```
/*
================================================================================
 *      Filename:  shell.c                    Version:  1.0
 *      Created:  2015/09/20                  Author:  Shuaiqi Cao
================================================================================
 */
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

char *read_line(void);
char *shell_prompt(char *prompt);
int input_phrase(char *input);
int execute_pipe(char *input);
int execute_redirection(char *input);
int check_if_file(char* input, int start, int end);
int cd_command(char **args);
int EOF_command(char **args);
int exit_command(char **args);
int execute_single(char *args);
#define DELIM " \t\r\n\a"
#define PATHSIZE 1024

int main(int argc, char **argv)
{
        int status=1;
        while(status){
                //Shell prompt by using getcwd()
                char prompt[PATHSIZE];
                fprintf(stdout, "[myShell]:~%s", shell_prompt(prompt));

                //Get input
                char *input=read_line();

                //Phrase the input command, return the execution type
                int type = input_phrase(input);

                //Four types of executions, 0-invalid input; 1-single command; 2-pipe; 3-
redirection
                switch(type){
                case 0:
                        printf("Invalid input\n");
                        break;
```

```
                case 1:
                        status=execute_single(input);
                        break;
                case 2:
                        execute_pipe(input);
                        break;
                case 3:
                        execute_redirection(input);
                        break;
                }

        }
        return 0;
}

//Read in the input by getline()
char *read_line(void)
{
        char *line = NULL;
        ssize_t bufsize = 0;
        getline(&line, &bufsize, stdin);
        return line;
}

//Read CWD and print out as shell prompt
char *shell_prompt(char *prompt)
{
        char prompt_Buf[PATHSIZE];
        memset(prompt, 0, PATHSIZE );
        memset(prompt_Buf, 0, sizeof(prompt_Buf));
        if(getcwd(prompt_Buf, sizeof(prompt_Buf)) == NULL){
                fprintf(stderr, "%s:%d: Unknown: %s\n", __FILE__, __LINE__, strerror(errno));
        }
        snprintf(prompt, PATHSIZE, "%s$ ", prompt_Buf);
        return prompt;
}

//Return integer, 0-invalid input; 1-single command; 2-pipe; 3-redirection
int input_phrase(char *input)
{
        //check input length
        int len=strlen(input);
        if(len>101){
                return 0;
        }
        //check the first char of input by Ascii code
        if((toascii(input[0])==47)||(toascii(input[0])>=65&&toascii(input[0])<=90)||
          (toascii(input[0])>=97&&toascii(input[0])<=122)){
                //check the last char of input
```

```
                if((toascii(input[len-2])>=65&&toascii(input[len-2])<=90)||
(toascii(input[len-2])>=97&&toascii(input[len-2])<=122)||
                (toascii(input[len-2])>=48&&toascii(input[len-2])<=57)){
                }else{
                        return 0;
                }

        }else{
                return 0;
        }

        //check the operators and invalid char in the whoel input
        int pipe_number=0;
        int rdin_number=0;
        int rdout_number=0;
        int last_pipe_position=0;
        int rdin_position=0;
        int rdout_position=0;
        int i=0;
        for(i;i<strlen(input)-1;i++){
                if((toascii(input[i])>=45&&toascii(input[i])<=57)||
(toascii(input[i])>=65&&toascii(input[i])<=90)||
              (toascii(input[i])>=97&&toascii(input[i])<=122)||toascii(input[i])==32 ||
toascii(input[i])==95||
                toascii(input[i])==60 ||toascii(input[i])==62||toascii(input[i])==124){
                        if((input[i]=='|')||(input[i]=='<')||(input[i]=='>')){
                                if((input[i-1]==' ')&&(input[i+1]==' ')){
                                        if(input[i]=='|'){
                                                pipe_number++;
                                                last_pipe_position=i;
                                        }else if(input[i]=='<'){
                                                rdin_number++;
                                                rdin_position=i;
                                        }else if(input[i]=='>'){
                                                rdout_number++;
                                                rdout_position=i;
                                        }else;
                                }else{
                                        return 0;
                                }
                        }
                }else{
                        return 0;
                }
        }

        //Check numbers of three operators and filter the invalid input
        //Use the position of operator to check wether the output, input files are valid
        if((pipe_number>0)&&(rdin_number==0)&&(rdout_number==0)){
```

```
                return 2;
        }else if((rdin_number==0)&&(rdout_number==1)){
                if(check_if_file(input, rdout_position, len)){
                        return 3;
                }else{
                        return 0;
                }
        }else if((rdin_number==1)&&(rdout_number==0)){
                if(check_if_file(input, rdin_position, len)){
                        return 3;
                }else{
                        return 0;
                }
        }
        else
if((pipe_number==0)&&(rdin_number==1)&&(rdout_number==1)&&(rdin_position<rdout_positio
n)){
                if(check_if_file(input, rdin_position, rdout_position) && check_if_file(input,
rdout_position, len)){
                        return 3;
                }else{
                        return 0;
                }
        }else if ( (pipe_number==0)&&(rdin_number==0)&&(rdout_number==0)){
                return 1;
        }else{
                return 0;
        }
}

//Subfunction for check wether the output and input files are valid
//return 0 means it is not a file, 1 means it is
int check_if_file(char* input, int head, int tail)
{
        int start = head+2;
        int end = tail-2;
        int begin_flag, stop_flag;
        int i = start;
        for(i;i<end;i++){
                if(input[i]!=' '){
                        begin_flag=i;
                        break;
                }else;
        }

        int j=end;
        for(j;j>start;j--){
                if(input[j]!=' '){
                        stop_flag=j;
```

```
                        break;
                }else;
        }
        int n = begin_flag;
        for(n; n<=stop_flag; n++){
                if(toascii(input[n])==32){
                        return 0;
                }else;
        }
        return 1;
}

//execute pipe operation. Use a 2d array to store all words.
int execute_pipe(char *input)
{
        char *token;
        char *tokens[50][50];
        char *pipe_tokens[50];
        char *pipe_command;

        //Firstly split input according to |, then split single command according to space
      int position = 0;
        pipe_command = strtok(input, "|");
      while (pipe_command!= NULL) {
                pipe_tokens[position] = pipe_command;
                position++;
                pipe_command = strtok(NULL, "|");
        }
        pipe_tokens[position] = NULL;

        int status;
        int i=0;
      for(i;i<=(position-1);i++){
                int position1 = 0;
                token=strtok(pipe_tokens[i],DELIM);
                while (token != NULL){
                        tokens[i][position1] = token;
                        position1++;
                        token= strtok(NULL, DELIM);
                }
                tokens[i][position1] = NULL;
      }

        //build pipes according to the number of |
      int pipe_number;
      pipe_number=position-1;
        int pipefd[25][2];
        int pid;
      for(i=0;i<pipe_number;i++){
```

```
        pipe(pipefd[i]);
    }
    for(i=0;i<position;i++){
            int arg=0;
        pid =fork();
        if(pid==0){
                if(i==0){
                        close(1);
                    dup(pipefd[i][1]);
                    for (arg = 0; arg < pipe_number; arg++){
                            //make pipes be one direction
                             close(pipefd[arg][0]);
                             close(pipefd[arg][1]);
                        }
                    execvp(tokens[i][0], tokens[i]);
                }
                else if((0<i)&&(i<pipe_number)){
                        close(0);
                        close(1);
                        dup(pipefd[i-1][0]);
                        dup(pipefd[i][1]);
                        for (arg = 0; arg < pipe_number; arg++){
                                close(pipefd[arg][0]);
                                close(pipefd[arg][1]);
                        }
                        execvp(tokens[i][0], tokens[i]);
                    }else{
                        close(0);
                        dup(pipefd[i-1][0]);
                        for (arg = 0; arg < pipe_number; arg++){
                                close(pipefd[arg][0]);
                                close(pipefd[arg][1]);
                        }
                        execvp(tokens[i][0], tokens[i]);
                    }
            }
        }
    for (i= 0; i<pipe_number; i++){
                close(pipefd[i][0]);
                close(pipefd[i][1]);
        }
        for(i= 0; i<position; i++){
                wait(&status);
        }
        return 0;
}

//execute redirection operation.
int execute_redirection(char *input)
```

```
{
	//use tokens to store the arguements of command
	char *token;
	char **tokens = malloc(64 * sizeof(char*));
	char *command;
	char *output_file, *input_file;

	command = strtok(input, ">");
	output_file = strtok(NULL, DELIM);

	command = strtok(input, "<");
	input_file = strtok(NULL, DELIM);

	//If the input file is not in current directory, report error and return to main function
	if((input_file!=NULL)&&(access(input_file,F_OK))==-1){
		printf("This file is not in current directory: %s\n",input_file);
		return;
	}

	int position = 0;
	token=strtok(command,DELIM);
	while (token != NULL) {
		tokens[position] = token;
		position++;
		token = strtok(NULL, DELIM);
	}
	tokens[position] = NULL;

	int in, out;
	pid_t pid;
	int status;
	pid = fork();
	if (pid==0){
		in = open(input_file, O_RDONLY);
		out = open(output_file, O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR |
S_IRGRP |S_IWGRP| S_IWUSR);
		dup2(in, 0);
		dup2(out, 1);
		close(in);
		close(out);
		execvp(tokens[0], tokens);
	}else if(pid<0){
		perror("Error forking.\n");
	}else{
		waitpid(0,&status,WUNTRACED);
	}
	return 0;
}
```

```
//Three builtin single commands
char *builtin_str[] = {
        "cd",
        "EOF",
        "exit"
};

int(*builtin_func[])(char **)={
        &cd_command,
        &EOF_command,
        &exit_command
};

int builtins_number()
{
        return sizeof(builtin_str)/sizeof(char *);
}
int cd_command(char **args)
{
        if(args[1] == NULL){
          fprintf(stderr, "Expected directory to \"cd\"\n");
        }else{
            if(chdir(args[1])!= 0){
               perror("myShell");
            }
        }
        return 1;
}

int exit_command(char **args)
{
        return 0;
}
int EOF_command(char **args)
{
        return 0;
}

//execute single command
int execute_single(char *line)
{
        //Split the input string into arguements
        int bufsize = 1024, position = 0;
        char **tokens = malloc(bufsize * sizeof(char*));
        char *token;

        if(!tokens){
                fprintf(stderr, "Fail allocation\n");
                exit(EXIT_FAILURE);
```

```
        }

        token = strtok(line, DELIM);
        while(token != NULL){
                tokens[position] = token;
                position++;

                if(position>=bufsize){
                        bufsize += 1024;
                        tokens = realloc(tokens, bufsize * sizeof(char*));
                        if(!tokens){
                              fprintf(stderr, "Fail allocation\n");
                        exit(EXIT_FAILURE);
                        }
                }
                token = strtok(NULL, DELIM);
        }
        tokens[position] = NULL;


        if(tokens[0]==NULL){
                return 1;
        }
        int i;
        for(i=0; i<builtins_number(); i++){
                if(strcmp(tokens[0], builtin_str[i]) == 0){
                        return (*builtin_func[i])(tokens);
                }
        }

        //fork a child process to execute
        pid_t pid, wpid;
        int status;
        pid = fork();
        if (pid == 0){
                if(execvp(tokens[0],tokens)==-1){
                        perror("myShell");
                }
                exit(EXIT_FAILURE);
        }else if(pid < 0){
                perror("myShell");
        }else{
                do{
                        wpid = waitpid(pid, &status, WUNTRACED);
                }while(!WIFEXITED(status) && !WIFSIGNALED(status));
        }
  return 1;
}
```