Today we will continue to write our Linux Shell and implement functions `parse_command()`, `execute_command()`, `execute_cd()`, `execute_pwd()` and `execute_external()`.

1. Go back to your working copy of your shell code. This is the code you checked out of the repository. It should be at ~$/CPSC457/WorkingCopy/Shell

2. Implement `parse_command()`. This function should take the command that the user entered during the `prompt()` function (which should be stored in the `command` array) and break it up using tokens into separate parts. These parts should be stored in the `argument_list` array. For example: If the user entered `cd ~/CPSC457` the value stored in the `command` array will be "cd ~/CPSC457" while the values stored in the `argument_list` array will be `argument_list[0] = "cd"`, `argument_list[1] = "~/CPSC457"

```c
char parse_command(){
    char* argument;
    argument = strtok(command, " "); //This takes the first part of command (up until the first space) and puts it into the local variable argument.

    Note: To continue to break up command at the spaces you must use
    argument = strtok(NULL, " ");
    for further calls to the `strtok()` function.
    Note 2: Don't forget to make sure that the `argument_count` variable does not exceed `MAX_ARGUMENT_COUNT`.

    Overall the outline of the `parse_command()` function will look like this:

    char parse_command(){
        char* argument;
        argument = strtok(command, " ");

        while (argument != NULL) {
            //You must fill in the rest of the code here!!!
            argument = strtok(NULL, " ");
        }

        return argument_count;
    }
```
3. The function `execute_command()` is used to call the function corresponding to the argument the user entered. For Example: If the user enters the command `cd ~/CPSC457` then `execute_command()` will call the `execute_cd()` function.

You can use the `strcmp()` function to compare strings in the `argument_list` array to given strings to decide what function to execute. You must have options corresponding to user commands for the `execute_cd()`, `execute_pushd()`, `execute_popd()`, `execute_pwd` and `execute_external()` functions.

```c
void execute_command() {
    if (!strcmp(argument_list[0], "cd")) {
        execute_cd();
        //Fill in the rest of the code for the other functions here.
    } else if (!strcmp(argument_list[0], "exit")) {
        exit(0);
    } else {
        execute_external();
    }
}
```

4. For the `execute_cd()` function we need to get the directory name out of the `argument_list` array and store it in a local variable `dirname`. Then we use the function `chdir(dirname)` to change directories.

Note: `chdir(dirname)` will return a -1 if it has issues completing. Your code should check for this and handle it appropriately such as printing an error message. If changing the directory succeeds you should print a message saying it succeeded.

Note 2: Using the ~ shortcut (specifies your home directory) with this function will result in an error. Why? We can work around this by using absolute paths when specifying arguments for this function. Absolute paths are paths that start from the root directory ./i.e. /home/grads/selaing/CPSC457/WorkingCopy/Shell is an absolute path to my current working directory. ~/CPSC457/WorkingCopy/Shell is the relative path. (Make sure you use a path that makes sense for you. You can find out the absolute path to your current working directory by using the `pwd` command.

```c
void execute_cd() {
    char * dirname;
    //Fill in the rest of the code here
```
5. For the `execute_pwd()` function we need to get the current directory name and store it in a local variable `dirname` and then print a statement telling the user what the result is.

```c
void execute_pwd()
{
    //Fill in code here
    dirname = (char *) get_current_dir_name();
    //Fill in code here
}
```

6. For the `execute_external()` function we need to use `fork` to create a child process that will run our external command and then `execvp` to start the external command. After calling `fork` we need to check to make sure it succeeded. After calling `execvp` we check to ensure that succeeded too. If it did succeed we wait until the child finishes execution of the external command before returning. The code for the function is complete below.

```c
void execute_external()
{
    int child = fork();
    if (child < 0 ) {
        printf("Error executing %s: %s", argument_list[0], strerror(errno));
    } else if (child == 0) {
        if (execvp(argument_list[0], argument_list) == -1) {
            printf("Error executing %s: %s\n", argument_list[0], strerror(errno));
            exit(-1);
        }
    } else {
        int status;
        wait(&status);
    }
}
```

7. You should commit your code to the repository if you haven’t done so already.

Note: All code for the Linux Shell was written Daniel de Castro and used with permission.