# Report: Design Choices and Documentation of Custom Shell Implementation

## Introduction

The custom shell implementation focuses on providing a user-friendly command-line interface (CLI) that mimics the functionality of popular shells like Bash. The shell supports built-in commands, external command execution, input/output redirection, process management, and piping, along with robust signal handling for enhanced user experience.  
This document outlines the design choices, key features, and implementation details, while providing insights into the code's functionality and usage.

## 1. Design Choices

### Command Parsing and Tokenization

- Design: The `tokenize\_command` function parses user input into an array of strings (tokens) based on whitespace delimiters.  
- Reasoning: This modular approach simplifies command processing and enables easy identification of commands, arguments, and special characters like `>`, `<`, and `|`.  
- Advantages: Handles varied user input formats and is scalable for additional features.

### Built-in Commands

- Design: Specific functions (`handle\_cd`, `handle\_pwd`, etc.) implement built-in commands directly in the shell.  
- Reasoning: Built-ins like `cd` and `setenv` require internal state changes that external processes cannot achieve.  
- Advantages: Streamlined execution and direct interaction with the shell environment.

### External Command Execution

- Design: Commands not implemented as built-ins are executed using `execvp` in child processes.  
- Reasoning: Allows execution of system utilities and user-defined scripts seamlessly.  
- Advantages: Leverages existing system capabilities and supports complex workflows with minimal overhead.

### Redirection and Piping

- Design: The shell identifies redirection (`>`, `<`) and piping (`|`) symbols during token parsing and configures file descriptors or pipes accordingly.  
- Reasoning: Enables advanced shell functionality for file manipulation and inter-process communication.  
- Advantages: Facilitates chained commands and adheres to standard shell conventions.

### Signal Handling

- Design: Custom signal handlers for `SIGINT` and `SIGALRM` manage process interruptions and timeouts.  
- Reasoning: Enhances robustness by preventing unwanted terminations and managing long-running processes.  
- Advantages: Graceful process management and improved user control during shell execution.

### Background Process Support

- Design: Commands ending with `&` are executed in the background, and the shell immediately returns the prompt.  
- Reasoning: Mimics behavior in traditional shells, allowing multitasking.  
- Advantages: Improves usability for concurrent tasks and ensures responsiveness of the shell.

## 2. Implementation Details

### Tokenization

The `tokenize\_command` function splits input into tokens and allocates memory dynamically. Each token represents a command, argument, or symbol like `>`.

### Built-in Commands

Commands such as `cd`, `pwd`, and `setenv` interact directly with system APIs. These functions are prioritized over external command execution for efficiency.

### Signal Handlers

`sigint\_handler` intercepts `Ctrl+C` signals and ensures the foreground process receives the signal, maintaining shell state. `sigalrm\_handler` sets a timeout for foreground processes, improving resource management.

### Redirection and Piping

Output redirection (`>`) is handled by duplicating file descriptors to `STDOUT`. Input redirection (`<`) duplicates descriptors to `STDIN`. Piping creates a pipeline between two commands using `pipe` and `dup2`.

### Process Management

`launch\_process` forks child processes for external commands. Foreground and background processes are managed through `waitpid` and `alarm`.

### Memory Management

Dynamic memory allocation is used for tokens, ensuring scalability. Allocated memory is freed after each command execution to avoid leaks.

## 3. Key Features

### Built-in Command Set

- `cd <dir>`: Changes the working directory.  
- `pwd`: Prints the current working directory.  
- `echo <args>`: Displays arguments or environment variables.  
- `env`: Lists all environment variables.  
- `setenv <var> <value>`: Sets environment variables.  
- `exit`: Exits the shell.

### Redirection and Piping

Supports output redirection (`>`), input redirection (`<`), and piping (`|`) between commands.

### Signal Handling

Intercepts `Ctrl+C` to handle process interruptions gracefully. Automatically terminates processes running longer than 10 seconds.

### Background Execution

Commands ending with `&` run in the background, displaying their process ID.

## 4. Usage

### Compilation

```bash  
gcc -o shell shell.c  
```

### Running the Shell

```bash  
./shell  
```

### Example Commands

Example usage of built-in and external commands with redirection and piping.