Custom Shell Implementation Report

I. Design Overview and Process Management

This report documents the design and implementation choices made for the custom shell program (`shell.c`). The primary goal was to create a functional shell capable of executing both built-in commands and external programs, while correctly managing standard process features like backgrounding, piping, and signal handling (Ctrl-C and timers).

1. Command Processing Pipeline

The shell's execution flow follows a standard process management pipeline:

1. **Input and Tokenization:** The main loop reads user input using `fgets`. The `tokenize` function splits the input by whitespace and performs **Environment Variable Expansion** (e.g., replacing `$HOME` with its value) and detects the **Background Operator (`&`)**.
2. **Built-in Check:** The `execute_builtin` function checks if the command is one of the six built-in commands (`cd`, `pwd`, `echo`, `exit`, `env`, `setenv`). Built-in commands are executed **directly in the parent shell process** to immediately affect the shell's state (e.g., changing the Current Working Directory via `cd`).
3. **Piping Check:** If the command is not built-in, the shell checks for the pipe operator (`|`).
    - **Piped Commands:** Handled by the **`execute_pipe`** function, which forks two child processes and uses the **`pipe()`** system call to connect the standard output of the first child (the "writer") to the standard input of the second child (the "reader"). The parent waits for both children to terminate.
4. **External Execution:** Simple commands are handled by the **`execute_single_command`** function. This involves a single **`fork()`** followed by **`execvp()`** in the child process to load and run the external program.

2. Process Forking and Execution

The core of running external programs is the `fork`/`execvp` pair:

- **`fork()`:** Creates a complete copy of the shell process (the child).
- **Child Process:** The child calls **`execvp()`** to replace its own image with the new program. `execvp` searches the system's `$PATH` for the executable, simplifying command execution.
- **Parent Process:** The parent uses **`waitpid()`** to manage the child's lifecycle, handling foreground/background distinction.

3. Background Process Management (Non-Blocking Wait)

Background jobs are defined by the `&` operator. The design ensures the shell remains responsive by **avoiding blocking waits** for background jobs:

- **Detection and Removal:** The `tokenize` and main loop logic identifies and removes the trailing `&`, setting the `is_background` flag.
- **Parent Action:** If `is_background` is true, the parent does **not** call `waitpid(pid, ..., 0)`. Instead, it calls `waitpid(-1, NULL, WNOHANG)` in a loop immediately after launching the job.
- **`WNOHANG` Flag:** This flag tells the kernel to return immediately if no child has terminated, preventing the shell from blocking, and allowing it to harvest (reap) any recently finished background zombie processes.

II. Signal Handling and Timer Implementation

Correct handling of system signals is critical for a robust shell experience, particularly for Ctrl-C and the mandated timeout feature.

1. `SIGINT` (Ctrl-C) Handling

The shell adheres to standard POSIX shell behavior for `SIGINT`:

- **Parent Shell:** `signal(SIGINT, SIG_IGN);` is set at startup. The shell **ignores** Ctrl-C, preventing the shell itself from terminating and ensuring it returns to the prompt after an operation is interrupted.
- **Foreground Child:** Before `execvp`, the child calls `signal(SIGINT, SIG_DFL);`. The signal handler is restored to its **default behavior**, meaning Ctrl-C will kill the foreground job, but not the parent shell.

2. Foreground Process Timer (`SIGALRM`)

The shell implements a 10-second timeout for **foreground single commands** using the `alarm()` and `kill()` system calls.

- **Global PID (`foreground_pid`):** A global variable stores the PID of the running foreground child. This is necessary because the `SIGALRM` signal handler is asynchronous and cannot access local variables.
- **Setting the Timer:** In `execute_single_command`, the parent process sets the timer: `alarm(10);`.

- **`alarm_handler` Logic:**
    - When the timer expires, the kernel sends **`SIGALRM`** to the parent shell.
    - The **`alarm_handler`** function executes, finds the PID in the global variable, and calls **`kill(foreground_pid, SIGKILL)`** to forcefully terminate the timed-out child process.
- **Cleanup:** Immediately after `waitpid()` returns (whether the job completed or was terminated by a signal), the parent calls **`alarm(0)`** to disable the timer, preventing a premature alarm during the next command loop.

3. `waitpid()` Interruption Handling

The `waitpid()` call in the parent must be robust against interruption by signals (`SIGINT` or `SIGALRM`). The code specifically checks:

```
if (waitpid(pid, &status, 0) == -1) {
   if (errno == EINTR) {
      // Signal caught (SIGALRM or SIGINT), check for termination
      waitpid(pid, &status, WNOHANG);
   }
}
```

If `waitpid` is interrupted (`errno == EINTR`), the shell executes a non-blocking `waitpid(..., WNOHANG)` to ensure the child process is reaped (cleaned up) immediately if it was killed by the `alarm_handler`.
While I cannot directly create and manage files in your GitHub repository, I can provide the complete content for the **two-page minimum report** that you should use for your GitHub README.md file. This report describes the design choices, process management strategies, and code documentation for your shell implementation.

You should copy the content below and paste it into a file named README.md in the root of your project directory.

Custom Shell Implementation Report

I. Design Overview and Process Management

This report documents the design and implementation choices made for the custom shell program (shell.c). The primary goal was to create a functional shell capable of executing both built-in commands and external programs, while correctly managing standard process features like backgrounding, piping, and signal handling (Ctrl-C and timers).

1. Command Processing Pipeline

The shell's execution flow follows a standard process management pipeline:

1. **Input and Tokenization:** The main loop reads user input using fgets. The tokenize function splits the input by whitespace and performs **Environment Variable Expansion** (e.g., replacing $HOME with its value) and detects the **Background Operator (&)**.
2. **Built-in Check:** The execute_builtin function checks if the command is one of the six built-in commands (cd, pwd, echo, exit, env, setenv). Built-in commands are executed **directly in the parent shell process** to immediately affect the shell's state (e.g., changing the Current Working Directory via cd).
3. **Piping Check:** If the command is not built-in, the shell checks for the pipe operator (|).
    - **Piped Commands:** Handled by the **execute_pipe** function, which forks two child processes and uses the **pipe()** system call to connect the standard output of the first child (the "writer") to the standard input of the second child (the "reader"). The parent waits for both children to terminate.
4. **External Execution:** Simple commands are handled by the **execute_single_command** function. This involves a single **fork()** followed by **execvp()** in the child process to load and run the external program.

2. Process Forking and Execution

The core of running external programs is the fork/execvp pair:

- **fork():** Creates a complete copy of the shell process (the child).
- **Child Process:** The child calls **execvp()** to replace its own image with the new program. execvp searches the system's $PATH for the executable, simplifying command execution.
- **Parent Process:** The parent uses **waitpid()** to manage the child's lifecycle, handling foreground/background distinction.

3. Background Process Management (Non-Blocking Wait)

Background jobs are defined by the & operator. The design ensures the shell remains responsive by **avoiding blocking waits** for background jobs:

- **Detection and Removal:** The tokenize and main loop logic identifies and removes the trailing &, setting the is_background flag.
- **Parent Action:** If is_background is true, the parent does **not** call waitpid(pid, ..., 0). Instead, it calls waitpid(-1, NULL, WNOHANG) in a loop immediately after launching the job.
- **WNOHANG Flag:** This flag tells the kernel to return immediately if no child has terminated, preventing the shell from blocking, and allowing it to harvest (reap) any recently finished background zombie processes.

II. Signal Handling and Timer Implementation

Correct handling of system signals is critical for a robust shell experience, particularly for Ctrl-C and the mandated timeout feature.

1. SIGINT (Ctrl-C) Handling

The shell adheres to standard POSIX shell behavior for SIGINT:

- **Parent Shell:** signal(SIGINT, SIG_IGN); is set at startup. The shell **ignores** Ctrl-C, preventing the shell itself from terminating and ensuring it returns to the prompt after an operation is interrupted.
- **Foreground Child:** Before execvp, the child calls signal(SIGINT, SIG_DFL);. The signal handler is restored to its **default behavior**, meaning Ctrl-C will kill the foreground job, but not the parent shell.

2. Foreground Process Timer (SIGALRM)

The shell implements a 10-second timeout for **foreground single commands** using the alarm() and kill() system calls.

- **Global PID (foreground_pid):** A global variable stores the PID of the running foreground child. This is necessary because the SIGALRM signal handler is asynchronous and cannot access local variables.
- **Setting the Timer:** In execute_single_command, the parent process sets the timer: alarm(10);.
- **alarm_handler Logic:**
    - When the timer expires, the kernel sends **SIGALRM** to the parent shell.
    - The **alarm_handler** function executes, finds the PID in the global variable, and calls **kill(foreground_pid, SIGKILL)** to forcefully terminate the timed-out child process.
- **Cleanup:** Immediately after waitpid() returns (whether the job completed or was terminated by a signal), the parent calls **alarm(0)** to disable the timer, preventing a premature alarm during the next command loop.

3. waitpid() Interruption Handling

The waitpid() call in the parent must be robust against interruption by signals (SIGINT or SIGALRM). The code specifically checks:

```
C
if (waitpid(pid, &status, 0) == -1) {
   if (errno == EINTR) {
      // Signal caught (SIGALRM or SIGINT), check for termination
      waitpid(pid, &status, WNOHANG);
   }
}
```

If waitpid is interrupted (errno == EINTR), the shell executes a non-blocking waitpid(..., WNOHANG) to ensure the child process is reaped (cleaned up) immediately if it was killed by the alarm_handler.

III. Code Documentation

Key functions and their parameters are documented below.

Function	Purpose	Arguments/Returns
tokenize	Parses the command line string, splits it into tokens, and performs environment variable expansion ($VAR).	char *command_line (Input string, modified by strtok). char *arguments[] (Output array of tokens). Returns int (Count of tokens).
print_prompt	Prints the shell prompt in the format /path/to/dir>.	N/A
execute_builtin	Checks if the command is cd, pwd, echo, exit, env, or setenv. Executes the command directly in the parent process if it's a built-in.	char *arguments[], int arg_count. Returns int (1 if built-in, 0 otherwise).
alarm_handler	Signal handler for SIGALRM. Kills the process stored in the global foreground_pid using SIGKILL.	int signum (The signal number, always SIGALRM).
find_pipe	Searches the token array for the `	` character.

execute_pipe	Forks two child processes, creates a pipe (pipefd), and uses dup2 to connect the output of the first command to the input of the second. The parent waits for both.	char *cmd1_args[], char *cmd2_args[], bool is_background (Ignored for simplicity).
execute_single_command	Forks a single child process to execute an external command. Handles setting the alarm, waitpid with WNOHANG (for background), and full blocking wait (for foreground).	char *arguments[], int arg_count, bool is_background.
main	Sets up signal handlers (SIGINT, SIGALRM, SIGCHLD), manages the read-tokenize-execute loop, and orchestrates the command routing.	