Operation Systems Project
Electrical Computer and Engineering Dept.
Aristotle University of Thessaloniki
November 2025

TinyShell

The TinyShell is a small, Unix shell written in C for Linux, designed to mirror the basic behavior of a modern command interpreter while remaining lightweight and educational. The project evolves through multiple phases, each adding new system-level capabilities based directly on POSIX system calls.We are currently focused on completing the core requirements of Phase 1 of the TinyShell project.The Phase 1 establishes the fundamental structure necessary for any command-line interpreter and validates the core process control mechanisms upon which all future capabilities will be built.It consists of the core command loop, achieving the following functionalities:

1. Displays a prompt,
2. Parses the arguments,
3. Locates executables within the PATH,
4. Executes programs using `fork` and `execve`,
5. Reports the exit codes, and
6. Can run external programs such as `ls`, `cat`, or `echo`, terminating gracefully either with EOF or the `exit` command.

The detailed breakdown of the implemented features is described in the following sections.

Headers and Preproccessor Directives

This initial section defines the project's dependencies and crucial constants, ensuring the shell has access to necessary system calls and external libraries.

- **POSIX/System Headers:**
  - `#include <stdio.h>`, `#include <stdlib.h>`, `#include <string.h>`: Standard C library headers for basic input/output, general utilities, and string manipulation.
  - `#include <unistd.h>`: Essential for POSIX calls such as `fork()` and `access()` (used in Path Resolution).
  - `#include <sys/wait.h>`, `#include <sys/types.h>`: Vital headers for process management, specifically providing the `waitpid()` function and macros (`WIFEXITED`, `WEXITSTATUS`) required for reporting the child process status. The need for these headers necessitates the installation of the `build-essential` package on Linux environments.
- **External Library Headers (Readline):**

- `#include <readline/readline.h>`, `#include <readline/history.h>`: These include files integrate the GNU Readline library, which provides advanced terminal editing features (e.g., backspace, cursor movement) and command history management. Their usage mandates linking the library during compilation via the `-lreadline` flag, and installing the `libreadline-dev` package.

- **Preprocessor Directives (Constants):**
  - `#define MAX_ARG_CAPACITY 50`: Defines the initial capacity for the dynamically allocated arguments array, controlling the initial memory footprint and reducing the frequency of `realloc()` calls.
  - `#define ARG_DELIMITERS " \t\r\n\a"`: Defines the set of characters (`space`, `tab`, `carriage return`, `newline`, `alert`) used by `strtok()` to effectively tokenize the user's input string, ensuring robustness against varying whitespace usage.

Core Command Loop and User Input Handling

The foundation of the TinyShell is the Core Command Loop, which is implemented inside the shellMainLoop() function, that governs the entire lifecycle of the command interpreter. This loop runs continuously, maintaining the shell's state until explicitly terminated.The implementation of input handling transitions from basic C library functions (like getline) to the more robust GNU Readline library to enable advanced shell features and prevent potential read errors. The relevant code is split between the loop function, shellMainLoop(), and the dedicated input function, getInputLine(). The critical task of gathering user input is managed by the function getInputLine(), which leverages the GNU Readline library. This integration is essential as it moves beyond basic buffered I/O to provide robust line editing capabilities, including managing the backspace key and resolving input buffer errors that plague simpler implementations. Furthermore, Readline automatically displays the shell's prompt (tinyshell) and manages command history, storing previous commands for retrieval. After reading, the loop checks for the EOF (Ctrl+D) condition, which, along with the built-in exit command, ensures the shell can terminate correctly and return control to the host operating system.We will represent the step-by-step analysis of the shellMainLoop() code, focusing on the system-level mechanics and POSIX principles involved in each step.

- **Loop Initialization and Argument Acquisition:** The loop starts and attempts to read the user's command line.It initiates by `while (isRunning)`: The main loop keeps the shell active until the isRunning flag is explicitly set to 0 (or exit is called).Then loop continues by acquiring user input through inputLine = getInputLine(), which relies on the Readline library. This step is crucial as Readline handles sophisticated terminal I/O (like backspace and history), delivering a clean, fully processed string. This string is then immediately passed to commandArgs = splitInputIntoArguments(inputLine), which tokenizes the command line using delimiters to produce the null-terminated array of strings (char **) required by POSIX execution functions. Immediate checks follow to handle EOF (Ctrl+D) and empty lines, ensuring the shell terminates or ignores non-commands, respectively.

- **Command Identification and Built-ins:** The shell prioritizes checks for internal commands to avoid unnecessary process creation. The built-in `exit` command is handled by retrieving an optional exit code, performing essential **memory cleanup** (`free()` operations), and forcing process termination via `exit(exitCode)`.

- **Change Directory (cd):** The implementation of the `cd` command is fundamentally a demonstration of the core distinction between **built-in commands** and **external programs** in a POSIX shell. Unlike programs such as `ls` or `cat` (which are executed via `fork` and `execve` process model), `cd` must execute **internally** within the shell's own process. This necessity stems from the fact that its goal is to modify the shell's **Current Working Directory (CWD)**. Executing `cd` in a child process would only change the directory for that child, which immediately terminates, leaving the parent shell unaffected. Therefore, the implementation utilizes the direct POSIX system call `chdir(path)`, which modifies the CWD of the active process. The shell first manages logical handling, such as resolving the user's home directory (`getenv("HOME")`) when no path is provided. If `chdir` is successful, the execution flow is **diverted** by using the `continue` statement, which bypasses the standard `fork`/`execve` sequence and returns immediately to the prompt. This ensures that the shell's environment is updated instantly and the shell is ready to execute the next command from the new location.

- **Path Resolution and Validation:** For external commands, the shell calls `execPath = resolveCommandPath(...)`. This function encapsulates the **Path Resolution** logic, iterating through directories specified in the `PATH` environment variable and using the POSIX `access(path, X_OK)` call to verify the executable's existence and permissions before execution.

- **POSIX Process Management (`fork`, `execve`, `waitpid`):** This is the central process orchestration block, triggered only when an executable path is successfully resolved.The `fork()` call duplicates the shell process, creating an identical **child process** (`childPID == 0`).The child's sole purpose is to execute the target program. execve(execPath, commandArgs, environ) overlays the child's process space with the new program's code. If this fails (e.g., file permissions changed), the child must immediately call `exit(EXITFAILURE)` to avoid continuing the shell's logic loop. Simultaneously, the parent process enters a blocking `do-while` loop centered on `waitpid()`, guaranteeing the shell's operation remains synchronized with the external program's lifecycle .The shell remains responsive only after the program has completed its task (either by exiting normally or being terminated by a signal) and waits until the child process has terminated.

- **Exit Status Reporting and Cleanup:** Upon termination of the child process, the shell fulfills the crucial requirement of reporting the outcome. The status is checked using `WIFEXITED` (or `WIFSIGNALED` for signals), and the actual exit code is extracted via `WEXITSTATUS(childStatus)` and reported via `fprintf(stderr, "[Exit Status: %d]", ...)`. Finally, meticulous cleanup is performed using `free(inputLine)` (for Readline's allocation), `free(commandArgs)`, and `free(execPath)` to prevent memory leaks and maintain resource integrity.

Readline Integration: Advanced I/O with GNU Readline Library

- **Prompt Display and Input Reading:** The core operation is executed by `char *line = readline("TinyShell> ");`. The `readline()` function is highly efficient: it simultaneously displays the specified prompt string (`TinyShell>`) and manages robust terminal input. This single call automatically handles line editing (including backspace functionality), preventing the input buffering issues common with simpler `getline()` implementations.

- **EOF Handling:** The conditional check `if (line == NULL)` handles the End-Of-File (EOF) condition, typically triggered by the user pressing `Ctrl+D`. If `readline()` returns `NULL`, the shell interprets this as a signal for graceful termination and returns `NULL` to the main loop.
- **History Management:** The block `if (*line) { add_history(line); }` ensures that every non-empty command entered by the user is added to the shell's history stack. The `add_history()` function then allows the user to navigate and reuse previous commands via the Up and Down arrow keys.
- **Memory Allocation:** It is important to note that `readline()` dynamically allocates memory for the input string. This mandates that the calling function, `shellMainLoop()`, must include a corresponding `free(inputLine)` call after the command has been processed, ensuring memory integrity and preventing leaks across continuous loop iterations.
- **Function Return Value Analysis** The `getInputLine()` function is designed to return a value that dictates the next action of the `shellMainLoop()`:
  - **Successful Return (Non-NULL Pointer):** The function returns a valid `char*` pointer containing the user's command line string (including non-empty strings and strings consisting only of spaces). This indicates that a command was successfully read and should be processed by the shell.
  - **Termination Signal (NULL Pointer):** The function returns `NULL` under the specific condition where the user inputs the **End-Of-File (EOF)** signal (typically `Ctrl+D`). This return value is immediately checked in the main loop to set the `isRunning` flag to false, ensuring the shell terminates gracefully.

Argument Parsing

The `splitInputIntoArguments()` function is responsible for the critical task of transforming the raw user input line into a structured format suitable for the `execve()` system call. This process involves tokenization and dynamic memory management.

- **Initial Allocation and Error Handling:** The function begins by dynamically allocating memory for the array of argument pointers (`char** arguments`). It uses a predefined initial capacity (`MAX_ARG_CAPACITY`) and includes a critical check (`if (!arguments)`) to handle potential memory allocation failures, terminating the shell gracefully if the OS cannot provide the necessary resources.
- **Tokenization with `strtok()`:** The function leverages the C standard library function `strtok()` for tokenization.
  - The initial call (`currentArg = strtok(inputLine, ARG_DELIMITERS)`) breaks the input string into the first token (the command name), automatically **ignoring any leading delimiters** (whitespace, tabs, etc.).
  - Subsequent calls (`currentArg = strtok(NULL, ARG_DELIMITERS)`) continue the process, returning the remaining arguments until no more tokens are found.
- **Dynamic Reallocation (Resizing):** Within the `while` loop, a check is performed (`if (currentPosition >= capacity)`) to ensure the arguments array does not overflow. If the current capacity is exceeded, the function uses `realloc()` to safely expand the memory

space (`capacity += MAX_ARG_CAPACITY`). This makes the shell robust against commands with an arbitrary number of arguments.

- **Final Structure and Return:** After all tokens have been processed, the essential **NULL terminator** is placed at the end of the array (`arguments[currentPosition] = NULL;`). The execve() function strictly requires this null-terminated arrangement, and the final array of argument pointers is then returned to the main loop for execution.

Path Resolution

The `resolveCommandPath()` function is responsible for locating the full, executable path of a command, a crucial step that allows the user to execute programs without specifying their full directory path.

- **Direct Path Check (Bypass PATH):** Initial verification involves using `strchr()` to determine if the commandName includes a forward slash (`/`). If a slash is present, the function assumes the user provided a full or relative path, bypassing the extensive `PATH` search. It immediately checks for existence and executability using `access(commandName, X_OK)` and returns a duplicate of the path (`strdup()`) if successful, or `NULL` otherwise.
- **Retrieving and Tokenizing `PATH`:** If a direct path is not provided, the function retrieves the system's `PATH` environment variable using `getenv("PATH")`. It then creates a duplicate of this variable (`pathCopy = strdup(pathEnv)`) because `strtok()` modifies the string it operates on, which would otherwise corrupt the original environment variable. The `pathCopy` is tokenized using the colon (`:`) delimiter to isolate individual directories.
- **Iterative Directory Search:** The code enters a `while` loop to iterate through every `directory` found in the tokenized `PATH`.
  - **Path Construction:** Inside the loop, `snprintf()` is used to construct the full potential path (e.g., `/bin/ls`) by concatenating the directory, a forward slash, and the `commandName`.
  - **Access Verification:** The `access(fullPathBuffer, X_OK) == 0` check is performed. If the system confirms that the assembled path leads to an existing file with **execution permission** (`X_OK`), the path is duplicated (`strdup()`) into `foundPath` and the loop terminates (`break`).
- **Cleanup and Return:** Regardless of success or failure, the dynamically allocated memory for the `pathCopy` must be released via `free(pathCopy)`. The function then returns the `foundPath` (which contains the full executable path if found, or `NULL` if the command was not located in any directory listed in the `PATH`).

Main Function

The `main()` function serves as the entry point for the entire TinyShell program, adhering to the standard convention of C programs.

- **Shell Initialization:** The function's primary responsibility is to initialize and start the shell's core operation by executing the main loop: `shellMainLoop()`. This call transfers control to the continuous command interpreter cycle, where all user interaction and process management occur.

- **Program Termination:** The `shellMainLoop()` function is designed to run until a specific termination condition is met (either `exit` or `EOF`). When `shellMainLoop()` eventually returns control back to `main()` (only upon an `EOF` signal), the `main` function executes `return EXIT_SUCCESS`.
- **Exit Status (Host System):** The `return EXIT_SUCCESS` statement ensures that the TinyShell program itself exits with a status code of **0** to the host operating system (e.g., the WSL Bash environment), signifying that the shell process completed its task successfully.

Compilation and Dependency Management

This section outlines the necessary steps and external dependencies required to successfully compile the TinyShell project, integrating both standard POSIX system calls and the external Readline library.

- **External Library Dependencies (Installation):** Before compilation, the following development packages must be installed on the Linux environment (WSL):
  - `sudo apt install libreadline-dev`: Required for the `<readline/readline.h>` header files.
  - `sudo apt install build-essential`: Required for the core POSIX headers, including `<sys/wait.h>`, and the `gcc` compiler.
- **The Compilation Command:** The build process utilizes a provided Makefile within the file to correctly link the external readline library. The Makefile manages the dependency chain and ensures the compiler is executed with the correct command-line arguments, including the -lreadline linker flag.The file is compiled using the GNU C Compiler (`gcc`) with specific flags to manage warnings and link external libraries.To compile the program, navigate to the source directory and execute the following command: make.
  Otherwise the full command is:

```
gcc -Wall tinyshell.c -o tinyshell -lreadline
```

  - `-Wall`: Enables all useful compiler warnings, aiding in debugging and code quality.
  - `-o tinyshell`: Specifies the name of the output executable file.
  - `-lreadline`: **Linker Flag.** This critical flag instructs the linker to integrate the compiled code of the `readline` library into the final executable, resolving references for functions like `readline()` and `add_history()`.

Test Analysis and Visual Validation of TinyShell

The following analysis corresponds to the provided screenshot, which demonstrates the successful implementation of all core functionalities required for TinyShell Phase 1 (tinyshell display,process execution, Path Resolution, and exit status reporting).

Analysis of specific requirements

- `echo Phase 1 of TinyShell Complete!` / `ls -l` **(Exit Status: 0):**

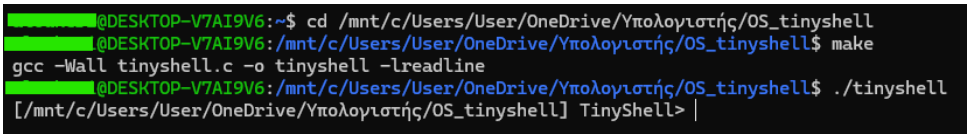

Figure 1: The TinyShell prompt (`TinyShell>`) displayed via the Readline library, ready for input.

  - **Functionality Verified:** Confirms the correct operation of the entire `fork-execve-waitpid` sequence. Both commands execute successfully and return the expected zero (`0`) exit status, proving that the shell can correctly launch external programs.

- `/bin/pwd` **(Exit Status: 0):**
  - **Functionality Verified:** Confirms that the shell correctly handles commands given via their **full absolute path**. The command executes, prints the path, and returns a successful status of `0`, bypassing the Path Resolution search.
- `true` **(Exit Status: 0)** and `false` **(Exit Status: 1):**
  - **Functionality Verified:** These tests are the definitive proof that the shell accurately reports the child process's termination status. `true` (which always exits successfully) reports `0`, while `false` (which always exits with an error) correctly reports `1`, confirming the proper use of the `WEXITSTATUS` macro.
- `invalid command` **(Error Handling):**
  - **Functionality Verified:** Demonstrates correct failure handling when the executable is not found. The output `myshell: Error: Command not found: invalid` verifies that the `resolveCommandPath` function returned `NULL`, and the shell successfully caught this error, preventing unnecessary process creation.
- `mkdir newdir` **(Exit Status: 0):**
  - **Functionality Verified:** This procedure assesses the fidelity of tokenization and argument transmission. The command `mkdir newdir` confirms that the `splitInputIntoArguments` function accurately segments the input line, ensuring the external binary receives the correct directory name (`newdir`) as a discrete argument in the child process's execution context.

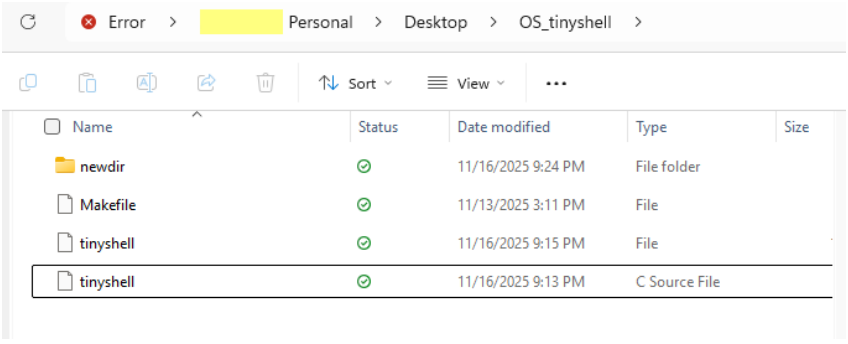

Figure 2: The directory newdir was created successfully.

- `cd newdir`
  - **Functionality Verified**:This core test verifies that the shell identifies `cd` via `strcmp` before attempting a `PATH` search. The invocation of `chdir()` must mod-

ify the shell's Current Working Directory (CWD), and this change must be instantly reflected in the prompt string constructed by the `getInputLine` function using `getcwd()`.

- `cd /`
  - **Functionality Verified**:The aim is to check the reliability of the `chdir()` call when presented with an absolute file system path (e.g., the root directory `/`). This test ensures the built-in function operates independently of the shell's current positional context.
- `exit` **(Graceful Termination):**
  - **Functionality Verified:** Confirms the correct behavior of the built-in `exit` command. The shell performs necessary cleanup and terminates, returning control to the host WSL/Bash shell.

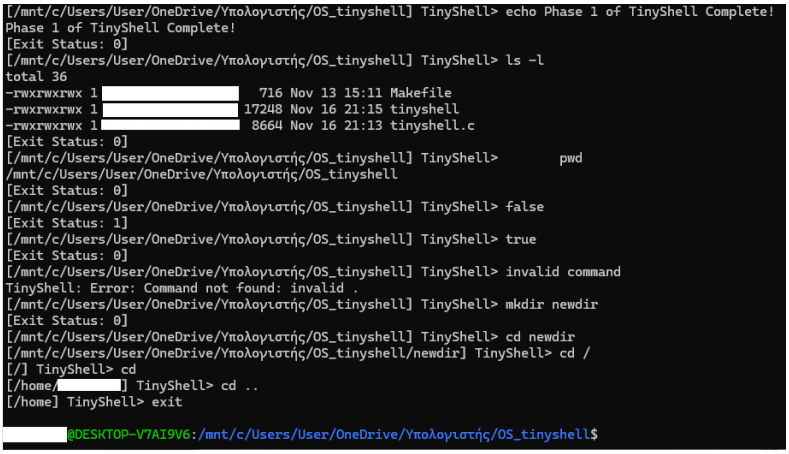

Figure 3: Composite validation of TinyShell Phase 1, demonstrating correct execution, Exit Status reporting, and error handling.

The successful outcome of all these tests confirms that TinyShell Phase 1 fully meets all specifications regarding process management, execution flow, and status reporting.

References

1. Modern Operating Systems - Andrew S.Tatenbaum
2. System Calls POSIX in C - Stack Overflow
3. POSIX Library Functions - Kompf.de
4. The GNU Readline Library - GNU.org