Madeline Bumpus @03007994

Create Your Own Shell Project Report

Description:

The command shell is implemented as a simple interactive program that takes user commands and executes them. The shell uses a loop to repeatedly prompt the user for input, tokenize the input, and execute the appropriate actions based on the provided commands.

Commands supported:

- cd: changes current directory
- pwd: prints the current working directory
- echo: prints a message and the values of environment variables
- exit: terminates the shell
- env: prints the current values of the environment variables
- setenv: sets an environment variable

Task 1: Print the Shell Prompt and Adding Built in Commands

The command line input is tokenized using a simple tokenization function that splits the input into individual arguments based on delimiters such as spaces, tabs, carriage returns, and newlines. This tokenization approach allows for flexibility in accepting various command formats.

In the main() function, the current working directory is obtained using the getcwd() function. Then the shell prompt is printed with the cwd.

```
while (true) {
    // Print the shell prompt with the current working directory
    char cwd[MAX_COMMAND_LINE_LEN];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("%s> ", cwd);
    } else {
        perror("getcwd");
        exit(EXIT_FAILURE);
    }
```

Then, we use fgets() to read the user input from the command line. The user input is split up into an array of tokens so we can access the command, its parameters, and flags separately. The first token represents the command, so we use if statements to check which command is being called. Depending on the command, we check for the required parameters and if it has any flags. An error is passed if the command doesn't match any of the ones we've implemented or the parameter is missing.

Task 2: Adding Processes

The execute_command function is where the process forking occurs. If the command is not a built-in command, a child process is forked using fork(), and the command is executed in the child process using execvp.

Task 3: Adding Background Processes

Before executing a command, the shell checks if the command ends with '&', indicating a request for running the process in the background. If '&' is found at the end, the background flag is set to true, and the ;&' is removed from the arguments.

Task 4: Signal Handling

A signal handler function (signal_handler) is defined to handle the SIGINT signal (Ctrl+C). When this signal is received, the handler prints a newline to the console to maintain the prompt format.

Task 5: Killing off Long Running Processes

A function (kill_process) is defined to terminate a process if it exceeds the allotted time limit (10 seconds). The kill() function is used to send the SIGTERM signal to terminate the process. In the main() function, the signal function is used to register the signal_handler function to handle the SIGINT signal. This ensures that whenever the SIGINT signal is received (Ctrl+C), the signal_handler function is called.

Task 6: I/O Redirection and Piping

Redirection

The shell checks for < (input redirection) and > (output redirection) symbols in the command. If found, it redirects input or output accordingly using dup2() to change the standard input or output file descriptors.

Piping

The shell detects the | symbol in a command and splits the command into segments, which are then executed individually. The standard output of one segment is connected to the standard input of the next.