OS ASSIGNMENT-2

GROUP-32

ARYAN VASHISHTHA & SAMAYRA MEENA

OVERVIEW:

The SimpleShell is a custom Unix shell in C that enables users to execute system commands, manage piping, and track the history of commands executed. The shell also stores details such as the Process ID (PID), the start time of execution, and the total duration for each command.

The program uses Unix system calls like fork(), execvp(), wait(), waitpid(), pipe() to execute commands, manage processes and implement pipes. The program also supports the command history and the background processes using “&” symbol.

The shell captures the `SIGINT` (Ctrl+C) signal using `signal()` and gracefully terminates by printing the command history before exiting.

CODE WALKTHROUGH:

GLOBAL ARRAYS:

history: Stores all the commands which have been entered by the user to the shell.
pidArray: Stores the Process ID (PID) of the respective commands entered by the user.
startArray: Stores the start time of the execution of the respective commands entered by the user.
endArray: Stores the total duration of the execution of the respective commands entered by the user.

read_userInput():

This function reads the user input & passes it as the command.

commandingArgs():
This parses the user-entered command to the space-separated args using **strtok()** function.

addStartTime():
This function adds the given time argument to the startArray of start time.

addEndTime():
This function adds the total execution time argument to the endArray of execution time.

addPid():
This function adds the PID of the command process to the pidArray of the PIDs

addHistory():
addHistory function adds the command to the history list of commands.

printHistory():
This function prints the history of all the commands entered by the user till now.
It specifically prints the following details about the command:

- The Serial Number
- The PID of the process
- The user input entered as the command.
- The start time of the process.
- The total execution time of the process.

launch():
the launch function takes the command and creates child processes out of parent processes using fork() and asks the parent to wait till the completion of the child process.
This function also implements the background processes using the “&” symbol.

ifPipe():
This function checks whether the pipe needs to be used to execute the given command.

ifPipe_Execute():
This function implements the pipe to execute the given command and add and write to the files using less time than the without-pipe method.

sys_call():
This function handles the system call SIGINT & prints the history of commands entered by the user.

loop_shell():
It calls the do-while loop, which prints “pvtshell:~$ ” and asks for user input. Then we get into the read_userInput() function to read the user input as the command. We add the command to the history list and check if the command needs to implement pipe. If yes, then it goes through ifPipe_Execute() otherwise, it goes through launch() after being parsed into args by the function commandingArgs(). It finally frees all the allocated memory and goes for the second iteration of the loop.
The program terminates when the user gives the input “Ctrl+C”.

SUPPORTED COMMANDS:

- ls
- wc -l
- wc -c
- echo
- grep
- ls -R
- ls -l
- sort
- uniq
- cat
- mkdir
- rm
- cp
- mv
- head

- tail
- gzip
- gunzip
- which ls
- whoami

UNSUPPORTED COMMANDS:

cd: The cd command is a shell built-in and not an external program, implying that it directly affects the current process's working directory. Here when cd is passed, it will execute only in the child process, changing its directory because the new processes are created using fork() and execvp(). The Shell remains unaffected, which means that the working directory would not persist after the child process terminates.

exit: The Shell runs in an infinite loop unless terminated by Ctrl+c and not exit because exit terminates only the current shell process, i.e., the ongoing child process. The main shell loop will continue running.

sudo: The sudo command is an external command and can run with execvp(), but will behave differently because it can be processed only if the shell supports interactive user input for passwords and potentially accessing the system. It would involve handling permissions, user authentication, and system security checks, which extends the complexity of a Simple-Shell implementation.

CREDITS:

ARYAN- launch implementation & signal handling
SAMAYRA- pipe implementation & bonus part
Debugging, testing and documentation were mutual.

GITHUB LINK