Project 2: Shell with History[1]

1. Overview

In this project, you will develop a simple UNIX shell. The shell accepts user commands and then executes each command in a separate process. The shell provides the user a prompt at which the next command is entered. One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line and then create a separate child process that performs the command.

Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background – or concurrently – as well by specifying the ampersand (&) at the end of the command. The separate child process is created using the `fork()` system call and the user's command is executed by using one of the system calls in the `exec()` family.

2. Simple Shell

A C program that provides the basic operation of a command line shell is given below. The `main()` function first calls `read_command()`, which reads a full command from the user and tokenizes it into separate words (arguments). These tokens can be passed directly to `execvp()` in the child process.

If the user enters an "&" as the final argument, `read_command()` will set the `in_background` parameter to true (and remove the "&" from the array of tokens). For example, if the user enters "ls –l" at the '>' prompt, `tokens[0]` will contain "ls", `tokens[1]` will contain (or point to) the string "-l", and tokens[2] will be a `NULL` pointer indicating the end of the arguments. (Each of these strings is a `NULL` terminated C-style string). Note that the character array `buff` will contain the text that the user entered; however, it will not be one single `NULL` terminated string but rather a bunch of `NULL` terminated strings, each of which is a token pointed to by the `tokens` array.

This project is organized into three parts: (1) creating the child process and executing the command in the child, (2) adding some internal commands, and (3) modifying the shell to allow a history feature.

[1]Created by Mohamed Hefeeda, modified by Brian Fraser.

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define COMMAND_LENGTH 1024
#define NUM_TOKENS (COMMAND_LENGTH / 2 + 1)

/**
 * Read a command from the keyboard into the buffer 'buff' and tokenize it
 * such that 'tokens[i]' points into 'buff' to the i'th token in the command.
 * buff: Buffer allocated by the calling code. Must be at least
 *      COMMAND_LENGTH bytes long.
 * tokens[]: Array of character pointers which point into 'buff'. Must be at
 *      least NUM_TOKENS long. Will strip out up to one final '&' token.
 *      'tokens' will be NULL terminated.
 * in_background: pointer to a boolean variable. Set to true if user entered
 *      an & as their last token; otherwise set to false.
 */
void read_command(char *buff, char *tokens[], _Bool *in_background)
{
        ... Full code available on course website (in shell.c)...
}

/**
 * Main and Execute Commands
 */
int main(int argc, char* argv[])
{
        char input_buffer[COMMAND_LENGTH];
        char *tokens[NUM_TOKENS];
        while (true) {

                // Get command
                // Use write because we need to use read()/write() to work with
                // signals, and they are incompatible with printf().
                write(STDOUT_FILENO, "> ", strlen("> "));
                _Bool in_background = false;
                read_command(input_buffer, tokens, &in_background);

                /**
                 * Steps For Basic Shell:
                 * 1. Fork a child process
                 * 2. Child process invokes execvp() using results in token array.
                 * 3. If in_background is false, parent waits for
                 *    child to finish. Otherwise, parent loops back to
                 *    read_command() again immediately.
                 */
        }
        return 0;
}
```

3. Creating Child Process

First, modify `main()` so that upon returning form `read_command()`, a child is forked and executes the command specified by the user. As noted above, `read_command()` loads the contents of the `tokens` array with the command specified by the user. This `tokens` array will be passed to the `execvp()` function, which has the following interface:

```
execvp(char *command, char * params[]);
```

where `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as

```
execvp(tokens[0], tokens);
```

Be sure to check the value of `in_background` to determine if the parent process is to wait for the child exit or not. Hint: use `waitpid()` vs `wait()` because you want to wait on the child you just launched. If you only use `wait()` and have previously launched any child processes in the background that have terminated, `wait()` will immediately return having "consumed" the previous zombie process, and your current process incorrectly acts as though it was run in the background. Note that we won't be testing with interactive command-line processes run in the background (think `vim`), or test using signals while running a command in the background.

If `execvp()` returns an error (see `man execvp`) then display an error message.

Note that using `printf()` may not work well for this assignment and that you should use `write()` instead (look up more with `man write`). The issue is that we need to use the `read()` function for getting the user's command and use `write()` when working with signals (later). And, it turns out that `printf()` and `read()`/`write()` don't always work well together.

Therefore, when printing to the screen, use the `write()` command. For common things, such as displaying a simple string, or writing a number to the screen, you may want to make your own functions which make it easier. (You can convert an integer to a string using `sprintf()`).

3.1 Waiting Aside

When a process in Linux finishes, it still exists in the kernel with some status information until the parent process waits on that child. These un-waited-on terminated child processes are known zombies. For this assignment, you won't lose any marks if you don't correctly wait() on zombie processes from background tasks; however, it's a good habit to correctly cleanup the zombies on your system!

Above it is suggested that you use `waitpid()` to wait on the correct child. However, this will leave any background process as zombie processes (having exited, the parent process will never `wait()` on the child). You can correct this by occasionally trying a non-blocking wait to handle any zombie child processes. We can pass the `WNOHANG` option to `waitpid()` to be non-blocking, and setting the PID to -1 will wait on any child process. For example, have the following code run after every user command is processed:

```
// Cleanup any previously exited background child processes
// (The zombies)
while (waitpid(-1, NULL, WNOHANG) > 0)
      ; // do nothing.
```

4. Internal Commands

Internal commands are commands that the shell itself implement, as opposed to a separate program that is executed. Implement the commands listed below. Note that for these you need not fork a new

process as they can be handled directly in the parent process.

- **`exit`**
  Exit the program. Does not matter how many arguments the user enters; they are all ignored.
- **`pwd`**
  Display the current working directory. Use `getcwd()` function. Run `man getcwd` for more. Ignore any extra arguments passed by the user.
- **`cd`**
  Change the current working directory. Use `chdir()` function. Pass `chdir()` the first argument the user enters (it will accept absolute and relative paths). Ignore any extra arguments passed by the user.
  - If `chdir()` returns an error, display an error message.
  - You do not need to support ~ for the home folder, or changing to the home folder with "`cd`".

If the user presses enter on an empty line, do nothing (much like a normal Linux terminal).

Plus change the prompt to include the current working directory. For example, if in the `/home/student` folder, the prompt should be:

`/home/student>`

5. Creating a History Feature

The next task is to modify your shell to provide a history feature that allows the user access up to the ten most recently entered commands. Start numbering the user's commands at 1 and increment for each command entered. These numbers will grow past 10. For example, if the user has entered 35 commands, then the most recent ten will be numbered 26 through 35. This history feature will be implementing using a few different techniques.

5.1 History Commands

First implement an internal command "history" which displays the ten most recent commands executed in the shell. If there are not yet ten commands entered, display all the commands entered so far (<10).

- Display the commands in ascending order (sorted by its command number).
- Display the command number on the left, and the command (with all its arguments) on the right.
  - Hint: Print a tab between the two outputs to have them line up easily.
  - If the command is run in the background using &, it must be added to the history with the &.
- A sample output of the history command is shown below:

```
/home/student> history
3     cd /proc
4     cat uptime
5     cd /usr
6     ls
7     man pthread_create
8     cd /home/student
9     ls
10    ls -la
11    echo Hello World from my shell!
12    history
/home/student>
```

Next, implement the ! commands which allows users to run commands directly from the history list:

- Command "!*n*" runs command number *n*, such as "`!11`" will re-run the eleventh command entered this session. In the above example, this will re-run the echo command.
  - If *n* is not a number, or an invalid value (not one of the previous ten command numbers) then display an error.
  - You may treat any command starting with ! as a history command. For example, if the user types "!hi", just display an error. (Note that `atoi("hi")` returns 0, which should naturally be an invalid command number, and hence may generate a reasonable error message without extra work).
- Command "!!" runs the previous command.
  - If there is no previous command, display an error message.
- When running a command using "!*n*" or "!!", display the command from history to the screen so the user can see what command they are actually running.
- Neither the "!!" nor the "!*n*" commands are to be added to the history list themselves, but rather the command being executed from the history must be added. Here is an example.

```
/home/brian> echo test
test
/home/brian> !!
echo test
test
/home/brian> history
6     ls
7     man pthread_create
8     cd /home/brian
9     ls
10    ls -la
11    echo Hello World from my shell!
12    history
13    echo test
14    echo test
15    history
/home/brian>
```

Suggestions

- Implement history as a global two dimensional array:

```
#define HISTORY_DEPTH 10

char history[HISTORY_DEPTH][COMMAND_LENGTH];
```

- Rather than have all your code directly access the history array, write some functions which encapsulate all access to this array. Suggested functions would include: add command to history, retrieve command (copy into buffer, likely), printing the last ten commands to the screen.

5.2 Signals

Change your shell program to display the list of history commands when the user presses ctrl-c (which is the `SIGINT` signal). See course website for a guide on using signals.

- In `main()`, register a custom signal handler for the `SIGINT` signal.

- Have the signal handler display the previous ten user commands (same as `history` command).
- Then re-display the command-prompt before returning.
- Note that when another child process is running, ctrl-c will likely cancel/kill that process. Therefore displaying the history with ctrl-c will only be tested when there are no child processes running.

Suggestions

- To implement this, you will also need change `read_command()` a little bit.
- The signal handler will do nothing but displaying the history commands and then display the prompt again. The signal will interrupt the `read()` system call and discard all data already read for this command.
- When `read()` fails, it returns -1. You can check why `read` fails: if it returns -1 and the environment variable `errno` equals `EINTR` it means that it was interrupted by a signal. If the return value is -1 and `errno` is any other value, it means `read` just failed and the program should exit.
  - To correctly check `read()`'s return value you can change the code that you now have:

```
if (length < 0){
      perror("Unable to read command. Terminating.\n");
      exit(-1);  /* terminate with error */
}
```

to the following:

```
if ( (length < 0) && (errno !=EINTR) ){
      perror("Unable to read command. Terminating.\n");
      exit(-1);  /* terminate with error */
}
```

6. Notes

You do not need to support either > or | from the terminal. These are features of the normal Linux terminal that we are not implementing.

Your code must not have any memory leaks or memory access errors. Your shell must free all memory before it exits. However, your child processes may exit without freeing all their memory (if exec() fails then the child process will have a copy of all the memory that the parent holds; freeing this memory would be unnecessarily time consuming for the OS). Therefore, you may ignore any Valgrind messages when a child process terminates to the effect of memory being held when program terminated.

7. Submission

Submit an archive (zip or tar.gz) on Canvas of your code and a make file. We will build your code using your makefile, and the run it using the command: `./shell`

You may use more than one .c/.h file in your solution if you like. If so, your `makefile` must correctly build your project.

Please remember that all submissions will automatically be compared for unexplainable similarities.