CMPT 300 Assignment 2: Shell Commands and Processes [1]

This assignment can be done individually or in a group (up to 3 people, please form your group in CourSys).

Total points: 100 + 15 bonus. There are three problems in this assignment that you need to solve for the mandatory 100 points, plus an (optional) extra of 15 points for extra functionality you may implement for your shell.

All code **must** run on a Linux machine. We will grade your code on a Linux machine. You should create a directory for this assignment, such as *~/cmpt300/a2/* and put all files related to this assignment in it.

Please follow the text below to understand the context, and attempt to finish the problems described later.

1. Overview

In this assignment, you will develop a simple Linux 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. A 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.

```
#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 in section 8 (Resources) of this page (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. Problems

There are three mandatory problems (100 points) that you will need to solve:

- Create the child process and executing the command in the child.
- Implement some internal commands.
- Implement a history feature.

There is also one optional problem for a 20-point bonus, but you must attempt the mandatory problems first, before attempting the bonus problem. Attempting the bonus problem without work done in the mandatory problems will not get you any points for the bonus problem.

Problem 1. Creating Child Process [30 points]

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()` instead of `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()`.

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.
```

Problem 2. Shell Prompt and Internal Commands [30 points]

First, make sure your shell prompt always shows the current working directory. For example, if in the `/home/cmpt300` folder, the prompt should be:

```
/home/cmpt300$
```

Next, let's implement some internal commands. Internal commands are built-in features of the shell itself, 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. All the commands here are **case-sensitive**.

- `exit`: Exit the shell program. If the user provided any argument, abort the operation (i.e., command not

executed) and display an error message.

- `pwd`: Display the current working directory. Use the `getcwd()` function. Run `man getcwd` for more. Again, abort the operation and display an error message if the user provided any argument.
- `cd`: Change the current working directory. Use the `chdir()` function. Pass `chdir()` the first argument the user enters (it will accept absolute and relative paths). If the user passed in more than one argument, abort the operation and display an error message. If `chdir()` returns an error, display an error message.
- `help`: Display help information on internal commands.
  - If the first argument is one of our internal commands, print "<argument> is a builtin command" plus a brief description on what the command does. For example, if argument is 'cd', the output should be:

    ```
    'cd' is a builtin command for changing the current working directory.
    ```

  - If the first argument is not an internal command, this command prints "<argument> is an external command or application". For example, if argument is 'ls', the output must be:

    ```
    'ls' is an external command or application
    ```

  - If there is more than one argument, display an error message
  - If there is no argument provided, list all the supported internal commands. For each command, include a short summary on what it does.

Problem 3. History Feature [40 points]

The next task is to modify your shell to provide a history feature that allows the user access up to the 10 most recently entered commands. Start numbering the user's commands at 0 and increment for each command entered. These numbers will grow past 9. For example, if the user has entered 35 commands, then the most recent 10 will be numbered 15 through 34.

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

- Display the commands in descending order (sorted by its command number).
- The output should include both external application commands and internal commands.
- 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/cmpt300$ history
30      history
29      cd /home/cmpt300
28      cd /proc
27      cat uptime
26      cd /usr
25      ls
24      man pthread_create
23      ls
22      echo Hello World from my shell!
21      ls -la
/home/cmpt300$
```

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

- Command "!n" runs command number n, such as "!22" will re-run the 23rd 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 10 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 not be treated as a valid command.
- 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/cmpt300$ echo test
test
/home/cmpt300$ !!
echo test
test
/home/cmpt300$ history
15      history
14      echo test
13      echo test
12      ls
11      man pthread_create
10      cd /home/cmpt300
9       ls
8       ls -la
7       echo Hello World from my shell!
6       history
/home/cmpt300$
```

Suggestions

- Implement history as a global two dimensional array:

```
#define HISTORY_DEPTH 10
char history[HISTORY_DEPTH][COMMAND_LENGTH];
```

- Rather than having all your code directly access the history array, write some functions which encapsulate accesses to this array. Suggested functions would include: add command to history, retrieve command (copy into buffer, likely), printing the last 10 commands to the screen.

Signals: Change your shell program to display the help information when the user presses ctrl-c (which is the `SIGINT` signal). A guide on using signals is available here.

- In `main()`, register a custom signal handler for the `SIGINT` signal.
- Have the signal handler display the help information (same as the `help` 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 help information with ctrl-c will only be tested when there are no child processes running.

Suggestions

- To implement this, you will also need to change `read_command()` a little bit.
- The signal handler will do nothing but displaying the help information 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 */
}
```

Problem 4 (Optional). Better `cd` Command [15 bonus points]

Note: You must finish Problems 1, 2 and 3 before attempting Problem 4. Submitting Problem 4 alone without solutions to Problems 1, 2 or 3 will leave you with 0 mark for Problem 4.

Implement the following features that are often supported by the `cd` command in modern shells (e.g., bash).

- Change to the home directory if no argument is provided. [4 points]
- Support ~ for the home folder. [6 points]
  For example, `cd ~/cmpt300` should change to the "cmpt300" directory under the current user's home directory. Issuing `cd ~` will switch to the home directory.
- Support - for changing back to the previous directory. [5 points]
  For example, suppose that the current working directory is `/home` and you issued `cd /` to change to the root directory. Then, `cd -` will switch back to the `/home` directory.

Hints: You may find the `getuid` and `getpwuid` functions useful. They allow you to gather useful information about the current user.

3. 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 (using Valgrind). 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.

4. Resources

- Sample code: shell.c, Makefile
- Guide to Signals
- Video Tutorial on Dynamic Memory Debugging using Valgrind

5. Submission

You need to submit a compressed archive named `a2.tar.gz` to CourSys of your code and a make file (sample code: shell.c, Makefile). If you did this assignment in a group, include also a simple readme text file in your package that details each group member's contributions; each of you should contribute to a reasonable share of the assignment, and every group member will receive the same marks. We will build your code using your make file, and 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.

Following the steps below to prepare your submission:

- Make sure that your files are stored in a directory as called `a2`
- Change to each of your folders and issue the command make clean. This will remove all object files as well as all output and temporary files
- Change to your a2 folder (assuming you put it under the cmpt300 folder):

  ```
  $ cd ~/cmpt300/a2
  ```

- Then, issue:

  ```
  $ tar cvf a2.tar *
  ```

  which creates a tar ball (i.e., a single file) that contains the contents of the folder.
- Compress your file using gzip:

  ```
  $ gzip a2.tar
  ```

- Submit via CourSys by the deadline posted there.

6. Grading Policies

Make sure you are familiar with the course policies. Especially, we do not accept late submissions, so please submit on time by the deadline. You can get partial marks for coding style if your code does not compile.

---

1. Created by Mohamed Hefeeda, modified by Brian Fraser, Keval Vora and Tianzheng Wang. top