Operating Systems (Comp Sci 3SH3), Fall 2024
Prof. Neerja Mhaskar

Assignment 2

Due by 11:59pm on October 6th, 2024

- **No late assignment accepted, unless an MSAF is provided**
- **If an MSAF is provided, then you will get 5 days extension on the assignment.**
- It is advisable to start your assignment early.
- Make sure to submit a version of your assignment ahead of time to avoid last minute uploading issues.
- Note that students/groups copying each other's solution will get a zero.
- The assignment should be submitted on Avenue under Assessments -> Assignments -> Assignment 2 -> [Group #] folder.
- In your C programs, you should follow good programming style, which includes providing instructive comments and well-indented code. If this is not followed, marks will be deducted.
- **If working in a group of two, a `Readme` file containing information on your individual contributions should be provided.**
- **Generative AI use is allowed only for**
  - **Input / output string processing**
  - **Storing and retrieving commands in the history**
  - **If you use generative AI for the above two tasks, document its usage in the readme file.**

[20 points] Question 1: UNIX Shell and History Feature

This question consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command:
`cat prog.c`. The UNIX/LINUX `cat` command displays the contents of the file `prog.c` on the terminal using the UNIX/LINUX cat command and your program needs to do the same.

```
osh> cat prog.c
```

The above can be achieved by running your shell interface as a parent process. Every time a command is entered, you create a child process by using `fork()`, which then executes the user's command using one of the system calls in the `exec()` family. A C program that provides the general operations of a command-line shell can be seen below.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

while (should_run) {
printf("osh>");
fflush(stdout);

/**
* After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
* (3) parent will invoke wait() unless command included &
*/
}
return 0;
}
```

Figure 3.32 Outline of simple shell.

The `main()` function presents the prompt `osh>` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

This question is organized into two parts:

[5 points] Creating the child process and executing the command in the child

Your shell interface needs to handle the following two cases.

1. **Parent waits while the child process executes.**

In this case, the parent process first reads what the user enters on the command line (in this case, `cat prog.c`), and then creates a separate child process that executes the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Lab 2.

2. **Parent executes in the background or concurrently while the child process executes (similar to UNIX/Linux)**

To distinguish this case from the first one, add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as `osh> cat prog.c &` the parent and child processes will run concurrently.

[15 points] Modifying the shell to allow a history feature

In this part your shell interface program should provide a ***history*** feature that allows the user to access the most recently entered commands **with arguments**. The user will be able to access up to 5 commands (with its arguments) by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 5. For example, if the user has entered 35 commands, the 5 most recent commands will be numbered 31 to 35. The user will be able to list the command history by entering the command

```
osh> history
```

As an example, assume that the history consists of the commands with arguments (from most to least recent): `ls -l, top, ps, who, date`. The command history should output:

```
5 ls -l
4 top
3 ps
2 who
1 date
```

Note:

- **The history command should print the last 5 commands in reverse order i.e. last command appears first.**
- **The `history` command itself should not be stored in the history.**

Your program should support the following technique for retrieving command from the command history: When the user enters **!!**, the most recent command (with its arguments) in the history is executed. In the example above, if the user entered the command:

```
osh> !!
```

The '`ls -l`' command should be executed and echoed on user's screen. The command should also be placed in the history buffer as the next command.

Error handling:

The program should also manage basic error handling. For example, if there are no commands in the history, entering !! should result in a message "No commands in history."

Deliverables:

1. **`shell.c`** - You are to provide your solution as a single C program named `shell.c` that contains your solution for this question. Please make sure to name your solution with the correct file name for grading purposes.