Product Requirements Document (PRD)

Simple Shell Development Project

Document Control

Field	Details
Project Name	SimpleSH - Educational Command Line Interface
Document Version	1.0
Status	Draft for Review
Author	Systems Programming Team
Created Date	February 4, 2026
Last Updated	February 4, 2026
Target Platform	Linux/Unix (32-bit & 64-bit)
Difficulty Level	Basic (Entry-level Systems Programming)
Project Type	Educational / Academic Lab Project

Table of Contents

---

1. Executive Summary

1.1 Project Vision

SimpleSH is a basic command-line interface (shell) designed as an educational tool to demonstrate fundamental operating system concepts including process management, inter-process communication, and system call interactions. The project targets students and learners who want hands-on experience with Unix/Linux system programming.

1.2 Problem Statement

Modern production shells (bash, zsh, fish) are complex systems with hundreds of thousands of lines of code, making them difficult for students to understand core shell mechanics. There's a clear need for a simplified, educational shell implementation that:

- Demonstrates core OS concepts (fork, exec, wait)
- Provides hands-on experience with system calls
- Supports multi-user environments
- Implements essential shell features (I/O redirection, piping)
- Maintains code clarity for learning purposes

1.3 Solution Overview

SimpleSH will be a POSIX-compliant command-line interface written in C that supports:

- **Command Execution**: Parse and execute user commands with arguments
- **Process Management**: Fork child processes and execute commands using exec family
- **I/O Redirection**: Support input (`<`) and output (`>`, `>>`) redirection
- **Command Piping**: Enable command chaining using pipes (`|`)
- **Multi-user Support**: Handle concurrent user sessions safely
- **Error Handling**: Provide clear error messages for debugging

1.4 Key Success Indicators

- Successfully execute 95%+ of common Unix commands
- Support concurrent multi-user access without conflicts
- Implement I/O redirection with 100% accuracy
- Process command pipelines with <100ms overhead
- Maintain zero crashes on valid input
- Achieve >80% code maintainability score

---

2. Project Overview

2.1 Background

This project serves as a foundational exercise in operating systems coursework, providing practical understanding of:

- **Process Creation**: How shells create and manage processes
- **Program Execution**: How shells load and execute programs
- **Inter-Process Communication**: How processes share data via pipes
- **File Descriptor Management**: How shells redirect I/O streams
- **System Call Interface**: How user-space programs interact with the kernel

2.2 Scope

In Scope (Version 1.0) Command parsing and tokenization
External command execution (ls, cat, grep, etc.)
Command arguments and parameter passing
Output redirection (`>` and `>>`)
Input redirection (`<`)
Command piping (`|`) for multiple commands
Built-in commands (`cd`, `exit`)
Background process execution (`&`)
Error handling and user feedback
Multi-user concurrent access support
Signal handling for child processes (SIGCHLD)

Out of Scope (Version 1.0) Job control (Ctrl+Z, fg, bg commands)
Command history and readline integration
Tab completion
Shell scripting features (variables, loops, conditionals)
Wildcard expansion (*, ?, [...])
Aliases and shell functions
Advanced signal handling (Ctrl+C propagation)
Command substitution (`$(...)` or backticks)
Environment variable manipulation beyond PATH

2.3 Assumptions

1. Target users have basic Linux/Unix command-line familiarity
2. System provides POSIX-compliant system calls (fork, exec, pipe, dup2)
3. Users will run standard Unix commands available in system PATH
4. Sufficient system resources (memory, process slots) are available
5. File system has standard Unix permissions structure
6. Development will use C programming language with GCC compiler

2.4 Constraints

Category	Constraint
Language	C (ANSI C99 or later)
Platform	Linux/Unix with POSIX support
Dependencies	Standard C library only (no external libraries)
Performance	Educational clarity prioritized over optimization
Security	Basic implementation; not production-hardened
Resource Limits	Use OS default limits (ulimit)

---

3. Goals & Objectives

3.1 Primary Goals

Goal	Description	Success Criteria
Educational Value	Teach OS concepts through practical implementation	Students can explain fork/exec/pipe after using
Functional Completeness	Implement core shell features reliably	95%+ command success rate
Code Quality	Maintain readable, maintainable codebase	<15 cyclomatic complexity per function
Multi-user Support	Handle concurrent users without conflicts	100+ concurrent sessions supported
Robustness	Handle errors gracefully without crashes	Zero segfaults on valid input

3.2 Learning Objectives

Students completing this project will understand:

1. **Process Lifecycle**: Creation, execution, termination
2. **Process Relationships**: Parent-child relationships, process trees
3. **System Calls**: fork(), exec family, wait(), pipe(), dup2()
4. **File Descriptors**: stdin (0), stdout (1), stderr (2), custom FDs
5. **I/O Redirection**: Manipulating file descriptors for I/O control
6. **Inter-Process Communication**: Pipes for process data sharing
7. **Signal Handling**: SIGCHLD for zombie process prevention
8. **Error Handling**: Proper error checking and user feedback

3.3 Business Objectives (Academic Context)

- **Curriculum Integration**: Align with OS course learning outcomes
- **Engagement**: Provide hands-on project for theoretical concepts
- **Assessment**: Enable evaluation of systems programming skills
- **Portfolio Building**: Give students demonstrable project for resumes

---

4. Target Audience

4.1 Primary Users

Profile: Computer Science Students (Sophomore/Junior Level)

- **Background**: Completed introductory programming (C/C++)
- **Experience Level**: Familiar with Linux basics, command-line usage
- **Use Case**: Learning OS concepts, completing coursework
- **Needs**: Clear examples, educational documentation, step-by-step guides

Profile: Self-Taught Systems Programmers

- **Background**: Programming experience in high-level languages
- **Experience Level**: New to systems programming
- **Use Case**: Learning Unix internals, exploring low-level programming
- **Needs**: Comprehensive documentation, reference implementations

4.2 Secondary Users

Profile: Course Instructors/TAs

- **Use Case**: Teaching tool, assignment template
- **Needs**: Modular code for demonstrating concepts, extensibility

Profile: Security Researchers/Penetration Testers

- **Use Case**: Understanding shell mechanics for exploitation research
- **Needs**: Clear implementation of vulnerable patterns (educational)

4.3 User Personas

Persona 1: Raj - OS Course Student

- **Age**: 20
- **Education**: 3rd year Computer Science, Sahyadri College
- **Technical Level**: Intermediate C programmer, basic Linux user
- **Goals**: Understand process management, complete lab assignments
- **Pain Points**: Confused by complex bash source code, needs simpler examples
- **Quote**: *"I want to see how fork and exec actually work together in a real shell"*

Persona 2: Maya - Self-Learner

- **Age**: 24
- **Background**: Web developer transitioning to systems programming
- **Technical Level**: Strong in Python/JavaScript, learning C
- **Goals**: Build systems programming portfolio, understand Unix internals
- **Pain Points**: Lacks structured guidance on system calls
- **Quote**: *"I need a project that teaches me system calls through practical implementation"*

---

5. Functional Requirements

5.1 Core Features

FR-1: Command Input & Parsing

ID	Requirement	Priority	Notes
FR-1.1	Display shell prompt (e.g., `simplesh>`)	Must Have	Indicates ready state
FR-1.2	Read user input from stdin using `fgets()` or `readline()`	Must Have	Max 1024 characters
FR-1.3	Tokenize input into command + arguments	Must Have	Use `strtok()` or custom parser
FR-1.4	Handle multiple arguments separated by spaces	Must Have	Support up to 64 arguments
FR-1.5	Preserve quoted strings as single arguments	Should Have	`"hello world"` → 1 arg
FR-1.6	Trim leading/trailing whitespace	Should Have	Improve parsing robustness
FR-1.7	Identify special operators (`>`, `>>`, `<`, `|`, `&`)	Must Have	For feature detection
FR-1.8	Handle empty input (pressing Enter)	Must Have	Re-display prompt
FR-1.9	Support maximum command line length of 1024 chars	Must Have	Prevent buffer overflow

Data Structure Example:

```c
typedef struct {
    char *command;          // Command name (e.g., "ls")
    char **args;            // NULL-terminated argument array
    char *input_file;       // Input redirection file (NULL if none)
    char *output_file;      // Output redirection file (NULL if none)
    int append_mode;        // 1 for >>, 0 for >
    int background;         // 1 if &, 0 otherwise
} Command;
```

FR-2: Process Management

ID	Requirement	Priority	Notes
FR-2.1	Create child process using `fork()` system call	Must Have	Core functionality
FR-2.2	Execute command in child using `execvp()` or `execve()`	Must Have	Load program
FR-2.3	Parent waits for foreground child completion using `waitpid()`	Must Have	Synchronization
FR-2.4	Display error message if `fork()` fails	Must Have	Resource exhaustion handling
FR-2.5	Display "command not found" if `exec()` fails	Must Have	User feedback
FR-2.6	Prevent zombie processes via proper wait handling	Must Have	Resource cleanup
FR-2.7	Support concurrent child process execution	Must Have	Multiple users/commands
FR-2.8	Handle SIGCHLD signal to reap background processes	Should Have	Automatic cleanup
FR-2.9	Return exit status of executed command	Should Have	For debugging

Process Flow Diagram:

```
         Shell Main Loop
  1. Display prompt
```

```
2. Read input
3. Parse command
4. Check if built-in
     Yes: Execute in parent
     No: Continue




     fork()




 Child         Parent


Setup I/O    If foreground:
Redirection    waitpid()

              If background:
execvp()       Continue loop

 (Exit)        (Loop back)
```

FR-3: Built-in Commands

ID	Requirement	Priority	Notes
FR-3.1	Implement `cd` <directory> to change working directory	Must Have	Uses `chdir()`
FR-3.2	Support `cd` with no args (go to $HOME)	Should Have	User convenience
FR-3.3	Support `cd -` to return to previous directory	Nice to Have	Bash compatibility
FR-3.4	Implement `exit` to terminate shell gracefully	Must Have	Clean shutdown
FR-3.5	Display error for invalid directory in `cd`	Must Have	User feedback
FR-3.6	Execute built-ins without forking	Must Have	Must run in parent

FR-3.7	Implement `help` command (optional)	Nice to Have	Display usage info

Why Built-ins Don't Fork: - Built-in commands like `cd` **must** run in the parent shell process - If `cd` ran in a child, only the child's directory would change - Parent shell's working directory would remain unchanged - Therefore, built-ins are executed directly without `fork()`

FR-4: Output Redirection

ID	Requirement	Priority	Notes
FR-4.1	Support `>` operator to redirect stdout to file (overwrite)	Must Have	Uses `open()` + `dup2()`
FR-4.2	Support `>>` operator to redirect stdout to file (append)	Must Have	O_APPEND flag
FR-4.3	Create output file with permissions 0644 if doesn't exist	Must Have	Standard file permissions
FR-4.4	Overwrite existing file when using `>`	Must Have	Standard behavior
FR-4.5	Append to existing file when using `>>`	Must Have	Standard behavior
FR-4.6	Display error if file cannot be created/opened	Must Have	Permission issues
FR-4.7	Support redirection with commands that have arguments	Must Have	`ls -la > out.txt`
FR-4.8	Close file descriptors after use to prevent leaks	Must Have	Resource management

Example Commands:

```
simplesh> ls -l > directory.txt          # Overwrite
simplesh> echo "Log entry" >> log.txt    # Append
simplesh> ps aux > processes.txt         # Process list to file
```

Implementation Pattern:

```c
// In child process before exec
int fd = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0) {
    perror("open");
    exit(1);
}
dup2(fd, STDOUT_FILENO);  // Redirect stdout to file
close(fd);                // Close original FD
execvp(cmd[0], cmd);      // Execute command
```

FR-5: Input Redirection

ID	Requirement	Priority	Notes
FR-5.1	Support < operator to redirect file to stdin	Must Have	Uses `open()` + `dup2()`
FR-5.2	Display error if input file doesn't exist	Must Have	File not found
FR-5.3	Display error if input file cannot be read	Must Have	Permission denied
FR-5.4	Support input redirection with command arguments	Must Have	`sort -r < data.txt`
FR-5.5	Support simultaneous input and output redirection	Should Have	`sort < in.txt > out.txt`
FR-5.6	Close file descriptors after use	Must Have	Resource management

Example Commands:

```bash
simplesh> wc -l < file.txt                # Count lines from file
simplesh> sort < unsorted.txt             # Sort file contents
simplesh> cat < input.txt > output.txt    # Copy via redirection
```

Implementation Pattern:

```c
// In child process before exec
int fd = open(input_file, O_RDONLY);
if (fd < 0) {
    perror("open");
    exit(1);
}
dup2(fd, STDIN_FILENO);   // Redirect stdin from file
```

```
close(fd);                  // Close original FD
execvp(cmd[0], cmd);        // Execute command
```

FR-6: Command Piping

ID	Requirement	Priority	Notes
FR-6.1	Support | operator to pipe command output to input	Must Have	Core shell feature
FR-6.2	Create pipe using `pipe()` system call	Must Have	Returns 2 FDs
FR-6.3	Connect stdout of command N to stdin of command N+1	Must Have	Chain commands
FR-6.4	Fork separate child process for each piped command	Must Have	Concurrent execution
FR-6.5	Close unused pipe ends in each process	Must Have	Critical for pipe EOF
FR-6.6	Support minimum 2 commands in pipeline	Must Have	Basic piping
FR-6.7	Support up to 10 commands in pipeline	Should Have	Complex chains
FR-6.8	Wait for all pipeline processes to complete	Must Have	Synchronization
FR-6.9	Support combining pipes with I/O redirection	Should Have	Advanced usage
FR-6.10	Handle pipe creation failures gracefully	Must Have	Resource limits

Example Commands:

```
simplesh> ls | wc -l                            # Count files
simplesh> cat file.txt | grep "pattern"         # Filter lines
simplesh> ps aux | grep python | wc -l          # Count Python processes
simplesh> cat data.txt | sort | uniq | wc -l    # Unique line count
```

Pipeline Architecture:

```
Command1  |  Command2  |  Command3


[stdin] → Cmd1 → [pipe1] → Cmd2 → [pipe2] → Cmd3 → [stdout]
```

```
Process Tree:
Shell (parent)
    Child1: Executes Command1 (writes to pipe1)
    Child2: Executes Command2 (reads pipe1, writes pipe2)
    Child3: Executes Command3 (reads pipe2, writes stdout)
```

Critical Pipe Rules: 1. **Each process must close unused pipe ends** - Otherwise pipe won't EOF 2. **Parent must close all pipe ends** - After forking children 3. **Pipes are unidirectional** - Write end → Read end only 4. **Processes execute concurrently** - Not sequential

Implementation Skeleton:

```c
int pipe1[2], pipe2[2];
pipe(pipe1);  // Create first pipe
pipe(pipe2);  // Create second pipe

// First command (writer)
if (fork() == 0) {
    dup2(pipe1[1], STDOUT_FILENO);  // Write to pipe1
    close(pipe1[0]); close(pipe1[1]);
    close(pipe2[0]); close(pipe2[1]);
    execvp(cmd1[0], cmd1);
}

// Second command (middle)
if (fork() == 0) {
    dup2(pipe1[0], STDIN_FILENO);   // Read from pipe1
    dup2(pipe2[1], STDOUT_FILENO);  // Write to pipe2
    close(pipe1[0]); close(pipe1[1]);
    close(pipe2[0]); close(pipe2[1]);
    execvp(cmd2[0], cmd2);
}

// Third command (reader)
if (fork() == 0) {
    dup2(pipe2[0], STDIN_FILENO);   // Read from pipe2
    close(pipe1[0]); close(pipe1[1]);
    close(pipe2[0]); close(pipe2[1]);
    execvp(cmd3[0], cmd3);
}

// Parent closes all pipes and waits
close(pipe1[0]); close(pipe1[1]);
close(pipe2[0]); close(pipe2[1]);
wait(NULL); wait(NULL); wait(NULL);
```

FR-7: Background Process Execution

ID	Requirement	Priority	Notes
FR-7.1	Support `&` operator at end of command for background execution	Should Have	Async execution
FR-7.2	Shell returns prompt immediately for background jobs	Should Have	Non-blocking
FR-7.3	Display background process PID when started	Should Have	User awareness
FR-7.4	Background processes execute independently	Should Have	Don't block shell
FR-7.5	Register SIGCHLD handler to reap zombie processes	Should Have	Automatic cleanup
FR-7.6	Do not propagate Ctrl+C to background processes	Nice to Have	Process isolation

Example Commands:

```
simplesh> sleep 10 &
[1] 12345
simplesh> long_running_task &
[2] 12346
simplesh>
```

FR-8: Error Handling

ID	Requirement	Priority	Notes
FR-8.1	Display "command not found" for invalid commands	Must Have	User feedback
FR-8.2	Display error for file permission issues	Must Have	I/O operations
FR-8.3	Display error for invalid directory in cd	Must Have	Built-in handling

ID	Requirement	Priority	Notes
FR-8.4	Handle fork failure (resource exhaustion)	Must Have	System limits
FR-8.5	Handle pipe creation failure	Must Have	Resource limits
FR-8.6	Use `perror()` for system call errors	Must Have	Standard error format
FR-8.7	Continue shell operation after command failure	Must Have	Robustness
FR-8.8	Validate command syntax before execution	Should Have	Prevent crashes
FR-8.9	Handle signals gracefully (SIGINT, SIGCHLD)	Should Have	Proper cleanup

Error Message Examples:

```
simplesh> invalidcommand
invalidcommand: command not found

simplesh> ls > /root/file.txt
open: Permission denied

simplesh> cd /nonexistent
cd: /nonexistent: No such file or directory
```

5.2 Feature Priority Matrix

Feature	Must Have	Should Have	Nice to Have
Command parsing			
Process creation (fork)			
Command execution (exec)			
Process waiting			
Output redirection (>, »)			
Input redirection (<)			
Command piping (|)			
Built-in: cd			
Built-in: exit			
Error handling			
Background jobs (&)			
SIGCHLD handling			

Feature	Must Have	Should Have	Nice to Have
Quoted argument handling			
Combined I/O + pipes			
cd with no args			
cd - (previous dir)			
Built-in: help			

---

6. Technical Requirements

6.1 Development Environment

Component	**Requirement**	**Version**
Operating System	Linux (Ubuntu/Debian/CentOS/Fedora)	Kernel 3.0+
Compiler	GCC (GNU Compiler Collection)	4.8+
Build Tool	GNU Make	3.8+
Standard Library	Glibc	2.17+
POSIX Compliance	POSIX.1-2008	Required
Debugger	GDB	7.0+ (optional)
Memory Checker	Valgrind	3.10+ (optional)

6.2 System Calls Required

Category	**System Call**	**Purpose**
Process Management	`fork()`	Create child process
	`execvp()` / `execve()`	Execute program
	`wait()` / `waitpid()`	Wait for child completion
	`exit()` / `_exit()`	Terminate process
	`getpid()` / `getppid()`	Get process IDs
File Operations	`open()`	Open/create files
	`close()`	Close file descriptors
	`read()`	Read from file descriptor
	`write()`	Write to file descriptor
	`dup2()`	Duplicate file descriptor

Category	System Call	Purpose
I/O Redirection	`pipe()`	Create pipe
	`dup()` / `dup2()`	Duplicate file descriptors
Directory Operations	`chdir()`	Change working directory
	`getcwd()`	Get current directory
Signal Handling	`signal()` / `sigaction()`	Register signal handlers
	`kill()`	Send signal to process
String Operations	`strtok()`	Tokenize strings
	`strcmp()`	Compare strings
	`strcpy()` / `strncpy()`	Copy strings

6.3 Hardware Requirements

Component	Minimum	Recommended
CPU	1 GHz single-core	2 GHz dual-core
RAM	512 MB	1 GB
Storage	10 MB	50 MB
Architecture	x86 (32-bit)	x86_64 (64-bit)

6.4 Software Dependencies

Zero External Dependencies - Only standard C library (`libc`) required

Compilation Flags:

```
gcc -Wall -Wextra -std=c99 -O2 -o simplesh main.c parser.c process.c
```

Flag Explanations: - `-Wall -Wextra`: Enable all warnings - `-std=c99`: Use C99 standard - `-O2`: Optimization level 2 - `-g`: Debug symbols (development) - `-fsanitize=address`: Address sanitizer (testing)

---

7. System Architecture

7.1 High-Level Architecture

```
                    SimpleSH Shell
                   (User Space)
```

```
      Command          Process          I/O Redir.
       Parser         Management       Module


      Pipeline         Built-in          Error
       Module          Commands         Handler



              Operating System (Kernel Space)
   System Calls: fork, exec, wait, pipe, dup2, open...
```

7.2 Module Architecture

Module 1: Main Shell Loop **File**: `main.c`
Responsibilities: - Initialize shell environment - Display prompt - Read user input - Coordinate module interactions - Handle main loop control flow - Register signal handlers

Key Functions:

```c
int main(void);
void shell_loop(void);
void handle_sigchld(int sig);
```

Module 2: Command Parser **File**: `parser.c`, `parser.h`
Responsibilities: - Tokenize input string into words - Identify special operators (>, <, |, &) - Build command data structures - Validate syntax - Handle quoted strings

Key Functions:

```c
Command* parse_command(char *input);
char** tokenize(char *input);
int detect_operator(char *token);
void free_command(Command *cmd);
```

Data Structures:

```c
typedef struct Command {
    char *name;              // Command executable
    char **args;             // NULL-terminated arg array
    char *input_file;        // Input redirection file
    char *output_file;       // Output redirection file
    int append_mode;         // 0 for >, 1 for >>
    int background;          // 0 foreground, 1 background
    struct Command *next;    // For pipeline (linked list)
} Command;
```

Module 3: Process Management **File**: `process.c`, `process.h`
Responsibilities: - Fork child processes - Execute commands via exec - Wait for process completion - Handle background processes - Manage process exit status

Key Functions:

```c
int execute_command(Command *cmd);
pid_t create_process(void);
int execute_program(char *name, char **args);
void wait_for_child(pid_t pid, int background);
```

Module 4: I/O Redirection **File**: `redirect.c`, `redirect.h`
Responsibilities: - Open files for redirection - Duplicate file descriptors - Redirect stdin/stdout/stderr - Close file descriptors - Handle file errors

Key Functions:

```c
int setup_output_redirect(char *file, int append);
int setup_input_redirect(char *file);
void restore_stdio(int saved_stdin, int saved_stdout);
```

Module 5: Pipeline Handler **File**: `pipeline.c`, `pipeline.h`
Responsibilities: - Create pipes between commands - Connect process file descriptors - Manage multiple pipes - Close unused pipe ends - Wait for pipeline completion

Key Functions:

```c
int execute_pipeline(Command *cmd_list);
int create_pipe_chain(Command *cmds[], int count);
void connect_pipe(int pipe_fds[2], int input, int output);
```

Pipeline Data Flow:

```
cmd1 | cmd2 | cmd3

Pipe Creation:
pipe1[2] = {read_end1, write_end1}
pipe2[2] = {read_end2, write_end2}

Process 1: stdin → cmd1 → pipe1[write]
Process 2: pipe1[read] → cmd2 → pipe2[write]
Process 3: pipe2[read] → cmd3 → stdout
```

Module 6: Built-in Commands **File**: `builtins.c`, `builtins.h`
Responsibilities: - Implement cd command - Implement exit command - Implement help command (optional) - Check if command is built-in - Execute built-ins in parent process

Key Functions:

```c
int is_builtin(char *command);
int execute_builtin(Command *cmd);
int builtin_cd(char **args);
int builtin_exit(char **args);
int builtin_help(char **args);
```

Module 7: Error Handler **File**: `error.c`, `error.h`
Responsibilities: - Format error messages - Print errors to stderr - Log errors (optional) - Return appropriate error codes

Key Functions:

```c
void print_error(const char *msg);
void print_system_error(const char *msg);
void print_command_error(const char *cmd);
```

7.3 File Structure

```
simplesh/

  src/                           # Source files
     main.c                      # Entry point, main loop
     parser.c                    # Command parsing
     process.c                   # Process management
     redirect.c                  # I/O redirection
     pipeline.c                  # Pipe handling
```

```
    builtins.c                # Built-in commands
    error.c                   # Error handling

include/                      # Header files
    shell.h                   # Main header, structures
    parser.h                  # Parser prototypes
    process.h                 # Process prototypes
    redirect.h                # Redirect prototypes
    pipeline.h                # Pipeline prototypes
    builtins.h                # Builtin prototypes
    error.h                   # Error prototypes

tests/                        # Test files
    test_parser.c             # Parser unit tests
    test_process.c            # Process unit tests
    test_redirect.c           # Redirection tests
    test_pipeline.c           # Pipeline tests
    integration_tests.sh      # Integration test script
    run_all_tests.sh          # Test runner

docs/                         # Documentation
    README.md                 # Project overview
    USAGE.md                  # User guide
    ARCHITECTURE.md           # Technical design
    EXAMPLES.md               # Command examples
    CONTRIBUTING.md           # Contribution guide

examples/                     # Example scripts
    basic_commands.txt        # Simple examples
    redirection_examples.txt  # I/O redirection
    pipeline_examples.txt     # Pipe examples

Makefile                      # Build configuration
.gitignore                    # Git ignore rules
LICENSE                       # License file
README.md                     # Quick start guide
```

7.4 Data Flow Diagram

User Command Execution Flow:

```
  User

      (1) Input command
```

```
  Main Loop
  - Display
    prompt
  - Read input

          (2) Raw string


  Parser
  - Tokenize
  - Detect ops
  - Build struct

          (3) Command struct

      Is Built-in?

       Yes       No


 Builtins     Process Mgr
 Execute      - Fork
 in parent    - Setup I/O
        - Exec
             - Wait



           (4) Exit status


  Main Loop
  - Check status
  - Loop back
```

7.5 Process Relationship Diagram

Multi-user Scenario:

```
            Operating System Kernel
  Process Table | File Descriptor Table | ...
```

```
  SimpleSH       SimpleSH       SimpleSH
  (User 1)       (User 2)       (User 3)
  PID: 1001      PID: 1050      PID: 1100




 Child1  Child2  Child1  Child2  Child1  Child2
 (ls)    (grep)  (ps)    (cat)   (find)  (wc)

Each shell instance operates independently
Children inherit file descriptors from parent shell
```

---

8. User Stories & Use Cases

8.1 User Stories

Epic 1: Basic Command Execution US-1.1: Execute Simple Commands
As a shell user
I want to execute basic Unix commands
So that I can interact with the file system

Acceptance Criteria: - Shell displays prompt - User types command (e.g., `ls`, `pwd`, `date`) - Command executes and displays output - Shell returns to prompt after completion

Technical Notes: - Use `fork()` to create child - Use `execvp()` to execute command - Parent waits with `waitpid()`

US-1.2: Execute Commands with Arguments
As a shell user
I want to pass arguments to commands
So that I can customize command behavior

Acceptance Criteria: - Commands accept multiple arguments (e.g., `ls -la /home`) - Arguments are parsed correctly - Quoted arguments treated as single unit

Example Commands:

```
simplesh> ls -l -a
simplesh> grep "error" log.txt
simplesh> gcc -Wall -o program source.c
```

Epic 2: I/O Redirection US-2.1: Redirect Output to File
As a shell user
I want to save command output to a file
So that I can store results for later analysis

Acceptance Criteria: - `>` operator creates/overwrites file - `>>` operator appends to file - File created if doesn't exist - Error shown if permission denied

Example Commands:

```
simplesh> ls -l > files.txt
simplesh> echo "Entry" >> log.txt
simplesh> ps aux > processes.txt
```

US-2.2: Redirect Input from File
As a shell user
I want to feed file contents as command input
So that I can process file data

Acceptance Criteria: - `<` operator reads from file - Error if file doesn't exist - Works with command arguments

Example Commands:

```
simplesh> wc -l < file.txt
simplesh> sort < unsorted.txt
simplesh> grep "pattern" < data.txt
```

Epic 3: Command Piping US-3.1: Chain Two Commands
As a shell user
I want to pipe output of one command to another
So that I can process data through multiple stages

Acceptance Criteria: - `|` operator connects commands - Output of cmd1 becomes input of cmd2 - Both commands run concurrently - Result displayed after both complete

Example Commands:

```
simplesh> ls | wc -l
simplesh> cat file.txt | grep "error"
simplesh> ps aux | grep python
```

US-3.2: Chain Multiple Commands
As a shell user
I want to create complex command pipelines
So that I can perform sophisticated data processing

Acceptance Criteria: - Support 3+ commands in pipeline - Each command receives previous output - Final result displayed correctly - All processes execute concurrently

Example Commands:

```
simplesh> cat data.txt | sort | uniq | wc -l
simplesh> ps aux | grep python | awk '{print $2}'
simplesh> ls -l | grep "\.txt" | wc -l
```

Epic 4: Background Jobs **US-4.1: Run Command in Background**
As a shell user
I want to execute long-running commands in background
So that I can continue using shell

Acceptance Criteria: - `&` operator starts background job - Shell returns prompt immediately - Background process PID displayed - No zombie processes accumulate

Example Commands:

```
simplesh> sleep 30 &
[1] 12345
simplesh> ./long_task &
[2] 12346
simplesh>
```

Epic 5: Built-in Commands **US-5.1: Change Directory**
As a shell user
I want to navigate the file system
So that I can work in different directories

Acceptance Criteria: - `cd <dir>` changes working directory - `cd` with no args goes to home - Error shown for invalid directory - Works without forking child

Example Commands:

```
simplesh> cd /home/user
simplesh> cd Documents
simplesh> cd ..
simplesh> cd
```

US-5.2: Exit Shell
As a shell user
I want to terminate the shell cleanly
So that I can return to parent shell

Acceptance Criteria: - `exit` command terminates shell - All resources cleaned up - Returns to parent shell/terminal

---

8.2 Use Case Scenarios

Use Case 1: File Search with Filtering **Actor**: Student searching for files
Precondition: User is in project directory
Trigger: User wants to find all .c files with "main"

Main Flow: 1. User types: `ls -l | grep "\.c" | grep "main"` 2. Shell parses command into 3-stage pipeline 3. Shell creates 2 pipes 4. Shell forks 3 child processes 5. Process 1 executes `ls -l`, writes to pipe1 6. Process 2 executes `grep "\.c"`, reads pipe1, writes pipe2 7. Process 3 executes `grep "main"`, reads pipe2, writes stdout 8. Shell waits for all 3 processes 9. Filtered results displayed 10. Shell returns to prompt

Alternative Flow: - 4a. Pipe creation fails → Display error, return to prompt - 6a. No .c files found → grep exits with status 1, continue - 8a. User presses Ctrl+C → Processes terminate, return to prompt

---

Use Case 2: Log File Analysis **Actor**: System administrator
Precondition: Server log file exists
Trigger: Admin wants to count error entries

Main Flow: 1. User types: `grep "ERROR" < server.log | wc -l > error_count.txt` 2. Shell parses: input redirect, pipe, output redirect 3. Shell forks child for grep - Opens server.log for reading - Redirects stdin from file - Connects stdout to pipe 4. Shell forks child for wc - Connects stdin from pipe - Opens error_count.txt for writing - Redirects stdout to file 5. Both processes execute concurrently 6. grep filters error lines → pipe → wc counts → file 7.

Shell waits for both children 8. Result saved in error_count.txt 9. Shell returns to prompt

Alternative Flow: - 3a. server.log doesn't exist → Error message, return to prompt - 3b. Permission denied on server.log → Error message, return - 4a. Cannot create error_count.txt → Error message, return

---

Use Case 3: Multi-User Environment **Actors**: 3 simultaneous users
Precondition: SimpleSH running on multi-user Linux system
Trigger: Multiple users SSH into system

Main Flow: 1. User 1 logs in, starts SimpleSH (PID 1001) 2. User 2 logs in, starts SimpleSH (PID 1050) 3. User 3 logs in, starts SimpleSH (PID 1100) 4. User 1 executes: `ls -R / > user1_files.txt &` - Background job started (PID 2001) - Shell returns prompt immediately 5. User 2 executes: `find /home -name "*.txt" | wc -l` - Foreground pipeline created - Shell waits for completion 6. User 3 executes: `top` - Interactive command takes over terminal 7. User 1's background job completes - SIGCHLD signal sent - Zombie process reaped 8. All users continue working independently

Key Points: - Each shell instance has separate process space - File descriptors not shared between users - Background jobs don't interfere - OS handles process isolation

---

9. Non-Functional Requirements

9.1 Performance Requirements

ID	Requirement	Target	Measurement
NFR-P1	Command execution overhead	<50ms	Time from enter to exec
NFR-P2	Shell memory footprint	<10MB	RSS in `ps` output
NFR-P3	Pipe throughput	>100MB/s	Large file piping
NFR-P4	Maximum concurrent children	100+	Stress test
NFR-P5	Command parsing time	<10ms	1024 char input
NFR-P6	Startup time	<100ms	Shell initialization
NFR-P7	Response time for built-ins	<5ms	cd, exit commands

9.2 Reliability Requirements

ID	Requirement	Target	Measurement
NFR-R1	Uptime for valid commands	>99.9%	No crashes on valid input
NFR-R2	Graceful error handling	100%	All errors caught
NFR-R3	Memory leak prevention	Zero leaks	Valgrind test
NFR-R4	File descriptor leak prevention	Zero leaks	`lsof` monitoring
NFR-R5	Zombie process prevention	Zero zombies	`ps` monitoring
NFR-R6	Recovery from system call failures	100%	Error handling tests

9.3 Usability Requirements

ID	Requirement	Target	Measurement
NFR-U1	Command syntax compatibility	Bash-like	User familiarity
NFR-U2	Error message clarity	>80% comprehension	User testing
NFR-U3	Prompt visibility	Clear & consistent	Visual inspection
NFR-U4	Learning curve	<2 hours	User onboarding time
NFR-U5	Documentation completeness	>90% coverage	Doc review

9.4 Maintainability Requirements

ID	Requirement	Target	Measurement
NFR-M1	Code documentation	>60%	Comment ratio
NFR-M2	Function complexity	<15 cyclomatic	Static analysis
NFR-M3	Module coupling	Low	Architecture review
NFR-M4	Code style consistency	100%	Style checker
NFR-M5	Build time	<5 seconds	`time make`

9.5 Portability Requirements

ID	Requirement	Target	Measurement
NFR-PO1	Linux distribution support	Ubuntu, Debian, CentOS, Fedora	Test matrix
NFR-PO2	Unix-like OS support	macOS, BSD	Compatibility testing
NFR-PO3	Architecture support	x86, x86_64, ARM	Multi-arch build
NFR-PO4	POSIX compliance	100% for used calls	Standards review
NFR-PO5	Compiler compatibility	GCC 4.8+, Clang 3.5+	Build testing

9.6 Security Requirements

ID	Requirement	Target	Notes
NFR-S1	Buffer overflow prevention	Zero occurrences	Use bounded functions
NFR-S2	Input validation	All user input	Prevent injection
NFR-S3	Resource limit enforcement	OS defaults	Use ulimit
NFR-S4	Privilege escalation prevention	No vulnerabilities	Security audit
NFR-S5	File permission respect	100% compliance	Honor system ACLs

Security Disclaimer: This is an educational project. It should NOT be used as a production shell or in security-sensitive environments.

---

10. Implementation Phases

Phase 1: Foundation (Week 1)

Goal: Basic shell loop with simple command execution

Tasks: 1. Setup project structure (directories, Makefile) 2. Implement main loop - Display prompt - Read user input (using `fgets()`) - Handle empty input 3. Implement exit command 4. Basic command execution (no arguments) - Fork child process - Execute command with `execvp()` - Parent waits for child 5. Basic error handling

Deliverables: - Shell displays prompt - Executes single-word commands (`ls`, `pwd`, `date`) - Exit command works - Basic error messages

Success Criteria:

```
simplesh> pwd
/home/user
simplesh> date
Wed Feb  4 07:59:00 IST 2026
simplesh> ls
file1.txt  file2.txt  directory/
simplesh> exit
```

Test Cases: - TC-1.1: Display prompt correctly - TC-1.2: Execute `ls` successfully - TC-1.3: Execute `pwd` successfully - TC-1.4: Exit cleanly with `exit` command - TC-1.5: Handle invalid command gracefully

---

Phase 2: Argument Parsing (Week 2)

Goal: Support commands with arguments and built-in cd

Tasks: 1. Implement command parser - Tokenize input string - Build argument array - Handle multiple arguments 2. Update process execution to use arguments 3. Implement `cd` built-in command - Change directory with `chdir()` - Handle errors (invalid path) - Support `cd` with no args (home directory) 4. Enhanced error handling

Deliverables: - Commands with arguments work - Parser handles multiple spaces - `cd` command functional - Better error messages

Success Criteria:

```
simplesh> ls -l -a
total 24
drwxr-xr-x 3 user user 4096 Feb  4 08:00 .
drwxr-xr-x 8 user user 4096 Feb  3 12:30 ..
simplesh> cd /tmp
simplesh> pwd
/tmp
simplesh> cd /nonexistent
cd: /nonexistent: No such file or directory
```

Test Cases: - TC-2.1: Execute `ls -la` - TC-2.2: Execute `grep "pattern" file.txt` - TC-2.3: Change directory with `cd /tmp` - TC-2.4: Handle invalid directory in `cd` - TC-2.5: `cd` with no args goes to home

---

Phase 3: I/O Redirection (Week 3)

Goal: Implement input and output redirection

Tasks: 1. Enhance parser to detect `>`, `>>`, `<` operators 2. Implement output redirection - Open file with appropriate flags - Use `dup2()` to redirect stdout - Close file descriptors 3. Implement input redirection - Open file for reading - Use `dup2()` to redirect stdin 4. Support combined I/O redirection 5. Add file operation error handling

Deliverables: - Output redirection (`>`, `>>`) works - Input redirection (`<`) works - Combined redirection works - File errors handled gracefully

Success Criteria:

```
simplesh> ls -l > output.txt
simplesh> cat output.txt
# (contents of ls -l)
simplesh> echo "New line" >> output.txt
simplesh> wc -l < output.txt
11
simplesh> sort < input.txt > sorted.txt
```

Test Cases: - TC-3.1: Redirect output with `>` - TC-3.2: Append output with `>>` - TC-3.3: Redirect input with `<` - TC-3.4: Combined: `sort < in.txt > out.txt` - TC-3.5: Handle permission denied - TC-3.6: Handle file not found

---

Phase 4: Command Piping (Week 4)

Goal: Implement command pipelines

Tasks: 1. Enhance parser to detect `|` operator 2. Implement pipe for 2 commands - Create pipe with `pipe()` - Fork children for each command - Connect stdout of cmd1 to stdin of cmd2 - Close unused pipe ends 3. Extend to support 3+ commands 4. Support pipes with I/O redirection 5. Proper wait for all pipeline processes

Deliverables: - Basic 2-command pipeline works - Multiple command pipeline (3+) works - Pipes with redirection work - All processes properly reaped

Success Criteria:

```
simplesh> ls | wc -l
15
simplesh> cat file.txt | grep "error" | wc -l
3
simplesh> ps aux | grep python | awk '{print $2}'
12345
```

```
12346
simplesh> ls | sort | uniq | wc -l
12
```

Test Cases: - TC-4.1: Two-command pipe: `ls | wc -l` - TC-4.2: Three-command pipe: `cat | sort | uniq` - TC-4.3: Four-command pipe - TC-4.4: Pipe with output redirect: `ls | grep txt > files.txt` - TC-4.5: Handle broken pipe gracefully

---

Phase 5: Background Jobs & Polish (Week 5)

Goal: Background execution and final refinements

Tasks: 1. Implement background job execution - Detect `&` operator - Fork without waiting - Display background PID 2. Implement SIGCHLD handler - Reap zombie processes - Non-blocking waitpid 3. Code cleanup and refactoring 4. Comprehensive testing 5. Documentation completion 6. Bug fixes

Deliverables: - Background jobs work - No zombie processes - All features integrated - Complete documentation - Passing test suite

Success Criteria:

```
simplesh> sleep 10 &
[1] 12345
simplesh> ps
  PID TTY          TIME CMD
12340 pts/0    00:00:01 bash
12344 pts/0    00:00:00 simplesh
12345 pts/0    00:00:00 sleep
simplesh> # shell immediately available
```

Test Cases: - TC-5.1: Background job execution - TC-5.2: Multiple background jobs - TC-5.3: No zombie processes accumulate - TC-5.4: Background job with pipeline - TC-5.5: Stress test: 50 background jobs

---

Phase 6: Testing & Deployment (Week 6)

Goal: Comprehensive testing and deployment preparation

Tasks: 1. Unit testing for all modules 2. Integration testing 3. Stress testing 4. Memory leak testing (Valgrind) 5. Multi-user testing 6. Documentation review 7. Code review and optimization 8. Release preparation

Deliverables: - >80% unit test coverage - All integration tests pass - Zero memory leaks - Performance benchmarks met - Complete user documentation

---

Implementation Timeline

```
Week 1: Foundation
  Day 1-2: Project setup, main loop
  Day 3-4: Basic command execution
  Day 5-7: Testing, documentation

Week 2: Arguments & Parsing
  Day 1-2: Parser implementation
  Day 3-4: Built-in commands
  Day 5-7: Testing, integration

Week 3: I/O Redirection
  Day 1-2: Output redirection
  Day 3-4: Input redirection
  Day 5-7: Combined I/O, testing

Week 4: Command Piping
  Day 1-2: Two-command pipeline
  Day 3-4: Multiple pipelines
  Day 5-7: Integration, testing

Week 5: Background & Polish
  Day 1-2: Background jobs
  Day 3-4: Signal handling
  Day 5-7: Code cleanup, docs

Week 6: Testing & Deployment
  Day 1-3: Comprehensive testing
  Day 4-5: Bug fixes, optimization
  Day 6-7: Final review, release
```

---

11. Testing Strategy

11.1 Testing Pyramid

```
       Manual       ← 5% (Exploratory)
      Testing

    Integration     ← 25% (End-to-end)
      Testing

        Unit        ← 70% (Component)
      Testing
```

11.2 Unit Testing

Module: Parser

Test Case	Input	Expected Output	Status
UT-P01	`"ls -la"`	`cmd="ls", args=["ls", "-la", NULL]`	
UT-P02	`"ls > file.txt"`	`cmd="ls", output_file="file.txt"`	
UT-P03	`"cat < in.txt"`	`cmd="cat", input_file="in.txt"`	
UT-P04	`"ls | wc"`	`2 commands in pipeline`	
UT-P05	`"sleep 10 &"`	`cmd="sleep", background=1`	
UT-P06	`""` (empty)	`NULL command`	
UT-P07	`"  ls  -l  "`	`Whitespace trimmed correctly`	

Module: Process Management

Test Case	Description	Expected Result	Status
UT-PR01	Fork child successfully	`pid > 0` in parent	
UT-PR02	Execute valid command	Command runs, returns 0	
UT-PR03	Execute invalid command	Error message, returns 1	
UT-PR04	Wait for child completion	Child reaped, no zombie	
UT-PR05	Handle fork failure	Error message displayed	

Module: I/O Redirection

Test Case	Description	Expected Result	Status
UT-IO01	Redirect stdout to file	File created with output	
UT-IO02	Append stdout to file	Content appended	
UT-IO03	Redirect stdin from file	File contents read	
UT-IO04	Handle file not found	Error message, no crash	
UT-IO05	Handle permission denied	Error message shown	
UT-IO06	Close FDs after use	No FD leaks (lsof check)	

Module: Pipeline

Test Case	Description	Expected Result	Status
UT-PI01	Create pipe successfully	`pipe_fds[2]` valid	
UT-PI02	Connect 2 commands	Output of cmd1 → input of cmd2	

Test Case	Description	Expected Result	Status
UT-PI03	Close unused pipe ends	Only necessary FDs open	
UT-PI04	Wait for all children	All processes reaped	
UT-PI05	Handle pipe creation failure	Error message, graceful exit	

11.3 Integration Testing

Test Suite 1: Basic Commands

```bash
#!/bin/bash
# integration_test_basic.sh

echo "=== Basic Command Tests ==="

# Test 1: Simple command
./simplesh <<< "ls" > output.txt
if [ $? -eq 0 ]; then echo "  Test 1 PASS"; else echo "  Test 1 FAIL"; fi

# Test 2: Command with arguments
./simplesh <<< "ls -la" > output.txt
if [ $? -eq 0 ]; then echo "  Test 2 PASS"; else echo "  Test 2 FAIL"; fi

# Test 3: Built-in cd
./simplesh <<< $'cd /tmp\npwd\nexit' > output.txt
if grep -q "/tmp" output.txt; then echo "  Test 3 PASS"; else echo "  Test 3 FAIL"; fi

# Test 4: Exit command
./simplesh <<< "exit" > output.txt
if [ $? -eq 0 ]; then echo "  Test 4 PASS"; else echo "  Test 4 FAIL"; fi

# Test 5: Invalid command
./simplesh <<< "invalidcommand123" > output.txt 2>&1
if grep -q "not found" output.txt; then echo "  Test 5 PASS"; else echo "  Test 5 FAIL"; fi
```

Test Suite 2: I/O Redirection

```bash
#!/bin/bash
# integration_test_io.sh

echo "=== I/O Redirection Tests ==="

# Test 6: Output redirection
./simplesh <<< "ls > test_output.txt"
```

```bash
if [ -f test_output.txt ]; then echo "  Test 6 PASS"; else echo "  Test 6 FAIL"; fi

# Test 7: Output append
./simplesh <<< $'echo "line1" > test_append.txt\necho "line2" >> test_append.txt'
if [ $(wc -l < test_append.txt) -eq 2 ]; then echo "  Test 7 PASS"; else echo "  Test 7 FAIL"

# Test 8: Input redirection
echo "test content" > test_input.txt
./simplesh <<< "cat < test_input.txt" > test_output2.txt
if grep -q "test content" test_output2.txt; then echo "  Test 8 PASS"; else echo "  Test 8 FA

# Test 9: Combined I/O
echo -e "zebra\napple\nbanana" > unsorted.txt
./simplesh <<< "sort < unsorted.txt > sorted.txt"
if [ "$(head -1 sorted.txt)" = "apple" ]; then echo "  Test 9 PASS"; else echo "  Test 9 FAIL
```

Test Suite 3: Pipelines

```bash
#!/bin/bash
# integration_test_pipes.sh

echo "=== Pipeline Tests ==="

# Test 10: Two-command pipe
result=$(./simplesh <<< "echo test | cat")
if [ "$result" = "test" ]; then echo "  Test 10 PASS"; else echo "  Test 10 FAIL"; fi

# Test 11: Three-command pipe
echo -e "line1\nline2\nline1" > pipe_test.txt
result=$(./simplesh <<< "cat pipe_test.txt | sort | uniq | wc -l")
if [ "$result" -eq 2 ]; then echo "  Test 11 PASS"; else echo "  Test 11 FAIL"; fi

# Test 12: Pipe with redirection
./simplesh <<< "ls | grep 'txt' > pipe_output.txt"
if [ -f pipe_output.txt ]; then echo "  Test 12 PASS"; else echo "  Test 12 FAIL"; fi
```

Test Suite 4: Background Jobs

```bash
#!/bin/bash
# integration_test_background.sh

echo "=== Background Job Tests ==="

# Test 13: Background execution
./simplesh <<< $'sleep 2 &\necho "immediate"\nexit' > bg_test.txt
if grep -q "immediate" bg_test.txt; then echo "  Test 13 PASS"; else echo "  Test 13 FAIL"; f

# Test 14: No zombie processes
```

```
./simplesh <<< $'sleep 1 &\nsleep 1 &\nsleep 1 &\nexit'
sleep 2
zombies=$(ps aux | grep 'Z' | grep -v grep | wc -l)
if [ "$zombies" -eq 0 ]; then echo "  Test 14 PASS"; else echo "  Test 14 FAIL"; fi
```

11.4 Stress Testing

Test 1: High Command Volume

```
for i in {1..1000}; do
    echo "pwd" | ./simplesh > /dev/null
done
echo "  1000 sequential commands executed"
```

Test 2: Concurrent Users

```
for i in {1..20}; do
    ./simplesh <<< "ls -R / > /tmp/output_$i.txt" &
done
wait
echo "  20 concurrent shell instances completed"
```

Test 3: Large Pipeline

```
# Create large file
seq 1 1000000 > large_file.txt

# Process through multiple pipes
time ./simplesh <<< "cat large_file.txt | sort | uniq | wc -l"
# Should complete in reasonable time (<10s)
```

Test 4: Memory Leak Test

```
valgrind --leak-check=full --show-leak-kinds=all ./simplesh <<< $'ls\npwd\nexit'
# Expected: "All heap blocks were freed -- no leaks are possible"
```

Test 5: File Descriptor Leak Test

```
# Before
fd_before=$(lsof -p $(pgrep simplesh) | wc -l)

# Execute 100 commands with I/O redirection
for i in {1..100}; do
    echo "ls > /tmp/test_$i.txt" | ./simplesh
done

# After
fd_after=$(lsof -p $(pgrep simplesh) | wc -l)

if [ "$fd_before" -eq "$fd_after" ]; then
```

```
    echo "  No file descriptor leaks"
else
    echo "  FD leak detected: $fd_before → $fd_after"
fi
```

11.5 Test Coverage Goals

Component	Coverage Target	Measurement
Parser	>85%	gcov/lcov
Process Management	>80%	gcov/lcov
I/O Redirection	>85%	gcov/lcov
Pipeline	>80%	gcov/lcov
Built-ins	>90%	gcov/lcov
Error Handling	>75%	Manual review
Overall	**>80%**	gcov/lcov

11.6 Testing Tools

Tool	Purpose	Usage
GCC with gcov	Code coverage analysis	`gcc -fprofile-arcs -ftest-coverage`
Valgrind	Memory leak detection	`valgrind --leak-check=full ./simplesh`
AddressSanitizer	Memory error detection	`gcc -fsanitize=address`
lsof	File descriptor monitoring	`lsof -p <pid>`
ps	Process monitoring	`ps aux | grep simplesh`
strace	System call tracing	`strace -f ./simplesh`

---

12. Security Considerations

12.1 Educational Context Disclaimer

CRITICAL: This shell is designed for **EDUCATIONAL PURPOSES ONLY**. It is NOT production-ready and should NEVER be used as: - A login shell (`/bin/sh`, `/bin/bash` replacement) - A system shell for scripts - A security-critical application - An internet-facing service

12.2 Known Security Limitations

Issue	Risk Level	Impact	Mitigation (Educational)
Command Injection	Critical	Malicious code execution	Input validation, bounded buffers
Buffer Overflow	Critical	Code execution, crash	Use `strncpy`, check buffer sizes
Path Traversal	Medium	Unauthorized file access	Validate `cd` paths
Resource Exhaustion	Medium	DoS via fork bomb	Rely on OS ulimits
Race Conditions	Low	Zombie processes	Proper signal handling
TOCTOU (Time-of-check-time-of-use)	Low	File manipulation	Not addressed (educational)

12.3 Security Best Practices Implemented

Input Validation - Maximum command length enforced (1024 chars) - Maximum argument count enforced (64 args) - NULL termination of strings - Buffer overflow checks

Resource Management - Proper file descriptor closure - Process reaping (zombie prevention) - Memory deallocation - Signal handler registration

Error Handling - System call return value checking - Error messages without sensitive information - Graceful degradation

Secure Coding Practices - Use of `strncpy` instead of `strcpy` - Bounds checking before array access - Validation of file operations - Proper use of `execvp` (prevents some injections)

12.4 Known Vulnerabilities (Educational)

1. Command Injection via Special Characters

```
# Potentially dangerous if not properly handled:
simplesh> command; rm -rf /  # Sequential execution
```

```
simplesh> command && malicious_cmd  # Conditional execution
simplesh> command `whoami`  # Command substitution
```

Mitigation: Current implementation doesn't support `;`, `&&`, backticks - safe by omission

2. Buffer Overflow in Input Handling

```c
// Vulnerable pattern (DO NOT USE):
char input[1024];
gets(input);  //  Dangerous!

// Safe pattern (USE THIS):
char input[1024];
fgets(input, sizeof(input), stdin);  //  Safe
```

3. Race Condition in File Creation

```
# TOCTOU vulnerability:
simplesh> ls > /tmp/output.txt
# Attacker could replace /tmp/output.txt with symlink
```

Mitigation: Not addressed in basic version (advanced topic)

4. Directory Traversal

```
simplesh> cd ../../../../etc
# Could navigate to restricted directories
```

Mitigation: Relies on OS file permissions

12.5 Security Testing Checklist

- ☐ **ST-1**: Test command injection attempts
- ☐ **ST-2**: Test buffer overflow with long inputs (>1024 chars)
- ☐ **ST-3**: Test special characters in filenames
- ☐ **ST-4**: Test path traversal in `cd` command
- ☐ **ST-5**: Test fork bomb scenarios (`:() { :|:& };:`)
- ☐ **ST-6**: Test file permission boundaries
- ☐ **ST-7**: Verify no sensitive information in error messages
- ☐ **ST-8**: Test signal handling (SIGINT, SIGCHLD)

12.6 Responsible Disclosure

If you discover a security vulnerability while studying this code: 1. Document it as a learning example 2. Understand the underlying cause 3. Research proper mitigation techniques 4. Do not exploit in production systems 5. Do not use for malicious purposes

---

13. Success Metrics

13.1 Functional Acceptance Criteria

Criterion	Target	Status
Shell displays prompt	100%	
Executes common Unix commands	>95%	
Handles command arguments	100%	
Output redirection works	100%	
Input redirection works	100%	
Command piping functions	>95%	
Background processes execute	100%	
Built-in commands work	100%	
Error messages are clear	>80%	
No memory leaks	Zero leaks	
No zombie processes	Zero zombies	
Multi-user support	100+ concurrent	

13.2 Quality Metrics

Metric	Target	Measurement Method	Status
Code compilation warnings	Zero	`gcc -Wall -Wextra`	
Unit test coverage	>80%	gcov/lcov	
Integration test pass rate	100%	Test script	
No segmentation faults	Zero on valid input	Testing	
Code documentation	>60%	Comments/code ratio	
Function complexity	<15 per function	Cyclomatic complexity	
Memory leaks	Zero	Valgrind	
File descriptor leaks	Zero	lsof monitoring	

13.3 Performance Benchmarks

Benchmark	Target	Actual	Status
Command execution overhead	<50ms	TBD	
Shell memory footprint	<10MB	TBD	
Pipe throughput (100MB file)	>100MB/s	TBD	
1000 sequential commands	<30s	TBD	
50 concurrent background jobs	No crash	TBD	

13.4 Educational Outcomes

Outcome	Assessment Method	Target
Understanding of fork/exec	Verbal explanation	>80% comprehension
Understanding of pipes	Code explanation	>80% comprehension
Understanding of I/O redirection	Diagram creation	>75% accuracy
Ability to extend code	Add new feature	>70% success
Code quality awareness	Peer review	>75% positive feedback

13.5 Deployment Readiness Checklist

Code Completeness: - [ ] All must-have features implemented - [ ] All built-in commands functional - [ ] Error handling comprehensive - [ ] Code follows style guidelines - [ ] No TODOs or placeholders in release

Testing: - [ ] All unit tests passing - [ ] All integration tests passing - [ ] Stress tests completed - [ ] Memory leak tests passed - [ ] Multi-user tests passed

Documentation: - [ ] README.md complete - [ ] USAGE.md with examples - [ ] ARCHITECTURE.md written - [ ] Code comments adequate - [ ] Build instructions clear

Build System: - [ ] Makefile functional - [ ] Clean build (no warnings) - [ ] `make test` works - [ ] `make clean` works - [ ] Installation script (optional)

Release: - [ ] Version tagged in git - [ ] Changelog updated - [ ] License file included - [ ] Demo video/screenshots (optional) - [ ] GitHub release created (optional)

---

14. Risks & Mitigation

14.1 Technical Risks

Risk	Probability	Impact	Mitigation Strategy	Contingency Plan
Complexity Underestimated	Medium	High	Incremental development, early prototyping, time-boxed phases	Reduce scope to must-haves only
File Descriptor Leaks	Medium	Medium	Systematic close() after use, lsof testing	Code review focused on FD management

Risk	Probability	Impact	Mitigation Strategy	Contingency Plan
Race Conditions in Signals	Low	Medium	Use sigaction over signal, careful handler design	Simplify signal handling, defer to Phase 2
Platform Incompatibility	Low	Medium	Test on multiple distros early	Document platform-specific issues
Memory Management Errors	High	High	Valgrind testing, defensive programming, code review	AddressSanitizer during development
Parsing Edge Cases	Medium	Medium	Comprehensive test cases, fuzz testing	Robust error handling, input validation
Pipe Deadlocks	Low	High	Proper pipe end closure, process flow review	Add timeout mechanisms

14.2 Project Management Risks

Risk	Probability	Impact	Mitigation	Contingency
Time Constraints	Medium	High	Focus on core features first, defer nice-to-haves	MVP with core features only
Skill Gap (System Calls)	Medium	Medium	Study materials, code examples, mentor support	Pair programming, office hours
Scope Creep	High	Medium	Strict feature prioritization (Must/Should/Nice)	Feature freeze after Phase 4

Risk	Probability	Impact	Mitigation	Contingency
Testing Time Underestimated	Medium	High	Allocate dedicated testing phase (Week 6)	Reduce test coverage target
Debugging Complexity	Medium	Medium	Use GDB, strace, systematic debugging	Simplify design if too complex

14.3 Educational Risks

Risk	Probability	Impact	Mitigation	Contingency
Difficulty Too High	Low	High	Start with simple phases, incremental complexity	Provide reference implementation
Insufficient Documentation	Medium	Medium	Document as you code, code review docs	Dedicated doc day before submission
Copy-Paste from Internet	Medium	Low	Require explanation of code, oral examination	Code walkthrough interviews
Late Submission	Low	High	Milestone deadlines, regular check-ins	Grace period with penalty

14.4 Risk Monitoring

Weekly Risk Assessment: - Review progress against timeline - Identify blockers and risks - Adjust mitigation strategies - Escalate critical issues

Risk Indicators: - Missed milestone deadline - Test failure rate >20% - Memory leaks in testing - Segfaults in basic operations - Unable to explain code functionality

---

15. Future Enhancements

Version 1.1: Usability Improvements

Features: - Command history (up/down arrow keys) - Tab completion for commands and files - Better prompt (show username, hostname, CWD) - Color-coded output (errors in red, etc.) - Improved error messages with suggestions

Estimated Effort: 2-3 weeks

Version 1.2: Advanced Features

Features: - Environment variable support (`$PATH`, `$HOME`, `$USER`) - Variable expansion (`$VAR` in commands) - Wildcard expansion (`*.txt`, `file?.c`) - Conditional execution (`&&`, `||`) - Command grouping (`(cmd1; cmd2)`) - Here documents (`<<`)

Estimated Effort: 3-4 weeks

Version 1.3: Job Control

Features: - Ctrl+Z to suspend processes (SIGTSTP) - `fg` command to foreground job - `bg` command to background job - `jobs` command to list jobs - Job numbering and management - Process groups and terminal control

Estimated Effort: 3-4 weeks

Version 2.0: Shell Scripting

Features: - Shell script execution (`./script.sh`) - Variables and assignment - If/then/else conditionals - For/while loops - Functions - Aliases - Source command (`. script.sh`)

Estimated Effort: 6-8 weeks

Version 2.1: Advanced Scripting

Features: - Arrays - String manipulation - Arithmetic expansion (`$(( ))`) - Regular expression matching - Advanced parameter expansion - Debugging mode (`set -x`)

Estimated Effort: 4-6 weeks

Version 3.0: Production Features

Features: - Security hardening - Performance optimizations - Plugin architecture - Configuration file support - Comprehensive logging - Internationalization (i18n)

Estimated Effort: 8-12 weeks

---

16. Appendices

Appendix A: Code Examples

A.1: Basic Shell Loop

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_INPUT 1024
#define MAX_ARGS 64

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
    pid_t pid;
    int status;

    while (1) {
        // Display prompt
        printf("simplesh> ");
        fflush(stdout);

        // Read input
        if (fgets(input, MAX_INPUT, stdin) == NULL) {
            break;  // EOF (Ctrl+D)
        }

        // Remove newline
        input[strcspn(input, "\n")] = '\0';

        // Check for empty input
        if (strlen(input) == 0) {
            continue;
        }
```

```
        // Check for exit command
        if (strcmp(input, "exit") == 0) {
            break;
        }

        // Parse command (simplified tokenization)
        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL && i < MAX_ARGS - 1) {
            args[++i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;  // Empty command after parsing
        }

        // Fork child process
        pid = fork();

        if (pid == 0) {
            // Child process
            if (execvp(args[0], args) == -1) {
                perror("exec");
                exit(1);
            }
        } else if (pid > 0) {
            // Parent process
            waitpid(pid, &status, 0);
        } else {
            // Fork failed
            perror("fork");
        }
    }

    printf("Goodbye!\n");
    return 0;
}
```

A.2: Output Redirection Implementation

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

void execute_with_output_redirect(char **args, char *output_file, int append) {
    pid_t pid = fork();
```

```
    if (pid == 0) {
        // Child process
        int flags = O_WRONLY | O_CREAT;
        flags |= append ? O_APPEND : O_TRUNC;
        int mode = 0644;  // rw-r--r--

        int fd = open(output_file, flags, mode);
        if (fd < 0) {
            perror("open");
            exit(1);
        }

        // Redirect stdout to file
        if (dup2(fd, STDOUT_FILENO) < 0) {
            perror("dup2");
            exit(1);
        }
        close(fd);

        // Execute command
        if (execvp(args[0], args) == -1) {
            perror("exec");
            exit(1);
        }
    } else if (pid > 0) {
        // Parent waits
        int status;
        waitpid(pid, &status, 0);
    } else {
        perror("fork");
    }
}
```

A.3: Simple Pipe Implementation

```
void execute_pipeline(char **cmd1, char **cmd2) {
    int pipefd[2];
    pid_t pid1, pid2;

    // Create pipe
    if (pipe(pipefd) < 0) {
        perror("pipe");
        return;
    }
```

```c
    // First command (writer)
    pid1 = fork();
    if (pid1 == 0) {
        // Child 1: Redirect stdout to pipe write end
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]);
        close(pipefd[1]);

        execvp(cmd1[0], cmd1);
        perror("exec cmd1");
        exit(1);
    }

    // Second command (reader)
    pid2 = fork();
    if (pid2 == 0) {
        // Child 2: Redirect stdin from pipe read end
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);
        close(pipefd[1]);

        execvp(cmd2[0], cmd2);
        perror("exec cmd2");
        exit(1);
    }

    // Parent: Close pipe and wait for children
    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
}
```

A.4: SIGCHLD Handler for Background Jobs

```c
#include <signal.h>
#include <sys/wait.h>

void sigchld_handler(int sig) {
    (void)sig;  // Unused parameter

    // Reap all terminated children (non-blocking)
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        // Child reaped
    }
}
```

```c
void setup_signal_handler() {
    struct sigaction sa;
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;

    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }
}
```

Appendix B: Makefile

```makefile
# SimpleSH Makefile

# Compiler and flags
CC = gcc
CFLAGS = -Wall -Wextra -Werror -std=c99 -pedantic
DEBUGFLAGS = -g -O0
RELEASEFLAGS = -O2
SANITIZEFLAGS = -fsanitize=address -fsanitize=undefined

# Directories
SRCDIR = src
INCDIR = include
OBJDIR = obj
TESTDIR = tests
BINDIR = .

# Target executable
TARGET = $(BINDIR)/simplesh

# Source files
SOURCES = $(wildcard $(SRCDIR)/*.c)
OBJECTS = $(SOURCES:$(SRCDIR)/%.c=$(OBJDIR)/%.o)

# Include path
INCLUDES = -I$(INCDIR)

# Default target
all: release
```

```
# Release build
release: CFLAGS += $(RELEASEFLAGS)
release: $(TARGET)

# Debug build
debug: CFLAGS += $(DEBUGFLAGS)
debug: $(TARGET)

# Sanitizer build (for testing)
sanitize: CFLAGS += $(DEBUGFLAGS) $(SANITIZEFLAGS)
sanitize: $(TARGET)

# Link object files to create executable
$(TARGET): $(OBJECTS) | $(BINDIR)
    $(CC) $(CFLAGS) $(INCLUDES) -o $@ $^
    @echo "Build complete: $(TARGET)"

# Compile source files to object files
$(OBJDIR)/%.o: $(SRCDIR)/%.c | $(OBJDIR)
    $(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

# Create directories if they don't exist
$(OBJDIR):
    mkdir -p $(OBJDIR)

$(BINDIR):
    mkdir -p $(BINDIR)

# Run tests
test: $(TARGET)
    @echo "Running integration tests..."
    @bash $(TESTDIR)/run_all_tests.sh

# Memory leak check
valgrind: debug
    valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./$(TARGET)

# Code coverage
coverage: CFLAGS += -fprofile-arcs -ftest-coverage
coverage: clean $(TARGET)
    @echo "Running tests for coverage..."
    @bash $(TESTDIR)/run_all_tests.sh
    @echo "Generating coverage report..."
    @gcov -r $(SRCDIR)/*.c
    @lcov --capture --directory . --output-file coverage.info
    @genhtml coverage.info --output-directory coverage_html
```

```
	@echo "Coverage report generated in coverage_html/"

# Clean build artifacts
clean:
	rm -rf $(OBJDIR) $(TARGET) *.gcov *.gcda *.gcno coverage.info coverage_html

# Clean test outputs
clean-tests:
	rm -f $(TESTDIR)/*.txt $(TESTDIR)/test_* /tmp/simplesh_test_*

# Full clean
distclean: clean clean-tests

# Install (optional)
install: release
	@echo "Installing simplesh to /usr/local/bin..."
	@sudo cp $(TARGET) /usr/local/bin/
	@sudo chmod 755 /usr/local/bin/simplesh
	@echo "Installation complete"

# Uninstall
uninstall:
	@echo "Removing simplesh from /usr/local/bin..."
	@sudo rm -f /usr/local/bin/simplesh
	@echo "Uninstallation complete"

# Help
help:
	@echo "SimpleSH Makefile Targets:"
	@echo "  make / make all      - Build release version"
	@echo "  make debug           - Build with debug symbols"
	@echo "  make sanitize        - Build with sanitizers (testing)"
	@echo "  make test            - Run integration tests"
	@echo "  make valgrind        - Run with memory leak detection"
	@echo "  make coverage        - Generate code coverage report"
	@echo "  make clean           - Remove build artifacts"
	@echo "  make clean-tests     - Remove test output files"
	@echo "  make distclean       - Full clean (build + tests)"
	@echo "  make install         - Install to /usr/local/bin"
	@echo "  make uninstall       - Remove from /usr/local/bin"
	@echo "  make help            - Show this help message"

.PHONY: all release debug sanitize test valgrind coverage clean clean-tests distclean instal
```

Appendix C: Example Test Script

```bash
#!/bin/bash
# run_all_tests.sh - Integration test runner

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

# Test counters
TOTAL_TESTS=0
PASSED_TESTS=0
FAILED_TESTS=0

# Shell executable
SHELL_BIN="./simplesh"

# Test output directory
TEST_DIR="/tmp/simplesh_tests"
mkdir -p "$TEST_DIR"

# Helper function to run a test
run_test() {
    local test_name="$1"
    local test_command="$2"
    local expected_output="$3"
    local test_type="${4:-exact}"  # exact, contains, file_exists

    TOTAL_TESTS=$((TOTAL_TESTS + 1))

    # Execute command
    local output_file="$TEST_DIR/${test_name}_output.txt"
    echo "$test_command" | timeout 5 $SHELL_BIN > "$output_file" 2>&1
    local exit_code=$?

    # Check result based on test type
    local test_passed=false

    case "$test_type" in
        "exact")
            if [ "$(<$output_file)" = "$expected_output" ]; then
                test_passed=true
            fi
            ;;
```

```
        "contains")
            if grep -q "$expected_output" "$output_file"; then
                test_passed=true
            fi
            ;;
        "file_exists")
            if [ -f "$expected_output" ]; then
                test_passed=true
            fi
            ;;
        "exit_code")
            if [ "$exit_code" -eq "$expected_output" ]; then
                test_passed=true
            fi
            ;;
    esac

    # Display result
    if [ "$test_passed" = true ]; then
        echo -e "${GREEN} ${NC} $test_name"
        PASSED_TESTS=$((PASSED_TESTS + 1))
    else
        echo -e "${RED} ${NC} $test_name"
        FAILED_TESTS=$((FAILED_TESTS + 1))
        echo "  Command: $test_command"
        echo "  Expected: $expected_output"
        echo "  Got: $(<$output_file)"
    fi
}

# Start tests
echo "=========================================="
echo "  SimpleSH Integration Test Suite"
echo "=========================================="
echo

# Test Category: Basic Commands
echo "--- Basic Commands ---"
run_test "TC-01 Simple Command" "ls\nexit" "" "exit_code" 0
run_test "TC-02 Command with Args" "echo 'Hello World'\nexit" "Hello World" "contains"
run_test "TC-03 Exit Command" "exit" "" "exit_code" 0
run_test "TC-04 Invalid Command" "invalidcommand123\nexit" "not found" "contains"

# Test Category: Built-in Commands
echo
echo "--- Built-in Commands ---"
```

```
run_test "TC-05 CD Command" "cd /tmp\npwd\nexit" "/tmp" "contains"
run_test "TC-06 CD Invalid Path" "cd /nonexistent\nexit" "No such file" "contains"

# Test Category: Output Redirection
echo
echo "--- Output Redirection ---"
run_test "TC-07 Output Redirect" "echo 'test' > $TEST_DIR/out1.txt\nexit" "" "file_exists" '
run_test "TC-08 Output Append" "echo 'line1' > $TEST_DIR/out2.txt\necho 'line2' >> $TEST_DIF

# Test Category: Input Redirection
echo
echo "--- Input Redirection ---"
echo "test input" > "$TEST_DIR/input.txt"
run_test "TC-09 Input Redirect" "cat < $TEST_DIR/input.txt\nexit" "test input" "contains"

# Test Category: Pipes
echo
echo "--- Pipes ---"
run_test "TC-10 Simple Pipe" "echo 'line1\nline2' | wc -l\nexit" "2" "contains"
run_test "TC-11 Three Command Pipe" "echo -e 'apple\nbanana\napple' | sort | uniq | wc -l\ne

# Test Category: Background Jobs
echo
echo "--- Background Jobs ---"
run_test "TC-12 Background Job" "sleep 1 &\necho 'immediate'\nexit" "immediate" "contains"

# Summary
echo
echo "=========================================="
echo "  Test Summary"
echo "=========================================="
echo -e "Total Tests:  $TOTAL_TESTS"
echo -e "${GREEN}Passed:${NC}       $PASSED_TESTS"
echo -e "${RED}Failed:${NC}       $FAILED_TESTS"
echo

# Cleanup
rm -rf "$TEST_DIR"

# Exit with appropriate code
if [ "$FAILED_TESTS" -eq 0 ]; then
    echo -e "${GREEN}All tests passed!${NC}"
    exit 0
else
    echo -e "${RED}Some tests failed.${NC}"
    exit 1
```

```
fi
```

Appendix D: Glossary

Term	Definition
ANSI C	American National Standards Institute C standard (C89/C90)
POSIX	Portable Operating System Interface - Unix standard
System Call	Interface between user programs and OS kernel
Fork	Create a child process (duplicate of parent)
Exec	Replace process image with new program
Wait	Parent waits for child process termination
Pipe	Unidirectional IPC mechanism (interprocess communication)
File Descriptor	Integer handle representing open file/pipe/socket
stdin	Standard input (file descriptor 0)
stdout	Standard output (file descriptor 1)
stderr	Standard error (file descriptor 2)
dup2	Duplicate file descriptor to specific FD number
Zombie Process	Terminated child not yet reaped by parent
Orphan Process	Child whose parent terminated
Signal	Asynchronous notification to process
SIGCHLD	Signal sent when child terminates
SIGINT	Signal sent by Ctrl+C
SIGTSTP	Signal sent by Ctrl+Z
Built-in Command	Command executed by shell itself (not external program)
External Command	Separate executable invoked by shell
Redirection	Changing default stdin/stdout/stderr
Pipeline	Chain of commands connected by pipes

Appendix E: References & Resources

Books

1. **"Advanced Programming in the UNIX Environment"** by W. Richard Stevens & Stephen A. Rago
    - Comprehensive coverage of Unix system calls
    - Chapter 8: Process Control (fork, exec, wait)
    - Chapter 15: Interprocess Communication (pipes)
2. **"The Linux Programming Interface"** by Michael Kerrisk

- Modern Linux system programming guide
- Chapters 24-27: Process creation and execution
- Chapter 44: Pipes and FIFOs

3. **"Operating Systems: Three Easy Pieces"** by Remzi H. Arpaci-Dusseau
   - Free online textbook
   - Process API chapter: http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

Online Resources

1. **Man Pages**
   - `man 2 fork` - Process creation
   - `man 3 exec` - Program execution
   - `man 2 wait` - Process synchronization
   - `man 2 pipe` - Pipe creation
   - `man 2 dup2` - File descriptor duplication
2. **Tutorials**
   - GNU C Library Manual: https://www.gnu.org/software/libc/manual/
   - Linux man-pages project: https://www.kernel.org/doc/man-pages/
   - GeeksforGeeks Unix System Programming: https://www.geeksforgeeks.org/
3. **Example Implementations**
   - GNU Bash source code: https://git.savannah.gnu.org/cgit/bash.git
   - Simple shell tutorials on GitHub

Standards

1. **POSIX.1-2008** (IEEE Std 1003.1-2008)
   - https://pubs.opengroup.org/onlinepubs/9699919799/
   - System Interfaces section
2. **C99 Standard** (ISO/IEC 9899:1999)
   - Standard C language reference

Development Tools

1. **GDB (GNU Debugger)**: https://www.gnu.org/software/gdb/
2. **Valgrind**: https://valgrind.org/
3. **AddressSanitizer**: https://github.com/google/sanitizers

---

Appendix F: Contact & Support

Project Maintainer: [Your Name]
Email: [your.email@example.com]
GitHub: [https://github.com/username/simplesh]

Getting Help: 1. Read documentation in `docs/` directory 2. Check examples in `examples/` directory 3. Search existing GitHub issues 4. Ask question in course discussion forum 5. Contact instructor/TA

Contributing: - See `CONTRIBUTING.md` for guidelines - Submit pull requests on GitHub - Report bugs via GitHub Issues

---

Appendix G: Revision History

Version	Date	Author	Changes
0.1	2026-02-01	Initial Draft	PRD framework created
0.5	2026-02-03	Technical Review	Added architecture details
1.0	2026-02-04	Final Review	Complete PRD approved

END OF DOCUMENT

Document Approval

Role	Name	Signature	Date
Product Owner	[Name]	________________	________
Technical Lead	[Name]	________________	________
Course Instructor	[Name]	________________	________
Student Lead	[Name]	________________	________

Confidentiality: This document is intended for educational purposes.

Validity: This PRD is valid for the duration of the project (6 weeks).

Revisions: Any changes to this document must be reviewed and approved by the project stakeholders.