Project 2: Unix Shell and History Feature

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface provides the user a prompt after which the next command is entered.

What's more, a history record should be added for convenience. The user can execute a certain instruction according to the history.

To put it clearly, this project is organized into two parts: (1) creating the child process and executing the command in the child and (2) modifying the shell to allow a history feature.

Getting Started

Since I have constituted the linux environment in previous project, this part is skimmed. In this project, *Ubuntu-16.04.1* for linux is used, and its kernel version is *4.4.0*.

Simple Shell

The program should be composed of two functions: `main()` and `setup()`. I will introduce two these two part separately.

1. The `setup()` function

The `setup()` function reads in the user's next command, and parses it into separate tokens that are used to fill the argument vector for the command to be executed.

When a command is typed in, the string is divided by whitespace and saved in a char array *agrs[]* for future use. Notice the special character "&", which means the command is to be run in the background, should be treated specially. The parameter *background* will be updated so that the `main()` function can act accordingly.

```
//seperating the command into the distinct tokens
for(i = 0; i < len; ++i){
    if(inputBuffer[i] == ' ' || inputBuffer[i] == '\t'){ //use whitespace as the delimiters
        if(start != -1){
            args[j++] = &inputBuffer[start];
            start = -1;
        }
        inputBuffer[i] = '\0';
    }
    else if(inputBuffer[i] == '\n'){
        if(start != -1){
```

```
            args[j++] = &inputBuffer[start];
        }
        args[j] = NULL;
        inputBuffer[i] = '\0';
    }
    else{
        if(inputBuffer[i] == '&'){
            *background = 1;
            inputBuffer[i] = '\0';
        }
        if(start == -1)
            start = i;
    }
}
```

Code 1: Input Segmentation

According to the requirement, if the user enters *Control+D*, the program will be terminated. We can check the length of the input to judge whether the input is *Control+D*. That's because when *Control+D* is entered, a terminator *EOF* will be sent, whose length is 0.

2. The `main()` function

The `main()` function presents the prompt COMMAND -> and then invokes `setup()` function, which waits for the user to enter a command. As noted above, the contents of the command entered by the user is loaded into the *args[]* array.

Here a child process is forked and executes the command by invoking the `execvp()` function. The value of background is ought to be checked to determine if the parent process if to wait for the child to exit or not. The code for this part is exhibited below.

```
if(args[0]!= NULL){
    pid_t pid;
    pid = fork();
    if(pid < 0){
        printf("fork failed\n");
        exit(1);
    }
    else if(pid == 0){
        execvp(args[0], args);
        background = 0;
        printf("ERROR COMMAND!\n");
        exit(0);
    }
    else{
        if(background == 0)
            waitpid(pid, NULL, 0);
    }
}
```

Code 2: Creating a Child Process

Creating a History Feature

Next, we will modify the program above so that it provides a *history* feature that allows the user access up to 10 most recently entered commands. The user can also enter *r-type* commands to run any of the previous 10 commands.

1. Adding Commands to History

When the user enters the commands, they must be saved in an array named *History[]*. The code is presented below. They should be added in `setup()` function.

```
strcpy(history[(start_index + NumOfCommand) % 10], inputBuffer);
if(NumOfCommand == 10){
    start_index = (start_index + 1 + 10) % 10;
}
else{
    ++NumOfCommand;
}
```

Code 3: Add Commands to History

2. Looking up the History

The user will be able to list history commands when he presses *Contorl+C*, which is the SIGINT signal. Signals may be handled by first setting certain fields in the C structure struct *sigaction* and then passing this structure to the `sigaciton()` function.

We can set up our own signal-handling function by setting the *sa_handler* field in the struct *sigaction* to the name of the function which will handle the signal and then invoking the `sigaction()` function, passing it the signal we are setting up a handler for, and a pointer to struct *sigaction*.

```
// catch the ctrl c signal
void handle_SIGINT(){
    write(STDOUT_FILENO, buffer, strlen(buffer));
    printf("The Command History:\n");
    int i;
    i = start_index;
    int j = 0;
    fflush(stdout);
    for(; i < NumOfCommand + start_index; ++i){
        printf("%d. %s", ++j, history[i % 10]);
    }
    return;
}
```

Code 4: Presenting History

```
//set up the signal handler
struct sigaction handler;
handler.sa_handler = handle_SIGINT;
sigaction(SIGINT, &handler, NULL);

strcpy(buffer, "\nCaught Control C\n");
```

Code 5: Signal-handling

3. Dealing with r Type Commands

After the user enters *Ctrl+C*, the signal handler will output a list of the most recent 10 commands. With this list, the user can run any of the previous 10 commands by entering $r\ x$, where x is the first letter of that command. The command r alone means running the most recent command.

This part is not hard to realize. We can simply scan from the end of the history array and match the commands beginning with letter x. This part should be added to `setup()` function.

```
if(inputBuffer[0] == 'r' && NumOfCommand > 0){
    if(inputBuffer[1] == '\n'){
        strcpy(inputBuffer, history[NumOfCommand - 1]);
    }
    else if(inputBuffer[1] == ' ' && inputBuffer[2] != '\0' && inputBuffer[3] == '\n'){
        for(i = NumOfCommand - 1; i >= 0; --i){
            if(inputBuffer[2] == history[i][0]){
                strcpy(inputBuffer, history[i]);
                flag = 1;
                break;
            }
        }
        //if there is no command match to the r x, then return and no action to the history
        if(!flag){
            args[0] = NULL;
            return;
        }
    }
    len = strlen(inputBuffer);
}
```

Code 6: Dealing with r-type commands

Till now, all the functions of the shell have been completed. The structure of the shell is clear as well. We will test the shell later on.

Running Result

We create a *C* program and compile it. After running the program, we can type in commands to check the result, thereby judging whether our shell is perfect or not.

Figure 1 shows the results of some commands. They are all executed correctly.

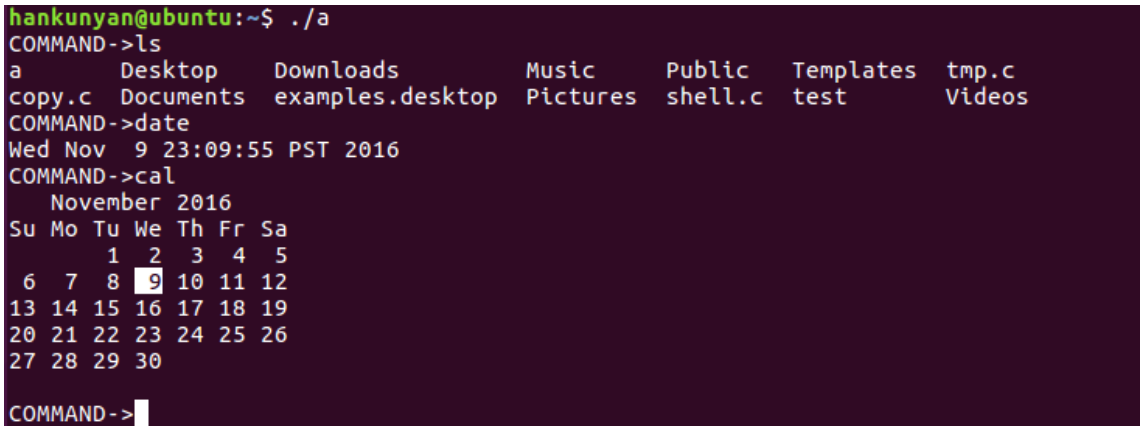

Figure 1: Some ordinary commands running on the shell

When *Ctrl+C* is entered, the command history will be displayed. Moreover, *r* commands can recall old commands. If there exists more than one commands begin with the same prefix, the recent one will be executed. The result is shown in Figure 2.

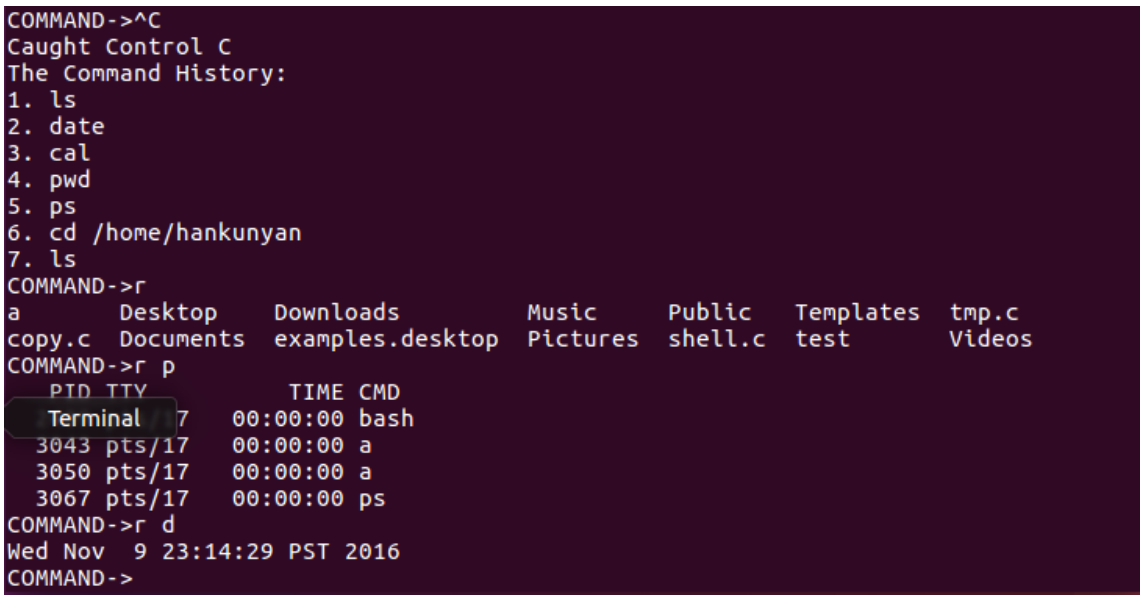

Figure 2: Command History and *r* commands

The special commands that end with '&' should be executed in the background. The results of running *ls* and *ls*& will show you the difference. When *ls* is executed, a child process will be forked, and its parent process has to wait until it finishes. However, after *ls*& is read, the parameter *background* will be set to 1, telling that it is executed background.

That's why COMMAND -> appears before the result of *ls*. The parent process needn't wait for the child process and print COMMAND -> immediately.

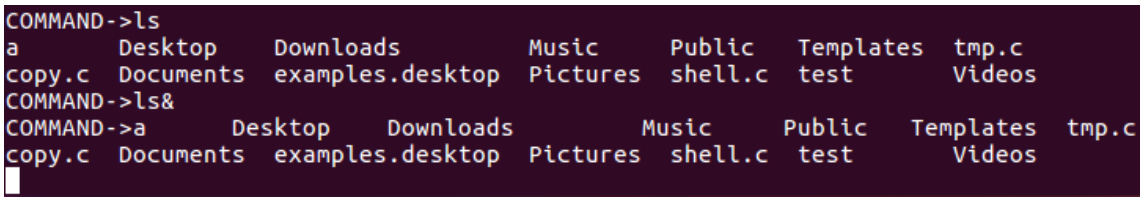

Figure 3: Running background

However, the result of child process won't be shown until it comes to an end.

For some commands that can create a new window, such as *gedit*, the shell will wait until the window is closed. If '&' is added, you can type in the next commands immediately.

Finally, you can enter *Ctrl+D* to quit the shell.

Figure 4: Quit the shell

Summary

I encountered various obstacles when coding the shell. The first I dealt with the *r* type commands, I considered them as a function merely belonging to the history and many problems occurred. Then I put them outside and it seemed more reasonable. By searching on the Internet, many problems got tackled and I really learned a lot.

To sum up, the project is quite meaningful and I have a clearer understanding about how a shell works. The full code is enclosed in the appendix.

Student name: Han Kunyan
Student No.:5140309534

Appendix

Full code of Shell.c

```
#include<stdio.h>
#include<errno.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/wait.h>

#define MAX_LINE 80
#define BUFFER_SIZE 50

char buffer[BUFFER_SIZE];
char history[10][BUFFER_SIZE]; //store the history command
int NumOfCommand = 0;   //the num of history command, the maximum value is 10
int start_index = 0;

//deal with the input and transfer the command to execute
void setup(char inputBuffer[], char *args[], int *background){
    int len;
    int start = -1;
    int j = 0;
    int i;
    int flag = 0;

    len = read(STDIN_FILENO, inputBuffer, MAX_LINE); //read the command
    inputBuffer[len] = '\0';

    if(strcmp(inputBuffer, "quit\n") == 0 || strcmp(inputBuffer, "exit\n") == 0)
        exit(0);

    if(inputBuffer[0] == 'r' && NumOfCommand > 0){
        if(inputBuffer[1] == '\n'){
            strcpy(inputBuffer, history[NumOfCommand - 1]);
        }
        else if(inputBuffer[1] == ' ' && inputBuffer[2] != '\0' && inputBuffer[3] == '\n'){
            for(i = NumOfCommand - 1; i >= 0; --i){
                if(inputBuffer[2] == history[i][0]){
                    strcpy(inputBuffer, history[i]);
                    flag = 1;
                    break;
                }
            }
            //if there is no command match to the r x, then return and no action to the history
            if(!flag){
                args[0] = NULL;
                return;
            }
        }
        len = strlen(inputBuffer);
    }

    if(len < 0){  //ctrl c
        args[0] = NULL;
        return;
    }

    if(len == 0){  //ctrl d
        printf("\n");
```

```
        exit(0);
    }

    strcpy(history[(start_index + NumOfCommand) % 10], inputBuffer);
    if(NumOfCommand == 10){ //if the num of history command is 10,move the olddest command out
        start_index = (start_index + 1 + 10) % 10;
    }
    else{
        ++NumOfCommand;
    }

    //seperating the command to the distinct tokens
    for(i = 0; i < len; ++i){
        if(inputBuffer[i] == ' ' || inputBuffer[i] == '\t'){ //find the delimiters
            if(start != -1){
                args[j++] = &inputBuffer[start];
                start = -1;
            }
            inputBuffer[i] = '\0';
        }
        else if(inputBuffer[i] == '\n'){
            if(start != -1){
                //inputBuffer[i] = '\0';
                args[j++] = &inputBuffer[start];
            }
            args[j] = NULL;
            inputBuffer[i] = '\0';
        }
        else{
            if(inputBuffer[i] == '&'){
                *background = 1;
                inputBuffer[i] = '\0';
            }
            if(start == -1)
                start = i;
        }
    }
    args[j] = NULL;
}


// catch the ctrl c signal
void handle_SIGINT(){
    write(STDOUT_FILENO, buffer, strlen(buffer));
    printf("The Command History:\n");
    int i;
    i = start_index;
    int j = 0;
    fflush(stdout);
    for(; i < NumOfCommand + start_index; ++i){
        printf("%d. %s", ++j, history[i % 10]);
    }
    return;
}


int main(){
    char input[MAX_LINE];
    int background;
    char *args[MAX_LINE/2+1];

    //set up the signal handler
    struct sigaction handler;
```

```
    handler.sa_handler = handle_SIGINT;
    sigaction(SIGINT, &handler, NULL);

    strcpy(buffer, "\nCaught Control C\n");

    while(1)
    {
        background = 0;
        fflush(stdin);
        printf("COMMAND->");
        fflush(stdout);   //print the stdout buffer
        background = 0;
        setup(input, args, &background); //read the command

        //deal with cd command
        if(strcmp(input, "cd")==0){
            if(*args[1] == '~'){
                strcpy(args[1], "/home/hankunyan");
            }
            if(chdir(args[1]) == 0) continue;
        }

        if(args[0]!= NULL){
            pid_t pid;
            pid = fork();
            if(pid < 0){
                printf("fork failed\n");
                exit(1);
            }
            else if(pid == 0){
                execvp(args[0], args);
                background = 0;
                printf("ERROR COMMAND!\n");
                exit(0);
            }
            else{
                if(background == 0) waitpid(pid, NULL, 0);
            }
        }
    }
    return 0;
}
```