515030910605
李洋
2017年11月4日

Project2
Unix Shell History Feature

1. Introduction

In this experiment, we need to write a code file work as a shell to accept user's command.

These are the things I have done in this code file:

a) Accept and modify the input.

b) Execute the command (in the background)

c) Add the command to the history

d) Execute the latest command

e) Execute the command with the beginning of specific letter

f) Capture the Ctrl+C to run the code of signal.

g) Display the history list

I will introduce my method in the next part.

2. Steps

2.1. Experiment environment

Version of Ubuntu: 16.04

2.2. Workflow

(1) We need a code serves as a shell to accept the user's command. After the inputBuffer gets the input of user, we divide it with blank space and store each block into args[]. If we get 'r' or 'r x' command, we should replace the command with latest command in history ('r') or latest command with the beginning of specific letter 'x' ('r x').

(2) Add the command to the history queue. Record the index of latest command, for we will show the history according to the order of time.

(3) Use execvp() function to execute the command in main().

(4) Capture Ctrl+C to show the history list.

In a word, the project can be divided into 4 parts:

a. Get, modify the command.

b. Create child process to execute the command.

c. Record history.

d. Capture the Ctrl+C to show the history list.

2.3. Get, modify the command

We do these operations in setup() function.

1. We get the input first and put it into inputBuffer[]. Then we divide it with blank space. Luckily, strtok() is a useful function to help us. And if the command end with '&', we should change the value of 'background' to 1.

```
//get input
while( (c = getchar()) != EOF && len < MAX_LINE ){
      if ('\n' == c) break;
      inputBuffer[len++] = c;
}
inputBuffer[len] = '\0';
if(len == 0) return;
//background
if(inputBuffer[len - 1] == '&') {*background = 1; inputBuffer[len - 1] = '\0';}
```

2. Store these block in args[]. It's worth noting that args[] stores the pointer rather than string.

```
//divide by space
p = strtok(inputBuffer,d);
args[index] = p;
while ((p = strtok(NULL,d))){
    index++;
    args[index] = p;
}
```

3. "r" and "r x" command are special commands we should pay attention to. "r" means that we should replace the inputBuffer[] with latest command in history list. "r x" means that we should replace the inputBuffer[] with latest command with the beginning of specific letter 'x' in history list. In particularly, there are some special situations such as there is no latest history record or there is no history record with the beginning of specific letter 'x'. The corresponding disposal was done in my code.

```
//r & rx
if(inputBuffer[0] == 'r'){
      //r
      int tmp_n = n;
      if(tmp_n == 0) tmp_n = 9;
      else tmp_n--;
      if(inputBuffer[1] == '\0'){
         if(history[tmp_n] == '\0'){
              printf("No history!\n");
              return;
          }
         strcpy(inputBuffer,history[tmp_n]);
      }
      //rx
      if(inputBuffer[1] == ' ' && inputBuffer[2] != '\0' && inputBuffer[3] == '\0'){
        int cont = 1;
        int i = n - 1;
        if (i<0) i = i + 10;
        while(1){
          if( cont>10 || history[i] == NULL){
              printf("No proper history!\n");
              return;
          }
          if(history[i][0] == inputBuffer[2]){
              strcpy(inputBuffer,history[i]);
              break;
          }
          cont ++;
          i--;
          if (i<0) i = i + 10;
        }
      }

}
```

2.4. Create child process to execute the command.

This operation are executed in main() function.

If the command is not empty, we can create a child process to execute it. fork() is a function to create a child process. In child process, we use execvp() to execute our command. If our command is executed successfully, execvp() will return nothing and finish our child process. If there is an error in our command execution, execvp() will return -1. So I add 'printf("error!\n")' to tell the user that the command execution has an error.

Another key point we should concern is that running command in background. Running command in background means that parent process doesn't wait for child process' finish. So waitpid() can make child process doesn't run in background.

```
while(1){
     background = 0;

     fflush(stdin); // clean inputBuffer

     printf("COMMAND->");

     fflush(stdout); // clean outputBuffer
     setup(inputBuffer, args, &background);

     if(args[0] != NULL){
        pid_t pid;
        pid = fork();
        if(pid<0){
             printf("Process Error!\n");
             exit(1);
        }
        else if(pid == 0){
             execvp(args[0],args);
             background = 0;
             printf("error\n");
             exit(0);
        }
        else{
             if(background == 0) waitpid(pid,NULL,0);// No wait means run background
        }
     }
}
```

2.5. Record history.

This operation are executed in main() function.

We use a history[10] to record latest 10 commands which user input and a integer n to record the index. Every time we push a history record, we let n = (n + 1)%10. So we can cover old history if records more than 10.

In particular, if we input same commands continuously, we only record once.

```
//push in history
int tmp_n = n;
if(tmp_n == 0) tmp_n = 9;
else tmp_n--;
if(history[tmp_n] == NULL || strcmp(inputBuffer,history[tmp_n])){
        history[n] = (char*)malloc(strlen(inputBuffer));
        strcpy(history[n],inputBuffer);
        n = (n + 1)%10;
}
```

2.6. Capture the Ctrl+C to show the history list.

This operation are executed in main() function. And we should rewrite the handle_SIGINT() function to display the history.

We add signal() at the end of the main() function to capture the Ctrl+C signal.

```
//capture the ctrl+C
signal(SIGINT,handle_SIGINT);
```

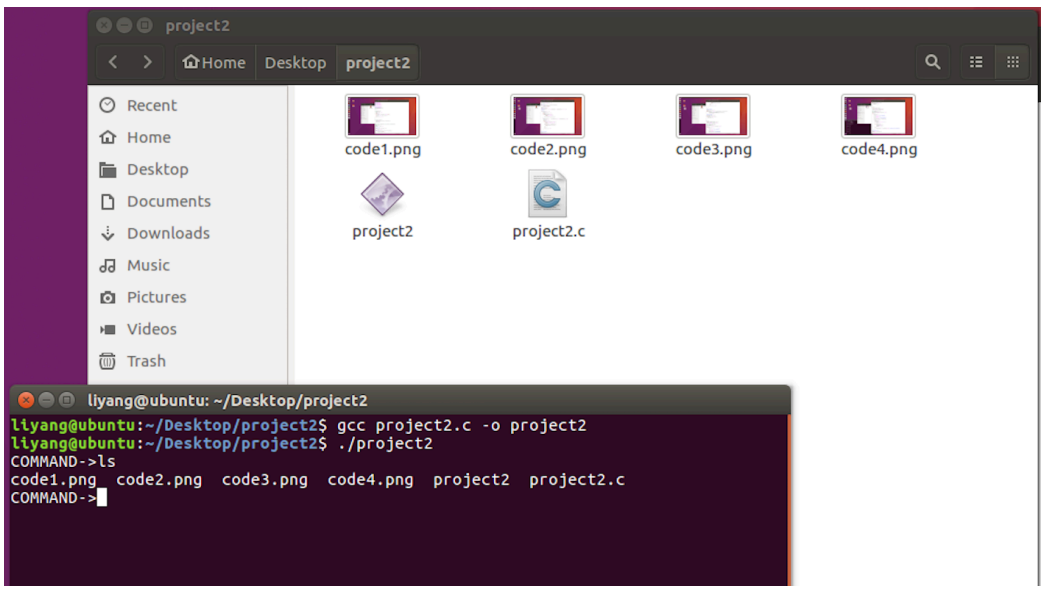
And we write a displayhistory() function to show history list. The handle_SIGINT() and displayhistory() are defined before the main() function.

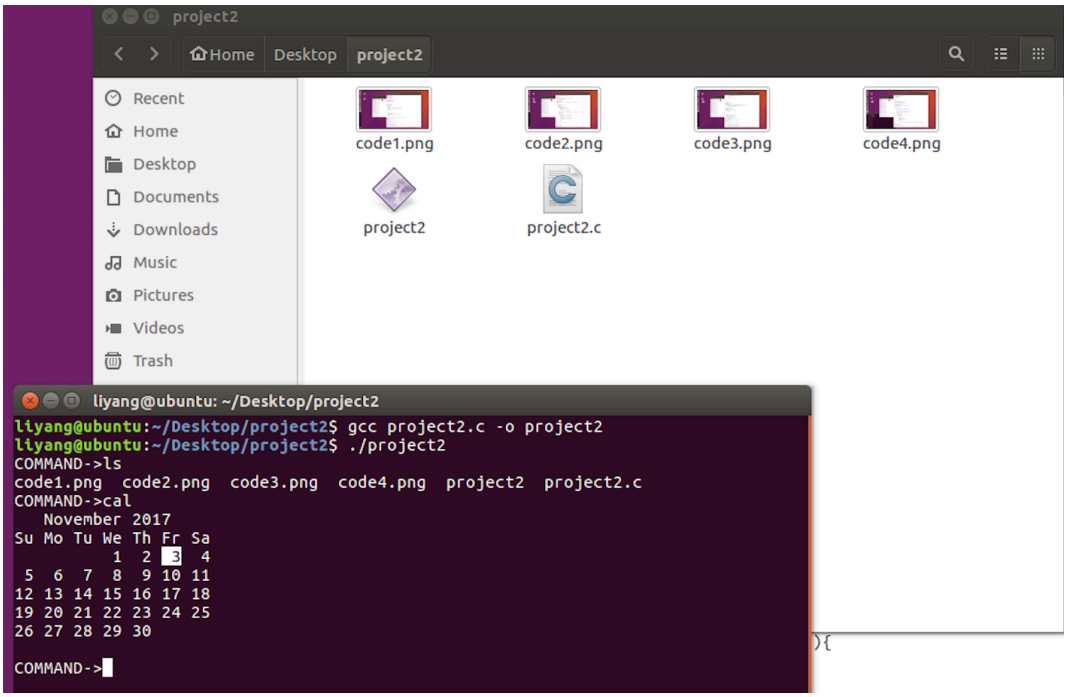
```c
void showhistory(){
  int cont = 1;
  int i = n - 1;
  if (i<0) i = i + 10;
  printf("\n");
  while(1){
    if( cont>10 || history[i] == NULL) break;
    printf("%d\t%s\n",cont,history[i]);
    cont ++;
    i--;
    if (i<0) i = i + 10;
  }
}


void handle_SIGINT(){
  write(STDOUT_FILENO, buffer, strlen(buffer));
  fflush(stdout);
  showhistory();

}
```

3. Test result show

a) Compile and run the code

We use ‘ls’ and ‘cal’ to show result.





b) Input Ctrl + C to show history.

c) Run command in the background.

We take an example of ‘ls &’. We can see that parent process didn’t wait for child process. So the “COMMAND->” shows before child process print the filename list.

d) Test ‘r’ and ‘r x’.

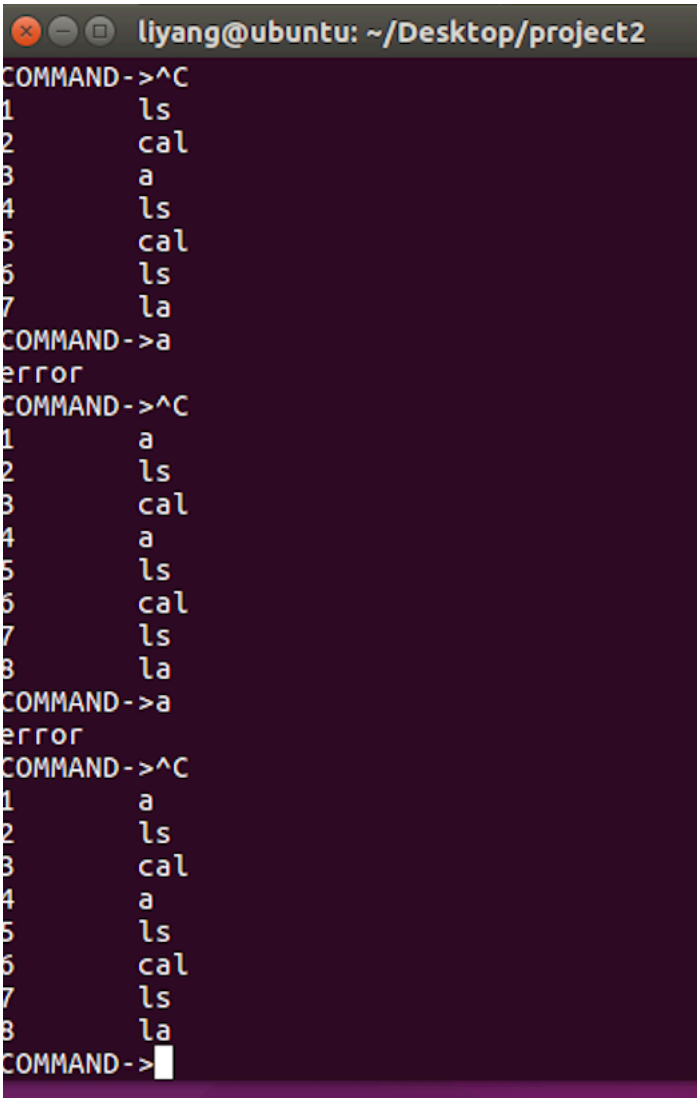
We test ‘r’ and ‘r l’. It is obviously that ‘r’ repeat the latest command and ‘r x’ repeat the latest command with beginning of letter ‘l’ (That is ‘ls’).

e) Error test.

We input an error command such as ‘a’. We can see that the console will print ‘error’.

f) Repeat command test.

We input the same command continuously such as ‘a’. It will be recorded in history only once.



4. Conclusion

In this experiment, I understand how a shell process a command: Get command from inputBuffer and modify it. Then run it in process. And I also know how to add a shell with history function: store the command in history list and

maintain a certain number of commands. In addition, I understand how to capture Ctrl + C signal.

This experiment let me understand how the operating system shell executed user command and how to show the history if command list. It really improve my skill of Linux.

Appendix: code

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.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];
int n = 0;


void showhistory(){
  int cont = 1;
  int i = n - 1;
  if (i<0) i = i + 10;
  printf("\n");
  while(1){
    if( cont>10 || history[i] == NULL) break;
    printf("%d\t%s\n",cont,history[i]);
    cont ++;
    i--;
    if (i<0) i = i + 10;
  }
}


void handle_SIGINT(){
  write(STDOUT_FILENO, buffer, strlen(buffer));
  fflush(stdout);
  showhistory();

}

void setup(char inputBuffer[], char *args[], int *background){
  char c;
  const char *d = " ";
  char *p;
  int len = 0, index = 0;
```

```
//initialize
for(int i = 0; i< MAX_LINE/2; i++)
        args[i] = NULL;

//get input
while( (c = getchar()) != EOF && len < MAX_LINE ){
        if ('\n' == c) break;
        inputBuffer[len++] = c;
}
inputBuffer[len] = '\0';
if(len == 0) return;
//background
if(inputBuffer[len - 1] == '&') {*background = 1; inputBuffer[len - 1] = '\0';}

//r & rx
if(inputBuffer[0] == 'r'){
        //r
    int tmp_n = n;
    if(tmp_n == 0) tmp_n = 9;
    else tmp_n--;
        if(inputBuffer[1] == '\0'){
      if(history[tmp_n] == '\0'){
                printf("No history!\n");
                return;
            }
          strcpy(inputBuffer,history[tmp_n]);
        }
        //rx
        if(inputBuffer[1] == ' ' && inputBuffer[2] != '\0' && inputBuffer[3] == '\0'){
          int cont = 1;
          int i = n - 1;
          if (i<0) i = i + 10;
          while(1){
            if( cont>10 || history[i] == NULL){
                printf("No proper history!\n");
                return;
            }
       if(history[i][0] == inputBuffer[2]){
                strcpy(inputBuffer,history[i]);
                break;
            }
            cont ++;
            i--;
            if (i<0) i = i + 10;
          }
        }

}

//push in history
int tmp_n = n;
if(tmp_n == 0) tmp_n = 9;
else tmp_n--;
if(history[tmp_n] == NULL || strcmp(inputBuffer,history[tmp_n])){
          history[n] = (char*)malloc(strlen(inputBuffer));
          strcpy(history[n],inputBuffer);
          n = (n + 1)%10;
}
//divide by space
p = strtok(inputBuffer,d);
args[index] = p;
while ((p = strtok(NULL,d))){
   index++;
```

```
    args[index] = p;
  }

}

int main()
{
  struct sigaction handler;
  handler.sa_handler = handle_SIGINT;
  sigaction(SIGINT, &handler, NULL);

  char inputBuffer[MAX_LINE];
  int background;
  char *args[MAX_LINE/2 + 1];
  for(int i = 0; i<10; i++) history[i] = NULL;

  while(1){
          background = 0;

          fflush(stdin); // clean inputBuffer

          printf("COMMAND->");

          fflush(stdout); // clean outputBuffer
          setup(inputBuffer, args, &background);

      if(args[0] != NULL){
            pid_t pid;
        pid = fork();
        if(pid<0){
                    printf("Process Error!\n");
            exit(1);
             }
             else if(pid == 0){
                    execvp(args[0],args);
                    background = 0;
            printf("error\n");
            exit(0);
             }
        else{
                    if(background == 0) waitpid(pid,NULL,0);// No wait means run background
             }
           }
  }
  //capture the ctrl+C
  signal(SIGINT,handle_SIGINT);
  return 0;
}
```