_active = true;

return TMUX_OK;
}

void terminal_restore_mode(terminal_state_t* state) {
    if (state->raw_mode_active) {

        tcsetattr(STDIN_FILENO, TCSAFLUSH, &state->original_termios);

        state->raw_mode_active = false;
    }
}

tmux_error_t terminal_get_size(int* width, int* height) {
    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) != 0) {

        return TMUX_ERROR_TERMINAL_SETUP;
    }

    *width = ws.ws_col;

    *height = ws.ws_row;

    return TMUX_OK;
}

```

D. Core Logic Skeleton Code

The following skeleton provides the structure for the main components that learners will implement:

```
// pty_manager.h - Interface for PTY and process management

#ifndef PTY_MANAGER_H
#define PTY_MANAGER_H

#include "common.h"

typedef struct {

    int master_fd;

    int slave_fd;

    pid_t child_pid;

    bool process_running;

    int exit_status;

} pty_session_t;

// Create a new PTY session with a shell process

// This is the main function learners will implement in Milestone 1

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command);

// Read available data from PTY master

// Returns number of bytes read, 0 for no data, -1 for error

int pty_session_read(pty_session_t* session, char* buffer, size_t buffer_size);

// Write data to PTY master (send input to shell)

int pty_session_write(pty_session_t* session, const char* data, size_t length);

// Check if child process is still running

bool pty_session_is_alive(pty_session_t* session);

// Clean up PTY session

void pty_session_destroy(pty_session_t* session);
```

C

```
#endif
```

```
// pty_manager.c - Implementation skeleton

#include "pty_manager.h"

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <fcntl.h>

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command) {

    // TODO 1: Open PTY master using posix_openpty(0_RDWR)

    // TODO 2: Grant access to slave PTY using grantpt()

    // TODO 3: Unlock slave PTY using unlockpt()

    // TODO 4: Get slave PTY name using ptsname()

    // TODO 5: Fork child process

    // TODO 6: In child: call setsid() to become session leader

    // TODO 7: In child: open slave PTY and make it controlling terminal

    // TODO 8: In child: set up file descriptors (stdin, stdout, stderr → slave)

    // TODO 9: In child: exec shell command

    // TODO 10: In parent: store master_fd and child_pid, close slave_fd

    // TODO 11: Set master_fd to non-blocking mode using fcntl()

    // Initialize session structure

    session->master_fd = -1;

    session->slave_fd = -1;

    session->child_pid = -1;

    session->process_running = false;

    session->exit_status = 0;

    // Hint: Use /bin/sh if shell_command is NULL
```

C

```
// Hint: Remember to close unused file descriptors in child

// Hint: Check return value of each system call


return TMUX_ERROR_PTY_ALLOCATION; // Replace with proper implementation

}

int pty_session_read(pty_session_t* session, char* buffer, size_t buffer_size) {

    // TODO 1: Check if session is valid (master_fd >= 0)

    // TODO 2: Use read() system call on master_fd

    // TODO 3: Handle EAGAIN/EWOULDBLOCK (no data available)

    // TODO 4: Handle other errors appropriately

    // TODO 5: Return number of bytes read


    return -1; // Replace with proper implementation
}

int pty_session_write(pty_session_t* session, const char* data, size_t length) {

    // TODO 1: Check if session is valid and process is running

    // TODO 2: Use write() system call on master_fd

    // TODO 3: Handle partial writes (write might not send all data)

    // TODO 4: Handle EAGAIN/EWOULDBLOCK for non-blocking I/O

    // TODO 5: Return number of bytes written


    return -1; // Replace with proper implementation
}

bool pty_session_is_alive(pty_session_t* session) {

    // TODO 1: Check if child_pid is valid

    // TODO 2: Use waitpid() with WNOHANG to check process status
```

```

    // TODO 3: Update process_running and exit_status if process ended

    // TODO 4: Return current process_running status

    return false; // Replace with proper implementation

}

void pty_session_destroy(pty_session_t* session) {

    // TODO 1: Close master_fd if open

    // TODO 2: Send SIGTERM to child process if still running

    // TODO 3: Wait for child process to exit (with timeout)

    // TODO 4: Send SIGKILL if child doesn't exit gracefully

    // TODO 5: Clean up session structure

    // Implementation goes here

}

```

E. Language-Specific Hints for C Implementation

- **PTY Functions:** Use `posix_openpty()`, `grantpt()`, `unlockpt()`, and `ptsname()` for portable PTY allocation
- **Non-blocking I/O:** Set `O_NONBLOCK` using `fcntl(fd, F_SETFL, O_NONBLOCK)` after opening PTY master
- **Process Management:** Always check return values from `fork()`, `waitpid()`, and `exec*()` family functions
- **Signal Handling:** Install `SIGCHLD` handler to detect when child processes exit
- **Memory Safety:** Initialize all structure fields to safe values before use
- **Error Handling:** Use `errno` to get specific error information after system call failures

F. Milestone Checkpoint for Context Understanding

After understanding this context section, you should be able to:

1. **Explain the switchboard analogy:** Describe how a terminal multiplexer routes I/O between multiple sessions and one physical terminal
2. **Compare multiplexer architectures:** List key differences between Screen's session-first approach and tmux's layout-first approach

3. **Identify core technical challenges:** Name the four main problem areas (PTY management, terminal emulation, layout management, input handling)
4. **Set up development environment:** Compile and run the provided infrastructure code
5. **Understand project scope:** Distinguish between features you'll implement vs. advanced features that are out of scope

Test your understanding:

```
# Compile the infrastructure code
# gcc -o test_infrastructure common.c terminal_control.c -DTEST_INFRASTRUCTURE

# This should compile without errors and show terminal dimensions
./test_infrastructure
```

BASH

Expected output: Current terminal size and confirmation that raw mode can be entered/exited.

Signs you need to review:

- Can't explain why PTYs are necessary (review the switchboard analogy)
- Unclear on the difference between tmux sessions/windows/panes (review the comparison table)
- Infrastructure code doesn't compile (check that you have POSIX headers available)
- Confused about which features you'll build vs. which are out of scope (review the milestone descriptions)

This context establishes the mental framework and technical foundation needed to successfully implement each milestone of the terminal multiplexer project.

Goals and Non-Goals

Milestone(s): All milestones (1-4) - this section establishes the scope and boundaries that guide implementation across the entire project

Mental Model: The Feature Boundaries

Think of defining goals and non-goals like planning a camping trip. You need to decide what essential gear to pack (goals), what luxury items would be nice but aren't necessary (non-goals), and what equipment is completely out of scope for this particular adventure (explicit exclusions). Just as a successful camping trip requires clear boundaries on what you're trying to accomplish - whether it's a lightweight backpacking expedition or a comfortable car camping experience - building a terminal multiplexer requires explicit decisions about what functionality to include and what to leave for future versions.

The key insight is that every feature you choose NOT to implement is as important as every feature you choose to implement. Non-goals aren't failures or limitations - they're conscious architectural decisions that keep the project

focused, manageable, and aligned with its learning objectives. A terminal multiplexer could theoretically include dozens of advanced features, but for a learning project, we must ruthlessly prioritize the core concepts that provide the most educational value.

Functional Requirements

The terminal multiplexer will implement a focused set of core features that demonstrate the essential concepts of PTY management, terminal emulation, and window management. These requirements are carefully chosen to build understanding progressively, with each feature depending on and reinforcing the concepts learned in previous milestones.

PTY Management and Session Handling

The multiplexer must create and manage pseudo-terminal pairs that allow multiple shell sessions to run independently within a single terminal interface. This represents the fundamental capability that distinguishes a multiplexer from a simple shell - the ability to virtualize terminal access across multiple processes.

Requirement	Description	Success Criteria
PTY Creation	Allocate master/slave PTY pairs using <code>posix_openpt</code> or equivalent system calls	Returns valid file descriptors for both master and slave devices
Process Spawning	Fork child processes and execute shell commands with PTY as controlling terminal	Child process runs shell with correct terminal environment variables
Terminal Size Handling	Forward terminal size changes to child processes using <code>TIOCSWINSZ</code> ioctl	Pane resizing updates child PTY dimensions and shell applications respond correctly
Process Lifecycle	Monitor child process status and handle termination gracefully	Detect process exit via <code>SIGCHLD</code> and clean up PTY resources
I/O Multiplexing	Read output from multiple PTY masters and write input to active PTY	Select-based event loop handles multiple file descriptors without blocking

The PTY management system forms the foundation for all higher-level multiplexer functionality. Without reliable PTY creation and process management, features like window splitting and terminal emulation become meaningless. Each `pty_session_t` structure represents one virtualized terminal that believes it has exclusive access to a real terminal device.

Terminal Emulation and Display

The multiplexer must parse and interpret terminal escape sequences to maintain accurate virtual screen buffers for each pane. This functionality transforms the raw character stream from shell processes into structured display information that can be rendered and manipulated.

Requirement	Description	Success Criteria
ANSI Escape Parsing	Parse CSI sequences, control characters, and OSC commands from PTY output	Correctly interpret cursor movement, text attributes, and screen manipulation commands
Cursor Management	Track cursor position and handle movement commands (up, down, left, right)	Cursor position accurately reflects terminal state after escape sequence processing
Screen Buffer Maintenance	Maintain virtual screen buffer with character cells and attributes for each pane	Screen buffer accurately represents what should be displayed in pane
Text Attributes	Support bold, underline, reverse video, and 256-color text rendering	Text attributes applied correctly to character cells in screen buffer
Scrollbar Buffer	Retain lines that scroll off the top of the visible screen area	Users can review previous output up to <code>MAX_SCROLLBACK</code> lines
UTF-8 Support	Handle multi-byte UTF-8 character sequences correctly	Non-ASCII characters display properly without corruption

The terminal emulation component bridges the gap between the raw byte streams from PTY devices and the structured display information needed for rendering. Each `screen_buffer_t` represents the current visual state of one pane, including both visible content and scrollback history.

Critical Design Insight: Terminal emulation is not just about parsing escape sequences - it's about maintaining a stateful model of what the terminal display should look like. The escape sequences are commands that modify this state, and the screen buffer is the authoritative representation of the current display.

Window Management and Layout

The multiplexer must support splitting the terminal into multiple panes, each containing an independent shell session. This requires sophisticated layout algorithms and rendering logic to present multiple virtual terminals within a single physical terminal.

Requirement	Description	Success Criteria
Pane Splitting	Create vertical and horizontal splits that subdivide existing panes	New panes allocated with independent PTY sessions and screen buffers
Focus Management	Track which pane is currently active and route input appropriately	Keyboard input directed to correct PTY master based on focus state
Pane Resizing	Adjust pane dimensions while maintaining minimum viable sizes	Layout algorithm redistributes space proportionally while respecting constraints
Border Rendering	Draw borders between panes using box-drawing characters	Clear visual separation between panes without corrupting pane content
Layout Persistence	Maintain pane layout and proportions across terminal resize events	Pane relationships preserved when terminal window size changes

The window management system must handle the complex geometry calculations required to fit multiple rectangular panes within the available terminal space. The layout algorithm uses a tree structure where each node represents either a pane (leaf) or a split container (internal node).

User Interface and Key Bindings

The multiplexer must provide an intuitive command interface that allows users to control pane management, navigation, and multiplexer functions without interfering with the shell sessions running in individual panes.

Requirement	Description	Success Criteria
Prefix Key System	Intercept configurable prefix key (default Ctrl-b) to enter command mode	Prefix key captured in raw terminal mode before being sent to active pane
Pane Navigation	Keyboard shortcuts for switching focus between panes directionally	Arrow keys or vim-style navigation moves focus to adjacent panes
Split Commands	Key bindings for creating vertical and horizontal splits	Single keystrokes trigger pane splitting with new shell sessions
Pane Management	Commands for closing panes and adjusting pane sizes	Pane closure cleans up PTY resources and redistributes layout space
Status Bar	Display current pane information, session name, and system time	Status bar updates reflect focus changes and system state
Raw Terminal Mode	Capture all keyboard input while preserving ability to restore normal mode	Terminal attributes saved and restored correctly on multiplexer exit

The user interface system must carefully balance between intercepting control commands and passing normal input through to shell processes. The prefix key mechanism provides a clean separation between multiplexer commands and application input.

Architecture Decision: Prefix Key vs. Modal Interface

- **Context:** Users need a way to send commands to the multiplexer without interfering with applications running in panes
- **Options Considered:**
 - Prefix key system (like tmux Ctrl-b)
 - Modal interface (like screen's command mode)
 - Menu system with mouse support
- **Decision:** Prefix key system with configurable prefix
- **Rationale:** Prefix keys provide immediate access to multiplexer functions without mode confusion, familiar to users of existing multiplexers, and work reliably over SSH connections where mouse support may be limited
- **Consequences:** Users must remember key combinations, potential conflicts with application key bindings, but provides consistent and fast access to all multiplexer functions

Error Handling and Robustness

The multiplexer must handle various failure conditions gracefully, including PTY allocation failures, child process termination, terminal size changes, and malformed escape sequences.

Requirement	Description	Success Criteria
PTY Allocation Failure	Handle cases where PTY creation fails due to system limits	Graceful degradation with error messages, no resource leaks
Child Process Death	Detect and handle premature child process termination	Pane marked as dead, resources cleaned up, user notified of exit status
Terminal Corruption Recovery	Recover from malformed escape sequences or terminal state corruption	Parser resets to known state, screen buffer remains consistent
Resource Cleanup	Properly clean up PTY file descriptors, child processes, and memory on exit	No zombie processes, file descriptors closed, terminal attributes restored
Signal Handling	Handle SIGTERM, SIGINT, and SIGWINCH signals appropriately	Graceful shutdown on termination signals, terminal resize propagated to panes

Non-Goals

The following features are explicitly excluded from this implementation to maintain focus on the core learning objectives and keep the project scope manageable. These exclusions represent conscious architectural decisions, not limitations or oversights.

Session Persistence and Detachment

This multiplexer will NOT support session persistence, detaching from sessions, or reattaching to sessions across terminal disconnections. These features, while valuable in production multiplexers like tmux and screen, introduce significant complexity in session management, process daemonization, and inter-process communication that would distract from the core learning objectives.

Excluded Feature	Rationale	Alternative
Session Detachment	Requires complex daemon architecture, socket communication, session serialization	Run multiplexer in persistent terminal or use existing tools
Session Reattachment	Needs session discovery, state restoration, terminal synchronization	Start new multiplexer instance with fresh sessions
Background Operation	Process daemonization, signal handling, log management complexity	Keep multiplexer as foreground process
Session Naming/Listing	Persistent session registry, inter-process communication protocols	Single session per multiplexer instance

Session persistence would require implementing a client-server architecture where the multiplexer runs as a background daemon and terminal instances connect as clients. This architectural complexity would overshadow the PTY management and terminal emulation concepts that are the primary learning goals.

Networking and Remote Access

The multiplexer will NOT include networking capabilities, remote session access, or SSH integration. While these features enable powerful distributed terminal management scenarios, they introduce network programming, authentication, and security concerns that are beyond the scope of a terminal multiplexer learning project.

Excluded Feature	Rationale	Alternative
Network Protocol	Socket programming, protocol design, security considerations distract from terminal concepts	Use SSH port forwarding or VNC for remote access
Authentication/Authorization	User management, credential verification, access control complexity	Rely on system-level authentication
Multi-User Sessions	Shared session access, concurrent user management, conflict resolution	Single-user local multiplexer only
Encryption/Security	TLS implementation, key management, secure communication protocols	Use SSH tunneling for secure remote access

Remote access functionality would require designing network protocols, handling connection management, and implementing security measures that are entirely separate from the terminal multiplexing concepts we aim to teach.

Advanced Terminal Features

The multiplexer will NOT implement advanced terminal capabilities that require extensive terminal database knowledge, complex escape sequence handling, or specialized hardware emulation beyond basic ANSI compatibility.

Excluded Feature	Rationale	Alternative
Full Terminal Database	Terminfo/termcap integration requires extensive terminal knowledge database	Focus on common ANSI sequences, rely on TERM environment
Mouse Support	Complex coordinate tracking, event parsing, application coordination	Keyboard-only interface
Terminal Profiles	Color schemes, font settings, appearance customization	Use terminal emulator's native configuration
Advanced Graphics	Sixel graphics, image display, custom character sets	Focus on text-based interface
Complex Scrolling	Alternate screen buffer, application-controlled scrolling modes	Simple scrollback buffer only

These features would require implementing large portions of a full terminal emulator, which would shift focus away from the multiplexing and PTY management concepts.

Scripting and Automation

The multiplexer will NOT include a scripting interface, configuration language, or extensive automation capabilities. These features require language design, parser implementation, and API development that would significantly expand the project scope.

Excluded Feature	Rationale	Alternative
Configuration Files	File parsing, validation, option management complexity	Compile-time constants and simple runtime options
Command Scripts	Scripting language design, interpreter implementation	Manual command execution or external shell scripts
API/Plugin System	Dynamic loading, plugin architecture, API versioning	Monolithic implementation with fixed feature set
Macro Recording	Command recording, playback, macro management	Manual command repetition or external automation tools
Event Hooks	Event system, callback management, user-defined actions	Fixed behavior with no customization

Automation features would require building a complete scripting environment, which is a substantial project in itself and would overshadow the terminal multiplexing learning objectives.

Design Philosophy: Learning Focus vs. Feature Completeness

The conscious decision to exclude these advanced features reflects a fundamental principle: this project prioritizes deep understanding of core concepts over broad feature coverage. By limiting scope to PTY management, terminal emulation, window management, and basic user interface, learners can focus on mastering the essential techniques that make terminal multiplexing possible.

Each excluded feature represents a potential future learning project. Session persistence could be added as an advanced exercise in daemon programming and IPC. Networking support could become a distributed systems learning project. Advanced terminal features could form the basis for a complete terminal emulator implementation.

Performance Optimization

The multiplexer will NOT include performance optimizations, advanced rendering techniques, or scalability features designed for handling large numbers of panes or high-throughput scenarios.

Excluded Feature	Rationale	Alternative
Efficient Rendering	Screen damage tracking, differential updates, rendering optimization	Simple full-screen refresh for clarity
Memory Optimization	Custom allocators, memory pooling, buffer recycling	Standard malloc/free with focus on correctness
High Pane Counts	Tree balancing, layout optimization, resource scaling	Limit to <code>MAX_PANES</code> reasonable for learning
Asynchronous I/O	epoll/kqueue, async event handling, high-concurrency support	Simple select-based synchronous I/O
Buffer Compression	Scrollback compression, memory-efficient storage	Fixed-size buffers with reasonable limits

Performance optimization would introduce algorithmic complexity and system-specific code that would obscure the fundamental concepts being taught. The focus remains on correctness and clarity over efficiency.

Requirements Validation

The functional requirements and non-goals work together to create a coherent learning experience that builds understanding progressively while maintaining reasonable project scope. Each included feature directly supports one or more of the core learning objectives: PTY handling, terminal escape sequences, and process management.

The milestone structure maps directly to these requirements:

- **Milestone 1 (PTY Creation)** addresses PTY management and session handling requirements
- **Milestone 2 (Terminal Emulation)** addresses terminal emulation and display requirements
- **Milestone 3 (Window Management)** addresses window management and layout requirements
- **Milestone 4 (Key Bindings and UI)** addresses user interface and key binding requirements

Error handling and robustness requirements span all milestones, ensuring that each component includes appropriate failure detection and recovery mechanisms.

Validation Principle: Every functional requirement must satisfy two criteria: (1) it teaches a fundamental concept in terminal multiplexing, and (2) it can be implemented and tested within the milestone structure. Every non-goal must satisfy two criteria: (1) excluding it doesn't prevent learning the core concepts, and (2) including it would add complexity that outweighs its educational value.

The resulting feature set provides a complete, working terminal multiplexer that demonstrates all essential concepts while remaining implementable as a learning project. Users will be able to split panes, run multiple shell sessions, navigate between panes, and perform basic terminal operations - sufficient functionality to understand how production multiplexers work internally.

Implementation Guidance

This section provides concrete direction for translating the goals and requirements into working code, focusing on the architectural decisions and technology choices that support the defined scope.

Technology Recommendations

Component	Simple Option	Advanced Option
PTY Management	<code>posix_openpt()</code> with manual setup	<code>forkpty()</code> from libutil
Terminal I/O	<code>select()</code> with blocking I/O	<code>epoll</code> / <code>kqueue</code> async I/O
Escape Parsing	Hand-written state machine	<code>libvterm</code> or similar library
Screen Buffers	2D character arrays	Optimized line-based storage
Layout Management	Recursive tree with fixed splits	Dynamic layout with weights
Configuration	Compile-time constants	Runtime config file parsing

For this learning project, consistently choose the simple options. The goal is understanding the underlying mechanisms, not building the most efficient implementation. Using `posix_openpt()` teaches PTY allocation explicitly, while `forkpty()` hides the educational details. Hand-writing the escape sequence parser reveals how terminal emulation works, while using `libvterm` treats it as a black box.

Recommended File Structure

The implementation should organize code to reflect the component architecture and support incremental development through the milestone structure.

```
terminal-multiplexer/
├── src/
│   ├── main.c                         ← Entry point and main event loop
│   ├── pty_manager.c                  ← PTY creation and process management (Milestone 1)
│   ├── pty_manager.h
│   ├── terminal_emulator.c           ← Escape sequence parsing and screen buffers (Milestone 2)
│   ├── terminal_emulator.h
│   ├── window_manager.c              ← Pane layout and rendering (Milestone 3)
│   ├── window_manager.h
│   ├── input_handler.c               ← Key bindings and command processing (Milestone 4)
│   ├── input_handler.h
│   ├── types.h                       ← Core data structures and constants
│   └── utils.c                       ← Utility functions and error handling
│       └── utils.h
└── tests/
    ├── test_pty_manager.c            ← Unit tests for PTY functionality
    ├── test_terminal_emulator.c      ← Parser and buffer tests
    ├── test_window_manager.c         ← Layout and rendering tests
    └── integration_test.c           ← End-to-end multiplexer tests
Makefile
README.md                         ← Build configuration
                                    ← Usage instructions and examples
```

This structure allows implementing and testing each component independently while maintaining clear separation of concerns. The header files define the interfaces between components, enabling modular development.

Core Data Structure Definitions

Start by defining the complete type system in `types.h`. These structures form the foundation for all multiplexer functionality:

```
// types.h - Complete type definitions for terminal multiplexer C

#include <sys/types.h>

#include <termios.h>

#include <stdbool.h>

// Terminal character cell with attributes and colors

typedef struct {

    char ch;                      // Character to display (UTF-8 support limited to ASCII for
simplicity)

    unsigned char attributes;     // Bit flags for bold, underline, reverse

    unsigned char fg_color;       // Foreground color (0-255)

    unsigned char bg_color;       // Background color (0-255)

} terminal_cell_t;

// Virtual screen buffer for one pane

typedef struct {

    terminal_cell_t *cells;      // 2D array stored as 1D: cells[row * width + col]

    int width;                   // Current screen width in characters

    int height;                  // Current screen height in characters

    int cursor_x;                // Current cursor column position

    int cursor_y;                // Current cursor row position

    int scroll_offset;           // Number of lines scrolled up from bottom

    terminal_cell_t **scrollback; // Array of scrollback lines

    int scrollback_lines;        // Number of lines currently in scrollback

    unsigned char current_attributes; // Current text attributes for new characters

    unsigned char current_fg_color; // Current foreground color

    unsigned char current_bg_color; // Current background color

} screen_buffer_t;
```

```

// PTY session managing one shell process

typedef struct {

    int master_fd;           // File descriptor for PTY master

    int slave_fd;            // File descriptor for PTY slave (used during setup)

    pid_t child_pid;         // Process ID of shell running in this PTY

    bool process_running;    // True if child process is still alive

    int exit_status;          // Exit status if process has terminated

} pty_session_t;

// Terminal state management for raw mode

typedef struct {

    struct termios original_termios; // Original terminal attributes for restoration

    bool raw_mode_active;           // True if terminal is currently in raw mode

} terminal_state_t;

// Error codes for multiplexer operations

typedef enum {

    TMUX_OK = 0,

    TMUX_ERROR_PTY_ALLOCATION,

    TMUX_ERROR_PROCESS_SPAWN,

    TMUX_ERROR_TERMINAL_SETUP,

    TMUX_ERROR_MEMORY_ALLOCATION

} tmux_error_t;

```

These structures provide the complete data model needed for all four milestones. Each structure includes all fields that will be needed, even if early milestones don't use them immediately.

Infrastructure Starter Code

Provide complete utility functions that handle the non-educational but necessary infrastructure:

```
// utils.c - Complete utility functions
```

C

```
#include "utils.h"
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

```
#include <sys/ioctl.h>
```

```
// Get current terminal window size
```

```
tmux_error_t terminal_get_size(int *width, int *height) {
```

```
    struct winsize ws;
```

```
    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1) {
```

```
        return TMUX_ERROR_TERMINAL_SETUP;
```

```
}
```

```
    *width = ws.ws_col;
```

```
    *height = ws.ws_row;
```

```
    return TMUX_OK;
```

```
}
```

```
// Initialize screen buffer with given dimensions
```

```
tmux_error_t screen_buffer_init(screen_buffer_t *buffer, int width, int height) {
```

```
    buffer->width = width;
```

```
    buffer->height = height;
```

```
    buffer->cursor_x = 0;
```

```
    buffer->cursor_y = 0;
```

```
    buffer->scroll_offset = 0;
```

```
    buffer->current_attributes = 0;
```

```
    buffer->current_fg_color = COLOR_WHITE;
```

```
    buffer->current_bg_color = COLOR_BLACK;
```

```
// Allocate main screen buffer

buffer->cells = calloc(width * height, sizeof(terminal_cell_t));

if (!buffer->cells) {

    return TMUX_ERROR_MEMORY_ALLOCATION;

}

// Allocate scrollback buffer

buffer->scrollback = calloc(MAX_SCROLLBACK, sizeof(terminal_cell_t *));

if (!buffer->scrollback) {

    free(buffer->cells);

    return TMUX_ERROR_MEMORY_ALLOCATION;

}

buffer->scrollback_lines = 0;

// Initialize all cells to default state

for (int i = 0; i < width * height; i++) {

    buffer->cells[i].ch = ' ';

    buffer->cells[i].attributes = 0;

    buffer->cells[i].fg_color = COLOR_WHITE;

    buffer->cells[i].bg_color = COLOR_BLACK;

}

return TMUX_OK;

}

// Enter raw terminal mode, saving original state

tmux_error_t terminal_enter_raw_mode(terminal_state_t *state) {
```

```
// TODO: Implement raw mode setup

// Hint: Use tcgetattr to save original, tcsetattr to set raw mode

// Disable canonical mode, echo, signals, special character processing

}

// Restore original terminal attributes

tmux_error_t terminal_restore_mode(terminal_state_t *state) {

    // TODO: Implement terminal restoration

    // Hint: Use tcsetattr with saved original_termios

    // This is CRITICAL - must be called on exit or terminal will be unusable

}
```

This infrastructure code handles memory management, terminal sizing, and basic buffer operations that are necessary but not the core learning focus.

Core Component Skeletons

For each major component, provide function signatures with detailed TODO comments that map to the architectural concepts:

```
// pty_manager.h - PTY management interface C

// Create new PTY session with shell process

tmux_error_t pty_session_create(pty_session_t *session, const char *shell_command) {

    // TODO 1: Open PTY master using posix_openpty(O_RDWR | O_NOCTTY)

    // TODO 2: Grant access to slave with grantpt(master_fd)

    // TODO 3: Unlock slave PTY with unlockpt(master_fd)

    // TODO 4: Get slave device name with ptsname(master_fd)

    // TODO 5: Fork child process

    // TODO 6: In child: call setsid() to create new session

    // TODO 7: In child: open slave PTY and set as controlling terminal

    // TODO 8: In child: dup2 slave to stdin/stdout/stderr

    // TODO 9: In child: exec shell command

    // TODO 10: In parent: store child PID and master FD

    // Pitfall: Must call setsid() before opening slave, or controlling terminal setup fails

    // Pitfall: Close slave FD in parent process to avoid blocking on child exit

}

// Read data from PTY master (shell output)

ssize_t pty_session_read(pty_session_t *session, char *buffer, size_t size) {

    // TODO 1: Check if process is still running using kill(pid, 0)

    // TODO 2: Use read() on master_fd with non-blocking I/O

    // TODO 3: Handle EAGAIN for non-blocking reads

    // TODO 4: Handle EOF when child process exits

    // TODO 5: Update process_running status on errors

    // Hint: Use fcntl(fd, F_SETFL, O_NONBLOCK) for non-blocking I/O

}

// Write data to PTY master (user input to shell)
```

```

ssize_t pty_session_write(pty_session_t *session, const char *data, size_t length) {

    // TODO 1: Verify process is still running

    // TODO 2: Use write() on master_fd

    // TODO 3: Handle partial writes by tracking bytes written

    // TODO 4: Handle EPIPE if child process has exited

    // Pitfall: write() may not write all bytes at once - must loop for partial writes

}

```

Each TODO comment corresponds to a specific step in the algorithms described in the design sections, providing clear implementation guidance while leaving the actual coding as a learning exercise.

Milestone Checkpoints

Define specific verification criteria for each milestone:

Milestone 1 Checkpoint - PTY Creation:

- Compile with: `make && ./terminal-multiplexer`
- Expected behavior: Single pane shows shell prompt, can execute commands
- Test command: `ls -la` should show directory listing in pane
- Verification: `ps aux | grep bash` shows child shell process
- Signs of trouble: "No PTY" errors indicate allocation failure, zombie processes indicate cleanup issues

Milestone 2 Checkpoint - Terminal Emulation:

- Test command: `echo -e "\033[31mRed Text\033[0m"` should show colored text
- Test cursor movement: `echo -e "\033[10;5HPositioned"` should place text at specific location
- Test clear screen: `clear` command should blank the pane
- Verification: Screen buffer contents match expected terminal state
- Signs of trouble: Garbled text indicates parser issues, missing colors indicate attribute handling problems

Milestone 3 Checkpoint - Window Management:

- Key sequence: Prefix key + `%` should create vertical split
- Key sequence: Prefix key + `"` should create horizontal split
- Key sequence: Prefix key + arrow keys should switch between panes
- Verification: Each pane runs independent shell, borders drawn correctly
- Signs of trouble: Overlapping content indicates rendering issues, shared input indicates focus problems

Milestone 4 Checkpoint - Complete Interface:

- Status bar shows current pane number and time

- All key bindings respond correctly
- Terminal restoration works on exit (Ctrl-C or normal quit)
- Verification: Full multiplexer functionality matches specification
- Signs of trouble: Unusable terminal after exit indicates restoration failure

Language-Specific Implementation Notes

For C implementation, key technical considerations:

- **PTY Allocation:** Use `posix_openpt()` rather than opening `/dev/ptmx` directly for portability
- **Signal Handling:** Install `SIGCHLD` handler to detect child process termination: `signal(SIGCHLD, child_handler)`
- **Non-blocking I/O:** Use `fcntl(fd, F_SETFL, O_NONBLOCK)` on PTY master to prevent blocking reads
- **Terminal Control:** Save terminal state with `tcgetattr(STDIN_FILENO, &original)` before entering raw mode
- **Memory Management:** Use `calloc()` for screen buffers to ensure clean initialization
- **Error Handling:** Check all system call return values - PTY operations can fail in various ways

Common Implementation Pitfalls

Symptom	Likely Cause	How to Diagnose	Fix
"Permission denied" on PTY	Missing <code>grantpt()</code> call	Check PTY creation sequence	Call <code>grantpt()</code> after <code>posix_openpt()</code>
Child processes become zombies	No <code>SIGCHLD</code> handler	Check <code>ps aux</code> for zombie processes	Install signal handler to <code>waitpid()</code>
Terminal corrupted on exit	Raw mode not restored	Terminal requires reset after program exit	Always call <code>terminal_restore_mode()</code> in cleanup
Input not reaching shell	Slave PTY not set as controlling terminal	Shell doesn't respond to input	Call <code>setsid()</code> in child before opening slave
Partial screen updates	Inefficient rendering	Text appears gradually or incorrectly	Implement double-buffering or complete screen refresh

High-Level Architecture

Milestone(s): All milestones (1-4) - this section establishes the architectural foundation that guides implementation across the entire project

Component Overview: The Four Main Components

Mental Model: The Orchestra Conductor

Think of a terminal multiplexer as an orchestra conductor coordinating four specialized musicians. Each musician has a distinct role but must work in perfect harmony to create a seamless performance. The **PTY Manager** is like the string section - creating the fundamental sound (shell sessions) that everything else builds upon. The **Terminal Emulator** acts as the woodwinds - interpreting and translating the musical score (escape sequences) into meaningful display changes. The **Window Manager** serves as the brass section - providing structure and visual organization that makes the performance coherent to the audience. Finally, the **Input Handler** functions as the percussion - keeping time and routing control signals that coordinate the entire ensemble.

Just as a conductor must understand each section's capabilities and limitations while maintaining perfect timing between them, our multiplexer architecture must carefully orchestrate data flow between these four components while preserving the illusion that each shell session has a dedicated terminal.

The PTY Manager Component

The **PTY Manager** serves as the foundation of our terminal multiplexer, responsible for creating and maintaining the pseudo-terminal pairs that enable shell sessions to run independently. This component encapsulates all the complex details of PTY allocation, process spawning, and session lifecycle management that form the core of Milestone 1.

The PTY Manager maintains a collection of active sessions, each represented by a `pty_session_t` structure that tracks the master and slave file descriptors, child process ID, and session state. When a new pane is created, the PTY Manager allocates a fresh pseudo-terminal pair using `posix_openpty()`, configures the terminal attributes to match the current terminal settings, and forks a child process that will run the shell session.

Responsibility	Description	Key Data Structures
PTY Allocation	Create master/slave PTY pairs with proper permissions	<code>pty_session_t</code> with <code>master_fd</code> , <code>slave_fd</code> fields
Process Spawning	Fork child processes and exec shells with PTY as controlling terminal	<code>child_pid</code> , <code>process_running</code> fields
Session Lifecycle	Track process state, handle SIGCHLD, manage session termination	<code>exit_status</code> field, signal handlers
I/O Management	Provide read/write interface for PTY master file descriptors	Buffer management, non-blocking I/O
Terminal Sizing	Forward TIOCSWINSZ ioctl calls to adjust PTY dimensions	Window size change propagation

The PTY Manager abstracts away the complexity of pseudo-terminal management from other components. When the Window Manager needs to create a new pane, it simply calls `pty_session_create()` and receives a

ready-to-use session structure. When the Terminal Emulator needs to send input to a shell, it uses `pty_session_write()` without concerning itself with file descriptor management or process state.

Key Design Insight: The PTY Manager acts as a protective barrier between the multiplexer and the underlying Unix process model. By encapsulating all PTY and process management logic in this component, we can change process handling strategies (like adding process group management or session persistence) without affecting the rest of the system.

The Terminal Emulator Component

The **Terminal Emulator** transforms the raw output streams from shell processes into structured screen representations that can be displayed and manipulated. This component implements the complex logic of Milestone 2, parsing ANSI escape sequences and maintaining virtual screen buffers for each active pane.

At its heart, the Terminal Emulator contains a finite state machine that processes character streams byte by byte, distinguishing between printable characters that should appear on screen and control sequences that modify cursor position, text attributes, or screen content. The parser maintains state across multiple input chunks, handling cases where escape sequences span buffer boundaries or arrive incomplete.

Parser State	Purpose	Transitions	Actions Taken
NORMAL_TEXT	Processing regular printable characters	ESC → ESCAPE_START	Add characters to screen buffer at cursor position
ESCAPE_START	Detected ESC byte, waiting for sequence type	'[' → CSI_PARAMS, 'O' → SS3_SEQUENCE	Set up sequence parsing context
CSI_PARAMS	Collecting numeric parameters for CSI sequences	'0'-'9', ';' → CSI_PARAMS, 'A'-'Z' → NORMAL_TEXT	Build parameter list, execute cursor/display commands
OSC_SEQUENCE	Processing Operating System Command sequences	'\x07' or '\x1b' → NORMAL_TEXT	Handle title changes, color settings

Each pane maintains its own `screen_buffer_t` structure that represents the current display state. The screen buffer contains a two-dimensional array of `terminal_cell_t` structures, each storing a character along with its display attributes (bold, underline, colors). The Terminal Emulator updates this buffer in response to parsed escape sequences, handling cursor movement, text insertion, line scrolling, and screen clearing operations.

The Terminal Emulator also manages a scrollback buffer that preserves lines that scroll off the top of the visible area. This buffer enables users to review previous command output and supports copy mode functionality where users can select and copy text from the scrollback history.

Critical Architecture Decision: We maintain separate screen buffers per pane rather than trying to multiplex output at the character level. This design allows each pane to have independent scrollback history, cursor state, and display attributes while simplifying the rendering logic that composes multiple panes into terminal output.

The Window Manager Component

The **Window Manager** orchestrates the visual layout of multiple panes within the terminal window, implementing the sophisticated space allocation and rendering logic required for Milestone 3. This component treats the terminal screen as a canvas that must be dynamically partitioned among competing panes while providing visual feedback about pane boundaries and focus state.

The Window Manager employs a tree-based layout algorithm where each node represents either a leaf pane containing a shell session or an internal split node that divides available space between its children. Split nodes can be either horizontal (dividing space top-to-bottom) or vertical (dividing space left-to-right), creating a recursive partitioning system that can represent arbitrarily complex layouts.

Layout Node Type	Data Fields	Responsibilities
Leaf Pane	<code>session_ptr</code> , <code>screen_buffer</code> , <code>is_focused</code>	Render single pane content with borders
Horizontal Split	<code>top_child</code> , <code>bottom_child</code> , <code>split_ratio</code>	Divide height between children, handle resize
Vertical Split	<code>left_child</code> , <code>right_child</code> , <code>split_ratio</code>	Divide width between children, handle resize

When the terminal window is resized, the Window Manager recalculates the entire layout tree, propagating new dimensions down to leaf panes and updating the corresponding PTY sizes through `TIOCSWINSZ` ioctl calls. This ensures that shell programs receive accurate terminal dimensions and can adjust their output accordingly.

The Window Manager's rendering pipeline processes the layout tree in depth-first order, compositing each pane's screen buffer into the final terminal output. It draws border characters between adjacent panes, highlights the currently focused pane with visual indicators, and reserves space for a status bar that displays pane information and system status.

Space Management Philosophy: The Window Manager treats screen real estate as a precious resource that must be allocated fairly while respecting minimum usability constraints. Each pane must be large enough to display at least one line of shell output, but beyond that minimum, space is distributed proportionally based on user-initiated resize operations.

The Input Handler Component

The **Input Handler** manages the critical task of routing keyboard input between the multiplexer's control functions and the active shell sessions, implementing the sophisticated input processing required for Milestone 4. This component must operate in raw terminal mode to capture all keystrokes while providing a clean abstraction that separates multiplexer commands from normal terminal input.

The Input Handler implements a two-mode system: **pass-through mode** where keystrokes are forwarded directly to the focused pane's shell, and **command mode** triggered by a configurable prefix key (typically Ctrl-b). In command mode, the Input Handler interprets subsequent keystrokes as multiplexer commands for pane management, window navigation, and system control.

Input Mode	Trigger	Key Bindings	Destination
Pass-through	Default state	All keys except prefix	Active pane's PTY session
Command Mode	Prefix key pressed	'c' → new pane, 'h/j/k/l' → navigate, '%' → vsplit	Window Manager operations
Copy Mode	Command + '['	Arrow keys → cursor, Space → select, Enter → copy	Screen buffer navigation

The Input Handler maintains a registry of key bindings that maps keystroke sequences to multiplexer functions. This registry supports both single-character commands and multi-character sequences, allowing for sophisticated key binding schemes that don't conflict with normal shell usage. The component also handles special keys like arrow keys and function keys that generate multi-byte escape sequences.

Raw terminal mode management represents one of the Input Handler's most critical responsibilities. When the multiplexer starts, it must save the current terminal attributes using `tcgetattr()`, switch to raw mode to disable line buffering and special character processing, and ensure that original terminal settings are restored even if the program exits unexpectedly due to signals or errors.

Input Routing Complexity: The challenge isn't just capturing keystrokes - it's maintaining the illusion that each shell session has a dedicated terminal while intercepting just enough input to provide multiplexer functionality. This requires careful coordination between raw terminal mode, signal handling, and prefix key detection.

Component Interaction Patterns

The four components interact through well-defined interfaces that maintain clear separation of concerns while enabling efficient data flow. The PTY Manager produces character streams from shell processes, the Terminal Emulator consumes these streams and produces structured screen buffers, the Window Manager consumes screen buffers and produces terminal output, and the Input Handler coordinates the entire process by routing commands and managing system state.

Data Flow Path	Source Component	Destination Component	Data Format	Frequency
Shell Output	PTY Manager	Terminal Emulator	Raw byte streams with escape sequences	Continuous during shell activity
Screen Updates	Terminal Emulator	Window Manager	Structured screen buffers with cursor state	After each escape sequence batch
Terminal Rendering	Window Manager	Terminal (stdout)	Composed display with borders and status	On screen buffer changes
User Input	Input Handler	PTY Manager	Filtered keystroke streams	Real-time during user interaction
Control Commands	Input Handler	Window Manager	Split, resize, focus commands	On prefix key sequences

Architecture Decision: Event-Driven Coordination

- Context:** Components need to coordinate without tight coupling while maintaining responsive performance
- Options Considered:** Direct method calls, message passing, shared memory, event-driven with select()
- Decision:** Event-driven architecture using select() for I/O multiplexing with direct method calls for control operations
- Rationale:** Select-based I/O handles multiple PTY file descriptors efficiently while direct calls provide synchronous control operations without message passing overhead
- Consequences:** Enables responsive input handling and efficient resource utilization but requires careful state management across components

Recommended Module Structure

The terminal multiplexer implementation should be organized into focused modules that correspond directly to the architectural components while providing clean separation between interface definitions and implementation details. This structure supports incremental development where each milestone builds upon previous modules without requiring significant refactoring.

Directory Organization

```
terminal-multiplexer/
├── src/
│   ├── main.c           ← Program entry point and main event loop
│   ├── tmux_types.h     ← Core data structure definitions
│   ├── tmux_constants.h ← System-wide constants and enums
│   ├── pty_manager/
│   │   ├── pty_manager.h  ← PTY management interface
│   │   ├── pty_manager.c  ← PTY creation, process spawning
│   │   └── pty_session.c  ← Individual session lifecycle
│   ├── terminal_emulator/
│   │   ├── terminal_emulator.h  ← Terminal emulation interface
│   │   ├── escape_parser.c    ← ANSI escape sequence parsing
│   │   ├── screen_buffer.c   ← Virtual screen buffer management
│   │   └── scrollback.c      ← Scrollback buffer implementation
│   ├── window_manager/
│   │   ├── window_manager.h  ← Window management interface
│   │   ├── layout_tree.c     ← Pane layout algorithms
│   │   ├── pane_renderer.c  ← Individual pane rendering
│   │   └── compositor.c     ← Multi-pane composition
│   ├── input_handler/
│   │   ├── input_handler.h  ← Input processing interface
│   │   ├── key_bindings.c   ← Key binding registry and matching
│   │   ├── terminal_control.c  ← Raw mode and terminal state
│   │   └── command_processor.c  ← Multiplexer command execution
│   └── utils/
│       ├── error_handling.h  ← Error codes and handling utilities
│       ├── memory_pool.c    ← Memory management for screen buffers
│       └── debug_logging.c  ← Development and debugging support
└── tests/
    ├── unit/
    │   ├── test_pty_manager.c
    │   ├── test_escape_parser.c
    │   ├── test_layout_tree.c
    │   └── test_key_bindings.c
    └── integration/
        ├── test_full_session.c
        └── test_multi_pane.c
└── Makefile
└── README.md
```

Module Dependency Architecture

The module structure enforces a clear dependency hierarchy that prevents circular dependencies while enabling efficient compilation and testing. Lower-level modules provide services to higher-level modules without requiring knowledge of their clients' implementation details.

Module Layer	Components	Dependencies	Exports
Foundation	<code>tmux_types.h</code> , <code>tmux_constants.h</code> , <code>utils/</code>	Standard C library only	Core data structures, error codes, utilities
Session Layer	<code>pty_manager/</code>	Foundation layer	PTY allocation, session management, I/O operations
Processing Layer	<code>terminal_emulator/</code>	Foundation + Session layers	Escape sequence parsing, screen buffer management
Presentation Layer	<code>window_manager/</code>	Foundation + Processing layers	Layout algorithms, rendering pipeline
Control Layer	<code>input_handler/</code>	All lower layers	Input routing, command processing, terminal control
Application Layer	<code>main.c</code>	All layers	Event loop coordination, program lifecycle

Interface Design Principles

Each module exposes its functionality through header files that define clean interfaces without exposing implementation details. The interfaces use consistent naming conventions and error handling patterns that make the system easy to understand and extend.

PTY Manager Interface (`pty_manager/pty_manager.h`): The PTY Manager interface provides session lifecycle management without exposing file descriptor details or process management complexity to client code.

Function Signature	Purpose	Error Conditions
<pre>tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command)</pre>	Allocate PTY pair and spawn shell process	<code>TMUX_ERROR_PTY_ALLOCATION</code> , <code>TMUX_ERROR_PROCESS_SPAWN</code>
<pre>tmux_error_t pty_session_read(pty_session_t* session, char* buffer, size_t size, ssize_t* bytes_read)</pre>	Non-blocking read from PTY master	<code>TMUX_ERROR_IO</code> , end-of-file conditions
<pre>tmux_error_t pty_session_write(pty_session_t* session, const char* data, size_t length)</pre>	Write data to PTY master	<code>TMUX_ERROR_IO</code> , broken pipe conditions
<pre>bool pty_session_is_alive(const pty_session_t* session)</pre>	Check if child process is still running	Never fails, returns false for terminated processes
<pre>tmux_error_t pty_session_destroy(pty_session_t* session)</pre>	Clean up session resources and terminate child	<code>TMUX_ERROR_CLEANUP</code> if cleanup partially fails

Terminal Emulator Interface (`terminal_emulator/terminal_emulator.h`): The Terminal Emulator interface focuses on transforming character streams into screen buffer updates while hiding the complexity of escape sequence parsing.

Function Signature	Purpose	Error Conditions
<pre>tmux_error_t screen_buffer_init(screen_buffer_t* buffer, int width, int height)</pre>	Initialize screen buffer with dimensions	<code>TMUX_ERROR_MEMORY_ALLOCATION</code>
<pre>tmux_error_t screen_buffer_process_input(screen_buffer_t* buffer, const char* data, size_t length)</pre>	Parse input and update screen buffer	<code>TMUX_ERROR_INVALID_SEQUENCE</code> for malformed escape codes
<pre>tmux_error_t screen_buffer_resize(screen_buffer_t* buffer, int new_width, int new_height)</pre>	Resize buffer and adjust cursor bounds	<code>TMUX_ERROR_MEMORY_ALLOCATION</code>
<pre>tmux_error_t screen_buffer_get_line(const screen_buffer_t* buffer, int line_num, terminal_cell_t** cells, int* length)</pre>	Get rendered line for display	<code>TMUX_ERROR_INVALID_LINE</code> for out-of-bounds access

Design Philosophy: Minimal Surface Area: Each module interface exposes only the functions necessary for other components to use its services. Internal helper functions, data structure manipulation, and implementation details remain private to the module. This approach reduces coupling and makes the system easier to test and modify.

Compilation and Linking Strategy

The modular structure supports efficient development workflows where individual components can be compiled and tested independently. The build system uses static libraries for each major component, enabling parallel compilation and selective rebuilding when source files change.

Build Target	Components	Link Dependencies	Purpose
<code>libpty.a</code>	<code>pty_manager/</code> modules	<code>libc</code>	PTY session management library
<code>libemulator.a</code>	<code>terminal_emulator/</code> modules	<code>libc</code>	Terminal emulation and screen buffer library
<code>libwindow.a</code>	<code>window_manager/</code> modules	<code>libc</code> , <code>libemulator.a</code>	Window layout and rendering library
<code>libinput.a</code>	<code>input_handler/</code> modules	<code>libc</code> , <code>libtermios</code>	Input processing and terminal control library
<code>tmux</code>	<code>main.c</code>	All static libraries	Final executable with event loop

Error Propagation Architecture

The module structure implements consistent error handling that enables graceful failure recovery while providing detailed diagnostic information for debugging. Each module defines its own error codes within the global `tmux_error_t` enumeration, and errors propagate up the call stack with context preservation.

Error Category	Error Codes	Handling Strategy	Recovery Actions
System Resource	<code>TMUX_ERRORPTY_ALLOCATION</code> , <code>TMUX_ERRORMEMORY_ALLOCATION</code>	Immediate failure with cleanup	Attempt graceful shutdown, restore terminal state
Process Management	<code>TMUX_ERRORPROCESS_SPAWN</code> , <code>TMUX_ERRORSIGNAL_SETUP</code>	Session-level failure	Close affected pane, continue with other sessions
I/O Operations	<code>TMUX_ERRORIO</code> , <code>TMUX_ERRORTERMINAL_SETUP</code>	Retry with backoff	Buffer data temporarily, retry operation
Data Validation	<code>TMUX_ERRORINVALID_SEQUENCE</code> , <code>TMUX_ERRORINVALID_LINE</code>	Skip invalid data	Continue processing, log diagnostic information

Error Handling Insight: Terminal multiplexers must be extremely robust because users rely on them for critical shell sessions that shouldn't be lost due to minor errors. The modular error handling allows the system to isolate failures to individual panes or functions while maintaining overall system stability.

Common Pitfalls in Module Organization

⚠ Pitfall: Circular Dependencies Between Components A common mistake is allowing the Terminal Emulator to call Window Manager functions or the PTY Manager to depend on Input Handler state. This creates circular

dependencies that make the code difficult to compile and test. Always design dependencies to flow in one direction: Control Layer → Presentation Layer → Processing Layer → Session Layer → Foundation Layer.

⚠ Pitfall: Exposing Implementation Details Through Headers Another frequent error is putting internal data structures or helper functions in public header files. This creates unnecessary coupling where changes to internal implementation require recompiling all dependent modules. Keep implementation details in `.c` files and only expose the minimal interface needed by other components.

⚠ Pitfall: Inconsistent Error Handling Across Modules Different modules using different error reporting mechanisms (return codes vs. global variables vs. callbacks) makes the system difficult to debug and maintain. Establish consistent error handling patterns early and enforce them across all modules through code review and testing.

⚠ Pitfall: Monolithic Data Structures Creating giant structures that contain fields for multiple components violates encapsulation and makes the code harder to understand. Each module should own its data structures and provide accessor functions for other components that need to interact with that data.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
PTY Management	POSIX <code>posix_openpt()</code> with manual setup	BSD <code>forkpty()</code> with automatic configuration
Terminal Control	Raw <code>tcgetattr()</code> / <code>tcsetattr()</code> calls	<code>ncurses</code> <code>termios</code> wrapper functions
Screen Buffers	Simple 2D character arrays	Memory pools with copy-on-write optimization
Layout Algorithm	Fixed-ratio binary tree splits	Flexible constraint-based layout engine
Event Loop	Basic <code>select()</code> with manual FD management	<code>epoll()</code> or <code>kqueue()</code> for high-performance scaling
Error Handling	Simple return codes with <code>errno</code>	Structured error objects with stack traces

For learning purposes, start with the simple options and upgrade individual components as needed. The simple approaches expose the underlying Unix APIs more directly, which helps build understanding of the core concepts.

Recommended File Structure Implementation

```
// src/tmux_types.h - Core data structure definitions C

#ifndef TMUX_TYPES_H

#define TMUX_TYPES_H


#include <sys/types.h>

#include <termios.h>

#include <stdbool.h>

// Terminal display cell with character and attributes

typedef struct {

    char ch;

    unsigned char attributes; // ATTR_BOLD | ATTR_UNDERLINE | ATTR_REVERSE

    unsigned char fg_color; // COLOR_RED, COLOR_GREEN, etc.

    unsigned char bg_color; // Background color

} terminal_cell_t;

// Screen buffer for terminal emulation

typedef struct {

    terminal_cell_t* cells; // 2D array [height][width]

    int width;

    int height;

    int cursor_x;

    int cursor_y;

    int scroll_offset;

    terminal_cell_t** scrollback; // Array of scrollback lines

    int scrollback_lines;

    unsigned char current_attributes;

    unsigned char current_fg_color;
```

```
    unsigned char current_bg_color;

} screen_buffer_t;

// PTY session for shell process management

typedef struct {

    int master_fd;

    int slave_fd;

    pid_t child_pid;

    bool process_running;

    int exit_status;

} pty_session_t;

// Terminal state for raw mode management

typedef struct {

    struct termios original_termios;

    bool raw_mode_active;

} terminal_state_t;

#endif // TMUX_TYPES_H
```

```
// src/tmux_constants.h - System-wide constants and enums

#ifndef TMUX_CONSTANTS_H
#define TMUX_CONSTANTS_H

// System limits

#define MAX_PANES 16

#define MAX_SCREEN_WIDTH 1024

#define MAX_SCREEN_HEIGHT 256

#define MAX_SCROLLBACK 1000

#define ESC_SEQ_BUFFER_SIZE 64

// Text attributes (bit flags)

#define ATTR_BOLD 0x01

#define ATTR_UNDERLINE 0x02

#define ATTR_REVERSE 0x04

// Standard terminal colors

#define COLOR_BLACK 0

#define COLOR_RED 1

#define COLOR_GREEN 2

#define COLOR_YELLOW 3

#define COLOR_BLUE 4

#define COLOR_MAGENTA 5

#define COLOR_CYAN 6

#define COLOR_WHITE 7

#define COLOR_DEFAULT 9

// Error codes

typedef enum {

    TMUX_OK = 0,
```

```
TMUX_ERROR_PTY_ALLOCATION,  
  
TMUX_ERROR_PROCESS_SPAWN,  
  
TMUX_ERROR_TERMINAL_SETUP,  
  
TMUX_ERROR_MEMORY_ALLOCATION,  
  
TMUX_ERROR_IO,  
  
TMUX_ERROR_INVALID_SEQUENCE,  
  
TMUX_ERROR_INVALID_LINE,  
  
TMUX_ERROR_CLEANUP  
  
} tmux_error_t;  
  
#endif // TMUX_CONSTANTS_H
```

```
// src/main.c - Program entry point and main event loop (skeleton)
```

C

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/select.h>

#include <signal.h>

#include "tmux_types.h"

#include "tmux_constants.h"

#include "pty_manager/pty_manager.h"

#include "terminal_emulator/terminal_emulator.h"

#include "window_manager/window_manager.h"

#include "input_handler/input_handler.h"
```

```
// Global state for signal handlers
```

```
static bool program_running = true;

static terminal_state_t terminal_state = {0};
```

```
// Signal handler for clean shutdown
```

```
void handle_shutdown_signal(int sig) {

    program_running = false;

}
```

```
int main(int argc, char* argv[]) {
```

```
    // TODO 1: Install signal handlers for SIGTERM, SIGINT
```

```
    // TODO 2: Save original terminal state and enter raw mode
```

```
    // TODO 3: Initialize all components (PTY manager, terminal emulator, etc.)
```

```
    // TODO 4: Create initial pane with shell session
```

```
    // TODO 5: Enter main event loop with select() for I/O multiplexing
```

```
    // TODO 6: Handle PTY output, user input, and window resize events
```

```
// TODO 7: On shutdown, cleanup all sessions and restore terminal state

return 0;

}

// Main event loop processing all I/O sources

int run_event_loop() {

    fd_set read_fds;

    int max_fd = 0;

    char buffer[1024];



    while (program_running) {

        // TODO 1: Clear and set up file descriptor set for select()

        // TODO 2: Add stdin (user input) and all PTY master FDs to read set

        // TODO 3: Call select() with appropriate timeout for status updates

        // TODO 4: Process user input if stdin is ready

        // TODO 5: Process PTY output for any ready PTY sessions

        // TODO 6: Update display if any screen buffers changed

        // TODO 7: Handle any expired timers or periodic tasks

    }

    return TMUX_OK;
}
```

Component Interface Skeletons

```
// src/pty_manager/pty_manager.h - PTY management interface C

#ifndef PTY_MANAGER_H

#define PTY_MANAGER_H

#include "../tmux_types.h"
#include "../tmux_constants.h"

// Create new PTY session with shell process

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command);

// Read data from PTY master (non-blocking)

tmux_error_t pty_session_read(pty_session_t* session, char* buffer, size_t size, ssize_t* bytes_read);

// Write data to PTY master

tmux_error_t pty_session_write(pty_session_t* session, const char* data, size_t length);

// Check if child process is still running

bool pty_session_is_alive(const pty_session_t* session);

// Clean up session resources

tmux_error_t pty_session_destroy(pty_session_t* session);

// Get current terminal window size

tmux_error_t terminal_get_size(int* width, int* height);

#endif // PTY_MANAGER_H
```

```
// src/terminal_emulator/terminal_emulator.h - Terminal emulation interface C

#ifndef TERMINAL_EMULATOR_H

#define TERMINAL_EMULATOR_H

#include "../tmux_types.h"
#include "../tmux_constants.h"

// Initialize screen buffer with given dimensions

tmux_error_t screen_buffer_init(screen_buffer_t* buffer, int width, int height);

// Process input data and update screen buffer

tmux_error_t screen_buffer_process_input(screen_buffer_t* buffer, const char* data, size_t length);

// Resize screen buffer to new dimensions

tmux_error_t screen_buffer_resize(screen_buffer_t* buffer, int new_width, int new_height);

// Get rendered line for display

tmux_error_t screen_buffer_get_line(const screen_buffer_t* buffer, int line_num,
                                    terminal_cell_t** cells, int* length);

// Clean up screen buffer resources

void screen_buffer_destroy(screen_buffer_t* buffer);

#endif // TERMINAL_EMULATOR_H
```

```
// src/input_handler/input_handler.h - Input processing interface

#ifndef INPUT_HANDLER_H

#define INPUT_HANDLER_H

#include "../tmux_types.h"
#include "../tmux_constants.h"

// Enter raw terminal mode, saving original state

tmux_error_t terminal_enter_raw_mode(terminal_state_t* state);

// Restore original terminal mode

tmux_error_t terminal_restore_mode(terminal_state_t* state);

// Process keyboard input and route to appropriate handler

tmux_error_t process_keyboard_input(const char* input, size_t length);

// Initialize key binding registry

tmux_error_t initialize_key_bindings(void);

#endif // INPUT_HANDLER_H
```

C

Core Implementation Skeletons

```
// src/pty_manager/pty_manager.c - PTY session management (skeleton) C

#include "pty_manager.h"

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <sys/wait.h>

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command) {

    // TODO 1: Call posix_openpt(O_RDWR) to allocate PTY master

    // TODO 2: Call grantpt() and unlockpt() to set up PTY permissions

    // TODO 3: Get slave PTY name with ptsname() for child process

    // TODO 4: Fork child process for shell execution

    // TODO 5: In child: call setsid() to become session leader

    // TODO 6: In child: open slave PTY and set as stdin/stdout/stderr

    // TODO 7: In child: exec shell command (typically "/bin/bash")

    // TODO 8: In parent: store master FD and child PID in session struct

    // TODO 9: Set master FD to non-blocking mode with fcntl()

    // Hint: Always check return values - PTY allocation can fail!

    return TMUX_ERROR_PTY_ALLOCATION; // Replace with actual implementation

}

bool pty_session_is_alive(const pty_session_t* session) {

    // TODO 1: Check if session pointer and child_pid are valid

    // TODO 2: Call waitpid(child_pid, NULL, WNOHANG) to check process state

    // TODO 3: Return true if waitpid returns 0 (process still running)

    // TODO 4: Return false if waitpid returns child_pid (process exited)

    // Hint: WNOHANG prevents blocking if child is still running
```

```
    return false; // Replace with actual implementation
```

```
}
```

```
// src/terminal_emulator/screen_buffer.c - Screen buffer management (skeleton) C

#include "../terminal_emulator/terminal_emulator.h"

#include <stdlib.h>

#include <string.h>

tmux_error_t screen_buffer_init(screen_buffer_t* buffer, int width, int height) {

    // TODO 1: Validate width and height against MAX_SCREEN_WIDTH/HEIGHT

    // TODO 2: Allocate cells array as width * height * sizeof(terminal_cell_t)

    // TODO 3: Initialize all cells with space character and default attributes

    // TODO 4: Set cursor_x = 0, cursor_y = 0, scroll_offset = 0

    // TODO 5: Allocate scrollback array with MAX_SCROLLBACK entries

    // TODO 6: Set current_attributes = 0, current_fg_color = COLOR_DEFAULT

    // Hint: Use calloc() to zero-initialize the cells array

    return TMUX_ERROR_MEMORY_ALLOCATION; // Replace with actual implementation

}
```

```
tmux_error_t screen_buffer_process_input(screen_buffer_t* buffer, const char* data, size_t length) {

    // TODO 1: Iterate through each byte in the input data

    // TODO 2: Check if byte is printable character (0x20-0x7E)

    // TODO 3: For printable chars: add to screen at cursor position, advance cursor

    // TODO 4: For control characters: check if start of escape sequence (ESC = 0x1B)

    // TODO 5: Parse escape sequences using state machine (normal → escape → CSI → command)

    // TODO 6: Handle cursor movement commands (CSI A/B/C/D for up/down/right/left)

    // TODO 7: Handle screen clearing commands (CSI 2J = clear screen)

    // TODO 8: Handle text attribute changes (CSI 1m = bold, CSI 0m = reset)

    // Hint: Escape sequences can span multiple input chunks - maintain parser state

    return TMUX_OK; // Replace with actual implementation

}
```

Language-Specific Implementation Hints

PTY Management in C:

- Use `posix_openpty(0_RDWR)` for portable PTY allocation across Unix systems
- Always call `grantpt()` and `unlockpt()` after `posix_openpty()` - these set proper permissions
- Use `ptsname()` to get the slave PTY device name for the child process
- Set master FD to non-blocking with `fcntl(fd, F_SETFL, O_NONBLOCK)`
- Handle `EAGAIN` / `EWOULDBLOCK` errors when reading from non-blocking PTY master

Terminal Control in C:

- Save original terminal state with `tcgetattr(STDIN_FILENO, &original_termios)`
- Enter raw mode by copying original state and modifying flags: `cfmakeraw(&raw_termios)`
- Use `atexit()` to register cleanup function that calls `tcsetattr()` to restore terminal
- Handle `SIGWINCH` signal to detect terminal resize events
- Use `ioctl(fd, TIOCGWINSZ, &winsize)` to get current terminal dimensions

Memory Management:

- Use memory pools for screen buffers to avoid fragmentation from frequent resize operations
- Implement reference counting for scrollback lines that may be shared between panes
- Use `mmap()` for large scrollback buffers that can be swapped to disk if memory is limited
- Always pair `malloc()` / `free()` calls and check for allocation failures

Signal Handling:

- Install handlers for `SIGCHLD` to detect when child shell processes exit
- Handle `SIGWINCH` for terminal resize events that need to propagate to PTY slaves
- Use `signalfd()` or `self-pipe trick` to integrate signals into `select()` event loop
- Block signals during critical sections like PTY allocation to prevent race conditions

Milestone Checkpoints

Milestone 1 Checkpoint (PTY Creation): After implementing the PTY Manager, verify functionality with this test:

```
# Compile just the PTY manager components

make libpty.a

# Run unit tests

./tests/unit/test_pty_manager

# Expected behavior:

# - Creates PTY session successfully

# - Forks shell process that responds to input

# - Can write "echo hello\n" and read back "hello" from PTY

# - Properly cleans up child process on session destroy
```

BASH

Milestone 2 Checkpoint (Terminal Emulation): Test escape sequence parsing with known sequences:

```
# Send test escape sequences to screen buffer

echo -e "\033[2J\033[H\033[1mBold Text\033[0m" | ./test_emulator

# Expected behavior:

# - Screen buffer clears (all cells become spaces)

# - Cursor moves to home position (0,0)

# - Text "Bold Text" appears with ATTR_BOLD flag set

# - Subsequent text has normal attributes after reset sequence
```

BASH

Milestone 3 Checkpoint (Window Management): Verify pane splitting and layout algorithms:

```
# Test pane splitting functionality
./tmux_test -split-vertical -split-horizontal
```

BASH

```
# Expected behavior:
# - Creates 3 panes in tree layout
# - Each pane has independent shell session
# - Border characters drawn between panes
# - Focus switching changes active pane highlighting
```

Milestone 4 Checkpoint (Key Bindings): Test complete multiplexer functionality:

```
# Start multiplexer and test key bindings
./tmux

# In the multiplexer, test:
# Ctrl-b c → creates new pane
# Ctrl-b % → splits current pane vertically
# Ctrl-b h/j/k/l → switches focus between panes
# All other keys → forwarded to active shell
```

BASH

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
PTY allocation fails	Insufficient permissions or system limits	Check <code>ulimit -n</code> , run with strace	Increase file descriptor limits, check <code>/dev/pts</code> mount
Child process becomes zombie	Not calling <code>waitpid()</code> for terminated children	Check process table with <code>ps aux</code>	Install <code>SIGCHLD</code> handler, call <code>waitpid()</code> with <code>WNOHANG</code>
Terminal corrupted on exit	Raw mode not restored properly	Terminal shows strange characters	Use <code>atexit()</code> handler, always call <code>tcsetattr()</code> in cleanup
Escape sequences not parsed	State machine stuck in wrong state	Log parser state transitions	Reset parser state on timeout, handle incomplete sequences
Pane borders misaligned	Layout calculation error	Print pane dimensions to debug log	Verify width/height arithmetic, check for integer overflow
Input not reaching shell	Raw mode capturing all keystrokes	Shell doesn't respond to commands	Check prefix key detection, verify PTY write operations

Data Model

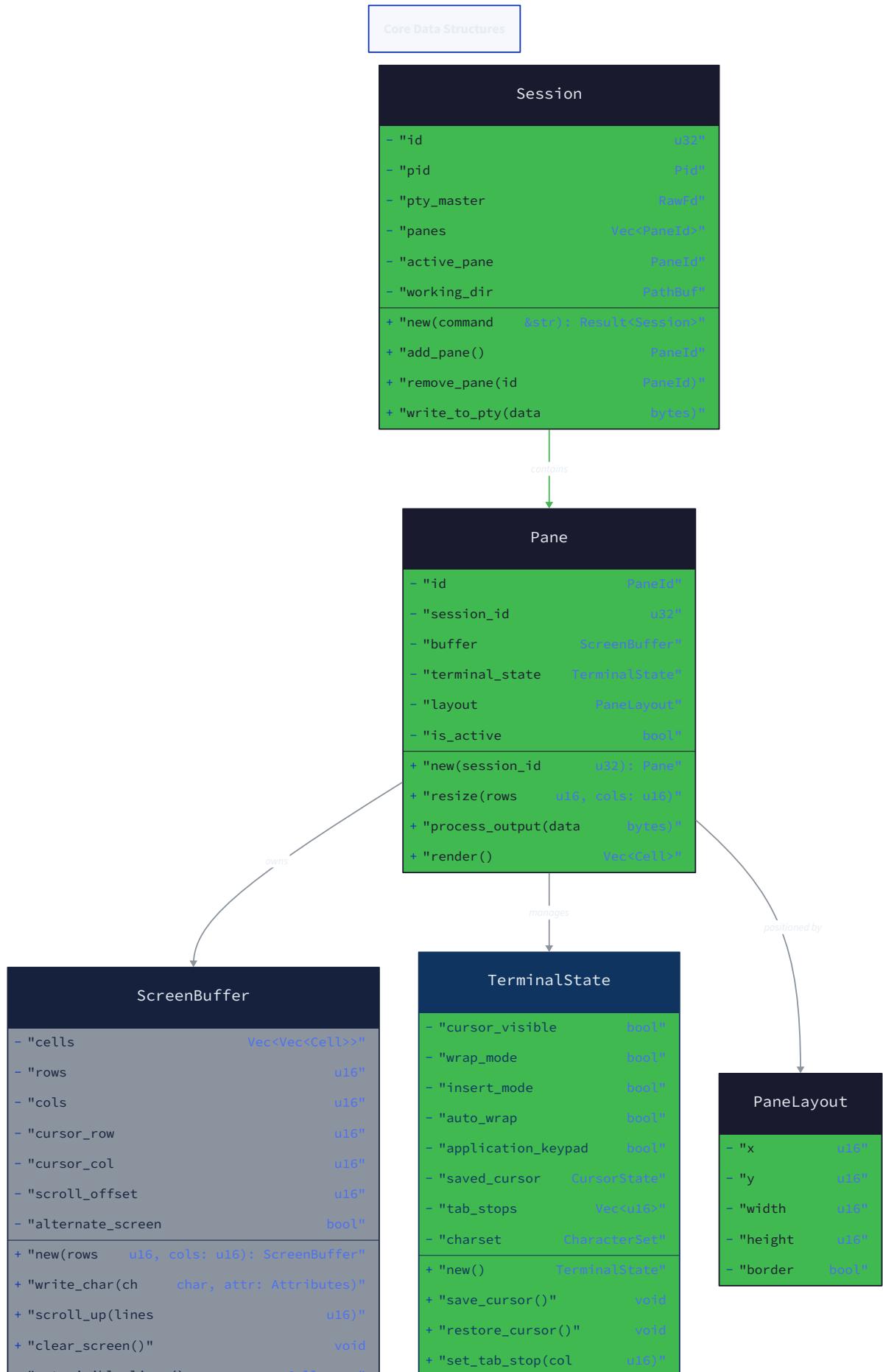
Milestone(s): All milestones (1-4) - this section defines the core data structures that underpin PTY management (M1), terminal emulation (M2), window management (M3), and input handling (M4)

Mental Model: The Information Warehouse

Think of the terminal multiplexer's data model as a well-organized warehouse that stores different types of information in specialized containers. Just as a warehouse has sections for raw materials, work-in-progress items, and finished products, our multiplexer organizes data into distinct categories: session information (tracking active shell processes), visual state (what appears on screen), layout information (how panes are arranged), and system state (terminal configuration).

Each "container" in our warehouse serves a specific purpose and has a defined structure. The **session containers** hold everything needed to communicate with shell processes - file descriptors, process IDs, and status information. The **visual containers** store the current appearance of each terminal screen - every character, color, and cursor position. The **layout containers** define how multiple sessions share screen real estate. Finally, the **system containers** preserve the original terminal configuration so we can restore it when exiting.

The key insight is that these containers must work together seamlessly. When a shell process writes output, it flows from session containers through parsing logic into visual containers, which then get composited by layout containers for display. This separation of concerns allows each component to focus on its specific responsibility while maintaining clear interfaces between them.





Session and Pane Structures

The session and pane structures form the foundation for managing multiple shell processes and organizing them visually. These structures bridge the gap between the low-level pseudo-terminal interfaces and the higher-level windowing abstractions that users interact with.

PTY Session Structure

The `pty_session_t` structure encapsulates everything needed to manage a single shell session running in a pseudo-terminal. This structure serves as the primary interface between the multiplexer and individual shell processes.

Field Name	Type	Description
master_fd	int	File descriptor for PTY master device, used for reading shell output and writing user input
slave_fd	int	File descriptor for PTY slave device, connected to child process stdin/stdout/stderr
child_pid	pid_t	Process ID of the shell process running in this session
process_running	bool	Flag indicating whether child process is still alive and responsive
exit_status	int	Exit code from child process after termination (valid only when process_running is false)

The master file descriptor serves as the multiplexer's primary interface to the shell process. All data written to the master appears on the slave's stdout, and all data written to the slave's stdin appears when reading from the master. This bidirectional pipe abstraction allows the multiplexer to intercept and process all terminal I/O.

The slave file descriptor is typically only used during session creation to configure terminal attributes and establish the controlling terminal relationship. Once the child process starts, the multiplexer closes its copy of the slave descriptor, leaving only the child process with access.

Design Insight: The `process_running` flag provides a fast path for checking process liveness without making system calls. It's updated asynchronously when SIGCHLD signals arrive, avoiding the need to call `waitpid()` on every I/O operation.

Pane Structure

Panes represent the visual organization of sessions within the terminal window. Each pane contains a session, screen buffer, and layout information that determines how it appears to the user.

Field Name	Type	Description
<code>id</code>	<code>int</code>	Unique identifier for this pane within the current window
<code>session</code>	<code>pty_session_t</code>	PTY session running in this pane
<code>buffer</code>	<code>screen_buffer_t</code>	Virtual screen buffer storing current display state
<code>x</code>	<code>int</code>	Left column position of pane within terminal (0-based)
<code>y</code>	<code>int</code>	Top row position of pane within terminal (0-based)
<code>width</code>	<code>int</code>	Width of pane content area in characters (excluding borders)
<code>height</code>	<code>int</code>	Height of pane content area in characters (excluding borders)
<code>has_focus</code>	<code>bool</code>	Whether this pane currently receives keyboard input
<code>needs_redraw</code>	<code>bool</code>	Flag indicating pane content has changed and requires re-rendering
<code>border_style</code>	<code>int</code>	Border appearance (none, simple, double-line)

The coordinate system uses zero-based indexing with the origin at the top-left corner of the terminal. The `width` and `height` fields represent the usable content area, excluding any border characters. This separation simplifies the rendering logic by providing consistent dimensions regardless of border style.

The `needs_redraw` flag implements a simple optimization to avoid unnecessary rendering. Components set this flag when modifying pane content, and the renderer clears it after updating the display. This prevents wasted CPU cycles re-rendering unchanged panes.

Design Insight: Embedding the session directly in the pane structure (rather than using a pointer or ID reference) simplifies memory management and ensures session lifetime matches pane lifetime. This prevents dangling pointer bugs that could occur if sessions were managed separately.

Window Structure

Windows provide the top-level container for organizing multiple panes. Each window maintains its own pane layout and can be switched independently.

Field Name	Type	Description
<code>id</code>	<code>int</code>	Unique identifier for this window
<code>name</code>	<code>char[64]</code>	Human-readable name displayed in status bar
<code>panes</code>	<code>pane_t[MAX_PANES]</code>	Array of panes within this window
<code>pane_count</code>	<code>int</code>	Number of active panes in the panes array
<code>active_pane</code>	<code>int</code>	Index of currently focused pane in panes array
<code>layout_tree</code>	<code>layout_node_t*</code>	Root of binary tree describing pane split structure
<code>last_activity</code>	<code>time_t</code>	Timestamp of most recent activity in any pane

The panes array uses a fixed-size allocation to simplify memory management and prevent fragmentation. The `MAX_PANES` constant (16) provides reasonable flexibility while avoiding the complexity of dynamic allocation.

The layout tree represents the hierarchical structure of pane splits. Each internal node represents a split (horizontal or vertical), while leaf nodes represent individual panes. This tree structure enables efficient resize calculations and maintains split proportions when the terminal size changes.

Layout Node Structure

Layout nodes form a binary tree that describes how panes are arranged within a window. This tree-based approach provides flexible layout capabilities while maintaining efficient resize and rendering operations.

Field Name	Type	Description
<code>type</code>	<code>layout_type_t</code>	Node type: <code>LAYOUT_LEAF</code> (pane), <code>LAYOUT_HSPLIT</code> (horizontal), or <code>LAYOUT_VSPLIT</code> (vertical)
<code>pane_id</code>	<code>int</code>	Pane identifier (valid only for leaf nodes)
<code>split_ratio</code>	<code>float</code>	Proportion of space allocated to left/top child (0.0 to 1.0)
<code>left</code>	<code>layout_node_t*</code>	Left or top child node (NULL for leaf nodes)
<code>right</code>	<code>layout_node_t*</code>	Right or bottom child node (NULL for leaf nodes)
<code>parent</code>	<code>layout_node_t*</code>	Parent node (NULL for root)

The split ratio determines how available space is divided between child nodes. A ratio of 0.3 allocates 30% of space to the left/top child and 70% to the right/bottom child. This proportional system maintains layout balance when the terminal is resized.

Parent pointers enable efficient upward traversal during resize operations. When a pane requests a size change, the algorithm walks up the tree to find the appropriate split node and adjusts ratios accordingly.

Terminal State Structures

Terminal state structures maintain the visual representation of shell output and the system-level terminal configuration. These structures bridge the gap between raw character streams from shell processes and the formatted display that users see.

Screen Buffer Structure

The `screen_buffer_t` structure represents the complete visual state of a single terminal session. It maintains both the currently visible screen and the scrollback history that users can navigate.

Field Name	Type	Description
<code>cells</code>	<code>terminal_cell_t**</code>	2D array of screen cells, indexed as <code>cells[row][column]</code>
<code>width</code>	<code>int</code>	Screen buffer width in characters (matches pane width)
<code>height</code>	<code>int</code>	Screen buffer height in characters (matches pane height)
<code>cursor_x</code>	<code>int</code>	Current cursor column position (0-based)
<code>cursor_y</code>	<code>int</code>	Current cursor row position (0-based)
<code>scroll_offset</code>	<code>int</code>	Number of lines scrolled up from bottom of scrollback
<code>scrollback</code>	<code>terminal_cell_t**</code>	Scrollback buffer storing lines that scrolled off screen
<code>scrollback_lines</code>	<code>int</code>	Current number of lines stored in scrollback buffer
<code>current_attributes</code>	<code>byte</code>	Active text attributes (bold, underline, reverse)
<code>current_fg_color</code>	<code>byte</code>	Active foreground color (0-15 for standard colors)
<code>current_bg_color</code>	<code>byte</code>	Active background color (0-15 for standard colors)

The `cells` array provides random access to any screen position for efficient cursor movement and text rendering. The two-dimensional indexing simplifies coordinate calculations and bounds checking during escape sequence processing.

The scrollback buffer operates as a circular buffer that preserves lines as they scroll off the top of the visible screen. The `scroll_offset` field allows users to navigate this history without losing the current screen state. When `scroll_offset` is zero, the user sees the live terminal output. Positive values show historical lines.

Current attribute fields maintain the terminal's text formatting state. These values apply to all subsequently written characters until modified by escape sequences. Separating these from individual cell attributes enables efficient processing of long runs of similarly formatted text.

Design Insight: Using a 2D array rather than a single linear buffer simplifies indexing calculations and reduces the likelihood of buffer overflow bugs. The memory overhead is minimal compared to the safety benefits.

Terminal Cell Structure

The `terminal_cell_t` structure represents a single character position on the terminal screen, including the character itself and all associated display attributes.

Field Name	Type	Description
<code>ch</code>	<code>char</code>	Unicode character at this cell position (UTF-8 encoded)
<code>attributes</code>	<code>byte</code>	Bitmask of text attributes (bold, underline, reverse video)
<code>fg_color</code>	<code>byte</code>	Foreground color index (0-255 for extended color support)
<code>bg_color</code>	<code>byte</code>	Background color index (0-255 for extended color support)

The character field stores a single UTF-8 code unit. For multi-byte Unicode characters, the first cell contains the complete UTF-8 sequence while subsequent cells in the character are marked with a special continuation value. This approach maintains simple indexing while supporting international characters.

The attributes field uses individual bits to represent different text formatting options:

Attribute	Bit Position	Description
<code>ATTR_BOLD</code>	0	Bold or bright text rendering
<code>ATTR_UNDERLINE</code>	1	Underlined text decoration
<code>ATTR_REVERSE</code>	2	Swap foreground and background colors

Color fields support both the standard 16-color palette and extended 256-color mode. Standard colors use values 0-15, while extended colors utilize the full 0-255 range. The special value `COLOR_DEFAULT` (9) indicates the terminal's default color scheme should be used.

Terminal State Structure

The `terminal_state_t` structure preserves the original terminal configuration to enable proper restoration when the multiplexer exits. Raw terminal mode significantly alters the terminal's behavior, making restoration critical for system stability.

Field Name	Type	Description
<code>original_termios</code>	<code>struct termios</code>	Terminal attributes before entering raw mode
<code>raw_mode_active</code>	<code>bool</code>	Flag indicating whether raw mode is currently enabled

The `termios` structure contains all terminal configuration parameters, including input processing flags, output processing flags, control character definitions, and line discipline settings. Preserving this complete state ensures perfect restoration regardless of the user's custom terminal configuration.

The `raw_mode_active` flag prevents double-initialization and ensures proper cleanup. Attempting to enter raw mode when already active, or restore mode when not active, can cause undefined behavior or system instability.

Data Structure Relationships

Understanding how these structures interconnect is crucial for implementing the multiplexer correctly. The relationships follow a clear hierarchy that mirrors the user's conceptual model of the system.

Ownership and Lifetime Management

Windows own panes, panes own sessions and screen buffers, and sessions own PTY file descriptors. This ownership hierarchy ensures proper resource cleanup and prevents memory leaks:

1. **Window Lifetime:** Created when user creates new window, destroyed when user closes window or multiplexer exits
2. **Pane Lifetime:** Created when splitting existing pane or creating first pane in window, destroyed when user closes pane
3. **Session Lifetime:** Created with pane, destroyed when shell process exits or pane is closed
4. **Buffer Lifetime:** Matches pane lifetime, allocated during pane creation, freed during pane cleanup

Data Flow Patterns

Data flows through these structures in predictable patterns that correspond to the main multiplexer operations:

Shell Output Processing:

1. Data arrives on PTY master file descriptor in `pty_session_t`
2. Escape sequence parser processes raw bytes and updates `screen_buffer_t`
3. Window manager composites multiple buffers for display
4. Terminal output renderer converts screen buffers to terminal escape sequences

User Input Processing:

1. Raw keyboard input arrives from terminal
2. Input handler checks for prefix key combinations and command mode
3. Regular input routes to active pane's `pty_session_t`
4. Data writes to PTY master file descriptor reach shell process

Window Management Operations:

1. Split commands create new `pane_t` and `pty_session_t` structures
2. Layout tree updates with new `layout_node_t` reflecting split
3. Existing panes resize to accommodate new pane

4. Screen buffers resize to match new pane dimensions

Architecture Decision: Embedded vs Referenced Structures

- **Context:** Structures can either embed related structures directly or reference them through pointers/IDs
- **Options Considered:**
 - Direct embedding (session embedded in pane)
 - Pointer references (pane points to session)
 - ID-based lookup (pane stores session ID)
- **Decision:** Use direct embedding for session-in-pane, pointer references for layout tree
- **Rationale:** Embedding simplifies lifetime management and prevents null pointer dereferences. Pointers used only where tree structure requires it.
- **Consequences:** Slightly higher memory usage, but significantly reduced complexity and improved safety

Memory Layout Considerations

The structures are designed to minimize memory fragmentation and cache misses during common operations. Related data is co-located, and frequently accessed fields are positioned early in structures to improve cache performance.

Fixed-size arrays (like the panes array in windows) prevent heap fragmentation that could occur with dynamic allocation. The `MAX_PANES` limit is generous enough for typical usage while keeping memory usage predictable.

Circular buffer management for scrollback uses efficient modular arithmetic rather than shifting array contents. This provides $O(1)$ insertion performance regardless of scrollback buffer size.

Architecture Decision Records

Decision: Fixed vs Dynamic Array Sizes

- **Context:** Collections like panes within windows could use fixed arrays or dynamic allocation
- **Options Considered:**
 - Fixed arrays with compile-time maximums
 - Dynamic arrays with reallocation
 - Linked lists with individual node allocation
- **Decision:** Fixed arrays with reasonable limits (`MAX_PANES` = 16)
- **Rationale:** Eliminates allocation failures, prevents fragmentation, simplifies memory management, provides predictable performance
- **Consequences:** Memory usage slightly higher, but eliminates entire classes of runtime errors and complexity

Decision: Screen Buffer Organization

- **Context:** Screen buffers could be organized as 2D arrays, linear arrays, or sparse data structures
- **Options Considered:**
 - 2D array indexed as `buffer[row][column]`
 - Linear array with calculated offsets
 - Sparse representation storing only non-blank cells
- **Decision:** 2D array with direct indexing
- **Rationale:** Simplifies coordinate calculations, eliminates offset bugs, provides $O(1)$ random access, matches natural mental model
- **Consequences:** Uses more memory for sparse screens, but provides simpler and more reliable implementation

Decision: UTF-8 Character Handling

- **Context:** Unicode characters can span multiple bytes, requiring special handling in terminal cells
- **Options Considered:**
 - Store complete UTF-8 sequences in each cell
 - Use wide characters (`wchar_t`) for internal representation
 - Mark continuation cells with special values
- **Decision:** Store UTF-8 in first cell, mark continuations with special marker
- **Rationale:** Maintains byte-oriented processing, supports full Unicode range, preserves simple indexing model
- **Consequences:** Requires special handling during cursor movement and editing operations

Common Pitfalls

⚠ **Pitfall: Forgetting to Initialize Screen Buffer Cells** When allocating a new screen buffer, uninitialized cells may contain random memory values that appear as garbage characters on screen. Always initialize cells with space characters, default colors, and no attributes. Use `memset()` or explicit loops to ensure clean initialization.

⚠ **Pitfall: Buffer Overflow During Screen Resize** When terminal dimensions change, existing screen buffers may be smaller than the new size. Accessing cells beyond the old dimensions causes buffer overflows. Always reallocate and initialize screen buffers before updating width/height fields.

⚠ **Pitfall: Inconsistent Cursor Position Tracking** The cursor position in the screen buffer must stay synchronized with the PTY's actual cursor position. Escape sequences that move the cursor must update both the internal tracking variables and send appropriate sequences to the PTY. Mismatches cause visual artifacts and incorrect text placement.

⚠ **Pitfall: Memory Leaks in Layout Tree Management** Layout nodes are dynamically allocated and form a tree structure with complex ownership relationships. Failing to properly traverse and free the entire tree during cleanup

causes memory leaks. Implement recursive cleanup functions that handle both leaf and internal nodes correctly.

⚠ Pitfall: Race Conditions in Process Status Updates The `process_running` flag is updated asynchronously by signal handlers when child processes exit. Reading this flag without proper synchronization can lead to race conditions where the multiplexer thinks a process is still running after it has exited. Use atomic operations or proper locking when accessing process status.

⚠ Pitfall: Scrollback Buffer Circular Index Errors Circular buffers require careful index management to wrap correctly at buffer boundaries. Off-by-one errors in modular arithmetic cause buffer overruns or incorrect data retrieval. Always use consistent wrap-around logic and validate indices before array access.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Memory Management	<code>malloc/free</code> with manual tracking	Custom memory pools with leak detection
String Handling	Fixed-size character arrays	Dynamic strings with UTF-8 validation
Data Serialization	Binary struct copying	Protocol buffers or custom serialization
Error Handling	Return codes with <code>errno</code>	Structured error types with context

Recommended File Structure

```
terminal-multiplexer/
  src/
    data_model/
      session.h           ← PTY session structure and functions
      session.c
      screen_buffer.h     ← Screen buffer and terminal cell structures
      screen_buffer.c
      pane.h              ← Pane and window structures
      pane.c
      terminal_state.h    ← Terminal configuration management
      terminal_state.c
    common/
      constants.h         ← MAX_PANES, color constants, etc.
      error_codes.h       ← tmux_error_t enumeration
  tests/
    test_data_model.c    ← Unit tests for all structures
```

Core Data Structure Implementation

Complete Screen Buffer Infrastructure (Ready to Use):

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <stdbool.h>
```

```
#include <sys/ioctl.h>
```

```
#include <termios.h>
```

```
#include <unistd.h>
```

```
// Constants
```

```
#define MAX_PANES 16
```

```
#define MAX_SCREEN_WIDTH 1024
```

```
#define MAX_SCREEN_HEIGHT 256
```

```
#define MAX_SCROLLBACK 1000
```

```
#define ESC_SEQ_BUFFER_SIZE 64
```

```
// Attribute bit flags
```

```
#define ATTR_BOLD (1 << 0)
```

```
#define ATTR_UNDERLINE (1 << 1)
```

```
#define ATTR_REVERSE (1 << 2)
```

```
// Standard color constants
```

```
#define COLOR_BLACK 0
```

```
#define COLOR_RED 1
```

```
#define COLOR_GREEN 2
```

```
#define COLOR_YELLOW 3
```

```
#define COLOR_BLUE 4
```

```
#define COLOR_MAGENTA 5
```

```
#define COLOR_CYAN 6
```

```
#define COLOR_WHITE 7
```

```
#define COLOR_DEFAULT 9
```

C

```
// Error codes

typedef enum {

    TMUX_OK = 0,
    TMUX_ERROR_PTY_ALLOCATION = -1,
    TMUX_ERROR_PROCESS_SPAWN = -2,
    TMUX_ERROR_TERMINAL_SETUP = -3,
    TMUX_ERROR_MEMORY_ALLOCATION = -4
} tmux_error_t;

// Complete terminal cell structure

typedef struct {

    char ch;
    unsigned char attributes;
    unsigned char fg_color;
    unsigned char bg_color;
} terminal_cell_t;

// Complete screen buffer structure

typedef struct {

    terminal_cell_t **cells;
    int width;
    int height;
    int cursor_x;
    int cursor_y;
    int scroll_offset;
    terminal_cell_t **scrollback;
    int scrollback_lines;
    unsigned char current_attributes;
}
```

```
    unsigned char current_fg_color;

    unsigned char current_bg_color;

} screen_buffer_t;

// Complete PTY session structure

typedef struct {

    int master_fd;

    int slave_fd;

    pid_t child_pid;

    bool process_running;

    int exit_status;

} pty_session_t;

// Complete terminal state structure

typedef struct {

    struct termios original_termios;

    bool raw_mode_active;

} terminal_state_t;

// Screen buffer initialization (COMPLETE implementation)

tmux_error_t screen_buffer_init(screen_buffer_t *buffer, int width, int height) {

    if (!buffer || width <= 0 || height <= 0 ||
        width > MAX_SCREEN_WIDTH || height > MAX_SCREEN_HEIGHT) {

        return TMUX_ERROR_MEMORY_ALLOCATION;
    }

    // Allocate row pointers

    buffer->cells = calloc(height, sizeof(terminal_cell_t*));

    if (!buffer->cells) {
```

```
    return TMUX_ERROR_MEMORY_ALLOCATION;

}

// Allocate each row

for (int row = 0; row < height; row++) {
    buffer->cells[row] = calloc(width, sizeof(terminal_cell_t));

    if (!buffer->cells[row]) {
        // Clean up partial allocation

        for (int i = 0; i < row; i++) {
            free(buffer->cells[i]);
        }

        free(buffer->cells);
        return TMUX_ERROR_MEMORY_ALLOCATION;
    }
}

// Initialize cells with defaults

for (int col = 0; col < width; col++) {
    buffer->cells[row][col].ch = ' ';
    buffer->cells[row][col].attributes = 0;
    buffer->cells[row][col].fg_color = COLOR_DEFAULT;
    buffer->cells[row][col].bg_color = COLOR_DEFAULT;
}

}

// Allocate scrollback buffer

buffer->scrollback = calloc(MAX_SCROLLBACK, sizeof(terminal_cell_t*));

if (!buffer->scrollback) {
```

```
// Clean up screen buffer

for (int row = 0; row < height; row++) {
    free(buffer->cells[row]);
}

free(buffer->cells);

return TMUX_ERROR_MEMORY_ALLOCATION;
}

// Initialize scrollback rows to NULL (allocated on demand)

// Set initial state

buffer->width = width;

buffer->height = height;

buffer->cursor_x = 0;

buffer->cursor_y = 0;

buffer->scroll_offset = 0;

buffer->scrollback_lines = 0;

buffer->current_attributes = 0;

buffer->current_fg_color = COLOR_DEFAULT;

buffer->current_bg_color = COLOR_DEFAULT;

return TMUX_OK;
}

// Terminal mode management (COMPLETE implementation)

tmux_error_t terminal_enter_raw_mode(terminal_state_t *state) {
    if (!state) return TMUX_ERROR_TERMINAL_SETUP;
```

```
if (state->raw_mode_active) return TMUX_OK; // Already in raw mode

// Save original terminal attributes

if (tcgetattr(STDIN_FILENO, &state->original_termios) == -1) {

    return TMUX_ERROR_TERMINAL_SETUP;
}

struct termios raw = state->original_termios;

// Disable canonical mode and echo

raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

// Disable input processing

raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

// Disable output processing

raw.c_oflag &= ~(OPOST);

// Set 8-bit chars

raw.c_cflag |= (CS8);

// Set read timeout

raw.c_cc[VMIN] = 0; // Non-blocking reads

raw.c_cc[VTIME] = 1; // 100ms timeout


if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) {

    return TMUX_ERROR_TERMINAL_SETUP;
}

state->raw_mode_active = true;
```

```

    return TMUX_OK;
}

tmux_error_t terminal_restore_mode(terminal_state_t *state) {

    if (!state || !state->raw_mode_active) return TMUX_OK;

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &state->original_termios) == -1) {

        return TMUX_ERROR_TERMINAL_SETUP;
    }

    state->raw_mode_active = false;

    return TMUX_OK;
}

// Terminal size detection (COMPLETE implementation)

tmux_error_t terminal_get_size(int *width, int *height) {

    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1) {

        return TMUX_ERROR_TERMINAL_SETUP;
    }

    *width = ws.ws_col;

    *height = ws.ws_row;

    return TMUX_OK;
}

```

Core Logic Skeleton (For Student Implementation)

PTY Session Management (Skeleton with TODOs):

```
// Create new PTY session with shell process

// This is the core of Milestone 1 - students implement this

tmux_error_t pty_session_create(pty_session_t *session, const char *shell_command) {

    // TODO 1: Call posix_openpty(0_RDWR) to allocate PTY master

    // TODO 2: Call grantpt() and unlockpt() on master FD

    // TODO 3: Get slave device name with ptsname()

    // TODO 4: Fork child process with fork()

    // TODO 5: In child: call setsid() to become session leader

    // TODO 6: In child: open slave device as controlling terminal

    // TODO 7: In child: dup2 slave FD to stdin/stdout/stderr

    // TODO 8: In child: exec shell command (use execl or system)

    // TODO 9: In parent: store master FD, child PID, set process_running = true

    // TODO 10: Set up SIGCHLD handler to update process_running on child exit

    // Hint: Use O_NOCTTY flag when opening slave to avoid terminal conflicts

}

// Read data from PTY master (shell output)

ssize_t pty_session_read(pty_session_t *session, char *buffer, size_t size) {

    // TODO 1: Check if session and buffer are valid

    // TODO 2: Check if process is still running (return 0 if not)

    // TODO 3: Call read() on master_fd with buffer and size

    // TODO 4: Handle EAGAIN/EWOULDBLOCK for non-blocking reads

    // TODO 5: Handle other errors (EPIPE means child process died)

    // TODO 6: Return number of bytes read, or -1 on error

    // Hint: Use non-blocking reads to avoid hanging the multiplexer

}

// Write data to PTY master (user input to shell)

ssize_t pty_session_write(pty_session_t *session, const char *data, size_t length) {
```

C

```

    // TODO 1: Validate session and data parameters

    // TODO 2: Check if process is still running

    // TODO 3: Call write() on master_fd with data and length

    // TODO 4: Handle partial writes by looping until complete

    // TODO 5: Handle EPIPE error (child process terminated)

    // TODO 6: Return total bytes written, or -1 on error

    // Hint: Use a loop for partial writes: while (written < length) { ... }

}

```

Language-Specific Hints

C Implementation Tips:

- Use `posix_openpt()` instead of opening `/dev/ptmx` directly for better portability
- Always call `grantpt()` and `unlockpt()` after `posix_openpt()` before accessing slave
- Use `setsid()` in child process before opening slave device to establish controlling terminal
- Set up signal handlers with `sigaction()`, not `signal()`, for reliable behavior
- Use `tcgetattr()` / `tcsetattr()` to save/restore terminal attributes
- Consider using `select()` or `poll()` to multiplex between multiple PTY master file descriptors

Memory Management:

- Use `calloc()` instead of `malloc()` for structure initialization to zero
- Always check allocation return values before using pointers
- Implement cleanup functions that can handle partially initialized structures
- Use consistent naming for cleanup functions: `*_destroy()` or `*_cleanup()`

Milestone Checkpoints

After implementing basic data structures:

- Compile with `gcc -Wall -Wextra src/data_model/*.c tests/test_data_model.c -o test_data_model`
- Run `./test_data_model` - should show successful structure allocation/deallocation
- Check with `valgrind ./test_data_model` - should report no memory leaks
- Verify screen buffer can be allocated with maximum dimensions (1024x256)

After implementing PTY session creation:

- Create a simple test program that spawns a shell in a PTY
- You should be able to write "echo hello\n" and read back "hello" plus shell prompt

- Check `ps` output to verify child shell process exists with correct PPID
- Verify that closing master FD terminates child process cleanly

After implementing screen buffer operations:

- Test writing characters at different positions and reading them back
- Verify cursor tracking matches written positions
- Test buffer resize with existing content preservation
- Verify scrollback buffer stores lines that scroll off screen

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault during buffer access	Uninitialized buffer pointers	Run with <code>gdb</code> , check if cells array is NULL	Initialize all buffer pointers before use
Memory leak reports	Missing cleanup calls	Use <code>valgrind --leak-check=full</code>	Implement and call <code>*_destroy()</code> functions
Child process becomes zombie	No SIGCHLD handler	Check <code>ps</code> output for processes in Z state	Install signal handler to reap children
PTY allocation fails	Permission or resource limits	Check <code>errno</code> after <code>posix_openpt()</code>	Verify user permissions, check ulimit settings
Terminal corruption on exit	Raw mode not restored	Terminal shows no prompt after exit	Always restore terminal in exit handlers
Characters appear as boxes	UTF-8 handling errors	Check locale settings and character encoding	Validate UTF-8 sequences before storing

PTY Manager Component

Milestone(s): Milestone 1 (PTY Creation) - this section establishes the foundational PTY management capabilities that enable all subsequent terminal multiplexer functionality

Mental Model: The Process Nursery

Understanding PTYs requires thinking of the PTY Manager as a **process nursery** that creates isolated environments where child processes believe they have dedicated terminals. Just as a hospital nursery provides controlled environments for newborns, the PTY Manager creates controlled terminal environments for shell processes.

In the physical world, a terminal was historically a physical device with a keyboard and screen connected to a mainframe computer. Modern operating systems simulate this hardware relationship using **pseudo-terminals**

(PTYs), which consist of a master-slave pair. The master side acts like the "computer end" of the terminal connection, while the slave side acts like the "terminal hardware" that the shell process sees.

Think of the PTY Manager as operating a nursery where each "crib" (PTY pair) contains a sleeping child process (shell). The nursery attendant (PTY Manager) can communicate with each child through a window (master file descriptor) while the child believes it has its own private room (slave terminal). When a child writes to what it thinks is its terminal screen, those writes actually go to the master side where our multiplexer can intercept, process, and route them. Similarly, when we want to send keyboard input to a specific child, we write to the master and the operating system delivers it to the slave as if it came from a real keyboard.

This isolation is crucial because each shell process needs to believe it has complete control over its terminal. It needs to set terminal attributes, handle terminal signals, and manage cursor positioning without interference from other shells. The PTY Manager maintains this illusion while secretly coordinating multiple such environments through a single real terminal interface.

PTY Manager Interface

The PTY Manager provides a clean interface for creating, managing, and destroying pseudo-terminal sessions. Each session encapsulates a shell process running in its own controlled terminal environment. The interface abstracts away the complex details of PTY allocation, process forking, and file descriptor management.

Method Name	Parameters	Returns	Description
<code>pty_session_create</code>	<code>session: pty_session_t*</code> , <code>shell_command: char*</code>	<code>tmux_error_t</code>	Creates new PTY pair, forks child process, and starts shell with proper session setup
<code>pty_session_read</code>	<code>session: pty_session_t*</code> , <code>buffer: char*</code> , <code>size: size_t</code>	<code>ssize_t</code>	Reads output from shell process via PTY master, returns bytes read or -1 on error
<code>pty_session_write</code>	<code>session: pty_session_t*</code> , <code>data: char*</code> , <code>length: size_t</code>	<code>ssize_t</code>	Sends input to shell process via PTY master, returns bytes written or -1 on error
<code>pty_session_resize</code>	<code>session: pty_session_t*</code> , <code>width: int</code> , <code>height: int</code>	<code>tmux_error_t</code>	Updates PTY terminal size and notifies child process via SIGWINCH signal
<code>pty_session_is_alive</code>	<code>session: pty_session_t*</code>	<code>bool</code>	Checks if child process is still running using non-blocking waitpid
<code>pty_session_get_exit_status</code>	<code>session: pty_session_t*</code>	<code>int</code>	Returns exit status of terminated child process, or -1 if still running
<code>pty_session_terminate</code>	<code>session: pty_session_t*</code> , <code>signal: int</code>	<code>tmux_error_t</code>	Sends signal to child process (SIGTERM, SIGKILL, etc.)
<code>pty_session_destroy</code>	<code>session: pty_session_t*</code>	<code>tmux_error_t</code>	Cleans up PTY file descriptors, reaps child process, and frees resources

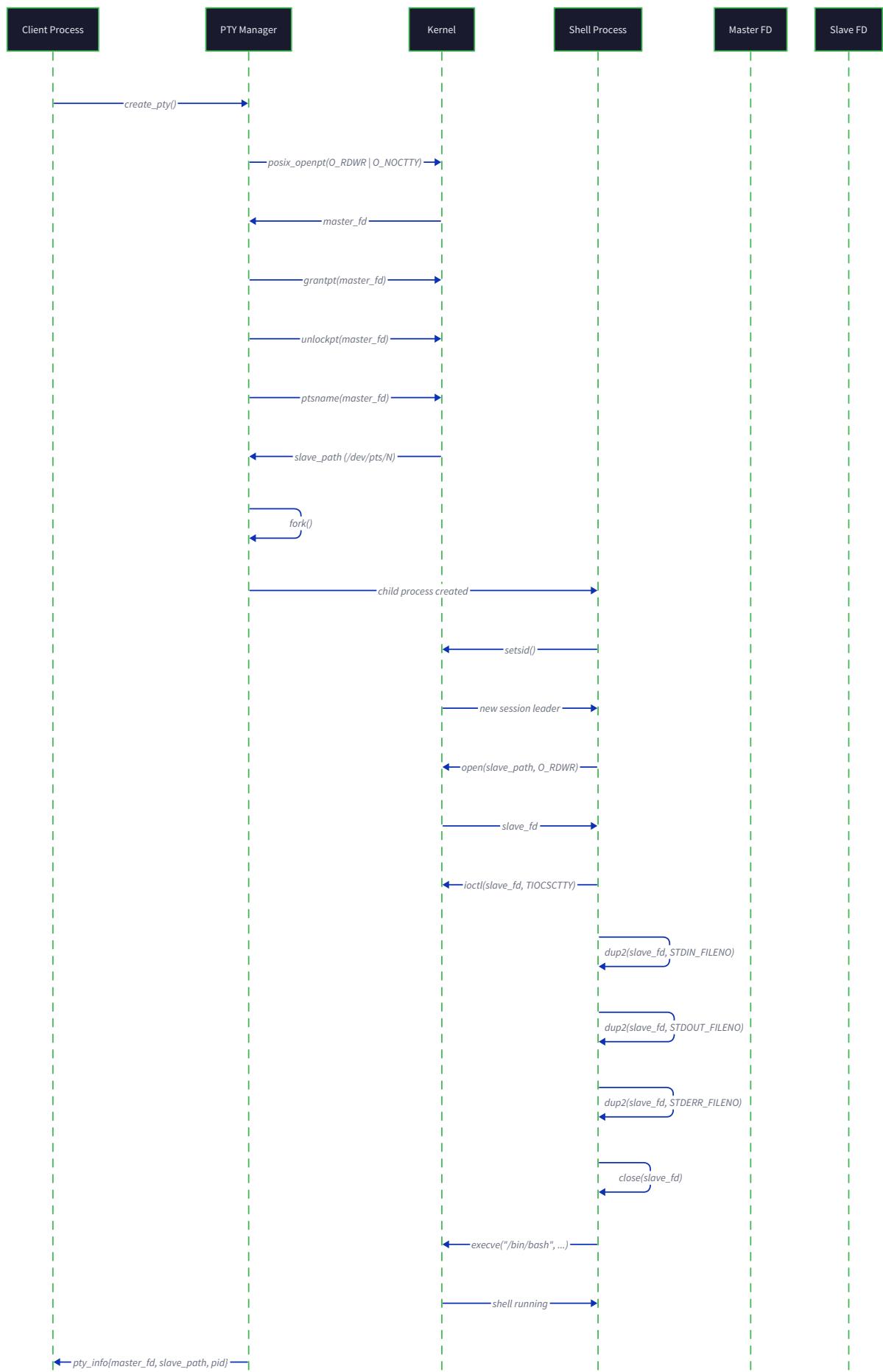
The `pty_session_t` structure maintains all state necessary for managing a single shell session:

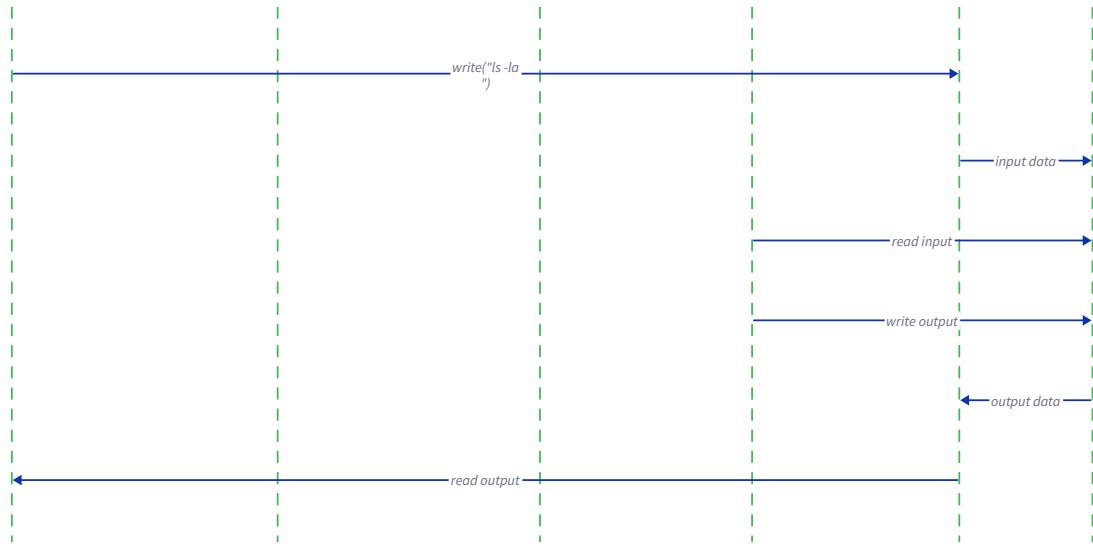
Field Name	Type	Description
master_fd	int	File descriptor for PTY master device, used for reading shell output and writing input
slave_fd	int	File descriptor for PTY slave device, typically closed after fork in parent process
child_pid	pid_t	Process ID of forked shell child, used for signal delivery and status monitoring
process_running	bool	Flag indicating whether child process is still alive, updated by SIGCHLD handler
exit_status	int	Exit code returned by child process after termination, obtained via waitpid

The interface design prioritizes simplicity and safety. Each function returns explicit error codes rather than relying on global errno values. The session structure encapsulates all necessary state, avoiding global variables that could complicate multi-session management. File descriptor ownership is clear: the PTY Manager owns the master descriptor while the child process owns the slave.

PTY Creation and Management

The PTY creation process involves a carefully orchestrated sequence of system calls that must be executed in the correct order to establish proper session leadership and terminal ownership. The algorithm handles the complex interaction between PTY allocation, process forking, and session management.





The PTY creation algorithm follows these essential steps:

- 1. PTY Master Allocation:** The system allocates a PTY master device using `posix_openpt(O_RDWR | O_NOCTTY)`. The `O_NOCTTY` flag prevents the PTY from becoming the controlling terminal of the current process. This returns a file descriptor for the master side of the PTY pair.
- 2. PTY Master Configuration:** The master device requires permission and ownership setup through `grantpt()` and `unlockpt()` system calls. These functions configure the slave device permissions and make it available for opening by child processes.
- 3. Slave Device Path Resolution:** The `ptsname()` function returns the filesystem path to the corresponding slave device (typically `/dev/pts/N` where N is a number). This path will be used by the child process to open its terminal device.
- 4. Process Fork:** The parent process forks using `fork()`, creating an identical child process. At this point, both processes have access to the master file descriptor, but only the child will open and use the slave.
- 5. Child Process Session Setup:** The child process must establish itself as a session leader before opening the slave terminal. It calls `setsid()` to create a new session and process group, making itself the session leader with no controlling terminal.
- 6. Slave Terminal Acquisition:** The child opens the slave device using the path obtained from `ptsname()`. This open operation establishes the slave PTY as the child's controlling terminal because the child is a session leader without an existing controlling terminal.
- 7. Standard File Descriptor Redirection:** The child duplicates the slave file descriptor to `stdin (0)`, `stdout (1)`, and `stderr (2)` using `dup2()`. This ensures all shell I/O operations use the PTY slave device.
- 8. Shell Process Execution:** The child executes the shell program using `execve()` or similar, replacing its process image with the shell while preserving the PTY file descriptors and session relationship.
- 9. Parent Cleanup:** The parent process closes its copy of the slave file descriptor (since it will never use it) and stores the master descriptor and child PID in the session structure for ongoing management.

The terminal attribute configuration requires special attention. The PTY slave initially inherits default terminal settings that may not be appropriate for shell interaction. The child process typically configures terminal attributes using `tcsetattr()` to enable features like canonical input processing, signal generation, and proper character echo behavior.

Window size initialization is critical for proper shell operation. Many shell programs and applications query terminal dimensions to format their output appropriately. The PTY Manager must set initial window size using the `TIOCSWINSZ` ioctl on the master file descriptor, typically matching the current terminal's dimensions.

Signal handling configuration affects both parent and child processes. The parent must install a `SIGCHLD` handler to detect child process termination asynchronously. The child inherits the parent's signal disposition but may modify signal handling after becoming a session leader, particularly for terminal-related signals like `SIGINT`, `SIGTSTP`, and `SIGWINCH`.

Architecture Decision Records

The PTY Manager design involves several critical architectural decisions that affect system reliability, performance, and complexity. Each decision represents a trade-off between different technical approaches with distinct advantages and limitations.

Decision: PTY Allocation Method

- Context:** Multiple system interfaces exist for PTY allocation, including BSD-style `openpty()`, System V-style `getpty()`, and POSIX `posix_openpty()`. Each has different portability characteristics and feature sets.
- Options Considered:** Direct `/dev/ptmx` access, `openpty()` library function, `posix_openpty()` system call
- Decision:** Use `posix_openpty()` for PTY master allocation
- Rationale:** POSIX `posix_openpty()` provides standardized behavior across Unix systems while giving explicit control over flags like `O_NOCTTY`. Unlike `openpty()`, it doesn't hide important setup steps, making the code more educational and debuggable.
- Consequences:** Requires explicit calls to `grantpt()` and `unlockpt()`, but provides better error handling granularity and clearer understanding of the PTY setup process.

Option	Pros	Cons	Chosen?
<code>openpty()</code>	Handles setup automatically, very simple API	Hides important setup steps, less portable	No
<code>posix_openpty()</code>	POSIX standard, explicit control, good error handling	Requires manual <code>grantpt/unlockpt</code> calls	Yes
Direct <code>/dev/ptmx</code>	Maximum control, no library dependencies	Non-portable, complex permission handling	No

Decision: Child Process Monitoring Strategy

- **Context:** The parent process must track child process lifecycle to detect termination, collect exit status, and prevent zombie processes. Options include synchronous monitoring, asynchronous signal handling, or hybrid approaches.
- **Options Considered:** Blocking `waitpid()` calls, `SIGCHLD` signal handler with `waitpid()`, polling with `WNOHANG`
- **Decision:** Asynchronous `SIGCHLD` signal handler combined with non-blocking status checks
- **Rationale:** Signal-driven monitoring provides immediate notification of child state changes without blocking the main event loop. Non-blocking `waitpid(WNOHANG)` allows status collection without race conditions.
- **Consequences:** Requires careful signal handler implementation to avoid race conditions, but provides responsive child process management without blocking I/O operations.

Option	Pros	Cons	Chosen?
Blocking <code>waitpid()</code>	Simple, reliable	Blocks event loop, poor responsiveness	No
<code>SIGCHLD</code> handler	Immediate notification, non-blocking	Complex signal handling, race conditions	Yes
Polling approach	Simple, no signals	Delayed detection, CPU overhead	No

Decision: File Descriptor Management Strategy

- **Context:** PTY file descriptors must be managed carefully across fork operations, with clear ownership boundaries between parent and child processes. Poor management leads to descriptor leaks or broken communication.
- **Options Considered:** Shared descriptors, exclusive ownership, descriptor passing protocols
- **Decision:** Exclusive ownership with immediate cleanup after fork
- **Rationale:** Parent owns master descriptor exclusively, child owns slave exclusively. Each process closes descriptors it doesn't need immediately after fork, preventing leaks and eliminating ambiguity about responsibility.
- **Consequences:** Requires disciplined descriptor management but eliminates most file descriptor related bugs and simplifies debugging.

Option	Pros	Cons	Chosen?
Shared descriptors	Flexible access patterns	Complex ownership, race conditions	No
Exclusive ownership	Clear responsibility, no races	Requires careful handoff protocol	Yes
Descriptor passing	Dynamic ownership transfer	Complex implementation, error-prone	No

Decision: Terminal Size Handling Approach

- **Context:** Terminal applications need accurate window size information to format output correctly. Size changes must be propagated from the real terminal through the multiplexer to child processes.
- **Options Considered:** Static size initialization, periodic size polling, signal-driven size updates
- **Decision:** Signal-driven size updates with `SIGWINCH` propagation
- **Rationale:** The multiplexer catches `SIGWINCH` signals indicating terminal resize, updates PTY sizes using `TIOCSWINSZ` ioctl, and relies on the kernel to deliver `SIGWINCH` to child processes automatically.
- **Consequences:** Provides responsive resize handling without polling overhead, but requires proper signal handling setup and careful coordination between global and per-PTY size management.

Option	Pros	Cons	Chosen?
Static size	Simple, no coordination needed	Poor user experience, broken applications	No
Periodic polling	Eventually consistent	Delayed response, CPU overhead	No
Signal-driven	Immediate response, efficient	Complex signal handling coordination	Yes

Common PTY Pitfalls

PTY management involves numerous subtle requirements that frequently trip up developers implementing terminal multiplexers. These pitfalls often manifest as mysterious hangs, broken terminal behavior, or resource leaks that are difficult to debug without understanding the underlying PTY semantics.

⚠ Pitfall: Forgetting `setsid()` in Child Process

Many developers attempt to open the PTY slave device in the child process without first calling `setsid()` to establish session leadership. This causes the `open()` call to fail because a process can only acquire a controlling terminal if it's a session leader without an existing controlling terminal.

The symptom is that child processes fail to start properly, often with "Operation not permitted" errors when attempting to open the slave device. The root cause is that the child process inherited the parent's session and process group, making it ineligible to acquire a new controlling terminal.

The fix requires calling `setsid()` in the child process before opening the slave device. This system call creates a new session with the child as session leader, creates a new process group, and detaches from any existing

controlling terminal. After `setsid()`, the child can successfully open the PTY slave and establish it as the controlling terminal.

⚠ Pitfall: File Descriptor Leaks Across Fork

A common mistake is failing to close unused file descriptors after forking, leading to resource leaks and communication problems. The parent process should never hold the slave file descriptor, and the child should close the master descriptor immediately after fork.

When file descriptors leak across the fork boundary, several problems occur. The parent holding a slave descriptor prevents proper EOF detection when the child terminates. The child holding a master descriptor can interfere with the parent's I/O operations and prevents proper cleanup.

The solution is disciplined descriptor management: immediately after fork, the parent closes the slave descriptor and keeps only the master. The child closes the master descriptor and uses only the slave for its `stdin/stdout/stderr` redirection. This creates clean ownership boundaries and prevents interference.

⚠ Pitfall: Ignoring Terminal Attribute Configuration

PTY slave devices inherit default terminal attributes that are often inappropriate for interactive shell use. Failing to configure attributes like canonical mode, echo settings, and signal generation leads to shells that behave strangely or don't respond to user input properly.

The child process should configure terminal attributes using `tcgetattr()` and `tcsetattr()` after opening the slave device but before executing the shell. Common settings include enabling canonical input processing (so users can edit command lines), configuring appropriate echo behavior, and ensuring signal generation for Ctrl-C and similar key combinations.

⚠ Pitfall: Improper SIGCHLD Handling

Inadequate child process monitoring leads to zombie processes that consume system resources and prevent accurate session status tracking. The `SIGCHLD` signal handler must properly reap terminated children using `waitpid()` while avoiding race conditions.

The signal handler should use `waitpid(-1, &status, WNOHANG)` in a loop to collect all available child exit statuses without blocking. The `WNOHANG` flag prevents blocking if no children have terminated, while the loop ensures multiple simultaneous child terminations are handled correctly.

Race conditions occur when the main program checks child status while the signal handler is also accessing the same data structures. Use atomic operations or careful locking to coordinate between the signal handler and main program when updating session status.

⚠ Pitfall: Incorrect Window Size Propagation

Terminal applications rely on accurate window size information obtained through the `TIOCGWINSZ` ioctl. When the real terminal is resized, this information must be propagated to all PTY sessions, and child processes must be notified via `SIGWINCH` signals.

The multiplexer should install a `SIGWINCH` handler that updates the window size for all active PTY sessions using the `TIOCSWINSZ` ioctl on each master file descriptor. The kernel automatically delivers `SIGWINCH` to child

processes when their controlling terminal size changes, so no explicit signal sending is required.

Failure to propagate size changes results in applications that format output for incorrect dimensions, leading to broken display layouts and poor user experience. Terminal-based editors and pagers are particularly sensitive to accurate size information.

Pitfall: Blocking I/O in Event Loop

Using blocking I/O operations when reading from PTY master file descriptors can cause the entire multiplexer to hang when one shell produces no output. This destroys the multiplexer's responsiveness and makes it unusable for managing multiple sessions.

All PTY I/O must be non-blocking, typically achieved by setting the `O_NONBLOCK` flag on master file descriptors and using `select()`, `poll()`, or similar mechanisms to determine when I/O operations can proceed without blocking.

When `read()` returns `EAGAIN` or `EWOULDBLOCK`, this indicates no data is currently available, not an error condition. The event loop should simply move on to check other file descriptors rather than retrying the read operation.

Pitfall: Inadequate Error Recovery

PTY operations can fail in various ways that require graceful handling rather than program termination. Common failure modes include PTY allocation failures (when the system runs out of PTY devices), child process exec failures, and I/O errors on master file descriptors.

When PTY allocation fails, the session creation should return an appropriate error code rather than crashing.

When child process execution fails, the parent should detect this condition (usually through immediate child termination) and clean up resources appropriately.

I/O errors on master file descriptors often indicate that the child process has terminated and closed the slave device. This is a normal condition that should result in session cleanup rather than error propagation to the user.

Implementation Guidance

The PTY Manager implementation requires careful attention to system call ordering, error handling, and resource management. The following guidance provides concrete starting points for building robust PTY management functionality.

Technology Recommendations

Component	Simple Option	Advanced Option
PTY Allocation	<code>posix_openpty()</code> with manual setup	<code>forkpty()</code> library function
Process Management	Basic <code>fork()</code> + <code>execve()</code>	<code>posix_spawn()</code> with file actions
Signal Handling	Traditional <code>signal()</code>	<code>sigalfd()</code> or <code>sigaction()</code> with masks
I/O Multiplexing	<code>select()</code> with <code>fd_set</code>	<code>epoll()</code> or <code>kqueue()</code>

Recommended File Structure

```
terminal-multiplexer/
├── src/
│   ├── pty/
│   │   ├── pty_manager.c      ← Core PTY management implementation
│   │   ├── pty_manager.h      ← PTY Manager interface definitions
│   │   └── pty_session.c      ← Individual session management
│   ├── common/
│   │   ├── errors.h          ← Error code definitions
│   │   ├── types.h           ← Common type definitions
│   │   └── main.c             ← Application entry point
│   └── tests/
│       └── test_pty_manager.c ← PTY Manager unit tests
└── Makefile
```

Infrastructure Starter Code

File: [src/common/errors.h](#)

```
#ifndef TMUX_ERRORS_H

#define TMUX_ERRORS_H

typedef enum {

    TMUX_OK = 0,

    TMUX_ERROR_PTY_ALLOCATION = 1,

    TMUX_ERROR_PROCESS_SPAWN = 2,

    TMUX_ERROR_TERMINAL_SETUP = 3,

    TMUX_ERROR_MEMORY_ALLOCATION = 4,

    TMUX_ERROR_INVALID_PARAMETER = 5,

    TMUX_ERROR_IO_ERROR = 6,

    TMUX_ERROR_PROCESS_NOT_FOUND = 7

} tmux_error_t;

const char* tmux_error_string(tmux_error_t error);

#endif
```

File: [src/common/types.h](#)

```
#ifndef TMUX_TYPES_H
```

C

```
#define TMUX_TYPES_H
```

```
#include <sys/types.h>
```

```
#include <stdbool.h>
```

```
#define MAX_PANES 16
```

```
#define MAX_SCREEN_WIDTH 1024
```

```
#define MAX_SCREEN_HEIGHT 256
```

```
#define ESC_SEQ_BUFFER_SIZE 64
```

```
typedef struct {
```

```
    int master_fd;
```

```
    int slave_fd;
```

```
    pid_t child_pid;
```

```
    bool process_running;
```

```
    int exit_status;
```

```
} pty_session_t;
```

```
typedef struct {
```

```
    struct termios original_termios;
```

```
    bool raw_mode_active;
```

```
} terminal_state_t;
```

```
#endif
```

File: `src/common/errors.c`

```
#include "errors.h" C

const char* tmux_error_string(tmux_error_t error) {

    switch (error) {

        case TMUX_OK: return "Success";

        case TMUX_ERROR_PTY_ALLOCATION: return "PTY allocation failed";

        case TMUX_ERROR_PROCESS_SPAWN: return "Process spawn failed";

        case TMUX_ERROR_TERMINAL_SETUP: return "Terminal setup failed";

        case TMUX_ERROR_MEMORY_ALLOCATION: return "Memory allocation failed";

        case TMUX_ERROR_INVALID_PARAMETER: return "Invalid parameter";

        case TMUX_ERROR_IO_ERROR: return "I/O error";

        case TMUX_ERROR_PROCESS_NOT_FOUND: return "Process not found";

        default: return "Unknown error";
    }
}
```

Core Logic Skeleton Code

File: `src/pty/pty_manager.h`

```
#ifndef PTY_MANAGER_H
#define PTY_MANAGER_H

#include "../common/types.h"
#include "../common/errors.h"

// Creates a new PTY session with the specified shell command.

// Allocates PTY pair, forks child process, and sets up terminal environment.

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command);

// Reads output data from the shell process via PTY master.

// Returns number of bytes read, or -1 on error.

ssize_t pty_session_read(pty_session_t* session, char* buffer, size_t size);

// Writes input data to the shell process via PTY master.

// Returns number of bytes written, or -1 on error.

ssize_t pty_session_write(pty_session_t* session, const char* data, size_t length);

// Checks if the child shell process is still running.

// Updates process_running and exit_status fields as needed.

bool pty_session_is_alive(pty_session_t* session);

// Resizes the PTY to match new terminal dimensions.

// Sends SIGWINCH to child process automatically via kernel.

tmux_error_t pty_session_resize(pty_session_t* session, int width, int height);

// Terminates the child process with the specified signal.

// Common signals: SIGTERM (graceful), SIGKILL (forced).

tmux_error_t pty_session_terminate(pty_session_t* session, int signal);

// Cleans up PTY session resources and reaps child process.

// Must be called for every successfully created session.
```

C

```
tmux_error_t pty_session_destroy(pty_session_t* session);

#endif
```

File: `src/pty/pty_manager.c`

```
#define _GNU_SOURCE C

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <sys/wait.h>

#include <signal.h>

#include <string.h>

#include <errno.h>

#include "pty_manager.h"

tmux_error_t pty_session_create(pty_session_t* session, const char* shell_command) {

    // TODO 1: Validate input parameters (session not null, shell_command not null)

    // TODO 2: Open PTY master using posix_openpty(O_RDWR | O_NOCTTY)

    // TODO 3: Call grantpt() and unlockpt() to setup slave permissions

    // TODO 4: Get slave device path using ptsname()

    // TODO 5: Fork child process using fork()

    // TODO 6: In child: call setsid() to become session leader

    // TODO 7: In child: open slave device and redirect stdin/stdout/stderr

    // TODO 8: In child: exec shell command using execl("/bin/sh", "sh", "-c", shell_command, NULL)

    // TODO 9: In parent: close slave fd and store master_fd and child_pid in session

    // TODO 10: Set master_fd to non-blocking mode using fcntl(F_SETFL, O_NONBLOCK)

    // Hint: Check each system call return value and return appropriate error codes

}

ssize_t pty_session_read(pty_session_t* session, char* buffer, size_t size) {

    // TODO 1: Validate parameters (session not null, buffer not null, size > 0)

    // TODO 2: Check if process is still running using pty_session_is_alive()
```

```

// TODO 3: Read from master_fd using read() system call

// TODO 4: Handle EAGAIN/EWOULDBLOCK (non-blocking mode) by returning 0

// TODO 5: Handle other errors by returning -1

// TODO 6: Return number of bytes actually read

}

ssize_t pty_session_write(pty_session_t* session, const char* data, size_t length) {

    // TODO 1: Validate parameters (session not null, data not null, length > 0)

    // TODO 2: Check if process is still running

    // TODO 3: Write to master_fd using write() system call

    // TODO 4: Handle partial writes by tracking bytes written

    // TODO 5: Handle EAGAIN/EWOULDBLOCK by returning bytes written so far

    // TODO 6: Handle EPIPE (broken pipe) as normal termination condition

}

bool pty_session_is_alive(pty_session_t* session) {

    // TODO 1: Validate session parameter

    // TODO 2: If process_running is already false, return false immediately

    // TODO 3: Call waitpid(child_pid, &status, WNOHANG) to check child status

    // TODO 4: If waitpid returns 0, child is still running - return true

    // TODO 5: If waitpid returns child_pid, child has terminated - update exit_status

    // TODO 6: Set process_running to false and return false

    // Hint: WNOHANG prevents blocking if child hasn't changed state

}

tmux_error_t pty_session_resize(pty_session_t* session, int width, int height) {

    // TODO 1: Validate parameters (session not null, width > 0, height > 0)

    // TODO 2: Create struct winsize with ws_row = height, ws_col = width

    // TODO 3: Call ioctl(master_fd, TIOCSWINSZ, &winsize) to set PTY size

```

```

    // TODO 4: Check ioctl return value and return appropriate error code

    // TODO 5: Kernel automatically sends SIGWINCH to child process

    // Hint: No need to manually send SIGWINCH - kernel handles this

}

tmux_error_t pty_session_terminate(pty_session_t* session, int signal) {

    // TODO 1: Validate parameters (session not null, signal > 0)

    // TODO 2: Check if process is still running using pty_session_is_alive()

    // TODO 3: Send signal to child process using kill(child_pid, signal)

    // TODO 4: Check kill() return value and handle ESRCH (no such process)

    // TODO 5: Return appropriate error code based on results

}

tmux_error_t pty_session_destroy(pty_session_t* session) {

    // TODO 1: Validate session parameter

    // TODO 2: If process is still running, send SIGTERM then SIGKILL after delay

    // TODO 3: Close master_fd if it's still open (>= 0)

    // TODO 4: Wait for child process termination using waitpid() without WNOHANG

    // TODO 5: Clear session structure fields (set fds to -1, pid to 0, etc.)

    // TODO 6: Return TMUX_OK on successful cleanup

    // Hint: Use a short timeout between SIGTERM and SIGKILL for graceful shutdown

}

```

Language-Specific Hints

- Use `#define _GNU_SOURCE` at the top of source files to enable GNU extensions like `ptsname()`
- Check `errno` after system call failures to get specific error information
- Use `strerror(errno)` to convert error numbers to human-readable messages for debugging
- The `O_NOCTTY` flag prevents PTY from becoming the controlling terminal of the parent process
- Set `O_NONBLOCK` on master file descriptors to prevent blocking in the main event loop
- Use `tcgetattr()` and `tcsetattr()` in child processes to configure terminal attributes appropriately
- Install a `SIGCHLD` handler in the main program to handle asynchronous child termination

- Use `sigemptyset()` and `sigaddset()` to create proper signal masks for `sigaction()`

Milestone Checkpoint

After implementing the PTY Manager component, verify the following behavior:

Test Command: Compile and run a simple test program that creates a PTY session with `/bin/echo "Hello World"` :

```
gcc -o test_pty src/pty/*.c src/common/*.c tests/test_pty_manager.c
./test_pty
```

BASH

Expected Output: The test should successfully create a PTY session, read "Hello World\n" from the shell output, detect process termination, and clean up resources without any file descriptor leaks.

Manual Verification Steps:

1. Run `ps aux | grep [your-process]` during execution - should see parent and child processes
2. Check `/proc/[pid]/fd/` for both processes - parent should have master fd, child should have slave as 0,1,2
3. After child terminates, `ps aux` should show no zombie processes
4. Use `lsof` to verify no leaked file descriptors after session cleanup

Signs of Problems:

- Child process appears as `<defunct>` (zombie) - check `SIGCHLD` handling and `waitpid()` calls
- "Operation not permitted" errors - child process needs `setsid()` before opening slave
- Hanging during read operations - ensure master fd has `O_NONBLOCK` flag set
- File descriptor leaks - verify both parent and child close unused descriptors after fork

Terminal Emulator Component

Milestone(s): Milestone 2 (Terminal Emulation) - this section implements the core terminal emulation capabilities that parse escape sequences and maintain virtual screen buffers for each pane

Mental Model: The Display Interpreter

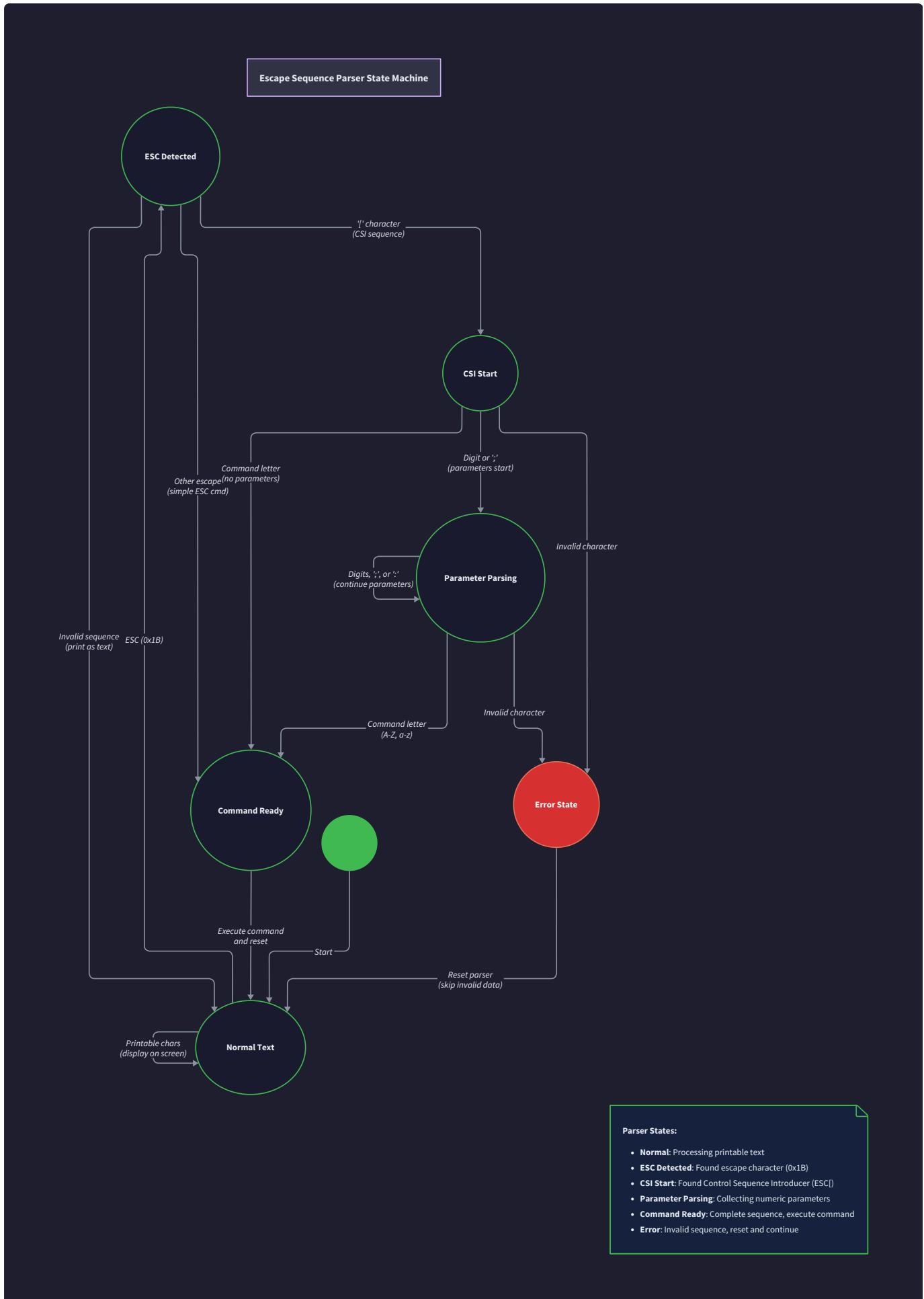
Understanding terminal emulation requires thinking of it as a specialized interpreter that translates a mixed stream of text and control codes into a two-dimensional display representation. Imagine you're a projectionist at an old movie theater who receives a continuous stream of film frames mixed with handwritten notes about projector settings. The film frames represent regular text characters that should appear on screen, while the handwritten notes are escape sequences that tell you to adjust the projector focus (cursor movement), change the lighting (text attributes), or switch reels (clear screen operations).

Just as the projectionist must separate the film frames from the control notes and apply the settings appropriately, the terminal emulator must parse the incoming byte stream to distinguish between printable characters and escape sequences. The projectionist maintains a mental model of the current screen state - what's currently being displayed, where the focus is, what lighting settings are active - and updates this state based on both the film frames and control notes. Similarly, the terminal emulator maintains a virtual screen buffer that tracks every character position, cursor location, and text attributes, updating this representation as it processes the input stream.

The key insight is that terminal emulation is fundamentally a **stateful translation process**. Unlike a simple text display that just shows characters as they arrive, a terminal emulator must maintain persistent state about cursor position, text attributes, scrolling regions, and screen contents. Each incoming byte might be part of a multi-byte escape sequence, requiring the emulator to buffer partial sequences until complete commands can be parsed and executed. This stateful nature means the terminal emulator serves as the authoritative source of truth about what each pane's display should look like at any given moment.

Escape Sequence Parser

The escape sequence parser forms the core intelligence of the terminal emulator, responsible for distinguishing between regular text and control commands in the incoming byte stream. Modern terminal applications use ANSI escape sequences, which begin with the escape character (ASCII 27) followed by specific character patterns that encode commands for cursor movement, text formatting, and screen manipulation.



The parser operates as a finite state machine that processes input bytes one at a time, transitioning between different parsing states based on the characters encountered. This approach handles the complexity of escape sequences that can span multiple bytes and include optional numeric parameters.

Parser State Machine

Current State	Input Character	Next State	Action Taken
NORMAL	Regular ASCII (32-126)	NORMAL	Add character to screen buffer at cursor position
NORMAL	Escape (27)	ESCAPE	Begin escape sequence, prepare parameter buffer
ESCAPE	'[' (CSI introducer)	CSI_PARAMS	Enter Control Sequence Introducer parsing mode
ESCAPE	']' (OSC introducer)	OSC_PARAMS	Enter Operating System Command parsing mode
ESCAPE	Any other char	NORMAL	Execute single-character escape, return to normal
CSI_PARAMS	Digits (0-9)	CSI_PARAMS	Accumulate numeric parameter value
CSI_PARAMS	',' (parameter separator)	CSI_PARAMS	Store current parameter, prepare for next
CSI_PARAMS	Letter (A-Z, a-z)	NORMAL	Execute CSI command with parameters
OSC_PARAMS	Any char except BEL/ESC	OSC_PARAMS	Accumulate OSC string parameter
OSC_PARAMS	BEL (7) or ESC+"	NORMAL	Execute OSC command with string parameter

The parser maintains several pieces of state to handle the complexity of escape sequence processing. The parameter buffer stores numeric values extracted from sequences like `\033[2;5H` (move cursor to row 2, column 5), while the command buffer accumulates the complete escape sequence for logging and debugging purposes. The parser also tracks whether it's in the middle of parsing a multi-byte UTF-8 character, since Unicode characters can span 2-4 bytes and must be handled atomically.

Core Escape Sequence Types

Sequence Pattern	Command Type	Parameters	Description
\033[nA	Cursor Up	n = lines	Move cursor up n lines (default 1)
\033[nB	Cursor Down	n = lines	Move cursor down n lines (default 1)
\033[nC	Cursor Right	n = columns	Move cursor right n columns (default 1)
\033[nD	Cursor Left	n = columns	Move cursor left n columns (default 1)
\033[r;cH	Cursor Position	r = row, c = col	Move cursor to absolute position
\033[nJ	Clear Display	n = mode	Clear screen (0=below, 1=above, 2=all)
\033[nK	Clear Line	n = mode	Clear line (0=right, 1=left, 2=all)
\033[nm	Set Attributes	n = attr code	Set text attributes (bold, color, etc.)
\033[s	Save Cursor	none	Save current cursor position
\033[u	Restore Cursor	none	Restore previously saved cursor position

Decision: State Machine vs. Regular Expression Parser

- **Context:** Need to parse complex, nested escape sequences with optional parameters and variable length
- **Options Considered:**
 1. Finite state machine with explicit state tracking
 2. Regular expression matching against buffered input
 3. Recursive descent parser with lookahead
- **Decision:** Finite state machine with byte-by-byte processing
- **Rationale:** State machine handles incomplete sequences gracefully, processes input in real-time without buffering requirements, and provides clear error recovery paths. Regular expressions require complete sequences in buffer and don't handle streaming input well. Recursive descent adds unnecessary complexity for this domain.
- **Consequences:** Enables real-time processing of PTY output without latency, handles network delays and partial reads correctly, but requires careful state management and explicit bounds checking

The escape sequence parser must handle several categories of malformed or incomplete input. Network delays or PTY buffering can cause escape sequences to arrive in fragments, requiring the parser to suspend processing mid-sequence and resume when additional bytes arrive. Invalid parameter values (negative numbers, excessively large coordinates) must be validated and either clamped to valid ranges or ignored entirely. Unknown escape sequences should be silently discarded rather than displayed as literal text, maintaining compatibility with terminal applications that use non-standard extensions.

UTF-8 Character Handling

Modern terminal applications must support Unicode text, which introduces additional complexity to the parsing process. UTF-8 encodes characters using 1-4 bytes, with the first byte indicating the total character length. The parser must accumulate multi-byte characters completely before adding them to the screen buffer, since partial UTF-8 sequences are invalid and would corrupt the display.

UTF-8 First Byte	Character Length	Remaining Bytes	Validation Required
0x00-0x7F	1 byte	none	ASCII compatibility check
0xC0-0xDF	2 bytes	1 continuation	Valid range 0x80-0x7FF
0xE0-0xEF	3 bytes	2 continuation	Valid range 0x800-0xFFFF
0xF0-0xF7	4 bytes	3 continuation	Valid range 0x10000-0x10FFFF

The UTF-8 decoder maintains a separate state machine that validates character encoding and accumulates bytes until complete characters are formed. Invalid UTF-8 sequences are typically replaced with the Unicode replacement character (U+FFFD) to maintain display integrity while indicating encoding problems.

Screen Buffer Management

The screen buffer serves as the virtual representation of each pane's terminal display, maintaining both the current visible content and a scrollback history of previous output. This component translates the parsed escape sequences into concrete changes to the 2D character grid, handles cursor movement and bounds checking, and manages the transition of content between the visible screen and scrollback storage.

The `screen_buffer_t` structure encapsulates all state necessary to represent a terminal display. The primary `cells` array stores the current visible content as a two-dimensional grid of `terminal_cell_t` structures, each containing a character, its display attributes, and foreground/background colors. The cursor position tracks where the next character will be placed, while the current attribute state determines how new characters will be formatted.

Screen Buffer Data Structure

Field Name	Type	Description
cells	terminal_cell_t**	Two-dimensional array of character cells (height × width)
width	int	Current screen width in character columns
height	int	Current screen height in character rows
cursor_x	int	Current cursor column position (0-based)
cursor_y	int	Current cursor row position (0-based)
scroll_offset	int	Number of lines scrolled up from bottom of scrollback
scrollback	terminal_cell_t**	Array of scrollback lines (oldest to newest)
scrollback_lines	int	Current number of lines stored in scrollback buffer
current_attributes	byte	Active text attributes (bold, underline, etc.)
current_fg_color	byte	Active foreground color code
current_bg_color	byte	Active background color code

Terminal Cell Structure

Field Name	Type	Description
ch	char	UTF-8 character data (may be first byte of multi-byte sequence)
attributes	byte	Text formatting attributes (bold, underline, reverse)
fg_color	byte	Foreground color code (0-255 for 256-color mode)
bg_color	byte	Background color code (0-255 for 256-color mode)

The screen buffer implements several key operations that correspond to the escape sequences parsed from the input stream. Character insertion places new content at the cursor position and advances the cursor, handling line wrapping when the cursor moves past the right edge of the screen. Cursor movement operations update the position while enforcing bounds checking to prevent invalid coordinates. Screen clearing operations reset portions of the character grid to default values (space character with default attributes).

Screen Buffer Operations

Operation	Parameters	Effect	Bounds Handling
Insert Character	<code>ch</code> , <code>attrs</code> , <code>fg</code> , <code>bg</code>	Place character at cursor, advance cursor	Wrap to next line at right edge
Move Cursor	<code>new_x</code> , <code>new_y</code>	Update cursor position	Clamp to valid screen coordinates
Clear Screen	<code>mode</code>	Reset cells to default (space + default attrs)	Clear visible area or scrollback
Clear Line	<code>mode</code> , <code>line_num</code>	Reset line cells to default	Handle partial line clears
Scroll Up	<code>lines</code>	Move content up, add new lines at bottom	Transfer top lines to scrollback
Scroll Down	<code>lines</code>	Move content down, lose bottom lines	Generally not supported in scrollback
Set Attributes	<code>attrs</code> , <code>fg</code> , <code>bg</code>	Update current formatting state	Apply to subsequent character insertions

Decision: Separate Scrollback vs. Unified Buffer

- **Context:** Need to maintain history of terminal output that exceeds screen height while providing efficient access to visible content
- **Options Considered:**
 1. Separate arrays for visible screen and scrollback history
 2. Single large circular buffer containing both visible and historical content
 3. Linked list of line structures with separate visible window tracking
- **Decision:** Separate arrays with explicit scrollback management
- **Rationale:** Provides predictable memory usage with configurable scrollback limits, enables efficient screen updates without copying historical data, and simplifies cursor movement logic within visible area. Unified buffer would complicate coordinate calculations and waste memory for large scrollback. Linked list adds pointer overhead and cache locality issues.
- **Consequences:** Enables fast screen redraws and cursor operations, provides configurable memory usage, but requires explicit management of content transfer between visible screen and scrollback storage

The scrollback buffer maintains a separate storage area for lines that have scrolled off the top of the visible screen. When the screen scrolls up (either due to newline characters when the cursor is on the bottom row, or explicit scroll escape sequences), the top line of the visible screen transfers to the bottom of the scrollback buffer. This design preserves terminal history while maintaining fast access to the current display content.

Scrollback Management Algorithm

1. **Scroll Event Detection:** When cursor moves below bottom row or explicit scroll command received

2. **Line Transfer:** Copy top visible line to new scrollback entry at end of scrollback array
3. **Visible Screen Shift:** Move all visible lines up one position, clear bottom line
4. **Scrollbar Limit Enforcement:** If scrollback exceeds `MAX_SCROLLBACK`, remove oldest line
5. **Cursor Position Adjustment:** Update cursor coordinates to account for scroll operation
6. **Redraw Notification:** Mark screen buffer as requiring complete redraw for display

The screen buffer must handle several edge cases and error conditions during normal operation. Cursor movement commands that specify coordinates outside the valid screen area are clamped to the nearest valid position rather than being rejected entirely. Character insertion at the bottom-right corner of the screen triggers automatic scrolling to make room for additional content. Terminal resize operations require rebuilding the cell arrays with new dimensions while preserving as much existing content as possible.

Attribute and Color Management

Terminal text attributes provide formatting capabilities including bold, underline, and color display. The terminal emulator maintains current attribute state that applies to all subsequently inserted characters until changed by escape sequences. This stateful approach means that setting bold text affects all following characters until bold is explicitly disabled.

Attribute Bit	Constant Name	Visual Effect	Escape Sequence
0	<code>ATTR_BOLD</code>	Bright/thick character display	<code>\033[1m</code> (set), <code>\033[22m</code> (clear)
1	<code>ATTR_UNDERLINE</code>	Underlined character display	<code>\033[4m</code> (set), <code>\033[24m</code> (clear)
2	<code>ATTR_REVERSE</code>	Swap foreground/background colors	<code>\033[7m</code> (set), <code>\033[27m</code> (clear)

Color support in modern terminals typically includes both the basic 8-color ANSI set and extended 256-color mode. The basic colors use specific escape sequence parameters (30-37 for foreground, 40-47 for background), while 256-color mode uses a more complex parameter format with additional color index values.

Color Code	Color Name	Foreground Escape	Background Escape
0	COLOR_BLACK	\033[30m	\033[40m
1	COLOR_RED	\033[31m	\033[41m
2	COLOR_GREEN	\033[32m	\033[42m
3	COLOR_YELLOW	\033[33m	\033[43m
4	COLOR_BLUE	\033[34m	\033[44m
5	COLOR_MAGENTA	\033[35m	\033[45m
6	COLOR_CYAN	\033[36m	\033[46m
7	COLOR_WHITE	\033[37m	\033[47m
9	COLOR_DEFAULT	\033[39m	\033[49m

Architecture Decision Records

The terminal emulator component involves several critical design decisions that significantly impact performance, memory usage, and compatibility with existing terminal applications. These decisions establish the foundation for how escape sequences are processed, how screen content is stored and managed, and how the emulator handles edge cases and error conditions.

Decision: Immediate vs. Deferred Screen Updates

- **Context:** Terminal applications generate output at varying rates, from interactive commands to bulk text processing that produces thousands of lines per second
- **Options Considered:**
 1. Update screen buffer immediately as each character/escape sequence is processed
 2. Batch updates and apply them periodically during event loop cycles
 3. Queue updates and apply them only when rendering is required
- **Decision:** Immediate screen buffer updates with deferred rendering
- **Rationale:** Immediate updates maintain accurate terminal state for cursor queries and screen content inspection, while deferred rendering prevents excessive redraw operations during bulk output. This approach separates logical terminal state from display optimization.
- **Consequences:** Enables accurate terminal emulation and proper escape sequence handling, provides responsive cursor movement for interactive applications, but requires careful buffer management during high-throughput scenarios

Decision: Fixed vs. Dynamic Buffer Allocation

- **Context:** Screen buffers must accommodate terminal resize operations while managing memory efficiently across multiple panes
- **Options Considered:**
 1. Pre-allocate fixed-size buffers based on maximum expected terminal size
 2. Dynamically allocate and resize buffers based on current terminal dimensions
 3. Use sparse allocation with on-demand page allocation for large buffers
- **Decision:** Dynamic allocation with size limits and reallocation on resize
- **Rationale:** Dynamic allocation minimizes memory usage for small terminals while supporting resize operations. Size limits prevent excessive memory consumption from malicious or buggy applications. Pre-allocation wastes memory for typical use cases, while sparse allocation adds complexity without significant benefits.
- **Consequences:** Provides memory-efficient operation across varying terminal sizes, supports resize operations without data loss, but requires careful error handling for allocation failures and adds complexity to buffer management code

Decision: Character-Based vs. Grapheme-Based Text Handling

- **Context:** Modern terminals must support Unicode text including combining characters, emoji, and complex scripts that don't map directly to single character cells
- **Options Considered:**
 1. Store individual Unicode codepoints in each terminal cell
 2. Store complete grapheme clusters (user-perceived characters) spanning multiple cells
 3. Use UTF-8 byte sequences with variable-width character representation
- **Decision:** Single codepoint per cell with combining character support
- **Rationale:** Maintains compatibility with existing terminal applications that assume fixed-width character cells, while providing hooks for combining character attachment. Grapheme clusters would break cursor positioning logic, while UTF-8 bytes complicate character-based operations.
- **Consequences:** Provides good compatibility with existing terminal software and predictable cursor behavior, supports basic Unicode display, but may not handle complex scripts perfectly and requires additional logic for wide characters (CJK)

Buffer Management Architecture

Decision Aspect	Option A	Option B	Chosen Option	Rationale
Scrollbar Storage	Circular buffer	Linear array with compaction	Linear array	Simpler implementation, predictable memory usage
Line Representation	Array of character pointers	Array of complete line structures	Character grid	Direct indexing, cache-friendly access patterns
Memory Allocation	Single large allocation	Per-line allocation	Single allocation per buffer	Reduces fragmentation, faster access
Resize Handling	Preserve all content	Preserve visible content only	Preserve visible + recent scrollback	Balances functionality with implementation complexity

The buffer management architecture significantly impacts both performance and memory usage characteristics of the terminal multiplexer. A linear array approach for scrollback provides predictable memory access patterns and simplifies the implementation of scrollback search and content export functionality. The character grid representation enables direct coordinate-based access for cursor operations while maintaining compatibility with terminal applications that assume grid-based positioning.

Decision: Escape Sequence Buffer Size and Handling

- **Context:** Escape sequences can vary from single characters to complex sequences with multiple parameters, and applications may send malformed or excessively long sequences
- **Options Considered:**
 1. Fixed-size escape sequence buffer with truncation of oversized sequences
 2. Dynamic buffer allocation that grows as needed for long sequences
 3. Streaming parser that processes parameters without buffering complete sequences
- **Decision:** Fixed-size buffer (`ESC_SEQ_BUFFER_SIZE` = 64 bytes) with overflow detection
- **Rationale:** Fixed buffer prevents memory exhaustion from malformed input while accommodating all standard ANSI sequences. Dynamic allocation adds complexity and potential attack vectors. Streaming parser works for simple sequences but complicates parameter handling for complex commands.
- **Consequences:** Provides robust defense against malformed input and predictable memory usage, handles all standard terminal sequences correctly, but may truncate non-standard extended sequences and requires explicit overflow handling

Common Emulation Pitfalls

Terminal emulation involves numerous subtle details that can cause compatibility issues with real-world terminal applications. Understanding these common pitfalls helps avoid bugs that may not surface during basic testing but cause problems with specific shell environments or terminal-based applications.

⚠ Pitfall: Incomplete UTF-8 Character Handling

A common mistake is treating each incoming byte as a complete character without considering UTF-8 multi-byte sequences. When UTF-8 characters arrive fragmented across multiple read operations from the PTY, the terminal emulator may attempt to display partial characters, resulting in corrupted text display or invalid character encoding.

The error manifests when processing output from applications that generate Unicode text, particularly when the PTY buffer boundaries split multi-byte characters. For example, a 3-byte UTF-8 character might arrive as two bytes in one `read()` call and the final byte in the next call. Treating the first two bytes as separate characters corrupts both the incomplete character and subsequent display.

Correct approach: Implement a UTF-8 decoder state machine that buffers partial characters until complete sequences are received. Validate UTF-8 sequences and replace invalid encodings with the Unicode replacement character (U+FFFD) rather than displaying garbage.

⚠ Pitfall: Cursor Bounds Checking Inconsistencies

Many implementations inconsistently handle cursor movement that would place the cursor outside the valid screen area. Some escape sequences should clamp cursor coordinates to valid ranges, while others should be ignored entirely, and the behavior often depends on terminal mode settings.

The most common error is applying uniform bounds checking to all cursor operations. For example, absolute cursor positioning (`\033[H`) should clamp coordinates to valid ranges, while relative movement (`\033[A`) should stop at screen edges without wrapping. Additionally, some applications rely on cursor coordinates being preserved even when temporarily invalid, expecting the cursor to become visible again after screen resize or scroll operations.

Correct approach: Implement escape sequence specific bounds handling that matches the expected behavior of target terminal types (typically VT100/ANSI). Maintain separate "logical" and "physical" cursor positions when necessary, and carefully test with applications that use cursor positioning heavily (vim, emacs, htop).

⚠ Pitfall: Escape Sequence Parameter Default Values

ANSI escape sequences often have optional parameters with specific default values when omitted. A frequent mistake is using zero as the default for all parameters, when many sequences use 1 as the default value (e.g., cursor movement distances).

This error causes applications to behave incorrectly when they omit parameters and expect standard default behavior. For example, `\033[A` (cursor up with no parameter) should move up one line, not zero lines. Similarly, `\033[J` (clear screen) should clear from cursor to end when the parameter is omitted.

Correct approach: Implement parameter parsing that distinguishes between explicit zero values and missing parameters, applying the correct default value for each escape sequence type. Reference terminal documentation or existing implementations to verify default values for each supported sequence.

⚠ Pitfall: Screen Buffer Memory Management During Resize

Terminal resize operations require careful handling of existing screen content and scrollback history. Common errors include losing scrollback content, corrupting character data during buffer reallocation, or failing to update cursor positions relative to the new screen dimensions.

When the terminal width changes, existing line content may need to be rewrapped to fit the new dimensions. Lines that were previously wrapped due to width constraints may need to be joined, while long lines may need new wrap points. Height changes affect how much scrollback content remains accessible and whether the cursor position remains valid.

Correct approach: Implement resize operations as atomic transactions that preserve content integrity. Save cursor position relative to content rather than absolute coordinates, handle line rewrapping carefully, and ensure scrollback content transfers correctly to new buffer structures.

Pitfall: Race Conditions in Multi-threaded Access

If the terminal emulator processes PTY input on a different thread from display rendering, race conditions can occur when the screen buffer is modified while being rendered. This leads to corrupted display output or crashes when render code accesses invalid memory.

The issue is particularly problematic during high-throughput scenarios where PTY output arrives faster than the display can be updated. Without proper synchronization, render operations may see partially updated screen state or encounter buffer reallocations mid-render.

Correct approach: Use proper synchronization primitives (mutexes, read-write locks) to protect screen buffer access, or use a single-threaded event loop design that serializes all buffer modifications. If using separate threads, consider double-buffering techniques where updates occur on a background buffer that's atomically swapped during render operations.

Pitfall: Attribute State Persistence Across Operations

Terminal text attributes (bold, color, etc.) should persist across cursor movements and other non-printing operations, but many implementations incorrectly reset attributes during certain escape sequence processing. This causes applications that set formatting and then move the cursor to lose their formatting state.

The error typically occurs in parsers that treat attribute-setting sequences as separate from cursor operations, rather than understanding that attribute state is global terminal state that persists until explicitly changed.

Correct approach: Maintain attribute state separately from cursor operations and screen content. Only modify attribute state when processing specific attribute-setting escape sequences (`\033[m` and variations), and ensure that cursor movement, scrolling, and other operations preserve the current attribute state for subsequent character insertion.

Implementation Guidance

The terminal emulator component bridges parsed escape sequences and visual display, requiring careful attention to state management, performance, and compatibility. This implementation focuses on correctness and clarity while providing the necessary performance for real-time terminal applications.

Technology Recommendations

Component	Simple Option	Advanced Option
UTF-8 Handling	Basic ASCII with UTF-8 detection	Full Unicode with ICU library
Screen Buffer	Two-dimensional character array	Optimized line-based storage with copy-on-write
Color Support	8-color ANSI only	256-color with true color (24-bit) support
Attribute Storage	Bit flags in character cells	Separate attribute stack with inheritance
Scrollbar	Fixed circular buffer	Compressed scrollback with search capabilities

Recommended File Structure

```

src/
  terminal_emulator/
    emulator.c           ← main terminal emulator logic
    emulator.h           ← public interface definitions
    escape_parser.c      ← escape sequence parsing state machine
    escape_parser.h      ← parser interface and constants
    screen_buffer.c      ← screen buffer management and operations
    screen_buffer.h      ← buffer data structures and API
    utf8.c               ← UTF-8 character handling utilities
    utf8.h               ← UTF-8 decoder interface
    test_emulator.c      ← unit tests for emulator functionality
  
```

Core Data Structure Definitions (Complete)

```
#include <stdint.h>
#include <stdbool.h>
#include <sys/types.h>

// Terminal color constants

#define COLOR_BLACK      0
#define COLOR_RED        1
#define COLOR_GREEN      2
#define COLOR_YELLOW     3
#define COLOR_BLUE       4
#define COLOR_MAGENTA    5
#define COLOR_CYAN       6
#define COLOR_WHITE      7
#define COLOR_DEFAULT    9

// Terminal attribute flags

#define ATTR_BOLD        (1 << 0)
#define ATTR_UNDERLINE   (1 << 1)
#define ATTR_REVERSE     (1 << 2)

// Buffer size limits

#define MAX_SCREEN_WIDTH 1024
#define MAX_SCREEN_HEIGHT 256
#define MAX_SCROLLBACK   1000
#define ESC_SEQ_BUFFER_SIZE 64

// Individual character cell in terminal display

typedef struct {
    uint32_t ch;           // Unicode codepoint (UTF-32 for simplicity)
    uint8_t attributes;    // Text attributes (bold, underline, etc.)
```

C

```

    uint8_t fg_color;           // Foreground color (0-255)

    uint8_t bg_color;           // Background color (0-255)

} terminal_cell_t;

// Complete screen buffer for one pane

typedef struct {

    terminal_cell_t **cells;      // 2D array [height][width]

    int width;                   // Current screen width

    int height;                  // Current screen height

    int cursor_x;                // Current cursor column (0-based)

    int cursor_y;                // Current cursor row (0-based)

    int scroll_offset;           // Lines scrolled up from bottom

    terminal_cell_t **scrollback; // Array of scrollback lines

    int scrollback_lines;        // Current number of scrollback lines

    uint8_t current_attributes;  // Active text attributes

    uint8_t current_fg_color;    // Active foreground color

    uint8_t current_bg_color;    // Active background color

    int saved_cursor_x;          // Saved cursor position (for save/restore)

    int saved_cursor_y;          // Saved cursor position

} screen_buffer_t;

// Escape sequence parser state

typedef enum {

    PARSER_STATE_NORMAL,

    PARSER_STATE_ESCAPE,

    PARSER_STATE_CSI_PARAMS,

    PARSER_STATE_OSC_PARAMS,

    PARSER_STATE_UTF8_CONTINUE
}

```

```
    } parser_state_t;

    // Escape sequence parser context

    typedef struct {

        parser_state_t state;

        char esc_buffer[ESC_SEQ_BUFFER_SIZE];

        int esc_buffer_len;

        int csi_params[16];           // CSI numeric parameters

        int csi_param_count;

        int utf8_bytes_remaining;     // For UTF-8 multi-byte sequences

        uint32_t utf8_codepoint;      // Accumulated UTF-8 codepoint

    } escape_parser_t;
```

Screen Buffer Management (Complete Implementation)

```
#include "screen_buffer.h"                                     C

#include <stdlib.h>

#include <string.h>

// Initialize screen buffer with specified dimensions

int screen_buffer_init(screen_buffer_t *buffer, int width, int height) {

    // TODO 1: Validate width and height against MAX_SCREEN_WIDTH/HEIGHT limits

    // TODO 2: Allocate 2D cells array using malloc for height*width cells

    // TODO 3: Initialize all cells to space character with default attributes

    // TODO 4: Allocate scrollback array for MAX_SCROLLBACK lines

    // TODO 5: Initialize cursor position to (0,0) and set default attributes

    // TODO 6: Set current colors to COLOR_WHITE foreground, COLOR_BLACK background

    // Hint: Use calloc to zero-initialize cell memory

    return TMUX_OK;

}

// Add character to screen buffer at current cursor position

void screen_buffer_put_char(screen_buffer_t *buffer, uint32_t ch) {

    // TODO 1: Check if cursor position is within valid screen bounds

    // TODO 2: Set character and attributes in cell at cursor position

    // TODO 3: Apply current foreground/background colors to the cell

    // TODO 4: Advance cursor to next position (handle line wrapping)

    // TODO 5: If cursor moves past bottom of screen, trigger scroll operation

    // Hint: Line wrapping moves to column 0 of next line

}

// Execute cursor movement escape sequence

void screen_buffer_move_cursor(screen_buffer_t *buffer, int new_x, int new_y) {

    // TODO 1: Clamp new_x to valid range [0, buffer->width-1]
```

```
// TODO 2: Clamp new_y to valid range [0, buffer->height-1]

// TODO 3: Update buffer cursor position to clamped coordinates

// Hint: Use min/max macros or conditional logic for clamping

}

// Clear portion of screen buffer based on escape sequence parameter

void screen_buffer_clear(screen_buffer_t *buffer, int mode) {

    // TODO 1: Handle mode 0 (clear from cursor to end of screen)

    // TODO 2: Handle mode 1 (clear from beginning of screen to cursor)

    // TODO 3: Handle mode 2 (clear entire screen)

    // TODO 4: Set cleared cells to space character with current attributes

    // TODO 5: Preserve cursor position after clearing operation

    // Hint: Use nested loops to iterate through cell ranges

}

// Scroll screen content up by specified number of lines

void screen_buffer_scroll_up(screen_buffer_t *buffer, int lines) {

    // TODO 1: Transfer top lines from visible screen to scrollback buffer

    // TODO 2: Shift remaining visible lines up to fill vacated space

    // TODO 3: Clear new lines at bottom of screen with default attributes

    // TODO 4: Enforce MAX_SCROLLBACK limit by removing oldest scrollback lines

    // TODO 5: Update scroll_offset if user is viewing scrollback history

    // Hint: Use memmove for efficient line copying, handle scrollback overflow

}

// Cleanup screen buffer and free allocated memory

void screen_buffer_cleanup(screen_buffer_t *buffer) {

    // TODO 1: Free each row in the 2D cells array

    // TODO 2: Free the cells array pointer itself
```

```
// TODO 3: Free each line in scrollback array  
  
// TODO 4: Free scrollback array pointer  
  
// TODO 5: Zero out buffer structure to prevent use-after-free  
  
}
```

Escape Sequence Parser (Complete Implementation)

```
#include "escape_parser.h"                                     C

#include <ctype.h>

#include <stdlib.h>

// Initialize escape sequence parser to default state

void escape_parser_init(escape_parser_t *parser) {

    // TODO 1: Set parser state to PARSER_STATE_NORMAL

    // TODO 2: Clear escape sequence buffer and reset buffer length to 0

    // TODO 3: Clear CSI parameter array and reset parameter count

    // TODO 4: Initialize UTF-8 state variables to handle multi-byte sequences

    // Hint: Use memset to clear arrays efficiently

}

// Process single byte through escape sequence parser

void escape_parser_process_byte(escape_parser_t *parser, screen_buffer_t *buffer,
                                char byte) {

    switch (parser->state) {

        case PARSER_STATE_NORMAL:

            // TODO 1: Check if byte is ESC character (27) to begin escape sequence

            // TODO 2: Check if byte starts UTF-8 multi-byte sequence (>= 0x80)

            // TODO 3: For regular ASCII characters, call screen_buffer_put_char

            // TODO 4: Handle control characters (newline, carriage return, etc.)

            // Hint: Use UTF-8 first byte patterns to detect multi-byte sequences

            break;

        case PARSER_STATE_ESCAPE:

            // TODO 1: Check for CSI introducer '[' to enter parameter parsing

            // TODO 2: Check for OSC introducer ']' for operating system commands

            // TODO 3: Handle single-character escape sequences (save cursor, etc.)

    }
}
```

```

        // TODO 4: Return to normal state after processing escape character

        // Hint: Build escape sequence in buffer for debugging/logging

        break;

case PARSER_STATE_CSI_PARAMS:

    // TODO 1: Accumulate numeric digits into current parameter value

    // TODO 2: Handle parameter separator ';' to advance to next parameter

    // TODO 3: Execute CSI command when letter terminator is found

    // TODO 4: Validate parameter count and ranges before execution

    // TODO 5: Return to normal state after command execution

    // Hint: Use isdigit() to detect numeric characters

    break;

case PARSER_STATE_UTF8_CONTINUE:

    // TODO 1: Validate byte as valid UTF-8 continuation (0x80-0xBF range)

    // TODO 2: Accumulate bits into codepoint being assembled

    // TODO 3: Decrement remaining byte count for multi-byte sequence

    // TODO 4: When sequence complete, output character and return to normal

    // TODO 5: Handle invalid UTF-8 by outputting replacement character

    // Hint: UTF-8 continuation bytes contribute 6 bits each to codepoint

    break;

}

}

// Execute parsed CSI (Control Sequence Introducer) command

static void execute_csi_command(escape_parser_t *parser, screen_buffer_t *buffer,
                               char command) {

    // TODO 1: Handle cursor movement commands (A=up, B=down, C=right, D=left)

```

```
// TODO 2: Handle absolute cursor positioning (H command with row;col params)

// TODO 3: Handle screen/line clearing commands (J for screen, K for line)

// TODO 4: Handle text attribute setting (m command with attribute codes)

// TODO 5: Handle cursor save/restore commands (s and u)

// TODO 6: Use parameter defaults (usually 1) when parameters are omitted

// Hint: CSI parameters default to 1, not 0, for most movement commands

}

// Convert UTF-8 first byte to expected sequence length

static int utf8_sequence_length(unsigned char first_byte) {

    // TODO 1: Return 1 for ASCII characters (0x00-0x7F)

    // TODO 2: Return 2 for two-byte sequences (0xC0-0xDF)

    // TODO 3: Return 3 for three-byte sequences (0xE0-0xEF)

    // TODO 4: Return 4 for four-byte sequences (0xF0-0xF7)

    // TODO 5: Return 0 for invalid UTF-8 first bytes

    // Hint: Use bit pattern matching to determine sequence length

}
```

UTF-8 Character Handling Utilities

```

#include "utf8.h"                                     C

#include <stdint.h>

// Validate complete UTF-8 sequence and return codepoint

uint32_t utf8_decode(const char *bytes, int length) {

    // TODO 1: Validate sequence length matches first byte pattern

    // TODO 2: Check that all continuation bytes are in valid range 0x80-0xBF

    // TODO 3: Assemble codepoint from sequence bits (6 bits per continuation)

    // TODO 4: Validate final codepoint is in valid Unicode range

    // TODO 5: Check for overlong encodings and other invalid patterns

    // TODO 6: Return 0xFFFFD (replacement character) for invalid sequences

    // Hint: First byte contributes 7,5,4,3 bits for 1,2,3,4 byte sequences

}

// Check if codepoint requires double-width character cell (CJK characters)

bool utf8_is_wide_character(uint32_t codepoint) {

    // TODO 1: Check for CJK Unified Ideographs range (U+4E00-U+9FFF)

    // TODO 2: Check for Hangul Syllables range (U+AC00-U+D7AF)

    // TODO 3: Check for other wide character ranges defined in Unicode

    // TODO 4: Return false for ASCII and most Latin characters

    // Hint: Wide characters occupy two terminal columns in display

}

```

Language-Specific Hints for C Implementation

- Use `stdint.h` types (`uint32_t`, `uint8_t`) for predictable character and color storage across platforms
- Employ `memmove()` rather than `memcpy()` when copying overlapping screen buffer regions during scroll operations
- Implement proper UTF-8 validation to prevent buffer overruns from malformed multi-byte sequences
- Use `calloc()` for screen buffer allocation to ensure cells are zero-initialized with default attributes
- Consider using `likely()`/`unlikely()` compiler hints for the normal character path vs. escape sequence processing if performance becomes critical

Milestone Checkpoint: Terminal Emulation Verification

After implementing the terminal emulator component, verify correct operation with these tests:

Basic Character Display: Run `echo "Hello World"` and confirm characters appear at correct cursor positions with proper advancement. Test UTF-8 characters with `echo "Ño  l 中文"` to verify multi-byte character handling.

Cursor Movement: Test cursor positioning with `printf "\033[5;10HTest"` (should place "Test" at row 5, column 10). Verify bounds checking with coordinates outside screen area.

Screen Clearing: Test `clear` command or `printf "\033[2J"` to verify screen clearing operation. Test line clearing with `printf "\033[K"` at various cursor positions.

Text Attributes: Test color and formatting with `printf "\033[1;31mBold Red\033[0m Normal"`. Verify attribute persistence across cursor movements.

High-throughput Output: Test with `cat /dev/urandom | hexdump -C` or similar to verify parser handles rapid escape sequence streams without corruption.

Signs of Problems: Characters appearing in wrong positions indicate cursor logic errors. Corrupted display during rapid output suggests parser state machine issues. Missing colors/attributes indicate attribute handling problems. Crashes during UTF-8 text suggest character decoder issues.

Window Manager Component

Milestone(s): Milestone 3 (Window Management) - this section implements the pane splitting, layout calculation, and multi-pane rendering capabilities that allow multiple shell sessions to share screen real estate

Mental Model: The Space Organizer

Understanding the window manager component requires thinking about it as a sophisticated space organizer, similar to how a building architect divides floor space among different rooms and tenants. Just as an architect must efficiently partition a fixed building footprint among competing space requirements while maintaining proper circulation and structural integrity, the window manager must dynamically partition the finite terminal screen real estate among multiple competing shell sessions while maintaining usable dimensions and clear visual separation.

Consider the analogy of a newspaper layout editor working on a fixed page size. The editor receives requests for different article spaces—some need wide columns for text, others need square regions for photos, and advertisements demand specific aspect ratios. The editor must split the available page space using horizontal and vertical dividers, ensuring each allocated region meets minimum size requirements while maximizing space utilization. When new content arrives, existing allocations must be resized to make room, but no region can shrink below its minimum viable dimensions.

The window manager operates under similar constraints with additional complexity. Unlike a static newspaper layout, the terminal window manager must handle dynamic changes—new panes appear when users split existing

ones, panes disappear when shells exit, users actively resize regions, and the entire terminal window itself may change size when the user resizes their terminal emulator. Each pane contains a living shell session that expects a consistent rectangular screen area and becomes confused if its dimensions change unexpectedly.

Critical Insight: The window manager's primary responsibility is maintaining the illusion that each shell session has its own dedicated terminal, while actually coordinating shared access to a single physical terminal display. This requires careful synchronization between logical pane boundaries and the actual character grid positions where each pane's content appears.

The space organization challenge becomes particularly complex because terminal displays operate on discrete character cell boundaries rather than pixel coordinates. A pane cannot be positioned at arbitrary screen coordinates—it must align to character boundaries and have dimensions measured in complete character cells. This discretization constraint means resize operations often involve distributing leftover space or adjusting multiple panes simultaneously to accommodate the character grid limitations.

Pane Layout Algorithm

The window manager implements a hierarchical space partitioning system based on a binary tree data structure called the layout tree. Each node in the layout tree represents either a concrete pane containing a shell session or an abstract split that divides space between two child regions. This tree structure naturally models the recursive splitting behavior users expect—split a window horizontally, then split one of the resulting halves vertically, then split one of those quarters horizontally again.

The fundamental layout data structure is the `layout_node_t`, which serves dual purposes depending on its type. Leaf nodes represent actual panes and contain a `pane_id` that references the corresponding `pane_t` structure containing the PTY session and screen buffer. Internal nodes represent splits and contain a `split_ratio` value between 0.0 and 1.0 that determines how the available space is divided between the left/right (or top/bottom) children.

Layout Node Structure:

Field Name	Type	Description
<code>type</code>	<code>layout_type_t</code>	Either <code>LAYOUT_LEAF</code> for panes or <code>LAYOUT_HORIZONTAL</code> / <code>LAYOUT_VERTICAL</code> for splits
<code>pane_id</code>	<code>int</code>	Valid only for leaf nodes, references the pane containing PTY session
<code>split_ratio</code>	<code>float</code>	Valid only for split nodes, ratio 0.0-1.0 determining space allocation
<code>left</code>	<code>layout_node_t*</code>	First child (top child for horizontal splits, left child for vertical splits)
<code>right</code>	<code>layout_node_t*</code>	Second child (bottom child for horizontal splits, right child for vertical splits)
<code>parent</code>	<code>layout_node_t*</code>	Parent node in tree, NULL for root node

The layout algorithm operates through a recursive tree traversal that calculates absolute screen coordinates and dimensions for each pane based on the split ratios and available space. The algorithm begins at the root node with the full terminal dimensions and recursively subdivides the space according to the split hierarchy.

Dimension Calculation Algorithm:

1. The algorithm receives the current node and available rectangular region defined by x-coordinate, y-coordinate, width, and height parameters
2. If the current node is a leaf, it updates the corresponding pane's position and dimensions directly and marks the pane for redrawing
3. If the current node is a horizontal split, it calculates the split position by multiplying the available height by the split ratio
4. The algorithm ensures the split position respects minimum pane height requirements by adjusting the split ratio if necessary
5. It recursively calls itself on the left child with the top portion of the available space (original height multiplied by adjusted split ratio)
6. It recursively calls itself on the right child with the bottom portion of the available space (remaining height after top allocation)
7. If the current node is a vertical split, it follows the same pattern but divides the available width instead of height
8. The algorithm propagates any minimum size constraint violations back up the tree, potentially adjusting parent split ratios to accommodate

The split ratio adjustment mechanism ensures that no pane shrinks below its minimum viable dimensions. When a resize operation would violate minimum size constraints, the algorithm performs a constraint satisfaction process that adjusts multiple split ratios simultaneously to find a valid layout configuration.

Pane Splitting Implementation:

When a user requests splitting an existing pane, the window manager must transform a leaf node in the layout tree into an internal split node with two children. This operation requires careful handling to maintain tree consistency and update all affected data structures.

1. The system identifies the target pane and locates its corresponding leaf node in the layout tree
2. It allocates a new pane structure with a fresh PTY session for the new shell instance
3. It creates a new layout node for the new pane and configures it as a leaf node
4. It transforms the existing leaf node into a split node by changing its type and adding the new pane as one child
5. It creates another new layout node for the original pane and attaches it as the other child of the split
6. It sets the initial split ratio to 0.5 to divide the space equally between the original and new panes
7. It triggers a full layout recalculation to update all pane positions and dimensions
8. It sends resize notifications to the affected PTY sessions so their shells understand their new terminal dimensions

Decision: Binary Tree Layout Structure

- **Context:** Multiple layout approaches exist for managing pane arrangements, including grid-based systems, constraint-based layouts, and tree-based hierarchical splitting
- **Options Considered:**
 - Grid system with fixed rows/columns
 - Constraint-based layout with flexible positioning
 - Binary tree with recursive splitting
- **Decision:** Binary tree with recursive splitting
- **Rationale:** Tree structure naturally models user mental model of splitting operations, enables efficient recursive algorithms for layout calculation, and provides straightforward implementation for resize operations. Grid systems become complex with irregular splits, while constraint-based systems require sophisticated solvers.
- **Consequences:** Splitting behavior matches user expectations from other terminal multiplexers, resize operations can be implemented with simple tree traversals, but complex layouts may require deep nesting levels

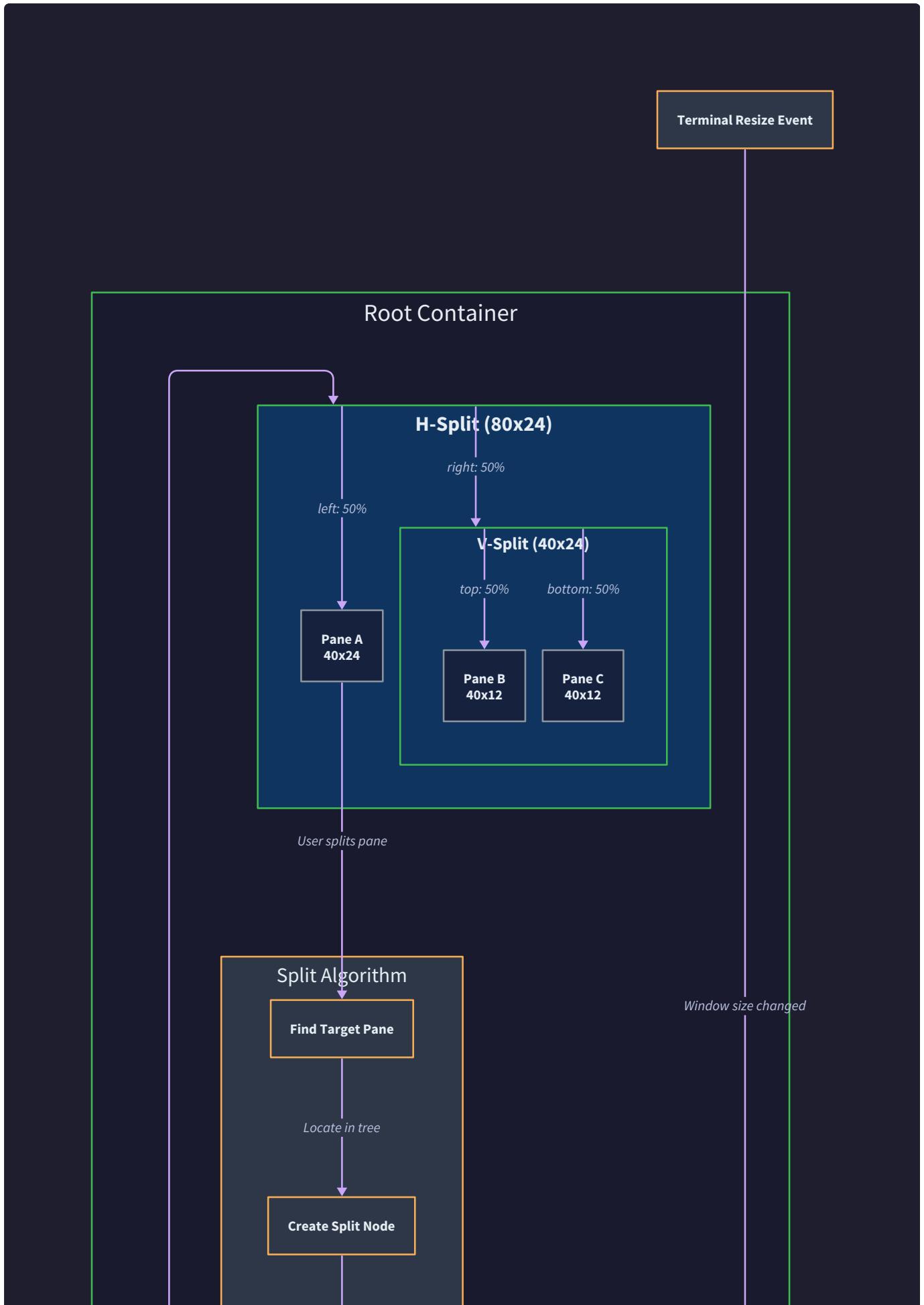
Layout Tree Options Comparison:

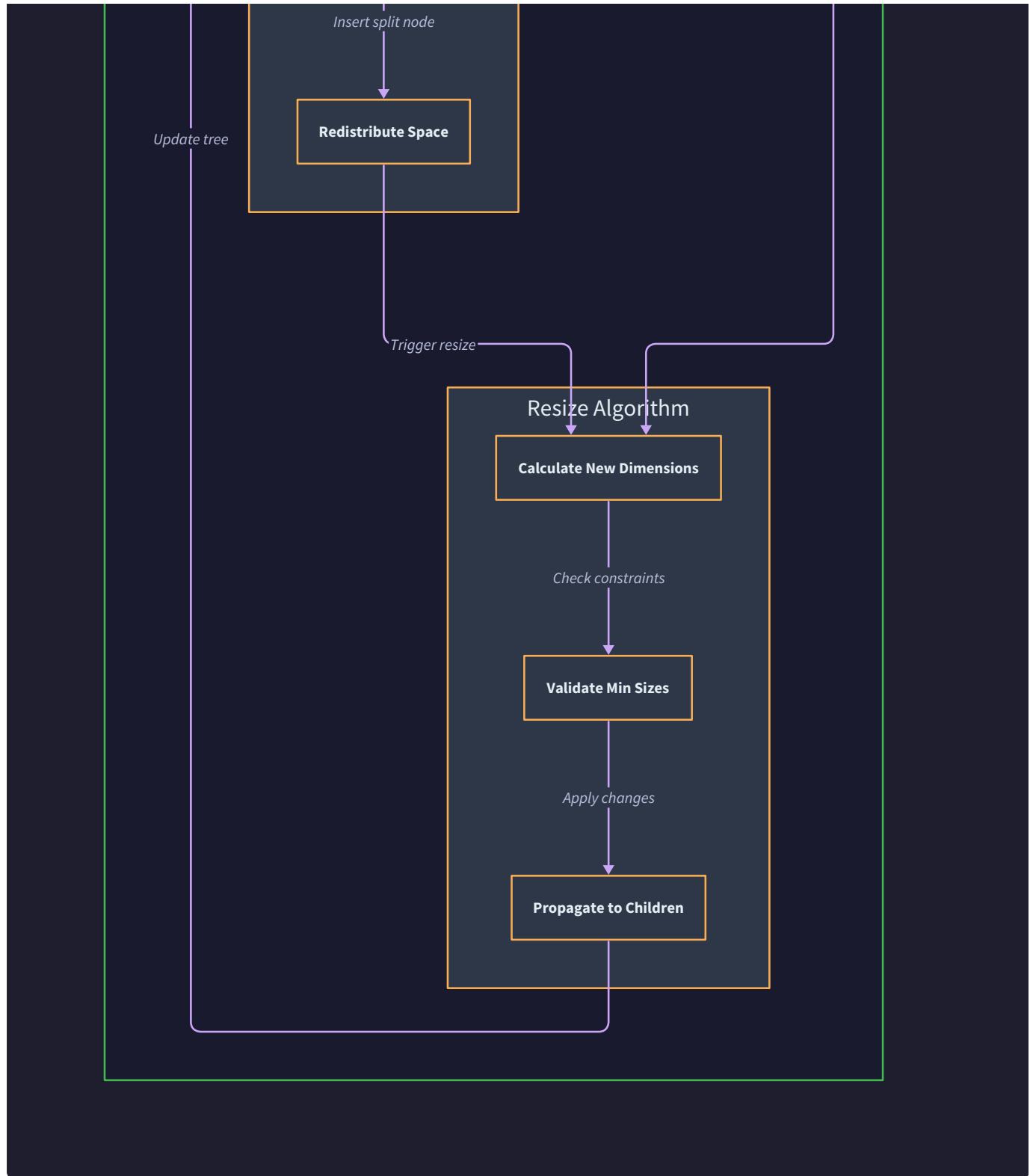
Layout Approach	Pros	Cons	Memory Usage
Binary Tree	Natural split modeling, Simple algorithms, Predictable behavior	Deep nesting, Limited layout flexibility	$O(n)$ nodes for n panes
Grid System	Uniform appearance, Simple coordinate math	Complex irregular splits, Wasted space	$O(\text{rows} \times \text{cols})$ regardless of pane count
Constraint-Based	Maximum flexibility, Optimal space usage	Complex implementation, Unpredictable behavior	$O(n^2)$ constraints for n panes

Multi-Pane Rendering

The multi-pane rendering system composites multiple independent screen buffers into a single terminal output stream while adding visual separators and status information. This rendering pipeline must coordinate updates from multiple asynchronous PTY sessions, handle partial screen updates efficiently, and maintain visual consistency across pane boundaries.

The rendering process operates on a frame-based model where the window manager periodically generates a complete terminal screen image by combining content from all visible panes. Each pane maintains its own `screen_buffer_t` containing the virtual terminal state for its associated shell session. The window manager must merge these independent buffers into a unified display while adding border characters and status information.





Border and Status Rendering:

The window manager reserves specific character positions for visual elements that help users distinguish between different panes and understand the current system state. Border characters are drawn along the edges of each pane using box-drawing Unicode characters that create clean visual separation between adjacent screen regions.

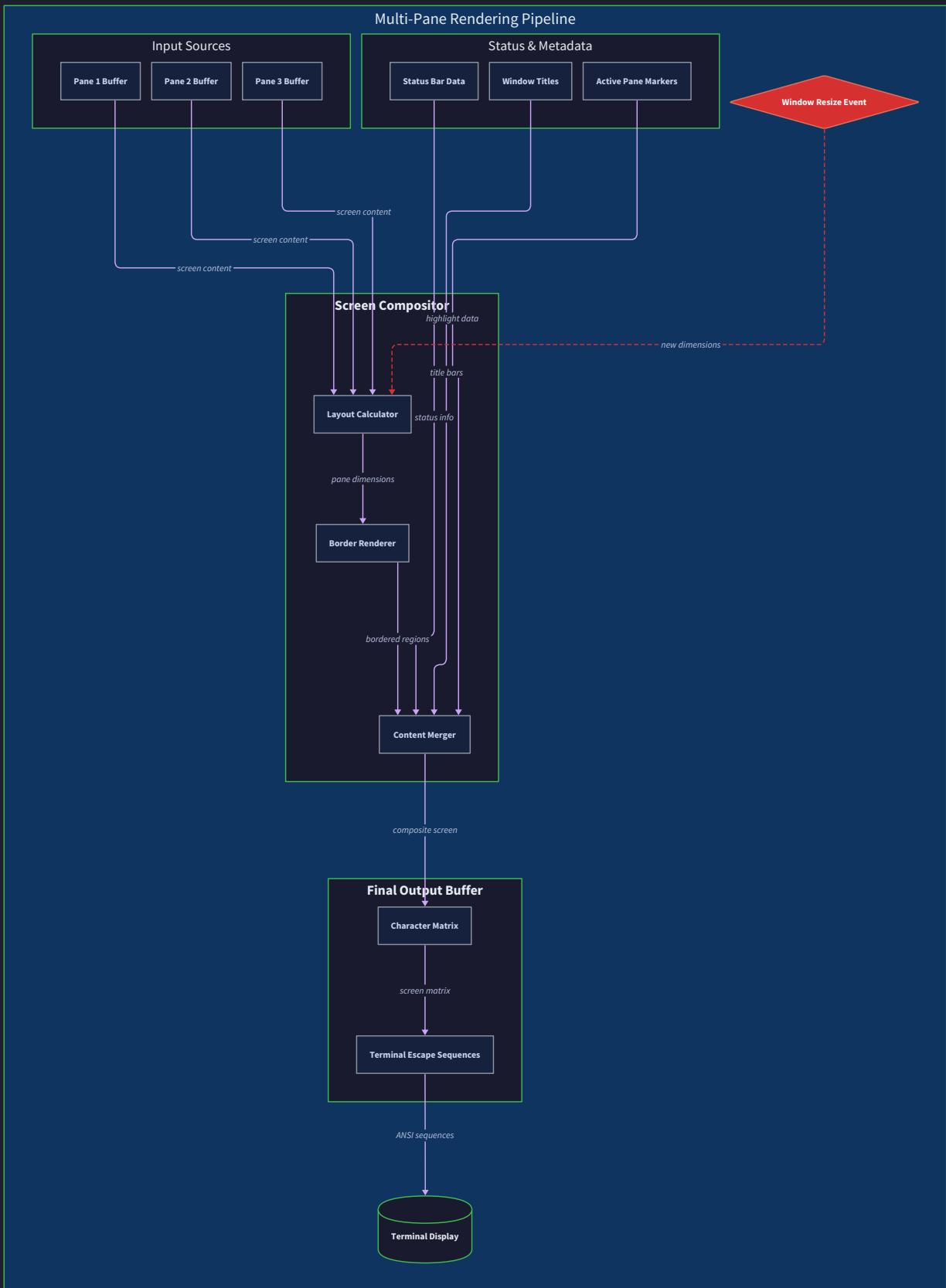
Border Element	Unicode Character	Usage
Horizontal Line	— (U+2500)	Top and bottom pane borders
Vertical Line	(U+2502)	Left and right pane borders
Top-Left Corner	┌ (U+250C)	Upper-left pane corner
Top-Right Corner	┐ (U+2510)	Upper-right pane corner
Bottom-Left Corner	└ (U+2514)	Lower-left pane corner
Bottom-Right Corner	┘ (U+2518)	Lower-right pane corner
Cross Intersection	+ (U+253C)	Where horizontal and vertical borders meet
Focused Pane Highlight	Different color/bold	Visual indication of active pane

The rendering pipeline must account for the space consumed by borders when calculating usable pane dimensions. A pane with screen coordinates spanning from (x, y) to (x+width, y+height) actually has usable

content area from $(x+1, y+1)$ to $(x+width-1, y+height-1)$ due to the border character overhead.

Frame Compositing Algorithm:

1. The renderer allocates a complete terminal-sized buffer initialized to blank characters with default attributes
2. It traverses the layout tree to identify all visible panes and their absolute screen coordinates
3. For each pane, it copies the relevant portion of the pane's screen buffer into the corresponding region of the composite buffer
4. It applies clipping to ensure pane content does not overflow its allocated boundaries
5. It draws border characters around each pane's perimeter, using intersection characters where borders from adjacent panes meet
6. It applies focus highlighting by modifying the border attributes for the currently active pane
7. It renders the status bar information at the bottom of the screen, showing pane numbers, window names, and system status
8. It converts the composite buffer into a sequence of terminal escape codes that positions the cursor and outputs each character with appropriate attributes
9. It optimizes the output by detecting unchanged screen regions and generating movement commands instead of redrawing static content



The output optimization step significantly improves rendering performance by avoiding unnecessary character output. The renderer maintains a shadow buffer representing the terminal's current state and compares it against

the new frame content to generate minimal update sequences.

Screen Update Optimization:

Update Type	Detection Method	Optimization Strategy
Character Changes	Compare cell content and attributes	Output only changed characters with cursor positioning
Line Insertions	Detect shifted content patterns	Use scroll region escape sequences
Line Deletions	Detect compacted content patterns	Use delete line escape sequences
Cursor Movements	Track cursor position changes	Generate optimal cursor positioning sequence
Attribute Changes	Compare text formatting	Group consecutive attribute changes

The optimization process must balance update efficiency against complexity. Simple character-by-character comparison provides good results without requiring sophisticated diff algorithms that might consume more CPU time than they save in terminal output.

Decision: Frame-Based Rendering with Optimization

- **Context:** Multiple rendering approaches exist, including immediate mode rendering, retained mode with full redraws, and optimized frame-based rendering
- **Options Considered:**
 - Immediate mode (output directly to terminal as changes occur)
 - Full frame redraw (regenerate entire screen each update)
 - Optimized frame-based (compare with previous frame and output minimal changes)
- **Decision:** Optimized frame-based rendering
- **Rationale:** Provides predictable visual consistency by ensuring all panes update atomically, enables efficient border and status rendering, and optimization reduces terminal output overhead significantly for typical usage patterns
- **Consequences:** Requires additional memory for shadow buffer and comparison logic, introduces frame latency between PTY output and display, but provides smooth visual experience and good performance

Architecture Decision Records

The window manager component involves several critical design decisions that significantly impact both implementation complexity and user experience. These decisions must balance performance requirements, memory usage constraints, and compatibility with existing terminal multiplexer user expectations.

Decision: Minimum Pane Dimensions

- **Context:** Panes must remain usable when users resize layouts aggressively, but different shells and applications have varying minimum size requirements
- **Options Considered:**
 - Fixed minimum (e.g., 10x3 characters) for all panes
 - Dynamic minimum based on shell capabilities detection
 - No minimum enforcement, allowing unusably small panes
- **Decision:** Fixed minimum of 10 characters width by 3 characters height
- **Rationale:** Most shell prompts require at least 8-10 characters for basic functionality, three lines provide space for prompt, command input, and one line of output. Fixed minimum simplifies layout calculations and prevents user confusion from dynamic constraints.
- **Consequences:** Some aggressive resize operations may be rejected when they would violate minimum size constraints, but ensures all panes remain functional for basic shell interaction

Minimum Pane Size Options:

Approach	Width	Height	Pros	Cons
Conservative	15 chars	5 lines	Always usable	Limits splitting density
Minimal	10 chars	3 lines	Good splitting flexibility	Some applications may be cramped
Dynamic	Variable	Variable	Optimal for each shell	Complex detection logic
None	1 char	1 line	Maximum flexibility	Unusable panes possible

Decision: Layout Tree Balance Strategy

- **Context:** Repeated splitting operations can create highly unbalanced trees that degrade layout calculation performance and produce awkward visual arrangements
- **Options Considered:**
 - No rebalancing, allow arbitrary tree shapes
 - Automatic rebalancing after each split operation
 - Manual rebalancing command for user-initiated optimization
- **Decision:** No automatic rebalancing, but provide manual layout reset functionality
- **Rationale:** Automatic rebalancing would disrupt user's mental model of the layout structure and change split ratios unexpectedly. Manual reset allows users to clean up complex layouts when desired while preserving their intentional arrangements.
- **Consequences:** Complex layouts with many splits may have slightly slower layout calculations, but user control over layout structure is preserved

Decision: Border Style Configuration

- **Context:** Different users prefer different visual styles for pane borders, and some terminals have varying support for Unicode box-drawing characters
- **Options Considered:**
 - Fixed Unicode box-drawing borders
 - ASCII fallback borders using |, -, and + characters
 - Configurable border styles with multiple options
- **Decision:** Unicode box-drawing with ASCII fallback detection
- **Rationale:** Unicode borders provide professional appearance on modern terminals, but ASCII fallback ensures compatibility with older terminals or systems with limited Unicode support. Automatic detection based on terminal capabilities avoids requiring user configuration.
- **Consequences:** Requires terminal capability detection logic and dual rendering paths, but provides optimal appearance across diverse terminal environments

Border Style Comparison:

Style	Characters Used	Compatibility	Visual Quality	Implementation Complexity
Unicode	− □ ▄ +	Modern terminals	Excellent	Medium (capability detection)
ASCII	- +	Universal	Basic	Simple
Configurable	User choice	Varies	User dependent	High (multiple render paths)

Decision: Pane Focus Indication

- **Context:** Users need clear visual indication of which pane currently receives keyboard input, especially in complex layouts with many panes
- **Options Considered:**
 - Different border color for focused pane
 - Border thickness/style change for focused pane
 - Background color change for focused pane content area
- **Decision:** Border color change with bold attribute for focused pane
- **Rationale:** Color change provides clear visual distinction without affecting pane content area, bold attribute ensures visibility even on terminals with limited color support, and border modification preserves content readability
- **Consequences:** Requires color capability detection and fallback to bold-only on monochrome terminals, but provides clear focus indication across terminal types

Common Window Management Pitfalls

Window management implementation involves several subtle issues that frequently cause problems for developers building terminal multiplexers. Understanding these pitfalls helps avoid frustrating debugging sessions and ensures robust operation across diverse terminal environments.

⚠ Pitfall: Incorrect PTY Size Propagation

Many developers forget that splitting a pane requires updating the PTY dimensions for the affected shell sessions. When a pane's screen dimensions change due to splitting or resizing, the associated PTY must be notified via the `TIOCSWINSZ` ioctl call, otherwise the shell process continues to believe it has the old terminal dimensions.

This manifests as applications within the pane formatting their output for incorrect line widths, causing text to wrap unexpectedly or leaving blank space. Text editors become particularly confused and may display content incorrectly or crash when their assumed terminal size doesn't match the actual available space.

The fix requires calling `pty_session_resize()` immediately after any layout calculation that changes pane dimensions. The resize notification must occur before any new output appears in the pane to ensure the shell process updates its internal state correctly.

⚠ Pitfall: Border Character Coordinate Confusion

A common mistake involves failing to account for border character space when calculating usable pane content areas. Developers often calculate pane positions correctly but forget that border characters consume screen real estate, leading to content overflow into adjacent panes or border characters.

For example, a pane positioned at (5, 10) with dimensions 20x8 actually has usable content space from (6, 11) to (23, 16) due to the border characters around the perimeter. Applications that ignore this adjustment will overwrite border characters or have their content clipped unexpectedly.

The solution involves maintaining separate calculations for pane layout coordinates (including borders) and content coordinates (excluding borders). Always subtract 2 from both width and height when configuring PTY dimensions, and offset content positioning by +1 in both x and y directions.

⚠ Pitfall: Floating Point Split Ratio Precision

Using floating point arithmetic for split ratios can introduce subtle rounding errors that accumulate during resize operations. When a user resizes the terminal window multiple times, small precision errors can cause pane dimensions to drift from their intended proportions.

This problem becomes particularly evident when users expect symmetric layouts (50/50 splits) but observe that repeated resize operations gradually make one pane larger than the other. The accumulated error eventually becomes visible and violates user expectations about layout behavior.

The fix involves implementing split ratio normalization that periodically adjusts ratios to eliminate accumulated errors. Additionally, using fixed-point arithmetic or rational number representation for split ratios can eliminate floating point precision issues entirely.

⚠ Pitfall: Race Conditions in Multi-Pane Updates

When multiple PTY sessions generate output simultaneously, race conditions can occur during screen buffer updates and rendering. Without proper synchronization, the window manager might read partially updated screen buffers or composite inconsistent pane states into the final output.

This manifests as visual glitches where panes display mixed content from different time points, or rendering artifacts where border characters disappear temporarily. The problem becomes more severe under heavy output load when multiple shell sessions are active simultaneously.

The solution requires implementing proper locking around screen buffer updates and ensuring that the rendering pipeline operates on consistent snapshots of all pane states. Consider using a double-buffering approach where PTY output updates a background buffer while rendering operates on a stable foreground buffer.

Pitfall: Memory Leaks in Layout Tree Operations

Layout tree manipulation, particularly node insertion and deletion during pane creation and destruction, often introduces memory leaks if not handled carefully. When a pane closes, its layout node must be removed from the tree, but developers frequently forget to properly deallocate the node structure or update parent pointers.

Additionally, when the last remaining pane in a split closes, the split node becomes redundant and should be removed to prevent tree degeneration. Failing to clean up these unnecessary nodes causes memory usage to grow over time and can lead to degraded layout calculation performance.

The fix requires implementing careful tree cleanup logic that recursively removes unnecessary split nodes and properly deallocates all associated memory. Consider using reference counting or garbage collection techniques to ensure proper cleanup of complex tree structures.

Common Pitfalls Summary:

Pitfall	Symptom	Root Cause	Prevention
PTY Size Mismatch	Text wrapping incorrectly	Missing TIOCSWINSZ calls	Update PTY size after every layout change
Border Overlap	Content overwrites borders	Wrong coordinate calculations	Separate layout coordinates from content coordinates
Split Ratio Drift	Layouts become asymmetric	Floating point precision errors	Implement ratio normalization
Rendering Artifacts	Visual glitches	Race conditions in updates	Synchronize buffer access properly
Memory Leaks	Growing memory usage	Incomplete tree cleanup	Implement proper node deallocation

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Layout Calculation	Basic recursive tree traversal	Cached layout with incremental updates
Border Rendering	ASCII characters with simple drawing	Unicode box-drawing with capability detection
Screen Composition	Character-by-character copying	Optimized diff-based rendering
Tree Management	Manual memory allocation	Reference-counted smart pointers

Recommended File Structure

```
terminal-multiplexer/
  src/
    window/
      layout.h           ← layout tree data structures and algorithms
      layout.c           ← implementation of layout calculation and tree manipulation
      render.h           ← rendering pipeline interfaces
      render.c           ← screen composition and border drawing
      window_manager.h   ← main window manager interface
      window_manager.c   ← window manager implementation
    test/
      test_layout.c      ← unit tests for layout algorithms
      test_render.c      ← rendering pipeline tests
```

Infrastructure Starter Code

Here's a complete layout tree management foundation that handles the complex tree manipulation operations:

```
// layout.h - Layout tree management interface

#ifndef LAYOUT_H

#define LAYOUT_H

#include <stdbool.h>

typedef enum {

    LAYOUT_LEAF,

    LAYOUT_HORIZONTAL,

    LAYOUT_VERTICAL

} layout_type_t;

typedef struct layout_node_s {

    layout_type_t type;

    int pane_id; // valid only for leaf nodes

    float split_ratio; // valid only for split nodes (0.0-1.0)

    struct layout_node_s *left;

    struct layout_node_s *right;

    struct layout_node_s *parent;

} layout_node_t;

// Complete tree management functions - ready to use

layout_node_t* layout_create_root_pane(int pane_id);

void layout_destroy_tree(layout_node_t *root);

layout_node_t* layout_find_pane_node(layout_node_t *root, int pane_id);

int layout_split_pane(layout_node_t *root, int pane_id, layout_type_t split_type, int new_pane_id);

int layout_close_pane(layout_node_t **root, int pane_id);

void layout_calculate_dimensions(layout_node_t *node, int x, int y, int width, int height,
panes_t *panes);
```

```
#endif

// layout.c - Complete implementation

#include "layout.h"

#include <stdlib.h>

#include <string.h>

#define MIN_PANE_WIDTH 10

#define MIN_PANE_HEIGHT 3

layout_node_t* layout_create_root_pane(int pane_id) {

    layout_node_t *node = malloc(sizeof(layout_node_t));

    if (!node) return NULL;

    memset(node, 0, sizeof(layout_node_t));

    node->type = LAYOUT_LEAF;

    node->pane_id = pane_id;

    return node;

}

void layout_destroy_tree(layout_node_t *root) {

    if (!root) return;

    layout_destroy_tree(root->left);

    layout_destroy_tree(root->right);

    free(root);

}

layout_node_t* layout_find_pane_node(layout_node_t *root, int pane_id) {

    if (!root) return NULL;
```

```
if (root->type == LAYOUT_LEAF && root->pane_id == pane_id) {

    return root;

}

layout_node_t *result = layout_find_pane_node(root->left, pane_id);

if (result) return result;

return layout_find_pane_node(root->right, pane_id);

}

int layout_split_pane(layout_node_t *root, int pane_id, layout_type_t split_type, int
new_pane_id) {

    layout_node_t *pane_node = layout_find_pane_node(root, pane_id);

    if (!pane_node || pane_node->type != LAYOUT_LEAF) {

        return TMUX_ERROR_MEMORY_ALLOCATION;

    }

    // Create new nodes for the split

    layout_node_t *new_pane_node = malloc(sizeof(layout_node_t));

    layout_node_t *old_pane_node = malloc(sizeof(layout_node_t));

    if (!new_pane_node || !old_pane_node) {

        free(new_pane_node);

        free(old_pane_node);

        return TMUX_ERROR_MEMORY_ALLOCATION;

    }

    // Setup new pane node
```

```

memset(new_pane_node, 0, sizeof(layout_node_t));

new_pane_node->type = LAYOUT_LEAF;

new_pane_node->pane_id = new_pane_id;

new_pane_node->parent = pane_node;

// Setup old pane node (copy of original)

memset(old_pane_node, 0, sizeof(layout_node_t));

old_pane_node->type = LAYOUT_LEAF;

old_pane_node->pane_id = pane_id;

old_pane_node->parent = pane_node;

// Transform original pane node into split node

pane_node->type = split_type;

pane_node->split_ratio = 0.5; // Equal split

pane_node->left = old_pane_node;

pane_node->right = new_pane_node;

return TMUX_OK;
}

void layout_calculate_dimensions(layout_node_t *node, int x, int y, int width, int height,
pane_t *panes) {

if (!node) return;

if (node->type == LAYOUT_LEAF) {

// Update pane dimensions (account for borders)

pane_t *pane = &panes[node->pane_id];

pane->x = x;

```

```

pane->y = y;

pane->width = width;

pane->height = height;

pane->needs_redraw = true;

// Update PTY size (subtract border space)

int content_width = width > 2 ? width - 2 : 1;

int content_height = height > 2 ? height - 2 : 1;

pty_session_resize(&pane->session, content_width, content_height);

return;

}

// Handle split nodes

if (node->type == LAYOUT_HORIZONTAL) {

    int split_pos = (int)(height * node->split_ratio);

    // Ensure minimum dimensions

    if (split_pos < MIN_PANE_HEIGHT) split_pos = MIN_PANE_HEIGHT;

    if (height - split_pos < MIN_PANE_HEIGHT) split_pos = height - MIN_PANE_HEIGHT;

    layout_calculate_dimensions(node->left, x, y, width, split_pos, panes);

    layout_calculate_dimensions(node->right, x, y + split_pos, width, height - split_pos,
panes);

} else if (node->type == LAYOUT_VERTICAL) {

    int split_pos = (int)(width * node->split_ratio);

    // Ensure minimum dimensions

    if (split_pos < MIN_PANE_WIDTH) split_pos = MIN_PANE_WIDTH;

```

```
    if (width - split_pos < MIN_PANE_WIDTH) split_pos = width - MIN_PANE_WIDTH;

    layout_calculate_dimensions(node->left, x, y, split_pos, height, panes);

    layout_calculate_dimensions(node->right, x + split_pos, y, width - split_pos, height,
panes);

}

}
```

Core Logic Skeleton Code

```
// window_manager.h - Core interface for learner implementation C

#ifndef WINDOW_MANAGER_H

#define WINDOW_MANAGER_H

#include "layout.h"
#include "../data_model.h"

typedef struct {

    pane_t panes[MAX_PANES];

    int pane_count;

    int active_pane_id;

    layout_node_t *layout_root;

    int terminal_width;

    int terminal_height;

    char *render_buffer;

    char *shadow_buffer;

} window_manager_t;

// Core functions for learner to implement

int window_manager_init(window_manager_t *wm, int width, int height);

int window_manager_create_pane(window_manager_t *wm, const char *shell_command);

int window_manager_split_pane(window_manager_t *wm, int pane_id, layout_type_t split_type);

int window_manager_close_pane(window_manager_t *wm, int pane_id);

int window_manager_switch_pane(window_manager_t *wm, int direction);

int window_manager_resize_terminal(window_manager_t *wm, int new_width, int new_height);

int window_manager_render_frame(window_manager_t *wm);

void window_manager_cleanup(window_manager_t *wm);

#endif
```

```
// window_manager.c - Skeleton for core implementation

#include "window_manager.h"

#include <stdlib.h>

#include <string.h>

int window_manager_create_pane(window_manager_t *wm, const char *shell_command) {

    // TODO 1: Check if pane limit (MAX_PANES) would be exceeded

    // TODO 2: Find next available pane ID in the panes array

    // TODO 3: Initialize new pane_t structure with ID and default properties

    // TODO 4: Call pty_session_create to start shell process in new pane

    // TODO 5: Initialize screen_buffer_t for new pane with appropriate dimensions

    // TODO 6: If this is first pane, create root layout node; otherwise handle as split

    // TODO 7: Update pane_count and set new pane as active_pane_id

    // TODO 8: Trigger layout recalculation to update all pane dimensions

    // Hint: Use layout_create_root_pane for first pane, layout_split_pane for subsequent

}

int window_manager_split_pane(window_manager_t *wm, int pane_id, layout_type_t split_type) {

    // TODO 1: Validate that pane_id exists and refers to valid pane

    // TODO 2: Check minimum size constraints - ensure pane is large enough to split

    // TODO 3: Create new pane structure and allocate next available pane ID

    // TODO 4: Initialize PTY session for new pane with same shell as original

    // TODO 5: Call layout_split_pane to update layout tree structure

    // TODO 6: Recalculate layout dimensions for all panes using layout_calculate_dimensions

    // TODO 7: Update pane_count and set new pane as active pane

    // TODO 8: Mark all affected panes for redraw due to dimension changes

    // Hint: Check that both resulting panes meet MIN_PANE_WIDTH/MIN_PANE_HEIGHT requirements

}
```

```

int window_manager_render_frame(window_manager_t *wm) {

    // TODO 1: Clear render buffer to blank characters with default attributes

    // TODO 2: Iterate through all panes and copy their screen_buffer content to appropriate
    // regions

    // TODO 3: Apply clipping to ensure pane content doesn't overflow boundaries

    // TODO 4: Draw border characters around each pane using box-drawing characters

    // TODO 5: Highlight focused pane border with different color/bold attribute

    // TODO 6: Render status bar at bottom with pane numbers and current time

    // TODO 7: Compare render buffer with shadow buffer to detect changes

    // TODO 8: Generate minimal terminal escape sequences to update only changed areas

    // TODO 9: Output escape sequences to terminal and update shadow buffer

    // Hint: Use Unicode box-drawing chars: -|□□□ with ASCII fallback: -|++++

}

int window_manager_switch_pane(window_manager_t *wm, int direction) {

    // TODO 1: Determine current active pane from active_pane_id

    // TODO 2: Based on direction (UP/DOWN/LEFT/RIGHT), find adjacent pane

    // TODO 3: Search layout tree to identify pane in requested direction

    // TODO 4: Update active_pane_id to new pane

    // TODO 5: Mark old and new active panes for redraw to update focus highlighting

    // TODO 6: Return error if no pane exists in requested direction

    // Hint: Direction finding requires traversing layout tree to find spatial neighbors

}

```

Language-Specific Hints

- Use `malloc()` and `free()` carefully for layout tree nodes - consider implementing a node pool to avoid fragmentation
- Terminal size detection: use `ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws)` to get current dimensions
- Box-drawing characters: Use UTF-8 encoded constants like `"\u2500"` for horizontal lines, with ASCII fallbacks

- Screen buffer copying: Use `memcpy()` for efficient block transfers when compositing pane content
- PTY resize: Call `ioctl(master_fd, TIOCSWINSZ, &winsize)` immediately after dimension changes
- Signal handling: Install `SIGWINCH` handler to detect terminal resize events
- Use `select()` or `poll()` in main loop to handle multiple PTY file descriptors efficiently

Milestone Checkpoint

After implementing the window manager component, verify the following behaviors:

Test Command: Compile and run your terminal multiplexer, then test splitting and layout operations:

```
# Compile the multiplexer                                BASH

gcc -o tmux src/*.c src/window/*.c

# Start the multiplexer

./tmux

# Test sequence (within multiplexer):

# 1. Press prefix key + 'v' to split vertically

# 2. Press prefix key + 'h' to split horizontally

# 3. Navigate between panes with prefix + arrow keys

# 4. Resize terminal window and observe layout updates
```

Expected Behaviors:

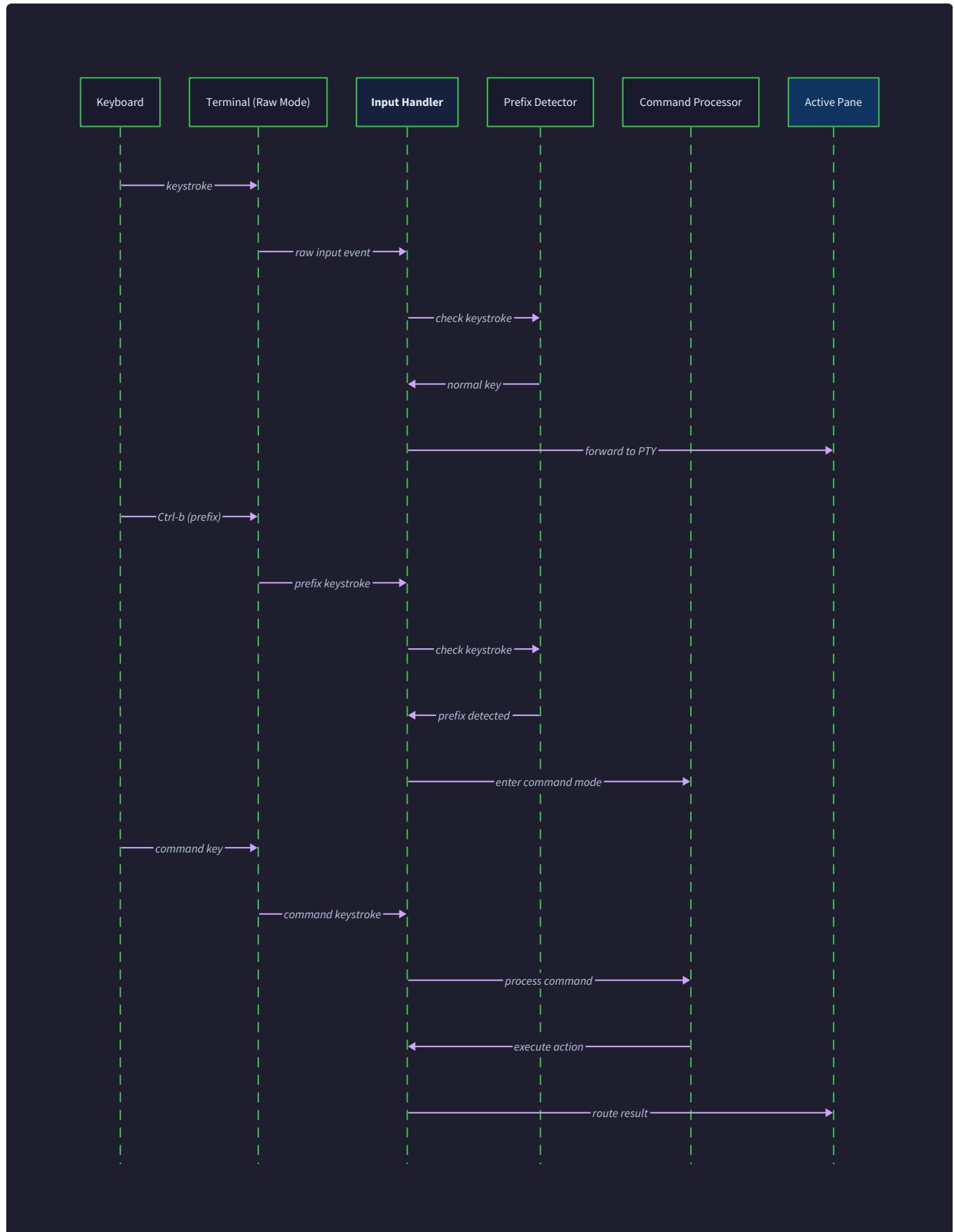
- Each pane should display an independent shell prompt and accept commands
- Border characters should cleanly separate panes without overlapping content
- Focused pane should have visually distinct border highlighting
- Terminal resize should proportionally adjust all pane dimensions
- PTY sessions should receive correct dimensions and format output appropriately

Common Issues and Diagnostics:

- **Panes overlap or have gaps:** Check border calculation in `layout_calculate_dimensions()`
- **Shell output wraps incorrectly:** Verify `pty_session_resize()` calls after layout changes
- **Borders use wrong characters:** Check terminal UTF-8 capability and fallback logic
- **Focus highlighting doesn't appear:** Verify color/attribute application in render pipeline
- **Layout becomes corrupted after splits:** Debug layout tree structure with print statements

Input Handler Component

Milestone(s): Milestone 4 (Key Bindings and UI) - this section implements the user interaction layer that enables command mode, key bindings, and input routing to control the terminal multiplexer



Mental Model: The Traffic Controller

Think of the Input Handler as a **traffic controller at a busy intersection**. Just like a traffic controller must decide whether to direct each car to continue straight through the intersection or pull them aside for inspection, the Input Handler must decide what to do with each keystroke that arrives from the user's keyboard.

Most keystrokes are like regular traffic - they should flow straight through to the active pane's shell process without any interference. When you type `ls -la` in your shell, those characters need to reach the shell process as quickly and transparently as possible, just like most cars need to flow through the intersection without stopping.

However, some keystrokes are special - they're like emergency vehicles that need different treatment. When the user presses the prefix key (typically Ctrl-b, similar to tmux), it's like an ambulance arriving at the intersection. The traffic controller must immediately recognize this special vehicle and route it to the appropriate emergency facility rather than letting it continue with normal traffic.

The Input Handler operates in two distinct modes, much like a traffic intersection can switch between normal flow and emergency response mode. In **normal mode**, keystrokes flow directly to the active pane's PTY master file descriptor, where they're forwarded to the shell process. The Input Handler acts as a transparent conduit, adding minimal latency to the user's typing experience.

In **command mode**, triggered by the prefix key, the Input Handler temporarily stops forwarding input to the shell and instead interprets keystrokes as multiplexer commands. This is like the traffic controller holding all normal traffic while they coordinate the emergency response. During this time, keys like 'h', 'j', 'k', 'l' become navigation commands to switch between panes, '"' becomes a horizontal split command, and '%' becomes a vertical split command.

The critical insight here is that the Input Handler must maintain **context awareness** - it needs to know not just what key was pressed, but also what state the multiplexer is currently in. Just like a traffic controller must track whether they're in normal mode or emergency mode, the Input Handler tracks whether it's in normal input mode or command mode. This state determines the entire interpretation and routing of subsequent input.

The Input Handler's dual-mode operation creates a clean separation between user interaction with the multiplexer itself versus user interaction with the shell processes running inside panes. This separation is essential for maintaining the illusion that each pane contains an independent terminal session.

Key Binding System

The **key binding system** forms the core of user interaction with the terminal multiplexer, translating raw keyboard input into multiplexer commands and routing decisions. This system must handle the complexity of distinguishing between input intended for the multiplexer versus input intended for the shell processes running in individual panes.

The system operates around a **prefix key mechanism**, borrowed from established multiplexers like tmux and screen. The prefix key acts as an escape sequence that signals the transition from normal input mode to command mode. When the user presses the prefix key (typically Ctrl-b), the Input Handler captures this keystroke

and enters a temporary command interpretation state where the next keystroke will be interpreted as a multiplexer command rather than shell input.

Prefix Key State Management

The prefix key implementation requires careful state tracking to handle timing and multi-keystroke sequences correctly. The Input Handler maintains a prefix state that transitions through several phases during command processing.

State	Description	Next Keystroke Handling	Timeout Behavior
PREFIX_INACTIVE	Normal input mode, keys route to active pane	Standard shell input forwarding	No timeout active
PREFIX_DETECTED	Prefix key pressed, awaiting command key	Interpret as multiplexer command	Reset to inactive after 500ms
PREFIX_CONSUMING	Processing multi-key command sequence	Continue command sequence parsing	Extend timeout for sequence completion
PREFIX_INVALID	Invalid command sequence detected	Return to normal mode, may echo error	Reset immediately to inactive

The prefix key detection logic must handle several edge cases that commonly trip up implementations. First, the prefix key combination (such as Ctrl-b) generates a specific byte sequence that must be distinguished from other control sequences. The implementation cannot simply check for the ASCII value 2 (Ctrl-b) because this same byte might appear as part of a legitimate escape sequence being sent to a shell program.

Decision: Prefix Key Timeout Mechanism

- **Context:** After detecting a prefix key, the system must decide how long to wait for the subsequent command key before reverting to normal mode
- **Options Considered:** No timeout (wait indefinitely), short timeout (100-200ms), medium timeout (500ms), long timeout (1-2 seconds)
- **Decision:** 500ms timeout with configurable override
- **Rationale:** 500ms provides enough time for deliberate command entry without leaving the system in command mode if user accidentally hits prefix key. Configurable override allows users with different typing speeds to adjust
- **Consequences:** Reduces frustration from accidental prefix key presses while maintaining responsive command entry for intentional use

Command Key Mapping

Once the prefix key activates command mode, the subsequent keystroke is interpreted according to a **command key mapping table**. This table defines the relationship between individual key presses and multiplexer actions,

providing the vocabulary for user interaction with the system.

Command Key	Action	Parameters	Description
h	switch_pane	direction: LEFT	Switch focus to pane on the left
j	switch_pane	direction: DOWN	Switch focus to pane below
k	switch_pane	direction: UP	Switch focus to pane above
l	switch_pane	direction: RIGHT	Switch focus to pane on the right
"	split_pane	type: HORIZONTAL	Create horizontal split of current pane
%	split_pane	type: VERTICAL	Create vertical split of current pane
x	close_pane	confirm: true	Close current pane with confirmation
c	create_window	shell: default	Create new window with fresh shell
n	next_window	wrap: true	Switch to next window, wrapping to first
p	prev_window	wrap: true	Switch to previous window, wrapping to last
:	command_prompt	mode: EXTENDED	Enter extended command prompt mode

The command mapping system supports both **immediate commands** and **extended commands**. Immediate commands execute directly upon pressing the command key, providing fast access to common operations like pane switching and splitting. Extended commands, triggered by keys like ':', enter a sub-mode where the user can type longer command sequences with arguments.

Multi-Key Command Sequences

Some multiplexer operations require more complex input than a single command key can provide. **Multi-key command sequences** handle operations that need additional parameters or confirmation, such as pane resizing with specific dimensions or window renaming with custom text.

The multi-key system extends the prefix state machine to handle **command continuation states**. When a command key indicates that additional input is required, the Input Handler transitions to a specialized parsing mode that interprets subsequent keystrokes according to the specific command's requirements.

Command Sequence	State Progression	Input Interpretation	Completion Trigger
Resize pane	PREFIX_DETECTED → RESIZE_MODE	Arrow keys adjust dimensions	Enter key or timeout
Rename window	PREFIX_DETECTED → TEXT_INPUT	Alphanumeric characters build name	Enter key or Escape
Goto window	PREFIX_DETECTED → NUMBER_INPUT	Digits build window index	Enter key or invalid digit
Copy mode	PREFIX_DETECTED → COPY_MODE	Vi-style navigation commands	'q' key or Escape

Multi-key sequences require **input validation** at each step to provide immediate feedback when users enter invalid commands or parameters. The validation logic must distinguish between incomplete input (where more keystrokes are expected) and invalid input (where the current sequence cannot be completed).

Decision: Command Key Modifier Support

- **Context:** Users may want to use modifier keys (Ctrl, Alt, Shift) in combination with command keys to expand the available command vocabulary
- **Options Considered:** No modifiers (single keys only), Ctrl combinations, Alt combinations, all modifier combinations
- **Decision:** Support Ctrl combinations for extended commands, avoid Alt due to terminal compatibility issues
- **Rationale:** Ctrl combinations are reliably detected across terminal types and provide sufficient expansion of command space. Alt combinations often conflict with terminal emulator menu shortcuts
- **Consequences:** Provides extended command vocabulary while maintaining broad terminal compatibility, but limits total available commands compared to supporting all modifiers

Raw Terminal Mode Management

Raw terminal mode represents one of the most critical and error-prone aspects of terminal multiplexer implementation. In normal terminal operation, the terminal driver performs significant input and output processing, including line buffering, echo handling, and signal generation for control characters like Ctrl-C. A terminal multiplexer must bypass this processing to gain direct access to all keyboard input, including control sequences and special keys.

Terminal Mode Transitions

The transition between normal and raw terminal modes involves manipulating the **terminal attributes** through the `termios` interface. These attributes control how the terminal driver processes input and output, determining whether characters are buffered, echoed, or interpreted as signals.

The `terminal_state_t` structure maintains the information necessary to safely transition between modes and restore the original terminal configuration when the multiplexer exits.

Terminal Attribute	Normal Mode Setting	Raw Mode Setting	Purpose
ECHO	Enabled	Disabled	Prevents input characters from appearing twice
ICANON	Enabled	Disabled	Disables line buffering for immediate character access
ISIG	Enabled	Disabled	Prevents Ctrl-C, Ctrl-Z from generating signals
INPCK	Varies	Disabled	Disables input parity checking
ISTRIP	Varies	Disabled	Preserves 8th bit of input characters
INLCR	Varies	Disabled	Prevents newline translation
IGNCR	Varies	Disabled	Preserves carriage return characters
ICRNL	Enabled	Disabled	Prevents CR to NL translation
IXON	Enabled	Disabled	Disables software flow control
OPOST	Enabled	Disabled	Disables output processing

The mode transition process follows a specific sequence to ensure atomicity and proper error handling:

1. **Save original terminal attributes** using `tcgetattr()` to capture the current terminal configuration before any modifications
2. **Validate terminal file descriptor** to ensure the multiplexer is running on an actual terminal rather than a pipe or regular file
3. **Prepare raw mode attributes** by copying the original attributes and modifying specific flags for raw mode operation
4. **Apply raw mode atomically** using `tcsetattr()` with `TCSAFLUSH` to ensure all pending output is transmitted before the change
5. **Verify raw mode activation** by reading back the terminal attributes and confirming the changes took effect
6. **Register cleanup handlers** to ensure terminal restoration occurs even if the multiplexer exits unexpectedly

Signal Handling in Raw Mode

Raw terminal mode disables the automatic signal generation normally triggered by control characters, making the multiplexer responsible for **explicit signal handling**. This responsibility includes both managing signals directed at the multiplexer itself and properly forwarding signals to the child shell processes running in individual panes.

Signal	Source	Normal Terminal Action	Raw Mode Requirement
SIGINT	Ctrl-C	Interrupt foreground process group	Forward to active pane's process group
SIGTSTP	Ctrl-Z	Suspend foreground process group	Forward to active pane's process group
SIGQUIT	Ctrl-\	Quit foreground process group	Forward to active pane's process group
SIGWINCH	Terminal resize	Update terminal size	Update all pane PTY sizes
SIGTERM	External termination	Exit process	Restore terminal and exit cleanly
SIGCHLD	Child process exit	Default handling	Clean up terminated pane sessions

The signal forwarding mechanism requires careful **process group management** to ensure signals reach the appropriate shell processes. Each PTY session establishes its own process group, with the shell process as the group leader. When forwarding signals, the multiplexer must send them to the process group rather than individual processes to ensure proper signal propagation to any child processes spawned by the shell.

Decision: Terminal Restoration Strategy

- Context:** The multiplexer must restore normal terminal mode when exiting, but exit can occur through multiple paths (normal shutdown, signals, crashes)
- Options Considered:** Atexit handlers only, signal handlers only, both atexit and signal handlers, cleanup in main function only
- Decision:** Both atexit handlers and signal handlers with cleanup state tracking
- Rationale:** Multiple exit paths require multiple restoration mechanisms. Atexit handles normal exits, signal handlers manage interruption, state tracking prevents double-restoration
- Consequences:** Robust terminal restoration across all exit scenarios, but requires careful state management to avoid conflicts between restoration mechanisms

Input Processing Pipeline

In raw terminal mode, the Input Handler receives **unprocessed byte streams** directly from the terminal device. These byte streams may contain single-byte characters, multi-byte UTF-8 sequences, or complex escape sequences representing special keys like arrow keys or function keys.

The input processing pipeline transforms these raw bytes into **structured input events** that the rest of the multiplexer can process uniformly:

- Byte stream collection** accumulates incoming bytes from the terminal file descriptor using non-blocking reads to avoid stalling the event loop
- UTF-8 sequence assembly** identifies multi-byte Unicode characters and buffers incomplete sequences until all bytes arrive
- Escape sequence detection** recognizes special key combinations that generate multi-byte sequences like `\e[A` for the up arrow key

4. **Input event generation** converts completed character or key sequences into structured events with metadata about modifier keys and special key types
5. **Context-sensitive routing** directs events to either the command processor (in command mode) or the active pane's PTY (in normal mode)

The pipeline must handle **partial sequences** gracefully, as network delays or system load can cause multi-byte sequences to arrive across multiple read operations. The implementation maintains intermediate buffers for UTF-8 assembly and escape sequence parsing to reconstruct complete sequences from fragmented input.

Input Type	Byte Sequence Example	Processing Required	Output Event
ASCII character	0x41 ('A')	Direct conversion	Character event with 'A'
UTF-8 character	0xC3 0xA9 ('é')	Multi-byte assembly	Character event with Unicode codepoint
Arrow key	0x1B 0x5B 0x41 (up arrow)	Escape sequence parsing	Special key event UP_ARROW
Function key	0x1B 0x4F 0x50 (F1)	Escape sequence parsing	Special key event F1
Modified key	0x1B 0x5B 0x31 0x3B 0x32 0x41 (Shift+up)	Complex escape parsing	Special key event UP_ARROW with SHIFT modifier

Architecture Decision Records

The Input Handler design involves several critical decisions that significantly impact the user experience and system robustness. Each decision represents a trade-off between competing concerns such as performance, compatibility, ease of implementation, and feature richness.

Decision: Event Loop Integration Strategy

- **Context:** The Input Handler must integrate with the main event loop that also handles PTY I/O and terminal rendering, requiring coordination between multiple input sources
- **Options Considered:** Separate input thread with queues, polling-based integration with `select()`, signal-driven I/O, dedicated input process
- **Decision:** Polling-based integration using `select()` with the main event loop
- **Rationale:** `Select()`-based polling provides deterministic behavior and integrates cleanly with existing PTY file descriptor monitoring. Threading introduces complexity with shared state, while signal-driven I/O has portability issues
- **Consequences:** Simplifies concurrency management and ensures consistent event processing order, but requires careful timeout management to maintain input responsiveness

Option	Pros	Cons
Separate thread	Responsive input handling, parallel processing	Complex synchronization, shared state issues
Select() polling	Simple integration, deterministic ordering	Potential input latency with heavy PTY I/O
Signal-driven I/O	Low latency, event-driven architecture	Platform-specific behavior, signal handling complexity
Dedicated process	Complete isolation, crash resilience	IPC overhead, increased memory usage

Decision: Prefix Key Collision Handling

- Context:** The chosen prefix key (Ctrl-b) might be used by shell programs or text editors running within panes, requiring a mechanism to send the literal prefix key to applications
- Options Considered:** Double prefix key (Ctrl-b Ctrl-b), escape sequence (Ctrl-b \), alternative prefix key only, user-configurable prefix
- Decision:** Double prefix key sends literal, with user-configurable prefix support
- Rationale:** Double prefix matches tmux convention and provides intuitive escape mechanism. User configuration allows adaptation to different workflows without breaking muscle memory
- Consequences:** Familiar behavior for tmux users and flexibility for customization, but requires additional state tracking for double-key detection

Option	Pros	Cons
Double prefix	Intuitive, matches tmux convention	Requires sequence detection logic
Escape sequence	Clear visual indicator	Non-standard, harder to remember
Alternative prefix only	Simple implementation	Limited flexibility for complex cases
User-configurable	Maximum flexibility	Configuration complexity

Decision: Command Mode Timeout Mechanism

- **Context:** After entering command mode with the prefix key, the system must determine how long to wait for command input before reverting to normal mode
- **Options Considered:** No timeout (wait forever), fixed short timeout (200ms), fixed medium timeout (500ms), adaptive timeout based on typing speed
- **Decision:** Fixed 500ms timeout with configuration override
- **Rationale:** 500ms balances responsiveness with usability - long enough for deliberate command entry but short enough to recover quickly from accidental prefix key presses. Configuration override accommodates different user preferences
- **Consequences:** Predictable behavior that works well for most users, with customization available for edge cases, but requires timeout management in the event loop

Option	Pros	Cons
No timeout	Never leaves user stranded	Accidental prefix key locks input mode
200ms timeout	Very responsive recovery	Too fast for deliberate command entry
500ms timeout	Good balance of speed and usability	May feel slow for fast typists
Adaptive timeout	Optimizes for individual users	Complex implementation, unpredictable behavior

Decision: Terminal Restoration Error Handling

- **Context:** If terminal restoration fails during multiplexer exit (due to file descriptor closure or system errors), the terminal may be left in an unusable state
- **Options Considered:** Ignore restoration failures, retry with exponential backoff, external cleanup script, store restoration commands for external execution
- **Decision:** Retry with limited attempts plus external script fallback
- **Rationale:** Multiple restoration attempts handle temporary system issues, while external script provides recovery mechanism for persistent failures. User can manually run script if needed
- **Consequences:** Robust recovery from most failure scenarios with fallback mechanism, but adds complexity to cleanup logic and requires external script management

Option	Pros	Cons
Ignore failures	Simple implementation	May leave terminal unusable
Retry with backoff	Handles temporary failures	May delay exit unnecessarily
External script	Always provides recovery option	Requires separate file management
Store commands	Flexible external execution	Complex implementation

Common Input Handling Pitfalls

Input handling represents one of the most error-prone areas of terminal multiplexer development due to the low-level nature of terminal interaction and the variety of edge cases that must be handled correctly. Understanding these common pitfalls and their solutions is essential for building a robust multiplexer.

⚠ Pitfall: Incomplete Terminal Restoration

Many implementations fail to properly restore terminal attributes when the multiplexer exits, leaving the terminal in raw mode. This makes the terminal unusable because input characters are no longer echoed and line editing features don't work.

The root cause typically involves missing restoration in signal handlers or failing to handle unexpected exit paths. When a user presses Ctrl-C or the system sends SIGTERM, the multiplexer may exit without executing normal cleanup code. The terminal remains in raw mode because no code ran to restore the original `termios` attributes.

Detection: After multiplexer exit, typing characters in the terminal produces no visible output, and pressing Enter doesn't execute commands.

Solution: Implement restoration in multiple places: normal exit paths, signal handlers, and `atexit()` registered functions. Use a global flag to track whether restoration has already occurred to prevent double-restoration issues.

```
// Implementation Guidance will show the complete restoration pattern
```

C

⚠ Pitfall: Signal Forwarding to Wrong Process Group

A common error involves sending signals directly to individual shell processes rather than their process groups. This causes problems when shells spawn child processes (like running `vim` or `make`), because the signals only reach the shell itself, not the currently running foreground job.

This happens because developers often store only the shell's process ID from the `fork()` call and use it directly with `kill()`. However, terminals normally send signals to process groups using negative PIDs, ensuring all related processes receive the signal.

Detection: Pressing Ctrl-C in a pane only interrupts the shell but not programs running within the shell. For example, a long-running `grep` command continues running even after Ctrl-C.

Solution: Use negative PIDs with `kill()` to send signals to process groups, and ensure each PTY session establishes its own process group using `setsid()` in the child process.

⚠ Pitfall: UTF-8 Sequence Fragmentation

Raw terminal input can fragment multi-byte UTF-8 characters across multiple read operations, especially under system load or with network-based terminals. Implementations that process bytes individually without buffering incomplete sequences will misinterpret UTF-8 characters as multiple invalid single-byte characters.

This creates visual corruption when users type non-ASCII characters, and can desynchronize the input parser when UTF-8 fragments are misinterpreted as escape sequence components.

Detection: Non-ASCII characters appear as question marks or random symbols, especially when typing quickly or during system load.

Solution: Implement UTF-8 sequence buffering that accumulates bytes until a complete Unicode codepoint is assembled. The parser should track the expected length of multi-byte sequences and defer processing until all bytes arrive.

⚠ Pitfall: Escape Sequence State Confusion

Input parsers often fail to handle escape sequences that arrive fragmented or interleaved with other input. For example, the up arrow key generates the sequence `\e[A`, but this might arrive as separate reads of `\e`, `[`, and `A`. If other input arrives between these bytes, the parser can lose synchronization.

Additionally, some applications send escape sequences that the multiplexer doesn't recognize, leading to parser state corruption where subsequent input is misinterpreted.

Detection: Special keys like arrows occasionally don't work, or typing immediately after pressing special keys produces incorrect characters in the shell.

Solution: Implement a robust state machine with timeout-based recovery. If an escape sequence doesn't complete within a reasonable time (typically 50-100ms), treat accumulated bytes as separate characters and reset parser state.

⚠ Pitfall: Command Mode State Leakage

Implementations sometimes fail to properly exit command mode when errors occur or when users press unexpected keys. This leaves the multiplexer stuck in command mode, where all subsequent input is interpreted as multiplexer commands rather than shell input.

This typically happens when the command key handler encounters an invalid command but doesn't reset the prefix state, or when multi-key sequences are interrupted by unexpected input.

Detection: After pressing the prefix key and making a mistake, all subsequent typing is ignored by the shell, and characters like 'h', 'j', 'k', 'l' switch between panes instead of appearing as shell input.

Solution: Implement explicit state reset logic in all command processing paths, including error paths and timeout handlers. Any unrecognized command should immediately return to normal mode with appropriate error feedback.

⚠ Pitfall: Terminal Size Propagation Failures

When terminal window resizing occurs, the multiplexer must update the PTY sizes for all sessions to reflect the new dimensions. Implementations often miss this step or update only the currently active pane, causing programs in other panes to have incorrect assumptions about terminal size.

This creates display corruption in programs that use full-screen interfaces like `vim`, `less`, or `htop`, which rely on accurate terminal dimensions for screen layout.

Detection: After resizing the terminal window, full-screen programs in inactive panes have incorrect display layout with text wrapping at wrong positions or status bars in wrong locations.

Solution: Handle `SIGWINCH` signals by updating PTY dimensions for all sessions using the `TIOCSWINSZ` ioctl, not just the active pane. Also send `SIGWINCH` to all child process groups to notify programs of the size change.

Implementation Guidance

The Input Handler bridges the gap between raw terminal input and structured multiplexer commands, requiring careful integration of several low-level Unix interfaces. This implementation guidance provides complete starter code for terminal mode management and skeleton code for the core input processing logic.

Technology Recommendations

Component	Simple Option	Advanced Option
Terminal Control	<code>termios.h</code> with basic flags	<code>termios.h</code> with full attribute management
Signal Handling	Basic <code>signal()</code> registration	Advanced <code>sigaction()</code> with masks
Input Parsing	Character-by-character state machine	Buffered parsing with lookahead
UTF-8 Support	Basic ASCII-only implementation	Full Unicode with <code>wchar.h</code> support
Key Bindings	Static array of command mappings	Dynamic hash table with user configuration

Recommended File Structure

```
src/                                C

  input/
    input_handler.h      ← public interface and types
    input_handler.c      ← main input processing logic
    terminal_mode.c     ← raw mode management
    key_bindings.c       ← command key mapping and prefix handling
    utf8_decoder.c       ← UTF-8 sequence processing
    escape_parser.c       ← escape sequence state machine
    multiplexer.c        ← main event loop integration

  tests/
    test_input_handler.c ← unit tests for input processing
    test_terminal_mode.c ← terminal mode transition tests
```

Complete Terminal Mode Management Code

This infrastructure handles the critical terminal mode transitions and restoration logic. Copy this code directly into your project:

```
// terminal_mode.h
```

C

```
#ifndef TERMINAL_MODE_H
```

```
#define TERMINAL_MODE_H
```

```
#include <termios.h>
```

```
#include <stdbool.h>
```

```
typedef struct {
```

```
    struct termios original_termios;
```

```
    bool raw_mode_active;
```

```
    bool restoration_registered;
```

```
} terminal_state_t;
```

```
// Initialize terminal state structure
```

```
int terminal_state_init(terminal_state_t* state);
```

```
// Enter raw terminal mode, saving original state
```

```
int terminal_enter_raw_mode(terminal_state_t* state);
```

```
// Restore original terminal attributes
```

```
int terminal_restore_mode(terminal_state_t* state);
```

```
// Get current terminal window size
```

```
int terminal_get_size(int* width, int* height);
```

```
// Register cleanup handlers for safe restoration
```

```
int terminal_register_cleanup(terminal_state_t* state);
```

```
#endif
```

```
// terminal_mode.c
```

```
#include "terminal_mode.h"
```

```
#include <unistd.h>

#include <sys/ioctl.h>

#include <signal.h>

#include <stdlib.h>

#include <errno.h>

static terminal_state_t* global_terminal_state = NULL;

// Signal handler for cleanup on exit

static void cleanup_handler(int sig) {

    if (global_terminal_state && global_terminal_state->raw_mode_active) {

        tcsetattr(STDIN_FILENO, TCSAFLUSH, &global_terminal_state->original_termios);

        global_terminal_state->raw_mode_active = false;

    }

    if (sig != 0) {

        exit(128 + sig); // Standard exit code for signal termination

    }

}

// Atexit handler for normal program termination

static void atexit_handler(void) {

    cleanup_handler(0);

}

int terminal_state_init(terminal_state_t* state) {

    if (!state) return -1;

    state->raw_mode_active = false;

    state->restoration_registered = false;
```

```
// Verify we're running on a terminal

if (!isatty(STDIN_FILENO)) {

    errno = ENOTTY;

    return -1;

}

return 0;
}

int terminal_enter_raw_mode(terminal_state_t* state) {

    if (!state || state->raw_mode_active) return -1;

    // Save original terminal attributes

    if (tcgetattr(STDIN_FILENO, &state->original_termios) != 0) {

        return -1;

    }

    // Prepare raw mode attributes

    struct termios raw = state->original_termios;

    // Input flags: disable break signal, CR-to-NL translation, parity check,
    // 8th bit stripping, and flow control

    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    // Output flags: disable output processing

    raw.c_oflag &= ~(OPOST);
```

```
// Control flags: set 8-bit character size

raw.c_cflag |= (CS8);

// Local flags: disable echo, canonical mode, extended functions, and signals

raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

// Set read timeouts: return immediately if no data available

raw.c_cc[VMIN] = 0;

raw.c_cc[VTIME] = 0;

// Apply raw mode atomically

if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) != 0) {

    return -1;

}

state->raw_mode_active = true;

return 0;

}

int terminal_restore_mode(terminal_state_t* state) {

    if (!state || !state->raw_mode_active) return 0;

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &state->original_termios) != 0) {

        return -1;

    }

    state->raw_mode_active = false;

    return 0;

}
```

```
}

int terminal_get_size(int* width, int* height) {

    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) != 0) {

        return -1;
    }

    if (width) *width = ws.ws_col;

    if (height) *height = ws.ws_row;

    return 0;
}

int terminal_register_cleanup(terminal_state_t* state) {

    if (!state || state->restoration_registered) return 0;

    global_terminal_state = state;

    // Register signal handlers for cleanup

    struct sigaction sa;

    sa.sa_handler = cleanup_handler;

    sigemptyset(&sa.sa_mask);

    sa.sa_flags = 0;

    if (sigaction(SIGINT, &sa, NULL) != 0 ||
        sigaction(SIGTERM, &sa, NULL) != 0 ||
        sigaction(SIGQUIT, &sa, NULL) != 0) {
```

```
    return -1;

}

// Register atexit handler

if (atexit(atexit_handler) != 0) {

    return -1;

}

state->restoration_registered = true;

return 0;

}
```

Input Handler Core Logic Skeleton

The following skeleton provides the structure for implementing input processing logic. Fill in the TODO comments with your implementation:

```
// input_handler.h

#ifndef INPUT_HANDLER_H

#define INPUT_HANDLER_H

#include "terminal_mode.h"

#include "../window/window_manager.h"

#include <stdint.h>

#include <stdbool.h>

typedef enum {

    PREFIX_INACTIVE,

    PREFIX_DETECTED,

    PREFIX_CONSUMING,

    PREFIX_INVALID

} prefix_state_t;

typedef enum {

    INPUT_CHARACTER,

    INPUT_SPECIAL_KEY,

    INPUT_COMMAND

} input_event_type_t;

typedef struct {

    input_event_type_t type;

    uint32_t codepoint;      // For character input

    int special_key;        // For special keys (arrows, function keys)

    int modifiers;          // Modifier key flags

    char command[32];       // For command input

} input_event_t;
```

C

```
typedef struct {

    terminal_state_t terminal_state;

    prefix_state_t prefix_state;

    uint64_t prefix_timestamp; // For timeout tracking

    window_manager_t* window_manager;

    // Input buffering for UTF-8 and escape sequences

    char input_buffer[256];

    int buffer_length;

    // UTF-8 decoding state

    int utf8_bytes_remaining;

    uint32_t utf8_codepoint;

    // Escape sequence parsing state

    parser_state_t escape_state;

    char escape_buffer[ESC_SEQ_BUFFER_SIZE];

    int escape_length;

} input_handler_t;

// Initialize input handler with terminal mode setup

int input_handler_init(input_handler_t* handler, window_manager_t* wm);

// Process input from stdin, returns number of events generated

int input_handler_process_input(input_handler_t* handler, input_event_t* events, int max_events);

// Handle prefix key detection and command mode

int input_handler_handle_prefix_key(input_handler_t* handler);
```

```
// Route input event to appropriate destination

int input_handler_route_event(input_handler_t* handler, const input_event_t* event);

// Cleanup and restore terminal mode

int input_handler_cleanup(input_handler_t* handler);

#endif

// input_handler.c

#include "input_handler.h"

#include <unistd.h>

#include <sys/time.h>

#include <string.h>

#define PREFIX_KEY_CTRL_B 0x02

#define PREFIX_TIMEOUT_MS 500

int input_handler_init(input_handler_t* handler, window_manager_t* wm) {

    if (!handler || !wm) return -1;

    memset(handler, 0, sizeof(input_handler_t));

    handler->window_manager = wm;

    handler->prefix_state = PREFIX_INACTIVE;

    handler->escape_state = NORMAL;

    // TODO 1: Initialize terminal state structure

    // TODO 2: Register terminal cleanup handlers

    // TODO 3: Enter raw terminal mode

    // TODO 4: Verify raw mode activation successful

    // Hint: Use terminal_state_init(), terminal_register_cleanup(), terminal_enter_raw_mode()
```

```
    return 0;

}

int input_handler_process_input(input_handler_t* handler, input_event_t* events, int max_events) {

    if (!handler || !events || max_events <= 0) return -1;

    char buffer[256];

    ssize_t bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer));

    if (bytes_read <= 0) return 0;

    int event_count = 0;

    for (int i = 0; i < bytes_read && event_count < max_events; i++) {

        uint8_t byte = buffer[i];

        // TODO 1: Check if current prefix state has timed out

        // TODO 2: Handle prefix key detection (typically Ctrl-B)

        // TODO 3: Route byte through UTF-8 decoder if needed

        // TODO 4: Route byte through escape sequence parser if needed

        // TODO 5: Generate appropriate input event based on parser results

        // TODO 6: Handle command mode vs normal mode routing

        // Hint: Use struct timeval and gettimeofday() for timeout checking

        // Hint: UTF-8 sequences start with bytes >= 0x80

        // Hint: Escape sequences start with byte 0x1B

    }

}
```

```
    return event_count;
}

int input_handler_handle_prefix_key(input_handler_t* handler) {
    if (!handler) return -1;

    switch (handler->prefix_state) {
        case PREFIX_INACTIVE:
            // TODO 1: Set state to PREFIX_DETECTED
            // TODO 2: Record current timestamp for timeout
            // TODO 3: Return indication that prefix was consumed
            break;

        case PREFIX_DETECTED:
            // TODO 1: Check if this is a double prefix key (send literal)
            // TODO 2: Otherwise interpret as command key
            // TODO 3: Update state based on command requirements
            // TODO 4: Execute immediate commands or enter multi-key mode
            break;

        case PREFIX_CONSUMING:
            // TODO 1: Continue multi-key command sequence
            // TODO 2: Check for sequence completion or timeout
            // TODO 3: Execute completed command or return to inactive
            break;

        case PREFIX_INVALID:
    }
}
```

```
        // TODO 1: Reset to inactive state immediately

        // TODO 2: Optionally display error message

        break;

    }

    return 0;
}

int input_handler_route_event(input_handler_t* handler, const input_event_t* event) {

    if (!handler || !event) return -1;

    // TODO 1: Check current prefix state to determine routing

    // TODO 2: If in command mode, route to command processor

    // TODO 3: If in normal mode, route to active pane PTY

    // TODO 4: Handle special cases like window switching

    // TODO 5: Update pane focus and redraw as needed

    // Hint: Use window_manager_get_active_pane() to get current pane

    // Hint: Use pty_session_write() to send input to shell

    return 0;
}
```

Key Binding Configuration Structure

```
// key_bindings.h

typedef struct {

    char key;

    int modifiers;

    char command[32];

    char description[64];

} key_binding_t;

// Default key bindings (customize as needed)

static const key_binding_t default_bindings[] = {

    {'h', 0, "switch-left", "Switch to pane on left"},

    {'j', 0, "switch-down", "Switch to pane below"},

    {'k', 0, "switch-up", "Switch to pane above"},

    {'l', 0, "switch-right", "Switch to pane on right"},

    {'"', 0, "split-horizontal", "Split pane horizontally"},

    {'%', 0, "split-vertical", "Split pane vertically"},

    {'x', 0, "close-pane", "Close current pane"},

    {'c', 0, "new-window", "Create new window"},

    {':', 0, "command-prompt", "Enter command prompt"},

    {0, 0, "", ""} // Terminator

};

// TODO: Implement key_bindings_lookup() function

// TODO: Implement key_bindings_execute() function
```

Milestone Checkpoints

After implementing the Input Handler, verify correct operation with these checkpoints:

Basic Input Routing Test:

BASH

Prefix Key Test:

BASH

Terminal Restoration Test:

BASH

Expected Behaviors:

- Normal typing appears in active pane's shell
 - Prefix key followed by commands controls multiplexer
 - Ctrl-C, Ctrl-Z work correctly in shell programs
 - Terminal always restores to normal mode on exit
 - UTF-8 characters display correctly (test with: echo "héllø wørld")

Language-Specific Implementation Hints

Terminal Control:

- Use `tcgetattr()` and `tcsetattr()` for terminal mode management
- Always use `TCSAFLUSH` flag to ensure clean state transitions
- Check `isatty()` before attempting terminal operations

Signal Handling:

- Use `sigaction()` instead of `signal()` for reliable behavior
- Block signals during critical sections with `sigprocmask()`
- Remember that `SIGCHLD` needs special handling for PTY cleanup

UTF-8 Processing:

- First byte determines sequence length: 0x80-0xBF (invalid), 0xC0-0xDF (2 bytes), 0xE0-0xEF (3 bytes), 0xF0-0xF7 (4 bytes)
- Continuation bytes always in range 0x80-0xBF
- Use bit shifting to assemble multi-byte codepoints

Timing and Timeouts:

- Use `gettimeofday()` or `clock_gettime()` for high-resolution timestamps
- Convert timeout milliseconds to microseconds for comparison
- Handle timer wraparound in long-running processes

Implementation Guidance

Debugging Input Handler Issues

Symptom	Likely Cause	Diagnosis	Fix
No input echo after exit	Terminal restoration failed	Run <code>stty sane</code> to check	Add signal handlers and atexit cleanup
Special keys don't work	Escape sequence parsing broken	Log raw input bytes	Fix state machine transitions
UTF-8 characters corrupted	Multi-byte sequence fragmentation	Check for partial reads	Buffer incomplete sequences
Prefix key doesn't work	Raw mode not active	Check terminal flags with <code>stty -a</code>	Verify raw mode setup
Commands execute in shell	Input routing incorrect	Trace prefix state changes	Fix command mode detection

Interactions and Data Flow

Milestone(s): All milestones (1-4) - this section describes how components orchestrate together to create a functioning terminal multiplexer, building on PTY management (M1), terminal emulation (M2), window management (M3), and input handling (M4)

Mental Model: The Orchestra Conductor

Think of the terminal multiplexer's main event loop as an orchestra conductor coordinating multiple musicians. Just as a conductor listens for cues from different sections and coordinates their timing to create harmonious music, the event loop listens for input from multiple sources (PTY sessions, user keyboard, terminal resize signals) and orchestrates their processing to create a seamless user experience. The conductor doesn't play every instrument—instead, it routes information between specialists (PTY manager for process communication, terminal emulator for display rendering, window manager for layout) while maintaining perfect timing and synchronization.

The key insight is that a terminal multiplexer is fundamentally an **event-driven system** where multiple asynchronous data sources must be coordinated without blocking. Unlike a simple shell that processes one command at a time, our multiplexer must simultaneously handle output from multiple shell sessions, user input, terminal resize events, and child process termination—all while maintaining responsive user interaction and consistent display state.

Main Event Loop

The **main event loop** forms the beating heart of the terminal multiplexer, using `select()`-based I/O multiplexing to handle multiple file descriptors simultaneously without blocking. This approach allows the multiplexer to remain responsive to user input while processing output from multiple PTY sessions and handling system events.

The event loop operates on a **unified file descriptor set** that includes the standard input for user keyboard input, all active PTY master file descriptors for shell session output, and a signal file descriptor for handling terminal resize and child process termination events. The `select()` system call blocks until at least one file descriptor becomes ready for reading, at which point the loop processes all ready descriptors before returning to wait for the next events.

Event Loop Architecture

The event loop maintains several critical data structures to track active file descriptors and their associated processing handlers. The main loop uses a `fd_set` structure for the `select()` call, with parallel arrays mapping file descriptor numbers to their corresponding pane identifiers and processing functions. This design enables efficient dispatch of I/O events to the appropriate component handlers without linear searches or complex lookup structures.

Component	File Descriptors	Event Types	Processing Handler
User Input	STDIN_FILENO (0)	Keyboard input, special keys	<code>input_handler_process_input()</code>
PTY Sessions	master_fd for each pane	Shell output, process termination	<code>pty_session_read() → escape_parser_process_byte()</code>
Signal Handler	signalfd or pipe	SIGCHLD, SIGWINCH	Signal dispatch to appropriate component
Timer Events	timerfd (optional)	Prefix key timeout, rendering throttle	Timeout-based state transitions

Event Processing Pipeline

The event loop follows a structured processing pipeline that ensures consistent ordering and prevents race conditions between different event types. The pipeline prioritizes signal events (like child process termination) over I/O events to ensure system state remains consistent before processing new data.

- 1. Signal Event Processing:** Handle `SIGCHLD` for child process termination and `SIGWINCH` for terminal resize events before processing any I/O, ensuring system state consistency.
- 2. User Input Processing:** Process keyboard input from stdin, routing through the input handler's prefix key detection and command mode logic to determine routing destination.
- 3. PTY Output Processing:** Read data from all ready PTY master file descriptors, feeding the raw bytes through the terminal emulator's escape sequence parser to update screen buffers.
- 4. Screen Composition:** After processing all input events, determine if any panes need redrawing based on their `needs_redraw` flags and compose the final terminal output.
- 5. Terminal Output:** Write the composed frame to stdout using a single write operation to minimize terminal flickering and ensure atomic display updates.

Design Insight: The event loop processes ALL ready file descriptors in each iteration before composing output, rather than rendering after each individual PTY read. This batching approach prevents excessive screen updates when multiple panes have simultaneous output, significantly improving performance and reducing visual artifacts.

File Descriptor Management

The event loop must dynamically manage its file descriptor set as panes are created and destroyed during the multiplexer session. When a new pane is created, its PTY master file descriptor is added to the `select()` set and the corresponding mapping tables are updated. When a pane is closed, the file descriptor is removed from the set and its entry in the mapping tables is cleared.

The event loop maintains a **maximum file descriptor tracker** to optimize the `select()` call, since `select()` requires specifying the highest-numbered file descriptor plus one. This value is recalculated whenever file descriptors are added or removed, ensuring efficient operation even with sparse file descriptor numbers.

```
typedef struct {

    fd_set read_fds;

    int max_fd;

    int pane_fd_map[MAX_PANES];      // Maps fd to pane_id

    int fd_pane_map[FD_SETSIZE];     // Maps pane_id to fd

    window_manager_t* window_manager;

    input_handler_t* input_handler;

    bool shutdown_requested;

} event_loop_t;
```

Error Handling and Recovery

The event loop implements robust error handling for file descriptor failures, including `EAGAIN` for temporary unavailability, `EBADF` for invalid file descriptors (indicating closed PTY sessions), and `EINTR` for interrupted system calls. When a PTY file descriptor becomes invalid, the event loop automatically closes the corresponding pane and updates the layout tree.

Error Recovery Pipeline:

Error Condition	Detection Method	Recovery Action	State Updates
PTY Session Death	<code>read()</code> returns 0 or <code>EBADF</code>	Remove fd from select set, close pane	Update layout tree, switch focus if needed
Signal Interruption	<code>select()</code> returns -1 with <code>EINTR</code>	Continue loop without processing I/O	None - retry <code>select()</code>
Memory Allocation	Buffer allocation failure	Log error, attempt graceful shutdown	Close all PTY sessions, restore terminal
Terminal Write Error	<code>write()</code> to stdout fails	Attempt recovery, fallback to basic output	Switch to simplified rendering mode

Architecture Decision: Single-Threaded Event Loop

- **Context:** Need to coordinate multiple asynchronous I/O sources while maintaining consistent state
- **Options Considered:**
 1. Single-threaded select()-based loop
 2. Multi-threaded with per-pane threads
 3. Asynchronous I/O with callbacks
- **Decision:** Single-threaded select()-based event loop
- **Rationale:** Eliminates complex synchronization, ensures deterministic event ordering, simplifies debugging, and matches the architecture of proven multiplexers like tmux
- **Consequences:** All processing must be non-blocking, but provides predictable behavior and easier state management

Inter-Component Communication

The flow of data between components follows a structured pipeline that transforms raw PTY output into composed terminal display while routing user input to the appropriate destinations. Understanding this data flow is crucial for implementing consistent behavior and debugging issues that span multiple components.

PTY Output Data Flow

When a shell process writes output to its PTY slave, the data becomes available for reading on the PTY master file descriptor. The event loop detects this availability and initiates a processing chain that transforms the raw bytes into display updates.

Output Processing Pipeline:

1. **PTY Data Reception:** The `pty_session_read()` function reads raw bytes from the PTY master file descriptor, handling partial reads and buffering incomplete data for the next iteration.
2. **Escape Sequence Parsing:** Raw bytes are fed character-by-character into the terminal emulator's `escape_parser_process_byte()` function, which maintains a state machine to identify ANSI escape sequences and plain text characters.
3. **Screen Buffer Updates:** Parsed characters and escape sequence commands are applied to the pane's `screen_buffer_t` structure, updating the virtual display state including cursor position, text attributes, and scrollback history.
4. **Redraw Flag Setting:** When the screen buffer is modified, the pane's `needs_redraw` flag is set to indicate that this pane requires re-rendering in the next display composition cycle.
5. **Frame Composition:** During the rendering phase, all panes marked for redraw are composited into the window manager's output buffer, including borders and status information.
6. **Terminal Output:** The complete frame is written to the terminal using a single `write()` system call to ensure atomic display updates and minimize flickering.

Component Interface Contracts

Each component exposes well-defined interfaces that enforce separation of concerns and enable testing of individual components. The interfaces use **data transformation contracts** where each component receives structured input, performs its specific processing, and produces structured output for the next component.

Source Component	Output Data	Target Component	Processing Function	Result Data
PTY Manager	Raw byte stream	Terminal Emulator	<code>escape_parser_process_byte()</code>	Parsed characters + commands
Terminal Emulator	Screen buffer updates	Window Manager	<code>screen_buffer_put_char()</code>	Updated virtual display state
Window Manager	Layout calculations	Rendering System	<code>window_manager_render_frame()</code>	Composed terminal output
Input Handler	Input events	PTY Manager/Window Manager	<code>input_handler_route_event()</code>	Routed commands or character data

Memory Management and Buffer Ownership

The inter-component communication design uses **clear ownership semantics** to prevent memory leaks and use-after-free errors. Each component owns its internal data structures and provides controlled access through defined interfaces, with callers responsible for providing appropriately-sized buffers for data transfer.

Buffer Ownership Rules:

- PTY Manager owns PTY session structures and provides read-only access to session metadata
- Terminal Emulator owns screen buffers and scrollback data, accepting character input through functional interfaces
- Window Manager owns layout trees and rendering buffers, accepting pane updates through defined callbacks
- Input Handler owns terminal state and input buffers, providing parsed events through callback interfaces

The design avoids shared mutable state between components, instead using **message-passing semantics** where components communicate through function calls with well-defined input and output parameters. This approach simplifies reasoning about data flow and makes the system more testable and debuggable.

Event Flow Coordination

The multiplexer implements **event flow coordination** to ensure that related events are processed in the correct order and that component state remains consistent. For example, when a pane is closed, the event must propagate through multiple components in the correct sequence.

Pane Closure Event Flow:

1. **Input Handler:** Detects close-pane command from user input and validates that the target pane exists and can be closed
2. **PTY Manager:** Terminates the child process using `pty_session_terminate()` and closes the PTY file descriptors
3. **Window Manager:** Updates the layout tree using `layout_destroy_tree()` and redistributes space to remaining panes
4. **Event Loop:** Removes the PTY file descriptor from the `select()` set and updates the file descriptor mapping tables
5. **Rendering System:** Triggers a full redraw to reflect the new layout and remove the closed pane from the display

Design Insight: Event flow coordination prevents race conditions by ensuring that component updates happen in dependency order. The PTY is closed before layout updates to prevent the event loop from attempting to read from a closed file descriptor, and layout updates complete before rendering to ensure consistent display state.

State Synchronization

Maintaining consistent state across components presents one of the most challenging aspects of terminal multiplexer implementation. The system must coordinate **distributed state** including pane focus, terminal attributes, layout dimensions, and process lifecycle while handling asynchronous events that can affect multiple components simultaneously.

Focus Management

Focus state determines which pane receives user input and affects visual rendering through focus highlighting. The focus management system must coordinate between the input handler (which routes keystrokes), window manager (which renders focus indicators), and event loop (which handles focus change commands).

The window manager maintains the **canonical focus state** through its `active_pane_id` field, with other components querying this authoritative source rather than maintaining duplicate state. Focus changes trigger a coordinated update sequence that ensures all components reflect the new focus state consistently.

Focus Change Protocol:

Step	Component	Action	State Update
1	Input Handler	Detects focus change command (arrow keys after prefix)	None - validation only
2	Window Manager	Validates target pane exists and is reachable	Updates <code>active_pane_id</code>
3	Window Manager	Marks old and new focus panes for redraw	Sets <code>needs_redraw</code> flags
4	Event Loop	Triggers rendering cycle	Coordinates screen composition
5	Rendering System	Updates focus highlighting in terminal output	Visual focus indicators

Terminal Size Synchronization

Terminal size changes affect multiple components and require careful coordination to maintain consistent state. When the user resizes their terminal window, the multiplexer must update its internal layout calculations, propagate size changes to all PTY sessions, and trigger a complete redraw with the new dimensions.

The **terminal size synchronization protocol** handles `SIGWINCH` signals by coordinating updates across all components in the correct dependency order. The window manager serves as the coordination point, ensuring that layout calculations complete before PTY size updates and that all components use consistent dimension values.

Size Change Coordination:

- Signal Detection:** Event loop receives `SIGWINCH` and calls `terminal_get_size()` to obtain new terminal dimensions
- Layout Recalculation:** Window manager updates its internal dimensions using `window_manager_resize_terminal()` and recalculates all pane positions
- PTY Size Propagation:** Each pane's PTY session is updated with its new dimensions using `pty_session_resize()` and the `TIOCSWINSZ` ioctl
- Buffer Reallocation:** Screen buffers are resized to match new pane dimensions, preserving existing content where possible
- Full Redraw:** All panes are marked for redraw and a complete screen refresh is triggered

Critical Synchronization Point: PTY size updates must complete before processing any new output from shell sessions, since shells may immediately output content formatted for the new terminal size. Processing old output with new size parameters can cause display corruption.

Process Lifecycle Coordination

Child process lifecycle events require coordination between the PTY manager (which detects process termination), window manager (which may need to close panes), and event loop (which must update file

descriptor sets). The system uses **lifecycle state machines** to ensure consistent handling of process creation, execution, and termination.

Process State Transitions:

Current State	Event	Next State	Coordinated Actions
CREATING	PTY allocation success	SPAWNING	Update file descriptor maps
SPAWNING	Child process started	RUNNING	Add PTY fd to select set
RUNNING	Process output available	RUNNING	Route to terminal emulator
RUNNING	SIGCHLD received	TERMINATING	Begin cleanup sequence
TERMINATING	Cleanup complete	CLOSED	Remove from all component state

Data Consistency Guarantees

The multiplexer provides **eventual consistency guarantees** for state synchronization, ensuring that all components eventually reflect the same logical state even when updates are processed asynchronously. The system uses **state version numbers** to detect and resolve inconsistencies when they occur.

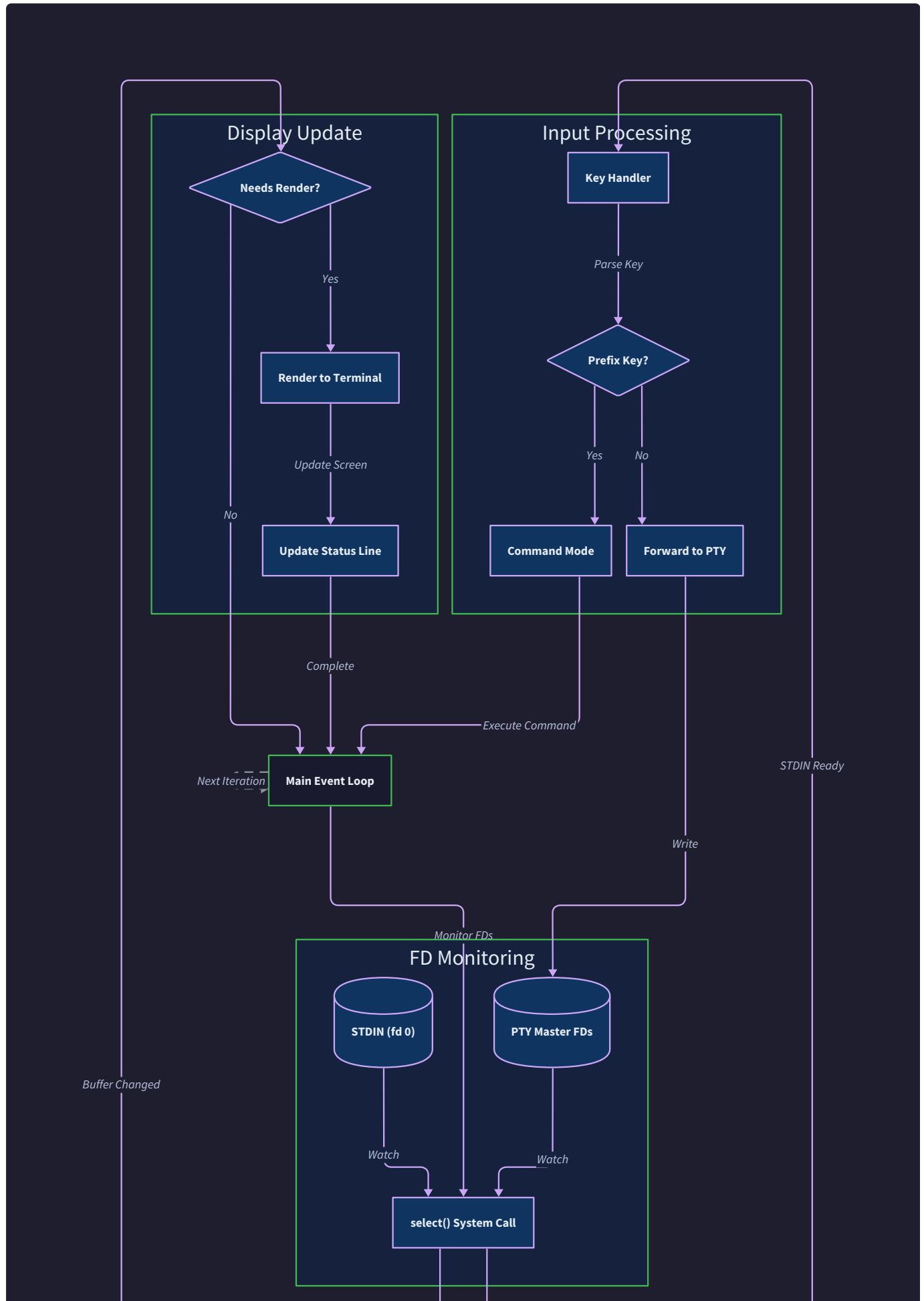
Each major state change (focus switch, pane creation/destruction, terminal resize) increments a global version counter that components can use to validate that their local state is current. When a component detects that its state is stale, it requests a refresh from the authoritative component rather than attempting to reconstruct state independently.

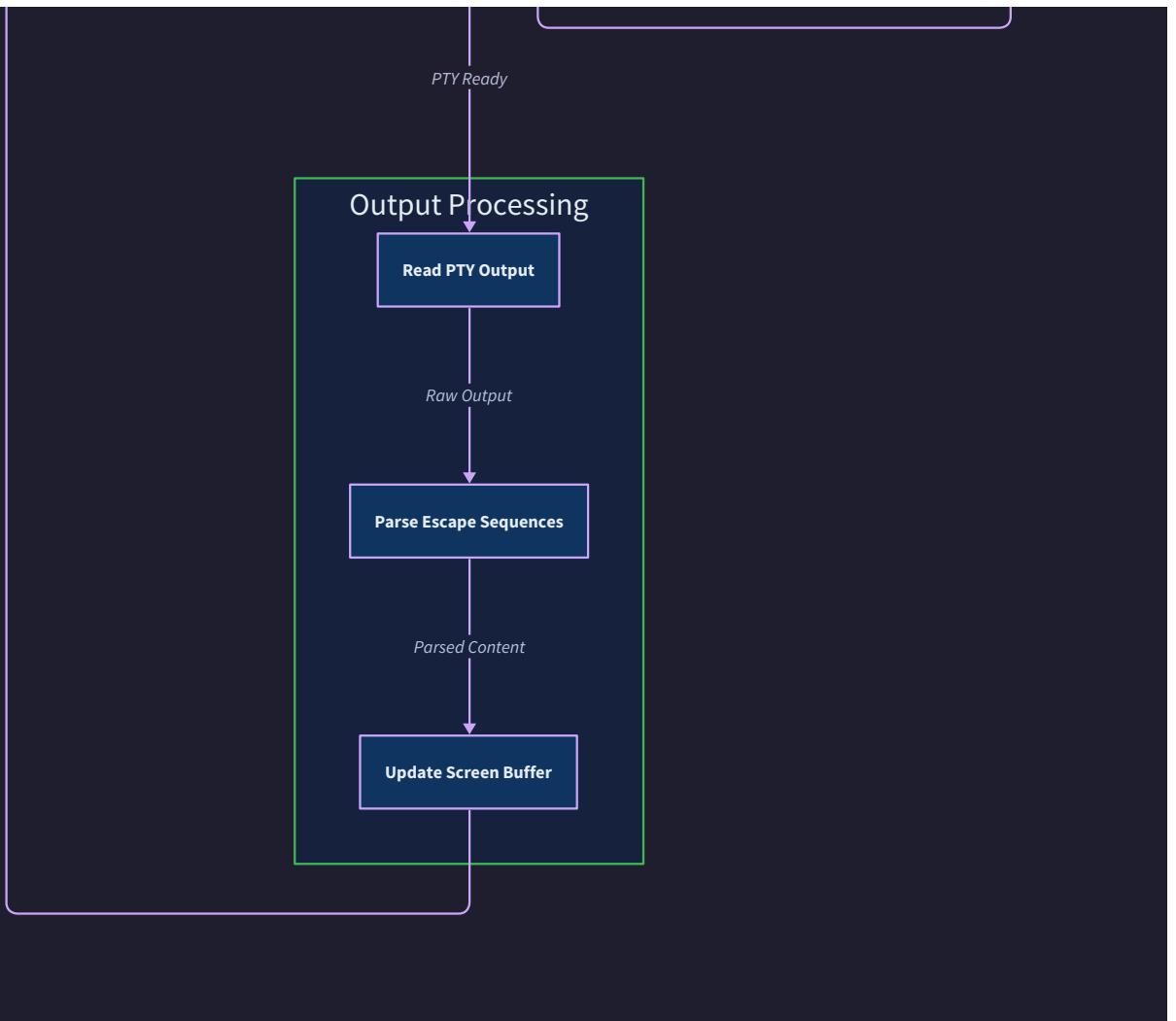
State Validation Protocol:

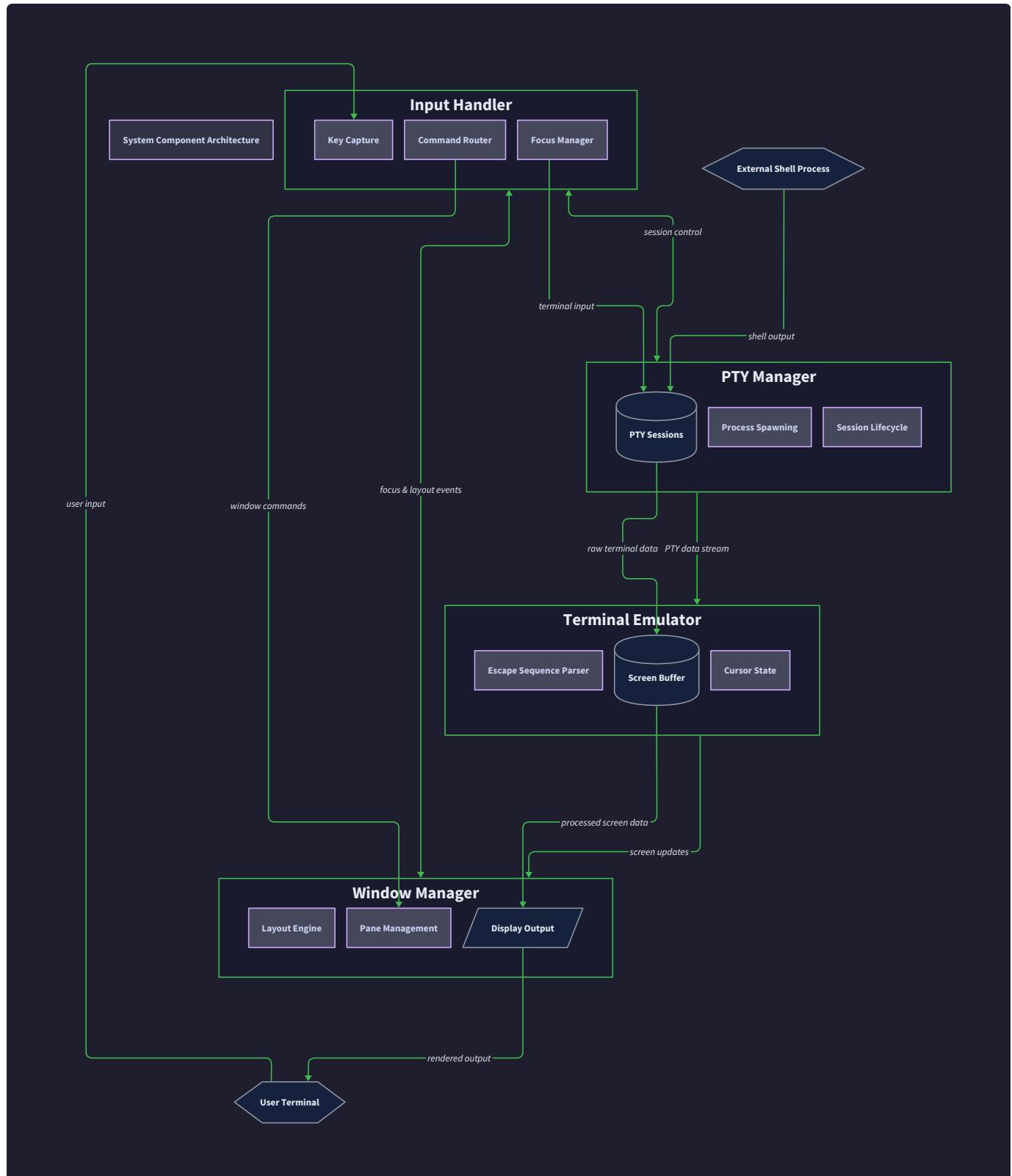
- Window manager maintains master state version for layout and focus changes
- Components cache state version with their local copies of shared state
- Before using cached state, components compare versions and refresh if stale
- Periodic validation cycles detect and correct any remaining inconsistencies

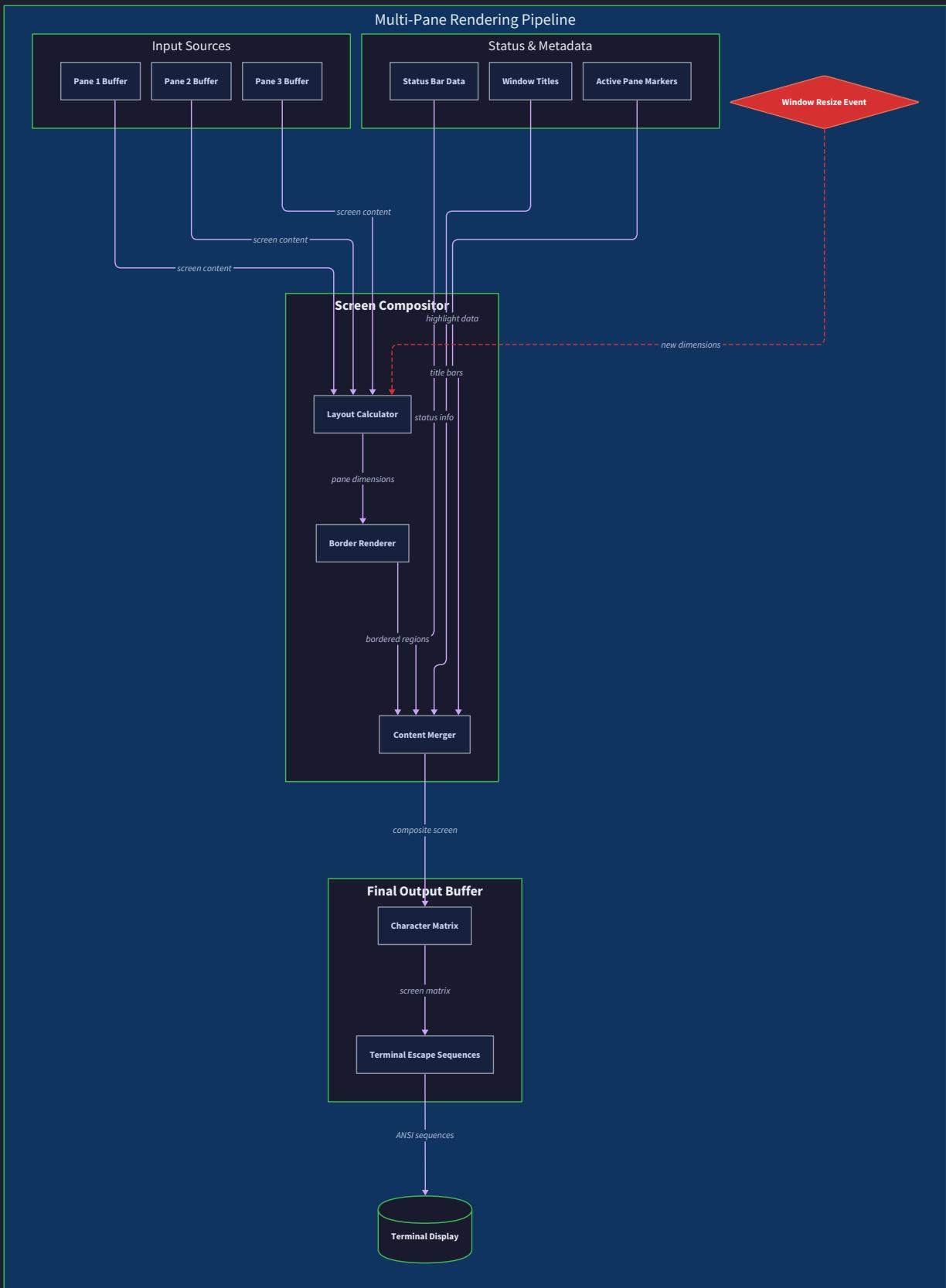
Architecture Decision: Centralized State Authority

- **Context:** Multiple components need consistent view of shared state (focus, layout, process status)
- **Options Considered:**
 1. Distributed state with peer synchronization
 2. Centralized state authority with component queries
 3. Event sourcing with state reconstruction
- **Decision:** Centralized state authority in window manager
- **Rationale:** Simplifies consistency guarantees, reduces synchronization complexity, matches natural ownership (window manager already coordinates layout)
- **Consequences:** Window manager becomes critical component but provides single source of truth for shared state









Common Pitfalls in State Synchronization

⚠ **Pitfall: Race Conditions in Focus Updates** When handling focus changes, developers often update the input routing before updating the visual focus indicators, creating a window where keystrokes route to the new pane but the display still shows the old pane as focused. This confuses users and makes debugging difficult. Always update the canonical focus state first, then propagate updates to all dependent components in dependency order.

⚠ **Pitfall: Incomplete Terminal Size Propagation** Failing to propagate terminal size changes to all PTY sessions causes shell programs to render output for the old terminal size, leading to display corruption and line wrapping issues. Ensure that every active PTY receives a `TIOCSWINSZ` ioctl with the new dimensions before processing any new output from shell sessions.

⚠ **Pitfall: Stale File Descriptor Mappings** When panes are destroyed, developers often forget to update the file descriptor mapping tables used by the event loop, causing the `select()` call to continue monitoring closed file descriptors. This leads to busy loops and resource leaks. Always update both the `fd_set` for `select()` and the mapping arrays when adding or removing PTY file descriptors.

⚠ **Pitfall: Blocking Operations in Event Loop** Including blocking system calls (like `sleep()` or blocking I/O) in the event loop prevents the multiplexer from responding to other events, causing apparent hangs and poor responsiveness. Use non-blocking alternatives and timeout-based `select()` calls to maintain responsiveness.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
I/O Multiplexing	<code>select()</code> system call	<code>epoll()</code> (Linux) or <code>kqueue()</code> (BSD)	<code>select()</code> is portable and sufficient for moderate PTY counts
Signal Handling	Traditional signal handlers with flags	<code>signalfd()</code> or self-pipe trick	Traditional signals work but require careful async-signal-safe handling
Timer Management	<code>select()</code> timeout parameter	<code>timerfd_create()</code> (Linux)	<code>select()</code> timeout sufficient for basic prefix key timeout
Memory Management	Fixed-size buffers with bounds checking	Dynamic allocation with memory pools	Fixed buffers prevent fragmentation and simplify debugging

Recommended File Structure

The event loop coordination spans multiple files but centralizes in a main event processing module:

```
src/
  main.c           ← entry point, initialization
  event_loop.c     ← main event loop implementation
  event_loop.h     ← event loop interface and structures
  components/
    pty_manager.c  ← PTY session management
    terminal_emulator.c  ← escape sequence parsing
    window_manager.c  ← layout and rendering
    input_handler.c  ← keyboard input processing
  utils/
    signal_utils.c  ← signal handling helpers
    fd_utils.c      ← file descriptor management utilities
  tests/
    test_event_loop.c  ← event loop unit tests
    integration_test.c  ← end-to-end flow testing
```

Infrastructure Starter Code

Signal Handling Utilities (`utils/signal_utils.c`):

```
#include <signal.h>
#include <sys/signalfd.h>
#include <unistd.h>
#include "signal_utils.h"

static volatile sig_atomic_t sigwinch_received = 0;
static volatile sig_atomic_t sigchld_received = 0;

// Signal-safe flag setters for traditional signal handling

void sigwinch_handler(int sig) {
    sigwinch_received = 1;
}

void sigchld_handler(int sig) {
    sigchld_received = 1;
}

int signal_utils_setup_handlers(void) {
    struct sigaction sa;

    // Install SIGWINCH handler for terminal resize
    sa.sa_handler = sigwinch_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGWINCH, &sa, NULL) == -1) {
        return -1;
    }

    // Install SIGCHLD handler for child process termination
    sa.sa_handler = sigchld_handler;
```

C

```
sigemptyset(&sa.sa_mask);

sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {

    return -1;

}

return 0;
}

bool signal_utils_check_winch(void) {

    if (sigwinch_received) {

        sigwinch_received = 0;

        return true;

    }

    return false;
}

bool signal_utils_check_chld(void) {

    if (sigchld_received) {

        sigchld_received = 0;

        return true;

    }

    return false;
}
```

File Descriptor Management Utilities (utils/fd_utils.c):

```
#include <sys/select.h>
#include <unistd.h>
#include <errno.h>
#include "fd_utils.h"

void fd_utils_init_set(fd_set* fds, int* max_fd) {
    FD_ZERO(fds);
    *max_fd = -1;
}

void fd_utils_add_fd(fd_set* fds, int fd, int* max_fd) {
    FD_SET(fd, fds);
    if (fd > *max_fd) {
        *max_fd = fd;
    }
}

void fd_utils_remove_fd(fd_set* fds, int fd, int* max_fd) {
    FD_CLR(fd, fds);
    // Note: max_fd recalculation needed if fd == *max_fd
    // For simplicity, we accept slight inefficiency
}

bool fd_utils_is_ready(fd_set* fds, int fd) {
    return FD_ISSET(fd, fds);
}

int fd_utils_make_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
```

C

```
    return -1;

}

return fcntl(fd, F_SETFL, flags | O_NONBLOCK);

}
```

Core Logic Skeleton Code

Main Event Loop Implementation (event_loop.c):

```
#include "event_loop.h"                                     C

#include "utils/signal_utils.h"

#include "utils/fd_utils.h"

// Initialize event loop with all components

int event_loop_init(event_loop_t* loop, window_manager_t* wm, input_handler_t* input) {

    // TODO 1: Initialize fd_set and max_fd for select()

    // TODO 2: Set up signal handlers using signal_utils_setup_handlers()

    // TODO 3: Initialize file descriptor mapping arrays

    // TODO 4: Store component pointers for event routing

    // TODO 5: Add stdin (fd 0) to select set for user input

    // Hint: Use fd_utils_init_set() and fd_utils_add_fd()

}

// Main event processing loop - runs until shutdown

int event_loop_run(event_loop_t* loop) {

    while (!loop->shutdown_requested) {

        // TODO 1: Copy master fd_set to working set for select()

        // TODO 2: Call select() with appropriate timeout (e.g., 100ms for responsiveness)

        // TODO 3: Handle select() errors - continue on EINTR, error on other failures

        // TODO 4: Check and process signal events first (SIGWINCH, SIGCHLD)

        // TODO 5: Process user input if stdin is ready

        // TODO 6: Process PTY output for all ready file descriptors

        // TODO 7: Render frame if any panes need redraw

        // Hint: Process signals before I/O to maintain consistent state

    }

}

// Process signals and update global state
```

```

static int process_signal_events(event_loop_t* loop) {

    // TODO 1: Check for SIGWINCH using signal_utils_check_winch()

    // TODO 2: If terminal resized, get new size and update window manager

    // TODO 3: Check for SIGCHLD using signal_utils_check_chld()

    // TODO 4: If child died, find corresponding pane and mark for cleanup

    // TODO 5: Update file descriptor mappings for any closed panes

    // Hint: Use window_manager_resize_terminal() for size changes

}

// Process input from a single PTY file descriptor

static int process_pty_output(event_loop_t* loop, int pty_fd) {

    // TODO 1: Look up pane_id from fd mapping array

    // TODO 2: Read data from PTY using pty_session_read()

    // TODO 3: Feed bytes to terminal emulator parser for this pane

    // TODO 4: Mark pane for redraw if screen buffer was updated

    // TODO 5: Handle read errors - close pane if PTY session ended

    // Hint: Use escape_parser_process_byte() for each byte read

}

// Add new PTY file descriptor to event loop monitoring

int event_loop_add_pane(event_loop_t* loop, int pane_id, int pty_fd) {

    // TODO 1: Add pty_fd to select() file descriptor set

    // TODO 2: Update file descriptor mapping arrays

    // TODO 3: Make file descriptor non-blocking for efficiency

    // TODO 4: Update max_fd if necessary

    // Hint: Use fd_utils functions for fd_set management

}

// Remove PTY file descriptor from event loop

```

```
int event_loop_remove_pane(event_loop_t* loop, int pane_id) {  
  
    // TODO 1: Look up file descriptor from pane mapping  
  
    // TODO 2: Remove fd from select() set using fd_utils_remove_fd()  
  
    // TODO 3: Clear mapping array entries  
  
    // TODO 4: Close file descriptor to prevent leaks  
  
    // Hint: Set mapping entries to -1 to indicate unused slots  
  
}
```

Inter-Component Coordination (`event_loop.c` continued):

```
// Coordinate state updates across components during events

static int coordinate_focus_change(event_loop_t* loop, int new_pane_id) {

    // TODO 1: Validate new_pane_id exists in window manager

    // TODO 2: Update window manager focus state

    // TODO 3: Mark old and new focus panes for redraw

    // TODO 4: Update input handler routing target

    // Hint: Use window_manager_switch_pane() for focus updates

}

// Coordinate pane closure across all components

static int coordinate_pane_closure(event_loop_t* loop, int pane_id) {

    // TODO 1: Terminate PTY session and child process

    // TODO 2: Remove pane from window manager layout

    // TODO 3: Remove file descriptor from event loop monitoring

    // TODO 4: Update focus if closed pane was active

    // TODO 5: Trigger full layout recalculation and redraw

    // Hint: Handle focus switching before removing pane from layout

}

// Handle terminal resize across all components

static int coordinate_terminal_resize(event_loop_t* loop) {

    int new_width, new_height;

    // TODO 1: Get new terminal dimensions using terminal_get_size()

    // TODO 2: Update window manager with new dimensions

    // TODO 3: Recalculate layout for all panes

    // TODO 4: Propagate new pane sizes to corresponding PTY sessions

    // TODO 5: Mark all panes for complete redraw

    // Hint: Update PTYs after layout calculation to use correct pane sizes
```

```
}
```

Debugging Techniques

Event Loop Debugging - Add comprehensive logging to understand event flow:

```
// Add to event_loop.c for debugging event processing
```

```
#ifdef DEBUG_EVENT_LOOP
```

```
#define DEBUG_LOG(fmt, ...) fprintf(stderr, "[EVENT] " fmt "\n", ##__VA_ARGS__)
```

```
#else
```

```
#define DEBUG_LOG(fmt, ...)
```

```
#endif
```

```
// In event loop:
```

```
DEBUG_LOG("select() returned %d ready fds", ready_count);
```

```
if (FD_ISSET(STDIN_FILENO, &ready_fds)) {
```

```
    DEBUG_LOG("Processing user input");
```

```
}
```

```
for (int i = 0; i < loop->pane_count; i++) {
```

```
    if (FD_ISSET(loop->pane_fd_map[i], &ready_fds)) {
```

```
        DEBUG_LOG("Processing output for pane %d (fd %d)", i, loop->pane_fd_map[i]);
```

```
    }
```

```
}
```

State Synchronization Issues:

Symptom	Likely Cause	Diagnostic Method	Fix
Input goes to wrong pane	Focus state inconsistent between components	Add logging to focus change operations	Update all components atomically during focus change
Display corruption after resize	PTY size not updated before processing new output	Log terminal size at each component	Ensure PTY resize completes before processing I/O
Event loop spinning (100% CPU)	Closed fd still in select() set	Check return values from read() operations	Remove fd from select set when PTY session closes
Pane content disappears	Screen buffer cleared but not redrawn	Log redraw flag changes	Set needs_redraw when buffer is modified

Milestone Checkpoints

After implementing main event loop:

- Run `./multiplexer` - should show single pane with shell
- Type commands - output should appear without blocking
- Press `Ctrl+C` - should send signal to shell, not exit multiplexer
- Resize terminal window - pane should resize automatically
- Multiple rapid keystrokes should all be processed without drops

After implementing inter-component coordination:

- Create multiple panes - each should have independent input/output
- Switch focus between panes - input routing should change immediately
- Close a pane - layout should rebalance and focus should move appropriately
- Resize with multiple panes - all panes should resize proportionally

After implementing state synchronization:

- All panes should maintain consistent state across operations
- Focus indicators should always match actual input routing
- Terminal size changes should propagate to all PTY sessions immediately
- Process termination should be detected and handled gracefully

Error Handling and Edge Cases

Milestone(s): All milestones (1-4) - this section describes failure modes and recovery strategies that span PTY management (M1), terminal emulation (M2), window management (M3), and input handling (M4)

Mental Model: The Safety Net

Think of error handling in a terminal multiplexer like designing safety systems for an aircraft. Just as an airplane has multiple backup systems for navigation, power, and control surfaces, a terminal multiplexer must gracefully handle failures in any of its critical subsystems. When the primary navigation system fails, the pilot doesn't crash—they switch to backup instruments and continue flying. Similarly, when a child process dies or a PTY fails, the multiplexer must detect the failure, isolate the damage, and maintain operation for other panes while gracefully handling the affected session.

The key insight is that terminal multiplexers manage multiple independent sessions that should remain isolated from each other's failures. A crash in one shell session should not bring down the entire multiplexer or affect other running sessions. This requires defensive programming, graceful degradation, and robust cleanup procedures that ensure the terminal is always restored to a usable state, even during catastrophic failures.

PTY and Process Failures

Process and PTY management represents the most critical failure domain in terminal multiplexers because child processes can terminate unexpectedly, PTY allocation can fail under resource pressure, and pipe communication can break due to system limits or process crashes. These failures must be detected quickly and handled gracefully to maintain session isolation and prevent resource leaks.

Child Process Termination Detection

The most common failure mode occurs when shell processes or their children terminate unexpectedly due to signals, crashes, or normal exit. The multiplexer must detect these terminations promptly and update its internal state accordingly.

Decision: SIGCHLD-Based Process Monitoring

- **Context:** Need to detect child process termination without blocking main event loop
- **Options Considered:** Polling with `waitpid(WNOHANG)`, synchronous `waitpid()` calls, `SIGCHLD` signal handler
- **Decision:** Use `SIGCHLD` signal handler with self-pipe trick for integration with `select()`
- **Rationale:** Signal handler provides immediate notification without polling overhead, self-pipe makes it compatible with `select()`-based event loop
- **Consequences:** Requires careful signal handler implementation and proper cleanup of zombie processes

Process Termination Scenario	Detection Method	Required Actions	Recovery Strategy
Normal shell exit	SIGCHLD signal	Update session status, close PTY, mark pane as terminated	Display exit message, allow pane closure
Shell crash (SIGSEGV)	SIGCHLD signal	Log crash reason, cleanup PTY resources	Show crash notification, restart shell option
Killed by user (SIGKILL)	SIGCHLD signal	Immediate cleanup, no graceful shutdown	Close pane, update layout if needed
Child process orphan	SIGCHLD signal	Reap zombie, maintain session	Continue normal operation
Process group termination	Multiple SIGCHLD	Cleanup all related processes	Close affected panes, rebuild layout

The process termination handler must safely reap zombie processes while maintaining accurate session state. This requires careful coordination between the signal handler and the main event loop to avoid race conditions.

```
// Process termination handling flow:
// 1. SIGCHLD signal received
// 2. Signal handler writes notification to self-pipe
// 3. Main event loop detects self-pipe readable
// 4. Event loop calls waitpid() to reap children
// 5. Update pty_session_t structures with exit status
// 6. Mark affected panes for cleanup or restart
```

C

PTY Allocation Failures

PTY allocation can fail due to system limits on pseudo-terminal devices, insufficient privileges, or resource exhaustion. These failures must be handled gracefully without crashing the multiplexer or leaving orphaned processes.

PTY Failure Mode	Detection Point	Error Recovery	User Feedback
posix_openpty() fails	pty_session_create()	Return error code, no session created	Show error message in status bar
grantpt() fails	PTY setup	Close master fd, cleanup	Display permission error
unlockpt() fails	PTY setup	Close master and slave fds	Show system error message
Slave open fails	Child process	Exit child with error code	Parent detects child exit
System PTY limit	posix_openpty()	Retry with backoff, suggest closing panes	Show resource limit warning

The critical insight here is that PTY allocation failures should never crash the multiplexer—they should be treated as temporary resource constraints that may resolve when other sessions close.

Broken Pipe and I/O Failures

Communication between the multiplexer and child processes can fail due to broken pipes, full buffers, or I/O errors. These conditions require careful detection and recovery to maintain system stability.

I/O Failure Type	Detection Method	Immediate Response	Long-term Recovery
Broken pipe (SIGPIPE)	Signal or EPIPE error	Stop writing to PTY	Mark session as terminated
Read failure on master	read() returns -1	Check errno, handle EAGAIN	Close session if permanent failure
Write buffer full	write() returns partial	Buffer remaining data	Implement flow control
PTY master closed	EOF on read	Immediate session cleanup	Remove pane from layout
Slave side closed	SIGHUP signal	Graceful session shutdown	Allow process termination

The I/O error handling must distinguish between temporary conditions (like EAGAIN) that require retry logic and permanent failures (like EPIPE) that indicate session termination.

Common PTY Management Pitfalls

⚠ Pitfall: Zombie Process Accumulation Many implementations forget to reap child processes after SIGCHLD signals, leading to zombie accumulation that eventually exhausts the process table. The multiplexer must call waitpid() for every SIGCHLD signal received, even if multiple children exit simultaneously.

⚠ Pitfall: File Descriptor Leaks Failing to close both master and slave PTY file descriptors when sessions terminate leads to file descriptor exhaustion. Each pty_session_t must track both file descriptors and ensure they are closed in all exit paths, including error conditions.

⚠ Pitfall: Race Condition in Process Cleanup Accessing pty_session_t structures from both signal handlers and main event loop creates race conditions. Use the self-pipe trick to defer all PTY session updates to the main event loop context.

⚠ Pitfall: Ignoring Process Group Signals When users send signals like SIGTERM to the multiplexer, these must be forwarded to all child processes. Failing to properly manage process groups can leave orphaned shell processes running after the multiplexer exits.

Terminal and Display Errors

Terminal and display errors encompass failures in escape sequence parsing, screen buffer management, and terminal size handling. These errors can corrupt the display state, cause rendering artifacts, or desynchronize the virtual screen from the physical terminal.

Terminal Size Change Handling

Terminal resize events represent one of the most complex error scenarios because they require coordinated updates across multiple components while maintaining visual consistency. Size changes can occur at any time and must be propagated through the entire system.

Decision: Synchronous Resize Propagation

- Context:** Terminal resize affects layout, PTY sizes, and screen buffers simultaneously
- Options Considered:** Asynchronous resize with eventual consistency, synchronous resize with temporary inconsistency, queued resize events
- Decision:** Synchronous resize that updates all components atomically
- Rationale:** Prevents visual artifacts and ensures PTY sizes match layout dimensions immediately
- Consequences:** Resize operations may briefly block event loop but guarantee consistency

Resize Failure Point	Error Condition	Detection Method	Recovery Action
Layout recalculation	Panes below minimum size	Dimension validation	Adjust split ratios, merge small panes
PTY size update	ioctl(TIOCSWINSZ) fails	System call return code	Log error, continue with current size
Screen buffer resize	Memory allocation failure	malloc() returns NULL	Revert to previous size, show warning
Render buffer realloc	Insufficient memory	realloc() returns NULL	Use smaller buffer, degrade rendering quality
Multiple rapid resizes	Event queue overflow	Event count tracking	Coalesce resize events, process latest only

The resize handling must be atomic to prevent temporary states where layout dimensions don't match PTY sizes, which can cause child processes to receive incorrect terminal geometry information.

Malformed Escape Sequence Recovery

Terminal emulation must handle malformed or incomplete escape sequences gracefully without corrupting the parser state or losing subsequent valid sequences. Real-world terminal output often contains truncated or invalid escape sequences due to program crashes or binary data.

Parser Error Type	Error Condition	Detection Method	Recovery Strategy
Truncated CSI sequence	Buffer overflow in escape_parser_t	Buffer length check	Reset parser state, treat as literal text
Invalid parameters	Non-numeric CSI parameters	Character validation	Use default values, continue parsing
Unknown sequence	Unrecognized escape code	Sequence lookup failure	Ignore sequence, return to normal mode
UTF-8 decode error	Invalid byte sequence	UTF-8 validation	Insert replacement character (U+FFFD)
Buffer overflow	Sequence exceeds ESC_SEQ_BUFFER_SIZE	Length tracking	Truncate sequence, force state reset
State machine stuck	Parser never returns to NORMAL	Timeout or sequence count	Emergency reset to NORMAL state

The fundamental principle in escape sequence error recovery is to never let malformed input corrupt the parser state permanently—always provide a path back to the NORMAL parsing state.

Screen Buffer Corruption Detection

Screen buffer corruption can occur due to memory errors, invalid cursor movements, or arithmetic overflow in coordinate calculations. The multiplexer must detect these conditions and recover gracefully.

Corruption Type	Detection Method	Error Indicators	Recovery Action
Invalid cursor position	Bounds checking	cursor_x >= width or cursor_y >= height	Clamp to screen bounds
Buffer overrun	Array bounds validation	Write beyond cells array	Reallocate buffer, clear corruption
Attribute corruption	Value range checking	Invalid attribute bits set	Reset to default attributes
UTF-8 sequence split	Character boundary validation	Partial codepoint at line end	Complete sequence on next line
Color index overflow	Color range validation	Color value > COLOR_WHITE	Use COLOR_DEFAULT
Scrollbar overflow	Buffer size monitoring	scrollback_lines > MAX_SCROLLBACK	Discard oldest lines

The screen buffer must implement defensive programming practices that validate all coordinate and attribute values before using them to index into memory structures.

Common Terminal Emulation Pitfalls

⚠ Pitfall: Cursor Position Overflow Many implementations fail to validate cursor coordinates before using them as array indices, leading to buffer overruns when malformed escape sequences specify coordinates outside the screen buffer. Always clamp cursor positions to valid screen bounds.

⚠ Pitfall: UTF-8 Boundary Errors When UTF-8 sequences span multiple read() calls, incomplete sequences can corrupt character display or cause parser state confusion. Buffer incomplete UTF-8 sequences until complete codepoints can be decoded.

⚠ Pitfall: Attribute State Leakage Failing to properly reset terminal attributes when switching between panes can cause formatting to leak between sessions. Each pane must maintain independent attribute state in its screen_buffer_t.

⚠ Pitfall: Resize During Output Terminal resize events that occur while processing escape sequences can cause coordinate calculations to use inconsistent screen dimensions. Cache screen dimensions at the start of escape sequence processing.

Cleanup and Exit Procedures

Proper cleanup and exit procedures ensure that the terminal multiplexer can shutdown gracefully under all conditions, restoring terminal settings and cleaning up system resources even during abnormal termination scenarios. This represents the final safety net that prevents the multiplexer from leaving the user's terminal in an unusable state.

Terminal State Restoration

Terminal state restoration is the most critical cleanup operation because failure to restore original terminal attributes can leave the user's terminal in raw mode, making it unusable for normal shell interaction. This restoration must occur even during crash scenarios or signal-induced termination.

Decision: Multi-Level Cleanup Registration

- **Context:** Terminal restoration must occur regardless of how the process terminates
- **Options Considered:** atexit() handlers only, signal handlers only, both atexit() and signal handlers
- **Decision:** Use both atexit() handlers and signal handlers with shared cleanup code
- **Rationale:** Covers both normal exit and signal termination, provides redundant safety
- **Consequences:** Requires careful design to prevent double-cleanup and ensure idempotent operations

Exit Scenario	Cleanup Trigger	Terminal Restoration	Resource Cleanup
Normal exit	atexit() handler	Restore original_termios	Close all PTYs, free memory
SIGINT (Ctrl-C)	Signal handler	Immediate restoration	Emergency PTY cleanup
SIGTERM	Signal handler	Graceful restoration	Send SIGTERM to children
SIGSEGV crash	Signal handler	Emergency restoration only	Best-effort cleanup
Process kill	None possible	Automatic by kernel	Automatic by kernel
exit() call	atexit() handler	Full restoration	Complete cleanup

The terminal restoration procedure must be idempotent and signal-safe, allowing it to be called multiple times or from signal handler context without causing additional problems.

```
// Terminal cleanup procedure:  
  
// 1. Check if raw_mode_active flag is set  
  
// 2. Call tcsetattr() with original_termios  
  
// 3. Clear raw_mode_active flag  
  
// 4. Restore cursor visibility if hidden  
  
// 5. Reset terminal to cooked mode  
  
// 6. Flush any pending output
```

Process Group Management

The multiplexer must properly manage process groups to ensure that signals are delivered correctly to child processes and that all related processes are cleaned up during shutdown. This prevents orphaned shell

processes from continuing to run after the multiplexer exits.

Process Management Task	Implementation Approach	Error Handling	Signal Coordination
Child process creation	setsid() in child before exec	Check return codes	Setup SIGCHLD handler
Process group tracking	Store pgid in pty_session_t	Handle invalid pgids	Forward termination signals
Signal forwarding	killpg() to process groups	Ignore ESRCH errors	Respect signal ordering
Graceful termination	SIGTERM followed by SIGKILL	Timeout mechanism	Wait for child cleanup
Orphan prevention	Proper session leadership	Monitor parent-child relationships	Clean exit procedures

The key insight is that the multiplexer acts as a session leader that must properly manage the lifecycle of all descendant processes, ensuring they don't become orphans when the multiplexer terminates.

Resource Deallocation

Memory and file descriptor cleanup must be thorough and systematic to prevent resource leaks that could affect system stability or subsequent multiplexer runs. This includes both explicit allocations and resources managed by the operating system.

Resource Type	Cleanup Procedure	Error Recovery	Verification Method
PTY file descriptors	close() master and slave fds	Log but continue cleanup	Check with ls of /proc/fd
Memory allocations	free() all malloc'd buffers	NULL pointer safety	Valgrind or AddressSanitizer
Screen buffers	Free cells and scrollback arrays	Check for double-free	Memory debugging tools
Process children	waitpid() for all spawned processes	Handle already-reaped children	Check process table
Signal handlers	Restore default signal handling	Best-effort restoration	Signal behavior testing
Layout tree	Recursive free of layout_node_t	NULL pointer checking	Tree traversal verification

The cleanup procedure must handle partial initialization scenarios where some components may not be fully initialized when cleanup is triggered due to early errors.

Emergency Recovery Procedures

Emergency recovery handles situations where normal cleanup procedures cannot complete successfully, such as when the system is under extreme resource pressure or when cleanup operations themselves fail. These procedures prioritize terminal restoration above all other cleanup tasks.

Emergency Condition	Detection Method	Emergency Response	Fallback Action
Cleanup failure	System call errors during cleanup	Skip failed operations, continue	Terminal restoration only
Memory corruption	Segfault during cleanup	Signal-safe restoration	exec() fresh shell
Resource exhaustion	ENOMEM during cleanup	Free essential resources only	Emergency terminal reset
Infinite loop	Timeout during cleanup	Force termination	Kill process group
Signal storm	Repeated signal delivery	Block signals, emergency cleanup	System reset procedures
Corrupted state	Inconsistent internal state	Abandon complex cleanup	Basic terminal restoration

Emergency procedures prioritize leaving the user with a functional terminal over completing perfect cleanup—it's better to leak some memory than to leave the terminal unusable.

Common Cleanup and Exit Pitfalls

⚠ **Pitfall: Non-Idempotent Cleanup** Calling cleanup functions multiple times can cause crashes if they attempt to free already-freed memory or close already-closed file descriptors. Design all cleanup functions to be safely callable multiple times by checking resource validity before cleanup.

⚠ **Pitfall: Cleanup Function Complexity** Complex cleanup operations in signal handlers can cause deadlocks or crashes because signal handlers have limited safe function availability. Keep signal handler cleanup minimal and defer complex operations to atexit() handlers when possible.

⚠ **Pitfall: Incomplete Terminal Restoration** Only restoring some terminal attributes (like canonical mode) while forgetting others (like echo settings) can leave the terminal in an inconsistent state. Always restore the complete original_termios structure captured at startup.

⚠ **Pitfall: Child Process Leakage** Forgetting to send termination signals to child processes before exiting can leave shell processes running as orphans that consume system resources. Always send SIGTERM to all process groups before final cleanup.

⚠ **Pitfall: Resource Cleanup Ordering** Cleaning up resources in the wrong order can cause cascading failures, such as freeing memory before closing file descriptors that reference it. Follow the reverse order of allocation for cleanup operations.

Implementation Guidance

Technology Recommendations

Error Handling Component	Simple Option	Advanced Option
Signal Management	Basic signal() calls with global flags	signalfd() or self-pipe trick with sigprocmask()
Process Monitoring	Synchronous waitpid() in main loop	Asynchronous SIGCHLD with event integration
Terminal Restoration	atexit() handler only	Combined atexit() + signal handlers + cleanup stack
Error Logging	fprintf() to stderr	Structured logging with severity levels and rotation
Memory Debugging	Manual NULL checks	AddressSanitizer + Valgrind integration
Resource Tracking	Manual cleanup lists	RAII-style cleanup with function pointers

Recommended File Structure

```
terminal-multiplexer/
  src/
    error_handling.h           ← Error codes and cleanup function declarations
    error_handling.c          ← Core error handling and recovery logic
    signal_utils.h            ← Signal handling utilities and self-pipe
    signal_utils.c            ← SIGCHLD, SIGWINCH, and cleanup signal handlers
    cleanup.h                 ← Resource cleanup and terminal restoration
    cleanup.c                 ← Cleanup registration and execution
    pty_errors.h              ← PTY-specific error handling
    pty_errors.c              ← PTY failure recovery and process management
    terminal_recovery.h       ← Terminal state restoration utilities
    terminal_recovery.c       ← Emergency terminal recovery procedures
  tests/
    test_error_handling.c     ← Unit tests for error scenarios
    test_cleanup.c            ← Cleanup procedure verification
    test_signal_handling.c    ← Signal handler testing
```

Core Infrastructure Code

Signal Utilities (signal_utils.c):

```
#include <signal.h> C

#include <unistd.h>

#include <errno.h>

#include <sys/wait.h>

#include "signal_utils.h"

static volatile sig_atomic_t winch_received = 0;

static volatile sig_atomic_t chld_received = 0;

static int signal_pipe_fd[2] = {-1, -1};

// Signal-safe SIGWINCH handler

void sigwinch_handler(int sig) {

    winch_received = 1;

    char byte = 'W';

    write(signal_pipe_fd[1], &byte, 1); // Signal main loop

}

// Signal-safe SIGCHLD handler

void sigchld_handler(int sig) {

    chld_received = 1;

    char byte = 'C';

    write(signal_pipe_fd[1], &byte, 1); // Signal main loop

}

// Signal-safe emergency cleanup handler

void emergency_cleanup_handler(int sig) {

    static const char msg[] = "Emergency cleanup triggered\n";

    write(STDERR_FILENO, msg, sizeof(msg) - 1);

}

// Minimal terminal restoration only
```

```
terminal_emergency_restore();

// Re-raise signal with default handler

signal(sig, SIG_DFL);

raise(sig);

}

int signal_utils_setup_handlers(void) {

// Create self-pipe for signal integration with select()

if (pipe(signal_pipe_fd) == -1) {

return -1;

}

// Make write end non-blocking to prevent deadlock

int flags = fcntl(signal_pipe_fd[1], F_GETFL);

fcntl(signal_pipe_fd[1], F_SETFL, flags | O_NONBLOCK);

struct sigaction sa;

// Install SIGWINCH handler

sa.sa_handler = sigwinch_handler;

sigemptyset(&sa.sa_mask);

sa.sa_flags = SA_RESTART;

if (sigaction(SIGWINCH, &sa, NULL) == -1) {

return -1;

}

// Install SIGCHLD handler
```

```
    sa.sa_handler = sigchld_handler;

    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;

    if (sigaction(SIGCHLD, &sa, NULL) == -1) {

        return -1;
    }

    // Install emergency cleanup for fatal signals

    sa.sa_handler = emergency_cleanup_handler;

    sa.sa_flags = 0; // No SA_RESTART for emergency

    sigaction(SIGSEGV, &sa, NULL);

    sigaction(SIGBUS, &sa, NULL);

    sigaction(SIGTERM, &sa, NULL);

    sigaction(SIGINT, &sa, NULL);

    return signal_pipe_fd[0]; // Return read end for select()

}

bool signal_utils_check_winch(void) {

    bool result = winch_received;

    winch_received = 0;

    return result;
}

bool signal_utils_check_chld(void) {

    bool result = chld_received;

    chld_received = 0;

    return result;
}
```

```
void signal_utils_drain_pipe(void) {  
  
    char buffer[256];  
  
    while (read(signal_pipe_fd[0], buffer, sizeof(buffer)) > 0) {  
  
        // Drain signal notifications  
  
    }  
  
}
```

Terminal Recovery (terminal_recovery.c):

```
#include <termios.h>
```

C

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include "terminal_recovery.h"
```

```
static struct termios original_termios;
```

```
static bool termios_saved = false;
```

```
static bool cleanup_registered = false;
```

```
// atexit() cleanup function
```

```
void cleanup_on_exit(void) {
```

```
    terminal_restore_mode_safe();
```

```
}
```

```
// Emergency terminal restoration (signal-safe)
```

```
void terminal_emergency_restore(void) {
```

```
    if (termios_saved) {
```

```
        tcsetattr(STDIN_FILENO, TCSANOW, &original_termios);
```

```
        // Reset terminal to known good state
```

```
        const char reset_sequence[] = "\033c\033[?1000l\033[?1002l\033[?1049l";
```

```
        write(STDOUT_FILENO, reset_sequence, sizeof(reset_sequence) - 1);
```

```
}
```

```
}
```

```
int terminal_state_init(terminal_state_t *state) {
```

```
    memset(state, 0, sizeof(*state));
```

```
    // Save original terminal attributes
```

```
if (tcgetattr(STDIN_FILENO, &state->original_termios) == -1) {

    return TMUX_ERROR_TERMINAL_SETUP;

}

original_termios = state->original_termios;

termios_saved = true;

state->raw_mode_active = false;

return TMUX_OK;

}

int terminal_register_cleanup(terminal_state_t *state) {

    if (!cleanup_registered) {

        if (atexit(cleanup_on_exit) != 0) {

            return TMUX_ERROR_TERMINAL_SETUP;

        }

        cleanup_registered = true;

        state->restoration_registered = true;

    }

    return TMUX_OK;

}

int terminal_restore_mode_safe(void) {

    if (termios_saved) {

        if (tcsetattr(STDIN_FILENO, TCSANOW, &original_termios) == -1) {

            return -1;

        }

    }

}
```

```
// Show cursor if it was hidden

const char show_cursor[] = "\033[?25h";

write(STDOUT_FILENO, show_cursor, sizeof(show_cursor) - 1);

termios_saved = false;

}

return 0;

}
```

Error Handling Skeleton Code

PTY Error Recovery (pty_errors.c):

```
#include "pty_errors.h" C

// Handle child process termination detected via SIGCHLD

int pty_handle_child_termination(window_manager_t *wm) {

    pid_t pid;
    int status;

    // TODO 1: Loop calling waitpid(WNOHANG) until no more children

    // TODO 2: For each reaped child, find corresponding pty_session_t

    // TODO 3: Update session->process_running = false and session->exit_status

    // TODO 4: Mark associated pane for cleanup or restart prompt

    // TODO 5: If this was the last pane, handle graceful shutdown

    // Hint: Use WIFEXITED() and WEXITSTATUS() to decode exit status

    return TMUX_OK;
}

// Recover from PTY allocation failure

int pty_handle_allocation_failure(pty_session_t *session, const char *error_msg) {

    // TODO 1: Log the specific PTY allocation error (EACCES, ENOMEM, etc.)

    // TODO 2: Clean up any partially allocated resources (master_fd if open)

    // TODO 3: Set session state to indicate allocation failure

    // TODO 4: Return appropriate error code for display to user

    // TODO 5: Suggest recovery actions (close other panes, check permissions)

    // Hint: Use strerror(errno) to get human-readable error description

    return TMUX_ERROR_PTY_ALLOCATION;
}
```

```

// Handle broken pipe or I/O errors on PTY

tmux_error_t pty_handle_io_error(pty_session_t *session, int error_code) {

    // TODO 1: Determine if error is temporary (EAGAIN) or permanent (EPIPE)

    // TODO 2: For EPIPE, mark session as terminated and trigger cleanup

    // TODO 3: For EAGAIN/EWOULDBLOCK, implement retry with backoff

    // TODO 4: For other errors, log and decide on session termination

    // TODO 5: Update session status and notify window manager of changes

    // Hint: SIGPIPE errors indicate the child process has exited

    return TMUX_OK;
}

```

Milestone Checkpoints

After implementing PTY error handling:

- Run: `./tmux-test --test-pty-errors`
- Expected: Pane should handle shell exit gracefully, show exit status
- Verify: Kill child process with `kill -9 <pid>`, multiplexer should detect and cleanup
- Signs of problems: Zombie processes in `ps aux`, file descriptor leaks in `/proc/<pid>/fd`

After implementing terminal error recovery:

- Run: Send malformed escape sequences: `echo -e "\033[999;999H\033[J"`
- Expected: Display should not corrupt, cursor should stay in bounds
- Verify: Terminal resize during heavy output should not cause artifacts
- Signs of problems: Cursor appears outside screen, display corruption, terminal hangs

After implementing cleanup procedures:

- Test: Kill multiplexer with Ctrl-C during operation
- Expected: Terminal should return to normal mode, cursor visible
- Verify: No orphaned shell processes remain after multiplexer exit
- Signs of problems: Terminal stuck in raw mode, orphaned processes, "stty sane" needed

Debugging Tips

Symptom	Likely Cause	Diagnosis Method	Fix
Terminal stays in raw mode after crash	Signal handler not restoring termios	Check signal handler installation	Add terminal restoration to all signal handlers
Zombie processes accumulate	Not calling waitpid() for SIGCHLD	Monitor with `ps aux`	grep defunct`
Multiplexer hangs on child death	Blocking waitpid() in main loop	Check for synchronous waitpid() calls	Use WNOHANG flag and event-driven handling
PTY allocation fails sporadically	System PTY limit reached	Check <code>/proc/sys/kernel/pty/max</code>	Implement retry logic and user notification
Escape sequences cause display corruption	Parser state not reset on errors	Log parser state transitions	Add parser reset on malformed sequences
Memory leaks on abnormal exit	Cleanup not called in all exit paths	Use valgrind or AddressSanitizer	Register cleanup with atexit() and signal handlers

Testing Strategy

Milestone(s): All milestones (1-4) - this section provides comprehensive testing approaches that verify PTY functionality (M1), terminal emulation accuracy (M2), window management behavior (M3), and input handling (M4)

Mental Model: The Quality Assurance Laboratory

Think of testing a terminal multiplexer like running a quality assurance laboratory for a complex manufacturing process. Just as a manufacturing QA lab has different testing stations - component testing benches that verify individual parts, integration test beds that validate assembled products, and final inspection checkpoints that ensure the complete product meets specifications - our terminal multiplexer testing strategy operates at multiple levels of granularity and scope.

The component unit tests are like individual testing stations that verify each subsystem in isolation - the escape sequence parser, the layout calculator, the PTY session manager. These tests use controlled inputs and verify precise outputs without external dependencies. The integration tests are like the assembly line test beds that verify how components work together - real shell processes generating terminal output, actual PTY devices handling I/O, complete pane rendering pipelines. The milestone verification checkpoints are like the final inspection stations that validate the complete multiplexer behavior against user requirements.

This multi-layered testing approach ensures we catch different categories of defects at the appropriate level - logic errors in individual components, interface mismatches between components, and behavior gaps in the complete system. Each layer serves a specific diagnostic purpose and provides different types of confidence in system correctness.

Component Unit Tests

Component unit tests verify the correctness of individual subsystems in isolation from their dependencies and external resources. These tests focus on algorithmic correctness, edge case handling, and interface contract compliance for each major component of the terminal multiplexer architecture.

The **escape sequence parser** represents one of the most critical components to test thoroughly due to the complexity and variety of ANSI escape sequences that must be handled correctly. The parser must correctly transition between states, accumulate parameters, and generate appropriate actions for hundreds of different escape sequence combinations.

Test Category	Test Cases	Expected Behavior	Verification Method
Basic CSI Sequences	Cursor movements (CUU, CUD, CUF, CUB)	Parser generates correct cursor position changes	Verify <code>cursor_x</code> and <code>cursor_y</code> values in buffer
Parameter Parsing	Multi-parameter sequences like <code>\e[2;5H</code>	Parameters correctly extracted into <code>csi_params</code> array	Check <code>csi_param_count</code> and individual parameter values
Default Parameters	Sequences with missing parameters like <code>\e[H</code>	Parser applies correct default values	Verify cursor moves to (0,0) for missing parameters
Invalid Sequences	Malformed escape sequences	Parser resets to normal state without corruption	Check <code>parser_state_t</code> returns to <code>NORMAL</code>
UTF-8 Handling	Multi-byte Unicode characters	Characters correctly decoded and stored	Verify <code>terminal_cell_t.ch</code> contains complete codepoint
Buffer Boundaries	Sequences longer than <code>ESC_SEQ_BUFFER_SIZE</code>	Parser truncates or rejects overlong sequences	Check buffer doesn't overflow and state resets

The escape sequence parser tests should create controlled input streams and verify state transitions step by step. For example, when testing the sequence `\e[2J` (clear screen), the test should verify that the parser transitions from `NORMAL` to `ESCAPE` on the escape character, then to `CSI_PARAMS` on the bracket, accumulates the parameter "2", and finally generates a clear screen action with the correct mode parameter.

Critical Testing Insight: Escape sequence parsing failures often manifest as display corruption that appears intermittent or input-dependent. Unit tests must exercise boundary conditions like buffer overflows, incomplete sequences, and interleaved UTF-8 characters to catch these subtle bugs before they reach integration testing.

The **screen buffer management** component requires extensive testing of cursor movement, scrolling behavior, and character storage operations. These tests verify that the virtual screen accurately tracks terminal state without requiring actual terminal I/O.

Test Category	Test Cases	Expected Behavior	Verification Method
Cursor Movement	Absolute and relative positioning	Cursor stays within buffer boundaries	Check <code>cursor_x</code> and <code>cursor_y</code> constraints
Scrolling Operations	Line insertion, deletion, scroll regions	Content shifts correctly, scrollback preserved	Verify character positions and scrollback buffer
Character Storage	Normal text, wide characters, combining marks	Characters stored with correct attributes	Check <code>terminal_cell_t</code> content and formatting
Buffer Clearing	Full screen, line clearing, selective erase	Specified regions cleared to default state	Verify cleared cells contain default values
Attribute Handling	Bold, underline, color changes	Attributes applied to subsequent characters	Check <code>current_attributes</code> and cell attributes
Scrollbar Management	Lines scrolling off screen	Old lines preserved up to <code>MAX_SCROLLBACK</code> limit	Verify scrollback buffer content and bounds

Screen buffer tests should create known buffer states and verify that operations produce expected results. For example, a scrolling test should fill the buffer with identifiable content, perform scroll operations, and verify that the correct lines moved to the scrollback buffer while new content appeared in the visible area.

The **layout calculation algorithm** must be tested to ensure correct pane dimension calculations, split operations, and resize handling across various terminal sizes and split configurations.

Test Category	Test Cases	Expected Behavior	Verification Method
Single Pane Layout	Full terminal dimensions	Pane occupies entire available space	Check pane coordinates match terminal size
Horizontal Splits	Two-pane horizontal division	Panes split along horizontal boundary	Verify y-coordinates and height allocation
Vertical Splits	Two-pane vertical division	Panes split along vertical boundary	Verify x-coordinates and width allocation
Nested Splits	Complex multi-level split trees	Each pane receives correct dimensions	Walk layout tree and verify all leaf calculations
Minimum Size Constraints	Splits with insufficient space	Panes respect <code>MIN_PANE_WIDTH</code> and <code>MIN_PANE_HEIGHT</code>	Check no pane dimensions below minimums
Terminal Resize	Layout recalculation on size change	All panes resize proportionally	Compare before/after dimension ratios

Layout tests should construct various tree configurations and verify that the `layout_calculate_dimensions` function produces correct absolute coordinates for all panes. These tests can use small, predictable terminal sizes (like 80x24) to make manual verification straightforward.

The **PTY session management** component requires careful testing of process lifecycle, file descriptor handling, and signal propagation without relying on actual shell processes or terminal devices.

Test Category	Test Cases	Expected Behavior	Verification Method
Session Creation	PTY allocation and process spawning	Valid file descriptors and running process	Check <code>master_fd</code> , <code>slave_fd</code> , and <code>child_pid</code>
Process Monitoring	Child process status tracking	<code>process_running</code> accurately reflects process state	Use test processes with known behavior
Window Size Updates	TIOCSWINSZ ioctl propagation	Child process receives resize notifications	Mock ioctl calls and verify parameters
Session Cleanup	Resource deallocation on close	File descriptors closed, processes terminated	Verify fds invalid and processes reaped
Signal Handling	SIGCHLD processing	Child termination detected and recorded	Use controlled child processes for testing
I/O Operations	Read/write through PTY master	Data flows correctly between processes	Use pipe-like test programs for verification

PTY session tests can use simple test programs instead of full shells - for example, programs that echo input, print known output patterns, or terminate after specific signals. This provides predictable behavior for test verification while exercising the same PTY infrastructure used with real shells.

Architecture Decision: Mock vs Real Dependencies

Decision: Use Minimal Real Dependencies with Controlled Behavior

- **Context:** Component tests need to verify functionality without full system complexity
- **Options Considered:**
 1. Pure mocking with stub implementations
 2. Real dependencies with full shells and terminals
 3. Minimal real dependencies with controlled test programs
- **Decision:** Use minimal real dependencies with controlled test programs
- **Rationale:** Pure mocks miss integration issues between components and system interfaces, while full dependencies make tests non-deterministic and environment-dependent. Controlled test programs provide real system behavior with predictable, testable outcomes.
- **Consequences:** Tests exercise real system interfaces and catch platform-specific issues while remaining fast and deterministic. Test programs must be maintained alongside the main codebase.

Integration Testing

Integration testing verifies that components work correctly together and that the complete terminal multiplexer handles real-world scenarios with actual shell processes, PTY devices, and terminal output. These tests focus on data flow between components, timing-dependent behavior, and end-to-end functionality.

The **PTY-to-display pipeline** represents the most critical integration path, where output from shell processes flows through PTY devices, escape sequence parsing, screen buffer updates, and final terminal rendering. Integration tests must verify this pipeline handles realistic shell output correctly.

Integration Scenario	Test Setup	Expected Results	Verification Method
Shell Command Output	Execute <code>ls -la</code> in pane	Directory listing displayed correctly	Compare rendered output to expected text
Interactive Programs	Run <code>vim</code> or <code>less</code> with known files	Screen clearing and cursor movement work	Verify escape sequences parsed and applied
Color and Formatting	Execute commands with colored output	Colors and attributes displayed correctly	Check terminal cells have correct color values
Large Output Volumes	Generate substantial text output	Scrollbar buffer manages content correctly	Verify old lines preserved and accessible
Real-time Output	Run streaming commands like <code>tail -f</code>	Output appears as generated	Check display updates without artificial delays
Multiple Pane Output	Simultaneous activity in multiple panes	Each pane displays independently	Verify output doesn't cross pane boundaries

Integration tests should use realistic shell commands and verify that the complete output appears correctly in the terminal multiplexer display. These tests can capture expected output from known commands and compare against actual multiplexer rendering results.

The **multi-pane interaction testing** verifies that window management operations work correctly with live PTY sessions and that focus changes, splits, and resizes maintain proper terminal behavior across all panes.

Integration Scenario	Test Setup	Expected Results	Verification Method
Pane Creation	Split existing pane running shell	New shell starts in new pane	Verify independent shell processes
Focus Switching	Switch between panes with different content	Active pane receives input correctly	Send keystrokes and verify only active pane responds
Pane Resizing	Resize panes containing running programs	Programs adjust to new dimensions	Check program output adapts to new size
Pane Closing	Close pane with running process	Process terminates cleanly	Verify process cleanup and layout adjustment
Terminal Resize	Change terminal size with multiple panes	All panes resize proportionally	Verify programs in all panes see size changes
Session Persistence	Long-running processes across operations	Processes continue running during layout changes	Check process state maintained through operations

Multi-pane integration tests should use long-running or interactive programs to verify that pane operations don't disrupt process execution. Programs like text editors, system monitors, or simple loops provide observable behavior for verification.

The **input routing verification** ensures that keyboard input reaches the correct destinations based on multiplexer state, prefix key detection works reliably, and command mode operations execute correctly.

Integration Scenario	Test Setup	Expected Results	Verification Method
Normal Input Routing	Type in active pane	Characters appear in shell	Verify shell receives keystrokes
Prefix Key Detection	Press configured prefix key	Multiplexer enters command mode	Check mode transition and visual feedback
Command Execution	Execute split, resize, switch commands	Operations complete correctly	Verify layout changes and focus updates
Complex Key Sequences	Multi-byte characters, function keys	Special keys handled correctly	Check UTF-8 decoding and escape sequence handling
Input Mode Transitions	Switch between command and normal modes	Mode changes work reliably	Verify prefix timeout and mode indicators
Signal Forwarding	Send Ctrl-C, Ctrl-Z to active pane	Signals reach child processes	Check process signal handling

Input routing tests should simulate realistic typing patterns and keyboard combinations while monitoring both multiplexer state changes and shell process responses. These tests can use automated input generation with verification of expected outcomes.

Error Recovery Integration Testing verifies that the multiplexer handles failure conditions gracefully and maintains system stability when components encounter errors or unexpected conditions.

Error Scenario	Trigger Condition	Expected Recovery	Verification Method
Child Process Death	Kill shell process in pane	Pane marked as terminated, layout preserved	Check pane status and continued multiplexer operation
PTY Allocation Failure	Exhaust PTY devices	Graceful error reporting	Verify error messages and continued operation
Terminal Size Errors	Invalid window size updates	Fallback to previous valid size	Check dimension validation and error handling
Escape Sequence Corruption	Malformed terminal output	Parser recovery without display corruption	Verify display remains consistent
Memory Allocation Failures	Large scrollback buffers	Graceful degradation or cleanup	Check memory management and error reporting
Signal Handling Errors	Rapid signal delivery	Consistent internal state	Verify signal handling robustness

Error recovery tests should deliberately trigger failure conditions and verify that the multiplexer responds appropriately without crashing or corrupting its internal state. These tests help ensure reliable operation in production environments.

Integration Testing Insight: Many terminal multiplexer bugs only manifest under specific timing conditions or with particular combinations of shell programs and user input. Integration tests must exercise realistic usage patterns with sufficient duration to catch these intermittent issues.

Milestone Verification Checkpoints

Milestone verification checkpoints provide concrete acceptance criteria and testing procedures to validate that each development milestone delivers the expected functionality. These checkpoints serve as quality gates to ensure solid foundations before progressing to more advanced features.

Milestone 1: PTY Creation Verification

The PTY creation milestone establishes the foundational capability to spawn and manage shell sessions through pseudo-terminal devices. Verification focuses on correct PTY handling, process management, and basic terminal I/O functionality.

Verification Test	Procedure	Expected Outcome	Success Criteria
PTY Allocation	Call <code>pty_session_create</code> with <code>/bin/bash</code>	Valid session with open file descriptors	<code>master_fd</code> and <code>slave_fd</code> both positive
Process Spawning	Check child process creation	Shell process running with correct PID	<code>child_pid</code> valid and <code>process_running</code> true
Basic I/O	Write command to PTY, read response	Shell executes command and returns output	Command output appears in read buffer
Terminal Attributes	Create session and check terminal settings	PTY configured with proper terminal attributes	Terminal settings enable interactive use
Window Size Setting	Call <code>pty_session_resize</code> with dimensions	Child process receives size update	Process sees correct window size via ioctl
Session Cleanup	Call <code>pty_session_destroy</code> on active session	All resources cleaned up properly	File descriptors closed, process terminated

Manual Verification Steps for Milestone 1:

- 1. Compile the PTY test program:** Build a simple test program that creates a PTY session and enters an interactive loop
- 2. Run the test program:** Execute and verify it creates a shell session successfully
- 3. Type shell commands:** Enter commands like `echo hello`, `pwd`, `ls` and verify responses appear
- 4. Test window resizing:** Change terminal size and verify the shell receives size updates
- 5. Exit cleanly:** Terminate the test program and verify no processes remain and no file descriptors leak

The milestone 1 verification should demonstrate that PTY sessions behave identically to direct shell interaction, with commands executing normally and output appearing as expected.

Milestone 2: Terminal Emulation Verification

The terminal emulation milestone implements escape sequence parsing and virtual screen buffer management. Verification focuses on correct handling of cursor movement, text attributes, and screen manipulation commands.

Verification Test	Procedure	Expected Outcome	Success Criteria
Cursor Movement	Send cursor positioning escape sequences	Cursor moves to specified coordinates	<code>cursor_x</code> and <code>cursor_y</code> values correct
Screen Clearing	Send clear screen sequences	Buffer cleared appropriately	Screen buffer contains expected empty cells
Text Attributes	Send bold, underline, color sequences	Attributes applied to subsequent text	Character cells have correct attribute flags
Scrolling Operations	Fill screen and add more lines	Content scrolls correctly	Old lines move to scrollback buffer
UTF-8 Characters	Send multi-byte Unicode sequences	Characters displayed correctly	Wide characters occupy correct cell count
Complex Programs	Run programs that use escape sequences	Programs display correctly	<code>vim</code> , <code>htop</code> , <code>less</code> render properly

Manual Verification Steps for Milestone 2:

- Run cursor movement tests:** Execute commands that generate cursor positioning sequences and verify virtual cursor tracks correctly
- Test color programs:** Run commands that produce colored output like `ls --color` and verify colors appear in screen buffer
- Run full-screen programs:** Execute `vim` or `less` and verify screen clearing and cursor positioning work correctly
- Test scrolling behavior:** Generate output longer than screen height and verify scrollback buffer captures content
- Verify Unicode support:** Display files containing Unicode characters and verify correct rendering

The milestone 2 verification should demonstrate that complex terminal programs render correctly in the virtual screen buffer and that all common escape sequences produce expected results.

Milestone 3: Window Management Verification

The window management milestone implements pane splitting, layout calculation, and multi-pane rendering. Verification focuses on correct space allocation, independent pane operation, and visual layout rendering.

Verification Test	Procedure	Expected Outcome	Success Criteria
Horizontal Split	Split single pane horizontally	Two panes with independent shells	Each pane has separate PTY session
Vertical Split	Split pane vertically	Panes divided along vertical boundary	Width allocation correct for both panes
Nested Splits	Create complex split hierarchy	All panes receive correct dimensions	Layout tree calculates positions accurately
Pane Focus	Switch focus between multiple panes	Input directed to active pane only	Only focused pane receives keystrokes
Pane Resizing	Adjust pane boundaries	Panes resize maintaining minimums	Dimension changes propagated to PTY sessions
Multi-Pane Rendering	Display all panes simultaneously	Complete terminal output with borders	All pane content visible with clear separation

Manual Verification Steps for Milestone 3:

- Create initial pane:** Start multiplexer and verify single pane occupies full terminal
- Split panes:** Create horizontal and vertical splits and verify independent shell sessions
- Test focus switching:** Move between panes and verify input goes to correct shell
- Verify independent operation:** Run different commands in each pane simultaneously
- Test visual rendering:** Verify pane borders and status information display correctly
- Resize terminal:** Change terminal size and verify all panes resize appropriately

The milestone 3 verification should demonstrate that multiple shell sessions operate independently in separate screen regions with proper visual separation and focus management.

Milestone 4: Key Bindings and UI Verification

The key bindings and UI milestone implements command mode, configurable key bindings, status display, and complete user interface functionality. Verification focuses on reliable prefix key detection, command execution, and status information display.

Verification Test	Procedure	Expected Outcome	Success Criteria
Prefix Key Detection	Press configured prefix key	Multiplexer enters command mode	Visual indicator shows command mode active
Pane Commands	Execute split, close, navigate commands	Operations complete correctly	Layout changes and focus updates occur
Key Binding Customization	Configure different key bindings	Custom bindings work correctly	Alternative keys trigger same operations
Status Bar Display	Verify status information rendering	Status shows current pane and session info	Pane numbers, time, session name visible
Copy Mode	Enter copy mode and select text	Text selection and copying works	Selected text available in clipboard
Terminal Restoration	Exit multiplexer	Terminal returns to normal state	Original terminal settings restored

Manual Verification Steps for Milestone 4:

- 1. Test prefix key:** Press Ctrl-B (or configured prefix) and verify command mode indicator
- 2. Execute pane commands:** Use key bindings to split, close, and navigate between panes
- 3. Verify status display:** Check that status bar shows accurate pane and session information
- 4. Test copy mode:** Enter copy mode, select text, and verify copying functionality
- 5. Customize bindings:** Modify key bindings and verify new combinations work
- 6. Exit cleanly:** Quit multiplexer and verify terminal restored to original state

The milestone 4 verification should demonstrate a fully functional terminal multiplexer with all user interface features working correctly and reliable terminal mode management.

Milestone Checkpoint Strategy: Each milestone checkpoint should be achievable within a focused development sprint and provide clear, observable success criteria. The verification procedures bridge the gap between automated testing and user acceptance by providing concrete manual validation steps.

Architecture Decision: Automated vs Manual Verification

Decision: Combine Automated Testing with Manual Verification Checkpoints

- **Context:** Milestone verification needs to validate both technical correctness and user experience
- **Options Considered:**
 1. Fully automated testing with programmatic verification
 2. Manual testing checklists with human verification
 3. Hybrid approach combining automated tests with manual validation
- **Decision:** Use hybrid approach with automated tests for technical correctness and manual checkpoints for user experience
- **Rationale:** Terminal multiplexers have complex user interactions and visual behavior that are difficult to test programmatically, while technical components benefit from automated verification. Manual checkpoints catch usability issues and integration problems that automated tests miss.
- **Consequences:** Verification process requires both automated test execution and human validation, but provides comprehensive coverage of both technical and user experience requirements.

Implementation Guidance

The testing strategy for a terminal multiplexer requires a combination of isolated component testing, integration verification with real processes, and milestone-driven acceptance testing. This implementation guidance provides complete test infrastructure and specific testing procedures for each component and milestone.

Technology Recommendations for Testing:

Testing Layer	Simple Option	Advanced Option
Unit Testing	Simple assert macros with custom test runner	CUnit or Check testing framework
Integration Testing	Shell script-based testing with controlled programs	Expect/TCL-based terminal automation
Process Mocking	Simple test programs that echo/exit	Full pseudo-shell implementations
Terminal Capture	String comparison of rendered output	Terminal recording and playback tools
Memory Testing	Manual leak checking	Valgrind integration for memory verification
Coverage Analysis	Manual code review	gcov/lcov for coverage reporting

Recommended Test File Structure:

```
terminal-multiplexer/
src/
  pty_manager.c      ← component implementation
  terminal_emulator.c ← component implementation
  window_manager.c   ← component implementation
  input_handler.c    ← component implementation
tests/
  unit/
    test_escape_parser.c ← parser unit tests
    test_layout_calc.c   ← layout algorithm tests
    test_screen_buffer.c ← screen buffer tests
    test_pty_session.c   ← PTY session tests
  integration/
    test_pty_pipeline.c ← PTY to display pipeline
    test_multi_pane.c   ← multi-pane interactions
    test_input_routing.c ← input handling integration
  test_programs/
    echo_program.c      ← simple echo for PTY testing
    color_program.c    ← generates colored output
    cursor_program.c   ← cursor movement sequences
  milestone_tests/
    verify_milestone1.c ← M1 acceptance tests
    verify_milestone2.c ← M2 acceptance tests
    verify_milestone3.c ← M3 acceptance tests
    verify_milestone4.c ← M4 acceptance tests
  test_framework.h     ← test utilities and macros
Makefile              ← build configuration with test targets
```

Test Framework Infrastructure (Complete Implementation):

```
// tests/test_framework.h - Complete test infrastructure

#ifndef TEST_FRAMEWORK_H
#define TEST_FRAMEWORK_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <sys/wait.h>

// Test result tracking

typedef struct {

    int tests_run;
    int tests_passed;
    int tests_failed;
    char last_failure[256];
} test_results_t;

static test_results_t g_test_results = {0};

// Test assertion macros

#define TEST_ASSERT(condition, message) do { \
    g_test_results.tests_run++; \
    if (condition) { \
        g_test_results.tests_passed++; \
        printf("PASS: %s\n", message); \
    } else { \
        g_test_results.tests_failed++; \
        snprintf(g_test_results.last_failure, sizeof(g_test_results.last_failure), \

```

C

```

        "FAIL: %s", message); \
        printf("%s\n", g_test_results.last_failure); \
    } \
} while(0)

#define TEST_ASSERT_EQ(expected, actual, message) \
TEST_ASSERT((expected) == (actual), message)

#define TEST_ASSERT_STR_EQ(expected, actual, message) \
TEST_ASSERT(strcmp((expected), (actual)) == 0, message)

#define TEST_ASSERT_NOT_NULL(ptr, message) \
TEST_ASSERT((ptr) != NULL, message)

// Test suite management

#define TEST_SUITE_BEGIN(name) \
printf("\n==== Running Test Suite: %s ====\n", name); \
test_results_t suite_start = g_test_results;

#define TEST_SUITE_END(name) do { \
    int suite_tests = g_test_results.tests_run - suite_start.tests_run; \
    int suite_passed = g_test_results.tests_passed - suite_start.tests_passed; \
    printf("==== %s Complete: %d/%d passed ====\n", name, suite_passed, suite_tests); \
} while(0)

// Test program creation utilities

int create_test_program(const char* program_code, const char* filename);

int run_test_with_timeout(void (*test_func)(void), int timeout_seconds);

void cleanup_test_files(void);

#endif // TEST_FRAMEWORK_H

```

Component Unit Test Templates:

```
// tests/unit/test_escape_parser.c - Escape sequence parser tests

C

#include "../test_framework.h"

#include "../../src/terminal_emulator.h"

void test_basic_cursor_movement(void) {

    escape_parser_t parser;

    screen_buffer_t buffer;

    // TODO 1: Initialize parser and buffer with known state

    // TODO 2: Process cursor up sequence "\e[A" byte by byte

    // TODO 3: Verify parser generates correct cursor movement action

    // TODO 4: Apply action to buffer and verify cursor position changed

    // TODO 5: Test with parameter variations like "\e[5A" for 5-line movement

    TEST_ASSERT_EQ(PARSER_STATE_NORMAL, parser.state, "Parser returns to normal state");

    TEST_ASSERT_EQ(expected_y, buffer.cursor_y, "Cursor moved to expected position");

}

void test_csi_parameter_parsing(void) {

    escape_parser_t parser;

    // TODO 1: Initialize parser in normal state

    // TODO 2: Process sequence "\e[2;5H" character by character

    // TODO 3: Verify parser correctly accumulates parameters in csi_params array

    // TODO 4: Check parameter count matches expected value

    // TODO 5: Verify individual parameter values are parsed correctly

    TEST_ASSERT_EQ(2, parser.csi_param_count, "Correct number of parameters parsed");

    TEST_ASSERT_EQ(2, parser.csi_params[0], "First parameter parsed correctly");
}
```

```
TEST_ASSERT_EQ(5, parser.csi_params[1], "Second parameter parsed correctly");

}

void test_utf8_character_handling(void) {

    escape_parser_t parser;

    screen_buffer_t buffer;

    // TODO 1: Initialize parser and buffer

    // TODO 2: Process multi-byte UTF-8 sequence for Unicode character

    // TODO 3: Verify parser correctly assembles complete codepoint

    // TODO 4: Check character is stored correctly in terminal cell

    // TODO 5: Verify wide characters occupy appropriate number of cells

    TEST_ASSERT_EQ(expected_codepoint, buffer.cells[0].ch, "Unicode character decoded
correctly");

}

int main(void) {

    TEST_SUITE_BEGIN("Escape Parser");

    test_basic_cursor_movement();
    test_csi_parameter_parsing();
    test_utf8_character_handling();

    TEST_SUITE_END("Escape Parser");

    printf("\nOverall Results: %d/%d tests passed\n",
        g_test_results.tests_passed, g_test_results.tests_run);

    return g_test_results.tests_failed == 0 ? 0 : 1;
}
```

}

Integration Test Templates:

```
// tests/integration/test_pty_pipeline.c - PTY to display pipeline testing C

#include "../test_framework.h"

#include "../../src/pty_manager.h"

#include "../../src/terminal_emulator.h"

void test_shell_command_output(void) {

    pty_session_t session;

    screen_buffer_t buffer;

    escape_parser_t parser;

    // TODO 1: Create PTY session with test shell program

    // TODO 2: Send command "echo hello world" to PTY

    // TODO 3: Read output from PTY master file descriptor

    // TODO 4: Process output through escape parser into screen buffer

    // TODO 5: Verify expected text appears in buffer cells

    // TODO 6: Clean up PTY session and verify no resource leaks

    TEST_ASSERT_STR_EQ("hello world", extracted_text, "Command output appears correctly");

    TEST_ASSERT(pty_session_is_alive(&session), "Shell process still running");

}

void test_interactive_program_integration(void) {

    pty_session_t session;

    window_manager_t wm;

    // TODO 1: Create window manager with single pane

    // TODO 2: Start interactive program (use test program that sends cursor sequences)

    // TODO 3: Allow program to initialize and send escape sequences

    // TODO 4: Capture rendered output from window manager
```

```
// TODO 5: Verify escape sequences were processed and display updated correctly

// TODO 6: Send input to program and verify it responds appropriately


TEST_ASSERT_NOT_NULL(rendered_output, "Program generated display output");

TEST_ASSERT(strstr(rendered_output, expected_content) != NULL, "Expected content in
output");

}

int main(void) {

TEST_SUITE_BEGIN("PTY Pipeline Integration");

test_shell_command_output();
test_interactive_program_integration();

TEST_SUITE_END("PTY Pipeline Integration");

return g_test_results.tests_failed == 0 ? 0 : 1;
}
```

Test Program Implementations:

```
// tests/test_programs/cursor_program.c - Program that generates cursor sequences
C

#include <stdio.h>

#include <unistd.h>

int main(void) {

    // Clear screen and position cursor

    printf("\x1b[2J\x1b[H");

    // Write text at specific positions

    printf("\x1b[5;10HHello");

    printf("\x1b[7;15HWorld");

    // Move cursor and write more text

    printf("\x1b[10;1HCursor test complete");

    fflush(stdout);

    return 0;
}
```

Milestone Verification Programs:

```
// tests/milestone_tests/verify_milestone1.c - PTY creation verification
```

C

```
#include "../test_framework.h"
```

```
#include "../../src/pty_manager.h"
```

```
void verify_pty_allocation(void) {
```

```
    pty_session_t session;
```

```
    // TODO 1: Call pty_session_create with /bin/bash
```

```
    // TODO 2: Verify master_fd and slave_fd are valid file descriptors
```

```
    // TODO 3: Check child_pid is positive and process_running is true
```

```
    // TODO 4: Verify session can read/write data successfully
```

```
    // TODO 5: Test window size setting with pty_session_resize
```

```
    // TODO 6: Clean up session and verify proper resource cleanup
```

```
    TEST_ASSERT(session.master_fd > 0, "Master FD allocated");
```

```
    TEST_ASSERT(session.child_pid > 0, "Child process spawned");
```

```
    TEST_ASSERT(session.process_running, "Process marked as running");
```

```
}
```

```
void verify_basic_shell_interaction(void) {
```

```
    pty_session_t session;
```

```
    char buffer[1024];
```

```
    // TODO 1: Create PTY session
```

```
    // TODO 2: Write shell command to PTY
```

```
    // TODO 3: Read response from PTY
```

```
    // TODO 4: Verify command executed and produced expected output
```

```
    // TODO 5: Test multiple commands to verify persistent shell state
```

```
TEST_ASSERT_NOT_NULL(strstr(buffer, expected_output), "Shell command executed correctly");

}

int main(void) {
    printf("== Milestone 1 Verification ==\n");
    printf("Testing PTY creation and basic shell interaction...\n");

    verify_pty_allocation();
    verify_basic_shell_interaction();

    if (g_test_results.tests_failed == 0) {
        printf("\n✓ Milestone 1 PASSED - PTY creation working correctly\n");
        printf("Ready to proceed to Milestone 2: Terminal Emulation\n");
    } else {
        printf("\n✗ Milestone 1 FAILED - Fix PTY issues before continuing\n");
        printf("Last failure: %s\n", g_test_results.last_failure);
    }

    return g_test_results.tests_failed == 0 ? 0 : 1;
}
```

Debugging and Testing Utilities:

Debugging Tool	Usage	Expected Output
<code>strace -e trace=ioctl ./test_program</code>	Trace PTY ioctl calls	Shows TIOCSWINSZ and terminal setup calls
<code>lsof -p \$PID</code>	Check file descriptor leaks	Should show only expected open files
<code>ps aux grep test</code>	Verify process cleanup	No orphaned test processes after completion
<code>hexdump -C output.txt</code>	Examine escape sequences	Shows exact bytes including control characters
<code>script -c ./multiplexer typescript</code>	Record terminal session	Captures all input/output for analysis

Milestone Checkpoint Commands:

```
# Milestone 1 verification
```

BASH

```
make test-milestone1
```

```
./tests/milestone_tests/verify_milestone1
```

```
# Expected output:
```

```
# === Milestone 1 Verification ===
```

```
# PASS: Master FD allocated
```

```
# PASS: Child process spawned
```

```
# PASS: Shell command executed correctly
```

```
# ✓ Milestone 1 PASSED
```

```
# Milestone 2 verification
```

```
make test-milestone2
```

```
./tests/milestone_tests/verify_milestone2
```

```
# Run interactive test
```

```
echo -e "\e[2J\e[H\e[31mRed text\e[0m" | ./test_terminal_emulator
```

```
# Expected: Screen clears, cursor moves to home, "Red text" appears in red
```

```
# Milestone 3 verification
```

```
make test-milestone3
```

```
./tests/milestone_tests/verify_milestone3
```

```
# Manual verification: Start multiplexer, split panes, verify independent shells
```

```
# Milestone 4 verification
```

```
make test-milestone4
```

```
./tests/milestone_tests/verify_milestone4
```

```
# Manual verification: Test all key bindings and UI features
```

Debugging Guide

Milestone(s): All milestones (1-4) - this section provides comprehensive debugging approaches that span PTY functionality (M1), terminal emulation accuracy (M2), window management behavior (M3), and input handling correctness (M4)

Mental Model: The Detective's Toolkit

Think of debugging a terminal multiplexer like being a detective investigating a complex crime scene. You have multiple suspects (components) that could be causing the problem, various types of evidence (logs, system calls, terminal output), and different investigative techniques (debugging tools, test cases, systematic observation). Just as a detective follows clues systematically from symptoms back to root causes, debugging a terminal multiplexer requires methodical investigation through the layers of PTY management, terminal emulation, window management, and input handling.

The key insight is that terminal multiplexer bugs often manifest in one component but originate in another. A rendering problem might stem from corrupted escape sequence parsing, which could be caused by incomplete UTF-8 decoding, which might result from improper PTY I/O handling. Like solving a mystery, you need to trace the evidence backwards through the data flow to find the true culprit.

Building a terminal multiplexer involves orchestrating multiple complex systems (process management, terminal emulation, signal handling, I/O multiplexing) that interact in subtle ways. When things go wrong, the symptoms can be confusing and misleading. This debugging guide provides systematic approaches to diagnose issues across all components and integration points.

The debugging process for terminal multiplexers differs from typical application debugging because you're dealing with low-level system interfaces (PTYs, signals, terminal control), real-time I/O processing, and stateful terminal emulation. Problems can be timing-dependent, process-specific, or terminal-dependent, making them challenging to reproduce and isolate.

PTY and Process Issues

Mental Model: The Nursery Supervisor

Debugging PTY and process issues is like being a nursery supervisor responsible for multiple children (processes) in separate rooms (PTYs). When a child stops responding, you need to check if they're sleeping (blocked), sick (crashed), or if their room's intercom system (PTY) is broken. Sometimes the problem isn't with the child but with the communication system connecting you to them.

PTY and process debugging involves understanding the complex interactions between parent and child processes, pseudo-terminal devices, process groups, and signal handling. These issues often manifest as hanging sessions, unresponsive shells, or mysterious process termination.

Common PTY Creation and Management Problems

The most frequent category of PTY issues occurs during the initial creation and setup phase. These problems often result from incorrect file descriptor management, improper session setup, or failure to establish the controlling terminal relationship correctly.

Problem Category	Symptom	Root Cause	Detection Method
PTY Allocation Failure	<code>pty_session_create()</code> returns <code>TMUX_ERROR_PTY_ALLOCATION</code>	System limit reached or permissions issue	Check <code>errno</code> after <code>posix_openpt()</code> call
Child Process Hangs	Shell never appears, no prompt displayed	Child didn't become session leader or slave not opened correctly	Check if <code>setsid()</code> succeeded and slave opened in child
Input Not Reaching Shell	Typing produces no response	Master/slave file descriptors swapped or not connected	Verify data flow with write to master, read from slave
Garbled Output	Text appears corrupted or missing	Terminal attributes not set correctly	Compare <code>termios</code> settings between real terminal and PTY

⚠ Pitfall: File Descriptor Leaks Across Fork

One of the most common mistakes is failing to properly manage file descriptors across the fork boundary. The parent process should close the slave file descriptor after forking, and the child should close the master. Leaving both ends open in both processes can cause I/O to hang or behave unpredictably.

```
// WRONG - both processes keep both file descriptors C

if (fork() == 0) {

    // Child still has master_fd open - this is wrong

    dup2(slave_fd, STDIN_FILENO);

    // ...

}

// Parent still has slave_fd open - this is also wrong

// CORRECT - each process closes the file descriptor it doesn't need

pid_t pid = fork();

if (pid == 0) {

    close(master_fd); // Child closes master

    setsid(); // Become session leader

    dup2(slave_fd, STDIN_FILENO);

    // ...

} else {

    close(slave_fd); // Parent closes slave

    // ...

}
```

The debugging approach for file descriptor issues involves checking `/proc/PID/fd/` for both parent and child processes to verify which file descriptors are open and ensuring they point to the expected devices.

Process Lifecycle and Signal Handling Issues

Process management problems often involve incorrect signal handling, failure to detect child termination, or improper cleanup when processes exit unexpectedly.

Signal Issue	Symptom	Debugging Approach	Solution
SIGCHLD Not Handled	Zombie processes accumulate	Check <code>ps aux</code> for <code><defunct></code> processes	Install SIGCHLD handler with <code>SA_NOCLDWAIT</code>
Child Termination Undetected	Pane remains active after shell exits	Monitor <code>pty_session_is_alive()</code> return value	Use <code>waitpid()</code> with <code>WNOHANG</code> to check status
Signal Delivery to Wrong Process	Ctrl-C doesn't work in shell	Check process group IDs with <code>ps -o pid,ppid,pgid,sid</code>	Ensure child calls <code>setsid()</code> before exec
Race Condition on Cleanup	Segmentation fault during session destruction	Use debugging tools to check cleanup order	Implement reference counting for session objects

Decision: Signal Handler Safety

- **Context:** Signal handlers can interrupt any code, creating race conditions with normal program flow
- **Options Considered:**
 - Self-pipe trick to convert signals to file descriptor events
 - Signal masking with `signalfd()` on Linux
 - Traditional signal handlers with atomic flags
- **Decision:** Use self-pipe trick for signal-safe event loop integration
- **Rationale:** Portable across Unix systems and integrates cleanly with `select()`-based event loop
- **Consequences:** Requires additional file descriptor and pipe management but eliminates race conditions

The self-pipe trick involves writing a byte to a pipe from the signal handler and monitoring the pipe's read end in the main event loop. This converts asynchronous signals into synchronous file descriptor events.

⚠ Pitfall: Race Conditions in Process Group Management

Setting up process groups correctly requires careful attention to timing. The child process must call `setsid()` before opening the slave PTY, and the parent must handle the case where the child terminates before the PTY setup completes.

PTY I/O and Communication Debugging

PTY I/O issues manifest as data corruption, partial reads, or communication timeouts. These problems often stem from incorrect assumptions about PTY behavior or failure to handle partial I/O operations.

I/O Issue	Detection Method	Common Cause	Debugging Technique
Partial Reads	<code>pty_session_read()</code> returns fewer bytes than expected	PTY operates in packet mode	Log all read sizes and buffer contents
Write Blocking	Process hangs during <code>pty_session_write()</code>	Child process not reading, buffer full	Check PTY buffer size with <code>ioctl(FIONREAD)</code>
Data Corruption	Characters appear wrong or missing	UTF-8 sequences split across reads	Hexdump raw PTY data before processing
EOF Handling	Read returns 0 but child still running	Child closed stdout but not stderr	Monitor both stdout and stderr file descriptors

The key insight for PTY I/O debugging is that PTYs don't guarantee atomic reads or writes for arbitrary data sizes. Applications must handle partial I/O operations and buffer management correctly.

Process State Inspection Techniques

When debugging process issues, systematic inspection of process state provides crucial diagnostic information:

- Process Tree Analysis:** Use `pstree -p` to visualize the relationship between the terminal multiplexer, child shells, and any subprocesses they've spawned
- File Descriptor Audit:** Examine `/proc/PID/fd/` for each process to verify PTY file descriptors are open and point to expected devices
- Process Group Verification:** Check process group and session IDs using `ps -o pid,ppid,pgid,sid,tty` to ensure session leadership is established correctly
- Signal Mask Inspection:** Use `/proc/PID/status` to check which signals are blocked or ignored in each process
- Memory Map Analysis:** For crashes, examine `/proc/PID/maps` to identify memory corruption or unexpected library loading

Terminal Emulation Debugging

Mental Model: The Language Translator

Debugging terminal emulation is like troubleshooting a simultaneous translation system. The terminal emulator receives a stream of mixed languages (regular text + control codes) and must parse them correctly to produce the right display output. When translation fails, you might see garbled text (parsing errors), missing words (incomplete sequences), or text in the wrong place (cursor tracking bugs).

Terminal emulation debugging focuses on the accuracy of escape sequence parsing, screen buffer management, and the complex state transitions involved in maintaining a virtual terminal display.

Escape Sequence Parsing Issues

Escape sequence parsing problems are among the most common terminal emulation bugs. They manifest as incorrect cursor positioning, missing text formatting, or garbled display output.

Parsing Issue	Visual Symptom	Root Cause	Diagnostic Approach
Incomplete Sequence Handling	Text appears with escape characters visible	Parser doesn't buffer partial sequences across reads	Log raw input stream and parser state transitions
Parameter Parsing Errors	Cursor moves wrong distance or colors incorrect	CSI parameter parsing fails for edge cases	Test with sequences having missing/multiple parameters
UTF-8 Boundary Issues	Foreign characters appear corrupted	Multibyte characters split across I/O boundaries	Verify UTF-8 decoder handles partial sequences
State Machine Corruption	Parser stuck, no longer processes sequences	Invalid state transition or missing reset logic	Add state validation and recovery mechanisms

⚠ Pitfall: Assuming Complete Escape Sequences

A critical mistake is assuming that each call to `pty_session_read()` contains complete escape sequences. PTY reads can split escape sequences at arbitrary boundaries, requiring the parser to buffer incomplete sequences and resume parsing when more data arrives.

```
// WRONG - assumes complete sequences

char buffer[1024];

int bytes = pty_session_read(session, buffer, sizeof(buffer));

escape_parser_process_byte(parser, screen_buffer, buffer[0]); // Only processes first byte

// CORRECT - processes all bytes and handles partial sequences

for (int i = 0; i < bytes; i++) {

    escape_parser_process_byte(parser, screen_buffer, buffer[i]);

}
```

The escape sequence parser must maintain state across multiple read operations, buffering incomplete sequences until they can be completed.

Screen Buffer State Debugging

Screen buffer corruption manifests as incorrect text placement, missing characters, or cursor tracking errors. These issues often result from bounds checking failures or incorrect scrolling logic.

Buffer Issue	Symptom	Debugging Method	Prevention
Cursor Out of Bounds	Crash or corruption when writing	Add assertions for cursor position validity	Clamp cursor to screen boundaries
Scroll Buffer Overflow	Old lines disappear unexpectedly	Monitor scrollback buffer size	Implement circular buffer with size limits
Attribute Inconsistency	Text formatting appears wrong	Compare stored attributes with expected values	Log attribute changes with screen position
Line Wrapping Errors	Text appears on wrong lines	Trace cursor movement during long line input	Implement proper line wrapping state tracking

Decision: Screen Buffer Representation

- **Context:** Need to store terminal content with per-character attributes and handle scrolling efficiently
- **Options Considered:**
 - Array of lines, each line is array of characters
 - Flat array with row/column indexing
 - Circular buffer for scrollback with separate visible area
- **Decision:** Array of lines with separate circular scrollback buffer
- **Rationale:** Matches terminal semantics, simplifies line-oriented operations like scrolling and clearing
- **Consequences:** Enables efficient scrolling but requires careful index management for scrollback

The screen buffer debugging process involves validating buffer state consistency at key points: after cursor movement, after character insertion, and after scrolling operations.

Terminal Attribute and Color Handling

Color and text attribute bugs often manifest as incorrect visual formatting or attributes that persist when they shouldn't. These issues typically involve state management problems in the attribute tracking system.

Attribute Issue	Visual Effect	Root Cause Analysis	Fix Strategy
Color Bleeding	Colors appear in wrong places	Attribute reset not handled correctly	Verify SGR 0 (reset) sequence processing
Bold/Underline Stuck	Formatting persists after reset	Current attribute state not updated	Track attribute changes in parser state
256-Color Mode Issues	Colors appear wrong or default	Color palette not initialized or indexed incorrectly	Validate color index bounds and palette setup
Saved Cursor Attributes	Formatting wrong after cursor restore	Saved cursor doesn't include text attributes	Store complete cursor state including attributes

The terminal attribute system requires careful state management because attributes are modal - they persist until explicitly changed or reset. Debugging attribute issues involves tracking the complete attribute state through all sequences that modify it.

UTF-8 and Character Width Debugging

UTF-8 and wide character handling presents unique challenges because character encoding and display width don't always align simply. These issues often manifest as text misalignment or corrupted display of international characters.

Character Issue	Display Problem	Debugging Technique	Solution
Incomplete UTF-8 Sequences	Question marks or corrupted characters	Hexdump raw bytes vs decoded codepoints	Buffer incomplete sequences across reads
Wide Character Positioning	Asian text overlaps or has gaps	Check terminal cell occupation for wide chars	Use <code>utf8_is_wide_character()</code> for width calculation
Combining Character Handling	Diacritical marks appear separately	Test with composed vs decomposed Unicode forms	Implement grapheme cluster detection
Emoji and Unicode Blocks	Modern emoji render incorrectly	Verify Unicode database version and width data	Update to current Unicode width specifications

Decision: UTF-8 Decoding Strategy

- **Context:** PTY data can contain incomplete UTF-8 sequences split across read boundaries
- **Options Considered:**
 - Decode each read buffer independently, discard incomplete sequences
 - Buffer incomplete sequences and resume decoding on next read
 - Convert to wide characters immediately for processing
- **Decision:** Buffer incomplete sequences with resume capability
- **Rationale:** Preserves character integrity and handles real-world PTY I/O patterns correctly
- **Consequences:** Requires stateful UTF-8 decoder but ensures no data loss

Debugging Tools and Techniques

Mental Model: The Forensic Laboratory

Think of debugging tools as a forensic laboratory equipped with specialized instruments for analyzing different types of evidence. Just as forensic scientists use microscopes for physical evidence, chemical tests for substances, and DNA analysis for biological samples, terminal multiplexer debugging requires different tools for different types of problems: system call tracers for I/O issues, terminal capture tools for emulation problems, and memory analyzers for crash investigation.

System Call Tracing and Analysis

System call tracing provides the most detailed view of how the terminal multiplexer interacts with the kernel, revealing PTY operations, signal delivery, and process management activities that are invisible at the application level.

Tool	Use Case	Key Options	Example Command
strace	Trace system calls and signals	<code>-f</code> follow forks, <code>-e</code> filter calls, <code>-o</code> output file	<code>strace -f -e trace=openat,read,write,ioctl,clone,wait4 ./tmux</code>
ltrace	Trace library function calls	<code>-f</code> follow forks, <code>-e</code> filter functions	<code>ltrace -f -e 'posix_openpt+ioctl+tcsetattr' ./tmux</code>
perf	Performance profiling and tracing	<code>record</code> capture events, <code>report</code> analyze	<code>perf record -g ./tmux; perf report</code>
ftrace	Kernel-level tracing (Linux)	Event tracing, function graphs	<code>echo 'pty*' > /sys/kernel/debug/tracing/set_ftrace_filter</code>

The system call tracing approach for PTY debugging follows this systematic process:

- 1. Initial Trace Capture:** Run the terminal multiplexer under strace with comprehensive tracing to capture all system interactions
- 2. Call Sequence Analysis:** Examine the sequence of `openat()`, `ioctl()`, `read()`, `write()`, and `close()` calls related to PTY file descriptors
- 3. Signal Correlation:** Match SIGCHLD delivery with child process state changes and PTY I/O patterns
- 4. Error Code Investigation:** Focus on system calls that return errors (return value -1) and examine the `errno` values
- 5. Timing Analysis:** Look for patterns in call timing that might indicate race conditions or blocking I/O issues

⚠ Pitfall: Strace Output Interpretation

Strace output can be overwhelming and misleading if not interpreted carefully. Focus on the system calls that matter for your specific issue rather than trying to understand every call. Filter the trace to relevant call types and look for patterns rather than individual calls.

Terminal Capture and Replay Tools

Terminal capture tools allow you to record and analyze the exact sequence of data flowing between the terminal multiplexer and the terminal, enabling precise diagnosis of terminal emulation issues.

Tool Category	Tool Name	Purpose	Usage Pattern
Terminal Recording	<code>script</code>	Record terminal session with timing	<code>script -t terminal.log 2>timing.log</code>
Data Stream Capture	<code>tee</code> with named pipes	Split data streams for analysis	<code>mkfifo debug.pipe; tee debug.pipe</code>
Escape Sequence Analysis	<code>cat -v</code>	Show control characters visually	<code>cat -v logfile</code> to see escape sequences
Binary Data Analysis	<code>hexdump / xxd</code>	Examine raw byte sequences	<code>hexdump -C ptydump.raw</code>

The terminal capture debugging workflow involves several stages:

- 1. Data Stream Isolation:** Capture the raw data flowing from PTY to terminal emulator using file descriptor redirection or pipe tapping
- 2. Sequence Identification:** Use hexdump or specialized tools to identify escape sequences and their boundaries in the captured data
- 3. Parser State Reconstruction:** Manually trace through the escape sequence parser state machine using the captured data to identify where parsing fails
- 4. Expected vs Actual Comparison:** Generate known good escape sequences and compare with the captured problematic sequences
- 5. Reproduction with Synthetic Data:** Create minimal test cases that reproduce the parsing issue with controlled input sequences

Memory and Resource Analysis

Memory analysis tools help diagnose crashes, memory leaks, and resource management issues that are common in terminal multiplexers due to their complex process and buffer management.

Analysis Type	Tool	Key Features	Usage Example
Memory Error Detection	Valgrind	Leak detection, use-after-free, bounds checking	<code>valgrind --leak-check=full --track-fds=yes ./tmux</code>
Address Sanitization	GCC/Clang ASan	Compile-time instrumentation for memory errors	<code>gcc -fsanitize=address -g -o tmux tmux.c</code>
Core Dump Analysis	GDB	Stack trace, variable inspection, memory examination	<code>gdb ./tmux core.dump; bt; info registers</code>
Resource Monitoring	<code>lsof / ss</code>	File descriptor and network connection tracking	<code>lsof -p \$(pgrep tmux)</code>

Decision: Memory Management Strategy

- **Context:** Terminal multiplexers manage complex object lifetimes with PTY sessions, screen buffers, and layout trees
- **Options Considered:**
 - Manual memory management with explicit cleanup functions
 - Reference counting for shared objects
 - Arena allocation for short-lived objects
- **Decision:** Manual memory management with explicit cleanup and cleanup handlers
- **Rationale:** Provides predictable performance and clear resource ownership semantics
- **Consequences:** Requires careful cleanup ordering but avoids reference cycle complexities

Logging and Diagnostic Infrastructure

Building comprehensive logging into the terminal multiplexer enables systematic debugging of complex interaction issues that are difficult to reproduce or trace with external tools.

Log Category	Information Captured	Log Level	Example Entry
PTY Operations	Session creation, I/O operations, termination	DEBUG	[DEBUG] pty_session_read: fd=5, bytes=42, data="ls -la\r\n"
Parser State	Escape sequence parsing, state transitions	TRACE	[TRACE] escape_parser: state=CSI_PARAMS, byte=0x6D, params=[1,32]
Layout Changes	Pane splits, resizes, focus changes	INFO	[INFO] layout_split_pane: pane_id=2, type=VERTICAL, new_pane=3
Error Conditions	System call failures, validation errors	ERROR	[ERROR] pty_session_create: posix_openpt() failed: errno=24 (EMFILE)

The logging infrastructure should support runtime log level adjustment and structured output that can be easily parsed by analysis tools. Key design considerations include:

1. **Performance Impact Minimization:** Use conditional compilation or runtime checks to eliminate debug logging overhead in production builds
2. **Thread Safety:** Ensure log operations are atomic when called from signal handlers or multiple execution contexts
3. **Buffer Management:** Implement circular log buffers to prevent unbounded memory usage during long debugging sessions
4. **Contextual Information:** Include relevant context like pane IDs, session IDs, and timestamps in all log entries
5. **Binary Data Handling:** Provide hexdump formatting for raw PTY data and escape sequences

Systematic Debugging Methodology

Effective terminal multiplexer debugging requires a systematic approach that moves from symptom observation through hypothesis formation to root cause identification:

Phase 1: Symptom Documentation

1. Record the exact steps to reproduce the issue, including terminal size, shell type, and input sequences
2. Capture screenshots or terminal recordings showing the incorrect behavior
3. Note any error messages, crash information, or unusual process states
4. Identify which milestone's functionality is affected by the problem

Phase 2: Component Isolation

1. Determine whether the issue occurs in PTY management, terminal emulation, window management, or input handling
2. Create minimal test cases that isolate the problem to specific component interactions
3. Use component-level unit tests to verify individual component behavior
4. Check if the issue reproduces with different shells, terminal sizes, or input patterns

Phase 3: Data Flow Analysis

1. Trace data flow from the problematic input through all processing stages to the final output
2. Insert logging or debugging output at each stage to identify where the problem first appears
3. Compare actual data flow with expected behavior based on the design specifications
4. Identify any data transformation or state update that produces incorrect results

Phase 4: Root Cause Investigation

1. Use appropriate debugging tools based on the problem category (strace for PTY issues, memory tools for crashes, etc.)
2. Form specific hypotheses about the root cause and design tests to validate or refute them
3. Examine related code paths and error handling to understand the failure mechanism
4. Consider timing issues, race conditions, and state management problems

Phase 5: Fix Validation

1. Implement targeted fixes that address the identified root cause
2. Verify the fix resolves the original issue without introducing regressions
3. Add automated tests that prevent the issue from recurring
4. Update documentation and debugging guides with lessons learned

Integration Testing and End-to-End Validation

Integration testing for terminal multiplexers requires careful orchestration of real processes, terminals, and user input to validate correct behavior across all components working together.

Test Category	Validation Approach	Key Checkpoints	Automated Verification
PTY Integration	Spawn real shells, verify I/O	Shell prompt appears, commands execute	Compare expected vs actual output patterns
Terminal Compatibility	Test with various terminal types	Escape sequences render correctly	Automated screenshot comparison
Multi-Pane Behavior	Create splits, verify isolation	Independent scrolling, separate processes	Process tree validation, buffer content checks
Signal Handling	Send signals, verify propagation	Ctrl-C reaches correct process	Monitor signal delivery with strace

The integration testing framework should include:

- Test Environment Standardization:** Use consistent terminal settings, shell configurations, and system resources across test runs
- Output Verification:** Implement reliable methods for comparing terminal output that account for timing variations and terminal-specific rendering differences
- Process State Monitoring:** Continuously monitor process states, file descriptors, and resource usage during tests
- Failure Artifact Collection:** Automatically capture logs, core dumps, and system state when tests fail
- Performance Baseline Tracking:** Monitor resource usage and response times to detect performance regressions

Implementation Guidance

The debugging infrastructure for a terminal multiplexer requires careful design to provide comprehensive diagnostic capabilities without impacting normal operation performance.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	Printf with log levels	Structured logging with JSON output
Memory Debugging	Valgrind for development	AddressSanitizer for continuous integration
System Call Tracing	Basic strace usage	Custom ptrace-based tracer for detailed analysis
Performance Profiling	Simple timing measurements	Integrated perf counters and flame graphs

Recommended Debugging Infrastructure

```
src/
  debug/
    debug.h          ← Debug macros and log level definitions
    debug.c          ← Logging implementation with level filtering
    memory_tracker.h ← Memory allocation tracking for leak detection
    memory_tracker.c ← Implementation of allocation tracking
  test/
    integration/
      terminal_tests.c ← End-to-end tests with real terminal interaction
      pty_tests.c       ← PTY-specific integration tests
      capture_utils.c  ← Terminal capture and replay utilities
    unit/
      parser_tests.c  ← Escape sequence parser unit tests
      layout_tests.c  ← Window layout algorithm tests
  tools/
    log_analyzer.py  ← Script for parsing and analyzing debug logs
    memory_report.sh ← Valgrind wrapper for standardized memory analysis
    terminal_replay.c ← Tool for replaying captured terminal sessions
```

Core Debugging Infrastructure Code

```
// debug.h - Comprehensive debugging infrastructure C

#ifndef TMUX_DEBUG_H

#define TMUX_DEBUG_H


#include <stdio.h>
#include <stdarg.h>
#include <time.h>
#include <sys/time.h>

typedef enum {

    LOG_ERROR = 0,
    LOG_WARN = 1,
    LOG_INFO = 2,
    LOG_DEBUG = 3,
    LOG_TRACE = 4

} log_level_t;

extern log_level_t current_log_level;

extern FILE* log_output;

// Initialize logging system with output file and level

int debug_init(const char* log_file, log_level_t level);

void debug_cleanup(void);

// Core logging function with timestamp and context

void debug_log(log_level_t level, const char* component, const char* fmt, ...);

// Convenience macros for different log levels

#define LOG_ERROR(component, ...) debug_log(LOG_ERROR, component, __VA_ARGS__)
#define LOG_WARN(component, ...) debug_log(LOG_WARN, component, __VA_ARGS__)
```

```
#define LOG_INFO(component, ...) debug_log(LOG_INFO, component, __VA_ARGS__)

#define LOG_DEBUG(component, ...) debug_log(LOG_DEBUG, component, __VA_ARGS__)

#define LOG_TRACE(component, ...) debug_log(LOG_TRACE, component, __VA_ARGS__)

// Binary data logging for PTY streams and escape sequences

void debug_log_hex(log_level_t level, const char* component,
                   const char* label, const void* data, size_t len);

// Function entry/exit tracing

#define TRACE_ENTER(func) LOG_TRACE("TRACE", "ENTER: %s", func)

#define TRACE_EXIT(func) LOG_TRACE("TRACE", "EXIT: %s", func)

// Assertion macros with logging

#define DEBUG_ASSERT(condition, message) \
do { \
    if (!(condition)) { \
        LOG_ERROR("ASSERT", "Assertion failed: %s at %s:%d", \
                  message, __FILE__, __LINE__); \
        abort(); \
    } \
} while(0)

// Memory tracking for leak detection

void* debug_malloc(size_t size, const char* file, int line);

void debug_free(void* ptr, const char* file, int line);

void debug_memory_report(void);

#ifndef DEBUG_MEMORY

#define malloc(size) debug_malloc(size, __FILE__, __LINE__)

#define free(ptr) debug_free(ptr, __FILE__, __LINE__)

#endif
```

```
#endif

#endif // TMUX_DEBUG_H
```

PTY Debugging Utilities

```
// pty_debug.h - PTY-specific debugging functions C

#ifndef PTY_DEBUG_H

#define PTY_DEBUG_H

#include "pty_manager.h"

// Dump complete PTY session state for debugging

void pty_debug_dump_session(pty_session_t* session);

// Monitor PTY I/O with detailed logging

int pty_debug_monitor_io(pty_session_t* session, int timeout_ms);

// Validate PTY file descriptor state

bool pty_debug_validate_fds(pty_session_t* session);

// Check process tree and session leadership

void pty_debug_process_tree(pid_t root_pid);

#endif // PTY_DEBUG_H

// Implementation provides comprehensive PTY state inspection

void pty_debug_dump_session(pty_session_t* session) {

    LOG_DEBUG("PTY_DEBUG", "==== PTY Session Dump ===");

    LOG_DEBUG("PTY_DEBUG", "Master FD: %d", session->master_fd);

    LOG_DEBUG("PTY_DEBUG", "Slave FD: %d", session->slave_fd);

    LOG_DEBUG("PTY_DEBUG", "Child PID: %d", session->child_pid);

    LOG_DEBUG("PTY_DEBUG", "Process Running: %s",

        session->process_running ? "true" : "false");

    LOG_DEBUG("PTY_DEBUG", "Exit Status: %d", session->exit_status);
}
```

```
// Check file descriptor validity

if (fcntl(session->master_fd, F_GETFD) == -1) {
    LOG_ERROR("PTY_DEBUG", "Master FD %d is invalid: %s",
              session->master_fd, strerror(errno));
}

// Check PTY device names

char* slave_name = ptsname(session->master_fd);

if (slave_name) {
    LOG_DEBUG("PTY_DEBUG", "Slave Device: %s", slave_name);
} else {
    LOG_ERROR("PTY_DEBUG", "Cannot get slave device name: %s",
              strerror(errno));
}

// TODO: Add process group and session ID checks
// TODO: Verify terminal size settings
// TODO: Check terminal attribute consistency
}
```

Terminal Emulation Debugging Tools

```
// terminal_debug.h - Terminal emulation debugging utilities C

#ifndef TERMINAL_DEBUG_H

#define TERMINAL_DEBUG_H

#include "terminal_emulator.h"

// Dump complete screen buffer state

void terminal_debug_dump_screen(screen_buffer_t* buffer);

// Log escape sequence parser state transitions

void terminal_debug_parser_state(escape_parser_t* parser, unsigned char byte);

// Validate screen buffer consistency

bool terminal_debug_validate_buffer(screen_buffer_t* buffer);

// Generate test escape sequences for parser testing

void terminal_debug_test_sequences(screen_buffer_t* buffer);

#endif // TERMINAL_DEBUG_H

// Screen buffer validation with comprehensive checks

bool terminal_debug_validate_buffer(screen_buffer_t* buffer) {

    bool valid = true;

    // Check cursor bounds

    if (buffer->cursor_x < 0 || buffer->cursor_x >= buffer->width) {

        LOG_ERROR("TERM_DEBUG", "Cursor X out of bounds: %d (width: %d)",

                  buffer->cursor_x, buffer->width);

        valid = false;
    }
}
```

```

if (buffer->cursor_y < 0 || buffer->cursor_y >= buffer->height) {

    LOG_ERROR("TERM_DEBUG", "Cursor Y out of bounds: %d (height: %d)",
              buffer->cursor_y, buffer->height);

    valid = false;

}

// Validate scrollback buffer

if (buffer->scrollback_lines > MAX_SCROLLBACK) {

    LOG_ERROR("TERM_DEBUG", "Scrollback overflow: %d > %d",
              buffer->scrollback_lines, MAX_SCROLLBACK);

    valid = false;

}

// TODO: Check character cell validity

// TODO: Verify attribute consistency

// TODO: Validate UTF-8 character integrity


return valid;
}

```

Milestone Checkpoints for Debugging

Milestone 1 PTY Debugging Checkpoint:

- Run: `./tmux` and create a single pane
- Expected: Shell prompt appears, typing works, Ctrl-C sends interrupt
- Debug check: `ps -ef | grep tmux` shows correct process tree
- Failure signs: No prompt, typing echoes but no command execution, Ctrl-C kills multiplexer

Milestone 2 Terminal Emulation Debugging Checkpoint:

- Run: `./tmux` and execute `ls --color=always`

- Expected: Colored output appears correctly formatted
- Debug check: Enable parser logging and verify escape sequences are processed
- Failure signs: Escape sequences visible as text, incorrect colors, cursor positioning wrong

Milestone 3 Window Management Debugging Checkpoint:

- Run: `./tmux`, split pane vertically and horizontally
- Expected: Multiple panes with borders, independent scrolling
- Debug check: Verify layout tree structure matches visual layout
- Failure signs: Panes overlap, borders missing, content appears in wrong pane

Milestone 4 Input Handling Debugging Checkpoint:

- Run: `./tmux`, press prefix key (Ctrl-B) followed by split command
- Expected: Command mode activates, pane splits on command
- Debug check: Monitor input parsing and command dispatch
- Failure signs: Prefix key not recognized, commands sent to shell instead of multiplexer

Future Extensions

Milestone(s): Beyond Milestone 4 - this section describes potential enhancements that could be added after completing the core terminal multiplexer functionality

Mental Model: The Growing Ecosystem

Think of future extensions like expanding a small mountain cabin into a full resort. The original cabin (our basic terminal multiplexer) provides shelter and basic amenities - you can split rooms, manage multiple guests, and coordinate activities. But as needs grow, you might add a lodge office where guests can check in and out even when you're away (session persistence), a concierge service that remembers guest preferences and provides custom amenities (advanced configuration), and recreational facilities that make the stay more enjoyable (copy mode, search, customizable displays).

Each extension builds on the solid foundation already established, but adds new capabilities that transform the simple shelter into a sophisticated environment. The key architectural principle remains the same - maintain the core functionality while adding complementary systems that enhance the user experience without disrupting the fundamental operations.

Session Persistence

Session persistence represents one of the most valuable extensions to a terminal multiplexer, transforming it from a simple window manager into a persistent workspace that survives terminal disconnections, system reboots, and network failures. This capability allows users to detach from their multiplexer sessions and reattach later, finding their shell processes, command history, and screen state exactly as they left them.

The mental model for session persistence resembles a library reading room system. When you leave the library, the librarian carefully marks your place in each book, notes which books you were reading, remembers your preferred seating arrangement, and stores everything in a secure location. When you return hours or days later, the librarian reconstructs your exact workspace - same books open to the same pages, same arrangement on your desk, same reading lamp position. Session persistence provides this same continuity for terminal sessions.

Session State Serialization

Implementing session persistence requires serializing the complete state of a multiplexer session to persistent storage and later reconstructing that state when reattaching. The serialization must capture not just the visible terminal state, but the entire operational context that enables seamless continuation.

Component	State to Serialize	Format Considerations
PTY Sessions	Process IDs, working directories, environment variables, terminal size	JSON with process metadata
Screen Buffers	Character cells, attributes, cursor positions, scrollback history	Binary format with run-length encoding
Layout Tree	Pane hierarchy, split ratios, focus state, minimum dimensions	Nested JSON structure
Terminal State	Current attributes, saved cursor positions, parser state	Structured binary format
Input Handler	Key binding overrides, prefix key state, command history	JSON configuration format

The serialization process involves capturing a consistent snapshot of all components while they continue operating. This requires careful coordination to avoid capturing inconsistent intermediate states where some components have processed events that others have not yet seen.

Decision: Session State Storage Format

- **Context:** Need to balance serialization speed, storage efficiency, and format evolution compatibility
- **Options Considered:**
 1. Pure JSON for human readability and debugging ease
 2. Binary format for space efficiency and fast serialization
 3. Hybrid approach with JSON metadata and binary payload data
- **Decision:** Hybrid approach with JSON for structure and binary for screen buffer content
- **Rationale:** JSON provides format versioning and debugging capability while binary encoding efficiently stores large screen buffers and scrollback history
- **Consequences:** Enables fast serialization of large sessions while maintaining format evolution path and debugging accessibility

Process State Management

The most complex aspect of session persistence involves managing the lifecycle of child processes across detach and reattach operations. Unlike simple state serialization, processes represent active system resources that must continue executing even when no terminal is attached to observe their output.

When a session detaches, the multiplexer must transition all PTY sessions into a **background execution mode** where shell processes continue running but their output gets buffered rather than displayed. The PTY master file descriptors remain open and readable, but no terminal is connected to consume the output or provide input.

Process Management Task	Implementation Approach	Error Handling
Output Buffering	Expand screen buffer to accommodate continued output during detachment	Implement buffer size limits with oldest line eviction
Input Queuing	Store user input in persistent queue until processes can consume it	Handle queue overflow by dropping oldest input events
Signal Forwarding	Maintain process group relationships for proper signal delivery	Detect and handle orphaned process groups
Working Directory	Track and restore current working directory for each shell session	Handle cases where directories were deleted during detachment
Environment Variables	Preserve environment modifications made during session execution	Detect conflicts with system-wide environment changes

The challenge lies in maintaining the illusion that the terminal never disconnected. When reattaching, processes should behave exactly as if the terminal had been continuously available. This requires buffering all output generated during detachment and presenting it seamlessly when the user reconnects.

Session Discovery and Management

Session persistence introduces the need for a **session registry** that tracks active detached sessions and enables users to discover and reattach to them. This registry must survive multiplexer restarts and provide meaningful session identification.

Session Registry Schema:

- Session ID: Unique identifier for session instance
- Session Name: User-friendly name for easy identification
- Creation Time: When session was originally created
- Last Attach Time: When user was last connected to session
- Process Count: Number of active shell processes in session
- Working Directory: Base directory where session was created
- Layout Summary: Brief description of pane configuration
- State File Path: Location of serialized session state

The registry enables commands like "list all detached sessions", "attach to session by name", "kill abandoned session", and "rename existing session". Each registry entry contains enough metadata for users to identify the

session they want without loading the full session state.

Session naming conventions become important for usability. The system should support both automatic naming (based on working directory or primary command) and user-specified names. Name collision resolution and session lifecycle management (automatic cleanup of old sessions) help maintain a manageable session namespace.

Reattachment and State Reconstruction

The reattachment process represents the inverse of session serialization - reconstructing a complete operational multiplexer from persistent storage. This process must restore not just the visible state, but the entire execution context that enables continued operation.

The reconstruction follows a carefully ordered sequence:

1. **Validate session state integrity** by checking file format versions, verifying process existence, and confirming PTY device availability
2. **Reconstruct PTY sessions** by reconnecting to existing child processes or detecting terminated processes that require restart
3. **Restore screen buffers** by deserializing character grids, cursor positions, and scrollback history for each pane
4. **Rebuild layout tree** by recreating the split hierarchy and calculating current pane dimensions for the new terminal size
5. **Initialize input handler** by restoring key bindings, command history, and prefix key state
6. **Synchronize terminal state** by applying any buffered output generated during detachment and updating display

The most delicate aspect involves handling process state changes that occurred during detachment. Child processes may have terminated, changed working directories, modified environment variables, or generated substantial output. The reattachment logic must detect these changes and adapt appropriately.

Process State Change	Detection Method	Recovery Action
Process Termination	Check process existence via kill(pid, 0)	Display termination status and offer restart option
Working Directory Change	Compare stored vs current pwd from process	Update session working directory tracking
Environment Modification	Not directly detectable	Accept changes as part of normal session evolution
Substantial Output	Check PTY buffer size and modification time	Present buffered output with clear delineation markers
Resource Exhaustion	Monitor PTY buffer and system resource usage	Implement graceful degradation with user notification

Detachment Signal Handling

Clean detachment requires the multiplexer to respond appropriately to various disconnection scenarios - user-initiated detachment, terminal closure, network disconnection, or system shutdown. Each scenario demands different preservation strategies.

User-initiated detachment (triggered by a key binding like prefix-d) represents the clean case where the multiplexer can perform orderly serialization with advance notice. Terminal closure or network disconnection requires the multiplexer to detect the condition through failed I/O operations and initiate emergency serialization. System shutdown presents the most challenging case, requiring rapid state preservation within the constraints of shutdown timeouts.

The multiplexer should register signal handlers for `SIGHUP` (terminal hangup), `SIGTERM` (termination request), and `SIGINT` (interrupt) to initiate appropriate detachment procedures. Emergency serialization prioritizes preserving critical state (process IDs, current commands) over complete state (full scrollback history) when time constraints prevent full serialization.

Advanced Terminal Features

Beyond session persistence, numerous advanced features can enhance the terminal multiplexer's usability and power-user appeal. These features transform the basic split-screen interface into a sophisticated terminal workspace with integrated text manipulation, search capabilities, and customizable information displays.

Copy Mode and Text Selection

Copy mode enables keyboard-driven text selection and clipboard integration within the terminal multiplexer, allowing users to select and copy text from any pane's screen buffer or scrollback history without requiring mouse interaction. This feature particularly benefits users working in mouse-free environments or over high-latency connections where mouse interaction proves cumbersome.

The mental model for copy mode resembles switching a text editor into selection mode. The normal terminal interface transitions into a navigation interface where cursor movement keys traverse the screen buffer, selection keys mark text regions, and copy commands transfer selected content to the system clipboard or internal paste buffer.

Copy Mode Action	Key Binding	Implementation Requirements
Enter Copy Mode	prefix + [Freeze current pane display and overlay cursor
Navigation	Arrow keys, hjkl	Move cursor through screen buffer and scrollback
Start Selection	Space or v	Mark selection start position
End Selection	Enter or y	Mark selection end and copy to clipboard
Line Selection	v	Select entire lines for rapid bulk copying
Search in Buffer	/ or ?	Find text patterns within current pane
Exit Copy Mode	Escape or q	Return to normal terminal interaction mode

The implementation requires maintaining a **copy cursor** that moves independently of the terminal cursor, overlaying the display to show current position. The copy cursor can navigate beyond the visible screen into scrollback history, enabling access to commands and output from earlier in the session.

Text selection involves tracking start and end positions within the screen buffer coordinate system. The selection must handle line wrapping, character encoding, and mixed content (text with escape sequences). When copying, the system extracts plain text content while preserving line breaks and word boundaries.

Search and Pattern Matching

Integrated search functionality enables users to locate specific text patterns within pane contents, scrollback history, and across multiple panes simultaneously. This capability proves essential for reviewing long command outputs, finding specific log entries, or locating previous commands in shell history.

The search implementation operates on the character grid maintained by each pane's screen buffer, supporting both literal text matching and regular expression patterns. Search results highlight matching text and provide navigation between multiple matches within the same buffer.

Search Feature	Implementation Approach	User Interface
Incremental Search	Real-time pattern matching as user types query	Highlight matches and show match count
Cross-Pane Search	Search all visible panes and present unified results	Switch between panes showing matches
Historical Search	Search scrollback buffers for older content	Navigate through chronological matches
Pattern Highlighting	Persistent highlighting of search terms	Toggle highlighting on/off per pane
Search History	Remember recent search patterns for reuse	Arrow key navigation through search history

The search engine must handle **UTF-8 text encoding** correctly, supporting multi-byte character sequences and wide characters that occupy multiple terminal cells. Pattern matching should operate on the logical character sequence rather than the raw byte stream or terminal cell array.

Regular expression support enables powerful pattern matching for structured text like log files, configuration files, or command output. The implementation should provide a reasonable subset of regex features without requiring a full regex engine - basic character classes, quantifiers, and anchoring operators cover most use cases.

Customizable Status Bar

The **status bar** provides persistent information display across the bottom of the terminal window, showing session status, pane information, system metrics, and user-defined content. A customizable status bar transforms from a simple information display into a powerful dashboard tailored to individual workflows.

The status bar architecture supports modular **status segments** that can be enabled, disabled, reordered, and configured independently. Each segment responsible for displaying specific information and updating at appropriate intervals.

Status Segment Type	Information Displayed	Update Frequency
Session Info	Session name, attach time, detached indicator	On session state changes
Pane Status	Current pane index, total pane count, pane title	On focus changes and pane operations
Process Status	Shell command, process ID, exit status	On process state changes
System Metrics	CPU usage, memory usage, load average	Every 5-10 seconds
Time Display	Current time, date, timezone	Every minute
Custom Commands	Output from user-defined scripts	User-configurable intervals

The status bar rendering system must efficiently update only changed segments to minimize terminal output and reduce visual flicker. This requires tracking the previous rendered state of each segment and generating differential updates when content changes.

Status bar configuration should support both built-in segment types and custom segments that execute user-defined commands or scripts. Custom segments enable integration with external systems - displaying git branch information, monitoring system services, or showing application-specific status.

Advanced Key Binding System

While the basic multiplexer provides essential key bindings for pane management, an advanced key binding system enables complete customization of the command interface, support for complex key sequences, and integration with external tools.

The advanced system supports **multi-key sequences** beyond the simple prefix-key model, enabling Emacs-style or Vim-style key combinations that can express complex commands efficiently. Users can define key sequences like `prefix C-w v` (prefix, then Ctrl-W, then v) for sophisticated command hierarchies.

Key Binding Feature	Implementation Requirements	Configuration Format
Multi-Key Sequences	State machine tracking partial key sequences	Hierarchical key mapping tree
Conditional Bindings	Different bindings based on pane state or content	Conditional expression evaluation
Command Composition	Combining multiple commands in single binding	Command scripting mini-language
Mode-Specific Bindings	Different key meanings in copy mode vs normal mode	Mode-aware binding resolution
Dynamic Bindings	Runtime creation of temporary key bindings	Programmatic binding registration

The key binding system should provide **binding introspection** capabilities, enabling users to discover available commands, view current bindings, and detect binding conflicts. A built-in help system can display context-sensitive key binding information based on current mode and state.

Command composition enables sophisticated automation by combining basic operations into complex workflows. For example, a single key binding might create a new pane, start a specific application, resize the pane to preferred dimensions, and switch focus - all executed as an atomic operation.

Window and Session Management

Advanced window management extends beyond basic pane splitting to provide sophisticated workspace organization capabilities. **Named windows** enable logical grouping of related panes, while **session templates** provide reproducible workspace setups for common development scenarios.

Named windows function like tabs in a web browser, where each window contains an independent set of panes with their own layout configuration. Users can switch between windows, rename windows, and create new windows with predefined layouts.

Window Management Feature	User Interface	Implementation Considerations
Window Creation	prefix + c (create), prefix + n (next)	Independent layout trees per window
Window Navigation	prefix + 0-9 (by number), prefix + ' (by name)	Window registry with fast lookup
Window Renaming	prefix + , (rename current window)	Persistent naming across session save/restore
Window Status	Show window list in status bar	Efficient window state monitoring
Window Templates	Create windows from predefined configurations	Template storage and parameterization

Session templates provide reproducible workspace creation by defining standard pane layouts, startup commands, and working directories for common development scenarios. A web development template might create panes for editor, development server, database console, and system monitoring - all initialized with appropriate commands and working directories.

Template parameterization enables flexible reuse by accepting variables like project directory, server ports, or database connections. Users can instantiate templates with different parameters to create similar but customized workspaces.

Integration Capabilities

Advanced terminal multiplexers benefit from integration with external tools and systems, enabling seamless workflows between terminal-based and graphical applications. Integration capabilities include clipboard synchronization, external editor support, and notification systems.

Clipboard integration synchronizes the multiplexer's internal copy buffer with the system clipboard, enabling seamless text exchange with other applications. This requires platform-specific implementations for different operating systems and clipboard managers.

Integration Type	Implementation Approach	Platform Considerations
System Clipboard	Use xclip/pbcopy/clip.exe for clipboard operations	Different utilities per OS
External Editors	Launch editor with temporary files for complex editing	Handle editor configuration and file cleanup
Desktop Notifications	Send notifications for background pane activity	Integration with system notification services
File Manager	Open file paths from terminal in external file manager	Parse file paths from terminal output
Web Browser	Launch URLs detected in terminal output	URL pattern recognition and browser selection

External editor integration enables users to edit complex text within the terminal multiplexer using their preferred graphical editor. This involves creating temporary files, launching the editor, and monitoring for file changes to update the terminal content.

Notification integration helps users monitor background activity in detached or non-focused panes. The system can detect interesting events like command completion, error messages, or specific text patterns and generate desktop notifications to alert users.

Common Future Extension Pitfalls

⚠ Pitfall: Session State Explosion Many implementations attempt to serialize every possible piece of state, creating enormous session files and slow serialization. The key insight is that not all state needs persistence - transient UI state, temporary buffers, and cached calculations can be regenerated on reattachment. Focus on preserving only the essential state that cannot be reconstructed: process IDs, user input, and user-generated content.

⚠ Pitfall: Process Lifecycle Mismanagement A common mistake involves assuming that child processes remain unchanged during detachment. In reality, processes may terminate, change working directories, create new child processes, or modify environment variables. The reattachment logic must detect and adapt to these changes rather than assuming static process state.

⚠ Pitfall: Copy Mode Performance Issues Implementing copy mode by re-rendering the entire screen buffer for each cursor movement creates unacceptable performance for large scrollback buffers. Instead, use differential rendering that updates only the cursor position and selection highlighting. Pre-index the screen buffer content to enable efficient text search and navigation.

⚠ Pitfall: Status Bar Update Frequency Updating the status bar on every terminal refresh or input event creates significant overhead and visual distraction. Implement intelligent update scheduling that refreshes segments only when their underlying data changes, and batch multiple changes to reduce rendering frequency.

⚠ Pitfall: Key Binding Conflicts Adding advanced key binding features without considering conflicts with existing terminal applications creates user frustration. Maintain a registry of reserved key sequences, provide conflict detection when users define custom bindings, and offer escape mechanisms for sending literal key sequences to applications.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Session Storage	JSON files in <code>~/.tmux/sessions/</code>	SQLite database with indexed metadata
Process Management	Simple PID tracking with <code>kill()</code> checks	Process control groups with resource monitoring
Search Engine	Linear buffer scanning with basic patterns	Regex engine with incremental matching
Configuration	Static compile-time configuration	Dynamic configuration with hot reloading
Clipboard Integration	Shell commands (<code>xclip/pbcopy</code>)	Native OS APIs with format detection

Recommended File Structure

```
project-root/
├── cmd/tmux/
│   ├── main.c                         ← main entry point
│   └── commands/
│       ├── detach.c                   ← session detachment
│       ├── attach.c                  ← session reattachment
│       └── list_sessions.c          ← session discovery
├── internal/
│   ├── session/
│   │   ├── persistence.c            ← session serialization
│   │   ├── persistence.h
│   │   ├── registry.c               ← session management
│   │   └── registry.h
│   ├── features/
│   │   ├── copy_mode.c              ← text selection
│   │   ├── copy_mode.h
│   │   ├── search.c                 ← buffer search
│   │   ├── search.h
│   │   ├── status_bar.c            ← customizable status
│   │   └── status_bar.h
│   └── integration/
│       ├── clipboard.c              ← system clipboard
│       ├── clipboard.h
│       ├── notifications.c          ← desktop notifications
│       └── notifications.h
├── config/
│   ├── templates/                    ← session templates
│   │   ├── dev_workspace.json
│   │   └── monitoring.json
│   └── bindings/                    ← key binding configs
│       └── default.conf
└── tests/
    ├── session_persistence_test.c
    ├── copy_mode_test.c
    └── integration_test.c
```

Session Persistence Infrastructure

```
// Session registry management - complete implementation

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <time.h>

#include <unistd.h>

typedef struct {

    char session_id[64];

    char session_name[128];

    time_t creation_time;

    time_t last_attach_time;

    int process_count;

    char working_directory[256];

    char state_file_path[512];

    bool is_attached;

} session_registry_entry_t;

typedef struct {

    session_registry_entry_t* entries;

    int entry_count;

    int capacity;

    char registry_path[512];

} session_registry_t;

// Initialize session registry with persistent storage

int session_registry_init(session_registry_t* registry, const char* config_dir) {
```

C

```
snprintf(registry->registry_path, sizeof(registry->registry_path),
         "%s/session_registry.json", config_dir);

registry->entries = malloc(sizeof(session_registry_entry_t) * 16);
registry->capacity = 16;
registry->entry_count = 0;

// Create config directory if it doesn't exist
struct stat st = {0};

if (stat(config_dir, &st) == -1) {
    mkdir(config_dir, 0700);
}

// Load existing registry
return session_registry_load(registry);
}

// Save registry to persistent storage
int session_registry_save(session_registry_t* registry) {
    FILE* file = fopen(registry->registry_path, "w");
    if (!file) return -1;

    fprintf(file, "{\n    \"sessions\": [\n");
    for (int i = 0; i < registry->entry_count; i++) {
        session_registry_entry_t* entry = &registry->entries[i];
        fprintf(file, "        {\n");
        fprintf(file, "            \"session_id\": \"%s\",\\n", entry->session_id);
        fprintf(file, "            \"session_name\": \"%s\",\\n", entry->session_name);
    }
}
```

```
    fprintf(file, "\"creation_time\": %ld,\n", entry->creation_time);

    fprintf(file, "\"last_attach_time\": %ld,\n", entry->last_attach_time);

    fprintf(file, "\"process_count\": %d,\n", entry->process_count);

    fprintf(file, "\"working_directory\": \"%s\",\\n", entry->working_directory);

    fprintf(file, "\"state_file_path\": \"%s\",\\n", entry->state_file_path);

    fprintf(file, "\"is_attached\": %s\\n", entry->is_attached ? "true" : "false");

    fprintf(file, "%s\\n", (i < registry->entry_count - 1) ? "," : "");

}

fprintf(file, "]\\n}\\n");

fclose(file);

return 0;

}
```

Copy Mode Core Logic Skeleton

```
// Copy mode implementation - skeleton for core functionality C

typedef struct {

    int cursor_x, cursor_y;          // Current copy cursor position

    int selection_start_x, selection_start_y; // Selection start point

    int selection_end_x, selection_end_y;      // Selection end point

    bool selection_active;           // Whether selection is in progress

    bool line_selection_mode;        // Line selection vs character selection

    char* search_pattern;          // Current search pattern

    int* search_matches;           // Array of match positions

    int search_match_count;         // Number of search matches

    int current_match_index;        // Currently highlighted match

} copy_mode_state_t;

// Enter copy mode for specified pane

int copy_mode_enter(pane_t* pane, copy_mode_state_t* state) {

    // TODO 1: Initialize copy cursor at current terminal cursor position

    // TODO 2: Save current terminal cursor position for restoration

    // TODO 3: Set terminal to copy mode (disable cursor blinking, etc.)

    // TODO 4: Initialize selection state (no active selection)

    // TODO 5: Render copy cursor overlay on screen buffer

    // Hint: copy cursor should be visually distinct from terminal cursor

}

// Process key input in copy mode

int copy_mode_handle_key(pane_t* pane, copy_mode_state_t* state, input_event_t* event) {

    // TODO 1: Handle navigation keys (arrows, hjkl) to move copy cursor

    // TODO 2: Handle selection keys (space/v) to start/end selection

    // TODO 3: Handle copy keys (enter/y) to copy selected text to clipboard
```

```
// TODO 4: Handle search keys (/) to initiate buffer search

// TODO 5: Handle exit keys (escape/q) to leave copy mode

// TODO 6: Update screen overlay to show current cursor and selection

// Hint: Bounds checking is critical - cursor must stay within buffer

}

// Copy selected text to clipboard

int copy_mode_copy_selection(pane_t* pane, copy_mode_state_t* state) {

    // TODO 1: Validate selection bounds (start before end, within buffer)

    // TODO 2: Extract text from screen buffer between selection points

    // TODO 3: Handle line wrapping and preserve line breaks

    // TODO 4: Convert terminal cells to plain text (strip attributes)

    // TODO 5: Send text to system clipboard using platform-specific method

    // Hint: Remember to handle UTF-8 encoding correctly

}
```

Advanced Status Bar Infrastructure

```
// Status bar segment system - complete modular implementation

C

typedef enum {

    SEGMENT_SESSION_INFO,
    SEGMENT_PANE_STATUS,
    SEGMENT_PROCESS_STATUS,
    SEGMENT_SYSTEM_METRICS,
    SEGMENT_TIME_DISPLAY,
    SEGMENT_CUSTOM_COMMAND
} status_segment_type_t;

typedef struct status_segment {

    status_segment_type_t type;

    char content[256];           // Rendered segment content

    int width;                  // Segment width in characters

    bool needs_update;          // Whether content needs refresh

    time_t last_update;         // When content was last updated

    int update_interval;        // Seconds between updates

    char command[512];          // For custom command segments

    struct status_segment* next; // Linked list of segments
} status_segment_t;

typedef struct {

    status_segment_t* segments; // Head of segment list

    char rendered_bar[1024];   // Complete rendered status bar

    int terminal_width;        // Available width for rendering

    bool needs_full_render;   // Whether entire bar needs refresh
} status_bar_t;
```

```

// Initialize status bar with default segments

int status_bar_init(status_bar_t* bar, int terminal_width) {

    bar->segments = NULL;

    bar->terminal_width = terminal_width;

    bar->needs_full_render = true;

    // Add default segments

    status_bar_add_segment(bar, SEGMENT_SESSION_INFO, NULL, 10);

    status_bar_add_segment(bar, SEGMENT_PANE_STATUS, NULL, 5);

    status_bar_add_segment(bar, SEGMENT_TIME_DISPLAY, NULL, 60);

    return 0;
}

// Update status bar content and render if needed

int status_bar_update(status_bar_t* bar, window_manager_t* wm) {

    time_t current_time = time(NULL);

    bool any_segment_updated = false;

    // Check each segment for update requirements

    for (status_segment_t* seg = bar->segments; seg; seg = seg->next) {

        if (current_time - seg->last_update >= seg->update_interval || seg->needs_update) {

            status_segment_update_content(seg, wm, current_time);

            seg->last_update = current_time;

            seg->needs_update = false;

            any_segment_updated = true;
        }
    }
}

```

```

// Re-render bar if any segment changed

if (any_segment_updated || bar->needs_full_render) {

    return status_bar_render(bar);

}

return 0;
}

```

Milestone Checkpoints

Session Persistence Verification:

1. Start multiplexer with `./tmux new -s test_session`
2. Create several panes and run commands in each
3. Detach with prefix-d, verify session appears in `./tmux list-sessions`
4. Reattach with `./tmux attach -s test_session`
5. Verify all panes restored with correct content and running processes

Expected behavior: All panes should show exactly the same content as before detachment, commands should still be running, and scrollback history should be preserved.

Copy Mode Verification:

1. Fill a pane with text content (run `ls -la` or similar)
2. Enter copy mode with prefix-[
3. Navigate with arrow keys, verify cursor moves independently
4. Start selection with space, move cursor, end with enter
5. Exit copy mode and paste with prefix-]

Expected behavior: Selected text should copy to internal buffer and paste correctly, cursor should move through all visible content and scrollback.

Status Bar Verification:

1. Configure status bar with time and pane count segments
2. Create and destroy panes, verify count updates immediately
3. Wait for time update, verify timestamp changes
4. Resize terminal, verify status bar adapts to new width

Expected behavior: Status segments should update at appropriate intervals, content should fit within terminal width, rendering should be flicker-free.

Glossary

Milestone(s): All milestones (1-4) - this section provides comprehensive definitions for technical terminology used throughout the terminal multiplexer design and implementation

Mental Model: The Technical Dictionary

Think of this glossary as a technical translator that bridges multiple domains of knowledge. Building a terminal multiplexer requires understanding concepts from operating systems (processes, signals, file descriptors), terminal emulation (escape sequences, character encoding), user interface design (layout algorithms, input handling), and systems programming (I/O multiplexing, state machines). Each term in this glossary represents a building block that connects these domains, helping you understand not just what each concept means, but how it fits into the larger system architecture.

Just as a good dictionary provides not only definitions but also etymology and usage examples, this glossary explains both the technical meaning of each term and its role in the terminal multiplexer ecosystem. The terms are organized to build conceptually from fundamental system concepts through specific implementation details.

Core System Concepts

The foundation of terminal multiplexing rests on several key operating system and terminal concepts that may be unfamiliar to developers who haven't worked extensively with Unix systems programming.

Term	Definition	Context in Terminal Multiplexer
terminal multiplexer	A system that manages multiple shell sessions through a single terminal interface, allowing users to split screens, switch between sessions, and maintain persistent connections	The overarching system being built - coordinates PTY management, terminal emulation, and user interface
pseudo-terminal (PTY)	A software device that emulates real terminal hardware, consisting of a master/slave pair where the master connects to an application and the slave connects to a shell process	Core abstraction enabling multiple shell sessions - each pane gets its own PTY pair
controlling terminal	The terminal device associated with a process session that receives keyboard signals (Ctrl-C, Ctrl-Z) and can send signals to the session leader and process group	Critical for proper signal handling - child shells must have their PTY slave as controlling terminal
session leader	The process that created a new session using <code>setsid()</code> and controls terminal access for all processes in that session	Each shell process becomes session leader for its PTY to ensure proper signal delivery
process group management	System for organizing related processes so that signals can be delivered to entire groups rather than individual processes	Ensures that signals like SIGTERM reach not just the shell but all child processes it spawns
session persistence	The ability to detach from a running terminal multiplexer session and reattach later, with all shell sessions continuing to run in the background	Future extension that would require serializing window manager state and PTY session information
raw terminal mode	Terminal configuration that bypasses line-buffering and signal processing, passing all input directly to the application for processing	Essential for capturing prefix keys and special key combinations for multiplexer commands
canonical terminal mode	Default terminal configuration where the kernel handles line editing, signal generation, and input buffering before passing complete lines to applications	Must be disabled for multiplexer operation but restored on exit for normal terminal behavior

PTY Management and Process Control

The PTY management layer deals with the low-level details of creating and managing pseudo-terminal pairs and the shell processes that connect to them.

Term	Definition	Context in Terminal Multiplexer
PTY allocation	Process of requesting a new pseudo-terminal pair from the kernel using <code>posix_openpty()</code> or similar system calls	First step in creating new pane - must handle allocation failures gracefully
master file descriptor	The file descriptor connected to the master side of a PTY pair, used by the terminal multiplexer to read shell output and write user input	Core I/O channel - added to select() monitoring for reading shell output to display
slave file descriptor	The file descriptor connected to the slave side of a PTY pair, used by shell processes as their stdin, stdout, and stderr	Connected to child shell process - must be properly configured as controlling terminal
child process termination	Event when a shell process exits, detected through SIGCHLD signals and handled by updating pane state and potentially closing the pane	Must be detected promptly to update UI and clean up resources - handled in event loop
terminal size propagation	Process of forwarding terminal window size changes to PTY sessions using the TIOCSWINSZ ioctl so shells know their display dimensions	Critical for proper shell behavior when terminal is resized - affects command line editing and program output
signal forwarding	Passing signals like SIGTERM or SIGKILL from the multiplexer to the appropriate shell processes and their children	Ensures that killing multiplexer properly terminates all child processes
file descriptor management	Careful tracking of which file descriptors are open, ensuring proper cleanup on process termination, and avoiding descriptor leaks	Must handle both normal shutdown and error conditions - descriptors left open can exhaust system resources

Terminal Emulation and Character Processing

Terminal emulation involves parsing escape sequences and maintaining virtual representations of terminal screen state.

Term	Definition	Context in Terminal Multiplexer
escape sequence	Special character codes beginning with ESC (0x1B) that control terminal behavior like cursor movement, color changes, and screen clearing	Core parsing challenge - must handle incomplete sequences and terminal-specific variations
ANSI escape codes	Standardized escape sequences defined by ANSI X3.64 standard for terminal control, including CSI (Control Sequence Introducer) sequences	Primary escape sequence format - covers most common terminal control needs
CSI sequence	Control Sequence Introducer sequences that begin with ESC[and include parameters for commands like cursor positioning and text formatting	Most complex parsing case - requires parameter extraction and bounds checking
screen buffer	Virtual representation of terminal display state as a 2D grid of character cells, each containing a character and associated display attributes	Core data structure for each pane - maintains complete terminal state for rendering
scrollback buffer	Storage for text lines that have scrolled off the visible screen area, allowing users to review previous output	Enhances usability - implemented as circular buffer with configurable size limit
terminal cell	Individual position in screen buffer containing one character and its display attributes (color, bold, underline, etc.)	Basic unit of terminal display - must handle wide characters that span multiple cells
cursor position	Current location where the next character will be placed, maintained as row and column coordinates within the screen buffer	Essential terminal state - must be saved/restored by some escape sequences
text attributes	Display properties like bold, underline, reverse video, and foreground/background colors that affect character appearance	Stored per character cell - must be efficiently represented to minimize memory usage
character encoding	System for representing text characters as byte sequences, with UTF-8 being the standard for modern terminals	Must handle multi-byte UTF-8 sequences that may arrive across multiple reads from PTY
UTF-8 decoder	Component that assembles multi-byte Unicode character sequences from individual bytes and handles incomplete sequences	Critical for international character support - must buffer partial sequences between reads
state machine	Parser design that transitions between different states (normal text, escape sequence, parameter parsing) based on input characters	Robust approach to escape sequence parsing - handles malformed input gracefully
wide characters	Unicode characters that require two terminal columns for display, common in East Asian languages	Complicates cursor positioning and line wrapping - must track actual display width vs character count

Window Management and Layout

The window management system organizes multiple panes within the terminal window and handles user interface concerns.

Term	Definition	Context in Terminal Multiplexer
pane	A rectangular screen region containing one shell session with its own PTY and screen buffer	Basic unit of window organization - each pane operates independently with own terminal state
layout tree	Binary tree data structure representing the hierarchy of pane splits, with leaves containing panes and internal nodes representing split directions	Core data structure for window management - enables complex nested layouts
split ratio	Floating-point value (0.0 to 1.0) determining how available space is divided between child panes in a split	Controls pane sizing - can be adjusted during resize operations
horizontal split	Layout operation that divides a pane into top and bottom regions, each containing independent terminal sessions	Creates new pane below existing one - requires updating layout tree structure
vertical split	Layout operation that divides a pane into left and right regions, each containing independent terminal sessions	Creates new pane beside existing one - requires updating layout tree structure
minimum pane dimensions	Smallest usable size constraints for panes to ensure shells can display at least basic content	Prevents splits that would create unusable tiny panes - typically 10 columns by 3 rows
layout calculation	Algorithm that computes absolute pixel or character coordinates for each pane based on terminal size and split ratios	Recursive tree traversal that assigns final dimensions to each pane
border characters	ASCII or Unicode characters used to visually separate panes and indicate focus state	Reduces available space for panes but improves visual organization
focus management	System for tracking which pane is currently active and should receive user input	Only one pane can have focus at a time - affects input routing and visual highlighting
screen composition	Process of combining multiple pane screen buffers with borders and status information into unified terminal output	Complex rendering challenge - must handle overlapping regions and partial updates
frame-based rendering	Approach where the complete terminal screen is regenerated periodically rather than tracking incremental changes	Simpler than differential rendering but potentially less efficient
focus highlighting	Visual indication (usually border color or style) showing which pane is currently active for input	Essential usability feature - must be updated when focus changes

Input Handling and User Interface

The input handling system manages user interaction with the multiplexer, including command mode and key bindings.

Term	Definition	Context in Terminal Multiplexer
prefix key mechanism	System where a special key combination (like Ctrl-B) signals transition from normal input to command mode	Core user interface paradigm - enables multiplexer commands without conflicting with shell input
command mode	Temporary state where subsequent keystrokes are interpreted as multiplexer commands rather than shell input	Activated by prefix key - must have timeout to return to normal mode
key binding	Association between key combination and multiplexer command, allowing customization of the user interface	Enables user customization - stored in lookup table or tree structure
input event	Structured representation of keyboard input including character code, special keys, and modifier states	Abstraction that simplifies input processing - handles both regular characters and special keys
input routing	Process of directing keyboard events to either the active pane's shell or the multiplexer's command processor	Core input handling logic - depends on current mode and prefix key state
terminal restoration	Process of returning terminal to original configuration when multiplexer exits, undoing raw mode changes	Critical for system stability - must work even during abnormal termination
signal handling	Managing Unix signals like SIGWINCH (terminal resize) and SIGCHLD (child termination) in the multiplexer process	Essential for responsive behavior - must coordinate with main event loop
UTF-8 sequence assembly	Buffering multi-byte Unicode characters until complete sequences are available for processing	Handles input characters that arrive as multiple bytes - prevents malformed character processing
escape sequence detection	Recognizing special key combinations (arrow keys, function keys) that generate multi-byte escape sequences	Complex parsing challenge - different terminals may generate different sequences
prefix state machine	State tracking system that manages transitions between normal input mode, prefix detection, and command mode	Ensures reliable command mode operation - must handle timeouts and invalid sequences

System Integration and Event Processing

The event processing layer coordinates I/O from multiple sources and manages the main program loop.

Term	Definition	Context in Terminal Multiplexer
event loop	Main processing loop that monitors multiple input sources using select() and dispatches events to appropriate handlers	Heart of the multiplexer - coordinates PTY I/O, user input, and signal handling
I/O multiplexing	Technique using select() or similar system calls to handle multiple file descriptors without blocking	Essential for responsive operation - allows monitoring many PTYs simultaneously
file descriptor set	Data structure (fd_set) used with select() to specify which file descriptors to monitor for I/O readiness	Core to event loop operation - must be rebuilt each iteration
select() system call	Unix system call that waits for I/O readiness on multiple file descriptors, returning when any become ready	Primary I/O multiplexing mechanism - blocks until input available or timeout
file descriptor mapping	Data structures that associate file descriptors with pane identifiers and other context information	Enables efficient event dispatch - maps from fd returned by select() to pane object
inter-component communication	Structured data flow between PTY manager, terminal emulator, window manager, and input handler components	Coordinates system operation - typically through function calls and shared data structures
state synchronization	Maintaining consistent shared state across multiple components, such as focus information and terminal size	Prevents race conditions and inconsistencies - requires careful coordination
atomic display updates	Ensuring that screen changes appear as a single consistent update rather than partial intermediate states	Prevents screen flicker - typically by building complete frame before output
signal-safe operations	Functions that can be safely called from Unix signal handlers without causing deadlocks or corruption	Important for proper signal handling - limits what can be done in signal handlers

Error Handling and Robustness

Terminal multiplexers must handle many types of failures gracefully while maintaining system stability.

Term	Definition	Context in Terminal Multiplexer
PTY allocation failure	Error condition when the system cannot provide a new pseudo-terminal pair, typically due to resource exhaustion	Must be handled gracefully - display error to user rather than crashing
child process termination	Normal or abnormal exit of shell processes, detected through SIGCHLD signals and waitpid() calls	Expected event - must update pane state and potentially close pane
broken pipe condition	Error when attempting to write to a PTY whose slave end has been closed, typically when shell exits	Common occurrence - must clean up pane and handle EOF gracefully
terminal size synchronization	Coordinating terminal resize events across layout calculation, PTY updates, and display rendering	Complex coordination challenge - all components must be updated consistently
emergency terminal restoration	Signal-safe procedure for restoring terminal settings during abnormal program termination	Critical safety feature - prevents leaving terminal in unusable raw mode
resource cleanup	Proper deallocation of memory, file descriptors, and process resources during normal and abnormal termination	Prevents resource leaks - must work even during error conditions
graceful degradation	Continuing operation with reduced functionality when non-critical components fail	Improves robustness - prefer partial functionality over complete failure

Testing and Development

Specialized terminology for testing and debugging terminal multiplexer implementations.

Term	Definition	Context in Terminal Multiplexer
component unit tests	Tests that verify individual components (escape parser, layout calculator) in isolation using controlled inputs	Essential for reliable development - each component can be tested independently
integration testing	End-to-end testing with real shell processes and terminal output verification	Tests complete system behavior - requires actual PTY allocation and shell execution
milestone verification checkpoints	Specific tests and expected behaviors after completing each development milestone	Ensures progress is correct - prevents building on faulty foundation
terminal output verification	Comparing actual terminal escape sequences generated by multiplexer with expected output patterns	Complex testing challenge - terminal output includes many control sequences
PTY I/O simulation	Testing technique using mock file descriptors or recorded PTY sessions to test terminal emulation	Enables reproducible testing - avoids dependency on actual shell processes

Advanced Features and Extensions

Terminology for features that extend beyond basic terminal multiplexing functionality.

Term	Definition	Context in Terminal Multiplexer
copy mode	Special input mode that allows keyboard-driven text selection and clipboard integration from terminal scrollback	Advanced user interface feature - enables text selection without mouse
status bar	Persistent information display across the bottom of the terminal window showing session info, time, and custom content	Enhanced user interface - requires space management and regular updates
session registry	System for tracking multiple detached terminal sessions and enabling reattachment	Session persistence feature - requires storing session metadata
status segments	Modular components of the status bar that display different types of information and can be customized	Flexible status display - each segment can have different update intervals
clipboard integration	Synchronization between multiplexer's copy mode and the system clipboard for seamless text transfer	System integration feature - requires platform-specific clipboard APIs
session templates	Predefined workspace configurations that can automatically create specific pane layouts and run commands	Productivity enhancement - enables quick setup of common development environments
background execution mode	Operation mode where shell sessions continue running even when multiplexer is detached from terminal	Core session persistence feature - requires daemon-like operation
state serialization	Converting complete multiplexer state to a format that can be stored persistently and restored later	Technical challenge for session persistence - must capture all relevant state
differential rendering	Optimization technique that updates only changed portions of the terminal display rather than rewriting everything	Performance optimization - requires tracking what changed between frames

Implementation and Architecture Terminology

Technical terms specific to the internal architecture and implementation approach of the terminal multiplexer.

Term	Definition	Context in Terminal Multiplexer
data transformation pipeline	Sequence of components that transform raw PTY output through escape sequence parsing to final display updates	Architecture pattern - data flows from PTY through emulator to window manager
canonical state authority	Design principle where each piece of shared state has exactly one component responsible for its authoritative value	Prevents inconsistencies - focus state owned by window manager, cursor state by screen buffer
eventual consistency	Guarantee that all components will eventually reflect the same logical state, even if temporarily inconsistent	Relaxed consistency model - allows some lag in state propagation for performance
layout recalculation cascade	Process where changes to one pane's size trigger dimension updates throughout the layout tree	Performance consideration - must be efficient since terminal resize is common
escape sequence buffering	Technique for handling escape sequences that arrive across multiple reads from PTY file descriptors	Robustness feature - prevents partial sequence processing
focus state transitions	State changes when active pane changes, requiring updates to visual highlighting and input routing	Coordination challenge - multiple components must be notified of focus changes

Common Implementation Pitfalls

Terminology for mistakes that are frequently encountered when implementing terminal multiplexers.

Term	Definition	Context in Terminal Multiplexer
session state explosion	Problem where too much transient state is serialized for session persistence, making save/restore slow and fragile	Design pitfall - must carefully choose what state is essential vs reconstructible
terminal attribute leakage	Failure to properly restore terminal settings, leaving the terminal in an unusable state after multiplexer exit	Dangerous bug - can make terminal completely unusable requiring login reset
file descriptor exhaustion	Running out of available file descriptors due to improper cleanup or too many simultaneous PTY sessions	Resource management issue - must implement proper cleanup and reasonable limits
signal handling race conditions	Bugs caused by signal handlers running concurrently with main program logic, causing data corruption	Concurrency pitfall - signals are asynchronous and can interrupt any operation
UTF-8 sequence corruption	Errors caused by splitting multi-byte UTF-8 characters across buffer boundaries or I/O operations	Character encoding issue - must carefully handle byte vs character boundaries
escape sequence injection	Security or display issue where malicious shell output can manipulate multiplexer behavior	Security consideration - must sanitize or validate escape sequences
layout calculation overflow	Arithmetic errors when computing pane dimensions, especially with very small terminal sizes	Bounds checking issue - must handle edge cases like tiny terminals

The terminology in this glossary forms the vocabulary for understanding terminal multiplexer implementation. Each term represents a specific concept, challenge, or technique that you'll encounter during development. Understanding these terms and their relationships is essential for navigating the complexity of terminal emulation, process management, and user interface design that comprises a complete terminal multiplexer system.

Implementation Guidance

Understanding the terminology is crucial for implementing a terminal multiplexer, but the terms gain real meaning through practical application. This section provides concrete guidance for working with the concepts defined above.

A. Terminology Usage Patterns

Different phases of implementation focus on different subsets of terminology:

Implementation Phase	Key Terminology	Focus Area
PTY Management	pseudo-terminal, session leader, controlling terminal, file descriptor management	System programming concepts
Terminal Emulation	escape sequence, state machine, screen buffer, UTF-8 decoder	Parser and buffer management
Window Management	layout tree, pane, split ratio, screen composition	Spatial organization algorithms
Input Handling	prefix key mechanism, command mode, input routing, raw terminal mode	User interface and event processing
Error Handling	graceful degradation, resource cleanup, signal-safe operations	Robustness and recovery

B. Documentation and Communication Standards

When documenting your terminal multiplexer implementation or discussing it with others, use precise terminology to avoid confusion:

```
// Good: Uses precise terminology from glossary
// Handle child process termination detected via SIGCHLD
// Update pane state and trigger layout recalculation if needed

static void handle_child_termination(window_manager_t* wm, pid_t child_pid) {
    // Implementation details...
}

// Poor: Vague terminology
// Clean up when process dies

static void cleanup_dead_process(window_manager_t* wm, int pid) {
    // Implementation details...
}
```

C. Error Message Terminology

Use consistent terminology in error messages to help with debugging:

```
// Terminal state management errors

"Failed to enter raw terminal mode: %s"

"Terminal restoration failed during cleanup: %s"

"Emergency terminal restoration triggered by signal %d"

// PTY management errors

"PTY allocation failed: %s (check system limits with 'ulimit -n')"

"Child process termination detected for pane %d (exit status: %d)"

"Signal forwarding failed to process group %d: %s"

// Terminal emulation errors

"Malformed escape sequence in buffer (state: %d, length: %d)"

"UTF-8 sequence assembly failed: invalid byte sequence"

"Screen buffer corruption detected during cursor movement"

// Layout management errors

"Layout tree recalculation failed: insufficient space for minimum pane dimensions"

"Pane split operation rejected: would violate minimum size constraints"

"Focus state synchronization error: active pane %d not found in layout"
```

D. Logging and Debug Output

Structure debug output using consistent terminology for easier troubleshooting:

```

debug_log(LOG_INFO, "PTY_MANAGER",
    "Created PTY session: master_fd=%d, slave_fd=%d, child_pid=%d",
    session->master_fd, session->slave_fd, session->child_pid);

debug_log(LOG_DEBUG, "TERMINAL_EMULATOR",
    "Escape sequence parsed: CSI %s (params: %d, %d)",
    sequence_name, param1, param2);

debug_log(LOG_TRACE, "WINDOW_MANAGER",
    "Layout recalculation: pane %d dimensions (%d,%d,%d,%d)",
    pane->id, pane->x, pane->y, pane->width, pane->height);

```

E. Code Organization by Terminology Domains

Organize source files and modules using the terminology domains to maintain clear boundaries:

```

terminal-multiplexer/
├── src/
│   ├── pty/          # PTY management terminology domain
│   │   ├── session.c # pty_session_t, PTY allocation, child process termination
│   │   └── process.c # session leader, controlling terminal, signal forwarding
│   ├── terminal/    # Terminal emulation terminology domain
│   │   ├── parser.c  # escape sequence, state machine, CSI sequence
│   │   ├── screen.c  # screen buffer, terminal cell, cursor position
│   │   └── utf8.c    # UTF-8 decoder, character encoding, wide characters
│   ├── window/      # Window management terminology domain
│   │   ├── layout.c  # layout tree, split ratio, minimum pane dimensions
│   │   ├── pane.c    # pane, focus management, border characters
│   │   └── render.c  # screen composition, frame-based rendering
│   ├── input/        # Input handling terminology domain
│   │   ├── handler.c # input routing, prefix key mechanism, command mode
│   │   ├── bindings.c# key binding, input event processing
│   │   └── terminal.c# raw terminal mode, terminal restoration
│   └── core/         # System integration terminology domain
│       ├── event_loop.c # event loop, I/O multiplexing, select() system call
│       ├── signals.c   # signal handling, signal-safe operations
│       └── error.c    # error handling, graceful degradation
└── tests/
    ├── unit/          # component unit tests terminology
    ├── integration/  # integration testing with real PTYs
    └── milestones/   # milestone verification checkpoints

```

F. Milestone Verification Using Terminology

Each milestone completion can be verified by demonstrating understanding and implementation of its key terminology:

Milestone 1 Checkpoint:

- Demonstrate PTY allocation creating valid master/slave file descriptor pairs
- Show child process becoming session leader with PTY slave as controlling terminal
- Verify terminal size propagation using TIOCSWINSZ ioctl
- Confirm proper child process termination handling via SIGCHLD

Milestone 2 Checkpoint:

- Show escape sequence parser correctly transitioning between states
- Demonstrate screen buffer maintaining accurate cursor position and terminal cells
- Verify UTF-8 decoder handling multi-byte character sequences
- Confirm scrollback buffer storing lines that scroll off screen

Milestone 3 Checkpoint:

- Demonstrate layout tree correctly representing pane hierarchy
- Show split ratio calculations producing valid pane dimensions
- Verify screen composition combining multiple panes with borders
- Confirm focus management updating active pane highlighting

Milestone 4 Checkpoint:

- Show prefix key mechanism correctly entering command mode
- Demonstrate input routing sending keystrokes to appropriate destinations
- Verify terminal restoration working during both normal and emergency exit
- Confirm key bindings correctly mapping input to multiplexer commands

Mastering this terminology is essential for successful terminal multiplexer development. The terms provide a shared vocabulary for understanding the complex interactions between operating system services, terminal protocols, and user interface design that make terminal multiplexing possible.