The Gauntlet Shell: Design Document

Overview

This document outlines the design for building a Unix shell from scratch. The core architectural challenge is managing concurrent processes, coordinating their input/output streams, and correctly handling user signals, all while maintaining a responsive command-line interface. The design must bridge the gap between a linear user command and the complex, parallel world of Unix processes and file descriptors.

> 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 (1-6) – This foundational section frames the entire project's challenge.

A Unix shell is the primary interface through which users and systems interact with an operating system. It sits between the user and the kernel, translating human-readable commands into the complex system calls that drive computation. Building a shell is a **rite of passage** in systems programming because it forces engagement with the core mechanisms of Unix: process creation, inter-process communication, file descriptor management, and signal handling. Unlike applications that merely *use* these APIs, a shell must **orchestrate** them, creating and managing the very processes that constitute the system's workload.

This design document addresses the fundamental challenge: constructing a reliable, responsive command interpreter that faithfully implements the core behaviors of standard shells like `bash` or `zsh`, while remaining understandable and maintainable as an educational codebase. The core tension lies in bridging the **linear, textual command input** with the **parallel, stateful world of Unix processes**. A user types `ls -l | grep foo > output.txt &` as a single line, but the shell must spawn two concurrent processes, connect them with a pipe, redirect the final output to a file, and run them in the background—all while remaining responsive to the next user command and cleaning up resources appropriately.

Mental Model: The Shell as an Air Traffic Control Tower

Imagine a busy airport. Aircraft (processes) need to take off, fly, and land. The **air traffic control tower (the shell)** does not fly the planes itself. Instead, it:

1. **Receives flight plans (command lines)** from pilots (users).
2. **Allocates runways (standard streams - stdin, stdout, stderr)** for takeoff and landing.
3. **Coordinates sequencing (pipes and redirections)** so one plane's output becomes another's input.
4. **Manages ground holds (foreground/background jobs)** deciding which planes are actively using the runway (terminal) and which wait in holding patterns.
5. **Handles emergencies (signals)** like sudden cancellations (Ctrl+C) or temporary holds (Ctrl+Z).

The tower's critical responsibility is to maintain **safety and liveness**: no plane should crash due to miscommunication (process errors), and the runway must never be permanently blocked (the shell must stay responsive). This analogy highlights the shell's role as a **manager and dispatcher**, not the entity doing the primary work. It creates an environment where processes can run safely and cooperatively, mediates their interactions, and reports their outcomes.

The Core Problem: Translating Text into Parallel System Actions

The primary technical challenge decomposes into three tightly-coupled sub-problems:

1. **Interpretation:** Converting a string of characters, with its embedded spaces, quotes, and special operators (`|` , `>` , `&`), into a structured **abstract syntax tree (AST)** that unambiguously represents command relationships.
2. **Environment Setup:** For each node in the AST (a command), configuring its execution context before launch. This includes:
   - **Argument vector:** Building the `argv` array for `execvp` .
   - **I/O connections:** Manipulating file descriptors via `dup2` to establish pipes between commands or redirect input/output to files.
   - **Process attributes:** Setting process group IDs for job control, and signal dispositions.
3. **Lifecycle Management:** Spawning processes (via `fork` / `exec`), monitoring their state changes, reaping their exit statuses (via `waitpid`), and propagating signals (like SIGINT for Ctrl+C) to the correct foreground process group—all while preventing zombie processes and race conditions.

The difficulty amplifies because these sub-problems are not sequential phases but interleaved. Setting up a pipe requires creating the pipe *before* forking, and correctly closing file descriptors *in each process* to avoid deadlocks. Managing background jobs requires handling the `SIGCHLD` signal *asynchronously* while the main loop might be blocked on reading user input. The design must carefully separate concerns while managing these intricate, stateful dependencies.

> **Key Insight:** The shell's main loop is a **finite state machine** managing concurrent child processes. Each command line expands into a graph of processes and file descriptors, and the shell must track this graph's state until all its nodes (processes) terminate.

Existing Approaches and the Unix Philosophy

Shell design exists on a spectrum between two architectural philosophies:

Approach	Description	Example	Pros	Cons
Monolithic Shell	The shell process itself contains built-in implementations for most commands (e.g., `cd` , `exit`) and may even implement complex logic for loops and conditionals directly in its codebase.	Early shells like the Thompson shell, some embedded system shells.	Tight integration can be faster for built-ins; simpler deployment (single binary).	Bloated; violates the Unix "do one thing well" principle; hard to extend with new commands.
Microkernel Shell	The shell is a minimal coordinator. It only handles parsing, process creation, and job control. All actual work (even for `ls` , `grep`) is delegated to external programs. Built-ins are limited to actions that *must* affect the shell's own state (like `cd`).	`bash` , `zsh` , `dash` (the standard model).	Adheres to Unix philosophy; extremely extensible (any program becomes a command); keeps shell codebase small and focused.	Overhead of forking/execing for simple tasks; requires a rich ecosystem of external utilities.

> **Decision: Adopt the Microkernel (Unix) Architecture**
>
> - **Context:** Our shell is an educational project aiming to teach core Unix systems programming concepts. The microkernel model directly exposes these concepts (fork, exec, PATH lookup) and is the industry standard.
> - **Options Considered:**
>   1. **Monolithic:** Implement common utilities (`ls` , `grep`) as internal functions.
>   2. **Microkernel:** Only implement process orchestration and a few state-changing built-ins.
> - **Decision:** Adopt the Microkernel architecture.
> - **Rationale:** This model forces the implementation to correctly handle the full `fork` / `exec` lifecycle, PATH searching, and inter-process communication (pipes). It mirrors real-world shells, making the learning experience transferable. It also keeps the project scope focused on the shell's *unique* responsibilities, rather than re-implementing utilities.
> - **Consequences:** The shell will depend on the system's external command suite. Performance for trivial operations will be slower than a built-in implementation, but this is acceptable for learning. The design must include a clean mechanism for distinguishing and handling built-in commands (like `cd`) that cannot be external programs.

This design document follows the microkernel philosophy. Our shell, which we'll call **The Gauntlet Shell**, will be a lean process manager that excels at parsing, process creation, and job control, delegating real work to the vast ecosystem of Unix tools. The subsequent sections detail how to structure such a system to be correct, robust, and understandable.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option (Recommended)	Advanced Option (Alternative)
Language / Stdlib	Standard C (C11) with POSIX APIs (`<unistd.h>` , `<sys/wait.h>` , `<signal.h>`)	Rust (using `nix` and `libc` crates) or Go (using `syscall` and `os/exec` packages)
Parser Implementation	Recursive descent parser on a tokenized string	Table-driven parser or using a parser generator (lex/yacc)
Job Table Data Structure	Dynamic array of structs	Linked list or hash map keyed by job ID
Signal Handling	Self-pipe trick or `pselect()` to handle signals safely in main loop	Using `signalfd()` (Linux-specific) or dedicated signal handler thread

Recommended File/Module Structure:

```
gauntlet_shell/
├── Makefile                    # Build automation
├── shell.c                     # Main entry point, REPL loop
├── include/                    # Public header files
│   └── shell.h                 # Main shell API and data structures
├── src/
│   ├── parser.c                # Tokenization and AST construction
│   ├── parser.h                # Parser module interface
│   ├── executor.c              # fork/exec, pipe, redirection logic
│   ├── executor.h              # Executor module interface
│   ├── job_control.c           # Job table, fg/bg, signal forwarding
│   ├── job_control.h           # Job control module interface
│   ├── builtins.c              # Implementations of cd, exit, export, etc.
│   └── builtins.h
└── tests/                      # Manual and automated test scripts
    ├── test_runner.sh
    └── test_cases.txt
```

Core Logic Skeleton Code:

Since this is the context section, we provide a skeleton for the **main shell entry point** (the REPL driver) which ties everything together. Detailed skeletons for parser, executor, and job control will appear in their respective component sections.

File: `shell.c`

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include "include/shell.h"

#include "src/parser.h"

#include "src/executor.h"

#include "src/job_control.h"


/**

 * Main Read-Eval-Print Loop (REPL) for the Gauntlet Shell.

 * This is the air traffic control tower—the central coordinator.

 */

int main(int argc, char **argv) {

    ShellState state;

    shell_state_init(&state); // TODO 1: Initialize global shell state (job table, etc.)


    // TODO 2: Set up signal handlers for SIGINT, SIGTSTP, SIGCHLD.

    //         Use sigaction() for robustness. Ensure SIGCHLD handler reaps background jobs.


    char *line = NULL;

    size_t linecap = 0;

    ssize_t linelen;


    while (true) {

        // TODO 3: Before printing prompt, check for and report any completed background jobs.

        //         Use job_control_check_completed_jobs(&state).


        // Print prompt and get input

        printf("gshell> ");

        fflush(stdout); // Ensure prompt appears before blocking input

        linelen = getline(&line, &linecap, stdin);


        // TODO 4: Handle EOF (Ctrl+D) – linelen will be -1. Break loop and exit.


        // Remove trailing newline

        if (linelen > 0 && line[linelen - 1] == '\n') {
```

```
            line[linelen - 1] = '\0';
        }

        // TODO 5: Skip empty lines (just newline or whitespace). Use a helper function.

        // TODO 6: Parse the input line into an abstract syntax tree (AST).
        //         ParseResult result = parse_command_line(line, &state);
        //         Handle parse errors: print message, free resources, continue loop.

        // TODO 7: Execute the parsed command AST.
        //         int exit_status = execute_ast(result.ast, &state);

        // TODO 8: Clean up any memory allocated for the AST and tokens.
        //         This is critical to prevent leaks on every loop iteration.
    }

    // TODO 9: Cleanup before exit: free line buffer, clean up job table, etc.
    free(line);
    shell_state_cleanup(&state);
    return 0;
}
```

Language-Specific Hints (C):

- Use `getline()` (POSIX) for safe, dynamic line reading. If portability is a concern, provide a fallback using `fgets()` and reallocation.
- Always check the return value of `getline()` for `-1` (EOF).
- Use `sigaction()` instead of `signal()` for reliable signal handling. Block signals in handlers and use a volatile sig_atomic_t flag to communicate with the main loop.
- For the job table, a simple fixed-size array (e.g., `struct Job jobs[MAX_JOBS];`) is sufficient for learning. A linked list is more flexible.
- Remember that `getcwd()` can fail; always check its return value when implementing `pwd` or updating `PWD`.

Common Pitfalls to Avoid from the Start:

- ⚠️ **Pitfall: Blocking reads that ignore signals.** If you use `fgets()` or `getline()` directly, a SIGCHLD from a background job completion won't interrupt the call, delaying job status reports. **Fix:** Use a signal handler that writes to a self-pipe, and use `select()` / `poll()` on stdin and the self-pipe, or use `pselect()` to atomically unblock signals before reading.
- ⚠️ **Pitfall: Memory leaks on every loop.** Forgetting to free the parsed AST, token arrays, or command strings after each command line will cause the shell's memory footprint to grow continuously. **Fix:** Establish a clear ownership model: the parser allocates, the executor may use, and the REPL loop must free before the next iteration. Use tools like `valgrind`.
- ⚠️ **Pitfall: Not handling EOF (Ctrl+D) gracefully.** This should terminate the shell cleanly, not crash. **Fix:** Check if `getline()` returns `-1` and break the main loop.
- ⚠️ **Pitfall: Prompt not appearing due to buffering.** If you print the prompt without a newline, it might stay in the buffer. **Fix:** Use `fflush(stdout)` after `printf` for the prompt.

> **Milestone(s):** All (1-6) – This section defines the project's scope and quality expectations, framing the concrete deliverables and constraints for the entire system.

Goals and Non-Goals

Defining a clear scope is the first step in taming a complex project. A shell, by its nature, is an application that could grow without bound—adding scripting languages, complex expansions, plugins, and compatibility layers. To provide a focused, educational journey from zero to a functional tool, we must draw explicit boundaries around what we will and will not build. This section establishes those boundaries, ensuring the project remains a manageable exploration of core systems programming concepts without spiraling into a multi-year endeavor.

Functional Goals (Must-Have Features)

Think of these goals as the **flight checklist for a commercial airliner**. Before the plane (your shell) is certified to fly (be considered complete), it must demonstrably perform a set of essential, non-negotiable operations. Each milestone corresponds to a critical system coming online, from basic propulsion (launching processes) to advanced autopilot (job control). The following table enumerates these required capabilities, mapping each to its corresponding project milestone.

Milestone	Feature Category	Capability Description	Expected User Experience & Key Behaviors
1	**Basic REPL & Command Execution**	The foundational interactive loop and ability to launch external programs.	The shell displays a prompt (`gauntlet$ `), reads a line of input, and executes a simple command like `ls -la` . It correctly finds binaries in the user's `PATH` , waits for the child process to finish, and reports a non-zero exit status if the command fails. Empty lines and extra whitespace are handled gracefully.
2	**Built-in Commands**	Commands implemented directly within the shell, not via external binaries.	The user can change directories with `cd [dir]` (defaulting to `HOME`), exit the shell with `exit [code]` , set environment variables with `export VAR=value` , and print text with `echo` . These commands affect the shell's own state (like its current working directory) and that state is correctly inherited by subsequently launched child processes.
3	**I/O Redirection**	Redirecting a command's standard input, output, and error streams to/from files.	The user can write command output to a file with `ls > file.txt` (overwrites) or `ls >> file.txt` (appends). They can read command input from a file with `wc -l < file.txt` . Standard error can be separately redirected with `cmd 2> errors.log` . Multiple redirections in one command work correctly (e.g., `cmd < input.txt > output.txt 2> errors.txt`).
4	**Pipes**	Connecting the output of one command to the input of another, forming a pipeline.	The user can chain commands with the pipe operator: `ls -l
5	**Background Jobs & Listing**	Executing commands asynchronously and tracking them.	Appending `&` to a command (e.g., `sleep 10 &`) causes it to run in the background, immediately returning the user to the prompt. The shell assigns a job ID and prints it. The `jobs` command lists all active background jobs with their ID, status ("Running", "Stopped", "Done"), and the command string. The shell automatically reports completed background jobs before the next prompt.
6	**Job Control (fg, bg, Ctrl+Z)**	Suspending, resuming, and moving jobs between foreground and background.	Pressing `Ctrl+Z` sends `SIGTSTP` to the entire foreground process group (e.g., a pipeline), suspending it and returning control to the shell. The `fg [%jobid]` command brings a background or stopped job to the foreground, resuming it if stopped. The `bg [%jobid]` command resumes a stopped job in the background. `Ctrl+C` (`SIGINT`) is correctly delivered to the foreground job, not the shell itself.

Non-Functional Goals (Quality Attributes)

Beyond *what* the shell does, we must define *how well* it should perform. These are the quality attributes—the engineering standards for our system. Imagine building a **bridge**. Functional goals are the lanes, ramps, and load-bearing pillars. Non-functional goals are the safety factors, the smoothness of the ride, and its resilience to high winds or heavy traffic. Our shell must be correct, robust, and responsive within its defined scope.

Quality Attribute	Description & Rationale	Concrete, Measurable Target
Correctness	The shell's behavior should match the behavior of a reference shell (e.g., `bash` or `dash`) for all implemented features. This is critical for user intuition and script compatibility.	For a suite of test commands covering redirections, pipes, and built-ins, the output and exit status produced by our shell should be byte-for-byte identical to that produced by `bash` (in POSIX-compliant mode) for the same command.
Robustness	The shell must not crash, hang, or leak resources (memory, processes, file descriptors) due to malformed user input, command failures, or signals. It should degrade gracefully.	1. **No Crashes:** Fuzzing the input with random strings, unmatched quotes, or giant commands does not cause a segmentation fault or abort. 2. **No Hangs:** Ill-formed pipelines (e.g., `ls
Responsiveness	The user interface must feel immediate. The prompt should reappear as soon as possible after a command finishes, and the shell must remain interactive during long-running foreground commands (i.e., accept and handle signals).	1. **Prompt Latency:** After a foreground command (or pipeline) exits, the new prompt appears within a perceptually instantaneous timeframe (<100ms in normal conditions). 2. **Signal Handling:** While a foreground command like `sleep 100` is running, `Ctrl+C` terminates it and returns the prompt promptly. `Ctrl+Z` suspends it promptly.
Simplicity & Learnability	The internal design should be clear and well-structured, serving as an educational artifact. This prioritizes straightforward algorithms and data structures over hyper-optimized but obscure implementations.	The core logic for parsing and execution (excluding helpers) should be understandable by a single developer reading the source code within a reasonable timeframe. Complex state is encapsulated, and control flow avoids deep, nested conditionals where possible.

> **Design Insight:** The trade-off between correctness and simplicity is a key tension. We prioritize matching bash's *observable behavior* for core features, but not necessarily its ultra-complex internal implementation. For example, we will implement a simpler, recursive descent parser rather than a fully POSIX-compliant grammar parser, as long as it handles the test cases correctly.

Explicit Non-Goals

To prevent scope creep and maintain focus on core systems concepts, we explicitly exclude several common shell features. Consider this the **"not-in-this-version" list** for a product launch. These are features that, while useful, would distract from the fundamental lessons about processes, file descriptors, and signals. They can be added later as extensions but are not required for the shell to be considered functionally complete per our learning objectives.

Feature Category	Specific Examples	Reason for Exclusion / Notes
Advanced Scripting & Control Flow	`if` / `then` / `fi`, `for` / `while` loops, functions, `case` statements.	These are language interpreter features, shifting focus from process management to parsing and evaluating a complex language. They can be a large project unto themselves.
Complex Expansions	Brace expansion (`{a,b,c}`), arithmetic expansion (`$((1+2))`), advanced parameter expansion (`${var:-default}`), tilde expansion for users other than `~` (e.g., `~username`).	These require significant parsing complexity and, in some cases, interaction with system databases (`/etc/passwd`). We limit variable expansion to simple `$VAR` and `~` for `$HOME` only.
Interactive Convenience Features	Command history (Up/Down keys), tab completion, aliases, custom prompt strings with escape codes (`\u` , `\w`), line editing (readline-style).	These features heavily involve terminal I/O control (`termios`) and UI state management, which is a separate domain from core shell process orchestration.
Advanced Job Control Features	`nice` , `disown` , job specification with `%` prefixes beyond simple job ID (e.g., `%string` , `%+` , `%-`).	These add complexity to job management without introducing new core concepts. Our job control focuses on the fundamental process group and signal mechanics.
Networking & Advanced Redirection	Here-documents/here-strings (`<<` , `<<<`), process substitution (`<(cmd)`), TCP/UDP redirection (`/dev/tcp/`).	These are either syntactic sugar or involve significant additional subsystems (networking). They build upon, but do not fundamentally change, the core redirection model.
POSIX Full Compliance	Supporting every edge case and all special parameters (`$@` , `$#` , `$?` , `$$` , etc.) exactly as specified by the POSIX standard.	Full compliance is an enormous task. We aim for *compatibility* in common use cases, not certification.
Security Hardening	Restricted shells, `sudo` integration, advanced auditing, and sandboxing.	While important for production shells, these are orthogonal to the educational goal of understanding the process lifecycle.

> **Key Principle:** By strictly defining these non-goals, we channel development effort into deeply understanding and correctly implementing the foundational mechanics of `fork` , `exec` , `dup2` , `pipe` , `setpgid` , and signal handling—the true educational heart of the project.

Implementation Guidance

For this foundational "Goals and Non-Goals" section, implementation guidance takes the form of a **progress validation framework**. Rather than writing code, you will establish a system to verify that your shell meets each of the stated functional and non-functional goals as you build it.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (Optional)
Progress Tracking	Manual checklist and ad-hoc testing in terminal.	Automated test suite using a shell script or testing framework (e.g., TAP - Test Anything Protocol).
Reference Shell	`bash` (use `bash --posix` for stricter compliance) or `dash` (a simpler, POSIX-focused shell).	Writing golden-file tests that capture reference output for a fixed set of commands.
Resource Leak Detection	Manual observation using `ps` and `lsof` in another terminal.	Integrating `valgrind` or using AddressSanitizer (`-fsanitize=address`) during compilation for automatic detection.

B. Recommended File/Module Structure

Create a `tests/` directory from the project's start to organize validation artifacts.

```
gauntlet-shell/
  src/                # Your shell source code
    main.c
    parser.c
    executor.c
    job_control.c
    # ... other modules
  tests/              # Validation artifacts
    milestone1.bats    # Optional: Automated tests using Bash Automated Testing System
    milestone2.sh      # A script to test built-in commands
    test_commands.txt  # A list of manual test commands
    expected/          # Directory for expected output files (for golden testing)
  Makefile
  README.md
```

C. Infrastructure Starter Code: A Simple Test Runner

While you can test manually, a basic test script helps systematize validation. Create `tests/run_tests.sh`:

```bash
#!/bin/bash

# Simple test runner for the Gauntlet Shell

# Usage: ./run_tests.sh <path_to_shell_executable>


SHELL=$1

if [ -z "$SHELL" ]; then

    SHELL="../gauntlet"  # default path

fi


echo "Testing shell: $SHELL"

echo "======================"


test_command() {

    local description="$1"

    local command="$2"

    local expected_status="${3:-0}"



    echo -n "Testing: $description... "

    # Run the command in our shell, capture exit status

    # Use a here-document to feed the command and exit

    $SHELL <<-EOF 2>&1

        $command

        exit

        EOF

    actual_status=$?



    if [ $actual_status -eq $expected_status ]; then

        echo "PASS"

    else

        echo "FAIL (expected $expected_status, got $actual_status)"

    fi

}


# Example test for Milestone 1: Exit status

test_command "Successful command exit status" "/bin/true" 0

test_command "Failing command exit status" "/bin/false" 1
```

```
# More tests can be added here per milestone
```

D. Core Logic Skeleton Code: A Validation Checklist Integrator

Within your shell's main source file, you might add a built-in command `:test` (a no-op in standard shells) that runs internal diagnostics. This is a more advanced pattern.

```c
// In builtins.c, alongside cd, exit, etc.
int builtin_test(int argc, char **argv, ShellState *state) {
    // This is an advanced concept: a built-in for self-validation.
    // It's provided here as a skeleton for future extension.
    fprintf(stderr, "Shell self-test not yet implemented.\n");
    // TODO 1: Check job table integrity (no invalid PIDs).
    // TODO 2: Verify no file descriptors are unexpectedly open in the shell process.
    // TODO 3: Run a suite of predefined commands and compare output to reference.
    return 0;
}
```

E. Language-Specific Hints (C)

- **Exit Status:** Use `WEXITSTATUS(status)` from `<sys/wait.h>` to extract the child's exit code from the status integer returned by `waitpid`. Remember that `WIFEXITED(status)` must be true first.
- **Resource Leak Checks:** Compile with `gcc -fsanitize=address -g` to detect memory leaks. Run your shell under `valgrind --leak-check=full` to trace the origin of leaks.
- **Signal Safety:** Avoid calling non-async-signal-safe functions (like `printf`, `malloc`) in signal handlers. Use a volatile sig_atomic_t flag and check it in the main loop.

F. Milestone Checkpoint

After completing each milestone, run the following manual verification steps:

Milestone	Validation Command	Expected Outcome
1	`./gauntlet` then type `ls -la`	Lists directory contents, then returns to `gauntlet$` prompt.
1	`./gauntlet` then type `/bin/false; echo $?`	Prints `1` (or the exit code of false).
2	`./gauntlet` then type `cd /tmp; pwd`	Prints `/tmp`. The `pwd` built-in shows the changed directory.
3	`echo "hello"	./gauntlet then type` cat < input.txt > output.txt`	
4	`./gauntlet` then type `ls	wc -l`	
5	`./gauntlet` then type `sleep 2 &`	Immediately prints a job ID like `[1] 12345` and shows the prompt. After 2 seconds, prints `[1]+ Done sleep 2`.
6	`./gauntlet`, type `sleep 100`, then press `Ctrl+Z`	Prints `[1]+ Stopped sleep 100` and returns to prompt. Typing `fg` resumes the sleep in the foreground.

G. Debugging Tips for Goal Validation

Symptom	Likely Cause	How to Diagnose	Fix
Shell output differs from bash for simple commands like `ls`.	Incorrect argument passing to `execvp` or mishandling of the `PATH` environment variable.	Add debug prints to show the `argv` array passed to `execvp`. Compare the resolved path using `which ls`.	Ensure `argv` is NULL-terminated. Use `execvp` correctly, which does `PATH` search.
Background jobs are not reported as 'Done'.	`SIGCHLD` handler is missing or incorrectly implemented, or job table cleanup is not triggered.	Add a print statement in the `SIGCHLD` handler. Use `jobs` command to see if completed jobs linger in the table.	Implement a proper `SIGCHLD` handler that calls `waitpid` with `WNOHANG` to reap all terminated children and updates the job table.
`Ctrl+C` exits the shell instead of the foreground command.	The shell is not setting itself as the foreground process group in the terminal, or not correctly assigning the child to a new process group.	Print the process group IDs (`getpgrp()`) of the shell and child in the execution phase.	Use `tcsetpgrp()` to give the terminal to the child's process group before waiting, and take it back after.

High-Level Architecture

> **Milestone(s):** All (1-6) – This foundational architectural overview establishes the structural blueprint for the entire shell system, showing how all components interact to transform user commands into process execution while maintaining responsive job control.

This section presents the architectural blueprint of the Gauntlet Shell. At the highest level, the shell transforms a linear string of user input into potentially complex, parallel process execution while maintaining interactive responsiveness. The architecture follows a **component pipeline model** where data flows sequentially through four specialized components, each with clearly defined responsibilities and interfaces. This separation of concerns allows each component to focus on a single aspect of the shell's functionality, making the system more maintainable, testable, and understandable.

Component Overview and Data Flow

Think of the shell's architecture as a **specialized factory assembly line** for process creation. Raw user input enters at one end as a text string and emerges at the other end as running processes with coordinated I/O streams. Each station in the assembly line has a specialized role: one station parses the blueprint (command syntax), another station sets up the workshop environment (I/O redirection and pipes), and a control station manages the workers (processes) throughout their lifecycle.

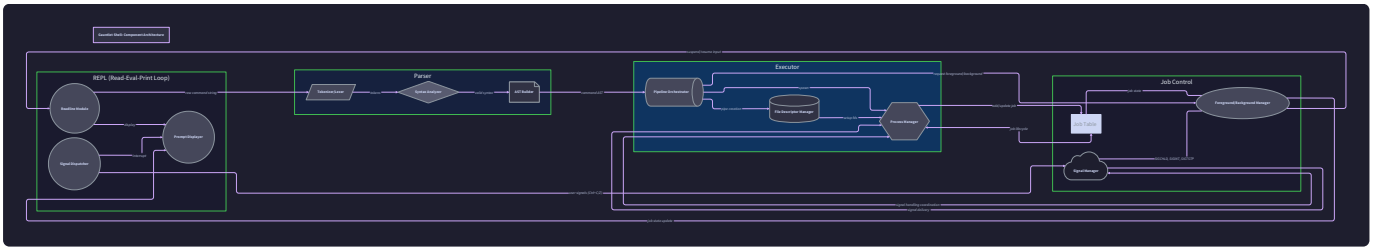
The architecture comprises four primary components, each with distinct responsibilities and interfaces:

Component	Primary Responsibility	Key Sub-Modules	Critical Dependencies
REPL (Read-Eval-Print Loop)	Orchestrates the interactive shell session, coordinating all other components and maintaining the main event loop.	Signal handlers, prompt display, input reader, job status reporter	Standard I/O streams, terminal control, signal delivery
Parser	Translates raw command-line text into structured internal representations (AST) that the executor can process.	Tokenizer, recursive descent parser, syntax validator	String manipulation, memory allocation, command grammar
Executor	Transforms parsed command structures into running processes with correct I/O setup and executes built-in commands.	Process creator (fork/exec), file descriptor manipulator (dup2), built-in command dispatcher	Unix process API, file system operations, environment variables
Job Control Manager	Manages background and suspended processes, maintains job table, and handles terminal signal forwarding.	Job table operations, SIGCHLD handler, foreground/background state manager, terminal control group manager	Process groups, signal handling, asynchronous event processing

The data flow through these components follows a predictable path for each command cycle:

1. **Input Phase**: The REPL displays a prompt and reads a line of user input (e.g., `ls -l | grep foo > output.txt &`).
2. **Parse Phase**: The REPL passes the raw input string to the Parser, which tokenizes the input and constructs an Abstract Syntax Tree (AST) represented by a `Command` struct linked list (for pipelines). The Parser returns a `ParseResult` containing either a valid AST or an error message.
3. **Job Creation Phase**: For commands ending with `&` (background execution), the REPL creates a `Job` entry in the `JobTable` managed by the Job Control Manager, assigning it a unique job ID and setting its initial state.
4. **Execution Phase**: The REPL invokes the Executor with the parsed AST. The Executor traverses the command structure, setting up any required I/O redirections, creating pipes for command pipelines, forking child processes, executing commands (either external via `execvp` or built-in via dispatch table), and managing process group assignments.
5. **Completion Phase**: For foreground commands, the Executor waits for process completion and returns the exit status. For background commands, control returns immediately to the REPL. The Job Control Manager's SIGCHLD handler asynchronously updates job states when background processes complete.
6. **Status Reporting Phase**: Before displaying the next prompt, the REPL calls `job_control_check_completed_jobs` to report any recently completed background jobs and clean up finished entries from the job table.

This data flow is visualized in the component architecture diagram:



The diagram shows not only the primary data flow (solid arrows) but also critical event flows (dashed arrows), particularly the SIGCHLD signal from completed child processes to the Job Control Manager, which then notifies the REPL of status changes.

> **Design Insight:** The unidirectional data flow from REPL → Parser → Executor creates a clean separation where each component only needs to understand the interface of the next component in the chain. This modularity is crucial for managing the inherent complexity of a shell, which must handle synchronous user interaction, asynchronous process events, and complex process relationships simultaneously.

The `ShellState` struct serves as the **global coordination hub** that all components access (either directly or through function parameters). It contains:

- The `JobTable` tracking all background and suspended jobs
- The `foreground_pgid` indicating which process group currently controls the terminal
- The `terminal_fd` file descriptor for terminal-specific operations
- The `last_exit_status` storing the exit code of the most recently completed foreground command

This shared state enables coordination between components without tight coupling. For example, when the Executor runs a foreground pipeline, it sets the `foreground_pgid` in `ShellState`. When the user presses Ctrl+Z, the REPL's SIGTSTP handler reads this `foreground_pgid` to determine which process group to signal.

Recommended File/Module Structure

Organizing the codebase with clear separation of concerns from the beginning prevents the common "big ball of mud" anti-pattern that often emerges in shell implementations. The recommended structure follows a **layered architecture** with clear interface boundaries between components.

Layer	Purpose	Files
Application Entry Point	Contains the `main()` function that initializes the shell and starts the REPL	`main.c`
Core Component Implementations	Implements the four main architectural components with clear separation	`repl.c`, `parser.c`, `executor.c`, `job_control.c`
Component Headers	Public interfaces for each component (function declarations, struct definitions)	`repl.h`, `parser.h`, `executor.h`, `job_control.h`
Data Model Definitions	Central definition of all shared data structures and constants	`datamodel.h`
Built-in Command Implementations	Self-contained implementations of each built-in command	`builtins/` directory with individual files
Utility Functions	Reusable helper functions for string manipulation, memory management, etc.	`utils.c`, `utils.h`

The complete recommended directory structure:

```
gauntlet_shell/
├── Makefile                    # Build configuration and dependencies
├── main.c                      # Entry point: main() function, shell initialization
├── datamodel.h                 # Central definitions: ShellState, Command, Job, etc.
├── repl.h                      # REPL component interface
├── repl.c                      # REPL component implementation
├── parser.h                    # Parser component interface
├── parser.c                    # Parser component implementation
├── executor.h                  # Executor component interface
├── executor.c                  # Executor component implementation
├── job_control.h               # Job Control Manager interface
├── job_control.c               # Job Control Manager implementation
├── builtins/                   # Built-in command implementations
│   ├── cd.c                    # cd command implementation
│   ├── exit.c                  # exit command implementation
│   ├── export.c                # export command implementation
│   ├── jobs.c                  # jobs command implementation
│   ├── fg.c                    # fg command implementation
│   ├── bg.c                    # bg command implementation
│   └── builtins.h              # Built-in command dispatch table and prototypes
├── utils.h                     # Utility function prototypes
├── utils.c                     # Utility function implementations
├── tests/                      # Test suite
│   ├── test_parser.c           # Parser unit tests
│   ├── test_executor.c         # Executor unit tests
│   └── integration.sh          # End-to-end integration tests
└── README.md                   # Project documentation
```

Each header file follows a consistent pattern defining the **public interface** for its component:

- **Function prototypes** for all functions other components may call
- **Struct definitions** for types owned by that component (when appropriate)
- **Documentation comments** explaining each function's purpose, parameters, return values, and error conditions
- **Preprocessor guards** to prevent multiple inclusion

The `datamodel.h` header deserves special attention as it defines the **common vocabulary** used throughout the system:

```c
#ifndef DATAMODEL_H

#define DATAMODEL_H


#include <sys/types.h>

#include <stdbool.h>


// Enum definitions

typedef enum { REDIR_INPUT, REDIR_OUTPUT, REDIR_APPEND, REDIR_ERROR } RedirType;

typedef enum { JOB_RUNNING, JOB_STOPPED, JOB_DONE } JobState;


// Core data structures (as specified in naming conventions)

typedef struct Redirection {

    RedirType type;

    char* filename;

    int fd;

} Redirection;


typedef struct Command {

    char** argv;

    int argc;

    Redirection* input_redir;

    Redirection* output_redir;

    struct Command* next;  // For pipeline chaining

} Command;


typedef struct Job {

    int id;

    pid_t pgid;

    char* command_string;

    JobState state;

    Command* command_list;

} Job;


typedef struct JobTable {

    Job* jobs;

    int capacity;

    int count;
```

```
} JobTable;

typedef struct ShellState {
    JobTable job_table;
    pid_t foreground_pgid;
    int terminal_fd;
    int last_exit_status;
} ShellState;

// Constants
#define MAX_JOBS 100
#define MAX_LINE_LEN 4096

#endif // DATAMODEL_H
```

This centralized data model definition ensures consistency across all components. When a component needs to modify a data structure (e.g., the Parser building a `Command` AST or the Job Control Manager updating a `Job` state), it includes `datamodel.h` to access the definitions.

> **Design Insight:** Placing all core data structure definitions in a single header file (`datamodel.h`) creates a single source of truth for the system's data model. This prevents subtle bugs that arise when different components have slightly different understandings of struct layouts or enum values. It also makes the data dependencies between components explicitly visible in the `#include` directives.

The **dependency direction** between components is strictly controlled:

- `main.c` depends on all component headers (`repl.h` , `parser.h` , etc.)
- `repl.c` depends on `parser.h` , `executor.h` , `job_control.h` , and `datamodel.h`
- `parser.c` depends only on `datamodel.h` and its own header
- `executor.c` depends on `job_control.h` , `builtins/builtins.h` , and `datamodel.h`
- `job_control.c` depends only on `datamodel.h` and its own header
- Built-in command implementations depend only on `datamodel.h` and `builtins.h`

This controlled dependency graph ensures that changes to one component have minimal ripple effects on others. For example, if the Parser's internal tokenization algorithm changes, only `parser.c` needs recompilation—the REPL continues to call the same `parse_command_line` function with the same interface.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (Production-Ready)
Memory Management	Manual `malloc`/`free` with careful tracking	Reference-counted objects or arena allocator
String Handling	Standard C library (`strdup`, `strtok_r`)	Custom string buffer with safe operations
Error Reporting	`fprintf(stderr, ...)` with descriptive messages	Structured error objects with error codes
Signal Handling	Simple handlers setting volatile sig_atomic_t flags	Self-pipe trick with signalfd for async-safe handling
Command Lookup	Iterate through PATH directories with `access()`	Hash table cache of executable locations
Job Table Storage	Fixed-size array in `ShellState`	Dynamic array with growth factor or linked list

B. Recommended File/Module Structure Implementation

Create the following directory structure and files:

```
# Create the project root directory
mkdir gauntlet_shell
cd gauntlet_shell

# Create main source files
touch main.c datamodel.h repl.h repl.c parser.h parser.c executor.h executor.c \
      job_control.h job_control.c utils.h utils.c

# Create builtins directory and files
mkdir builtins
touch builtins/cd.c builtins/exit.c builtins/export.c builtins/jobs.c \
      builtins/fg.c builtins/bg.c builtins/builtins.h

# Create tests directory
mkdir tests
touch tests/test_parser.c tests/test_executor.c tests/integration.sh

# Create supporting files
touch Makefile README.md
```

C. Infrastructure Starter Code

`datamodel.h` - Complete Data Model Definitions:

```c
#ifndef DATAMODEL_H

#define DATAMODEL_H


#include <sys/types.h>

#include <stdbool.h>


// Enum definitions for redirection types and job states

typedef enum {

    REDIR_INPUT,    // < filename

    REDIR_OUTPUT,   // > filename

    REDIR_APPEND,   // >> filename

    REDIR_ERROR     // 2> filename

} RedirType;


typedef enum {

    JOB_RUNNING,    // Job is currently executing

    JOB_STOPPED,    // Job is suspended (Ctrl+Z)

    JOB_DONE        // Job has completed (but not yet reaped)

} JobState;


// Redirection structure: represents a single I/O redirection

typedef struct Redirection {

    RedirType type;     // Type of redirection (input, output, append, error)

    char* filename;     // Target filename (dynamically allocated)

    int fd;             // File descriptor number (2 for stderr, etc.)

} Redirection;


// Command structure: represents a single command in a pipeline

typedef struct Command {

    char** argv;            // Argument vector (NULL-terminated)

    int argc;               // Number of arguments

    Redirection* input_redir;   // Input redirection (NULL if none)

    Redirection* output_redir;  // Output redirection (NULL if none)

    struct Command* next;   // Next command in pipeline (NULL for last)

} Command;


// Job structure: represents a background or suspended job
```

```c
typedef struct Job {
    int id;                 // Job ID (assigned by shell, e.g., [1], [2])
    pid_t pgid;             // Process group ID for the job
    char* command_string;   // Original command string (for display)
    JobState state;         // Current state (running, stopped, done)
    Command* command_list;  // Parsed command pipeline (linked list)
} Job;

// JobTable structure: manages all background/suspended jobs
typedef struct JobTable {
    Job* jobs;              // Array of jobs (dynamically allocated)
    int capacity;           // Current capacity of the array
    int count;              // Number of active jobs in the array
} JobTable;

// ShellState structure: global shell state accessible to all components
typedef struct ShellState {
    JobTable job_table;     // Table of background/suspended jobs
    pid_t foreground_pgid;  // Process group ID of current foreground job
    int terminal_fd;        // File descriptor for the terminal
    int last_exit_status;   // Exit status of last foreground command
} ShellState;

// Constants for system limits
#define MAX_JOBS 100        // Maximum number of concurrent background jobs
#define MAX_LINE_LEN 4096   // Maximum input line length (fallback for getline)

// Function prototypes for shell state management
void shell_state_init(ShellState* state);
void shell_state_cleanup(ShellState* state);

#endif // DATAMODEL_H
```

`utils.h` - Utility Function Interface:

```c
#ifndef UTILS_H

#define UTILS_H


#include <stdbool.h>


// String utilities

char* string_duplicate(const char* src);

char* string_trim(char* str);

char** string_split(const char* str, const char* delimiter, int* count);

void string_array_free(char** array, int count);


// Memory utilities

void* xmalloc(size_t size);

void* xrealloc(void* ptr, size_t size);


// Error reporting

void print_error(const char* format, ...);

void print_system_error(const char* context);


// Safe wrappers for system calls with error checking

int safe_open(const char* pathname, int flags, mode_t mode);

pid_t safe_fork(void);

int safe_dup2(int oldfd, int newfd);


#endif // UTILS_H
```

`utils.c` - Complete Utility Implementation:

```c
#include "utils.h"
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

// String duplication with error checking
char* string_duplicate(const char* src) {
    if (src == NULL) return NULL;
    char* dst = malloc(strlen(src) + 1);
    if (dst == NULL) {
        print_system_error("malloc in string_duplicate");
        exit(EXIT_FAILURE);
    }
    strcpy(dst, src);
    return dst;
}

// Trim leading and trailing whitespace
char* string_trim(char* str) {
    if (str == NULL) return NULL;

    // Trim leading space
    while (*str == ' ' || *str == '\t' || *str == '\n') {
        str++;
    }

    if (*str == 0) return str;  // All spaces

    // Trim trailing space
```

```
    char* end = str + strlen(str) - 1;
    while (end > str && (*end == ' ' || *end == '\t' || *end == '\n')) {
        end--;
    }

    // Write new null terminator
    *(end + 1) = '\0';

    return str;
}

// Memory allocation with error checking
void* xmalloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr == NULL && size > 0) {
        print_system_error("malloc");
        exit(EXIT_FAILURE);
    }
    return ptr;
}

void* xrealloc(void* ptr, size_t size) {
    void* new_ptr = realloc(ptr, size);
    if (new_ptr == NULL && size > 0) {
        print_system_error("realloc");
        exit(EXIT_FAILURE);
    }
    return new_ptr;
}

// Error reporting functions
void print_error(const char* format, ...) {
    va_list args;
    va_start(args, format);
    vfprintf(stderr, format, args);
```

```c
    fprintf(stderr, "\n");
    va_end(args);
}

void print_system_error(const char* context) {
    fprintf(stderr, "%s: %s\n", context, strerror(errno));
}

// Safe wrapper for fork() with error checking
pid_t safe_fork(void) {
    pid_t pid = fork();
    if (pid < 0) {
        print_system_error("fork");
        exit(EXIT_FAILURE);
    }
    return pid;
}

// Safe wrapper for dup2() with error checking
int safe_dup2(int oldfd, int newfd) {
    int result = dup2(oldfd, newfd);
    if (result < 0) {
        print_system_error("dup2");
        exit(EXIT_FAILURE);
    }
    return result;
}

// Safe wrapper for open() with error checking
int safe_open(const char* pathname, int flags, mode_t mode) {
    int fd = open(pathname, flags, mode);
    if (fd < 0) {
        print_system_error(pathname);
        return -1;
    }
    return fd;
```

```
}
```

D. Core Logic Skeleton Code

`main.c` - Application Entry Point:

```c
#include "datamodel.h"

#include "repl.h"

#include "utils.h"

#include <stdio.h>

#include <stdlib.h>


int main(int argc, char** argv) {

    // TODO 1: Initialize the global shell state

    ShellState state;

    shell_state_init(&state);



    // TODO 2: Set up signal handlers for SIGINT, SIGTSTP, SIGCHLD

    //         (Implement in repl.c, but register them here)



    // TODO 3: Save the original terminal settings if doing job control

    //         (Milestone 6)



    // TODO 4: Enter the main REPL loop

    int exit_status = run_repl_loop(&state);



    // TODO 5: Clean up shell state before exiting

    shell_state_cleanup(&state);



    // TODO 6: Restore terminal settings if they were modified



    return exit_status;

}
```

`repl.h` - REPL Component Interface:

```c
#ifndef REPL_H

#define REPL_H


#include "datamodel.h"


// Main REPL loop function - runs the interactive shell session

int run_repl_loop(ShellState* state);


// Signal handler setup functions

void setup_signal_handlers(void);

void sigchld_handler(int sig);

void sigint_handler(int sig);

void sigtstp_handler(int sig);


// Job status reporting (called before displaying each prompt)

void job_control_check_completed_jobs(ShellState* state);


// Prompt display function

void display_prompt(void);


// Input reading function (with line editing support)

char* read_command_line(void);


#endif // REPL_H
```

`parser.h` - Parser Component Interface:

```c
#ifndef PARSER_H

#define PARSER_H


#include "datamodel.h"


// ParseResult structure: result of parsing (either AST or error)

typedef struct ParseResult {

    Command* ast;           // Abstract Syntax Tree (NULL on error)

    char* error_message;    // Error description (NULL on success)

    bool success;           // True if parsing succeeded

} ParseResult;


// Main parsing function: converts command line string to AST

ParseResult parse_command_line(char* line, ShellState* state);


// AST cleanup function: frees all memory associated with a command AST

void free_command_ast(Command* ast);


// Helper function for tokenization

char** tokenize_command_line(const char* line, int* token_count);


#endif // PARSER_H
```

`executor.h` - Executor Component Interface:

```c
#ifndef EXECUTOR_H

#define EXECUTER_H


#include "datamodel.h"


// Main execution function: executes a command AST

int execute_ast(Command* ast, ShellState* state, bool is_background);


// Built-in command test function (for diagnostics)

int builtin_test(int argc, char** argv, ShellState* state);


// Helper function to set up I/O redirections for a command

int setup_redirections(Command* cmd);


// Helper function to restore original file descriptors after execution

void restore_redirections(void);


// Function to find and execute external commands

int execute_external_command(Command* cmd);


#endif // EXECUTOR_H
```

`job_control.h` - Job Control Manager Interface:

```c
#ifndef JOB_CONTROL_H

#define JOB_CONTROL_H


#include "datamodel.h"


// Job table management functions

void job_table_init(JobTable* table);

void job_table_add(JobTable* table, pid_t pgid, const char* command,

                   Command* ast, JobState state);

void job_table_remove(JobTable* table, int job_id);

Job* job_table_find_by_pgid(JobTable* table, pid_t pgid);

void job_table_cleanup(JobTable* table);


// Job state management

void job_mark_as_stopped(JobTable* table, pid_t pgid);

void job_mark_as_running(JobTable* table, pid_t pgid);

void job_mark_as_done(JobTable* table, pid_t pgid);


// Terminal control functions

void give_terminal_to(pid_t pgid, int terminal_fd);

void take_back_terminal(int terminal_fd);


// Built-in job control commands (implemented in builtins/)

int builtin_jobs(int argc, char** argv, ShellState* state);

int builtin_fg(int argc, char** argv, ShellState* state);

int builtin_bg(int argc, char** argv, ShellState* state);


#endif // JOB_CONTROL_H
```

E. Language-Specific Hints for C

1. **Memory Management**: Always check `malloc` / `realloc` return values. Use the `xmalloc` and `xrealloc` wrappers provided in `utils.c` for automatic error handling.
2. **String Handling**: Remember that C strings are null-terminated. When building argument vectors for `execvp`, ensure the array ends with a `NULL` pointer.
3. **File Descriptors**: The golden rule: after `dup2(oldfd, newfd)`, immediately close `oldfd` unless you specifically need to keep it open elsewhere. File descriptor leaks are a common source of bugs in shells.
4. **Signal Safety**: Only async-signal-safe functions may be called from signal handlers. `write`, `_exit`, and `sigaction` are safe; `printf`, `malloc`, and most other stdio functions are not.

5. **Process Groups**: Always use `setpgid` to place child processes in process groups before starting them. This ensures proper signal delivery for pipelines.
6. **Error Handling**: Check system call return values consistently. When a system call fails, use `perror` or `strerror(errno)` to report what went wrong.

F. Milestone Checkpoint - Initial Architecture

After setting up the basic architecture with the files above, verify the structure compiles:

```bash
# In the gauntlet_shell directory

gcc -c utils.c -o utils.o

gcc -c main.c -o main.o

echo "Compilation should succeed (though linking will fail until components are implemented)"
```

Expected Outcome: The compilation should produce object files without errors. If you encounter "undefined reference" errors for functions like `shell_state_init` or `run_repl_loop`, that's expected—these will be implemented in subsequent milestones.

Signs of Correct Structure:

- All header files have proper include guards (`#ifndef HEADER_H`)
- `main.c` includes `datamodel.h` and `repl.h`
- The Makefile (when created) compiles each `.c` file to a `.o` file separately
- No circular dependencies in `#include` directives

Common Setup Issues:

- **Issue**: "Multiple definition" errors during linking.
- **Cause**: Function definitions in header files without `static` or `inline`.
- **Fix**: Ensure all function *definitions* are in `.c` files, only *declarations* in `.h` files.
- **Issue**: "Implicit declaration" warnings.
- **Cause**: Using functions before they're declared.
- **Fix**: Ensure all `.c` files include the necessary headers, and headers include each other as needed.

Data Model

> **Milestone(s):** All (1-6) – The data model is the fundamental representation of commands, jobs, and shell state that every component operates on. It's established in Milestone 1 (basic command parsing) and evolves through Milestone 4 (pipeline representation) and Milestones 5-6 (job control).

The data model defines the core data structures that translate ephemeral user input into persistent, actionable representations within the shell. These structures serve as the common language between components: the **Parser** produces them, the **Executor** operates on them, and the **Job Control Manager** tracks their process lifecycles. A well-designed data model provides the structural integrity needed to handle the shell's inherent complexity—piping multiple commands together, redirecting their I/O, and managing their execution concurrently in foreground and background.

Think of the data model as the **blueprint language of construction**. When an architect (the user) describes a building ("a three-story structure with connecting skybridges and underground parking"), the construction team needs precise technical drawings. The `Command` struct is the drawing for a single floor plan, `Redirection` specifies where utilities connect, the linked list of `Command` nodes represents the interconnected floors (pipeline), and the `Job` structure is the project management binder tracking the entire construction project's status, workers, and timeline.

The Command Structure

The `Command` structure represents a single executable unit in a pipeline—one "node" in what becomes an abstract syntax tree (AST). Each command corresponds roughly to one process that will be created via `fork()` and `exec()`. The critical insight is that a pipeline like `ls -l | grep foo | wc -l` is not a single command but a chain of three `Command` nodes linked together, each with its own arguments and potential I/O redirections.

> **Mental Model: Recipe Card for a Single Dish** Think of a `Command` struct as a recipe card for preparing one dish in a multi-course meal. The `argv` lists ingredients (the executable and arguments), `input_redir` and `output_redir` specify where to get ingredients and where to plate the dish (file redirections), and `next` points to the recipe card for the next dish in the meal sequence (the pipeline). The chef (executor) prepares each dish in order, passing the output of one dish as the input to the next through pipes.

Field	Type	Description
`argv`	`char**`	Array of argument strings, null-terminated. `argv[0]` is the command name (e.g., "ls"), subsequent elements are arguments (e.g., "-l", "/home"). This array is passed directly to `execvp()`.
`argc`	`int`	Count of arguments in `argv`. This is the length of the array (excluding the terminating `NULL`). Useful for iteration and memory management.
`input_redir`	`Redirection*`	Pointer to a `Redirection` struct describing input redirection (`<`, `<<`). `NULL` if no input redirection specified (default to stdin).
`output_redir`	`Redirection*`	Pointer to a `Redirection` struct describing output redirection (`>`, `>>`, `2>`). `NULL` if no output redirection specified (default to stdout/stderr).
`next`	`Command*`	Pointer to the next `Command` in a pipeline. `NULL` if this is the last command in the pipeline (or a standalone command). This forms a singly-linked list representing the pipeline flow from left to right.

The `Redirection` struct encapsulates the details of an I/O redirection operation. A command may have zero, one, or multiple redirections (e.g., `2>&1` would require two `Redirection` structs, though our simplified model handles standard cases). We define an enumeration `RedirType` to distinguish the redirection variants:

Enum Value	Description	Typical Shell Syntax
`REDIR_INPUT`	Redirect input from a file	`< file`
`REDIR_OUTPUT`	Redirect output to a file (truncate)	`> file`
`REDIR_APPEND`	Redirect output to a file (append)	`>> file`
`REDIR_ERROR`	Redirect stderr to a file	`2> file`

Field	Type	Description
`type`	`RedirType`	Enum value indicating the type of redirection (input, output, append, error).
`filename`	`char*`	The name of the file to redirect from/to. Dynamically allocated string.
`fd`	`int`	The target file descriptor for the redirection. For input: usually 0 (stdin). For output: 1 (stdout). For error: 2 (stderr). This allows flexibility for advanced redirections like `3>&1`.

Example: The command `grep "error" < log.txt > output.txt 2>&1` would be represented by one `Command` struct with:

- `argv = {"grep", "error", NULL}`, `argc = 2`
- `input_redir` pointing to a `Redirection` with `type=REDIR_INPUT`, `filename="log.txt"`, `fd=0`
- `output_redir` pointing to a `Redirection` with `type=REDIR_OUTPUT`, `filename="output.txt"`, `fd=1`
- A second `Redirection` for stderr (not shown in our simplified single-pointer model—advanced implementations might use a list)

ADR: Linked List vs. Array for Pipeline Representation

- **Context**: We need to represent a pipeline of commands (e.g., `cmd1 | cmd2 | cmd3`). The structure must support dynamic length (any number of commands) and preserve order.
- **Options Considered**:
  1. **Linked list of `Command` nodes**: Each `Command` has a `next` pointer to the following command.
  2. **Array of `Command*`**: Dynamically allocated array of pointers to `Command` structs.
  3. **Binary tree**: For complex command combinations with subshells or logical operators (beyond our scope).
- **Decision**: Linked list of `Command` nodes.
- **Rationale**:
  - **Natural fit for sequential flow**: A pipeline is inherently sequential; each command's output feeds directly into the next's input. A linked list mirrors this linear flow.
  - **Dynamic growth trivial**: Adding another pipe segment just requires allocating a new node and updating the tail's `next` pointer.
  - **Execution algorithm simplicity**: The executor can traverse the list with a simple `while (cmd != NULL)` loop, forking each node in sequence.
  - **Memory efficiency for sparse pipelines**: No need to preallocate or reallocate arrays; each command uses memory proportional to its complexity.
- **Consequences**:
  - **Linear traversal only**: To find the last command (needed for waiting on pipeline exit status), we must traverse the entire list. This is acceptable as pipelines are typically short.
  - **More pointer management**: Care must be taken to properly link and free all nodes.
  - **No random access**: Not needed for our use case.

Option	Pros	Cons	Chosen?
Linked List	Dynamic size, simple sequential traversal, memory efficient	Linear time to find tail, pointer management complexity	✅
Array	Random access, contiguous memory, simpler iteration	Fixed size or reallocation overhead, memory waste for short pipelines	❌
Binary Tree	Can represent complex nested structures	Overkill for linear pipelines, complex traversal algorithms	❌

The Job Structure

A **job** represents a logical unit of work that the shell tracks—typically a pipeline (multiple commands connected by pipes) or a single command. Jobs are managed by the shell when run in the background (`&`) or when suspended (`Ctrl+Z`). The `Job` struct tracks the process group ID, command string for display, current state, and the command AST that spawned it.

Mental Model: Restaurant Order Ticket Imagine a restaurant kitchen where each order ticket represents a job. The ticket contains: the table number (job ID), the list of dishes to prepare (command AST), the current status ("cooking", "ready", "served"), and the kitchen staff assigned (process group ID). The head chef (shell) can check all tickets, prioritize which to work on next (foreground vs. background), and update status as dishes complete.

Field	Type	Description
`id`	`int`	Unique job identifier (1, 2, 3...). Displayed by the `jobs` command. Incremented for each new job.
`pgid`	`pid_t`	Process group ID of the job. All processes in a pipeline belong to the same process group. Used to send signals to the entire job (e.g., `SIGINT` to cancel).
`command_string`	`char*`	The original command line string (e.g., `"ls -l
`state`	`JobState`	Current state of the job: `JOB_RUNNING`, `JOB_STOPPED`, `JOB_DONE`. Determines how the job appears in `jobs` output and what actions (`fg`, `bg`) are valid.
`command_list`	`Command*`	Pointer to the head of the command AST (linked list) that this job executes. This allows re-execution or inspection of the job's structure.

The `JobState` enumeration defines the possible states a job can be in:

Enum Value	Description	Display in `jobs`
`JOB_RUNNING`	Job is currently executing (in background)	`Running`
`JOB_STOPPED`	Job has been suspended (by `Ctrl+Z` or `SIGTSTP`)	`Stopped`
`JOB_DONE`	Job has terminated (exited or signaled) but not yet reaped	Not shown (or shown as `Done` briefly before removal)

The `JobTable` is a container that manages all active and recently completed jobs. It provides a fixed-capacity store with basic operations.

Field	Type	Description
`jobs`	`Job*`	Array of pointers to `Job` structs. Fixed capacity of `MAX_JOBS` (typically 100).
`capacity`	`int`	Maximum number of jobs the table can hold (`MAX_JOBS`).
`count`	`int`	Current number of jobs in the table (active + done).

Job Lifecycle State Machine: A job transitions through states based on user actions and process events:

Current State	Event	Next State	Actions Taken
(none)	User runs command without `&`	(Foreground, not in job table)	Process group created, shell waits.
(none)	User runs command with `&`	`JOB_RUNNING`	Job added to table with `JOB_RUNNING` state, shell continues.
`JOB_RUNNING`	Process receives `SIGTSTP` (Ctrl+Z)	`JOB_STOPPED`	Shell updates state, reports job stopped.
`JOB_RUNNING`	Process exits normally	`JOB_DONE`	Shell updates state on next `SIGCHLD`.
`JOB_STOPPED`	User runs `bg %jobid`	`JOB_RUNNING`	Shell sends `SIGCONT` to process group, updates state.
`JOB_STOPPED`	User runs `fg %jobid`	(Foreground)	Shell moves job to foreground, waits, removes from table on completion.
`JOB_DONE`	Shell reaps process (handles `SIGCHLD`)	*(removed)*	Job removed from table, memory freed.

ADR: Storing Command AST vs. Re-parsing for Job Control

- **Context**: When a user runs `fg %1` to resume a suspended job, the shell must know which command(s) to resume. We need to store either the original command string or the parsed AST.
- **Options Considered**:
  1. **Store only command string**: Keep the original input string; re-parse when needed.
  2. **Store parsed AST**: Keep the fully parsed `Command*` linked list.
  3. **Store both**: Keep both for different purposes.
- **Decision**: Store parsed AST in `command_list`.
- **Rationale**:
  - **Performance**: Re-parsing on `fg` is unnecessary overhead for a simple operation.
  - **Correctness**: The parsed AST is the authoritative representation; re-parsing might produce different results if parsing logic changes.
  - **Simplicity**: The executor already expects an AST; we can pass the stored AST directly.
  - **Memory cost acceptable**: AST memory footprint is small relative to process overhead.
- **Consequences**:
  - Must carefully manage AST memory lifecycle (free when job removed).
  - AST must be "re-executable"—cannot contain transient state like open file descriptors.
  - Slightly more complex job creation (must copy/retain AST).

Option	Pros	Cons	Chosen?
Store only command string	Less memory, simpler job struct	Re-parsing overhead, potential parse differences	❌
Store parsed AST	Fast resume, authoritative representation	Memory management complexity, AST must be persistent	✅
Store both	Flexibility, human-readable string	Redundant storage, synchronization complexity	❌

Shell Global State

The `ShellState` struct encapsulates all global runtime state of the shell. This includes the job table, terminal configuration, current foreground process group, and the last exit status. Using a single state struct promotes cleaner code organization, enables potential future multi-shell instances, and simplifies function signatures (many functions take a `ShellState*`).

Mental Model: The Shell's Dashboard and Control Panel Imagine the `ShellState` as the dashboard of a car combined with the air traffic control tower's main console. It shows: active flights (job table), which flight has radio contact (foreground process group), the runway currently in use (terminal file descriptor), and the last system status alert (last exit status). All controllers (shell functions) reference this central dashboard to coordinate actions.

Field	Type	Description
`job_table`	`JobTable`	The table tracking all background and suspended jobs. Initialized empty.
`foreground_pgid`	`pid_t`	Process group ID of the currently foreground job (or 0 if none). Used to correctly deliver terminal signals (`SIGINT` , `SIGTSTP`) to the right process group.
`terminal_fd`	`int`	File descriptor of the controlling terminal (usually 0 for stdin). Stored to restore terminal settings and manage process groups with `tcsetpgrp()` .
`last_exit_status`	`int`	Exit status of the last executed foreground command (or pipeline). Used for `$?` expansion (in extended implementations) and reporting.

The relationships between these structures form the complete data model:


ShellState
+ job_table JobTable
+ foreground_pgid pid_t
+ terminal_fd int
+ original_termios struct termios
+ is_interactive bool
JobState
+ FOREGROUND 0
+ BACKGROUND 1
+ STOPPED 2
RedirType
+ INPUT 0
+ OUTPUT 1
+ APPEND 2
+ HEREDOC 3
contains
JobTable
+ jobs Job[]
+ next_job_id int
jobs
Job
+ job_id int
+ pgid pid_t
+ state JobState
+ command_line string
+ first_command Command*
first_command
Command
+ argv string[]
+ argc int
+ redirections Redirection[]
+ next Command*
next
redirections
Redirection

Key Invariants and Constraints:

1. **Ownership**: `ShellState` owns the `JobTable`, which owns an array of pointers to `Job` structs. Each `Job` owns its `command_string` and `command_list` AST, which owns its `argv` array and `Redirection` structs.
2. **Nullability**: Pointers can be `NULL` when not applicable (e.g., `input_redir` when no input redirection).
3. **Lifetimes**:
   - `ShellState` persists for the entire shell session.
   - `Job` entries live from job creation (`&` or suspension) until reaped after completion.
   - `Command` ASTs live for the duration of parsing and execution (for foreground commands) or until job removal (for background jobs).
4. **Thread Safety**: Not required—Unix shells are single-threaded (except signal handlers which must be careful).

Example Scenario Walkthrough: When the user inputs `ls -l | grep foo > output.txt &`:

1. **Parser** creates a linked list of two `Command` nodes:
   - Node 1: `argv = {"ls", "-l", NULL}`, `input_redir = NULL`, `output_redir = NULL`, `next` points to Node 2
   - Node 2: `argv = {"grep", "foo", NULL}`, `input_redir = NULL`, `output_redir` points to `Redirection{type=REDIR_OUTPUT, filename="output.txt", fd=1}`, `next = NULL`
2. **Executor** creates a job with a new process group ID, stores the command string and AST pointer.
3. **Job Control Manager** adds the job to `ShellState.job_table` with state `JOB_RUNNING`.
4. The `foreground_pgid` remains 0 (no foreground job).
5. When the pipeline completes, `SIGCHLD` handler updates the job state to `JOB_DONE`.
6. On next prompt display, `job_control_check_completed_jobs()` reports completion and removes the job from the table.

Common Pitfalls

⚠️ Pitfall: Not Deep-Copying Strings for Persistent Storage

- **Description**: Storing pointers to the original input buffer (e.g., from `fgets()`) in `command_string` or `argv` elements.
- **Why it's wrong**: The input buffer gets overwritten on the next command line read, corrupting the stored command string. For background jobs, this leads to garbage display in `jobs` output or crashes.
- **Fix**: Always use `strdup()` or equivalent to allocate and copy strings that need to persist beyond the current parsing cycle.

⚠️ Pitfall: Forgetting to NULL-Terminate the argv Array

- **Description**: Building the `argv` array but not ensuring the last element is `NULL`.
- **Why it's wrong**: `execvp()` expects a NULL-terminated array; missing termination causes it to read past allocated memory, leading to crashes or executing garbage as arguments.
- **Fix**: Explicitly set `argv[argc] = NULL` after filling all arguments.

⚠️ Pitfall: Not Handling Redirection Linked List Properly

- **Description**: Using single `input_redir` and `output_redir` pointers when a command might have multiple redirections (e.g., `2>&1`).

- **Why it's wrong**: Only the last redirection of each type takes effect; earlier ones are lost. This breaks complex redirection patterns.
- **Fix (simplified)**: For basic shell requirements, handle only one redirection per direction and ensure correct precedence (redirections are processed left-to-right). For advanced implementations, use a linked list of redirections per command.

⚠️ Pitfall: Job ID Reuse and Race Conditions

- **Description**: Reusing job IDs immediately after job removal, or checking job state without considering `SIGCHLD` might change it.
- **Why it's wrong**: If a user references `%1` and the job table has changed between reference and use, the wrong job might be affected. Signal handlers can modify job state asynchronously.
- **Fix**: Use atomic operations or careful sequencing: check job exists, get its `pgid`, then send signal. In signal handlers, only set flags or use async-signal-safe functions; do main processing in the main loop.

⚠️ Pitfall: Memory Leaks in Complex ASTs

- **Description**: Freeing the `Command` struct but not its owned `argv` strings, redirection structs, or filename strings.
- **Why it's wrong**: Memory leaks accumulate over shell session, especially with many background jobs or pipelines.
- **Fix**: Implement comprehensive cleanup function `free_command_ast()` that recursively frees all allocated memory in the correct order (strings → argv array → redirection structs → command nodes).

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Dynamic String Storage	`strdup()` / `free()`	String pool allocator for reduced fragmentation
Command AST Allocation	Individual `malloc()` per node	Arena allocator for entire parse tree
Job Table Storage	Fixed array of pointers	Hash table keyed by job ID for O(1) lookup
Redirection Representation	Single pointer per direction	Linked list of redirections per command

B. Recommended File/Module Structure:

```
gauntlet_shell/
├── src/
│   ├── main.c                  # Entry point, initializes ShellState, calls run_repl_loop()
│   ├── shell_state.h           # Declarations of ShellState, JobTable, Job, Command, Redirection
│   ├── shell_state.c           # Definitions of shell_state_init(), job_table_* functions
│   ├── parser.h                # parse_command_line(), tokenization functions
│   ├── parser.c                # Parser implementation
│   ├── executor.h              # execute_ast(), built-in handlers
│   ├── executor.c              # Executor implementation
│   ├── job_control.h           # Job control functions, signal handlers
│   └── job_control.c           # Job control implementation
├── include/                    # Additional headers if needed
└── Makefile                    # Build configuration
```

C. Core Data Structure Definitions (Complete Starter Code):

C

```c
/* shell_state.h */
#ifndef SHELL_STATE_H
#define SHELL_STATE_H

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

#define MAX_JOBS 100
#define MAX_LINE_LEN 4096

/* Redirection types */
typedef enum {
    REDIR_INPUT,    /* <  */
    REDIR_OUTPUT,   /* >  */
    REDIR_APPEND,   /* >> */
    REDIR_ERROR     /* 2> */
} RedirType;

/* Redirection structure */
typedef struct Redirection {
    RedirType type;
    char* filename;
    int fd;                /* Target file descriptor (0, 1, 2, etc.) */
    struct Redirection* next;  /* For multiple redirections (advanced) */
} Redirection;

/* Command structure - node in a pipeline */
typedef struct Command {
    char** argv;           /* Array of arguments, NULL-terminated */
    int argc;              /* Number of arguments */
    Redirection* input_redir;   /* Input redirection, NULL if none */
    Redirection* output_redir;  /* Output redirection, NULL if none */
    struct Command* next;  /* Next command in pipeline, NULL if last */
} Command;

/* Job states */
typedef enum {
```

```
    JOB_RUNNING,
    JOB_STOPPED,
    JOB_DONE
} JobState;

/* Job structure */
typedef struct Job {
    int id;                 /* Job ID (1, 2, 3, ...) */
    pid_t pgid;             /* Process group ID */
    char* command_string;   /* Original command line */
    JobState state;         /* Current state */
    Command* command_list;  /* Parsed command AST */
} Job;

/* Job table */
typedef struct JobTable {
    Job* jobs[MAX_JOBS];    /* Array of job pointers */
    int capacity;           /* Always MAX_JOBS */
    int count;              /* Current number of jobs */
    int next_job_id;        /* Next job ID to assign */
} JobTable;

/* Shell global state */
typedef struct ShellState {
    JobTable job_table;
    pid_t foreground_pgid;  /* Current foreground process group (0 if none) */
    int terminal_fd;        /* Controlling terminal file descriptor */
    int last_exit_status;   /* Exit status of last foreground command */
    int interactive;        /* Is shell interactive? (for prompt display) */
} ShellState;

/* Parser result */
typedef struct ParseResult {
    Command* ast;           /* Abstract syntax tree (linked list of Commands) */
    char* error_message;    /* Error description if parsing failed */
    int success;            /* 1 on success, 0 on error */
```

```c
} ParseResult;


/* Function prototypes */

void shell_state_init(ShellState* state);

void job_table_init(JobTable* table);

void free_command_ast(Command* cmd);

ParseResult parse_command_line(char* line, ShellState* state);

int execute_ast(Command* ast, ShellState* state);

void job_control_check_completed_jobs(ShellState* state);


#endif /* SHELL_STATE_H */
```

```c
/* shell_state.c - Core data structure implementations */

#include "shell_state.h"

#include <stdlib.h>

#include <string.h>

#include <stdio.h>


/* Initialize shell global state */

void shell_state_init(ShellState* state) {

    if (!state) return;



    job_table_init(&state->job_table);

    state->foreground_pgid = 0;

    state->terminal_fd = STDIN_FILENO;  /* Default to stdin */

    state->last_exit_status = 0;

    state->interactive = isatty(state->terminal_fd);

}


/* Initialize job table */

void job_table_init(JobTable* table) {

    if (!table) return;



    for (int i = 0; i < MAX_JOBS; i++) {

        table->jobs[i] = NULL;

    }

    table->capacity = MAX_JOBS;

    table->count = 0;

    table->next_job_id = 1;  /* Start job IDs from 1 */

}


/* Free all memory associated with a command AST */

void free_command_ast(Command* cmd) {

    // TODO 1: Implement recursive traversal of the command linked list

    // TODO 2: For each command node:

    //   a. Free each string in argv array (if not NULL)

    //   b. Free the argv array itself
```

```
    //   c. Free redirection structs (input_redir and output_redir)
    //      - Free filename string in each redirection
    //      - Free the redirection struct itself
    //   d. Free the command node
    // TODO 3: Handle both single commands and pipelines (multiple nodes)
    // Hint: Use a while loop, store next pointer before freeing current node
}

/* Helper: Free a single command node (without recursion) */
static void free_single_command(Command* cmd) {
    if (!cmd) return;

    // TODO 1: Free argv strings and array
    // TODO 2: Free input redirection if present
    // TODO 3: Free output redirection if present
    // TODO 4: Free the command struct itself
}

/* Add a job to the job table */
void job_table_add(JobTable* table, pid_t pgid, const char* command_str,
                   Command* cmd_ast, JobState state) {
    // TODO 1: Check if table is full (count >= capacity)
    // TODO 2: Find empty slot in jobs array (NULL entry)
    // TODO 3: Allocate new Job struct
    // TODO 4: Assign fields: id (from next_job_id), pgid, state
    // TODO 5: Duplicate command_str for command_string field
    // TODO 6: Assign command_list (caller retains ownership or copies?)
    // TODO 7: Insert into empty slot, increment count and next_job_id
    // TODO 8: Handle wraparound of job IDs after many jobs
}

/* Find job by job ID */
Job* job_table_find_by_id(JobTable* table, int job_id) {
    // TODO 1: Iterate through jobs array
    // TODO 2: For each non-NULL entry, check if id matches
```

```
    // TODO 3: Return pointer if found, NULL otherwise
    return NULL;
}

/* Remove job from table by index */
void job_table_remove_at(JobTable* table, int index) {
    // TODO 1: Check index bounds
    // TODO 2: Free the job's command_string
    // TODO 3: Free the job's command_list AST (call free_command_ast)
    // TODO 4: Free the Job struct itself
    // TODO 5: Set entry to NULL, decrement count
}
```

D. Language-Specific Hints (C):

1. **String Management**: Always use `strdup()` to copy strings for persistent storage, and pair with `free()`. Check `strdup()` returns non-NULL.
2. **NULL-Termination**: For `argv` arrays, allocate `argc + 1` elements and set the last to `NULL`.
3. **Deep Copying**: When storing the command AST in a job, you may need to deep-copy the entire AST if the parser reuses buffers. Alternatively, transfer ownership (parser yields AST, job takes ownership).
4. **Signal Safety**: Never call `malloc()`, `free()`, or `printf()` in signal handlers. In `SIGCHLD` handler, just set a flag; process job table changes in the main loop.
5. **Error Recovery**: In `free_command_ast()`, be defensive against partially-initialized structs (NULL pointers).

E. Milestone Checkpoint for Data Model:

After implementing the data structures (Milestone 1, but evolves through later milestones):

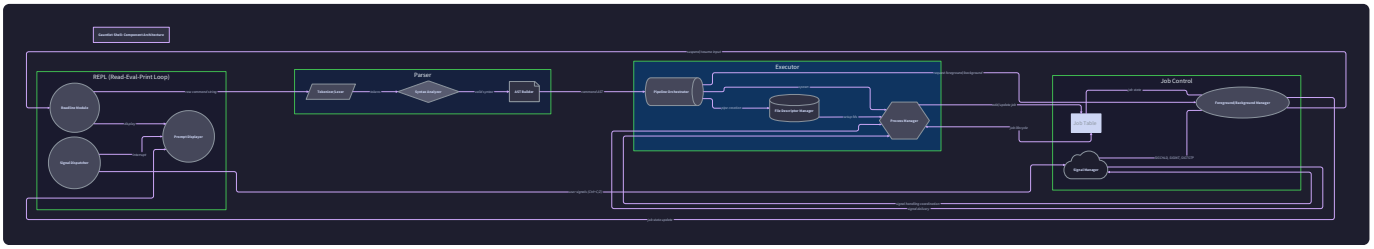
1. **Compilation Test**: `gcc -c src/shell_state.c -o shell_state.o` should succeed without warnings.
2. **Memory Test**: Write a small test program that:
   - Creates a `ShellState`, initializes it
   - Manually builds a simple `Command` AST (e.g., `ls -l`)
   - Adds a job to the table
   - Frees everything using the provided functions
   - Run with valgrind: `valgrind ./test_shell_state` should report "no leaks are possible".
3. **Invariant Check**: The job table should never have more than `MAX_JOBS` entries, and job IDs should be unique and increasing.

F. Debugging Tips for Data Model Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Shell crashes when running background jobs	`command_string` points to reused buffer	Add `printf("Job cmd: %s\n", job->command_string)` after job creation and before next prompt	Use `strdup()` to copy the input string
`execvp()` fails with bad address	`argv` not NULL-terminated	Print `argv` array with sentinel: `for(i=0; argv[i]; i++) printf("[%d]='%s'\n", i, argv[i])`	Explicitly set `argv[argc] = NULL`
Memory leaks grow with each command	Not freeing AST after execution	Use valgrind: `valgrind --leak-check=full ./shell`	Implement and call `free_command_ast()` for foreground commands too
`jobs` shows wrong command	Job's `command_string` corrupted	Store original string immediately after parsing, before any further processing	Copy string at parse time, not after potential modifications
Job IDs jump or repeat	Improper `next_job_id` management	Print job ID assignment sequence	Reset `next_job_id` only on overflow, not when jobs are removed

Component Design: The Read-Eval-Print Loop (REPL)

> **Milestone(s):** 1, 5, 6 – This component serves as the shell's central nervous system, responsible for orchestrating the interactive experience. It provides the primary user interface, coordinates background job management, and ensures signal handling integrates seamlessly with the command execution flow. While established in Milestone 1 for basic command execution, its complexity grows significantly with background job reporting (Milestone 5) and job control signal coordination (Milestone 6).



Mental Model: The Conductor of an Orchestra

Think of the **REPL** as the conductor of a complex orchestra. The conductor doesn't play any instruments themselves, but they orchestrate the entire performance:

- **Reading the Score (Input)**: First, the conductor looks at the musical score—the written instructions. In the shell, this is reading the command line typed by the user.
- **Directing Sections (Parsing & Execution)**: The conductor signals different sections of the orchestra (strings, woodwinds, brass) when to play. The REPL signals the **Parser** to translate the command text into a structured plan (the AST), then signals the **Executor** to bring that plan to life by spawning processes.
- **Managing Tempo and Dynamics (Job Control)**: During the performance, the conductor controls the pace (foreground jobs that block) and can direct certain sections to play softly in the background (background jobs). They can also pause the entire orchestra (Ctrl+Z) or resume a paused section (`fg` / `bg`).
- **Listening for Cues (Signals)**: A good conductor listens for cues from the orchestra, like if a section finishes early. The REPL "listens" for `SIGCHLD` signals from completed child processes to clean them up and report their status.

The conductor's primary goal is to maintain a smooth, coordinated performance where each section (component) knows exactly when to act, preventing cacophony (race conditions, zombie processes) and ensuring the audience (user) experiences a seamless performance.

Interface and Responsibilities

The REPL is implemented as a single primary function, `run_repl_loop`, which acts as the shell's entry point after initialization. It operates on the global `ShellState` and coordinates all other components.

Method Signature	Returns	Description
`run_repl_loop(ShellState* state)`	`int`	The main entry point for the interactive shell. Runs the Read-Eval-Print Loop until termination. Returns the shell's final exit status (typically 0 for normal exit, non-zero if `exit` was called with an error code).

Key Responsibilities:

1. **User Interface Management**: Displaying the prompt (e.g., `$ `), reading user input lines (handling EOF/Ctrl+D), and providing visual feedback.
2. **Orchestration**: Invoking the `parse_command_line` function to transform raw input into an AST, then passing the AST to `execute_ast` to run it.
3. **Background Job Stewardship**: Before displaying each new prompt, calling `job_control_check_completed_jobs` to report on and clean up any background jobs that have finished since the last command.
4. **Signal Handling Coordination**: Installing and managing global signal handlers (for `SIGCHLD`, `SIGINT`, `SIGTSTP`) and ensuring the main loop's blocking `read` operation can be safely interrupted.
5. **Lifecycle Management**: Properly cleaning up (freeing) the AST after each command execution and ensuring the shell terminates cleanly upon the `exit` built-in command.

Internal Behavior and Algorithm

The REPL follows a classic loop pattern, but with critical additions for robustness in a concurrent, signal-driven environment. The following numbered algorithm describes one complete iteration of the loop, starting from a fresh prompt.

Algorithm: Main REPL Iteration

1. **Check and Report Completed Background Jobs**: Invoke `job_control_check_completed_jobs(state)`. This function scans the `state->job_table` for any jobs in the `JOB_DONE` state, prints a notification to the user (e.g., `[1]+ Done ls -l`), and removes them from the table. This ensures the user is informed of completed work before being presented with a new prompt.
2. **Display Prompt**: Print the shell prompt (e.g., `$ `) to `stdout`. For an interactive shell (`state->interactive` is true), the prompt should be flushed to ensure it appears immediately.
3. **Read a Line of Input**: Read a line from `stdin` into a buffer of size `MAX_LINE_LEN`. This step must be **signal-aware**.
   - **On Success**: Proceed to step 4.
   - **On End-of-File (EOF / Ctrl+D)**: The `read` or `fgets` call will return `NULL`. This is the user's way of exiting the shell. Break out of the loop and proceed to step 7 (Cleanup & Exit).
   - **On Interruption by Signal**: If the `read` is interrupted by a signal (e.g., `SIGCHLD` from a background job finishing), the system call will fail with `errno` set to `EINTR`. The loop should **not** break. Instead, it should go back to step 1. This is crucial because a `SIGCHLD` handler may have changed the job table, and the user should see an updated prompt and job status.
4. **Parse Input**: Pass the input line to `parse_command_line(input_line, state)`. This function returns a `ParseResult` struct.
   - **On Parse Error**: If `ParseResult.success` is `false`, print the `error_message` to `stderr`, free any partial structures in the result, and loop back to step 1.
   - **On Empty Line**: If the input line is empty or contains only whitespace, the parser should return a successful but `NULL` AST. The loop should simply go back to step 1.
   - **On Success**: Proceed with the `ParseResult.ast`.
5. **Execute Command**: Pass the AST (`ParseResult.ast`) to `execute_ast(ast, state)`. This function is responsible for:
   - Determining if the command is a built-in (like `cd`, `exit`) or external.

- For built-ins, executing them directly in the parent (shell) process.
- For external commands/pipelines, forking child processes, setting up I/O redirections and pipes, and managing process groups.
- For **foreground** jobs, the parent (shell) will call `waitpid` to block until the entire job completes.
- For **background** jobs (indicated by a trailing `&`), the parent will *not* wait. Instead, it will add the job's process group ID and command string to the `state->job_table` with state `JOB_RUNNING` and return immediately.
- The function returns the exit status of the last command in the pipeline (or the built-in's status).

6. **Cleanup and Loop**: Call `free_command_ast(ParseResult.ast)` to release all memory associated with the parsed command tree. Store the returned exit status in `state->last_exit_status` (which can be used by the `$?` variable in future extensions). Loop back to step 1.
7. **Cleanup & Exit (Loop Termination)**: When the loop terminates (due to `exit` command or EOF), perform comprehensive cleanup: free the entire `state->job_table`, reset terminal attributes if job control was active, and return the final exit status to the operating system (which will terminate the shell process).

ADR: Signal Handling Strategy in the REPL

Decision: Synchronous Signal Handling via `pselect` and Self-Pipe

- **Context**: The REPL's main loop blocks on `read()` for user input. Asynchronous signals (`SIGCHLD` from child termination, `SIGINT` / `SIGTSTP` from terminal) must be handled promptly without causing race conditions or leaving zombie processes. Traditional `signal()` or `sigaction()` handlers that run asynchronously at any point in the program's execution can corrupt global data structures if they interrupt non-reentrant functions (like `malloc` / `free` or our job table manipulation).
- **Options Considered**:
  1. **Traditional Async Handlers with `sigaction`** : Install handlers that run when the signal is delivered. Requires all code interacting with global state (job table, memory allocator) to be signal-safe, which is complex and error-prone.
  2. **Signal Masking and `pselect`** : Block signals in the main thread, use `pselect()` (or `ppoll()`) to atomically unblock them while waiting for input. Signals are only processed when the system call returns, making their handling synchronous and safe.
  3. **Self-Pipe Trick**: Write a byte to a pipe from within a minimal signal handler. The main loop `select()` s on this pipe alongside `stdin` . This moves signal handling back into the main event loop.
- **Decision**: Use a hybrid approach: **Block all signals in the main thread, and use `pselect()` to wait for input**. This is the POSIX-recommended method for safe signal handling in event loops. For `SIGCHLD` , we couple this with a flag-based deferred processing strategy.
- **Rationale**:
  - `pselect` provides atomicity between unblocking signals and waiting, preventing race conditions where a signal arrives just before the `select` call.
  - By handling signals only when `pselect` returns (either due to input or a signal), we ensure the main loop is at a known, safe point—never inside a complex, non-reentrant operation on the job table or memory allocator.
  - It simplifies handler code: the signal handler itself can be a trivial function that does nothing, or just sets a global `volatile sig_atomic_t` flag. The main loop checks these flags after `pselect` returns.
  - This approach is portable, well-documented, and aligns with best practices for robust system daemons and servers.
- **Consequences**:
  - The implementation requires careful management of the signal mask using `sigprocmask()` .
  - `SIGCHLD` processing (reaping zombies, updating job status) is slightly delayed until the next iteration of the loop. This is imperceptible to the user and is a safe trade-off.
  - Terminal signals (`SIGINT` , `SIGTSTP`) for the *shell itself* (e.g., Ctrl+C when no foreground job is running) are also handled synchronously, allowing for clean shell termination or job suspension.

Option	Pros	Cons	Chosen?
Traditional Async Handlers	Immediate response to signals.	High risk of data corruption; requires all code to be reentrant or use async-signal-safe functions only. Very difficult to implement correctly.	❌
Signal Masking + `pselect`	Signals handled synchronously at safe points; eliminates reentrancy concerns; POSIX standard.	Slight delay in signal processing; requires understanding of signal masks.	✅
Self-Pipe Trick	Moves handling to main loop; works with plain `select()`.	Requires creating/managing a pipe; more complex setup; handler still has to do a non-async-signal-safe `write()`.	❌

Common Pitfalls

⚠️ Pitfall: Blocking Read That Can't Be Interrupted

- **Description**: Using a simple `fgets()` or `read()` call without considering signals. If a `SIGCHLD` arrives while blocked, the default behavior is to interrupt the call with `EINTR`, but if not handled, the shell might crash or miss the signal entirely.
- **Why It's Wrong**: Background jobs finishing won't be reaped promptly, leading to zombie process accumulation. The user also won't see job completion notifications until they press Enter.
- **Fix**: Use `pselect()` as described in the ADR. Always check the return value of `read()` / `fgets()` and if it fails with `errno == EINTR`, loop back to re-read input (after checking for completed jobs).

⚠️ Pitfall: Signal Handler Modifying Global Data

- **Description**: Writing a `SIGCHLD` handler that directly calls `waitpid()`, updates the `job_table`, and calls `printf()` to report job completion.
- **Why It's Wrong**: `printf()` and `malloc()` / `free()` (used by the job table) are **not** async-signal-safe. The handler could be called while the main loop is in the middle of those same functions, causing deadlock or memory corruption.
- **Fix**: The signal handler should be minimal. The recommended pattern is to set a global flag (e.g., `volatile sig_atomic_t got_sigchld = 1;`). The main loop, after `pselect` returns, checks this flag and calls a **non-signal-handler** function (like `job_control_check_completed_jobs`) to do the actual work.

⚠️ Pitfall: Not Checking for Completed Jobs Before Each Prompt

- **Description**: Only checking for completed background jobs *after* a foreground command finishes, or in response to `SIGCHLD`.
- **Why It's Wrong**: If a background job finishes while the user is idle (not typing), the shell won't report it until the user runs another command. This violates user expectations (bash reports completed jobs just before displaying the next prompt).
- **Fix**: Integrate `job_control_check_completed_jobs(state)` as the **first step** inside the REPL loop, before printing the prompt. This guarantees timely reporting.

⚠️ Pitfall: Memory Leak in Parse-Error Path

- **Description**: When `parse_command_line` returns an error, the `ParseResult` may contain partially allocated structures (like a half-built AST). Failing to free these before looping again.
- **Why It's Wrong**: The shell slowly leaks memory with each malformed command, eventually exhausting available memory.
- **Fix**: The `parse_command_line` function must clean up after itself on error, OR the REPL must call a cleanup function (like `free_command_ast`) on the returned `ParseResult.ast` even when `success` is false (if the function contract allows it). The cleaner design is for the parser to handle its own cleanup on failure.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Input Reading	`fgets()` with `pselect()` for signal safety. Straightforward and portable.	Using `readline()` or `libedit` for history, line editing, and tab completion. Much more user-friendly but adds external dependencies.
Signal Safety	Minimal handlers setting global flags. Main loop processes flags.	Using `signalfd()` on Linux to turn signals into file descriptors that can be `select()`ed on. Very elegant but less portable.

B. Recommended File/Module Structure

The REPL logic is central to the shell and should reside in the main source file. Helper functions for signal setup and safe input reading can be placed in a separate module.

```
gauntlet_shell/
├── src/
│   ├── main.c              ← Entry point: initializes ShellState, calls run_repl_loop.
│   ├── repl.c              ← Contains run_repl_loop and signal handling setup functions.
│   ├── repl.h              ← Declares run_repl_loop and related signal flags.
│   ├── parser.c            ← parse_command_line implementation.
│   ├── executor.c          ← execute_ast and built-in commands.
│   ├── job_control.c       ← job_control_check_completed_jobs, job table operations.
│   └── common/
│       ├── shell_state.c   ← shell_state_init and cleanup functions.
│       └── utils.c         ← String utilities, safe memory allocation wrappers.
├── include/                (Optional, for a more structured project)
│   └── gauntlet/
│       ├── repl.h
│       ├── parser.h
│       └── ...
└── Makefile
```

C. Infrastructure Starter Code

Here is a complete, working signal-safe input reading module. This is prerequisite infrastructure that learners can copy directly.

```c
/* repl_helpers.c - Safe, signal-aware input reading for the REPL */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <signal.h>

#include <sys/select.h>


#define MAX_LINE_LEN 4096


/* Global signal flags. Use 'sig_atomic_t' as it's the only type guaranteed

   to be safe to read/write from both a signal handler and main code. */

volatile sig_atomic_t g_got_sigchld = 0;


/* Trivial signal handler that only sets a flag.

   This function is async-signal-safe. */

void handle_sigchld(int sig) {

    (void)sig; /* Unused parameter */

    g_got_sigchld = 1;

}


/* Reads a line from stdin with protection from signal interruption.

   Returns:

     On success: Pointer to a static buffer containing the line (without newline).

     On EOF: NULL

     On error: NULL (errno is set)

   This function uses pselect to wait for input on stdin only when

   no signals are pending. If a SIGCHLD arrives, pselect returns early

   and the function returns an empty string (caller should check flag).

*/

char* read_line_safe(void) {

    static char buffer[MAX_LINE_LEN];

    fd_set readfds;

    int n;

    sigset_t empty_mask;
```

```
    /* Create an empty signal mask for pselect */
    sigemptyset(&empty_mask);

    while (1) {
        /* Check the signal flag before potentially blocking */
        if (g_got_sigchld) {
            /* Signal arrived. Clear flag and return empty to let main loop
               handle the signal. */
            g_got_sigchld = 0;
            buffer[0] = '\0';
            return buffer;
        }

        FD_ZERO(&readfds);
        FD_SET(STDIN_FILENO, &readfds);

        /* Block on stdin, but allow signals to interrupt.
           The first argument is nfds = highest fd + 1 */
        n = pselect(STDIN_FILENO + 1, &readfds, NULL, NULL, NULL, &empty_mask);

        if (n == -1) {
            if (errno == EINTR) {
                /* Interrupted by a signal (not caught by our mask).
                   Loop again to check g_got_sigchld. */
                continue;
            }
            /* Real error */
            return NULL;
        }

        if (n > 0 && FD_ISSET(STDIN_FILENO, &readfds)) {
            /* Data is available on stdin */
            if (fgets(buffer, MAX_LINE_LEN, stdin) == NULL) {
                /* EOF or error */
                return NULL;
            }
```

```
            /* Remove trailing newline, if present */
            size_t len = strlen(buffer);
            if (len > 0 && buffer[len - 1] == '\n') {
                buffer[len - 1] = '\0';
            }
            return buffer;
        }
    }
}
```

D. Core Logic Skeleton Code

This is the core REPL loop that learners must implement, integrating the safe input reader and other components.

```c
/* repl.c - Main Read-Eval-Print Loop implementation */

#include "repl.h"

#include "parser.h"

#include "executor.h"

#include "job_control.h"

#include "common/shell_state.h"

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>


/* External declaration of the signal flag defined in repl_helpers.c */

extern volatile sig_atomic_t g_got_sigchld;


/**

 * run_repl_loop - The main interactive loop of the shell.

 * @state: Pointer to the initialized global shell state.

 *

 * Returns: The exit code of the shell (0 on normal exit, other on error/exit cmd).

 *

 * This function implements the core Read-Eval-Print Loop:

 * 1. Check and report completed background jobs.

 * 2. Display prompt.

 * 3. Read a line of input (safely, allowing signal interruption).

 * 4. Parse the line into a command AST.

 * 5. Execute the AST (built-in or external, foreground or background).

 * 6. Clean up and repeat.

 */

int run_repl_loop(ShellState *state) {

    char *input_line;

    int exit_status = 0;

    int is_interactive = isatty(STDIN_FILENO);


    /* TODO 1: Set up signal handlers.

       Use sigaction to install handle_sigchld for SIGCHLD.
```

```
       For SIGINT and SIGTSTP in an interactive shell, you might want
       to ignore them initially (they will be forwarded to foreground jobs).
    */

    while (1) {
        /* TODO 2: Check for and report completed background jobs.
           Call job_control_check_completed_jobs(state).
           This should print notifications like "[1]+ Done ls -l" for any
           jobs that have finished since the last check.
        */

        /* TODO 3: Display the prompt for interactive shells.
           if (is_interactive) {
               printf("$ ");
               fflush(stdout);
           }
        */

        /* TODO 4: Read a line of input using the safe function.
           input_line = read_line_safe();
           Handle EOF (NULL return) by breaking the loop.
           If the returned line is empty string ("") but not NULL, it means
           a SIGCHLD was caught. Loop back to step 2 (check jobs) to handle it.
        */

        /* TODO 5: Parse the input line.
           ParseResult result = parse_command_line(input_line, state);
           if (!result.success) {
               fprintf(stderr, "Parse error: %s\n", result.error_message);
               // TODO: Free any partial AST inside result if necessary.
               continue;
           }
           if (result.ast == NULL) {
               // Empty line or comment, continue.
               continue;
           }
```

```
        */

        /* TODO 6: Execute the parsed command AST.
           exit_status = execute_ast(result.ast, state);
           // Store the exit status for potential use by future commands ($?)
           state->last_exit_status = exit_status;
        */

        /* TODO 7: Clean up the AST.
           free_command_ast(result.ast);
        */

        /* TODO 8: Handle the 'exit' built-in command.
           The execute_ast function should have handled the 'exit' built-in
           by returning a special status or setting a flag. Check for that
           here and break the loop if needed.
           Example: if (exit_status == EXIT_BUILTIN_CALLED) { break; }
        */
    }

    /* TODO 9: Perform final cleanup before returning.
       - Free the job table (job_table_free)
       - Reset terminal settings if job control was active
    */
    return exit_status;
}
```

E. Language-Specific Hints (C)

- **Signal Handler Installation**: Use `sigaction()` not `signal()`. It provides better control and portability. Remember to use the `SA_RESTART` flag for system calls you *want* to be restarted (though we handle `EINTR` manually in the REPL).
- **Blocking Signals**: Use `sigprocmask(SIG_BLOCK, &mask, NULL)` to block signals before creating the global state or spawning child processes, then unblock them appropriately. For the `pselect` strategy, you block signals globally, then pass an empty mask to `pselect` to temporarily unblock them.
- **Atomic Flags**: Always declare global signal flags as `volatile sig_atomic_t`. This ensures reads and writes are atomic and aren't cached in registers.
- **Checking for Background Jobs**: In `job_control_check_completed_jobs`, use `waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED)` in a loop. The `WNOHANG` flag makes it non-blocking, `WUNTRACED` catches stopped jobs (Ctrl+Z), and `WCONTINUED` catches resumed jobs (`bg`).
- **Avoiding `printf` in Handlers**: If you must output from a signal handler, use the async-signal-safe `write()` system call directly to file descriptor 2 (stderr).

F. Milestone Checkpoint (Milestone 1 & 5)

After implementing the basic REPL and background job reporting:

1. **Test Basic Loop**: Run the shell and type `echo hello`. It should print `hello` and show a new prompt.
2. **Test Background Jobs**: Run `sleep 5 &`. The shell should immediately print a job number (e.g., `[1] 1234`) and show a new prompt.
3. **Test Job Reporting**: Wait 5 seconds, then simply press **Enter** on an empty line. The shell should print `[1]+ Done sleep 5` before showing the next prompt.
4. **Test Signal Safety**: While `sleep 10 &` is running, quickly type another command like `pwd`. The `pwd` should execute immediately, and the `Done` message for `sleep` should appear before the next prompt *after* `pwd` completes, not before.

Signs of Trouble:

- The shell hangs after a background command (you can't type). → Likely the parent is incorrectly calling `waitpid` without `WNOHANG` for background jobs.
- No `Done` message appears until you run another *foreground* command. → `job_control_check_completed_jobs` is not being called at the start of every REPL iteration.
- The shell crashes when a background job finishes while you're typing. → Signal handler is not minimal or is modifying global data unsafely.

Component Design: The Parser

> **Milestone(s):** 1, 2, 3, 4 – This component is foundational to the entire shell, transforming raw user input into executable structures. It is built incrementally: basic tokenization in Milestone 1, support for built-in command detection in Milestone 2, recognition of redirection operators in Milestone 3, and finally pipeline parsing in Milestone 4.

The Parser is the shell's linguist, responsible for understanding the user's typed instructions and converting them into a precise, structured format that the Executor can act upon. It bridges the gap between human-readable command lines and the machine-executable operations of process creation and I/O manipulation.

Mental Model: Language Translator

Imagine the parser as a specialized language translator at an international airport. Travelers (users) arrive speaking various dialects of "Unix command line" – sometimes with pipes, sometimes with redirections, sometimes with quotes and spaces. The translator's job is to:

1. **Listen carefully** to the entire utterance, respecting its grammatical rules (like quoted phrases being single units).
2. **Break it down** into individual meaningful words and symbols (tokens).
3. **Analyze the structure** to understand the relationships: which words form a command, which symbols modify input/output, and how commands are connected.
4. **Produce a translation** into a formal "machine language" – in this case, an Abstract Syntax Tree (AST) of `Command` nodes – that can be handed off to the Executor (the "ground crew") for actual execution.

This translation must be precise and unambiguous. A misunderstanding (e.g., treating a `>` as part of a filename instead of a redirection operator) leads to incorrect execution, just as a mistranslation could send a traveler to the wrong gate.

Interface and Tokenization

The parser's primary interface is a single function that consumes a raw command line string and the current shell state, and returns a structured `ParseResult`.

Method Name	Parameters	Returns	Description
`parse_command_line`	`char* input_line`, `ShellState* state`	`ParseResult`	The main parsing entry point. Tokenizes the input, constructs an AST, and returns either a successful AST or an error message.

The `ParseResult` structure, defined in the Data Model, encapsulates the outcome:

Field Name	Type	Description
`ast`	`Command*`	The root of the parsed command AST (a linked list for pipelines). `NULL` on error.
`error_message`	`char*`	Human-readable error description if parsing failed. `NULL` on success.
`success`	`bool`	Indicates whether parsing succeeded.

Parsing is a two-phase process: **tokenization** followed by **syntax tree construction**.

Tokenization (or lexical analysis) scans the raw input string and breaks it into a sequence of **tokens**. A token is the smallest meaningful unit: a word (like `"ls"` or `"file.txt"`), an operator (like `"|"`, `">"`, `"<"`), or a quotation mark. The tokenizer must handle several important rules:

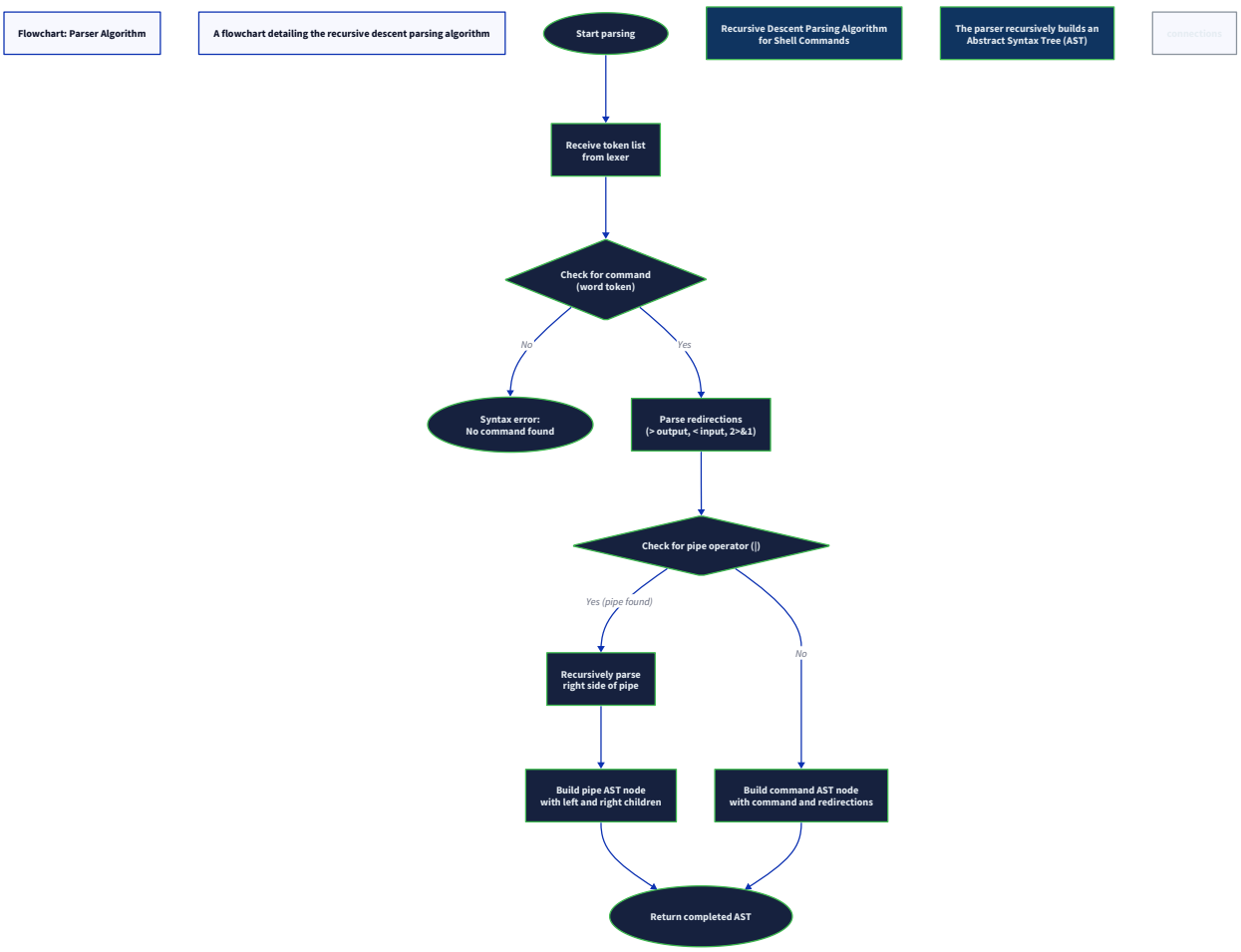
- **Whitespace separation**: Spaces and tabs generally separate tokens, except within quotes.
- **Quoted strings**: Text between single quotes (`'`) or double quotes (`"`) is treated as a single token, preserving all internal spaces and special characters. The quotes themselves are not included in the token's value.
- **Escape sequences**: A backslash (`\`) can "escape" the next character, allowing special characters (like spaces, quotes, or operators) to appear within a word. The tokenizer removes the backslash.
- **Operators**: The characters `|`, `>`, `<`, `&` and their combinations (`>>`, `2>`) are recognized as distinct operator tokens, not as part of a word.

The tokenizer produces a linear list of tokens, which is then passed to the syntax tree builder.

Parsing Algorithm and Abstract Syntax Tree (AST) Construction

The syntax tree construction phase analyzes the sequence of tokens to determine the hierarchical structure of the command. The shell's grammar is relatively simple but has important precedence rules: redirections bind tightly to an individual command, while pipes connect separate commands. The output is an **Abstract Syntax Tree (AST)**, which in our design is a linked list of `Command` nodes, where each node may have associated redirections and the `next` pointer represents a pipe.

The parsing algorithm follows these steps, which can be visualized in the provided flowchart:



1. **Initialization**: Create an empty `ParseResult`. If the input line is empty or contains only whitespace, return a successful result with a `NULL` AST (the REPL will handle this gracefully).
2. **Tokenization**: Invoke the tokenizer on the input string. It scans character by character, building tokens and handling quotes and escapes as described above. It returns a `NULL`-terminated array of token strings (or a suitable dynamic list). If an opening quote is never closed, the tokenizer should fail and return an error.
3. **Parse Pipeline (Recursive Descent)**: Starting at the beginning of the token array, the parser attempts to parse a **pipeline**. A pipeline is defined as one or more **commands** separated by pipe (`|`) operators. The parser uses a recursive approach: a. Parse a single **command** (including its arguments and redirections) from the current token position. This function returns a `Command` node and advances the token position. b. Check the current token. If it is a pipe (`|`) operator, consume it (advance the token position), then recursively parse the next pipeline (the right-hand side). The pipe operator is **left-associative**, meaning `a | b | c` is interpreted as `(a | b) | c`. c. Connect the left `Command` node's `next` pointer to the `Command` node returned from the right-hand side parse. This builds the linked list.
4. **Parse Command**: To parse a single command (a node in the pipeline AST), the algorithm: a. Create a new `Command` node. Initialize its `argv` list (arguments vector) as empty, its `argc` as 0, and its redirection lists as `NULL`. b. While the current token is not a pipe (`|`), background (`&`) operator, or end-of-tokens: - If the token is a **redirection operator** (`<` , `>` , `>>` , `2>`), call `parse_redirection`. This function consumes the operator token and the next token (the filename), creates a `Redirection` node, and adds it to the appropriate list (`input_redir` or `output_redir`) of the current `Command`. Redirections are not added to `argv`. - Otherwise, the token is a **word**. Add it to the `argv` array of the current command, incrementing `argc`. This includes the command name itself (the first word) and all its arguments. c. Ensure `argv` is `NULL`-terminated (required by `execvp`). d. If no words were found (i.e., `argc` is 0), but redirections were present (e.g., `> file`), this is an invalid command. Return an error. e. Return the populated `Command` node.

5. **Trailing Background Operator**: After the full pipeline is parsed, check for a trailing `&` token. If present, this information is not stored directly in the AST. Instead, the parser sets a flag in the `ParseResult` (or an extended structure) indicating a background job. The REPL or Executor will use this flag when creating a `Job`.
6. **Final Validation**: Ensure no extraneous tokens remain after parsing the pipeline and optional `&`. If any do, it's a syntax error (e.g., stray operator).

The resulting AST for the command `ls -l | grep "foo bar" > out.txt` is a linked list with two `Command` nodes:

- Node 1: `argv = ["ls", "-l"]`, `output_redir = NULL`, `next` points to Node 2.
- Node 2: `argv = ["grep", "foo bar"]`, `output_redir` points to a `Redirection` node of type `REDIR_OUTPUT` with filename `"out.txt"`, `next = NULL`.

This data structure cleanly separates the linear flow of data (the pipeline chain) from the per-command modifications (redirections).

ADR: Recursive Descent vs. Shunting Yard for Operator Parsing

> **Decision: Use Recursive Descent Parsing for Shell Command Lines**
>
> - **Context**: The shell must parse a simple but non-regular grammar featuring binary infix operators (pipe `|`) with left associativity, and prefix/postfix operators (redirections `>`, `<`) that bind more tightly than pipes. We need an algorithm that is clear, easy to implement correctly, and maps naturally to our AST structure.
> - **Options Considered**:
>   1. **Recursive Descent Parser**: Manually code parsing functions that mirror the grammar's production rules (e.g., `parse_pipeline()`, `parse_command()`), using recursion to handle nested structures.
>   2. **Shunting Yard Algorithm**: Convert the token stream to Reverse Polish Notation (RPN) using an operator stack, then build the AST from the RPN output. This is a classic algorithm for parsing expressions with operator precedence.
>   3. **Operator-Precedence Climbing**: A variant of recursive descent that uses precedence tables to guide recursion, often used in expression parsers.
> - **Decision**: Implement a **hand-written recursive descent parser**.
> - **Rationale**:
>   - **Simplicity and Clarity**: The shell's grammar is small and fixed. Recursive descent code directly reflects the grammar rules, making it easier to read, debug, and extend (e.g., adding new operators). The mental model of "parse a pipeline, which contains commands separated by pipes" maps directly to a function `parse_pipeline()` that calls `parse_command()` in a loop.
>   - **Natural AST Construction**: As we parse, we can immediately build the corresponding `Command` and `Redirection` structs and link them together. There's no intermediate RPN conversion step.
>   - **Sufficient for Requirements**: The grammar has only one level of operator precedence (redirections bind tighter than pipes), which is trivial to encode in a recursive descent parser by having `parse_pipeline` call `parse_command`, which consumes all redirections before returning.
>   - **Educational Value**: Implementing recursive descent is a fundamental compiler technique that reinforces understanding of parsing and tree construction.
> - **Consequences**:
>   - **Positive**: The parser will be straightforward to write and test in stages (tokenizer, then command parser, then pipeline parser). Error reporting can be localized to specific parsing functions.
>   - **Negative**: For a much more complex grammar with many precedence levels, the recursive descent code could become verbose. This is not a concern for our shell's simple syntax.

The following table summarizes the comparison:

Option	Pros	Cons	Why Not Chosen?
Recursive Descent	Direct mapping to grammar; Easy to implement and debug; Immediate AST construction; Excellent for simple grammars.	Can become cumbersome for grammars with many precedence levels; Manual tracking of token position needed.	**CHOSEN** – Perfect fit for the shell's simple, fixed grammar.
Shunting Yard	Handles arbitrary operator precedence elegantly; Table-driven approach separates grammar from code.	More abstract; Requires intermediate RPN representation; Overkill for our needs; Less intuitive for beginners.	Over-engineering. Adds complexity without benefit for our small grammar.
Operator-Precedence Climbing	Elegant handling of precedence and associativity; Often more concise than pure recursive descent for expressions.	Slightly more complex conceptual model; Still requires separate handling of non-expression constructs (like redirections).	Unnecessary complexity. Our simple precedence is easily handled by the structure of recursive descent functions.

Common Pitfalls

Learners implementing the parser often encounter several specific, frustrating issues. Awareness of these can save hours of debugging.

⚡ Pitfall 1: Memory Leaks in the AST

- **Description**: Allocating `Command`, `argv` strings, and `Redirection` nodes with `malloc()` but never freeing them, especially when parsing fails partway through or when the shell exits.
- **Why it's wrong**: The shell is a long-running process (REPL). Every leaked allocation accumulates, causing the shell's memory footprint to grow indefinitely with each command entered.
- **How to fix**: Implement a comprehensive `free_command_ast(Command*)` function that recursively walks the AST and frees all nested structures (arguments, redirection filenames, and nodes). Call this function in the REPL after command execution is complete, and also in the parser when abandoning a partially-built AST due to a parse error.

⚡ Pitfall 2: Incorrect Handling of Edge Spaces and Empty Commands

- **Description**: Not trimming leading/trailing whitespace from the input line, or incorrectly handling multiple spaces between tokens. This can lead to empty tokens or misidentified command boundaries. Also, failing to handle an empty input line (just pressing Enter) gracefully.
- **Why it's wrong**: Users expect the shell to be robust to extra spaces. An empty line should simply re-display the prompt, not crash or print an error.
- **How to fix**: After reading the input line, strip trailing newline characters. The tokenizer should skip over runs of whitespace, not produce empty tokens. The parser should treat an empty token list as a valid "no-op" and return a `NULL` AST (successfully). The REPL must check for this and continue the loop.

⚡ Pitfall 3: Forgetting to NULL-Terminate the argv Array

- **Description**: Building the `argv` array for a command by dynamically adding argument strings but neglecting to append a final `NULL` pointer.
- **Why it's wrong**: The `execvp()` system call expects a `NULL`-terminated array. Passing an array without the sentinel `NULL` causes `execvp` to read beyond the allocated memory, leading to a crash (segmentation fault) or the execution of garbage "arguments."
- **How to fix**: When building the `argv` array for a `Command`, allocate space for `argc + 1` pointers. Explicitly set `argv[argc] = NULL` after all arguments have been added. Double-check this in the `parse_command` function.

⚡ Pitfall 4: Misparsing Redirections Without a Target File

- **Description**: Correctly identifying a redirection operator token (`>`) but not consuming the following token as the filename, or consuming an operator as the filename (e.g., parsing `ls > |` as output redirection to a file named `"|"`).
- **Why it's wrong**: The shell should report a syntax error for malformed redirections. Silently treating the next token incorrectly leads to confusing behavior.

- **How to fix**: In `parse_redirection`, after consuming the operator token, check that the next token exists and is a word (not another operator or end-of-line). If not, produce a clear error message like "syntax error near unexpected token `|`".

⚡ **Pitfall 5: Not Handling Quoted Strings as Single Tokens**

- **Description**: The tokenizer splits on spaces without regard for quotes, so a command like `echo "hello world"` produces three tokens: `"echo"`, `"\"hello"`, `"world\""`. This breaks the command.
- **Why it's wrong**: It violates the fundamental shell grammar where spaces inside quotes are part of the argument.
- **How to fix**: Implement a stateful tokenizer that tracks whether it's inside single or double quotes. When inside quotes, spaces do not end the token. The quotes themselves should be stripped from the final token value. Remember to handle escape sequences (`\`) within quotes correctly.

Implementation Guidance

This section provides concrete C code skeletons to help you implement the parser. The core logic is left for you to fill in, following the algorithm described above.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Tokenizer	Manual character-by-character scanning with state flags for quotes.	Using `strtok_r` with custom delimiters and careful quote handling (tricky).
Parser	Hand-written recursive descent as described.	Generated parser (e.g., using Lex/Yacc or similar) – overkill for this project.
Dynamic Arrays	Use `realloc` for growing `argv` and token lists.	Implement a generic vector/array list utility.

B. Recommended File/Module Structure:

Place the parser in its own source file to separate concerns from the REPL and executor.

```
gauntlet_shell/
├── src/
│   ├── main.c              # Entry point, initializes ShellState, calls run_repl_loop.
│   ├── repl.c              # Contains run_repl_loop and prompt display logic.
│   ├── parser.c            # This component: parse_command_line, tokenizer, helper functions.
│   ├── parser.h            # Declarations for ParseResult, parsing functions.
│   ├── executor.c          # Contains execute_ast, built-in commands, redirection setup.
│   ├── job_control.c       # Job table management, signal handlers for job control.
│   └── utils.c             # Common utilities (e.g., string manipulation, safe malloc).
├── include/
│   └── shell.h             # Main header with all type definitions (ShellState, Command, etc.).
└── Makefile
```

C. Core Logic Skeleton Code:

First, the main parsing function that orchestrates tokenization and AST building:

```c
// parser.c

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include "shell.h"


// Forward declarations for helper functions.

static char** tokenize(const char* line, int* token_count, char** error_msg);

static Command* parse_pipeline(char*** tokens, int* token_pos, int token_count, char** error_msg);

static Command* parse_single_command(char*** tokens, int* token_pos, int token_count, char** error_msg);

static void parse_redirection(char*** tokens, int* token_pos, int token_count,

                              Command* cmd, char** error_msg);


ParseResult parse_command_line(char* input_line, ShellState* state) {

    ParseResult result = { .ast = NULL, .error_message = NULL, .success = false };

    // TODO 1: Trim trailing newline from input_line (if present).

    // TODO 2: Check for empty line or comment line (starting with '#'). If so,

    //         set result.success = true (no AST) and return.

    // TODO 3: Call tokenize() to convert input_line into an array of tokens.

    //         Handle tokenization error: set result.error_message and return.

    // TODO 4: Initialize token position index (e.g., pos = 0).

    // TODO 5: Call parse_pipeline() to build the AST from tokens.

    //         Pass the token array, position pointer, and token count.

    // TODO 6: Check for trailing '&' token. If found, consume it.

    //         We'll handle background flag later in the REPL/Executor.

    // TODO 7: Ensure no tokens remain after parsing. If extra tokens, set error.

    // TODO 8: On success, set result.ast and result.success = true.

    // TODO 9: Free the token array (but NOT the strings within, as they are now owned by the AST).

    // TODO 10: Return the ParseResult.

    return result;

}
```

Next, the tokenizer function that performs lexical analysis:

```c
// parser.c (continued)

static char** tokenize(const char* line, int* token_count, char** error_msg) {

    // TODO 1: Allocate an initial array of char* (e.g., capacity 16).

    // TODO 2: Iterate through each character in the line.

    // TODO 3: Skip leading whitespace.

    // TODO 4: Determine the start of a token.

    // TODO 5: Based on the first character, decide token type:

    //         - If it's a single/double quote, read until the matching quote.

    //         - If it's a backslash, escape the next character.

    //         - If it's an operator character (|, >, <, &), read the longest possible operator (e.g., ">>").

    //         - Otherwise, read until whitespace or special character.

    // TODO 6: For each token, allocate memory and copy the token value (without quotes).

    // TODO 7: Add the token pointer to the array. Resize array if necessary.

    // TODO 8: If a quote is not closed, set *error_msg and clean up allocated memory.

    // TODO 9: After loop, set *token_count and NULL-terminate the token array.

    // TODO 10: Return the token array.

    return NULL;

}
```

The recursive descent functions for pipelines and commands:

```c
// parser.c (continued)

static Command* parse_pipeline(char*** tokens, int* token_pos, int token_count, char** error_msg) {

    // TODO 1: Parse the left-hand side command by calling parse_single_command().

    // TODO 2: If parse_single_command returned NULL (error), propagate error.

    // TODO 3: While *token_pos < token_count and current token is "|":

    //         a. Consume the "|" token (increment *token_pos).

    //         b. Parse the right-hand side command (another parse_single_command).

    //         c. If right-hand side is NULL, set error and clean up left side.

    //         d. Set left_command->next = right_command.

    //         e. left_command = right_command (to continue chaining if more pipes).

    // TODO 4: Return the head of the command linked list.

    return NULL;

}


static Command* parse_single_command(char*** tokens, int* token_pos, int token_count, char** error_msg) {

    // TODO 1: Allocate and initialize a new Command struct (use calloc).

    // TODO 2: Create a dynamic array for argv (start with capacity, e.g., 8).

    // TODO 3: While *token_pos < token_count:

    //         - If token is "<", ">", ">>", or "2>": call parse_redirection().

    //         - Else if token is "|" or "&": break (end of this command).

    //         - Else: add token to argv (realloc if needed), increment argc.

    // TODO 4: Ensure argv is NULL-terminated (argv[argc] = NULL).

    // TODO 5: If argc == 0 and no redirections were found, free the command and return NULL (empty command).

    // TODO 6: If argc == 0 but redirections exist, set error: "syntax error near redirection".

    // TODO 7: Return the command node.

    return NULL;

}


static void parse_redirection(char*** tokens, int* token_pos, int token_count,

                              Command* cmd, char** error_msg) {

    // TODO 1: Save the operator token (e.g., ">").

    // TODO 2: Increment *token_pos to consume the operator.

    // TODO 3: Check if *token_pos >= token_count. If so, set error: "syntax error: expected filename".

    // TODO 4: The next token is the filename. Ensure it's not an operator (e.g., "|").

    //         If it is an operator, set error.
```

```
    // TODO 5: Create a new Redirection struct.
    // TODO 6: Set its type based on the operator (REDIR_INPUT, REDIR_OUTPUT, etc.).
    // TODO 7: Set its filename to a copy of the token (strdup).
    // TODO 8: Set its fd field appropriately (e.g., 0 for input, 1 for output, 2 for error).
    // TODO 9: Add the redirection to cmd's input_redir or output_redir linked list.
    //         Maintain the list order (order matters for shell semantics).
    // TODO 10: Increment *token_pos to consume the filename.
}
```

D. Language-Specific Hints:

- **String Duplication**: Use `strdup()` to copy token strings into the AST. Remember `strdup()` allocates memory with `malloc`, so you must `free` it later.
- **Dynamic Arrays**: When building `argv`, start with a small allocated array (e.g., 8 pointers). Use `realloc` to double the size when full. This is more efficient than reallocating for every new argument.
- **Error Messages**: For `error_msg` parameters, allocate a string with `malloc` and write a descriptive message. The caller (`parse_command_line`) is responsible for freeing this memory in the `ParseResult`.
- **Const Correctness**: The tokenizer receives a `const char*` line because it shouldn't modify the input. The parsing functions receive `char*** tokens` (a pointer to the token array pointer) so they can advance the token position by modifying `*tokens` or a separate index.

E. Milestone Checkpoint:

After implementing the parser for each milestone, verify its behavior:

- **Milestone 1 (Basic Tokenization)**: Test with `echo hello world`. The parser should produce a single `Command` node with `argv = ["echo", "hello", "world"]`. Use a debug function to print the AST.
- **Milestone 2 (Built-ins)**: The parser does not need to differentiate built-ins; that's the executor's job. Ensure built-in names like `cd` are correctly tokenized as regular words.
- **Milestone 3 (Redirections)**: Test `ls > file.txt`. The AST should have one command with `argv = ["ls"]` and an output redirection to `"file.txt"`. Test multiple redirections: `cat < input.txt > output.txt`.
- **Milestone 4 (Pipes)**: Test `ls | grep foo`. The AST should be a linked list of two commands. Test longer pipelines: `ls -l | grep foo | wc -l`.

You can write a simple test driver in `main.c` that calls `parse_command_line` with hardcoded strings and prints the resulting AST structure.

Component Design: The Executor

> **Milestone(s):** 1, 2, 3, 4 – This component is the engine room of the shell, responsible for transforming the parser's abstract syntax tree into actual running processes. It handles the intricate setup of file descriptors for redirections and pipes, executes both external programs and built-in commands, and manages the lifecycle of child processes. The executor's design must carefully balance correctness (ensuring processes see the right I/O streams), safety (avoiding resource leaks), and concurrency (handling pipelines and background jobs).

Mental Model: The Stage Manager and Set Designer

Imagine a theater production. The **Parser** is the scriptwriter who analyzes the playwright's text and produces a detailed scene breakdown. The **Executor** is the stage manager and set designer who brings that script to life. For each scene (command), the executor must:

1. **Cast actors (processes)**: Create child processes (`fork`) to play the roles specified in the script.
2. **Design the set (I/O setup)**: Arrange the scenery—pipes between actors, files for input/output, and connections to the audience (terminal). This involves manipulating file descriptors with `dup2` and `close` .
3. **Cue the actors (exec)**: Give each actor their script (program to run) and say "action!" (`exec`).
4. **Manage the stage (coordination)**: Ensure actors enter and exit at the right times (`waitpid`), handle interruptions from the director (signals), and clean up the set after the scene ends (close file descriptors).

This mental model emphasizes the executor's role as a **coordinator and enabler** rather than a performer. It doesn't execute code itself (except built-ins); it creates the precise environment in which other programs can run correctly, with their input, output, and error streams connected exactly as the user specified in their command line.

Interface and Core Functions

The executor's public interface consists of several key functions that other components (primarily the REPL) call to transform parsed commands into running processes.

Method Name	Parameters	Returns	Description
`execute_ast`	`Command* ast, ShellState* state`	`int` (exit status)	The main entry point for executing a command AST. Handles both single commands and pipelines. For built-in commands, executes them directly in the shell process. For external commands and pipelines, orchestrates forking, redirection setup, and exec. Returns the exit status of the last command in the pipeline or the built-in command.
`setup_redirections`	`Command* cmd`	`int` (0 on success, -1 on error)	Helper function that applies all I/O redirections for a single command. Opens files, duplicates file descriptors with `dup2` , and ensures unused descriptors are closed. Called in the child process before `exec` .
`execute_builtin`	`int argc, char** argv, ShellState* state`	`int` (exit status)	Dispatches to the appropriate built-in command handler (cd, exit, export, etc.) based on the first argument. Executes in the shell's own process (no fork).
`execute_external`	`Command* cmd, ShellState* state, int in_fd, int out_fd, int err_fd, int is_background, pid_t pgid`	`pid_t` (child PID or -1 on error)	Forks a child process, sets up the specified input/output/error file descriptors (which may be pipes or redirected files), optionally sets the process group, and calls `execvp` . Returns the child's PID to the parent.
`execute_pipeline`	`Command* pipeline, ShellState* state, int is_background`	`int` (exit status of last command)	Iterates through a linked list of `Command` nodes (the pipeline). Creates pipes between consecutive commands, forks child processes for each, connects their stdin/stdout appropriately via `dup2` , and waits for the entire pipeline to complete.

These functions rely on the following key data structures (defined in the Data Model section):

Structure	Key Fields for Execution	Purpose in Execution
`Command`	`argv`, `argc`, `input_redir`, `output_redir`, `next`	Represents a single command in a pipeline. `argv`/`argc` are passed to `execvp`. Redirection fields specify file operations. `next` pointer links to the next command in a pipeline.
`Redirection`	`type`, `filename`, `fd`	Specifies a single I/O redirection: type (input, output, append, error), target filename, and which file descriptor (0, 1, or 2) to redirect. Linked list allows multiple redirections per command.
`ShellState`	`foreground_pgid`, `last_exit_status`	Tracks which process group currently has control of the terminal (foreground job). Stores the exit status of the last executed command for use by the shell itself (e.g., in `$?` expansion, not covered in core milestones but useful for extensibility).

Execution Algorithm: From AST to Processes

The core algorithm for executing a command pipeline follows a carefully orchestrated sequence to avoid race conditions, deadlocks, and resource leaks. Consider a pipeline like `ls -l | grep foo > output.txt`. The executor must:

1. **Parse the AST**: The parser produces a linked list of two `Command` nodes: the first with `argv = ["ls", "-l"]` and no redirections; the second with `argv = ["grep", "foo"]` and an output redirection to `output.txt`.
2. **Determine execution mode**: Check if the pipeline should run in background (trailing `&`). If foreground, the shell must give the pipeline's process group terminal control and wait for it. If background, it's added to the job table.
3. **Create communication channels**: For N commands in a pipeline, create N-1 pipes. Each pipe is a pair of file descriptors: `pipe[0]` for reading, `pipe[1]` for writing.
4. **Fork and set up each child**: Iterate through commands, forking a child for each. In each child:
   - If not the first command, connect stdin to the previous pipe's read end.
   - If not the last command, connect stdout to the current pipe's write end.
   - Apply command-specific redirections (which may override pipe connections).
   - Close all pipe file descriptors not needed by this child.
   - Set process group (so all pipeline processes can be signaled together).
   - Call `execvp`.
5. **Parent cleanup and wait**: The parent shell closes all pipe ends (they're inherited by children but unused by parent). If foreground, it waits for the entire pipeline's process group to finish. If background, it records the pipeline as a job.

Here is the detailed step-by-step algorithm implemented in `execute_ast`:

Algorithm: execute_ast(Command ast, ShellState state)**

1. **Handle empty commands**: If `ast` is NULL (empty input line), return 0 (success).
2. **Check for built-in commands**: If the first word matches a built-in command (cd, exit, export, etc.), call `execute_builtin` with the arguments and return its status. Built-ins run in the shell process itself—no forking.
3. **Determine if background**: Check if the last command in the pipeline has a trailing `&` in its arguments (parser should mark this via a flag, but for simplicity, we can check the last argument of the last command). Set `is_background` flag.
4. **If single external command (no pipe)**:
   - Fork a child process.
   - **In child**: Setup redirections via `setup_redirections`, then call `execvp`.
   - **In parent**: If foreground, wait for this child with `waitpid` and return its exit status. If background, add to job table and return 0.
5. **If pipeline (multiple commands)**:
   - Create an array of pipes: `int pipes[(n_commands-1)][2]`.
   - For each command `i` from 0 to n_commands-1:

        - Fork child process.
        - **In child**:
            - Set process group: If first child, create new group with its PID. Subsequent children join this group.
            - Setup stdin: If `i > 0`, `dup2(pipes[i-1][0], STDIN_FILENO)`.
            - Setup stdout: If `i < n_commands-1`, `dup2(pipes[i][1], STDOUT_FILENO)`.
            - Close all pipe file descriptors in the child (all `pipes[j][0]` and `pipes[j][1]` for all j).
            - Apply command-specific redirections via `setup_redirections` (these may override pipe connections).
            - Call `execvp`.
        - **In parent**: Store child PID. If first child, set its process group as the foreground group if not background.
    - **In parent after forking all**:
        - Close all pipe file descriptors (parent doesn't need them).
        - If foreground: Wait for the entire process group. Use `waitpid(-pgid, &status, 0)` to wait for any child in the group. Repeat until no more children in that group remain. Return exit status of last command.
        - If background: Add pipeline to job table with process group ID, return 0.

The most delicate part is ensuring **every file descriptor is accounted for**. The golden rule: After `dup2(old_fd, target_fd)`, you must close `old_fd` if it's no longer needed. Pipes have two ends; each end must be closed in all processes that don't use it to avoid hanging (process waiting for EOF that never comes).

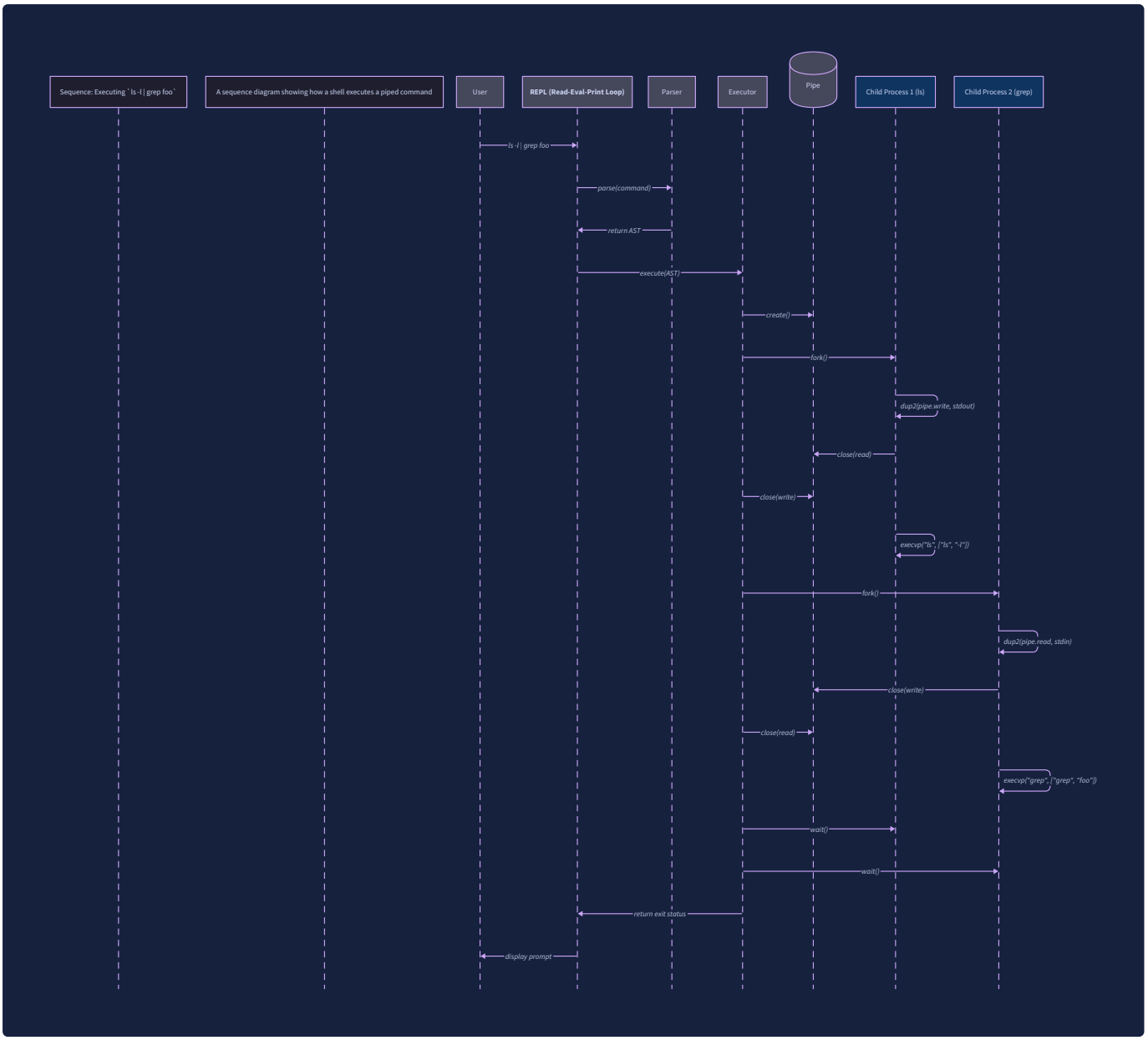


Diagram 1: Sequence of interactions for a simple pipeline. Shows the REPL calling the executor, which creates pipes, forks children, sets up file descriptors, and waits.

ADR: Fork-Then-Exec Pattern vs. vfork/clone

> **Decision: Use Standard fork()/exec() Pattern**
>
> - **Context**: The shell must create child processes to run external programs. The child inherits the shell's memory space, file descriptors, and other attributes, but then immediately replaces its memory image with a new program via `exec`. The performance cost of copying the parent's memory (via `fork`) is a concern, especially for large shell processes.
> - **Options Considered**:
>   1. **Standard `fork()` / `exec()`**: Copy entire address space then replace.
>   2. **`vfork()` / `exec()`**: Special variant that suspends the parent until child execs or exits, sharing address space.
>   3. **`clone()` (Linux-specific)**: More flexible process creation allowing shared memory, files, etc.
> - **Decision**: Use the standard `fork()` followed by `exec()`.
> - **Rationale**:
>   - **Simplicity and Portability**: `fork()` / `exec()` is the POSIX standard, works everywhere. `vfork()` has tricky semantics (parent is suspended) and is largely obsolete on systems with copy-on-write `fork`. `clone()` is Linux-specific.
>   - **Safety**: With `fork()`, parent and child run in separate address spaces immediately. Any bugs in child setup (before `exec`) won't corrupt parent shell state. `vfork()` shares memory, making the parent vulnerable.
>   - **Performance Modern Reality**: Modern Unix systems implement `fork()` with copy-on-write (COW) pages. The actual memory copy occurs only when either process modifies a page. Since the child calls `exec()` immediately (which replaces the address space), very few pages are actually copied. The overhead is minimal.
>   - **Consistency with Job Control**: `fork()` correctly duplicates process group IDs and other attributes needed for job control. The semantics are well-defined.
> - **Consequences**:
>   - **Slightly higher overhead than `vfork`**: In extreme cases with very large shell memory, there may be some overhead, but for an educational shell, this is negligible.
>   - **Robustness**: The shell is less likely to crash due to child memory corruption.
>   - **Code clarity**: The pattern is familiar to all Unix programmers, making the codebase more accessible.

Option	Pros	Cons	Chosen?
`fork()` / `exec()`	Portable, safe (separate address spaces), standard, works well with copy-on-write	May copy some pages before exec (minor overhead)	**Yes**
`vfork()` / `exec()`	Historically more efficient on systems without COW fork	Dangerous (shared memory), obscure semantics, deprecated on many systems	No
`clone()` with careful flags	Maximum flexibility, can optimize shared resources	Linux-specific, complex API, overkill for shell needs	No

Common Pitfalls

The executor is fraught with subtle bugs that can cause the shell to hang, leak resources, or behave incorrectly. Here are the most common pitfalls and how to avoid them:

⚠️ Pitfall 1: File Descriptor Leaks Causing Hangs in Pipelines

- **Description**: Forgetting to close unused pipe ends in the parent and children. For example, if the parent doesn't close the write end of a pipe, the reader (child) will never see EOF because the write end is still open in the parent.
- **Why it's wrong**: Causes processes to hang waiting for input that will never come (deadlock). Particularly common in multi-stage pipelines.
- **Fix**: Follow the **golden rule of pipes**: After forking, close **all** pipe ends you don't need in **every** process. Use a systematic approach: after creating pipes, immediately close unneeded ends in the parent before forking, then in each child close all pipe ends after

duplicating the needed ones with `dup2`.

⚠️ Pitfall 2: Incorrect Order of dup2 and close Calls

- **Description**: Calling `close(target_fd)` before `dup2(old_fd, target_fd)` can accidentally close the source file descriptor if `old_fd == target_fd`.
- **Why it's wrong**: If you're redirecting stdin (fd 0) from a pipe (fd 4), and you call `close(0)` before `dup2(4, 0)`, you close fd 0. Then `dup2(4, 0)` works, but if fd 4 happened to be 0 (unlikely but possible), you've lost the source. More commonly, if `old_fd == target_fd`, `dup2` does nothing and returns success, then you close it.
- **Fix**: Always call `dup2` first, then close the original descriptor if it's different from the target: `if (old_fd != target_fd) close(old_fd);`.

⚠️ Pitfall 3: Not Handling Redirection Errors Before exec

- **Description**: Failing to check return values of `open()`, `dup2()`, etc., and proceeding to `exec` when redirection setup failed.
- **Why it's wrong**: The child will `exec` with incorrect file descriptors, potentially causing the program to read from/write to unexpected places. The parent may not know about the failure.
- **Fix**: In the child, check all system calls during redirection setup. If any fails, print an error to stderr (fd 2 is still terminal at this point) and exit with a non-zero status (e.g., 1) instead of calling `exec`.

⚠️ Pitfall 4: Race Conditions in Pipeline Setup

- **Description**: The parent forks children in a loop but doesn't set up process groups correctly, leading to signals being delivered at wrong times. Or, children start executing (`exec`) before parent has closed pipe ends, causing unpredictable behavior.
- **Why it's wrong**: Can cause processes to receive signals intended for others, or I/O to work intermittently.
- **Fix**: Use process groups: first child sets its own process group, subsequent children join it. In the parent, after forking all children, set the pipeline's process group as the foreground group (if foreground) **before** waiting. Ensure all pipe ends are closed in parent before waiting.

⚠️ Pitfall 5: Forgetting to Wait for All Children in a Pipeline

- **Description**: Using `waitpid` with a specific PID instead of `-pgid` to wait for the entire process group, or only waiting once when multiple children exist.
- **Why it's wrong**: Leaves zombie processes (children that exited but whose status hasn't been reaped). Zombies consume kernel resources and eventually prevent new processes from being created.
- **Fix**: Use `waitpid(-pgid, &status, 0)` in a loop until it returns -1 with `errno == ECHILD`. This waits for all children in the process group. For background jobs, install a `SIGCHLD` handler that reaps children asynchronously.

⚠️ Pitfall 6: Built-in Commands in Children

- **Description**: Accidentally forking for built-in commands like `cd` or `export`.
- **Why it's wrong**: Built-ins must affect the shell's own state (current directory, environment variables). If run in a child, changes are lost when the child exits.
- **Fix**: Check for built-in commands **before** forking. Execute them directly in the shell process and return their status.

Implementation Guidance

This section provides concrete C code skeletons to implement the executor. The code follows the naming conventions and structures defined earlier.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Process Creation	`fork()` + `execvp()`	`posix_spawn()` (more control, but complex)
File Descriptor Manipulation	`dup2()`, `close()`	`fcntl()` with `F_DUPFD_CLOEXEC` (automatic close-on-exec)
Path Resolution	`execvp()` (searches PATH automatically)	Manual PATH search with `access()` for better error messages
Pipeline Synchronization	Simple loops with careful fd closing	Using `pipe2()` with `O_CLOEXEC` flag to avoid leaks

B. Recommended File/Module Structure

Add executor-related files to the project structure:

```
gauntlet_shell/
├── src/
│   ├── main.c              # Entry point, calls run_repl_loop
│   ├── repl.c              # REPL implementation
│   ├── parser.c            # Parser implementation
│   ├── executor.c          # **EXECUTOR IMPLEMENTATION (this section)**
│   ├── job_control.c       # Job table and signal handling
│   ├── builtins.c          # Built-in command implementations
│   ├── shell_state.c       # ShellState and JobTable operations
│   └── utils.c             # Helper functions (e.g., safe memory allocation)
├── include/
│   ├── shell.h             # Main header with all declarations
│   ├── executor.h          # **Executor function prototypes**
│   └── ...
└── Makefile
```

C. Infrastructure Starter Code

First, a helper function for safe string duplication (useful in many places):

```c
// utils.c

#include <stdlib.h>

#include <string.h>

#include <stdio.h>


char* shell_strdup(const char* str) {

    if (!str) return NULL;

    char* copy = malloc(strlen(str) + 1);

    if (!copy) {

        perror("malloc");

        exit(EXIT_FAILURE);

    }

    strcpy(copy, str);

    return copy;

}
```

D. Core Logic Skeleton Code

Now, the executor implementation skeletons with TODO comments mapping to algorithm steps:

```c
// executor.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

#include <errno.h>

#include <string.h>

#include "shell.h"

#include "executor.h"


// Helper: Set up all redirections for a single command

int setup_redirections(Command* cmd) {

    // TODO 1: Iterate through the linked list of Redirection structs in cmd->input_redir

    // TODO 2: For each redirection:

    //   - Determine target file descriptor based on redirection type:

    //     * REDIR_INPUT: target_fd = STDIN_FILENO (0)

    //     * REDIR_OUTPUT or REDIR_APPEND: target_fd = STDOUT_FILENO (1)

    //     * REDIR_ERROR: target_fd = STDERR_FILENO (2)

    // TODO 3: Open the file with appropriate flags:

    //   - REDIR_INPUT: O_RDONLY

    //   - REDIR_OUTPUT: O_WRONLY | O_CREAT | O_TRUNC, permissions 0644

    //   - REDIR_APPEND: O_WRONLY | O_CREAT | O_APPEND, permissions 0644

    //   - REDIR_ERROR: same as REDIR_OUTPUT/REDIR_APPEND based on type

    // TODO 4: Check for open error; if fails, perror and return -1

    // TODO 5: Use dup2 to copy the opened fd to target_fd

    // TODO 6: Close the original fd (from open) if it's different from target_fd

    // TODO 7: Continue with next redirection

    // TODO 8: Return 0 on success

    return 0;

}


// Execute a built-in command in the shell process

int execute_builtin(int argc, char** argv, ShellState* state) {
```

```
    // TODO 1: Check argc > 0, argv[0] is the command name
    // TODO 2: Compare argv[0] to known built-ins: "cd", "exit", "export", "pwd", "echo", "jobs", "fg", "bg"
    // TODO 3: Dispatch to appropriate built-in function (implemented in builtins.c)
    // TODO 4: For "cd": call chdir, update PWD environment variable
    // TODO 5: For "exit": set a flag in ShellState or call exit() with given status
    // TODO 6: For "export": putenv or setenv the variable
    // TODO 7: For others: implement accordingly
    // TODO 8: Return the exit status from the built-in (0 for success, non-zero for error)
    return 0;
}

// Execute a single external command (or first command in pipeline)
pid_t execute_external(Command* cmd, ShellState* state,
                       int in_fd, int out_fd, int err_fd,
                       int is_background, pid_t pgid) {
    // TODO 1: Fork a child process
    // TODO 2: In child process:
    //   - If pgid != 0, set process group: setpgid(0, pgid)
    //   - If in_fd != STDIN_FILENO, dup2(in_fd, STDIN_FILENO) and close(in_fd) if different
    //   - If out_fd != STDOUT_FILENO, dup2(out_fd, STDOUT_FILENO) and close(out_fd) if different
    //   - If err_fd != STDERR_FILENO, dup2(err_fd, STDERR_FILENO) and close(err_fd) if different
    //   - Call setup_redirections(cmd) to apply command-specific redirections
    //   - If any redirection fails, exit(EXIT_FAILURE)
    //   - Call execvp(cmd->argv[0], cmd->argv)
    //   - If execvp fails, perror and exit(EXIT_FAILURE)
    // TODO 3: In parent process: return child's PID
    return -1;
}

// Execute a pipeline of commands
int execute_pipeline(Command* pipeline, ShellState* state, int is_background) {
    // TODO 1: Count number of commands in the linked list (n_cmds)
    // TODO 2: Create (n_cmds - 1) pipes using pipe() or pipe2()
    // TODO 3: Initialize variables: pid_t child_pids[n_cmds]; pid_t pgid = 0;
    // TODO 4: For i from 0 to n_cmds-1:
```

```
    //   - Determine input fd: if i == 0, STDIN_FILENO else pipes[i-1][0]
    //   - Determine output fd: if i == n_cmds-1, STDOUT_FILENO else pipes[i][1]
    //   - Fork child
    //   - In child:
    //       if i == 0 and pgid == 0, setpgid to own pid, pgid = getpid()
    //       else setpgid to pgid
    //       execute_external with appropriate fds
    //   - In parent:
    //       if i == 0, set pgid = child's pid
    //       store child pid
    // TODO 5: In parent: Close ALL pipe ends (both read and write of all pipes)
    // TODO 6: If foreground (!is_background):
    //   - Give terminal control to pgid: tcsetpgrp(state->terminal_fd, pgid)
    //   - Wait for all children in process group: waitpid(-pgid, &status, 0) in loop
    //   - Take back terminal control: tcsetpgrp(state->terminal_fd, getpgrp())
    //   - Return exit status of last command
    // TODO 7: If background:
    //   - Add job to job table with pgid, command string, state JOB_RUNNING
    //   - Print job ID and PID to user
    //   - Return 0
    return 0;
}

// Main execution entry point
int execute_ast(Command* ast, ShellState* state) {
    // TODO 1: If ast is NULL, return 0 (success)
    // TODO 2: Check if first command is a built-in (compare ast->argv[0] to built-in list)
    // TODO 3: If built-in AND no pipes (ast->next == NULL):
    //   - Call execute_builtin(ast->argc, ast->argv, state)
    //   - Return its exit status
    // TODO 4: Determine if background: check last command's arguments for trailing "&"
    // TODO 5: If single external command (ast->next == NULL):
    //   - Fork and execute single child via execute_external
    //   - If foreground: wait for child, return status
```

```
    //   - If background: add to job table, return 0
    // TODO 6: If pipeline (ast->next != NULL):
    //   - Call execute_pipeline(ast, state, is_background)
    //   - Return its result
    return 0;
}
```

E. Language-Specific Hints

- **File Descriptor Constants**: Use `STDIN_FILENO` (0), `STDOUT_FILENO` (1), `STDERR_FILENO` (2) instead of magic numbers.
- **Error Checking**: Always check return values of system calls (`fork` , `pipe` , `dup2` , `open` , `execvp`). On error, `perror` gives descriptive messages.
- **Exit Status Extraction**: Use `WIFEXITED(status)` and `WEXITSTATUS(status)` macros after `waitpid` to get the child's exit code.
- **Signal Safety**: Avoid calling non-async-signal-safe functions (like `printf` , `malloc`) in signal handlers. In the `SIGCHLD` handler, just set a flag and check it in the main loop.
- **Atomic Operations**: For flags shared between signal handlers and main code, use `sig_atomic_t` type.

F. Milestone Checkpoint

After implementing the executor for Milestone 4 (pipes), test with these commands:

```bash
# In your shell
$ ls | grep .c
# Should list all .c files in current directory

$ cat /etc/passwd | cut -d: -f1 | head -5
# Should list first 5 usernames from passwd file

$ ls | wc -l > count.txt
# Should create count.txt with number of files

$ ls no-such-file 2> error.txt | grep foo
# Should create error.txt with "ls: cannot access..." error

$ sleep 5 | echo "immediate"
# Should print "immediate" immediately (sleep's stdin is closed, it exits)
```

Expected behavior: All commands work as in bash. No hanging processes. File descriptors aren't leaked (check with `lsof -p <shell_pid>`). Exit status reflects the last command's status.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Shell hangs after pipe command	Unclosed pipe file descriptors	Add debug prints showing which pipe ends are open in each process after fork. Use `ls -l /proc/<pid>/fd`.	Ensure parent closes ALL pipe ends after forking. Ensure each child closes ends it doesn't use.
Redirection overwrites pipe	Redirections applied after pipe setup	Print file descriptor table before/after `setup_redirections`.	Apply redirections after pipe connections in child, but note that later redirections override earlier ones.
Background job not reported	Job not added to job table or SIGCHLD not handled	Check `job_table_add` is called. Add debug to SIGCHLD handler.	Ensure job table is updated after forking background pipeline. Install SIGCHLD handler that calls `waitpid` with WNOHANG.
Ctrl+C kills shell instead of foreground command	Shell not setting foreground process group	Print process groups before/after `tcsetpgrp`.	Before waiting for foreground job, call `tcsetpgrp` to give terminal to child's process group. After job finishes, take it back.
"Command not found" even when PATH is set	`execvp` failing	Check `errno` after `execvp`. Print the resolved path.	Ensure `argv[0]` is not NULL. Ensure `execvp` is called, not `execlp`. Check PATH variable is passed to child (inherited).

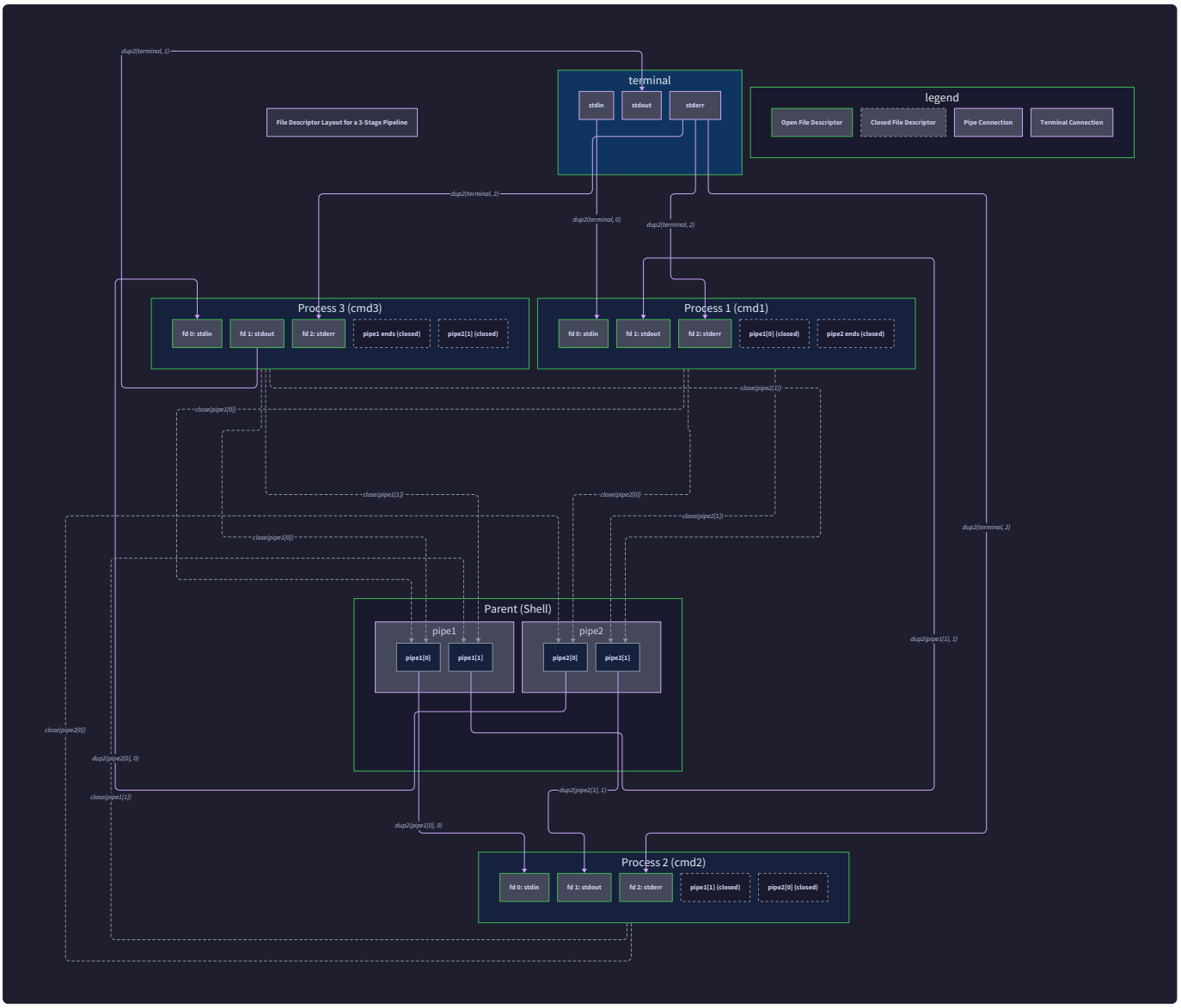


Diagram 2: Visual representation of how file descriptors are connected in a three-stage pipeline. Each child inherits all pipe ends but only keeps the ones it needs, closing others to avoid leaks.

Component Design: Job Control Manager

> **Milestone(s):** 5, 6 – This component manages background and suspended jobs, maintains the job table, and handles terminal signal forwarding for foreground process groups. It's the foundation for full job control functionality including background execution (`&`), job listing (`jobs`), and foreground/background transitions (`fg` , `bg` , Ctrl+Z).

Mental Model: A Restaurant Pager System

Imagine a popular restaurant during peak hours. When customers arrive and all tables are occupied, the host gives them a pager and adds their name to a waiting list. The customers can wander around (continue other activities) while waiting for their table to become available. When a table opens, the host activates the pager to notify the customers, who then return to be seated.

In this analogy:

Restaurant Element	Shell Equivalent	Purpose
Customers	Background jobs or suspended processes	Entities waiting for a resource (terminal attention) or completion
Host	Job Control Manager	Tracks all waiting entities and manages their state transitions
Pager	Job ID	Unique identifier that customers (users) use to reference a specific waiting group
Waiting List	Job table (`JobTable`)	Central registry of all background/suspended processes
Table (seated)	Terminal foreground access	Exclusive resource that allows interaction (receives keyboard signals)
"Your table is ready"	`fg` command	Moves a waiting job to the foreground (gives it terminal access)
"You can wait outside"	`bg` command	Moves a suspended job to background (continues execution without terminal access)
"The kitchen is closed"	SIGTERM/SIGKILL	External event that terminates a job

This mental model clarifies why we need job control: users want to launch multiple "tasks" (processes/pipelines) but only interact with one at a time through the terminal. The shell must track all launched jobs, manage which one currently "owns" the terminal, and provide mechanisms to move jobs between foreground (interactive) and background (non-interactive) states.

Interface: Job Table Operations

The Job Control Manager's primary interface is the job table—a dynamic collection that tracks all background and suspended jobs. The table supports CRUD operations and status queries through these well-defined functions:

Method Name	Parameters	Returns	Description
`job_table_init`	`JobTable* table`	`void`	Initializes an empty job table with default capacity
`job_table_add`	`JobTable* table`, `pid_t pgid`, `const char* command_string`, `Command* command_list`, `JobState initial_state`	`void`	Creates a new job entry with unique ID, stores process group ID, command string copy, and initial state
`job_table_find_by_id`	`JobTable* table`, `int job_id`	`Job*`	Returns pointer to job with matching ID, or `NULL` if not found
`job_table_find_by_pgid`	`JobTable* table`, `pid_t pgid`	`Job*`	Returns pointer to job with matching process group ID (for signal handlers)
`job_table_remove_at`	`JobTable* table`, `int index`	`void`	Removes job at specified table index, freeing its resources
`job_table_remove_by_id`	`JobTable* table`, `int job_id`	`bool`	Removes job with specified ID, returns `true` if found and removed
`job_table_list_all`	`JobTable* table`	`void`	Prints formatted list of all jobs (for `jobs` built-in)
`job_table_update_state`	`JobTable* table`, `pid_t pgid`, `JobState new_state`	`bool`	Updates job state based on process group ID (called from SIGCHLD handler)
`job_table_cleanup_completed`	`JobTable* table`	`void`	Removes all `JOB_DONE` entries from the table, freeing their resources

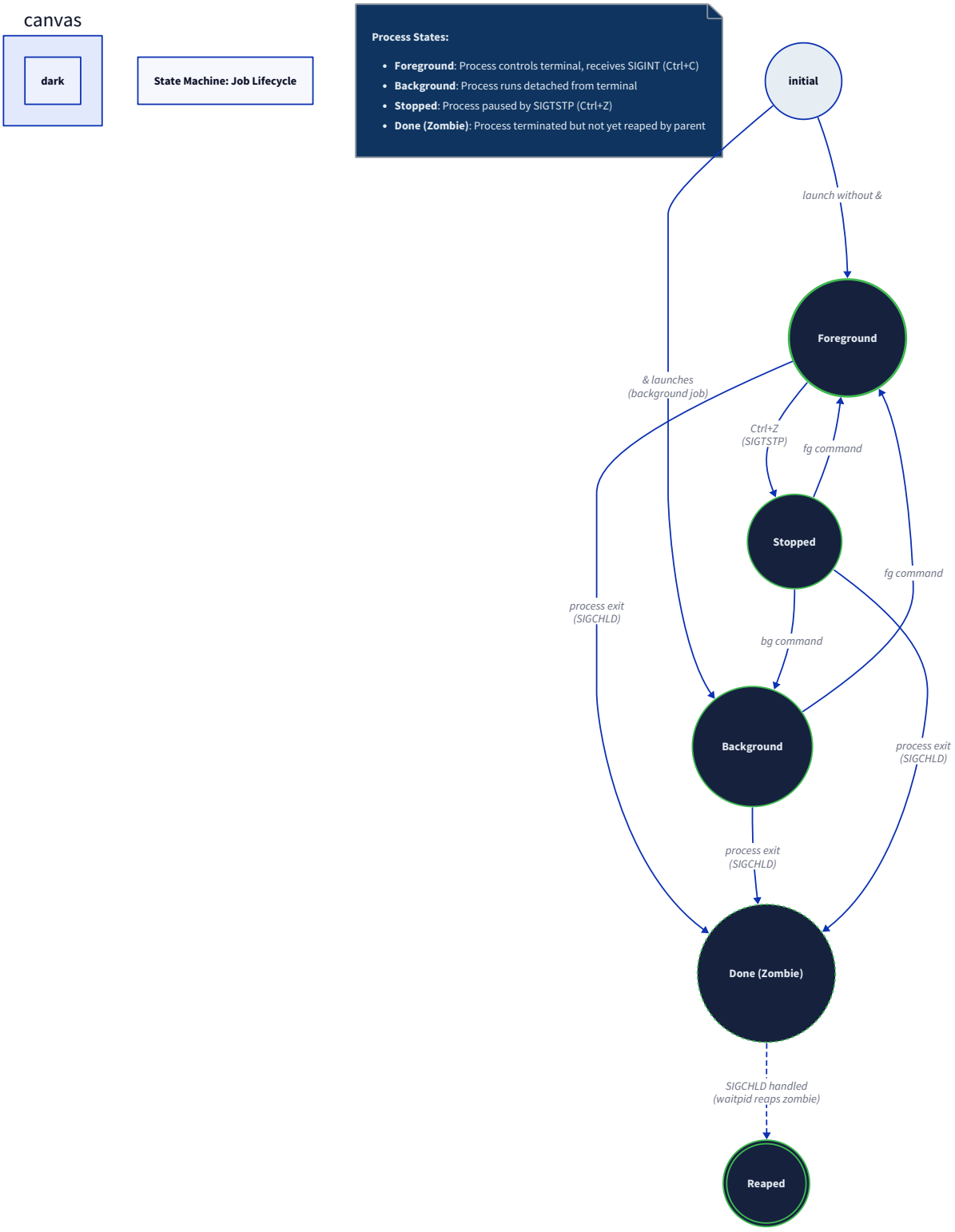
The job table is embedded within the `ShellState` struct and represents the shell's global view of all managed jobs. Each job entry contains:

Field Name	Type	Description
`id`	`int`	Unique sequential identifier (1, 2, 3...) shown to users
`pgid`	`pid_t`	Process group ID for the entire job (pipeline)
`command_string`	`char*`	Copy of the original command line string for display
`state`	`JobState`	Current state: `JOB_RUNNING`, `JOB_STOPPED`, or `JOB_DONE`
`command_list`	`Command*`	Pointer to the parsed command AST (for potential reuse in `fg` / `bg`)

> **Key Insight:** The job table is not just a passive registry—it's the central coordination point between three asynchronous actors: the user (typing commands), the shell (executing new jobs), and the operating system (delivering SIGCHLD signals when jobs change state). Proper synchronization (via careful signal handling and atomic operations) is crucial to avoid race conditions.

Job Lifecycle and State Management

Jobs transition through a well-defined state machine as they execute, suspend, resume, and terminate. Understanding these transitions is essential for implementing correct `jobs`, `fg`, `bg`, and Ctrl+Z behavior:



The complete state machine with all transitions, triggers, and actions:

Current State	Event	Next State	Actions Taken by Shell
(none)	User runs command *without* `&`	`JOB_RUNNING` (foreground)	1. Create process group with child's PID 2. Set terminal foreground process group to child's PGID 3. Wait for job completion with `waitpid(-pgid, ...)`
(none)	User runs command *with* `&`	`JOB_RUNNING` (background)	1. Create process group with child's PID 2. Job added to table with `JOB_RUNNING` state 3. Print job ID and PGID to user
`JOB_RUNNING` (foreground)	Child receives SIGTSTP (Ctrl+Z)	`JOB_STOPPED`	1. SIGCHLD handler detects stop with `WIFSTOPPED` 2. Update job state to `JOB_STOPPED` 3. Restore terminal foreground group to shell's PGID 4. Print suspension message with job ID
`JOB_RUNNING` (foreground)	Child exits (normally or by signal)	`JOB_DONE`	1. SIGCHLD handler reaps child with `waitpid` 2. Update job state to `JOB_DONE` 3. If foreground, collect and display exit status
`JOB_RUNNING` (background)	Child exits	`JOB_DONE`	1. SIGCHLD handler detects completion 2. Update job state to `JOB_DONE` 3. Mark job for cleanup at next prompt
`JOB_STOPPED`	User runs `fg %job_id`	`JOB_RUNNING` (foreground)	1. Send SIGCONT to job's process group 2. Set terminal foreground group to job's PGID 3. Wait for job completion with `waitpid`
`JOB_STOPPED`	User runs `bg %job_id`	`JOB_RUNNING` (background)	1. Send SIGCONT to job's process group 2. Update job state to `JOB_RUNNING` 3. Job continues execution without terminal access
`JOB_STOPPED`	User runs `kill -KILL %job_id`	`JOB_DONE`	1. Send SIGKILL to job's process group 2. SIGCHLD handler later reaps and marks as `JOB_DONE`
`JOB_DONE`	Shell displays next prompt	(removed)	1. `job_control_check_completed_jobs()` prints exit status 2. Job removed from table and memory freed

Concrete Example: A user runs `sleep 100 &` followed by `vim`. The shell creates job #1 (`sleep`) in `JOB_RUNNING` (background) state, then job #2 (`vim`) in `JOB_RUNNING` (foreground) state. When the user presses Ctrl+Z in vim:

1. The terminal sends SIGTSTP to the foreground process group (vim's PGID)
2. vim stops and the kernel sends SIGCHLD to the shell
3. The shell's SIGCHLD handler detects the stop, updates job #2 state to `JOB_STOPPED`
4. The shell restores itself as the terminal's foreground process group
5. The REPL prints `[2]+ Stopped vim` and displays a new prompt

Now the user can type `bg %2` to continue vim in background (though impractical for an interactive editor) or `fg %2` to bring it back to foreground.

ADR: Process Groups vs. PIDs for Job Control

> **Decision: Use Process Groups for Job Management Instead of Individual PIDs**
>
> - **Context**: When implementing job control, we need to manage entire pipelines (e.g., `ls | grep foo | wc -l`) as single units. The shell must be able to suspend/resume all processes in a pipeline simultaneously and forward terminal signals to the entire group. Using individual PIDs would require tracking and signaling each process separately, creating complexity and race conditions.
> - **Options Considered**:
>   1. **Individual PID Tracking**: Maintain a list of PIDs for each process in a job, iterate through them for operations
>   2. **Process Group with PGID**: Create a new process group for each job, use the PGID to control all processes
>   3. **Session-Based Groups**: Place all shell jobs in separate session (more isolation but breaks job control)
> - **Decision**: Use process groups with unique PGIDs for each job (pipeline). The first child in a pipeline calls `setpgid(0, 0)` to create a new process group, and subsequent children in the same pipeline join it.
> - **Rationale**:
>   - **Atomic Operations**: Signaling a process group (`kill(-pgid, sig)`) affects all members atomically, avoiding race conditions where some processes receive a signal and others don't
>   - **Terminal Integration**: The terminal driver delivers SIGINT (Ctrl+C) and SIGTSTP (Ctrl+Z) to the *foreground process group*, not individual PIDs. By setting the terminal's foreground group to our job's PGID, we get correct signal forwarding "for free"
>   - **Pipeline Consistency**: All processes in a pipeline share stdin/stdout through pipes; they should be controlled as a unit since suspending one without the others could deadlock the pipe
>   - **Unix Convention**: This matches how bash, zsh, and other shells implement job control, ensuring predictable behavior
> - **Consequences**:
>   - **Positive**: Simplified signal forwarding, correct terminal behavior, race-free job control
>   - **Negative**: Must carefully manage process group creation (potential race between parent setting PGID and child exec'ing)
>   - **Complexity**: Need to handle orphaned process groups when shell terminates unexpectedly

The comparison table highlights the trade-offs:

Option	Pros	Cons	Why Not Chosen
Individual PIDs	Simpler to implement initially; No need for `setpgid` calls	Race conditions when signaling; Manual iteration for pipelines; Incorrect terminal signal delivery	Fails for pipelines and causes inconsistent behavior
Process Groups (PGID)	Atomic group operations; Correct terminal integration; Standard Unix approach	Slightly more complex setup; Must handle `setpgid` race	**CHOSEN** – Standard, correct, race-free
Session-Based	Complete isolation between jobs; Can survive shell exit	Breaks job control (can't bring back to foreground); Complex session management	Over-isolation breaks required functionality

The process group approach requires careful coordination: after forking each child in a pipeline, the parent should call `setpgid(child_pid, pgid)` to ensure the child joins the correct group, and the child should also call `setpgid(0, pgid)` as a defensive measure (in case parent's call hasn't executed yet due to scheduling). The first process in the pipeline determines the PGID (typically its own PID).

Common Pitfalls

Job control involves subtle interactions between processes, signals, and terminal settings. These common mistakes can lead to hangs, zombie processes, or incorrect behavior:

⚠️ Pitfall: Orphaned Process Groups Causing Unstoppable Jobs

- **Description**: When the shell terminates (crashes or is killed) while background jobs are running, those jobs become "orphaned" — their parent process no longer exists. If they're also in their own process group (not the session leader's group), they become an *orphaned process group*. According to POSIX, when an orphaned process group stopped by SIGTSTP receives SIGCONT, it must first receive SIGHUP (termination signal), which kills the job unexpectedly.
- **Why It's Wrong**: This violates user expectations — background jobs should continue running even if the shell exits (unless specifically configured otherwise with `shopt -s huponexit` in bash).
- **Fix**: Before exiting, the shell should send SIGHUP to all its process groups (except its own). Alternatively, design the shell to avoid creating orphaned process groups by not setting the `setpgid` flag, though this breaks proper job control. The practical solution: document this as expected Unix behavior.

⚠️ Pitfall: Race Condition in SIGCHLD Handler

- **Description**: The SIGCHLD handler uses `waitpid(-1, &status, WNOHANG | WUNTRACED)` in a loop to reap all terminated/stopped children. If the handler calls `printf()` or other non-async-signal-safe functions, it could deadlock or corrupt memory when called during another `printf()` in the main REPL loop.
- **Why It's Wrong**: POSIX specifies only a small set of async-signal-safe functions that can be safely called from signal handlers. Using unsafe functions causes undefined behavior that may manifest as intermittent crashes or hangs.
- **Fix**: In the SIGCHLD handler, only:
  1. Use `waitpid()` with `WNOHANG` in a loop
  2. Update a volatile sig_atomic_t flag or write to a pipe
  3. Let the main REPL loop check the flag/read the pipe and perform non-safe operations like `printf()` and job table updates

⚠️ Pitfall: Incorrect Terminal Foreground Group Management

- **Description**: Forgetting to call `tcsetpgrp(STDIN_FILENO, pgid)` before waiting for a foreground job, or failing to restore the shell's own process group with `tcsetpgrp(STDIN_FILENO, getpgrp())` after the job stops/exits.
- **Why It's Wrong**: Terminal-generated signals (Ctrl+C, Ctrl+Z) are delivered to the *foreground process group*. If the shell doesn't properly transfer control, signals go to the wrong group (shell receives SIGINT instead of the child, or vice versa).
- **Fix**: Follow this strict protocol:
  1. Before launching a foreground job: `tcsetpgrp(STDIN_FILENO, job_pgid)`
  2. After job stops/completes: `tcsetpgrp(STDIN_FILENO, shell_pgid)`
  3. Always check `tcsetpgrp()` return value — it can fail if the shell is not in the controlling terminal's session (e.g., running in background itself)

⚠️ Pitfall: Not Closing All Inherited File Descriptors in Children

- **Description**: When forking children for background jobs, the child inherits all open file descriptors from the parent, including pipe ends from previous commands, open redirection files, etc. These remain open in the background job, potentially causing resource leaks or unexpected behavior (like a background job keeping a pipe open that the parent thinks is closed).
- **Why It's Wrong**: File descriptors are a finite resource (typically 1024 per process). Background jobs that inherit unnecessary FDs can exhaust system limits. Also, pipes won't close properly, causing writers to not receive EPIPE when readers terminate.
- **Fix**: After forking but before exec'ing in a background job, close all non-essential file descriptors. Either:
  - Track which FDs need to stay open (stdin/stdout/stderr plus any redirections)
  - Close all FDs above a certain number (e.g., > 2) using `close_range()` (Linux) or iterative `close()`
  - Use `fcntl(fd, F_SETFD, FD_CLOEXEC)` on pipes before forking so they close automatically on exec

⚠️ Pitfall: Job ID vs Process Group ID Confusion

- **Description**: Using the process group ID (PGID) in user-facing commands (`fg 12345`) instead of the sequential job ID (`fg %1`), or mixing them up internally.
- **Why It's Wrong**: PGIDs are large, unpredictable numbers (like 28457) while job IDs are small sequential numbers (1, 2, 3). Users expect the simpler job ID syntax. Internally, storing PGIDs but displaying job IDs requires maintaining a mapping.
- **Fix**: Consistently use:
  - **Job ID** (1, 2, 3): For user interaction, displayed by `jobs` command, accepted by `fg %1` syntax

- **PGID**: For internal operations (`kill(-pgid, sig)` , `tcsetpgrp(pgid)` , `waitpid(-pgid)`)
- Store both in the `Job` struct and provide lookup functions for conversion

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Job Table Storage	Dynamic array in `ShellState`	Hash table keyed by PGID with LRU eviction
Signal Handling	Self-pipe trick with `pselect()`	Signalfd (Linux) or kqueue (BSD) for synchronous signal handling
Terminal Control	`tcsetpgrp()` / `tcgetpgrp()`	Full termios manipulation for raw mode input
Async-Safe Communication	Volatile flag checked in REPL	Pipe write in handler, read in main loop with `pselect()`
Process Group Creation	`setpgid()` in parent after fork	`setpgid()` in child before exec to avoid race

B. Recommended File/Module Structure

```
gauntlet_shell/
├── src/
│   ├── main.c                    # Entry point, shell_state_init, main()
│   ├── repl.c                    # run_repl_loop() and prompt display
│   ├── parser.c                  # parse_command_line() and tokenization
│   ├── executor.c                # execute_ast(), execute_pipeline(), builtins
│   ├── job_control.c             # Job Control Manager (THIS COMPONENT)
│   ├── job_control.h             # JobTable, Job, JobState declarations
│   ├── redirections.c            # setup_redirections() helper
│   ├── signals.c                 # Signal handler setup and safe wrappers
│   └── utils.c                   # String utilities, memory helpers
├── include/
│   └── shell.h                   # Common structs, constants, typedefs
└── Makefile
```

C. Infrastructure Starter Code (COMPLETE, ready to use)

File: `src/signals.c` – Safe signal handling infrastructure

```c
#include "shell.h"
#include <signal.h>
#include <unistd.h>
#include <errno.h>

static volatile sig_atomic_t got_sigchld = 0;
static int signal_pipe[2] = {-1, -1};

/* Async-signal-safe write wrapper */
static void safe_write(int fd, const void *buf, size_t count) {
    const char *b = buf;
    while (count > 0) {
        ssize_t written = write(fd, b, count);
        if (written < 0 && errno == EINTR) continue;
        if (written <= 0) break;
        b += written;
        count -= written;
    }
}

/* SIGCHLD handler - only does minimal work */
void sigchld_handler(int sig) {
    (void)sig;  // Unused parameter
    got_sigchld = 1;
    // Optional: write to pipe for main loop notification
    if (signal_pipe[1] != -1) {
        char ch = 'C';  // 'C' for child status change
        safe_write(signal_pipe[1], &ch, 1);
    }
}

/* SIGINT handler for foreground job signal forwarding */
void sigint_handler(int sig) {
    (void)sig;
    // This handler only used when shell is foreground with no child.
    // When child is foreground, terminal sends SIGINT directly to child's process group.
```

```c
}

/* Setup all signal handlers */
void setup_signal_handlers(void) {
    struct sigaction sa;

    // Create pipe for signal notification
    if (pipe(signal_pipe) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    // Set pipe ends to non-blocking
    for (int i = 0; i < 2; i++) {
        int flags = fcntl(signal_pipe[i], F_GETFL);
        fcntl(signal_pipe[i], F_SETFL, flags | O_NONBLOCK);
        fcntl(signal_pipe[i], F_SETFD, FD_CLOEXEC);
    }

    // SIGCHLD - for reaping children
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
    sigaction(SIGCHLD, &sa, NULL);

    // SIGINT - ignore in shell (forwarded to foreground job)
    sa.sa_handler = SIG_IGN;
    sigaction(SIGINT, &sa, NULL);

    // SIGTSTP - ignore in shell (forwarded to foreground job)
    sa.sa_handler = SIG_IGN;
    sigaction(SIGTSTP, &sa, NULL);

    // SIGTTIN/SIGTTOU - background jobs trying to read/write terminal
    sa.sa_handler = SIG_IGN;
```

```
    sigaction(SIGTTIN, &sa, NULL);
    sigaction(SIGTTOU, &sa, NULL);
}

/* Get read end of signal pipe for pselect() */
int get_signal_pipe_readfd(void) {
    return signal_pipe[0];
}
```

File: `include/shell.h` – Job control type definitions (partial)

```c
#ifndef SHELL_H

#define SHELL_H


#include <stdbool.h>

#include <sys/types.h>


#define MAX_JOBS 100

#define MAX_LINE_LEN 4096


typedef enum {

    REDIR_INPUT,    // <

    REDIR_OUTPUT,   // >

    REDIR_APPEND,   // >>

    REDIR_ERROR     // 2>

} RedirType;


typedef enum {

    JOB_RUNNING,

    JOB_STOPPED,

    JOB_DONE

} JobState;


typedef struct Redirection {

    RedirType type;

    char* filename;

    int fd;  // Target fd (e.g., 2 for stderr redirect)

    struct Redirection* next;

} Redirection;


typedef struct Command {

    char** argv;

    int argc;

    Redirection* input_redir;

    Redirection* output_redir;

    struct Command* next;  // For pipelines

} Command;
```

```c
typedef struct Job {
    int id;                     // Sequential job number (1, 2, 3...)
    pid_t pgid;                 // Process group ID
    char* command_string;       // Original command line
    JobState state;             // Current state
    Command* command_list;      // Parsed command AST (optional)
} Job;

typedef struct JobTable {
    Job** jobs;                 // Array of pointers to Jobs
    int capacity;               // Allocated size
    int count;                  // Current number of jobs
    int next_job_id;            // Next job ID to assign
} JobTable;

typedef struct ShellState {
    JobTable job_table;         // Background/suspended jobs
    pid_t foreground_pgid;      // PGID of current foreground job (0 if none)
    int terminal_fd;            // Usually STDIN_FILENO
    int last_exit_status;       // Exit status of last foreground command
    int interactive;            // Non-zero if shell is interactive
} ShellState;

// Job control function prototypes
void job_table_init(JobTable* table);
void job_table_add(JobTable* table, pid_t pgid, const char* command_string,
                   Command* command_list, JobState initial_state);
Job* job_table_find_by_id(JobTable* table, int job_id);
Job* job_table_find_by_pgid(JobTable* table, pid_t pgid);
void job_table_remove_at(JobTable* table, int index);
bool job_table_remove_by_id(JobTable* table, int job_id);
void job_table_list_all(JobTable* table);
bool job_table_update_state(JobTable* table, pid_t pgid, JobState new_state);
void job_table_cleanup_completed(JobTable* table);
void job_control_check_completed_jobs(ShellState* state);
```

```
#endif // SHELL_H
```

D. Core Logic Skeleton Code

File: `src/job_control.c` – Job table implementation skeleton

```c
#include "shell.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>


/* Initialize empty job table with default capacity */

void job_table_init(JobTable* table) {

    // TODO 1: Allocate initial array of Job pointers (capacity = 16)

    // TODO 2: Set count = 0, next_job_id = 1

    // TODO 3: Initialize all pointers to NULL

}


/* Add new job to table, assigning next available job ID */

void job_table_add(JobTable* table, pid_t pgid, const char* command_string,

                   Command* command_list, JobState initial_state) {

    // TODO 1: Check if table needs resizing (count == capacity)

    // TODO 2: If resizing needed, double capacity and realloc

    // TODO 3: Allocate new Job struct

    // TODO 4: Set fields: id = table->next_job_id++, pgid, state = initial_state

    // TODO 5: Duplicate command_string with strdup

    // TODO 6: Store command_list (caller retains ownership)

    // TODO 7: Add to array at index count, increment count

    // TODO 8: Print job notification: "[id] pgid" if background

}


/* Find job by job ID (1, 2, 3...) */

Job* job_table_find_by_id(JobTable* table, int job_id) {

    // TODO 1: Linear search through table->jobs[0..count-1]

    // TODO 2: Compare each job->id with job_id

    // TODO 3: Return pointer if found, NULL otherwise

    return NULL;

}


/* Find job by process group ID (for SIGCHLD handler) */
```

```c
Job* job_table_find_by_pgid(JobTable* table, pid_t pgid) {
    // TODO 1: Linear search through table->jobs[0..count-1]
    // TODO 2: Compare each job->pgid with pgid
    // TODO 3: Return pointer if found, NULL otherwise
    return NULL;
}

/* Update job state based on PGID (called from SIGCHLD handler context) */
bool job_table_update_state(JobTable* table, pid_t pgid, JobState new_state) {
    // TODO 1: Find job using job_table_find_by_pgid
    // TODO 2: If found, update job->state = new_state
    // TODO 3: Return true if updated, false if not found
    return false;
}

/* Check for completed jobs and print notifications */
void job_control_check_completed_jobs(ShellState* state) {
    // TODO 1: Iterate through all jobs in table
    // TODO 2: For each job with state == JOB_DONE:
    //   a) Print completion message: "[id]+ Done command_string"
    //   b) Remove job from table using job_table_remove_at
    //   c) Free job->command_string and job struct
    // TODO 3: Skip jobs with state == JOB_RUNNING or JOB_STOPPED
}

/* Built-in jobs command implementation */
int builtin_jobs(int argc, char** argv, ShellState* state) {
    (void)argc; (void)argv;  // Unused parameters
    // TODO 1: Call job_table_list_all(&state->job_table)
    // TODO 2: Format output like: "[1] Running sleep 100 &"
    // TODO 3: Show stopped jobs with "Stopped" instead of "Running"
    // TODO 4: Return 0 for success
    return 0;
}

/* Built-in fg command implementation */
```

```c
int builtin_fg(int argc, char** argv, ShellState* state) {
    // TODO 1: Parse job spec: "fg %1" or "fg 1" or "fg" (defaults to most recent job)
    // TODO 2: Find job in table using job_table_find_by_id
    // TODO 3: If job not found, print error and return 1
    // TODO 4: Send SIGCONT to job->pgid: kill(-job->pgid, SIGCONT)
    // TODO 5: Set terminal foreground group to job->pgid: tcsetpgrp(state->terminal_fd, job->pgid)
    // TODO 6: Update job state to JOB_RUNNING
    // TODO 7: Wait for job completion with waitpid(-job->pgid, &status, WUNTRACED)
    // TODO 8: If job stopped (WIFSTOPPED), update state to JOB_STOPPED
    // TODO 9: Restore terminal to shell: tcsetpgrp(state->terminal_fd, getpgrp())
    // TODO 10: Return appropriate exit status
    return 0;
}

/* Built-in bg command implementation */
int builtin_bg(int argc, char** argv, ShellState* state) {
    // TODO 1: Parse job spec similar to fg
    // TODO 2: Find job in table (must be in JOB_STOPPED state)
    // TODO 3: Send SIGCONT to job->pgid: kill(-job->pgid, SIGCONT)
    // TODO 4: Update job state to JOB_RUNNING
    // TODO 5: Print message: "[id] command_string &"
    // TODO 6: Return 0 for success
    return 0;
}
```

File: `src/executor.c` – Modified to support job control (partial additions)

```c
#include "shell.h"

/* Modified execute_pipeline to handle background jobs */
int execute_pipeline(Command* pipeline, ShellState* state, int is_background) {
    // ... (previous pipe setup code from earlier milestone) ...

    // After creating all processes in the pipeline:
    if (is_background) {
        // TODO 1: Create process group using first child's PID as PGID
        // TODO 2: Add job to job table with JOB_RUNNING state
        // TODO 3: Print job notification: "[job_id] pgid"
        // TODO 4: Close all pipe FDs in parent (already done)
        // TODO 5: Return immediately (don't wait for children)
    } else {
        // TODO 1: Create process group for foreground pipeline
        // TODO 2: Set terminal foreground group to pipeline's PGID
        // TODO 3: Wait for entire pipeline with waitpid(-pgid, &status, WUNTRACED)
        // TODO 4: If pipeline stopped (WIFSTOPPED), add to job table with JOB_STOPPED state
        // TODO 5: Restore terminal foreground group to shell
        // TODO 6: Return pipeline exit status
    }
    return 0;
}
```

E. Language-Specific Hints

- **Signal Safety**: Never call `printf()`, `malloc()`, or most library functions from signal handlers. Use `write()` for output, or set a volatile flag and handle in main loop.
- **Atomic Operations**: Use `sig_atomic_t` for flags shared between signal handlers and main code. Compilers guarantee reads/writes are atomic.
- **Process Groups**: Use negative PID with `kill()` to signal entire process group: `kill(-pgid, SIGCONT)`.
- **Terminal Control**: Always check return values of `tcsetpgrp()` — it can fail (returns -1, errno = ENOTTY) if shell is not in controlling terminal (e.g., shell running in background itself).
- **waitpid Flags**: Use `WUNTRACED` to detect stopped children, `WNOHANG` for non-blocking checks, and `WCONTINUED` to detect resumed children (optional).
- **Job ID Parsing**: Support both `fg 1` and `fg %1` syntax. The percent sign is optional but conventional.

F. Milestone Checkpoint

After implementing Milestone 5 (Background Jobs):

1. Run the shell and test: `sleep 10 &`

- Expected: Immediate prompt return with message like `[1] 12345`

2. Type `jobs` immediately after
   - Expected: `[1] Running sleep 10 &`
3. Wait 10+ seconds, then press Enter (new prompt)
   - Expected: `[1]+ Done sleep 10`
4. Test pipeline in background: `ls -l | grep .c &`
   - Expected: Job notification, both processes run in background

After implementing Milestone 6 (Job Control):

1. Start an interactive foreground job: `sleep 100` (no `&`)
2. Press Ctrl+Z while it's running
   - Expected: `^Z [1]+ Stopped sleep 100` and new prompt
3. Type `jobs`
   - Expected: `[1]+ Stopped sleep 100`
4. Type `bg %1`
   - Expected: `[1] sleep 100 &` and job continues in background
5. Type `fg %1`
   - Expected: Job returns to foreground (terminal may wait)
6. Press Ctrl+C to interrupt
   - Expected: `^C` and immediate prompt return (not shell exit)

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Background jobs not appearing in `jobs` output	Job not added to table, or SIGCHLD reaping too quickly	Add debug prints in `job_table_add()` and SIGCHLD handler; check `is_background` flag in executor	Ensure `execute_pipeline()` calls `job_table_add()` for background jobs before returning
Ctrl+Z doesn't stop foreground job	Terminal foreground group not set correctly	Print shell PGID and child PGID before/after `tcsetpgrp()`; check with `ps -o pid,pgid,tpgid,command`	Call `tcsetpgrp(STDIN_FILENO, child_pgid)` before `waitpid()`
Shell hangs after `fg` command	Parent waiting for wrong process group, or pipe FDs not closed	Add debug prints showing which PGID `waitpid()` is waiting for; check open FDs with `lsof -p PID`	Ensure `waitpid(-pgid, ...)` uses negative PGID; close all pipe ends in parent after forking
Zombie processes accumulate	SIGCHLD handler not reaping, or not using `WNOHANG`	Check if SIGCHLD handler installed; add print in handler; use `ps aux	grep defunct`	
`bg` command doesn't resume job	Job not in `JOB_STOPPED` state, or SIGCONT not delivered	Print job state before sending SIGCONT; check with `kill -CONT -PGID` manually	Ensure `job_table_update_state()` called from SIGCHLD handler when `WIFSTOPPED()` is true
Terminal input/output messed up after foreground job	Terminal modes not restored, foreground group not returned to shell	Save/restore termios with `tcgetattr()` / `tcsetattr()`; ensure `tcsetpgrp()` to shell PGID in all exit paths	Always restore terminal foreground group in both normal exit and signal handler paths

Interactions and Data Flow

> **Milestone(s):** 3, 4, 5, 6 – This section synthesizes all previously designed components, tracing the complete lifecycle of a complex command. It demonstrates how the REPL, Parser, Executor, and Job Control Manager collaborate to transform a textual command into parallel processes with coordinated I/O and signal handling. The interactions are most complex for pipelines with redirection and background execution.

Understanding how components interact is crucial for debugging and extending the shell. This section provides a concrete, step-by-step narrative of what happens internally when a user enters a command like `ls | grep foo > out.txt &`. It also formalizes the communication patterns between components, clarifying how data and control flow through the system.

Sequence of Operations for a Pipeline

Imagine the shell as a **factory assembly line**. A raw command string enters at one end. At each workstation (component), specialized workers transform it: the **Parser** breaks it into parts and builds a blueprint (AST), the **Executor** uses the blueprint to construct and activate machinery (processes), and the **Job Control Manager** monitors and manages the running machines. The **REPL** is the foreman, coordinating the workflow and reporting status back to the user. The following walkthrough shows the assembly line in action for a non-trivial command.

We trace the execution of `ls | grep foo > out.txt &`, which combines a pipeline, output redirection, and background execution.

Initial State: The shell is at the prompt, having just completed `job_control_check_completed_jobs`. The `ShellState` contains an empty foreground process group ID (`foreground_pgid = 0`) and a `JobTable` with no active jobs. The terminal's foreground process group is the shell's own PID.

1. **REPL: Input Reading & Prompt**
   - The `run_repl_loop` function displays the prompt (`$ `). It calls a safe input function (e.g., using `pselect` to allow signal interruption) to read a line. The user types `ls | grep foo > out.txt &` and presses Enter. The REPL receives the raw string, strips the trailing newline, and notes the presence of a trailing `&` character (which it may strip and set a flag, or leave for the parser).
2. **REPL to Parser: Command String Transformation**
   - The REPL calls `parse_command_line("ls | grep foo > out.txt &", state)`. It passes the raw string and a pointer to the `ShellState` (which may be needed for variable expansion in future extensions, though not in our example).
3. **Parser: Tokenization and AST Construction**
   - The parser tokenizes the input, producing tokens: `["ls", "|", "grep", "foo", ">", "out.txt", "&"]`. It identifies `&` as a background operator and removes it from the parse stream, recording the job should run in the background.
   - Using recursive descent, it builds an **Abstract Syntax Tree (AST)**. The AST is a linked list of `Command` nodes:
     - **Node 1 (`ls`):** `argv = ["ls", NULL]`, `argc=1`. `output_redir = NULL`, `input_redir = NULL`. `next` points to Node 2.
     - **Node 2 (`grep foo`):** `argv = ["grep", "foo", NULL]`, `argc=2`. A `Redirection` struct is attached: `output_redir->type = REDIR_OUTPUT`, `filename = "out.txt"`. `input_redir = NULL`. `next = NULL`.
   - The parser returns a `ParseResult` containing this AST and a `success = true`.
4. **REPL to Executor: AST Execution with Background Flag**
   - The REPL, noting the original `&` (either from its own flag or by inspecting the parse result), calls `execute_ast(ast, state)` **without waiting**. In practice, `execute_ast` will itself check if the job is to be run in the background and handle process group and terminal control accordingly.
5. **Executor: Pipeline Setup & Job Creation**

- `execute_ast` identifies a pipeline (a linked list with more than one `Command` node). It calls `execute_pipeline(ast, state, is_background)`.
- `execute_pipeline` performs the following orchestration:
  - **a. Pipe Creation:** It creates **one pipe** (since we have two commands). `pipe_fd[0]` is the read end, `pipe_fd[1]` is the write end.
  - **b. Process Group Creation:** Before forking, it determines if a new process group is needed (always for background jobs, or for foreground pipelines). It will create a new PGID using the PID of the first child.
  - **c. First Child Fork (`ls`):**
    - `fork()` creates Child1. Child1 sets its process group ID (`setpgid(0, pgid)`) and, if it's a foreground job, gives the terminal control to its group (`tcsetpgrp`).
    - Child1 sets up I/O: Its `stdout` (FD 1) needs to go to the pipe. It calls `dup2(pipe_fd[1], STDOUT_FILENO)` and then closes both `pipe_fd[0]` and `pipe_fd[1]`.
    - Child1 has no other redirections. It calls `execvp("ls", ["ls", NULL])`.
  - **d. Parent Actions After First Fork:** The parent (shell) closes `pipe_fd[1]` (the write end) in its own descriptor table, as it will not be used by the shell or the next child. The read end (`pipe_fd[0]`) is left open to be passed to the next child.
  - **e. Second Child Fork (`grep foo > out.txt`):**
    - `fork()` creates Child2. Child2 joins the same process group (`setpgid(0, pgid)`).
    - Child2 sets up I/O: Its `stdin` (FD 0) needs to come from the pipe: `dup2(pipe_fd[0], STDIN_FILENO)`. It then closes `pipe_fd[0]`.
    - Child2 processes the `output_redir` attached to its `Command` node. It opens the file `"out.txt"` with `O_WRONLY|O_CREAT|O_TRUNC`, and `dup2`s that file descriptor to `STDOUT_FILENO`.
    - Child2 calls `execvp("grep", ["grep", "foo", NULL])`.
  - **f. Parent Cleanup:** After forking the last child, the parent shell closes its remaining copy of `pipe_fd[0]`. All pipe ends are now closed in the parent.
  - **g. Job Registration:** The parent shell calls `job_table_add` with the new PGID, the original command string `"ls | grep foo > out.txt"`, the AST (or a copy), and an initial state of `JOB_RUNNING`. The job is added to the background list because of the `&`.
  - **h. Non-Blocking Wait:** Since it's a background job, `execute_pipeline` does **not** call `waitpid`. It returns immediately with an exit status of 0 (or perhaps the job ID). The shell's `foreground_pgid` remains 0 (or is restored to itself).

6. **REPL: Immediate Prompt Return**
   - Control returns to `run_repl_loop`. Because the job was launched in the background, the REPL immediately displays the prompt again (`$ `) without waiting. The `ls` and `grep` processes are now running concurrently, managed by the kernel, with their I/O connected via the pipe and to the file `out.txt`.
7. **Asynchronous Signal: Job Completion**
   - Sometime later, the `grep` process (the last in the pipeline) exits. The kernel sends a `SIGCHLD` signal to the shell (the parent).
   - The shell's `SIGCHLD` handler (installed at startup) is invoked. It must be careful to only call async-signal-safe functions. A typical pattern is to record the PID and exit status in a global volatile variable or a self-pipe.
   - In our design, the handler might call `waitpid(-1, &status, WNOHANG | WUNTRACED)` in a loop to reap all changed children. When it finds a child from our job's process group, it calls `job_table_update_state(state->job_table, pgid, JOB_DONE)`.
8. **REPL: Job Status Reporting**
   - At the **next iteration** of the REPL, just before printing the prompt, `run_repl_loop` calls `job_control_check_completed_jobs(state)`. This function scans the `JobTable` for jobs in the `JOB_DONE` state, prints a notification (e.g., `[1]+ Done ls | grep foo > out.txt`), and removes them from the table via `job_table_remove_at`.

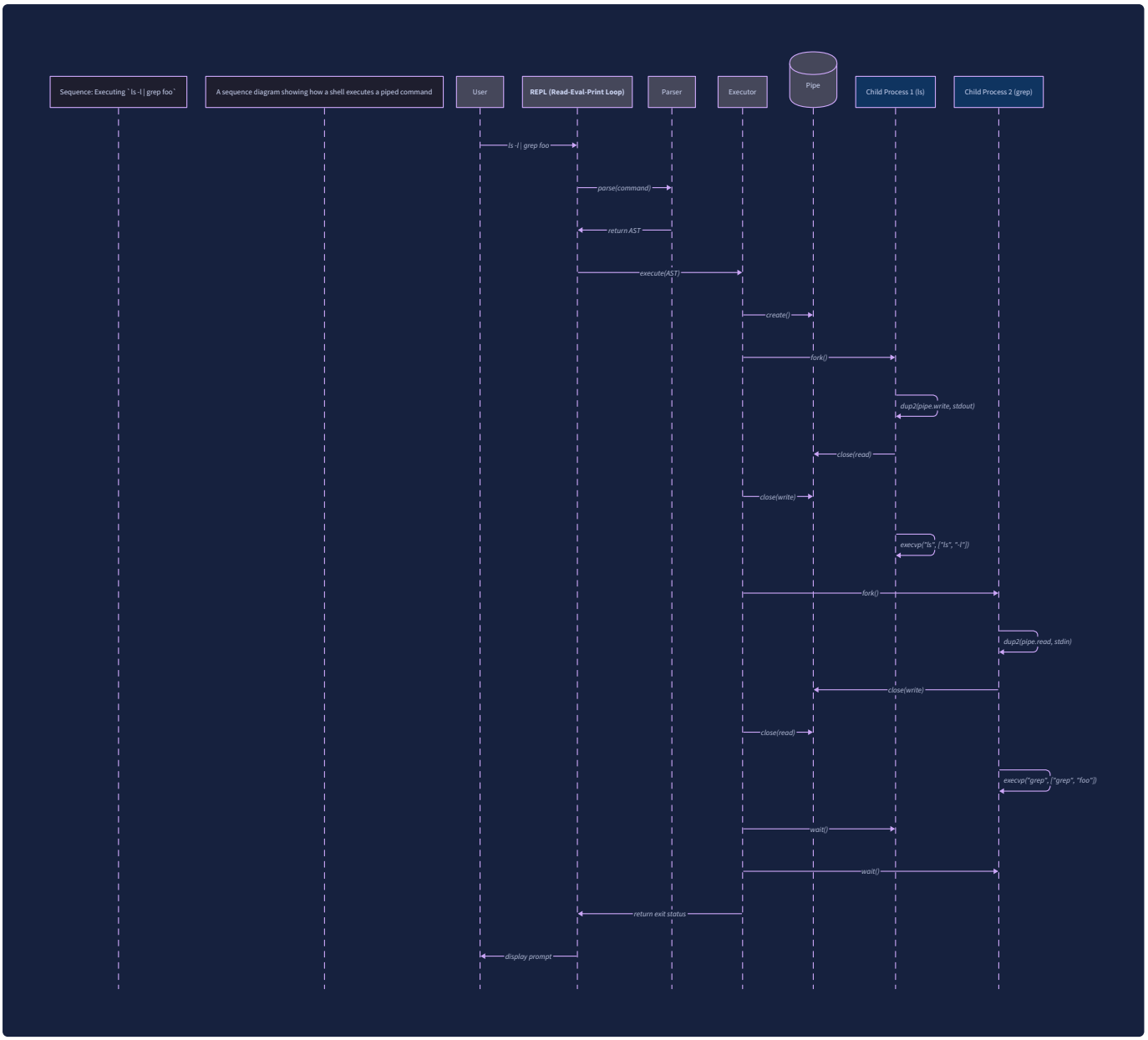


Diagram: The sequence of function calls and system calls for a simpler pipeline. Our example adds redirection and backgrounding, but the core flow of REPL→Parser→Executor (fork, dup2, exec) remains identical.

Internal Message and Event Flow

Component interactions follow three primary patterns: **synchronous function calls** for the main command flow, **shared data structure** manipulation for state management, and **asynchronous signal** handling for process lifecycle events. The table below catalogs these interactions.

Synchronous Function Call Flow

Caller	Callee	Function/Event	Data Passed	Return Value / Effect
`main()`	REPL	`run_repl_loop(state)`	Initialized `ShellState`	Exit status when shell terminates
REPL	Parser	`parse_command_line(line, state)`	Raw input string	`ParseResult` (AST or error)
REPL	Executor	`execute_ast(command_ast, state)`	Command AST	Exit status of (foreground) command
`execute_ast`	Built-in	`execute_builtin(argc, argv, state)`	Tokenized arguments	Built-in command's exit status
`execute_ast`	External	`execute_pipeline(ast, state, bg)`	Command AST, background flag	Exit status (or 0 for background launch)
`execute_pipeline`	System	`fork()`, `pipe()`, `dup2()`	-	New processes, file descriptors
`execute_pipeline`	Job Control	`job_table_add(...)`	PGID, command string, AST	Job added to table
`builtin_fg`	Job Control	`job_table_find_by_id(...)`	Job ID	Pointer to `Job` struct
`builtin_fg`	System	`tcsetpgrp()`, `kill(-pgid, SIGCONT)`	Process group ID	Terminal control transferred, job resumed
REPL	Job Control	`job_control_check_completed_jobs(state)`	-	Prints notifications, cleans table

Shared Data Structure Access

Data Structure	Primary Writer	Primary Readers	Synchronization Needed
`ShellState` (especially `job_table`)	REPL, Executor, Job Control functions, Signal Handlers	REPL, Parser, Executor, Built-in commands	Critical. Signal handlers and main loop can interleave. Use `sig_atomic_t` for flags, disable signals during updates with `sigprocmask`.
`JobTable` (`jobs` array)	`job_table_add`, `job_table_update_state` (from SIGCHLD), `job_table_remove_at`	`builtin_jobs`, `builtin_fg`, `builtin_bg`, `job_control_check_completed_jobs`	Same as above. The table is a shared resource between the main loop and asynchronous signal handlers.
Command AST (`Command*`)	Parser (`parse_command_line`)	Executor (`execute_ast`, `execute_pipeline`)	The AST is owned by the REPL for a single command iteration. It is passed to the executor, which may free it (`free_command_ast`) after execution. No concurrent access during a single command.

Asynchronous Signal Event Flow

Signal	Generated By	Handler in Shell	Handler's Typical Action
`SIGCHLD`	Kernel when child changes state (exits, stops)	`sigchld_handler`	Loop with `waitpid(-1, &status, WNOHANG|WUNTRACED)`. For each reaped child, update the corresponding job's state in `JobTable` via `job_table_update_state`.
`SIGINT`	User pressing Ctrl+C	`sigint_handler`	If there is a foreground job (`state->foreground_pgid > 0`), send `SIGINT` to its entire process group (`kill(-pgid, SIGINT)`). Otherwise, the shell itself may handle it (e.g., to cancel input).
`SIGTSTP`	User pressing Ctrl+Z	`sigtstp_handler`	If there is a foreground job, send `SIGTSTP` to its entire process group. The kernel will stop the processes and send a `SIGCHLD` with `WIFSTOPPED`, which the `SIGCHLD` handler will use to mark the job as `JOB_STOPPED`.
`SIGCONT`	Generated by `builtin_bg` or by system (`fg` after `SIGTSTP`)	Often ignored or handled to re-instate terminal control.	Shell may ignore. When a stopped job is continued by `SIGCONT`, it will later generate a `SIGCHLD` upon exit.

> **Key Insight:** The most delicate flow involves `SIGCHLD`. The handler runs **asynchronously**, interrupting the main REPL loop at unpredictable times. It must not directly call `printf` (not async-signal-safe) to report job completion. Instead, it updates the `JobTable` state, and the synchronous `job_control_check_completed_jobs` function—called from the safe context of the REPL—performs the reporting. This separates signal-catching from signal-handling logic.

Decision: Synchronous Polling vs. Asynchronous Notification for Job Status

- **Context:** The shell needs to know when background jobs terminate to report their completion and free resources. Two common approaches are: 1) Synchronously polling for child status before each prompt, or 2) Using a `SIGCHLD` signal handler for asynchronous notification.
- **Options Considered:**
  1. **Pure Polling:** In `job_control_check_completed_jobs`, call `waitpid(-1, &status, WNOHANG)` to check for any terminated children without blocking.
  2. **Pure Async Handler:** Have the `SIGCHLD` handler immediately print the job completion message and remove the job from the table.
  3. **Hybrid Approach (Chosen):** Use a `SIGCHLD` handler to **record** completion (by updating job state in a minimal, safe way), but defer **reporting and cleanup** to the next synchronous call to `job_control_check_completed_jobs` in the REPL loop.
- **Decision:** Hybrid Approach.
- **Rationale:** Pure polling can miss short-lived jobs that complete between polls, leaving zombies. Pure async handling is dangerous because `printf` and `free` are not async-signal-safe and could corrupt the shell's heap if interrupted mid-operation. The hybrid approach ensures that complex logic and output happen only in the main thread, while still providing timely reaping of child processes via the signal handler. It also centralizes job status reporting at a predictable point (before the prompt), matching the behavior of shells like `bash`.
- **Consequences:** The `JobTable` becomes a shared resource requiring careful access. The signal handler must be extremely minimal, perhaps only setting a global `volatile sig_atomic_t` flag. Alternatively, it can call `waitpid` and push the results onto a queue (using only async-signal-safe operations) for the main loop to process. This adds complexity but is robust.

Option	Pros	Cons	Chosen?
Pure Polling	Simple, no signal handling complexity. Safe from async reentrancy issues.	May leave zombies if a child dies and is reaped by `waitpid` *after* the poll but *before* the next poll. Less responsive status reporting.	No
Pure Async Handler	Immediate reaction to child termination. No zombies.	Risk of calling non-async-signal-safe functions (`printf` , `free`), leading to undefined behavior/crashes. Interleaves output unpredictably with the prompt.	No
Hybrid Approach	Timely child reaping (no zombies). Safe, complex logic runs in main thread. Centralized, predictable output.	Requires careful design of shared data (`JobTable`) between handler and main loop. Slight delay in status report (until next prompt).	**Yes**

Common Pitfalls in Component Interactions

⚠️ Pitfall: Signal Handler Corruption

- **Description:** Implementing the `SIGCHLD` handler to directly call `printf("[%d] Done %s\n", job_id, cmd)` and `job_table_remove_at` .
- **Why it's wrong:** `printf` and `free` (which may be called by remove functions) are **not** async-signal-safe. If the main loop is in the middle of a `malloc` or `printf` when the signal arrives, the handler's call could corrupt the heap or internal stdio buffers, causing crashes.
- **Fix:** The handler should only do minimal work: set a global flag (`volatile sig_atomic_t got_sigchld = 1`) or write a byte to a `self-pipe` . The main loop checks this flag or reads the pipe and then calls the non-safe functions.

⚠️ Pitfall: Race Condition in Job Table Updates

- **Description:** The main loop is iterating through the job table to display it (`builtin_jobs`) while a `SIGCHLD` handler modifies the same table (e.g., changes a state from `JOB_RUNNING` to `JOB_DONE`).
- **Why it's wrong:** This can cause the iteration to skip entries, access freed memory, or display inconsistent state (e.g., showing a job as "Running" when it just terminated).
- **Fix:** Block the relevant signals (`SIGCHLD`) during critical sections of code that traverse or modify the job table using `sigprocmask` . Alternatively, design the data structure to be lock-free for simple state updates (using atomic operations), but this is advanced in C.

⚠️ Pitfall: Orphaned Process Group on Shell Exit

- **Description:** When the shell exits (via `builtin_exit`), any background jobs still in the `JobTable` are not terminated. They become orphaned process groups and may receive `SIGHUP` from the terminal driver, causing unintended termination.
- **Why it's wrong:** It doesn't match standard shell behavior (which typically sends `SIGHUP` to all jobs). It can leave processes running that the user expected to be managed by the shell.
- **Fix:** Before exiting, iterate through the `JobTable` and send `SIGHUP` to all process groups. Then wait briefly for them to terminate. Alternatively, document that this shell does not manage job persistence on exit—but this is a non-standard behavior.

Implementation Guidance

This section provides the glue code that enables the interactions described above. Focus is on the **REPL's coordination logic**, the **signal handler setup**, and the **job status checking** that ties everything together.

A. Technology Recommendations Table

Component	Simple Option (Educational)	Advanced Option (Production-like)
Signal Handling	Standard `signal()` or `sigaction()` with safe flags. Global `volatile` flag for communication.	`sigaction` with `SA_RESTART` for certain syscalls, self-pipe trick for safe event loop integration.
Synchronization (Job Table)	Block signals (`sigprocmask`) during table updates in main loop.	Use a dedicated signal handling thread with a lock-free queue for child PIDs, or POSIX semaphores.
Input Reading	`fgets` with a large buffer, manual `&` detection.	`pselect` or `poll` to wait for input while allowing signal interruption, handling multi-byte characters.

B. Recommended File/Module Structure The interaction flow is primarily managed by the REPL (`repl.c`) and the signal handlers (likely in `signals.c` or `job_control.c`).

```
src/
  main.c                # Entry point, calls shell_state_init, run_repl_loop
  repl.c                # run_repl_loop, main coordination logic
  parser.c              # parse_command_line, tokenization, AST construction
  executor.c            # execute_ast, execute_pipeline, execute_builtin, execute_external
  job_control.c         # JobTable functions, job_control_check_completed_jobs, signal handlers
  builtins.c            # builtin_cd, builtin_exit, builtin_jobs, builtin_fg, builtin_bg
  signals.c             # sigchld_handler, sigint_handler, sigtstp_handler, signal setup helpers
  utils.c               # safe_strdup, free_command_ast, helper functions
  include/
    shell.h             # Main header with all struct/function declarations, constants
```

C. Infrastructure Starter Code: Safe Signal Handling Wrapper This code sets up a `SIGCHLD` handler that safely records child termination using a flag. The `self-pipe` trick is more robust but slightly more complex; here we show the flag approach for simplicity.

```c
/* signals.c */

#include "shell.h"

#include <signal.h>

#include <sys/wait.h>


volatile sig_atomic_t g_got_sigchld = 0;


void sigchld_handler(int sig) {

    /* Signal handler - only does minimal, safe operations. */

    int saved_errno = errno;  /* Protect errno in case waitpid uses it. */

    (void)sig;  /* Unused parameter. */


    /* Non-blocking wait to reap as many children as possible. */

    pid_t pid;

    int status;

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {

        /* We have a child state change. We need to update the job table,

           but we cannot safely call job_table_update_state here (it might use

           non-async-signal-safe functions). Instead, we set a flag and let the

           main loop handle it. For a more precise solution, we could write the

           PID and status to a lock-free ring buffer. */

        /* For now, just set the flag. The main loop will later call

           job_control_check_completed_jobs which does the actual updating. */

    }

    g_got_sigchld = 1;  /* Signal to main loop. */

    errno = saved_errno;

}


void setup_signal_handlers(void) {

    struct sigaction sa;

    sa.sa_handler = sigchld_handler;

    sigemptyset(&sa.sa_mask);

    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP; /* SA_NOCLDSTOP: don't get SIGCHLD for stopped children (we use
WUNTRACED) */

    if (sigaction(SIGCHLD, &sa, NULL) == -1) {

        perror("sigaction");
```

```
        exit(EXIT_FAILURE);
    }

    /* Ignore SIGINT and SIGTSTP in the shell itself; we will forward them to the foreground job.
       The handlers for these are set in job_control when a job is brought to foreground. */
    signal(SIGINT, SIG_IGN);
    signal(SIGTSTP, SIG_IGN);
}
```

D. Core Logic Skeleton: REPL Main Loop with Job Status Check This is the heart of the interaction flow, calling all components in the correct order.

```c
/* repl.c */

#include "shell.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>


int run_repl_loop(ShellState* state) {

    char input_line[MAX_LINE_LEN];

    int last_status = 0;


    /* Main REPL loop */

    while (1) {

        /* Step 1: Check for and report any completed background jobs.

                   This must happen BEFORE printing the prompt for standard shell behavior. */

        job_control_check_completed_jobs(state);


        /* Step 2: Display the prompt. A more advanced version might include

                   the last exit status or current directory. */

        printf("$ ");

        fflush(stdout);  /* Ensure prompt appears, especially if no newline. */


        /* Step 3: Read a line of input. Use fgets for simplicity.

                   TODO: Replace with pselect-based reading for proper signal interruption. */

        if (fgets(input_line, MAX_LINE_LEN, stdin) == NULL) {

            if (feof(stdin)) {

                /* EOF (Ctrl+D) typed. */

                printf("\n");

                break;

            } else {

                /* Read error. */

                perror("fgets");

                continue;

            }

        }
```

```
        /* Step 4: Remove trailing newline. */
        size_t len = strlen(input_line);
        if (len > 0 && input_line[len - 1] == '\n') {
            input_line[len - 1] = '\0';
        }

        /* Step 5: Check for empty line. */
        if (input_line[0] == '\0') {
            continue;
        }

        /* Step 6: Check for background operator '&' at the end.
                   This is a simplistic check; a real parser should handle it. */
        int run_in_background = 0;
        len = strlen(input_line);
        if (len > 0 && input_line[len - 1] == '&') {
            run_in_background = 1;
            input_line[len - 1] = '\0';
            /* Trim any trailing spaces before the &. */
            while (len > 1 && input_line[len - 2] == ' ') {
                input_line[len - 2] = '\0';
                len--;
            }
        }

        /* Step 7: Parse the input line into an Abstract Syntax Tree (AST). */
        ParseResult parse_result = parse_command_line(input_line, state);
        if (!parse_result.success) {
            fprintf(stderr, "Parse error: %s\n", parse_result.error_message);
            free(parse_result.error_message);
            continue;
        }

        /* Step 8: Execute the AST. The executor handles builtins, external commands,
                   pipelines, redirections, and background execution. */
        int exit_status = execute_ast(parse_result.ast, state, run_in_background);
```

```
        /* Step 9: Clean up the AST memory. */
        free_command_ast(parse_result.ast);

        /* Step 10: Store the exit status for use in $? (if implementing that).
                    For foreground commands, this is the exit status.
                    For background commands, execute_ast returns immediately (status 0). */
        if (!run_in_background) {
            state->last_exit_status = exit_status;
        }

        /* Step 11: Check if the last command was 'exit'. Builtins are handled
                    within execute_ast, so we need to check a flag or break here.
                    Alternatively, execute_builtin for exit can call exit() directly. */
        /* For simplicity, assume builtin_exit calls exit(). */
    }

    return state->last_exit_status;
}
```

E. Core Logic Skeleton: Job Status Check and Cleanup This function is called synchronously from the REPL to process the flag set by the asynchronous signal handler.

```c
/* job_control.c */

void job_control_check_completed_jobs(ShellState* state) {

    /* TODO 1: Check the global flag g_got_sigchld. If set, we know child statuses have changed. */

    if (g_got_sigchld) {

        /* TODO 2: To be safe, we can call waitpid again here in a loop (non-blocking) to ensure

                   we reap all changed children and update the job table states. This handles the case

                   where multiple children exited before we processed the signal. */

        pid_t pid;

        int status;

        while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {

            /* TODO 3: Find the job corresponding to this PID (or process group).

                       Hint: Use job_table_find_by_pgid. You may need to get the PGID from the PID. */

            /* TODO 4: Update the job's state based on WIFEXITED, WIFSIGNALED, or WIFSTOPPED.

                       Call job_table_update_state. */

        }

        /* TODO 5: Reset the global flag. */

        g_got_sigchld = 0;

    }


    /* TODO 6: Iterate through the job table (state->job_table). */

    for (int i = 0; i < state->job_table->count; i++) {

        Job* job = state->job_table->jobs[i];

        /* TODO 7: For any job in JOB_DONE state, print a notification:

                   e.g., "[%d]+ Done   %s\n", job->id, job->command_string */

        if (job->state == JOB_DONE) {

            printf("[%d]+ Done\t%s\n", job->id, job->command_string);

            /* TODO 8: Remove the job from the table and free its resources.

                       Use job_table_remove_at and free the command_string and command_list. */

            job_table_remove_at(state->job_table, i);

            i--; /* Adjust index because we removed an element. */

        }

    }

}
```

F. Language-Specific Hints (C)

- **Atomic Flag:** Use `volatile sig_atomic_t` for flags shared with signal handlers. This ensures the variable can be accessed atomically (read/written in a single instruction) even when a signal interrupts the main flow.
- **Blocking Signals:** Use `sigprocmask(SIG_BLOCK, &set, NULL)` to block signals during critical sections (like modifying the job table). Remember to save and restore the original mask.
- **Self-Pipe Trick:** For a more robust design, create a pipe in the main loop. The `SIGCHLD` handler writes a byte to the pipe. The main loop uses `select` or `poll` to monitor both the pipe and stdin. This avoids race conditions where a signal arrives just after checking the flag but before blocking in `fgets`.
- **`waitpid` Flags:** Use `WNOHANG` to avoid blocking. Use `WUNTRACED` to also receive status for stopped (suspended) children, which is essential for implementing `Ctrl+Z`.

G. Milestone Checkpoint To verify the interaction flow works for a complex command, run your shell and test:

```bash
$ ./myshell

$ ls | grep foo > out.txt &

[1] 12345  # Your shell might print the job ID and PGID.

$ # Prompt should return immediately.

$ jobs

[1]+ Running  ls | grep foo > out.txt &

$ # Wait a moment, then press Enter to get a new prompt.

$ # The shell should now print:

[1]+ Done    ls | grep foo > out.txt

$ # Verify the file out.txt was created with the correct output.
```

H. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Shell hangs after entering a pipeline command (no prompt).	Parent process is waiting (`waitpid`) for children, but a pipe end is not closed, causing a child to hang.	Add debug prints showing file descriptor numbers before and after `dup2` / `close` in each process. Use `strace -f ./myshell` to see which processes are blocking on read/write.	Ensure **all unused pipe file descriptors are closed** in all processes (parent and children). The rule: after `dup2` ing a pipe end to stdin/stdout, close both original pipe FDs in that process.
Background job completion is never reported.	`SIGCHLD` handler is not installed, or the global flag is not being checked in the main loop.	Add a print in the `SIGCHLD` handler (use `write` to STDERR_FILENO, which is async-signal-safe) to see if it fires. Check if `g_got_sigchld` is being set and cleared.	Ensure `sigaction` is called correctly. Ensure `job_control_check_completed_jobs` is called **before** printing the prompt each loop iteration.
`Ctrl+C` kills the shell instead of the foreground job.	The shell has not set up process groups correctly, or is not giving terminal control (`tcsetpgrp`) to the foreground job's process group.	Print the foreground PGID before and after launching a job. Use `ps -o pid,pgid,cmd` to see the process groups of your shell and its children.	In the child processes (and parent before waiting), call `setpgid` to put them in a new process group. For foreground jobs, call `tcsetpgrp(STDIN_FILENO, pgid)` . In the shell, ignore `SIGINT` . In a handler, forward `SIGINT` to the foreground PGID.

Error Handling and Edge Cases

> **Milestone(s):** All (1-6) – This section provides a systematic framework for managing failures and unexpected scenarios throughout the shell's lifecycle. Robust error handling is critical for a production-quality shell; this section defines the principles and specific behaviors required to achieve it.

A shell operates in a hostile environment: users provide malformed input, external commands fail, system resources are exhausted, and signals arrive asynchronously. Unlike batch programs that can abort on first error, an interactive shell must remain resilient—handling failures gracefully while preserving its internal state and remaining responsive to the user. This section defines a holistic strategy for error management and enumerates key edge cases with prescribed behaviors.

Holistic Error Strategy

> **Mental Model: The Shell as a Fault-Tolerant Service**
> Imagine the shell as a web service that must remain available 24/7. Users (clients) send requests (commands) that may be malformed, reference nonexistent resources, or trigger internal bugs. The service must: 1) validate requests, 2) isolate failures to the current request, 3) provide clear error messages, 4) release resources, and 5) return to a ready state for the next request. This mindset shifts error handling from an afterthought to a core architectural concern.

The shell's error strategy is built on four pillars: **validation before execution**, **per-command resource isolation**, **informative user feedback**, and **state consistency**. The following table summarizes the error phases and corresponding principles:

Error Phase	Detection Method	Primary Principle	Recovery Action	User Feedback
Input Parsing	Syntax analysis in `parse_command_line`	Fail-fast validation	Return `ParseResult` with `success=false` and error message	Print error to stderr, return to prompt
Built-in Execution	Return code and `errno` from system calls	Validate arguments before mutation	Roll back partial changes if possible (e.g., directory change)	Print descriptive error to stderr, set `$?` appropriately
External Command Setup	Failure of `fork()`, `pipe()`, `open()`, `dup2()`	Resource acquisition must be atomic where possible	Clean up any acquired resources (close FDs, kill child processes)	Print error to stderr, set `$?` to failure code
External Command Execution	`execvp()` failure, child termination with non-zero exit	Child failures don't crash the parent	Parent reaps child and propagates exit status	Set `$?` to child's exit status; optionally print if foreground
Signal Delivery	Asynchronous signal handlers	Signal safety: only async-signal-safe functions in handlers	Set atomic flags; defer complex handling to main loop	For SIGINT: interrupt foreground job; for SIGCHLD: report job completion
Resource Exhaustion	System calls return -1 with `errno=ENOMEM/EMFILE/ENFILE`	Graceful degradation over abrupt termination	Free available resources, terminate current command pipeline	Print "Resource temporarily unavailable" to stderr, set `$?` to failure

Validation Before Execution: Every command undergoes rigorous validation before any side effects occur. The parser rejects syntactically invalid input (unclosed quotes, misplaced operators). The executor validates semantic correctness: for built-ins, argument counts and types are checked; for external commands, the existence and executability of the target file are verified (though ultimate execution failure can still occur). This prevents the shell from entering an inconsistent state.

Per-Command Resource Isolation: Each command pipeline executes in a resource sandbox. All acquired resources—file descriptors from redirections, pipes, child processes—are scoped to that command's execution. The parent shell maintains a clean separation: it never leaks file descriptors to subsequent commands, and child processes are always reaped (via `waitpid` or SIGCHLD). This isolation is achieved through careful ownership conventions: the parent creates resources, children inherit them, and the parent cleans up any unused resources after `fork()`.

Informative User Feedback: Error messages must be actionable and follow conventions of standard shells. Use the `strerror()` function for system call failures, but augment with context: "cd: /nonexistent: No such file or directory" rather than just "No such file or directory". For parsing errors, indicate the location: "syntax error near unexpected token `|`". The exit status (`$?`) is set after every command, including built-ins, providing a machine-readable error code.

State Consistency: The shell's global state—current working directory, environment variables, job table—must remain consistent even when commands fail mid-execution. For built-ins with multiple steps (like `cd`), validate all arguments before changing state. For pipelines, if setup fails after creating some children, terminate the entire process group and clean up. The REPL loop must reset any temporary signal masks or terminal settings before prompting again.

Decision: Centralized Error Reporting vs. Distributed Handling

- **Context**: Errors can originate in the parser, executor, or job controller. We need a consistent way to report errors to the user and set the exit status.
- **Options Considered**:
  1. **Centralized**: All error reporting flows through a single `shell_error()` function that formats messages, prints to stderr, and updates `ShellState.last_exit_status`.
  2. **Distributed**: Each component handles its own error reporting, with conventions for message format and exit status setting.
- **Decision**: Use a hybrid approach: components return error codes or `ParseResult` structs, and the REPL centralizes reporting for user-facing errors. The executor sets `last_exit_status` directly.
- **Rationale**: Centralizing all reporting would create tight coupling between components and the REPL. The hybrid approach allows components to be tested independently (they return error indications) while ensuring consistent user output (the REPL formats final messages). The executor directly sets exit status because it's the authoritative source for command success/failure.
- **Consequences**: Components must document their error return conventions. The REPL must check multiple error sources (parse result, executor return value). This adds slight complexity but improves modularity.

Option	Pros	Cons	Chosen?
Centralized error reporting	Consistent message format; single place to update output behavior	Tight coupling; components harder to test in isolation	No
Distributed error handling	Loose coupling; components independent	Risk of inconsistent messages; duplicate formatting code	Partially
Hybrid (REPL formats, components indicate)	Good separation of concerns; testable components	REPL must understand all error types	**Yes**

Key Edge Cases and Behavior

Mental Model: The Shell's Boundary Cases as Stress Tests
Consider edge cases not as bugs to be fixed reactively, but as deliberate stress tests that validate the architecture. Each edge case probes a specific aspect of the design: resource management, signal safety, state isolation. By defining explicit behaviors for these cases, we ensure the shell behaves predictably under stress.

The following table enumerates critical edge cases, their expected behavior, and the implementation considerations required to achieve it. This serves as a specification for "correct" shell behavior.

Edge Case Category	Specific Scenario	Expected Behavior	Implementation Considerations
Empty/Whitespace Input	User presses Enter on empty line	Shell redisplays prompt without error; `$?` unchanged	REPL must detect empty line after trimming newline; skip parsing/execution
Parsing Oddities	Unclosed quotation marks (`echo "hello`)	Parser reports syntax error: "unexpected EOF while looking for matching `\"` "	Tokenizer must track quote state; parser validates balanced quotes before building AST
Parsing Oddities	Consecutive operators (`ls | | grep foo`)		
Parsing Oddities	Empty pipeline (`	`or` ls	
Redirection Edge Cases	Redirection to directory (`ls > /tmp`)	Child's `open()` fails with `EISDIR` ; command fails with exit status 1	Error occurs in child; parent reports child failure via exit status
Redirection Edge Cases	Multiple redirections for same FD (`ls > file1 > file2`)	Last redirection wins; first file is truncated but unused	Executor processes redirections sequentially; later `dup2()` overwrites earlier
Redirection Edge Cases	Redirection with non-existent input file (`cat < /nonexistent`)	Child's `open()` fails; command fails with exit status 1	Error occurs in child; parent reports child failure
Redirection Edge Cases	Output redirection on read-only filesystem (`echo hi > /proc/version`)	Child's `open()` or `write()` fails; command fails	Error occurs in child; parent reports child failure
Pipeline Edge Cases	Single-command pipeline (`ls	`)	Parser error (empty right-hand side)
Pipeline Edge Cases	Pipeline where first command fails immediately (`false	echo hello`)	Pipeline executes; `echo` runs; exit status is from last command (`echo` , which succeeds)
Pipeline Edge Cases	Pipeline where middle command fails (`ls	false	echo hello`)	
Pipeline Edge Cases	Very long pipeline (resource exhaustion)	`pipe()` or `fork()` fails; shell cleans up and reports error	Executor must check system call returns; clean up already-created pipes/children
Built-in Edge Cases	`cd` to non-existent directory	Error: "cd: /nonexistent: No such file or directory"; `$?` set to 1; CWD unchanged	Check `chdir()` return value; report error; do not update `PWD` environment variable
Built-in Edge Cases	`cd` with too many arguments	Error: "cd: too many arguments"; `$?` set to 1	Validate argument count before attempting `chdir()`
Built-in Edge Cases	`exit` with non-numeric argument (`exit foo`)	Error: "exit: foo: numeric argument required"; shell exits with status 2	Validate argument is numeric; use `strtol()` with error checking
Built-in Edge Cases	`export` with malformed assignment (`export =value`)	Error (optional): ignore or report; `$?` set to failure	Some shells accept; we'll reject: "export: `=value` : not a valid identifier"

Edge Case Category	Specific Scenario	Expected Behavior	Implementation Considerations
Signal Timing	Ctrl+C during parser execution (e.g., while typing long line)	Input line is cancelled; new prompt displayed	REPL must handle `EINTR` from `read()`; reset input buffer
Signal Timing	Ctrl+Z during built-in command	Built-in cannot be stopped; signal is ignored for built-ins	Only external commands have process groups that receive SIGTSTP
Signal Timing	SIGCHLD arrives while adding job to table	Race condition could cause missed job completion	Use signal blocking (`sigprocmask`) around critical sections of job table updates
Signal Timing	Signal during `fork()` or `pipe()` setup	System call returns `EINTR` ; must be restarted	Check for `EINTR` and retry system calls in loop
Background Job Edge Cases	Background job with output to terminal (no redirection)	Output interleaves with shell prompt; may cause messy display	This is expected behavior; user should redirect output
Background Job Edge Cases	`jobs` command while SIGCHLD handler is running	Job table may be in inconsistent state	Block SIGCHLD during `jobs` command execution
Background Job Edge Cases	Background job that exits before `jobs` is run	Job should appear as "Done" or be removed (depending on reporting)	SIGCHLD handler marks job as `JOB_DONE` ; `job_control_check_completed_jobs` reports and removes
Job Control Edge Cases	`fg` with non-existent job ID	Error: "fg: %1: no such job"; `$?` set to 1	Validate job exists and is not already in foreground
Job Control Edge Cases	`fg` on a completed job	Error: "fg: %1: no such job" (job removed after completion)	Job table removes completed jobs after reporting
Job Control Edge Cases	Orphaned process group (parent shell exits while background job running)	Background job receives SIGHUP and terminates (unless `nohup` used)	Shell should not explicitly kill background jobs on exit; let kernel send SIGHUP
Resource Exhaustion	`fork()` fails due to process limit	Error: "fork: Resource temporarily unavailable"; `$?` set to 1	Check `fork()` return value; clean up any pre-fork resources (pipes)
Resource Exhaustion	`pipe()` fails due to file descriptor limit	Error: "pipe: Too many open files"; fail command with cleanup	Check `pipe()` return; clean up any already-created pipes in pipeline
Resource Exhaustion	Memory allocation fails (`malloc`)	Shell should not crash; report error and fail current command	Check `malloc()` return; `parse_command_line` returns `ParseResult` with error

Empty Pipelines and Commands: The shell must reject syntactically invalid pipelines. An empty command (just whitespace) is skipped entirely. A pipeline containing only operators (`|` , `||`) is a syntax error. The parser must validate the abstract syntax tree contains at least one valid `Command` node before execution proceeds.

Redirection to Directories: When a redirection target is a directory, the `open()` system call fails with `EISDIR` . This failure occurs in the child process after `fork()` but before `exec()` . The child should print an error (if stderr is available) and exit with a non-zero status. The parent shell will see this as a normal command failure—the exit status will be non-zero, and `$?` will reflect it. The shell itself does not need special handling beyond ensuring the child's error message is visible (unless stderr was redirected).

Signals During Partial Pipeline Setup: If a signal (like SIGINT) arrives after the parent has forked some children but not all in a pipeline, the signal handler should interrupt the entire pipeline. Because all children in a pipeline share a process group, sending SIGINT to the process group will reach all children created so far. The parent must ensure the process group exists and is set before any child reaches `exec()`. The parent should then wait to reap all children. This highlights the importance of creating the process group early and adding each child to it immediately after `fork()`.

Interrupting Built-in Commands: Built-in commands run directly in the shell process; they don't have a separate process group. Therefore, SIGINT (Ctrl+C) during a built-in should interrupt the built-in and return to the prompt. However, many built-ins (like `cd`) execute quickly and are not interruptible. For built-ins that may block (like `read` in advanced shells), we must consider interruptibility. In our shell, built-ins are non-blocking and thus not interruptible—Ctrl+C will be handled by the REPL after the built-in completes. This matches simple shell behavior.

Orphaned Process Groups and Terminal Control: When a background job is suspended (via Ctrl+Z) and the shell exits, the suspended job becomes an orphaned process group. The kernel sends SIGHUP to all processes in that group, typically terminating them. Our shell should not attempt to clean up background jobs on exit (except to avoid zombies) because the kernel handles this. However, we must ensure the shell doesn't leave the terminal in an inconsistent state: when exiting, the shell should restore the terminal's foreground process group to itself (so the next shell gets control).

Memory Allocation Failures: In a low-memory situation, `malloc()` may return `NULL`. The shell should handle this gracefully. For parsing, `parse_command_line` should return a `ParseResult` with `success=false` and an appropriate error message. For execution, if memory fails during pipeline setup, the shell should clean up any allocated resources and report an error. The shell must not crash. This requires checking every `malloc()` and `realloc()` call.

Race Conditions in Job Reporting: The SIGCHLD handler runs asynchronously, potentially while the main loop is modifying the job table or printing a prompt. To prevent corruption, the handler should only set atomic flags or write to a pipe (self-pipe trick), deferring actual job table updates to the main loop. In our design, the SIGCHLD handler calls `job_table_update_state` (which must be async-signal-safe) to mark jobs as done, but the actual removal and reporting happens in `job_control_check_completed_jobs`, called from the main loop. Additionally, critical sections in the job table operations should block SIGCHLD via `sigprocmask`.

Common Pitfalls in Error Handling

⚠️ Pitfall: Ignoring System Call Return Values
Description: Forgetting to check the return value of `fork()`, `pipe()`, `dup2()`, `open()`, etc.
Why it's wrong: Failures silently propagate, causing undefined behavior. For example, if `fork()` fails and you proceed as if a child exists, you might call `waitpid()` incorrectly or leak file descriptors.
Fix: After every system call, check if return value indicates error; handle immediately with cleanup.

⚠️ Pitfall: Leaking File Descriptors on Execution Failure
Description: When `fork()` or `execvp()` fails, not closing file descriptors that were opened for redirections or pipes.
Why it's wrong: File descriptors are a finite resource; leaking them eventually causes "Too many open files" errors.
Fix: In parent process, close all pipe ends and redirection FDs after forking, regardless of success. In child, close all unnecessary FDs before `exec()`. Use a cleanup function that runs on all error paths.

⚠️ Pitfall: Incorrect Error Reporting for Failed Child
Description: Printing error messages from the parent about child failures (e.g., "command not found") before the child has attempted `execvp()`.
Why it's wrong: The child may have its own error output; duplicate messages confuse users. Also, the parent cannot know the exact reason for `execvp()` failure—only the child does.
Fix: Let the child print its own errors to stderr before exiting. The parent simply collects the exit status.

⚠️ Pitfall: Not Handling EINTR on Blocking System Calls
Description: System calls like `read()`, `waitpid()`, `pause()` can return with `errno=EINTR` when a signal arrives.
Why it's wrong: The shell may incorrectly treat an interrupted call as failure or EOF, causing unexpected termination.
Fix: Wrap such calls in a loop that retries when `errno==EINTR`. For `waitpid()` with `WNOHANG`, EINTR isn't an issue.

⚠️ Pitfall: Race Condition Between waitpid() and SIGCHLD

Description: Calling `waitpid()` in the main loop while also having a SIGCHLD handler that calls `waitpid()` can cause one to steal the exit status from the other.

Why it's wrong: A child's exit status may be lost, leaving a zombie or incorrectly marking a job as running.

Fix: Use a consistent strategy: either always wait in the handler (with proper reentrancy) or always in the main loop (using a flag set by the handler). Our design uses the handler to update job state and the main loop to reap and report.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Error Message Formatting	Direct `fprintf(stderr, ...)` with `strerror()`	Custom error catalog with localization support
Resource Cleanup	Explicit close()/free() on each error path	`goto cleanup` pattern with unified cleanup block
Signal-Safe Reporting	Write to a self-pipe, main loop reads	Use `signalfd()` (Linux-specific) for signal-as-file-descriptor

Recommended File/Module Structure:

```
shell/
  src/
    main.c                # Entry point, calls run_repl_loop
    repl.c                # REPL loop with error reporting integration
    parser.c              # parse_command_line with parse error reporting
    executor.c            # execute_ast with execution error handling
    job_control.c         # Job table with thread-safe(ish) operations
    builtins.c            # Built-in commands with validation
    error.c               # Central error formatting functions (optional)
    utils.c               # Safe wrappers for system calls (xmalloc, xfork)
  include/
    shell.h               # Main header with struct/function declarations
    error.h               # Error codes and formatting prototypes
```

Infrastructure Starter Code: The following provides safe wrappers for common system calls that handle EINTR and fatal errors appropriately. Place in `utils.c`.

```c
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include "shell.h"

/* Safe malloc: exits shell if allocation fails */
void *xmalloc(size_t size) {
    void *ptr = malloc(size);
    if (ptr == NULL && size != 0) {
        fprintf(stderr, "shell: memory allocation failed\n");
        exit(EXIT_FAILURE);  /* For shell infrastructure failure, not command */
    }
    return ptr;
}

/* Safe fork: retries on EINTR, exits on persistent failure */
pid_t xfork(void) {
    pid_t pid;
    while ((pid = fork()) == -1 && errno == EINTR) {
        /* Retry on signal interruption */
    }
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    return pid;
}

/* Safe dup2: retries on EINTR */
int xdup2(int oldfd, int newfd) {
    int ret;
    do {
        ret = dup2(oldfd, newfd);
    } while (ret == -1 && errno == EINTR);
    return ret;
```

```
}

/* Safe close: ignores EINTR (POSIX allows close to be restarted) */
int xclose(int fd) {
    int ret;
    do {
        ret = close(fd);
    } while (ret == -1 && errno == EINTR);
    return ret;
}
```

Core Logic Skeleton Code: The following shows how to integrate error handling into the main REPL loop and executor.

```c
/* In repl.c: Main loop with error integration */

int run_repl_loop(ShellState *state) {

    char line[MAX_LINE_LEN];



    while (1) {

        // Step 1: Check for completed background jobs (may print messages)

        job_control_check_completed_jobs(state);



        // Step 2: Display prompt and read input

        if (display_prompt(state) == -1) {

            // Handle EINTR from write

            continue;

        }

        if (read_line(line, sizeof(line), state) == NULL) {

            // EOF (Ctrl+D) or fatal read error

            break;

        }



        // Step 3: Parse line

        ParseResult result = parse_command_line(line, state);

        if (!result.success) {

            // Parse error: report to user

            fprintf(stderr, "shell: %s\n", result.error_message);

            state->last_exit_status = 2;  /* Syntax error exit status */

            free_parse_result(&result);

            continue;

        }



        // Step 4: Execute AST

        int background = is_background_command(result.ast);

        int exit_status = execute_ast(result.ast, state, background);



        // Step 5: Update exit status (unless background)

        if (!background) {
```

```
            state->last_exit_status = exit_status;
        }

        // Step 6: Clean up AST
        free_command_ast(result.ast);
    }

    return state->last_exit_status;
}

/* In executor.c: Executor with cleanup on failure */
int execute_pipeline(Command *pipeline, ShellState *state, int background) {
    int pipefds[2];
    int prev_read_fd = -1;
    Command *cmd = pipeline;
    pid_t last_child_pid = -1;
    int num_commands = 0;

    // TODO 1: Count commands in pipeline for process group setup

    while (cmd != NULL) {
        // TODO 2: Create pipe if not last command
        if (cmd->next != NULL) {
            if (pipe(pipefds) == -1) {
                perror("pipe");
                // TODO: Cleanup: kill any already forked children
                return 1;
            }
        }

        // TODO 3: Fork child process
        pid_t child_pid = fork();
        if (child_pid == -1) {
            perror("fork");
```

```
            // TODO: Cleanup: close pipefds, kill previous children
            return 1;
        }

        if (child_pid == 0) {
            /* Child process */
            // TODO 4: Set up process group (first child creates, others join)
            // TODO 5: Set up input redirection (prev_read_fd or stdin)
            // TODO 6: Set up output redirection (pipefds[1] or stdout)
            // TODO 7: Close all unnecessary file descriptors
            // TODO 8: Execute command (builtin or external)
            // TODO 9: If exec fails, print error and exit with appropriate code
            exit(127);  /* Command not found */
        } else {
            /* Parent process */
            // TODO 10: Track last child PID for waiting
            // TODO 11: Close pipe ends that belong to parent
            // TODO 12: Update prev_read_fd for next command
        }

        cmd = cmd->next;
        num_commands++;
    }

    // TODO 13: Parent waits for last child (if foreground)
    // TODO 14: Return exit status of last command
    return 0;
}
```

Language-Specific Hints:

- **Error Reporting:** Use `errno` and `strerror()` for system call errors. For custom errors, define an `enum` in `error.h`.
- **Resource Cleanup:** The `goto cleanup` pattern is idiomatic in C for complex cleanup:

```c
int function() {
    int *ptr = NULL;
    int fd = -1;

    ptr = malloc(100);
    if (ptr == NULL) goto error;

    fd = open("file", O_RDONLY);
    if (fd == -1) goto error;

    // ... success path ...

cleanup:
    if (fd != -1) close(fd);
    free(ptr);
    return 0;

error:
    // Handle error
    goto cleanup;
}
```

- **Signal Safety:** In signal handlers, only use async-signal-safe functions (`write` , `_exit` , `sigaction` , etc.). Never use `printf` , `malloc` , or `free` .

Milestone Checkpoint for Error Handling: After implementing error handling, test with these commands:

1. **Parse Errors:** `echo "unclosed quote` → Should print syntax error, not crash.
2. **Command Not Found:** `nonexistentcommand` → Should print "command not found" (by child) and set `$?` to 127.
3. **Permission Denied:** `./non_executable_file` → Should print "Permission denied" and set `$?` to 126.
4. **Failed Redirection:** `cat < /nonexistent` → Should print error from `cat` (No such file) and set `$?` to 1.
5. **Fork Failure:** Simulate with `ulimit -u 10` then run many processes → Shell should report "fork: Resource temporarily unavailable" but not crash.

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
Shell crashes on syntax error	Parser dereferencing NULL pointer	Add debug prints in parser to see which token causes issue	Check token array bounds and NULL checks
Background jobs become zombies	SIGCHLD handler not reaping	Use `ps aux	grep defunct`	`; check if handler calls waitpid with WNOHANG`
Ctrl+C doesn't interrupt foreground command	Wrong process group receiving signal	Print PGID of child and shell; use `ps -o pid,pgid,cmd`	Call `setpgid(0,0)` in child and `tcsetpgrp` in parent
Pipe hangs forever	Unclosed file descriptors	Use `lsof -p <pid>` to see open FDs in parent and children	Ensure parent closes both ends of pipe; children close unused ends
Error messages appear twice	Both parent and child print errors	Add PID to error prints to see origin	Let only child print exec errors; parent silent
Job table shows wrong status	Race between SIGCHLD and `jobs` command	Add prints in handler and `jobs` to see order	Block SIGCHLD during job table reads with `sigprocmask`

Testing Strategy

> **Milestone(s):** All (1-6) – This section provides a practical guide for learners to verify their implementation at each milestone, using both manual commands and automated test scripts. Thorough testing is crucial for a complex system like a shell where components interact with the operating system, manage concurrent processes, and handle unpredictable user input.

Mental Model: The Flight Simulator and Control Tower Log

Testing a shell is like operating a **flight simulator** while maintaining a **control tower log**. The simulator lets you practice specific maneuvers (individual commands) under controlled conditions, while the log records all system interactions to analyze later when something unexpected occurs. Effective testing requires both approaches: **interactive validation** (manual testing with specific commands to verify behavior matches expectations) and **automated verification** (scripted tests that compare your shell's output against a reference implementation like bash). This dual approach catches both semantic errors (wrong behavior) and implementation errors (crashes, leaks) that manual testing alone might miss.

Milestone-by-Milestone Verification

The following tables provide concrete, actionable test commands for each project milestone. For each test, run the command in **your shell** and compare the behavior to running the same command in a **reference shell** (like `bash` or `dash`). Pay attention not only to output but also to **exit status** (available via `echo $?` after the command) and **side effects** (files created, directory changes). Testing should be incremental: verify Milestone 1 thoroughly before proceeding to Milestone 2, as each layer builds upon the previous.

Milestone 1: Basic REPL and Command Execution

Test Category	Command to Run	Expected Behavior	What to Verify
REPL Basics	(start shell)	Prompt appears (e.g., `$ `) and cursor waits for input.	Prompt is displayed, shell doesn't exit immediately.
Empty Input	Press Enter on empty line	New prompt appears without error.	No segmentation fault or error message.
Simple Command	`ls` (or `echo hello`)	Lists directory contents (or prints "hello").	Output matches running same command in bash.
Command with Arguments	`ls -l /tmp`	Long listing of `/tmp` directory.	Arguments passed correctly to child process.
Command Not Found	`nonexistentcommand`	Error message like "command not found".	Exit status non-zero (typically 127). Shell continues running.
Exit Command	`exit`	Shell terminates, returns to parent shell/terminal.	Exit status is 0.
Exit with Code	`exit 42`	Shell terminates. In parent, `echo $?` prints `42`.	Custom exit code is propagated.
Path Resolution	`ls` (when `ls` is in `/bin`)	Correct listing. Works for any command in `PATH`.	Shell searches `PATH` environment variable correctly.
Tokenization with Spaces	`echo multiple spaces`	Prints "multiple spaces" (single spaces between words).	Extra whitespace is collapsed appropriately.
Quoted Strings	`echo "hello world"`	Prints "hello world" (including space).	Quoted text is treated as single token.
Mixed Quotes	`echo 'single' "double"`	Prints "single double".	Both quote types handled.
Exit Status Collection	`ls /nonexistent; echo $?`	Error message then prints `2` (or non-zero).	`waitpid` captures child's exit status correctly.

Key Validation Points: After each external command, immediately type `echo $?` to verify the exit status matches bash's behavior. Use `strace -f ./myshell` to watch `fork`, `execve`, and `waitpid` system calls, ensuring proper process creation and cleanup (no zombies).

Milestone 2: Built-in Commands

Test Category	Command to Run	Expected Behavior	What to Verify
cd Basic	`cd /tmp; pwd`	Changes to `/tmp`, prints `/tmp`.	Working directory updated (verify with `pwd`).
cd Home	`cd` (no args)	Changes to `HOME` directory.	Defaults to `$HOME` environment variable.
cd Relative	`cd ..; pwd`	Moves up one directory level.	Relative paths work.
cd Error	`cd /nonexistent`	Error message, shell stays in current directory.	Non-zero exit status, no crash.
cd with ~	`cd ~; pwd`	Changes to home directory.	Tilde expansion works.
export Basic	`export MYVAR=value; echo $MYVAR`	Prints `value`.	Variable set in environment for child processes.
export Multiple	`export A=1 B=2; env | grep -E '^A=`	^B='	Shows `A=1` and `B=2`.
export Persistence	`export TEST=foo; ./myshell -c 'echo $TEST'`	Prints `foo` (if supported).	Environment inherited by subprocesses.
echo Basic	`echo hello world`	Prints "hello world" followed by newline.	Spaces between arguments, trailing newline.
echo with Variables	`export X=test; echo $X`	Prints "test".	Variable expansion works.
echo No Newline	`echo -n hello; echo world`	Prints "helloworld" (no space between).	`-n` flag suppresses newline (if implemented).
pwd Built-in	`cd /tmp; pwd`	Prints absolute path of `/tmp`.	Matches system `pwd` command output.

Verification Technique: For `cd`, verify the shell's own working directory changed by checking with `pwd` built-in, not just child's perspective. For `export`, use `env` command in a child process (e.g., `export FOO=bar; env | grep FOO`) to see the variable in the environment.

Milestone 3: I/O Redirection

Test Category	Command to Run	Expected Behavior	What to Verify
Output Redirect	`echo hello > out.txt; cat out.txt`	Creates `out.txt` containing "hello\n".	File truncated if exists, created if not.
Append Redirect	`echo world >> out.txt; cat out.txt`	File now contains "hello\nworld\n".	Appends without truncating.
Input Redirect	`echo "input data" > in.txt; wc -l < in.txt`	Prints `1` (line count).	Command reads from file instead of stdin.
Combined Redirect	`cat < in.txt > out2.txt`	Copies `in.txt` to `out2.txt`.	Both input and output redirection work together.
Error Redirect	`ls /nonexistent 2> err.txt; cat err.txt`	Error message written to `err.txt`, not terminal.	`stderr` separated from `stdout`.
stdout+stderr	`(echo out; ls /nonexistent) > all.txt 2>&1`	Both output and error go to `all.txt`.	File descriptor duplication works.
Order Independence	`> file.txt echo hello`	Creates `file.txt` with "hello\n".	Redirections can appear anywhere in command.
Redirection with Built-ins	`echo builtin > builtin.txt`	Works (built-in commands honor redirection).	Redirection processed before built-in execution.
Permission Denied	`echo test > /root/test.txt`	Error message (Permission denied).	Shell doesn't crash, reports error appropriately.

Critical Check: After each redirection test, inspect file contents with `cat` and check file permissions (`ls -l`). Ensure no extra file descriptors are leaked (use `lsof -p <shell_pid>` to monitor open files).

Milestone 4: Pipes

Test Category	Command to Run	Expected Behavior	What to Verify
Simple Pipe	`echo hello | wc -c`	Prints `6` (bytes: "hello\n").	`stdout` of `echo` becomes `stdin` of `wc`.
Multi-stage Pipe	`seq 1 5 | grep 3 | wc -l`	Prints `1` (one line contains '3').	Data flows through all stages correctly.
Pipe with Redirection	`ls | head -5 > first5.txt`	Saves first 5 lines of `ls` to file.	Pipes and redirects combine correctly.
Exit Status Pipeline	`false | true; echo $?`	Prints `0` (exit status of last command).	Pipeline status reflects last command's exit.
Pipe with Built-in	`echo hello | cat`	Prints "hello".	Built-in commands can be in pipelines (both sides).
Large Data Pipe	`yes | head -1000 | wc -l`	Prints `1000`.	No deadlock with full pipe buffers.
Left Command Fails	`ls /nonexistent | echo hello`	Error message, then "hello".	Right command still executes.
All Commands Fail	`false | false; echo $?`	Prints `1` (non-zero).	Exit status from last failing command.

Diagnostic Method: For pipeline debugging, insert strategic `sleep` commands (`echo a \| sleep 1 \| cat`) and monitor process tree with `pstree -p` to see if all processes run concurrently. Check for hanging with `timeout`.

Milestone 5: Background Jobs

Test Category	Command to Run	Expected Behavior	What to Verify
Background Launch	`sleep 2 &`	Immediately prints job ID (e.g., `[1] 12345`) and new prompt.	Shell doesn't wait for `sleep` to finish.
jobs Listing	`sleep 5 &; jobs`	Shows `[1] Running sleep 5 &`.	Job table displays correct PID, status, command.
Multiple Background Jobs	Run several `sleep &` commands	`jobs` shows all with unique job IDs.	Job table manages multiple entries.
Job Completion Notification	`sleep 1 &` (wait 2 seconds)	Before next prompt, prints `[1] Done sleep 1`.	`SIGCHLD` handler reports finished jobs.
jobs After Completion	`sleep 1 &; wait; jobs`	`jobs` shows empty (no completed jobs).	Completed jobs removed from table after reporting.
wait for Specific Job	`sleep 2 &; wait %1`	Shell waits for that job to finish, then continues.	`wait` with job ID works.
wait for All	`sleep 1 &; sleep 2 &; wait`	Waits for all background jobs, then continues.	`wait` without arguments waits for all children.
Mixed Foreground/Background	`sleep 5 &; echo foreground`	"foreground" prints immediately, while `sleep` runs in background.	Foreground commands still block as expected.
Exit with Background Jobs	`sleep 10 &; exit`	Shell should warn "There are stopped jobs" or similar.	Shell doesn't terminate abruptly with active jobs (optional but good practice).

Verification Technique: After each background job, quickly run `ps -j` to see the process in its own process group and verify it's not a zombie. Use `jobs` output to confirm job state matches actual process state.

Milestone 6: Job Control (fg, bg, Ctrl+Z)

Test Category	Command to Run	Expected Behavior	What to Verify
Ctrl+Z Suspension	Run `sleep 100`, press Ctrl+Z	Prints `[1] Stopped sleep 100`, returns to prompt.	Process stops (visible in `ps -j` state `T`), job appears in `jobs` as `Stopped`.
fg Resume Stopped	After above, type `fg %1`	`sleep` resumes in foreground, shell waits again.	Job moves to foreground, continues execution.
bg Resume Background	After Ctrl+Z, type `bg %1`	Prints `[1] sleep 100 &`, job continues running in background.	Job state changes to `Running`, shell gets prompt back.
fg Background Job	`sleep 100 &; fg %1`	Brings background job to foreground, shell waits.	Background job transitions to foreground correctly.
Signal Forwarding Ctrl+C	`sleep 100` (foreground), press Ctrl+C	`sleep` terminates, prints `^C`, shell gets prompt.	`SIGINT` delivered to foreground process group, not shell.
Signal Forwarding Ctrl+Z	`sleep 100`, press Ctrl+Z	As above, stops the process.	`SIGTSTP` delivered to foreground process group.
fg with No Arguments	After single stopped job, type `fg`	Resumes that job (defaults to most recent).	Default job selection works.
Job Control with Pipeline	`sleep 100 | cat` then Ctrl+Z	Entire pipeline stops, one job entry for the group.	Process group receives signal together.
bg Pipeline	Stop above pipeline, then `bg`	Entire pipeline continues in background.	All processes in group receive `SIGCONT`.
Terminal Control	`sleep 100 &; fg %1`	Terminal foreground process group changes to job's PGID.	`tcsetpgrp` called appropriately.

Critical Testing: Use `ps -o pid,pgid,state,command -j` extensively to verify process group assignments and states (`S` for running, `T` for stopped). Ensure signals aren't delivered to the shell itself (which would cause it to exit on Ctrl+C).

Testing Approaches and Scripts

Beyond manual verification, systematic testing approaches help ensure robustness and catch regressions as you add features.

Mental Model: The Reference Implementation and Your Prototype

Imagine your shell as a **prototype aircraft** being tested against the **gold-standard production model** (bash). You'll run the same flight plan (test script) through both, comparing instrument readings (output, exit statuses). Discrepancies indicate where your prototype needs adjustment. Automated scripts act as flight data recorders, capturing behavior for precise comparison.

Comparison Testing Against Reference Shell

The most effective testing strategy for a shell is **comparison testing**: run the same command sequence in both your shell and a reference shell (like `bash` or `dash`), then compare outputs and exit statuses. This approach automatically validates many edge cases and complex interactions.

Procedure for Comparison Testing:

1. Create a test case file containing shell commands, one per line.
2. Run the commands through your shell and capture both `stdout` and `stderr`.
3. Run the same commands through the reference shell.
4. Compare outputs byte-for-byte, and compare exit statuses.

A simple test harness can automate this process. The harness should:

- Execute each command in a clean subprocess to avoid state contamination.
- Capture output streams separately.
- Record the exit status.
- Report mismatches with clear diffs.

Table: Test Harness Components and Responsibilities

Component	Responsibility	Implementation Note
Test Case Parser	Reads test file, splits into individual commands.	Ignore comments (lines starting with `#`).
Shell Invoker	Executes command in target shell (your shell or reference).	Uses `fork`, `exec`, pipes for capture. Must handle timeouts.
Output Capture	Captures `stdout`, `stderr` separately.	Use `pipe` and `dup2` to redirect to buffers.
Exit Status Collector	Obtains exit status via `waitpid`.	Translate signal termination to conventional exit codes.
Comparator	Compares outputs and exit statuses.	Use `diff` for outputs, exact match for status.
Result Reporter	Prints pass/fail with details for failures.	Show diff output and command that failed.

ADR: Integrated vs. External Test Harness

> **Decision: External Test Harness in Separate Language**
>
> - **Context**: We need to test the shell as a black box (like a user would), but also need precise control over execution environment and comparison logic.
> - **Options Considered**:
>   1. **Integrated C test suite**: Write tests within the shell codebase using a framework like Check, called via a special flag (e.g., `./myshell --test`).
>   2. **External script in shell**: Write test scripts in bash that invoke the shell and compare outputs using shell utilities.
>   3. **External program in higher-level language**: Write harness in Python, Ruby, or Go that spawns shells and compares outputs.
> - **Decision**: Use an external test harness written in Python.
> - **Rationale**: Python provides rich string handling, subprocess control, and diff utilities, making test code concise and maintainable. It avoids complexity of embedding tests in C and offers better error reporting than pure shell scripts. The harness can be run independently of the shell build.
> - **Consequences**: Requires Python interpreter on test system. Test execution is slightly slower than integrated C tests, but development velocity and clarity improve significantly.

Option	Pros	Cons	Chosen?
Integrated C suite	Fast execution, no external dependencies	Complex setup, harder to write tests, mixes test and production code	No
External shell script	Uses familiar shell syntax, no new languages	Limited programming constructs, error handling cumbersome	No
External Python program	Rich libraries, clean test logic, excellent string handling	Requires Python runtime	**Yes**

Holistic Testing Strategy

A comprehensive testing strategy includes multiple layers:

1. **Unit Testing for Pure Functions**: Functions like tokenizer, parser helpers, and job table operations can be unit-tested in isolation with mocked inputs.
2. **Integration Testing via Harness**: The comparison test harness validates the entire shell behavior from user input to output.
3. **Stress Testing**: Run long pipelines, many background jobs, rapid signal delivery to uncover resource leaks and race conditions.
4. **Memory Leak Detection**: Use tools like `valgrind` or AddressSanitizer during test runs to ensure no memory or file descriptor leaks.
5. **Concurrency Testing**: Use signals and background jobs simultaneously to test for race conditions in job management.

Common Pitfalls in Shell Testing

⚠️ Pitfall: Assuming Your Shell's Environment Matches Reference

- **Description**: Running tests without controlling environment variables (`PATH`, `HOME`, etc.) leads to spurious differences.
- **Why Wrong**: Different `PATH` may find different binaries; different `HOME` affects `cd ~`.
- **Fix**: In test harness, set a controlled environment (`env -i PATH=/bin:/usr/bin ...`) for both shells.

⚠️ Pitfall: Ignoring Timing and Race Conditions

- **Description**: Tests that depend on exact timing of background job reporting may pass intermittently.
- **Why Wrong**: `SIGCHLD` delivery timing is non-deterministic; background job completion message might appear before or after prompt.
- **Fix**: For reliable tests, either wait explicitly (`wait`) before checking job status, or design tests to be order-independent.

⚠️ Pitfall: Not Testing Error Paths

- **Description**: Only testing "happy path" successful commands.
- **Why Wrong**: Shells must handle errors gracefully; many bugs lurk in error recovery.
- **Fix**: Deliberately test invalid commands, permission denials, missing files, and signals.

⚠️ Pitfall: Overlooking Exit Status Details

- **Description**: Only checking output, not the exit status returned by commands.
- **Why Wrong**: Exit status is crucial for scripting; pipelines and conditionals depend on it.
- **Fix**: Always capture and compare exit status (`$?`) in your test harness.

Implementation Guidance

This subsection provides practical code and scripts to implement the testing strategy described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Test Harness Language	Python 3 (subprocess, difflib)	Go (os/exec, testing package)
Output Comparison	Byte-by-byte string compare	Normalized comparison (ignore trailing whitespace)
Exit Status Collection	`subprocess.returncode`	Distinguish exit code vs signal termination
Concurrent Test Execution	Sequential test runs	Parallel execution with isolated temp directories
Memory Leak Detection	Valgrind manual runs	Integrated ASan/LSan with test harness

B. Recommended File/Module Structure

Organize testing infrastructure separately from shell source code:

```
gauntlet_shell/
├── src/
│   ├── shell.c          # Main shell entry point
│   ├── parser.c         # Parser implementation
│   └── ...              # Other source files
├── tests/
│   ├── test_harness.py          # Main test runner
│   ├── test_cases/              # Directory for test case files
│   │   ├── milestone1.txt       # Tests for milestone 1
│   │   ├── milestone2.txt       # Tests for milestone 2
│   │   └── ...                  # Other test files
│   ├── run_tests.sh             # Convenience script to run all tests
│   └── valgrind_test.sh         # Script to run tests under valgrind
└── Makefile
```

C. Infrastructure Starter Code (Complete Test Harness)

The following Python test harness provides a complete foundation for comparison testing:

PYTHON

```python
#!/usr/bin/env python3
"""
Gauntlet Shell Test Harness
Runs commands in both the target shell and a reference shell (bash),
compares outputs and exit statuses.
"""

import subprocess
import sys
import os
import difflib
import tempfile
import shutil
from pathlib import Path

class ShellTester:
    def __init__(self, target_shell_path, reference_shell="bash"):
        """
        Initialize tester with paths to shells.
        target_shell_path: path to your shell executable
        reference_shell: name/path of reference shell (bash, dash)
        """
        self.target_shell = target_shell_path
        self.reference_shell = reference_shell
        self.env = {
            'PATH': '/bin:/usr/bin',
            'HOME': os.environ.get('HOME', '/tmp'),
            'TERM': 'xterm',
        }

    def run_command(self, shell_path, command, env=None):
        """
        Execute a single command in the given shell.
        Returns tuple (stdout, stderr, exit_code, timed_out)
        """
```

```
        if env is None:
            env = self.env

        try:
            # Use -c flag to execute command
            proc = subprocess.run(
                [shell_path, '-c', command],
                env=env,
                capture_output=True,
                text=True,
                timeout=5,  # seconds timeout
            )
            return proc.stdout, proc.stderr, proc.returncode, False
        except subprocess.TimeoutExpired:
            return "", "", 124, True  # 124 is timeout exit code convention
        except Exception as e:
            return "", str(e), 1, False


    def compare_outputs(self, test_command, target_out, target_err, target_code,
                        ref_out, ref_err, ref_code):
        """
        Compare outputs and return list of differences.
        Returns empty list if all match.
        """
        differences = []

        # Compare stdout
        if target_out != ref_out:
            diff = list(difflib.unified_diff(
                ref_out.splitlines(keepends=True),
                target_out.splitlines(keepends=True),
                fromfile='reference stdout',
                tofile='target stdout',
            ))
```

```
            if diff:
                differences.append(("stdout", ''.join(diff)))

        # Compare stderr (often less strict, but compare anyway)
        if target_err != ref_err:
            diff = list(difflib.unified_diff(
                ref_err.splitlines(keepends=True),
                target_err.splitlines(keepends=True),
                fromfile='reference stderr',
                tofile='target stderr',
            ))
            if diff:
                differences.append(("stderr", ''.join(diff)))

        # Compare exit codes
        if target_code != ref_code:
            differences.append(
                ("exit code", f"Reference: {ref_code}, Target: {target_code}")
            )

        return differences

    def run_test_file(self, test_file_path, verbose=False):
        """
        Run all commands in a test file.
        Returns list of (command, differences) for failures.
        """
        failures = []

        with open(test_file_path, 'r') as f:
            test_commands = []
            for line in f:
                line = line.strip()
                if not line or line.startswith('#'):
```

```
            continue
        test_commands.append(line)

print(f"Running {len(test_commands)} tests from {test_file_path}")

for i, cmd in enumerate(test_commands, 1):
    if verbose:
        print(f"\nTest {i}: {cmd}")

    # Run in target shell
    t_out, t_err, t_code, t_timeout = self.run_command(
        self.target_shell, cmd
    )

    # Run in reference shell
    r_out, r_err, r_code, r_timeout = self.run_command(
        self.reference_shell, cmd
    )

    # Handle timeouts
    if t_timeout:
        failures.append((cmd, [("timeout", "Target shell timed out")]))
        continue
    if r_timeout:
        failures.append((cmd, [("timeout", "Reference shell timed out")]))
        continue

    # Compare
    diffs = self.compare_outputs(cmd, t_out, t_err, t_code,
                                 r_out, r_err, r_code)
    if diffs:
        failures.append((cmd, diffs))
    elif verbose:
        print("  PASS")
```

```
        return failures


def main():
    if len(sys.argv) < 3:
        print(f"Usage: {sys.argv[0]} <target_shell> <test_file> [--verbose]")
        sys.exit(1)

    target_shell = sys.argv[1]
    test_file = sys.argv[2]
    verbose = '--verbose' in sys.argv

    if not os.path.exists(target_shell):
        print(f"Error: Target shell not found at {target_shell}")
        sys.exit(1)

    if not os.path.exists(test_file):
        print(f"Error: Test file not found at {test_file}")
        sys.exit(1)

    tester = ShellTester(target_shell)
    failures = tester.run_test_file(test_file, verbose)

    # Report results
    if failures:
        print(f"\n{'='*60}")
        print(f"FAILURES: {len(failures)}/{tester.run_test_file.count if hasattr(tester.run_test_file, 'count')
else '?'}")
        for cmd, diffs in failures:
            print(f"\nCommand: {cmd}")
            for diff_type, diff in diffs:
                print(f"  {diff_type}:")
                print(f"    {diff}")
        sys.exit(1)
```

```
    else:
        print(f"\nAll tests passed!")
        sys.exit(0)

if __name__ == '__main__':
    main()
```

D. Core Logic Skeleton Code for Test Cases

Create test case files with the following format (example for Milestone 1):

```bash
# milestone1.txt - Basic REPL and Command Execution tests
# Comments start with #
# Each non-comment line is a test command

# Empty input (just newline)
echo ""

# Simple command
ls

# Command with arguments
ls -l /

# Command not found
nonexistentcommand

# Exit with code
exit 42

# Quoted strings
echo "hello world"

# Variable PATH resolution
/bin/ls
```

For pipeline testing (Milestone 4):

```bash
# milestone4.txt - Pipe tests

# Simple pipe
echo hello | wc -c

# Multi-stage pipeline
seq 1 10 | grep 5 | wc -l

# Pipe with redirection
ls | head -5 > pipe_out.txt

# Exit status of pipeline
false | true
```

E. Language-Specific Hints (Python Test Harness)

- Use `subprocess.run()` with `capture_output=True` for Python 3.7+. For earlier versions, use `subprocess.Popen` with `communicate()`.
- Set `text=True` to get strings instead of bytes.
- The `timeout` parameter prevents hanging tests.
- Use `difflib.unified_diff` for readable output comparisons.
- For testing job control signals, you may need to use `subprocess.Popen` and send signals via `process.send_signal()`.

F. Milestone Checkpoint Verification Script

Create a convenience script to run all milestone tests sequentially:

```bash
#!/bin/bash

# run_milestone_tests.sh

TARGET_SHELL="./gauntlet_shell"

TEST_DIR="./tests/test_cases"

echo "Building shell..."

make gauntlet_shell || exit 1

for milestone in {1..6}; do

    test_file="$TEST_DIR/milestone${milestone}.txt"

    if [[ -f "$test_file" ]]; then

        echo -e "\n=== Testing Milestone $milestone ==="

        python3 tests/test_harness.py "$TARGET_SHELL" "$test_file"

        if [[ $? -ne 0 ]]; then

            echo "Milestone $milestone tests failed!"

            exit 1

        fi

    else

        echo "Test file $test_file not found, skipping milestone $milestone"

    fi

done

echo -e "\n=== All milestone tests passed! ==="
```

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Test fails on simple `ls`	PATH not set correctly in test environment	Print environment in harness: `print(env)`	Ensure `PATH` includes `/bin` and `/usr/bin`
Output matches but exit status differs	Not calling `waitpid` correctly or mishandling signals	Add debug prints to shell showing `waitpid` return values	Ensure `WIFEXITED` / `WIFSIGNALED` macros used correctly
Pipeline tests hang indefinitely	Not closing unused pipe file descriptors	Use `lsof -p <pid>` to see open file descriptors	Ensure all unused pipe ends closed in parent and children
Background job tests flaky	Race condition in `SIGCHLD` handler vs `jobs` command	Add sleep between launching job and checking status in test	Use `wait` in test to ensure job completes, or implement reliable job state tracking
Ctrl+Z tests fail in harness	Terminal signals not deliverable in non-interactive test	Test job control manually in interactive shell	For automated tests, simulate signals via `kill -TSTP <pgid>`
Memory leak reported by valgrind	Not freeing command AST or leaked file descriptors	Run `valgrind --leak-check=full ./gauntlet_shell -c 'simple command'`	Ensure `free_command_ast` called, all `open` / `pipe` fds closed
Test passes locally but fails in CI	Different environment (shell, libc, kernel version)	Compare `uname -a` and `ldd --version` between environments	Use Docker container with consistent environment for testing

Final Recommendation: Start with manual testing for each milestone using the tables provided, then implement the automated test harness to catch regressions as you add features. The combination of interactive exploration and automated verification provides the most robust testing approach for this complex system.

Debugging Guide

> **Milestone(s):** All (1-6) – This practical guide addresses the most common and frustrating issues encountered while building a shell, providing systematic diagnosis steps and concrete fixes based on the shell's architecture.

Building a Unix shell involves intricate coordination between processes, file descriptors, and signals—a combination that creates numerous subtle failure modes. This section provides a symptom-based troubleshooting guide to help identify and resolve common issues. The guide is organized by observable symptoms, their underlying causes, systematic diagnosis methods, and concrete fixes that align with the shell's architecture.

Symptom → Cause → Diagnosis → Fix Table

The following table catalogs the most common shell implementation issues, organized by observable symptom. Each entry provides a direct path from symptom to resolution.

Symptom	Likely Cause	Diagnosis Steps	Fix
Shell hangs after pipe command (e.g., `ls | grep foo` freezes)	Unclosed pipe file descriptors causing processes to wait indefinitely for EOF.	1. Run shell under `strace -f` to see which system calls block. 2. Check that all unused pipe ends are closed in parent and children. 3. Verify pipe ends are closed before `waitpid`.
Zombie processes accumulate (visible via `ps aux | grep defunct`)	Parent not reaping child exit status via `waitpid`.	1. Check if `SIGCHLD` handler is installed and properly reaps all children. 2. Verify `waitpid` uses `WNOHANG` in loop to reap multiple children. 3. Check that background job completion is being handled.	Install `SIGCHLD` handler that calls `waitpid(-1, &status, WNOHANG)` in a loop. In main REPL, call `job_control_check_completed_jobs` to clean up finished jobs from table.
Background jobs not reported when completed (no message at prompt)	Shell not checking/completing background jobs at prompt display.	1. Verify `job_control_check_completed_jobs` is called before printing prompt. 2. Check that `SIGCHLD` handler updates job state to `JOB_DONE`. 3. Confirm job table removal occurs after reporting.	Call `job_control_check_completed_jobs(state)` at start of each REPL iteration. Ensure `job_table_update_state` is called from `SIGCHLD` handler.
Ctrl+C doesn't interrupt foreground command (signal goes to shell instead)	Incorrect terminal foreground process group assignment.	1. Check that `tcsetpgrp` is called before exec in foreground. 2. Verify child process sets its own process group with `setpgid(0, 0)`. 3. Check that shell restores its own process group with `tcsetpgrp` after child exits.	In foreground execution: parent calls `setpgid` for child to new PGID, then `tcsetpgrp` to give child terminal control. Parent waits, then restores with `tcsetpgrp` to shell's PGID.
Ctrl+Z doesn't suspend foreground command	SIGTSTP not forwarded to foreground process group.	1. Verify shell's SIGTSTP handler is installed and calls `tcsetpgrp` to restore shell. 2. Check that foreground job is placed in its own process group. 3. Confirm shell doesn't ignore SIGTSTP for itself.	In SIGTSTP handler: call `tcsetpgrp` to give terminal back to shell, send SIGTSTP to foreground process group, update job state to `JOB_STOPPED`.
fg/bg commands report "No such job"	Job table lookup failure or incorrect job ID tracking.	1. Check that `job_table_find_by_id` correctly searches by sequential job ID. 2. Verify job IDs are assigned incrementally and not reused prematurely. 3. Confirm job removal only happens after explicit user acknowledgment.	Ensure `next_job_id` increments on each `job_table_add`. Remove jobs only when user is notified of completion or via explicit action.
Input/output redirection fails silently (command runs but no file created)	Incorrect `dup2` order or missing `close`.	1. Check `open` flags (O_CREAT|O_TRUNC for `>`, O_APPEND for `>>`). 2. Verify `dup2` target fd (1 for stdout, 2 for stderr, 0 for stdin). 3. Confirm original fd is closed after `dup2`.	In `setup_redirections`: open file with correct flags, call `dup2(fd, target_fd)`, then `close(fd)`. Perform before `execvp`.
Pipeline only executes first command	Incorrect pipeline execution logic—	1. Check `execute_pipeline` loops through all commands in linked list. 2. Verify each command (except last) has its	In `execute_pipeline`: for each command, create pipes as needed, fork child, setup redirections, exec. Parent closes all pipe fds and waits for all children.

Symptom	Likely Cause	Diagnosis Steps	Fix
(e.g., `ls | grep foo` only runs `ls`)	parent doesn't fork all commands.	stdout redirected to pipe write end. 3. Confirm parent waits for all children, not just first.	
Environment variables not passed to child processes	Incorrect handling of `environ` or missing `export` implementation.	1. Check that `export` modifies `environ` directly or maintains separate `env` array. 2. Verify `execvp` uses current environment (it does by default). 3. Confirm variable is added to environment before fork/exec.	Maintain global `char **environ` or equivalent. In `builtin_export`, use `setenv` or manually add to environment array.
cd command doesn't change directory	Changing directory in child process instead of shell process.	1. Verify `builtin_cd` calls `chdir` directly (not in forked child). 2. Check that `PWD` environment variable is updated after successful `chdir`. 3. Confirm error handling for non-existent directories.	Implement `builtin_cd` as built-in (not external). Call `chdir`, then update `PWD` environment variable with `getcwd`.
Shell crashes on empty input line	Null pointer dereference in parser or executor.	1. Check that `parse_command_line` handles empty string and returns `ParseResult` with `success=false`. 2. Verify REPL skips execution when parse fails. 3. Confirm tokenizer handles EOF/newline only input.	In REPL: after reading line, if `strlen(line) == 0` or only whitespace, continue to next iteration. Ensure parser returns gracefully.
Commands with quotes fail (e.g., `echo "hello world"` prints as two arguments)	Tokenizer not handling quoted strings as single tokens.	1. Check tokenizer state machine for quote handling. 2. Verify that quotes are stripped from final token. 3. Confirm escape character () handling.	Implement tokenizer with quote-aware finite state machine. Track opening quote, collect until matching quote, strip quotes from stored token.
Multiple redirections fail (only first works)	`Redirection` linked list not fully processed.	1. Check that `Command->input_redir` and `Command->output_redir` are linked lists. 2. Verify `setup_redirections` iterates through all redirections. 3. Confirm multiple redirections of same type are handled (last wins).	In `setup_redirections`: iterate through `input_redir` list, apply each (last will override). Same for `output_redir`. Close files after dup2.
Background job continues to receive terminal input	Process group not detached from terminal properly.	1. Check that background job is not given terminal control via `tcsetpgrp`. 2. Verify background job's process group differs from shell's foreground group. 3. Confirm background job's stdin is redirected from `/dev/null`.	For background jobs: after fork, redirect stdin from `/dev/null` if not otherwise redirected. Do not call `tcsetpgrp` for background jobs.
Shell becomes unresponsive	Memory leak in command AST or job table.	1. Check that `free_command_ast` is called after each command execution. 2. Verify `job_table_remove_at` frees	Implement and consistently call cleanup functions: `free_command_ast` after execution, `free` for all dynamically allocated strings and arrays.

Symptom	Likely Cause	Diagnosis Steps	Fix
after many commands		`Job` and `command_string`. 3. Use `valgrind` to track allocations.	

Debugging Tools and Techniques

Effective shell debugging requires a combination of systematic observation, strategic instrumentation, and Unix debugging tools. This section describes practical approaches for diagnosing issues in a shell implementation.

Mental Model: The Shell as a Theater Production

> Think of debugging the shell as investigating a theater production where something went wrong. The **actors** are processes, their **scripts** are the programs they execute, the **stage directions** are file descriptor manipulations, and the **stage manager** is the shell coordinating everything. When the play halts (shell hangs), you need to check: Are actors waiting for cues that never come? Are stage directions misunderstood? Is the stage manager distracted by signals from the audience?

Strategic Print Statement Instrumentation

The most direct debugging technique is adding strategic print statements to trace execution flow and inspect critical values. Always print to stderr (file descriptor 2) to avoid interfering with the shell's own redirection behavior.

What to Print	Where to Print	Information Gained
PID and PPID	In parent before fork, in child after fork.	Verify parent-child relationships and identify which code path executes.
File descriptor numbers	After each `open`, `pipe`, `dup2`, `close`.	Track fd leaks and verify correct redirection setup.
Command arguments	Before `execvp`.	Confirm tokenization and parsing produced correct argv array.
Signal handler calls	Inside signal handlers (use `write` not `printf`).	Verify signal delivery and handler execution order.
Job table state	After each job table modification.	Track job lifecycle and state transitions.

Example approach: Create a debug macro that conditionally compiles print statements:

```c
#ifdef DEBUG
#define DEBUG_PRINT(...) fprintf(stderr, __VA_ARGS__)
#else
#define DEBUG_PRINT(...)
#endif
```

System Call Tracing with `strace`

The `strace` tool intercepts and records system calls made by a process and its children, providing unparalleled visibility into the shell's interaction with the kernel.

Basic usage for shell debugging:

- `strace -f ./myshell`: Trace all system calls from shell and its children (`-f` follows forks)
- `strace -f -e trace=process,open,close,dup2,pipe,waitpid ./myshell`: Filter to specific relevant calls
- `strace -f -o strace.log ./myshell`: Output to file for analysis

Key patterns to identify:

- **Hanging processes**: Look for blocking calls like `read`, `waitpid` without `WNOHANG`
- **File descriptor leaks**: Count `open` / `pipe` calls vs `close` calls
- **Signal issues**: Check `rt_sigaction` calls for handler setup and `kill` for signal sending
- **Process group problems**: Look for `setpgid` and `tcsetpgrp` calls

Interpretation guide:

```
pipe([3, 4]) = 0                    # Created pipe with read fd=3, write fd=4
fork() = 1234                       # Child PID 1234 created
[pid 1234] close(3)                 # Child closed read end (good)
[pid 1234] dup2(4, 1)               # Child redirected stdout to pipe write end
[pid 1234] close(4)                 # Child closed original write end (good)
[pid 1233] close(4)                 # Parent closed write end (good)
[pid 1233] waitpid(-1,              # Parent waits for any child
```

Process Tree Visualization with `pstree`

Understanding process relationships is critical for debugging job control and pipeline issues. The `pstree` command displays processes in a tree format showing parent-child relationships.

Usage:

- In one terminal: `./myshell`
- In another terminal: `pstree -p -a -s <shell_pid>`
- Or from within shell: add `system("pstree -p $$ > /tmp/tree")` to debug code

What to look for:

- **Process groups**: Check if processes share the same PGID (shown in parentheses)
- **Zombie processes**: Marked with `<defunct>`
- **Orphaned processes**: Processes whose parent has exited
- **Pipeline structure**: Verify all commands in pipeline are children of shell

File Descriptor Inspection via `/proc`

The Linux `/proc` filesystem exposes detailed process information, including open file descriptors.

To inspect a running shell or its children:

1. Find the PID: `ps aux | grep myshell`
2. List open file descriptors: `ls -la /proc/<pid>/fd/`
3. Examine descriptor details: `readlink /proc/<pid>/fd/3`

Common diagnostics:

- **Leaked descriptors**: Unexpected numbers (high values) or unexpected types
- **Pipe connections**: Verify both ends are open somewhere
- **Redirection targets**: Confirm files are opened with correct paths

Signal Diagnostics

Signal-related bugs are particularly subtle because they involve asynchronous events. Use these techniques:

1. **Signal delivery tracing**: `strace -f -e signal=all ./myshell`
2. **Handler debugging**: Add `write(2, "SIGCHLD received\n", 18)` in handlers (use `write` as it's async-safe)
3. **Signal mask inspection**: Use `sigprocmask` to check blocked signals at critical sections
4. **Process group verification**: Check `/proc/<pid>/stat` for process group ID

Memory Debugging with `valgrind`

Memory leaks and corruption can cause gradual degradation or mysterious crashes.

Usage:

- `valgrind --leak-check=full --track-origins=yes ./myshell`
- For child processes: `valgrind --trace-children=yes ./myshell`

Focus areas:

- **Command parsing**: Ensure `free_command_ast` frees all allocated memory
- **Job table**: Verify `job_table_remove_at` frees strings and structures
- **Tokenization**: Check for unfreed tokens in parse error paths

Comparison Testing Against Reference Shell

When behavior differs from expectations, compare against a reference shell (bash, dash) using the same commands.

Technique:

1. Create test script: `echo -e "ls | grep foo\nexit" > test_input`
2. Run reference: `bash < test_input 2>&1 > ref_output`
3. Run your shell: `./myshell < test_input 2>&1 > my_output`
4. Compare: `diff -u ref_output my_output`

Common comparison points:

- Exit status: `echo $?` after command
- Environment variable propagation
- Signal behavior (Ctrl+C, Ctrl+Z)
- Error messages for invalid commands

Interactive Debugging with GDB

For deep investigation of crashes or complex state issues, use GDB with fork awareness.

Setup:

```bash
gdb ./myshell

(gdb) set follow-fork-mode child  # or parent

(gdb) break execute_external

(gdb) run
```

Useful GDB commands for shell debugging:

- `info registers`: Check process state
- `backtrace`: Stack trace after crash
- `p *command`: Inspect parsed command structure
- `call write(2, "debug\n", 6)`: Safe output from within debugger

The "Self-Pipe Trick" for Signal Race Condition Diagnosis

If experiencing signal race conditions (particularly with SIGCHLD), implement the self-pipe trick to convert signals to I/O events, making them synchronous and easier to debug.

Diagnostic implementation:

1. Create a pipe in shell initialization

2. In signal handler, write a byte to the pipe write end
3. In main loop, use `select` / `pselect` to monitor pipe read end
4. When bytes readable, process pending signals synchronously

This technique eliminates timing-dependent bugs and makes signal handling deterministic for debugging.

Implementation Guidance

> **Technology Note**: The debugging techniques described work with any Unix-like system, but some tools (`strace` , `valgrind` , `/proc` inspection) are most fully featured on Linux. On macOS, consider `dtruss` or `lldb` as alternatives.

Debugging Infrastructure Starter Code

The following complete, ready-to-use code provides debugging utilities that can be integrated into the shell implementation.

File: `debug_utils.c`

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <time.h>

/* Global debug flag - set via environment variable or compile flag */
int DEBUG_ENABLED = 0;

/* Initialize debug mode from environment */
void debug_init(void) {
    if (getenv("SHELL_DEBUG") != NULL) {
        DEBUG_ENABLED = 1;
        debug_printf("Debug mode enabled\n");
    }
}

/* Thread-safe debug output to stderr */
void debug_printf(const char *format, ...) {
    if (!DEBUG_ENABLED) return;

    va_list args;
    va_start(args, format);

    /* Add timestamp and PID prefix */
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    fprintf(stderr, "[%ld.%06ld pid=%d] ",
            ts.tv_sec, ts.tv_nsec / 1000, getpid());

    vfprintf(stderr, format, args);
    fflush(stderr);
    va_end(args);
}
```

```c
/* Print file descriptor table for current process */
void debug_print_fds(void) {
    if (!DEBUG_ENABLED) return;

    char buf[256];
    snprintf(buf, sizeof(buf), "/proc/%d/fd", getpid());

    debug_printf("File descriptors for pid=%d:\n", getpid());
    /* Note: /proc inspection is Linux-specific */
#ifdef __linux__
    FILE *fp = popen("ls -la /proc/self/fd 2>/dev/null", "r");
    if (fp) {
        char line[256];
        while (fgets(line, sizeof(line), fp)) {
            debug_printf("  %s", line);
        }
        pclose(fp);
    }
#endif
}

/* Safe signal-safe debug output (use in signal handlers) */
void debug_signal_safe(const char *msg) {
    write(STDERR_FILENO, "[SIGNAL] ", 9);
    write(STDERR_FILENO, msg, strlen(msg));
    write(STDERR_FILENO, "\n", 1);
}

/* Dump command structure for debugging */
void debug_dump_command(const struct Command *cmd, int depth) {
    if (!DEBUG_ENABLED) return;

    char indent[32];
    memset(indent, ' ', depth * 2);
    indent[depth * 2] = '\0';
```

```
    debug_printf("%sCommand at %p:\n", indent, (void*)cmd);
    debug_printf("%s  argc: %d\n", indent, cmd->argc);

    for (int i = 0; i < cmd->argc; i++) {
        debug_printf("%s  argv[%d]: '%s'\n", indent, i, cmd->argv[i]);
    }

    struct Redirection *redir = cmd->input_redir;
    while (redir) {
        debug_printf("%s  input_redir: type=%d, fd=%d, filename='%s'\n",
                     indent, redir->type, redir->fd, redir->filename);
        redir = redir->next;
    }

    redir = cmd->output_redir;
    while (redir) {
        debug_printf("%s  output_redir: type=%d, fd=%d, filename='%s'\n",
                     indent, redir->type, redir->fd, redir->filename);
        redir = redir->next;
    }

    if (cmd->next) {
        debug_dump_command(cmd->next, depth + 1);
    }
}

/* Dump job table state */
void debug_dump_job_table(const struct JobTable *table) {
    if (!DEBUG_ENABLED) return;

    debug_printf("Job Table (count=%d, capacity=%d):\n",
                 table->count, table->capacity);
```

```
    for (int i = 0; i < table->count; i++) {
        struct Job *job = table->jobs[i];
        debug_printf("  Job %d: id=%d, pgid=%d, state=%d, cmd='%s'\n",
                     i, job->id, job->pgid, job->state,
                     job->command_string ? job->command_string : "(null)");
    }
}
```

File: `debug_utils.h`

```c
#ifndef DEBUG_UTILS_H
#define DEBUG_UTILS_H

#include "data_model.h"  /* For Command, JobTable structures */

extern int DEBUG_ENABLED;

void debug_init(void);
void debug_printf(const char *format, ...);
void debug_print_fds(void);
void debug_signal_safe(const char *msg);
void debug_dump_command(const struct Command *cmd, int depth);
void debug_dump_job_table(const struct JobTable *table);

/* Macro for conditional debug prints */
#define DEBUG(...) if (DEBUG_ENABLED) debug_printf(__VA_ARGS__)

#endif /* DEBUG_UTILS_H */
```

Integration into Shell Components

In `main.c` or shell entry point:

```c
#include "debug_utils.h"

int main(int argc, char **argv) {
    debug_init();
    DEBUG("Shell starting\n");

    ShellState state;
    shell_state_init(&state);

    /* ... rest of initialization ... */

    return run_repl_loop(&state);
}
```

In `executor.c` at strategic points:

```c
pid_t execute_external(Command *cmd, ShellState *state,
                       int stdin_fd, int stdout_fd,
                       int stderr_fd, int is_background,
                       pid_t pgid) {
    DEBUG("execute_external: cmd=%s, background=%d, pgid=%d\n",
          cmd->argv[0], is_background, pgid);
    DEBUG("  stdin_fd=%d, stdout_fd=%d, stderr_fd=%d\n",
          stdin_fd, stdout_fd, stderr_fd);

    pid_t pid = fork();
    if (pid == 0) {
        /* Child process */
        DEBUG("  Child pid=%d starting\n", getpid());

        /* ... redirection and exec setup ... */

        DEBUG("  Child about to execvp: %s\n", cmd->argv[0]);
        execvp(cmd->argv[0], cmd->argv);

        /* If execvp returns, it failed */
        DEBUG("  execvp failed: %s\n", strerror(errno));
        _exit(EXIT_FAILURE);
    } else if (pid > 0) {
        /* Parent process */
        DEBUG("  Parent forked child pid=%d\n", pid);
        return pid;
    } else {
        DEBUG("  fork failed: %s\n", strerror(errno));
        return -1;
    }
}
```

In signal handlers (safe version):

```c
void sigchld_handler(int sig) {
    /* Signal-safe debug output */
    debug_signal_safe("SIGCHLD received");

    pid_t pid;
    int status;

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        char buf[64];
        snprintf(buf, sizeof(buf), "Child %d changed status", pid);
        debug_signal_safe(buf);

        /* ... rest of handler ... */
    }
}
```

Test Harness for Systematic Debugging

Create a simple test harness to automate bug reproduction:

File: `test_harness.sh`

```bash
#!/bin/bash

set -e


SHELL="./myshell"


# Enable debug output

export SHELL_DEBUG=1


# Test cases with expected behavior

test_cases=(

    "echo hello"                    # Basic command

    "ls | grep .c"                  # Simple pipe

    "cat < /etc/passwd | head -5"   # Pipe with input redirect

    "sleep 5 &"                     # Background job

    "cd /tmp && pwd"                # Built-in commands

)


for cmd in "${test_cases[@]}"; do

    echo "=== Testing: $cmd ==="

    echo "$cmd" | $SHELL 2>&1 | head -20

    echo "Exit status: $?"

    echo

done


# Stress test: multiple pipes

echo "=== Stress test: complex pipeline ==="

echo "ls -la | grep '^d' | wc -l | cat" | $SHELL 2>&1


# Signal test

echo "=== Testing Ctrl+C handling ==="

timeout --signal=SIGINT 2 $SHELL <<< 'sleep 10' 2>&1 || true
```

Debugging Checklist for Common Issues

When encountering an issue, follow this systematic checklist:

1. **Enable debug output**: `export SHELL_DEBUG=1`
2. **Run with strace**: `strace -f -o trace.log ./myshell`
3. **Reproduce minimally**: Create smallest command that exhibits the issue
4. **Check file descriptors**: Call `debug_print_fds()` at point of failure

5. **Verify process relationships**: Use `pstree` to visualize process tree
6. **Compare with reference**: Run same command in bash/dash
7. **Check error codes**: After each system call, verify return value and `errno`
8. **Test edge cases**: Empty input, very long commands, special characters

Language-Specific Debugging Hints for C

- **Async-signal safety**: In signal handlers, only use functions listed in `signal-safety(7)`. Prefer `write()` over `printf()`.
- **Memory debugging**: Use `valgrind` regularly. Initialize pointers to `NULL` and check before `free()`.
- **File descriptor management**: Always check return values of `open`, `pipe`, `dup2`. Close descriptors in reverse order of creation.
- **Process state**: Use `waitpid` with appropriate flags (`WNOHANG` for non-blocking, `WUNTRACED` to detect stopped processes).
- **Atomic operations**: For variables shared with signal handlers, use `volatile sig_atomic_t` type.

Milestone-Specific Debugging Focus

- **Milestone 1-2**: Focus on command parsing and basic execution. Use `strace` to verify `fork`/`exec` sequence.
- **Milestone 3-4**: Focus on file descriptors. Use `debug_print_fds()` after each redirection and pipe operation.
- **Milestone 5-6**: Focus on signals and process groups. Use `strace -e signal` and check `/proc/<pid>/stat` for PGIDs.

By combining these debugging techniques with the systematic symptom diagnosis table, you can effectively identify and resolve the most common shell implementation issues.

Future Extensions

> **Milestone(s):** All (1-6) – This section explores potential enhancements beyond the core project scope, demonstrating how the foundational architecture can evolve. It serves as inspiration for learners to continue their systems programming journey, highlighting both incremental improvements and major feature expansions.

The core shell architecture—built around the REPL, Parser, Executor, and Job Control Manager components, and the data model of `Command` ASTs and `Job` tracking—provides a robust foundation. This design intentionally isolates concerns, creating clear extension points for new functionality. While the project's primary goals focus on essential Unix shell operations, the system's modularity invites exploration of additional features that make a shell more practical and powerful. This section catalogs potential extensions, categorizing them by implementation complexity and analyzing how they integrate with—or require modification to—the existing design.

Easy Additions

These features require relatively modest changes, often confined to a single component. They leverage existing extension points like the built-in command dispatcher, the tokenizer, or the prompt display logic. Implementing them reinforces understanding of the shell's internal data flow without major architectural surgery.

Feature	Primary Component(s)	Key Changes Required	User Benefit
Command History	REPL	Add history storage/recall, keybinding handling	Faster command repetition and editing
Aliases	Parser, Built-ins	Add alias substitution phase, `alias` / `unalias` built-ins	Custom command shortcuts and macros
Prompt Customization	REPL	Expand prompt string with variable substitution (`\u` , `\w` , `\$`)	More informative shell context
Directory Stack (`pushd` , `popd` , `dirs`)	Built-ins	Add stack data structure, new built-in commands	Efficient directory navigation
Globbing (Wildcard Expansion)	Parser (post-tokenization)	Add `glob` function to expand `*` , `?` , `[]` in arguments	Convenient file selection
`$?` Variable Expansion	Parser/Executor	Expand `$?` in arguments to previous exit status	Scripting and conditionals

Mental Model: The Shell as a Customizable Workshop Toolbox

Think of the base shell as a well-organized toolbox containing essential tools (fork, exec, pipes). Easy additions are like adding custom grips, laser guides, or specialized bits to these tools—they don't change the fundamental purpose of the hammer (the executor) or the measuring tape (the parser), but they make them more ergonomic and suited to specific workflows. Each addition is a modular accessory that snaps onto an existing tool.

Command History

Concept: Persistent storage and recall of previously entered command lines, typically accessible via Up/Down arrow keys or history built-in commands (`history` , `!n`).

Integration Analysis: The REPL component already reads input lines; history requires storing these lines in a data structure (circular buffer or list) and adding a layer of input editing. The `readline` library provides this functionality comprehensively, but implementing a basic version illuminates terminal I/O.

> **ADR: History Implementation Strategy**
>
> - **Context:** The shell needs to provide command recall for user productivity. The REPL's linear input reading must be enhanced.
> - **Options Considered:**
>   1. **Use GNU Readline/Linenoise:** Link against a mature library for full features (history, editing, completion).
>   2. **Implement Minimal In-Memory History:** Store last N commands in an array, add Up/Down key handling via terminal raw mode.
>   3. **File-Backed History:** Persist history across sessions to a file (e.g., `~/.gauntlet_history`).
> - **Decision:** Start with option 2 (minimal in-memory) for educational value, then optionally add file persistence.
> - **Rationale:** Implementing basic history from scratch teaches terminal raw mode and keycode handling, which is valuable for understanding how interactive CLI tools work. It avoids external dependencies for core learning.
> - **Consequences:** Learners gain deeper insight into terminal I/O but must handle edge cases (multibyte characters, terminal resizing). The design can later be refactored to use Readline if desired.

Option	Pros	Cons	Recommended for Learning?
GNU Readline	Feature-complete, robust, handles all edge cases	External dependency, opaque implementation, less educational	No (for initial implementation)
Minimal In-Memory History	Teaches terminal raw mode, controllable complexity	Limited features, must handle keycodes manually	Yes
File-Backed History	Persistence across sessions, more realistic	Adds file I/O complexity, synchronization issues	As secondary enhancement

Required Changes:

1. **REPL State Enhancement:** Add `char** history` array and `int history_count`, `history_index` to `ShellState`.
2. **Input Loop Modification:** Before calling `fgets()` / `read()`, put terminal in non-canonical mode to read keys without waiting for Enter. Detect Up/Down arrows (escape sequences `\x1b[A`, `\x1b[B`), replace current input line with history item.
3. **History Built-in:** Add `builtin_history()` to print numbered history list.
4. **Persistence (Optional):** On shell start, read `~/.gauntlet_history`; on each command, append to in-memory list and periodically flush to file.

Aliases

Concept: User-defined string substitutions for commands, e.g., `alias ll='ls -l'` causes input `ll` to be replaced with `ls -l` before parsing.

Integration Analysis: Aliases are a pure text transformation that occurs after reading the input line but before the parser's tokenization. The `alias` and `unalias` built-ins manage a dictionary of alias mappings stored in `ShellState`.

Required Changes:

1. **Data Structure:** Add `AliasTable` to `ShellState` (e.g., hash table or linked list of `struct { char* name; char* value; }`).
2. **Preprocessing Step:** In `run_repl_loop()`, after reading a line and before calling `parse_command_line()`, perform alias substitution. This must handle recursive aliases (e.g., `alias a='b'`, `alias b='a'`) by imposing a substitution depth limit.
3. **Built-in Commands:** Implement `builtin_alias()` and `builtin_unalias()` to manipulate the table.
4. **Parser Consideration:** Since substitution happens before tokenization, aliases can include redirections and pipes (e.g., `alias lpipe='ls -l | less'`). The expanded text is re-tokenized normally.

Prompt Customization

Concept: Allow the prompt string (`PS1` environment variable) to contain escape sequences that expand to dynamic information: `\u` (username), `\w` (working directory), `\$` (`#` for root, `$` otherwise), `\h` (hostname), etc.

Integration Analysis: The REPL currently displays a static prompt. This feature moves prompt generation to a helper function that reads `PS1` (or uses a default) and performs substitution before each prompt display.

Required Changes:

1. **Prompt Expansion Function:** Create `char* expand_prompt(const char* fmt, ShellState* state)` that scans the format string, replacing `\u` with `getenv("USER")`, `\w` with `getcwd()` (possibly truncated), etc.
2. **REPL Integration:** In `run_repl_loop()`, call `expand_prompt()` before `printf()`.
3. **Environment Variable:** Respect `PS1`; update it via `export` built-in. The `ShellState` might cache the expanded prompt for efficiency.

Directory Stack (`pushd`, `popd`, `dirs`)

Concept: Maintain a stack of visited directories, allowing quick navigation between them via `pushd <dir>` (go to dir, push current), `popd` (go to top, pop), `dirs` (list stack).

Integration Analysis: This is a classic built-in command set that requires adding a stack data structure (array or linked list) to `ShellState` and implementing three new built-ins. It builds upon the `cd` command's directory-changing logic.

Required Changes:

1. **Data Structure:** Add `char** dir_stack` and `int dir_stack_size` to `ShellState`.
2. **Built-ins:** Implement `builtin_pushd()`, `builtin_popd()`, `builtin_dirs()`. `pushd` must call `chdir()` and update `PWD` environment variable (reusing `cd`'s logic).
3. **Error Handling:** Handle empty stack on `popd`, limit stack size.

Globbing (Wildcard Expansion)

Concept: Expand filename wildcards (`*`, `?`, `[abc]`) in command arguments to matching file list, e.g., `echo *.c` expands to `echo a.c b.c c.c`.

Integration Analysis: Globbing occurs after tokenization but before executing a command. The `glob()` POSIX function does this, but implementing a simple version illuminates filesystem interaction. Expansion must happen for each argument token that contains wildcard characters (unless quoted).

Required Changes:

1. **Expansion Point:** In the parser or executor, after building the `argv` array for a `Command`, scan each argument for unquoted wildcard characters. If found, replace that argument with a list of matching filenames.
2. **`glob()` Integration:** Use `glob(3)` for robustness, handling `GLOB_NOSORT`, `GLOB_NOCHECK`. Ensure memory management: `glob()` allocates an array that must be freed.
3. **Executor Adjustment:** The `argv` array becomes variable-length; need to reallocate to accommodate expanded list. This may affect the `Command` struct's `argv`/`argc` fields.

`$?` Variable Expansion

Concept: Expand the special variable `$?` to the exit status of the last executed foreground command.

Integration Analysis: The `ShellState` already tracks `last_exit_status`. Expansion is a text substitution similar to alias expansion but occurs for all variables (if extended to `$VAR`). Should happen after tokenization to distinguish `$?` as a single token.

Required Changes:

1. **Tokenizer Enhancement:** Recognize `$?` as a special token (or part of a broader variable tokenization scheme).
2. **Parser/Executor Integration:** After parsing, walk the `Command`'s `argv` and replace any token equal to `"$?"` with string representation of `state->last_exit_status`. Alternatively, expand in the executor just before `execvp()`.
3. **Edge Cases:** `$?` inside quoted strings: typically expanded in double quotes but not single quotes.

> **Design Insight:** Many easy additions are **pre-processors** or **post-processors** on the main data flow. History and aliases transform the input string before parsing. Globbing and variable expansion transform the token list after parsing but before execution. This pattern suggests a pipeline of transformations, each stage potentially optional. The clean separation between the Parser (syntax) and Executor (semantics) makes inserting these stages straightforward.

Advanced Extensions

These features require more significant architectural changes, often touching multiple components or introducing new complex subsystems. They represent the evolution of a simple shell into a more complete programming environment or system tool.

Feature	Primary Component(s)	Key Changes Required	User Benefit
Scripting (Conditionals, Loops)	Parser, Executor, New Control Flow Engine	New AST nodes, control flow evaluation, variable scoping	Shell scripting capability
Tab Completion	REPL, Parser, External Integrator	Real-time path/command suggestion, asynchronous I/O	Faster and more accurate typing
Remote Shell (ssh-like server)	New Network Component, REPL adaptation	Socket I/O, authentication, process spawning over network	Remote administration
Job Control Enhancements (notify, disown)	Job Control Manager	New job states, signal handling modifications	Better background job management
Coprocesses (`coproc`)	Executor, Parser	Bidirectional pipe setup, integration with job control	Interactive communication with background processes
Named Pipes and Process Substitution	Parser, Executor	`mkfifo` , `/dev/fd` handling, additional parsing operators	Advanced IPC and command composition

Mental Model: The Shell as a Distributed Operating System Kernel

Advanced extensions transform the shell from a local process launcher into a distributed, programmable runtime environment. Think of it as a microkernel where each component (REPL, executor) becomes a service that can be accessed locally or over the network, and where the command language evolves into a full programming language with variables, control flow, and modularity.

Scripting (Conditionals, Loops, Functions)

Concept: Support shell scripting constructs: `if-then-else` , `for/while` loops, shell functions, and variables with scoping.

Integration Analysis: This is the most significant extension, effectively turning the shell into an interpreter for a scripting language. It requires:

1. **Extended AST:** New node types (`IfNode` , `LoopNode` , `FunctionNode`) beyond the linear `Command` pipeline.
2. **Control Flow Engine:** A new component that evaluates conditional expressions and manages looping execution.
3. **Variable Store:** A scoped symbol table for shell variables (beyond environment variables).
4. **Parser Overhaul:** The recursive descent parser must recognize keywords and structured blocks.

Required Changes:

1. **AST Redesign:** Replace `Command` linked list with a union-based `Node` type that can be `Command` , `Pipeline` , `If` , `While` , `For` , etc.

```c
typedef enum { NODE_COMMAND, NODE_PIPELINE, NODE_IF, NODE_WHILE } NodeType;

typedef struct Node {

    NodeType type;

    union {

        Command* command;

        struct { struct Node* condition; struct Node* then_branch; struct Node* else_branch; } if_stmt;

        struct { struct Node* condition; struct Node* body; } while_loop;

    };

} Node;
```

2. **Parser Rewrite:** The parser becomes significantly more complex, requiring proper handling of block delimiters (`then` , `fi` , `do` , `done`) and nesting.
3. **Control Flow Evaluator:** Add `execute_node(Node*, ShellState*)` that recursively evaluates nodes based on type. Condition evaluation requires arithmetic and test operations (`[` , `test`).
4. **Variable System:** Add `VariableTable` with local and global scopes. Variables like `$foo` expand during tokenization/execution.

> **ADR: Scripting Language Implementation Strategy**
>
> - **Context:** The shell needs to support basic scripting for automation. The current linear pipeline AST is insufficient.
> - **Options Considered:**
>   1. **Embed a Scripting Language (e.g., Lua):** Delegate control flow to an embedded interpreter, exposing shell operations as Lua APIs.
>   2. **Extend AST with Control Flow Nodes:** Enhance the existing parser and executor to handle native shell syntax.
>   3. **Transpile to Another Language:** Convert shell scripts to, e.g., C or Python and execute that.
> - **Decision:** Option 2 (extend AST) for educational completeness and alignment with traditional shell architecture.
> - **Rationale:** Building a control flow engine from scratch provides deep understanding of interpreters and language implementation. It maintains the project's pedagogical focus on systems programming (not just embedding).
> - **Consequences:** Significant increase in complexity; the parser becomes one of the most complex components. Error handling for syntactic errors becomes crucial.

Option	Pros	Cons	Recommended for Learning?
Embed Lua	Robust scripting, less shell code to write	Adds dependency, shell semantics differ from traditional shells, less educational for language implementation	No
Extend AST	Full control, traditional shell semantics, deep learning opportunity	Very complex to implement correctly	Yes (as a capstone)
Transpile	Leverages existing language runtime	Complex transpiler, debugging confusion, performance overhead	No

Tab Completion

Concept: When user presses Tab, suggest completions for partially typed commands, file paths, or variables.

Integration Analysis: Requires real-time interaction: the REPL must intercept Tab key, compute completions based on current context (command word, argument, `$PATH` directories, local files), and update the display immediately. This involves terminal raw mode (like history) and potentially asynchronous I/O for slow operations (network paths).

Required Changes:

1. **REPL Input Overhaul:** Replace line-based input with a character-by-character editor that maintains cursor position and line state.
2. **Completion Engine:** Function `char** get_completions(const char* prefix, CompletionContext ctx)` that returns possible completions. Context derived from parsing the partial line (is it first word? after `cd` ?).
3. **Integration Points:**
   - **Command Completion:** Scan `$PATH` directories for executables matching prefix.
   - **File Completion:** Use `readdir()` on current directory or path prefix.
   - **Variable Completion:** Match against `env` and shell variables.
4. **Display Logic:** If multiple completions, list them (like bash). If single, insert immediately.

Challenges: The parser must be able to parse *partial* input (possibly syntactically invalid) to determine context. This may require a more error-tolerant parsing mode.

Remote Shell (ssh-like server)

Concept: Allow the shell to accept connections over a network socket, authenticate users, and provide a remote command execution environment.

Integration Analysis: This transforms the shell from a local interactive program into a network service. The REPL's input/output must be redirected to/from a network socket instead of `stdin/stdout`. Security becomes paramount (authentication, sandboxing). Job control and signal forwarding must work over network (forwarding SIGINT as a control message).

Required Changes:

1. **Network Layer:** New component `NetworkListener` that binds to a port, accepts connections, spawns a new shell session per connection.
2. **Session Isolation:** Each remote connection gets its own `ShellState` instance, separate job table, working directory (chroot considerations).
3. **REPL Adaptation:** The `run_repl_loop()` function must be generalized to work with any `FILE*` input/output streams (or file descriptors). The prompt may include remote host info.
4. **Authentication:** Simple password or key-based authentication before spawning shell.
5. **Signal Forwarding over Network:** Terminal signals must be translated into protocol messages (e.g., `^C` sends a `SIGINT` message to remote side).

> **Design Insight:** A remote shell exposes the fundamental architecture's **session independence**. Each `ShellState` represents a user session; whether that session's I/O is attached to a local terminal or a network socket is an abstraction the core components need not concern themselves with, provided the REPL uses abstracted I/O functions. This suggests a design refactor: the REPL should receive `input_fd` and `output_fd` parameters rather than assuming `stdin/stdout`.

Job Control Enhancements: `notify` and `disown`

Concept: `notify` (or `set -b`) causes immediate notification when a background job changes state (stops, completes). `disown` removes a job from the job table, letting it run detached from the shell (no longer reported by `jobs`, not sent SIGHUP on shell exit).

Integration Analysis: These are enhancements to the Job Control Manager. `notify` requires modifying the SIGCHLD handler to possibly print notifications synchronously (risky in signal handler) or set a flag checked in the REPL. `disown` requires a new job state `JOB_DISOWNED` and modification of shell termination logic to not send SIGHUP to disowned jobs.

Required Changes:

1. **New Job State:** Add `JOB_DISOWNED` to `JobState` enum.
2. **SIGCHLD Handler:** For `notify`, set a global `volatile sig_atomic_t jobs_changed` flag; in REPL main loop, if flag set, call a function to print status changes.
3. **Disown Built-in:** `builtin_disown()` that sets job state to `JOB_DISOWNED` and removes from `JobTable` (or marks as hidden). The process remains a child but is no longer tracked.
4. **Shell Exit Modification:** In shell termination, iterate through jobs and send SIGHUP only to those not disowned.

Coprocesses (`coproc`)

Concept: Bash's `coproc` creates a background process with a two-way pipe connected to the shell, allowing interactive communication via file descriptors.

Integration Analysis: A coprocess is a hybrid: a background job (managed by Job Control) with additional pipe file descriptors stored in shell variables (e.g., `COPROC_PID`, `COPROC_FD`). The executor must create two pipes (one for stdin to coproc, one for stdout from coproc) and assign them to the child. The shell retains the other ends for later I/O via `read` and `printf` built-ins or redirection.

Required Changes:

1. **Parser Extension:** Recognize `coproc` keyword, optional name, and command.

2. **Executor Enhancement:** `execute_coproc(Command*, ShellState*)` that creates two pipes, forks, sets up child's stdin/stdout accordingly, stores parent's pipe ends and PID in shell variables.
3. **Job Table Integration:** Add as a background job, but with additional metadata (pipe FDs).
4. **I/O Built-ins:** May need `read` built-in to read from coprocess's stdout FD.

Named Pipes and Process Substitution

Concept: Named pipes (FIFOs) allow communication between unrelated processes. Process substitution (`<(command)` , `>(command)`) provides a file-like interface to a command's output/input.

Integration Analysis: Process substitution is a parsing challenge: `diff <(ls dir1) <(ls dir2)` must replace `<(ls dir1)` with a filename (like `/dev/fd/63`) where the command's output is readable. Implementation typically uses anonymous pipes or named FIFOs.

Required Changes:

1. **Parser Recognition:** Detect `<( )` and `>( )` constructs during tokenization; treat them as a special token type.
2. **Executor Handling:** During command setup, for each process substitution:
   - Create a pipe (or named FIFO with `mkfifo()`).
   - Launch the inner command in a subshell, redirecting its stdout (for `<( )` to the pipe write end.
   - Replace the token with the path to the pipe read end (`/dev/fd/N` or FIFO path).
3. **Cleanup:** Ensure pipes/FIFOs are closed and removed after command completes.

Complexities: Synchronization—the main command should not start until the inner commands have started and produced some output? Typically the OS pipe buffer handles this. For `>(command)` , the main command's output goes to a pipe read by the inner command.

> **Design Insight:** Advanced extensions often reveal **implicit assumptions** in the original design. For example, the assumption that all I/O redirections are to files (not command substitutions) is baked into the `Redirection` struct. Adding process substitution might require a new `RedirectionType` (`REDIR_PROCESS_IN` , `REDIR_PROCESS_OUT`) and a union field to store the subcommand AST. This demonstrates how data models evolve with new requirements.

Implementation Guidance

While the core project focuses on the six milestones, these extensions provide excellent pathways for further exploration. The following guidance helps you approach these additions systematically.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
History	Manual terminal raw mode with `tcgetattr` / `tcsetattr`	GNU Readline or libedit integration
Globbing	Implement simple wildcard matching with `opendir` / `readdir`	Use POSIX `glob()` with all flags
Scripting	Extend AST with `if` / `while` nodes, implement tree-walking interpreter	Embed Lua or other scripting language
Tab Completion	Simple file completion based on current directory	Context-aware completion with command-specific rules (like `compgen`)
Remote Shell	Simple TCP server with password authentication, fork per connection	Use SSH protocol (libssh) or TLS encryption

B. Recommended File/Module Structure for Extensions

```
gauntlet-shell/
  src/
    main.c                          # Entry point, calls run_repl_loop
    repl.c/h                        # REPL loop, history, prompt expansion
    parser.c/h                      # Tokenizer, recursive descent parser, alias substitution
    executor.c/h                    # Execution, builtins, redirections, pipelines
    job_control.c/h                 # Job table, signal handlers, fg/bg
    data_structures.c/h             # Command, Job, ShellState definitions
    # Extension modules:
    history.c/h                     # History storage and retrieval
    aliases.c/h                     # Alias table management
    globbing.c/h                    # Wildcard expansion utilities
    scripting/                      # Advanced scripting features
        ast_extended.c/h            # Extended AST nodes (If, While, For)
        interpreter.c/h             # Control flow interpreter
        variables.c/h               # Shell variable table with scoping
    completion.c/h                  # Tab completion engine
    remote/                         # Remote shell server
        server.c/h                  # Network listener
        session.c/h                 # Remote session management
```

C. Infrastructure Starter Code: History with Terminal Raw Mode

Implementing history requires putting the terminal in non-canonical mode to read keys without waiting for Enter. Below is a complete helper module for terminal control.

File: `src/terminal.c`

```c
#include "terminal.h"

#include <termios.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>


static struct termios original_termios;


void terminal_set_raw_mode() {

    struct termios raw;

    tcgetattr(STDIN_FILENO, &original_termios);

    raw = original_termios;

    raw.c_lflag &= ~(ECHO | ICANON | ISIG);  // Disable echo, canonical mode, signals

    raw.c_cc[VMIN] = 1;  // Minimum number of chars for read

    raw.c_cc[VTIME] = 0; // Timeout in deciseconds

    tcsetattr(STDIN_FILENO, TCSANOW, &raw);

}


void terminal_restore() {

    tcsetattr(STDIN_FILENO, TCSANOW, &original_termios);

}


int read_key() {

    char c;

    if (read(STDIN_FILENO, &c, 1) != 1) return -1;

    if (c == '\x1b') { // Escape sequence (arrows)

        char seq[2];

        if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';

        if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

        if (seq[0] == '[') {

            switch (seq[1]) {

                case 'A': return KEY_UP;

                case 'B': return KEY_DOWN;

                case 'C': return KEY_RIGHT;

                case 'D': return KEY_LEFT;

            }
```

```
        }
        return '\x1b';
    }
    return c;
}
```

File: `src/terminal.h`

```c
#ifndef TERMINAL_H
#define TERMINAL_H

#define KEY_UP    1000
#define KEY_DOWN  1001
#define KEY_RIGHT 1002
#define KEY_LEFT  1003

void terminal_set_raw_mode();
void terminal_restore();
int read_key();

#endif
```

D. Core Logic Skeleton Code: Alias Substitution

This function would be called in the REPL after reading a line, before parsing.

File: `src/aliases.c`

```c
#include "aliases.h"

#include <string.h>

#include <stdlib.h>


// Assume AliasTable is defined elsewhere and accessible via shell state


char* expand_aliases(const char* input, AliasTable* table) {

    // TODO 1: Tokenize input into words (simple whitespace split)

    // TODO 2: Check if first word matches any alias in table

    // TODO 3: If matched, replace first word with alias value

    // TODO 4: Re-join words into a single string

    // TODO 5: Recursively expand again (to handle aliases of aliases) up to a limit (e.g., 10)

    // TODO 6: Return newly allocated string (caller must free)

    // TODO 7: If no alias matched, return a copy of input

    // Hint: Use strdup() for copying, strtok() for tokenization (but beware of side effects)

}
```

E. Language-Specific Hints (C)

- **Terminal I/O:** Use `tcgetattr()` / `tcsetattr()` for raw mode. Remember to restore original settings on shell exit (install atexit handler).
- **Signal Safety:** In signal handlers, only use async-signal-safe functions (`write()` , `_exit()` , etc.). Avoid `printf()` , `malloc()` .
- **Memory Management:** For alias and history expansions, always duplicate strings with `strdup()` ; free in the appropriate place (e.g., after command execution).
- **Networking:** For remote shell, use `getaddrinfo()` for portable address resolution. Use `poll()` or `select()` to handle multiple connections.
- **Scripting Interpreter:** For control flow, consider using a recursive evaluation function with a stack for variable scopes.

F. Milestone Checkpoint for History Implementation

After implementing basic history:

1. **Expected Behavior:**
    - Start shell, type a few commands (e.g., `ls` , `pwd`).
    - Press Up arrow: previous command (`pwd`) appears on line.
    - Press Up again: `ls` appears.
    - Press Down: goes forward.
    - Modify line and press Enter: command executes; new line added to history.
2. **Test Command:**

```
$ ./gauntlet
gauntlet> ls
... output ...
gauntlet> pwd
/home/user
gauntlet> [Press Up]
gauntlet> pwd   # appears on line
```

3. **Signs of Trouble:**
   - Arrow keys print `^[[A` etc.: Terminal raw mode not set correctly.
   - History contains only one command: Not storing properly.
   - Shell crashes after pressing Up on empty history: Boundary condition missing.

G. Debugging Tips for Advanced Extensions

Symptom	Likely Cause	How to Diagnose	Fix
Tab completion shows wrong files	Completion context misidentified	Print the token being completed and the computed context	Improve partial-line parser to identify command vs argument
Remote shell disconnects immediately after login	Authentication logic error or session cleanup	Add logging before/after authentication; check if shell forks properly	Ensure the session process stays alive; check for accidental `_exit()`
Script `if` condition always true	Condition expression not evaluated	Print the parsed condition AST; trace execution in interpreter	Implement proper test operators (`[` or `test`)
Coprocess hangs when reading	Deadlock due to full pipe buffer	Check pipe sizes; ensure both parent and child are reading/writing appropriately	Use non-blocking I/O or ensure proper closing of unused pipe ends
Process substitution `diff <(ls) <(ls)` fails with "No such file or directory"	Pipe FD path not correctly passed to command	Print the filename being substituted; check it's a valid open FD	Use `/dev/fd` if available, or create named FIFO with `mkfifo()`

Glossary

> **Milestone(s):** All (1-6) – This reference section defines key terminology used throughout the design document, providing learners with a common vocabulary for understanding shell internals and Unix systems programming concepts.

Terminology Reference

The following table provides alphabetical definitions of key terms used throughout this design document. Each entry includes a clear definition and references where the concept is first introduced or discussed in detail. These definitions bridge the gap between common shell usage and the underlying systems programming concepts needed to implement a shell.

Term	Definition	Reference
Abstract Syntax Tree (AST)	A hierarchical data structure representing the grammatical structure of a parsed command. In the Gauntlet Shell, the AST is implemented as a linked list of `Command` nodes, where each node represents a single command in a pipeline and contains its arguments and redirections. This structure enables the executor to traverse and execute complex command combinations.	First discussed in **Data Model**, elaborated in **Component Design: The Parser**
alias substitution	The process of replacing a command word with a defined string before parsing occurs. Aliases are stored in an `AliasTable` and expanded during the parsing phase. This feature, while a non-goal for the core project, demonstrates how the parser can be extended for user convenience.	Mentioned in **Future Extensions**, referenced in function `expand_aliases`
argv	An array of argument strings passed to the `execvp()` system call, representing the command name and its arguments. The array must be NULL-terminated to indicate its end. In the `Command` struct, `argv` stores the tokenized command and arguments after parsing.	Defined in **Data Model** for `Command` struct, used in **Component Design: The Executor**
background job	A process or pipeline that runs without control of the terminal. Background jobs are launched with a trailing `&` operator, do not receive terminal-generated signals (unless specifically targeted), and allow the shell to display a new prompt immediately. They are tracked in the `JobTable` with state `JOB_RUNNING` (background).	First introduced in **Milestone 5**, detailed in **Component Design: Job Control Manager**
built-in command	A command implemented directly within the shell's executable rather than as an external program. Built-ins have access to and can modify shell internal state (like `cd` changing the working directory or `export` modifying environment variables). The shell checks for built-ins before searching `PATH`.	First introduced in **Milestone 2**, implemented via `execute_builtin()`
Command structure	The core data structure representing a single command in a pipeline. Defined as `struct Command` with fields for `argv` (argument array), `argc` (argument count), `input_redir`, `output_redir` (redirection lists), and `next` (pointer to next command in pipeline). This forms the nodes of the shell's AST.	Defined in **Data Model**, used throughout all executor components
control flow	Programming constructs that alter the order of execution, such as conditionals (`if`) and loops (`while`). While not part of the core project, these would be represented as additional `Node` types in an extended AST and evaluated recursively by the shell's interpreter.	Mentioned in **Future Extensions**
copy-on-write	A memory management optimization used by `fork()` where the parent and child processes initially share the same physical memory pages. Pages are only copied when either process attempts to modify them. This makes `fork()` efficient even for large processes.	Explained in **Component Design: The Executor** ADR
coprocess	A background process with a two-way pipe connection to the shell, allowing the shell to both send input to and read output from the process. An advanced extension beyond simple pipelines.	Mentioned in **Future Extensions**
dup2	The system call `int dup2(int oldfd, int newfd)` that duplicates an existing file descriptor `oldfd` to the specified	First used in **Milestone 3**, detailed in **Component Design: The Executor**

Term	Definition	Reference
	descriptor number `newfd`, closing `newfd` first if it was open. This is the fundamental mechanism for implementing I/O redirection and pipes.	
environment variable	A key-value pair stored in a process's environment, inherited by child processes. The shell maintains environment variables and can modify them via `export`. Child processes receive a copy of the parent's environment at `exec()` time.	First discussed in **Milestone 2**, implemented via `export` built-in
execvp	The system call `int execvp(const char *file, char *const argv[])` that replaces the current process image with a new program. It searches for `file` in the directories listed in the `PATH` environment variable. The `argv` array provides the command-line arguments.	First introduced in **Milestone 1**, used in `execute_external()`
exit status	An integer value returned by a process to indicate success or failure. By convention, 0 indicates success, and non-zero values indicate various failure modes. The shell captures the exit status via `waitpid()` and reports it to the user.	First mentioned in **Milestone 1**, collected in `execute_ast()`
file descriptor	A small non-negative integer that represents an open I/O resource (file, pipe, socket, or device) within a process. The shell manipulates file descriptors 0 (stdin), 1 (stdout), and 2 (stderr) to implement redirections and pipes.	First introduced in **Prerequisites**, fundamental to **Milestone 3**
foreground job	A process or pipeline that has control of the terminal and receives terminal-generated signals (like SIGINT from Ctrl+C). The shell waits for foreground jobs to complete before displaying a new prompt, unless the job is stopped by SIGTSTP.	First introduced in **Milestone 6**, managed via `tcsetpgrp()`
fork	The system call `pid_t fork(void)` that creates a new process by duplicating the calling process. The child process is an exact copy of the parent except for its PID and some other details. Returns 0 in the child, the child's PID in the parent.	First introduced in **Milestone 1**, core to process creation
fork-then-exec pattern	The standard Unix pattern where a process calls `fork()` to create a child, then the child calls `exec()` to replace itself with a new program. This pattern separates process creation from program loading and is used for all external command execution.	Detailed in **Component Design: The Executor** ADR
globbing	Filename expansion using wildcard patterns (`*`, `?`, `[abc]`). The shell expands these patterns to matching filenames before passing arguments to commands. An advanced extension that would integrate with the tokenizer/parser.	Mentioned in **Future Extensions**, function `glob_expand()`
interactive shell	A shell that reads commands from a terminal (tty) and displays prompts. Interactive shells handle job control, signal handling, and command line editing differently from non-interactive (script) shells. The `interactive` field in `ShellState` tracks this mode.	Discussed in **Component Design: The REPL** signal handling
Job structure	The data structure representing a background or suspended job. Defined as `struct Job` with fields for `id` (sequential job number), `pgid` (process group ID), `command_string` (original command), `state` (`JOB_RUNNING`, `JOB_STOPPED`, `JOB_DONE`), and `command_list` (pointer to the AST).	Defined in **Data Model**, managed by **Job Control Manager**

Term	Definition	Reference
job control	The shell feature that allows users to manage multiple jobs (process groups) from a single terminal: starting jobs in background (`&`), suspending (Ctrl+Z), resuming in foreground (`fg`) or background (`bg`), and listing jobs (`jobs`). Requires careful management of process groups and signals.	The focus of **Milestone 5** and **6**
JobState	An enumeration defining the possible states of a job: `JOB_RUNNING` (executing in foreground or background), `JOB_STOPPED` (suspended by SIGTSTP), `JOB_DONE` (terminated but not yet reaped), and potentially `JOB_DISOWNED` (removed from job table but still running).	Defined in **Constants/Enums**, used throughout job management
JobTable	A container structure that holds all active and recently completed jobs. Defined as `struct JobTable` with fields `jobs` (array of Job pointers), `capacity` , `count` , and `next_job_id` . Provides operations for adding, finding, and removing jobs.	Defined in **Data Model**, implemented in **Job Control Manager**
MAX_JOBS	A constant defining the maximum number of jobs the shell's job table can hold (typically 100). When the table is full, the shell may refuse to start new background jobs or evict completed ones.	Defined in **Constants/Enums**, used in job table initialization
MAX_LINE_LEN	A constant defining the maximum length of an input line the shell will accept (typically 4096). Input longer than this will be truncated or rejected to prevent buffer overflows.	Defined in **Constants/Enums**, used in `run_repl_loop()`
Node	A union data structure used in extended shells to represent different types of AST nodes: commands, `if` statements, `while` loops, etc. The `NodeType` enum distinguishes between `NODE_COMMAND` , `NODE_IF` , and `NODE_WHILE` .	Mentioned in **Future Extensions** for control flow
orphaned process group	A process group where the parent of every member is either itself a member of the group or is in a different session. Orphaned process groups that are stopped receive SIGHUP followed by SIGCONT, which may terminate them. The shell must avoid creating orphaned process groups during job control.	Discussed in **Component Design: Job Control Manager** pitfalls
ParseResult	A structure returned by the parser containing either a successfully built AST (`Command* ast`) or an error message (`char* error_message`). The `success` boolean indicates which field is valid. This cleanly separates parsing errors from execution errors.	Defined in **Data Model**, returned by `parse_command_line()`
PATH	An environment variable containing a colon-separated list of directories where the shell searches for executable programs. When a command is not a built-in, the shell searches each directory in `PATH` in order for an executable file matching the command name.	First mentioned in **Milestone 1**, used by `execvp()`
PGID	Process Group ID – a positive integer identifying a collection of related processes (a process group). All processes in a pipeline share the same PGID. The shell uses `setpgid()` to assign processes to groups and `tcsetpgrp()` to give a group terminal control.	Explained in **Component Design: Job Control Manager** ADR
pipe	1. A system call `int pipe(int pipefd[2])` that creates a unidirectional communication channel between processes,	` operator that connects the stdout of one command to the stdin of the next.

Term	Definition	Reference
	returning two file descriptors: `pipefd[0]` for reading and `pipefd[1]` for writing. 2. The `	
pipeline	A sequence of commands connected by pipe operators (`	`), where the output of each command becomes the input of the next. The shell executes all commands in a pipeline concurrently, connecting their stdin/stdout via pipe file descriptors.
process group	A collection of related processes (typically a pipeline) that receive signals as a unit. When the shell runs a foreground pipeline, it places all processes in the same process group and gives that group control of the terminal. Signal delivery to the group ensures all pipeline members receive signals like SIGINT.	Explained in **Component Design: Job Control Manager** ADR
process substitution	A shell feature that allows command output to be used as if it were a file, e.g., `diff <(ls dir1) <(ls dir2)`. Implemented via named pipes (FIFOs) or `/dev/fd` on some systems. An advanced extension beyond simple redirection.	Mentioned in **Future Extensions**
pselect	The POSIX system call `int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t *sigmask)` that waits for I/O readiness with an atomically modified signal mask. Useful for safely waiting for input while handling signals.	Mentioned in **Component Design: The REPL** for safe input waiting
raw mode	A terminal mode where input is read character-by-character without line editing, echoing, or special character processing (like Ctrl+C generating SIGINT). Used for implementing interactive features like tab completion and command line editing.	Mentioned in **Future Extensions**, set by `terminal_set_raw_mode()`
read-eval-print loop (REPL)	The main interactive loop of the shell: read a command line from the user, evaluate (parse and execute) it, print any output, and loop to display the prompt again. The `run_repl_loop()` function implements this pattern, orchestrating the parser, executor, and job control components.	First introduced in **Milestone 1**, detailed in **Component Design: The REPL**
recursive descent	A top-down parsing technique where each grammar rule is implemented as a function that may call other rules recursively. The Gauntlet Shell parser uses recursive descent to handle the precedence of pipe and redirection operators (pipes have lower precedence than redirections).	Explained in **Component Design: The Parser** ADR
Redirection	The data structure representing a single I/O redirection operation. Defined as `struct Redirection` with fields `type` (`REDIR_INPUT`, `REDIR_OUTPUT`, `REDIR_APPEND`, `REDIR_ERROR`), `filename` (target file), `fd` (file descriptor to redirect, defaulting to 0 for input, 1 for output, 2 for error), and `next` (for multiple redirections).	Defined in **Data Model**, set up by `setup_redirections()`
RedirType	An enumeration of redirection types: `REDIR_INPUT` (`<`), `REDIR_OUTPUT` (`>`), `REDIR_APPEND` (`>>`), and `REDIR_ERROR` (`2>`). These values determine how the shell opens the target file and which file descriptor is redirected.	Defined in **Constants/Enums**, used in `Redirection` struct

Term	Definition	Reference
remote shell	A shell that accepts connections over a network, providing remote command execution capability. This would involve replacing stdin/stdout with socket file descriptors and handling authentication and session management. A complex extension beyond local interactive use.	Mentioned in **Future Extensions**, function `run_remote_session()`
self-pipe trick	A technique to convert signals into I/O events by having a signal handler write a byte to a pipe that the main loop monitors with `select()` or `pselect()`. This allows the main loop to handle signals synchronously rather than in signal handlers, simplifying race condition management.	Mentioned in **Component Design: The REPL** as an alternative signal strategy
ShellState	The global state structure containing all shell-wide data: `job_table` (JobTable), `foreground_pgid` (PID of current foreground process group), `terminal_fd` (file descriptor of controlling terminal), `last_exit_status` (exit status of last command), and `interactive` (boolean indicating interactive mode). Initialized by `shell_state_init()`.	Defined in **Data Model**, passed to most component functions
SIGCHLD	The signal sent to a parent process when a child process terminates, stops, or continues. The shell installs a handler for SIGCHLD to reap zombie processes and update the state of jobs in the job table. Must be handled carefully to avoid race conditions.	First mentioned in **Milestone 5**, handled by job control
SIGINT	The interrupt signal typically generated by Ctrl+C. In an interactive shell, SIGINT is sent to the foreground process group, not the shell itself (unless there's no foreground job). The shell arranges this by setting the terminal's foreground process group appropriately.	First mentioned in **Milestone 6**, forwarded by shell
SIGTSTP	The terminal stop signal typically generated by Ctrl+Z. Like SIGINT, it is sent to the foreground process group, causing it to suspend. The shell detects the suspension via SIGCHLD and updates the job's state to `JOB_STOPPED`.	First mentioned in **Milestone 6**, key to job control
sig_atomic_t	An integer type that can be accessed as an atomic entity even in the presence of asynchronous interrupts (signals). Used for signal handler flags to ensure safe communication between signal handlers and the main program. The `volatile` qualifier ensures the compiler doesn't optimize away accesses.	Mentioned in **Component Design: The REPL** signal handling
sigprocmask	The system call `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)` used to block or unblock delivery of signals. The shell blocks signals during critical sections (like modifying the job table) to prevent race conditions, then unblocks them afterward.	Mentioned in **Component Design: Job Control Manager** for safe job table updates
STDERR_FILENO	The integer file descriptor for standard error (2). Used in error redirection (`2>`) and when duplicating file descriptors for pipelines.	Defined in **Constants/Enums** (system constant)
STDIN_FILENO	The integer file descriptor for standard input (0). Used in input redirection (`<`) and when duplicating file descriptors for pipelines.	Defined in **Constants/Enums** (system constant)
STDOUT_FILENO	The integer file descriptor for standard output (1). Used in output redirection (`>` , `>>`) and when duplicating file descriptors for pipelines.	Defined in **Constants/Enums** (system constant)

Term	Definition	Reference
tab completion	An interactive feature that suggests completions for partially typed input when the user presses Tab. Requires raw terminal mode, maintaining a completion context, and searching through possible completions (commands, filenames, variables).	Mentioned in **Future Extensions**, function `get_completions()`
tcsetpgrp	The function `int tcsetpgrp(int fd, pid_t pgrp)` that sets the foreground process group of the terminal associated with file descriptor `fd` to `pgrp`. The shell uses this to give control of the terminal to a foreground job and take it back when the job stops or completes.	Explained in **Component Design: Job Control Manager**
token	The smallest meaningful unit from the input: a word (sequence of non-special characters), an operator (` `, `<`, `>`), or a quoted string (which may contain spaces and special characters). The tokenizer breaks the input line into tokens before the parser builds the AST.	
waitpid	The system call `pid_t waitpid(pid_t pid, int *wstatus, int options)` that waits for a child process to change state (terminate, stop, continue). The shell uses `waitpid` with options `WNOHANG` (non-blocking check) and `WUNTRACED` (report stopped children) to monitor child processes without blocking.	First used in **Milestone 1**, essential for process management
WNOHANG	A flag for `waitpid()` that makes the call non-blocking: it returns immediately if no child has changed state. Used by the shell to check for completed background jobs without pausing the REPL.	Defined in **Constants/Enums**, used in job status checking
WUNTRACED	A flag for `waitpid()` that causes it to return for children that have stopped (not just terminated). Essential for job control to detect when a foreground job is suspended by Ctrl+Z.	Defined in **Constants/Enums**, used in foreground job waiting
zombie process	A process that has terminated but whose exit status hasn't yet been reaped by its parent via `waitpid()`. Zombies consume minimal system resources but clutter process listings. The shell must reap all child processes to avoid zombies, including background jobs.	First mentioned in **Milestone 1** pitfalls, handled via SIGCHLD

Mental Model: Glossary as a Technical Dictionary

Think of this glossary as a **technical dictionary for shell builders**. When learning a new language, you need both the grammar (how to structure sentences) and the vocabulary (the meaning of words). The previous sections provided the grammar of shell design—the architecture, components, and algorithms. This glossary provides the vocabulary—the precise meaning of terms you'll encounter while implementing that design.

Just as a dictionary helps you understand individual words before reading complex literature, this glossary helps you understand the technical terms before delving into systems programming literature or Unix manual pages. Each term connects back to a specific part of the shell's implementation, grounding abstract concepts in concrete data structures and functions.

Understanding Terminology Relationships

While presented alphabetically, these terms relate to each other in meaningful ways that reflect the shell's architecture:

1. **Execution Chain**: `fork` → `execvp` → `waitpid` → **exit status**
2. **I/O Management**: **file descriptor** → `dup2` → **pipe** → **redirection**
3. **Job Control**: **process group** → **PGID** → `tcsetpgrp` → **foreground job**/**background job**
4. **Parsing Pipeline**: **token** → **recursive descent** → **AST** → **Command structure**

5. **Signal Ecosystem**: **SIGCHLD**/**SIGINT**/**SIGTSTP** → **sig_atomic_t** → **sigprocmask** → **WNOHANG**/**WUNTRACED**

These relationships illustrate how the shell's components work together. For example, to implement a pipeline (relationship #2), you need to understand file descriptors, `dup2`, pipes, and redirections. To implement job control (relationship #3), you need to understand process groups, PGIDs, `tcsetpgrp`, and the distinction between foreground and background jobs.

Evolution of Terminology Through Milestones

The terminology evolves in complexity through the project milestones:

- **Milestones 1-2**: Basic terms: `fork`, `execvp`, `waitpid`, **exit status**, **built-in command**, **PATH**, **argv**
- **Milestones 3-4**: I/O terms: **file descriptor**, `dup2`, **pipe**, **redirection**, **pipeline**, **STDIN_FILENO**/**STDOUT_FILENO**/**STDERR_FILENO**
- **Milestones 5-6**: Advanced terms: **process group**, **PGID**, **foreground job**, **background job**, **SIGCHLD**, **SIGINT**, **SIGTSTP**, **WNOHANG**, **WUNTRACED**, **zombie process**, **orphaned process group**

This progression mirrors the learning journey: first mastering process creation, then I/O manipulation, and finally sophisticated job and signal management.

Common Terminology Confusions

> **Key Insight**: Many shell implementation challenges stem from misunderstanding terminology. Clarifying these terms upfront prevents whole classes of bugs.

Several terms are commonly confused by learners:

1. **Process vs. Program**: A *program* is an executable file on disk. A *process* is an instance of that program running in memory. `fork()` creates a new process; `exec()` loads a new program into an existing process.
2. **File Descriptor vs. File**: A *file* is data on disk or a resource like a pipe. A *file descriptor* is an integer handle that a process uses to access that file. Multiple file descriptors (even in different processes) can refer to the same underlying file.
3. **Process Group vs. Pipeline**: A *pipeline* is a shell syntax concept (commands connected by `|`). A *process group* is a kernel mechanism for grouping processes. The shell creates a process group for each pipeline to manage signals collectively.
4. **Stopped vs. Terminated**: A *stopped* process (by SIGTSTP) is paused but can be resumed (by SIGCONT). A *terminated* process has exited and cannot be resumed. Both generate SIGCHLD to the parent but with different status codes.
5. **Foreground Job vs. Interactive Process**: A *foreground job* has control of the terminal. An *interactive process* reads from/writes to a terminal. A foreground job is always interactive, but an interactive process could be in the background if the shell has given terminal control to another group.

Understanding these distinctions is crucial for correct implementation. For example, confusing stopped with terminated processes leads to incorrect job table management; confusing process groups with pipelines leads to incorrect signal delivery.

Terminology in Error Messages

When implementing the shell, you'll encounter system call errors that use this terminology. Understanding these terms helps diagnose issues:

- "Bad file descriptor" (`EBADF`): You passed an invalid file descriptor number to `dup2`, `close`, or `read`/`write`.
- "No child processes" (`ECHILD`): You called `waitpid` when there were no unwaited-for children.
- "Interrupted system call" (`EINTR`): A signal interrupted a blocking call like `read` or `waitpid`. You may need to restart the call.
- "Resource temporarily unavailable" (`EAGAIN`/`EWOULDBLOCK`): Non-blocking operation would block (e.g., `waitpid` with `WNOHANG` and no child state change).

Each error maps directly to terminology in this glossary. For instance, understanding what a "file descriptor" is helps you fix "Bad file descriptor" errors.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Terminology Reference	Manual glossary in project README	Auto-generated from code comments using Doxygen or similar
Cross-reference	Manual links to sections in design doc	Hyperlinked glossary in online documentation

B. Recommended File/Module Structure

Include the glossary as part of your project documentation:

```
gauntlet-shell/
  src/                    # Source code
    main.c                # Entry point with REPL
    parser.c              # Parser implementation
    executor.c            # Executor implementation
    job_control.c         # Job control manager
    builtins.c            # Built-in commands
  include/                # Header files
    shell.h               # Main header with type definitions
    parser.h
    executor.h
    job_control.h
    builtins.h
  docs/                   # Documentation
    DESIGN.md             # Full design document
    GLOSSARY.md           # This glossary (standalone)
    IMPLEMENTATION.md     # Implementation notes
  tests/                  # Test suites
  README.md               # Project overview with link to glossary
```

C. Glossary Usage in Code Comments

Use consistent terminology from this glossary in your code comments to improve readability:

```c
/*
 * execute_pipeline - Execute a pipeline of commands
 *
 * This function implements the fork-then-exec pattern for a pipeline.
 * It creates pipes between commands, forks child processes for each
 * command, sets up redirections, and waits for the entire process group.
 *
 * Terminology references:
 * - Pipeline: series of commands connected by pipe operators
 * - Process group: collection of processes that receive signals together
 * - File descriptor: integer handle for I/O resources
 * - dup2: system call to duplicate file descriptors for redirection
 *
 * Returns: exit status of the last command in the pipeline
 */
int execute_pipeline(Command* pipeline, ShellState* state, int background) {
    // Implementation...
}
```

D. Terminology Validation Script

Create a simple script to check for consistent terminology usage in your codebase:

```bash
#!/bin/bash

# scripts/check_terminology.sh

# Check for consistent use of shell terminology

TERMS=("file descriptor" "process group" "zombie" "foreground job" "background job")

MISSPELLINGS=("file-descriptor" "file_descriptor" "processgroup" "zombie process" "forground")

echo "Checking terminology consistency..."

for term in "${TERMS[@]}"; do

    echo "Checking for '$term':"

    grep -r -i "$term" src/ include/ || echo "  Not found (may be okay)"

done

echo -e "\nChecking for common misspellings:"

for misspelling in "${MISSPELLINGS[@]}"; do

    echo "Checking for '$misspelling':"

    grep -r -i "$misspelling" src/ include/ && echo "  WARNING: Possible misspelling found!"

done
```

E. Language-Specific Hints for C

- Use `man` pages to explore terminology: `man 2 fork`, `man 2 pipe`, `man 2 dup2`
- Include glossary terms in Doxygen comments for automatic documentation generation
- Use `#define` constants for error messages to ensure consistent terminology:

```c
#define ERROR_BAD_FD "Bad file descriptor in dup2"

#define ERROR_NO_CHILD "No child processes to wait for"

#define ERROR_INTERRUPTED "System call interrupted by signal"
```

F. Milestone Checkpoint: Terminology Understanding

After reading this glossary, you should be able to:

1. Explain the difference between a process and a program in your own words
2. Describe how file descriptors enable pipes and redirections
3. Explain why process groups are necessary for correct signal delivery to pipelines
4. Distinguish between stopped, terminated, and zombie processes
5. Understand the purpose of `WNOHANG` and `WUNTRACED` flags to `waitpid`

Test your understanding by explaining these concepts to someone else or writing brief summaries of each.

G. Debugging Tips: Terminology-Related Issues

Symptom	Likely Terminology Misunderstanding	How to Diagnose	Fix
"Shell hangs after pipe command"	Not understanding that **pipes** have limited buffer capacity and all unused ends must be closed	Use `strace -f` to see which processes are blocked on read/write	Close unused pipe ends in parent and children
"Ctrl+C doesn't stop my pipeline"	Not understanding **process groups** and **foreground job** signal delivery	Check `ps -o pid,pgrp,cmd` to see if pipeline processes share a PGID	Use `setpgid()` to group pipeline processes, `tcsetpgrp()` to give them terminal control
"Zombie processes accumulate"	Not understanding **SIGCHLD** handling and **waitpid** with **WNOHANG**	Check `ps aux	grep defunct` for zombies	
"Background jobs not reported when complete"	Not understanding asynchronous **SIGCHLD** delivery timing	Add debug prints to SIGCHLD handler and job table functions	Check job status at prompt display using `job_control_check_completed_jobs()`
"Redirection works but command output disappears"	Not understanding **file descriptor** duplication order with `dup2`	Print file descriptor tables before/after `dup2` calls	Call `dup2` before closing original descriptor, check for errors

This glossary provides the foundational vocabulary for understanding and fixing these issues. When encountering a bug, revisit the relevant terms to ensure you understand the underlying concepts correctly.