75.08 Sistemas Operativos

Entrega Labs Parte 3

Nombre y Apellido: Martín Nicolás Pérez
Padrón: 97378
Fecha de Entrega: 13/04/2018

Índice

1. Shell Segunda Entrega

En este informe se mostrarán el código implementado que responde a las consignas de la primera parte del Lab Shell. Para cada parte, se detalla una breve descripción de las implementaciones, adjuntando las correspondientes fracciones de código. Ademas, se expone un ejemplo de la funcionalidad que debe realizar el shell, cumpliendo las consigas, mostrando los resultados esperados. Finalmente se adjunta el código fuente completo del esqueleto como anexo.

1.1. Comandos Built-in

El objetivo de la siguiente implementación es poder ejecutar ciertos comandos estandares que deben ejecutarse en el mismo proceso en el cual se esta ejecutando el shell, los conocidos built-in. En el siguiente archivo se mostrará los comandos *cd*, *pwd* y *exit* desarrollados para esta sección.

Código Fuente:

```
1 #include "builtin.h"
2
3 // returns true if the 'exit' call
4 // should be performed
5 int exit_shell(char* cmd) {
6
7     //Your code here
8
9     int ret = 0;
10    char* aux = malloc(strlen(cmd));
11    strcpy(aux,cmd);
12    split_line(aux,SPACE);
13    if(strcmp(aux,"exit") == 0){
14        ret = 1;
15    }
16
17    free(aux);
18    return ret;
19 }
20
21 // returns true if "chdir" was performed
22 // this means that if 'cmd' contains:
23 //   $ cd directory (change to 'directory')
24 //   $ cd (change to HOME)
25 // it has to be executed and then return true
26 int cd(char* cmd) {
27
28    //Your code here
```

```
29
30      char* aux = malloc(strlen(cmd));
31      int ret = 0;
32
33      strcpy(aux, cmd);
34      char* directory;
35      directory = split_line(aux,SPACE);
36      if(strcmp(aux,"cd") == 0){
37          ret = 1;
38          int cdErr = 0;
39          if(strlen(directory) != 0 && strlen(aux) > 2){
40              cdErr = chdir(directory);
41          }else{
42              cdErr = chdir(getenv("HOME"));
43              strcpy(promt,getenv("HOME"));
44          }
45          if(cdErr < 0){
46              perror("Error");
47          }
48          char* pwd = get_current_dir_name();
49          strcpy(promt,pwd);
50          free(pwd);
51      }
52
53      free(aux);
54      return ret;
55  }
56
57  // returns true if 'pwd' was invoked
58  // in the command line
59  int pwd(char* cmd) {
60
61      // Your code here
62
63      int ret = 0;
64      char* aux = malloc(strlen(cmd));
65
66      strcpy(aux,cmd);
67      split_line(aux,SPACE);
68      if(strcmp(aux,"pwd") == 0){
69          ret = 1;
70          //Al invocar este medoto, el mismo hace un malloc para
    ↪ setear el buffer, despues de usarlo
71          //deberia hacer un free para limpiar la memoria.
72          char* buffer = get_current_dir_name();
73          printf("%s\n", buffer);
74          free(buffer);
75      }
76
77      free(aux);
78      return ret;
```

```
79 }
```

builtin.c

Ejemplo:

```
1 CHANGE DIRECTORY
2
3       $cd
4       /home/user
5
6       $cd dir
7       /home/user/dir
8
9       $cd /usr/bin/
10      /usr/bin/
11
12 EXIT
13
14      $exit
15
16 PWD
17
18      $pwd
19      /home/user
```

exampleBuiltin.txt

Pregunta: ¿entre cd y pwd, alguno de los dos se podría implementar sin necesidad de ser built-in? ¿por qué? ¿cuál es el motivo, entonces, de hacerlo como built-in?

Respuesta: Podrian ser implementados como programas o comandos sin necesidad de ser built-in. Como observación a destacar, en este computador se intento llamar al comando *pwd* sin tener implementado el built-in, y funcionó perfectamente. Al ser programas, pueden ser implementados por cualquier persona y pueden ser invocados tanto directamente con su ruta completa, o setearlos para que su comando sea buscado en *PATH*. La idea de tenerlo en built-in es convencionalmente para que corran en el proceso padre donde corre el shell. Por ejemplo, con el built-in *exit* si este no corre en el padre, no se termina de manera elegante el shell, ya que si corre en un proceso hijo, terminara ese proceso y no afectara al que corre el shell, lo cual no tendria ninguna utilidad.

1.2. Variables de Entorno adicionales

En esta sección se le agrega al shell, la posibilidad de setear variables de entorno adicionales a las que ya contiene. Se concidera que solamente el objetivo es poder setear una variable de entorno nueva en el proceso hijo del proceso donde esta corriendo el shell. Esto implica que la variable de entorno permanecerá configurada en el proceso hijo y desaparecerá cuando este termine. Por lo tanto la unica aplicación para esto será ejecutar el programa */usr/bin/env* el cual mostrara el valor de dicha

variable. Posteriormente no se encontrará configurada. En el siguiente archivo se muestra el fragmento de código utilizado para esta funcionalidad, perteneciente al archivo *exec.c*

Código Fuente:

```
1  // sets the "key" argument with the key part of
2  // the "arg" argument and null-terminates it
3  static void get_environ_key(char* arg, char* key) {
4
5      int i;
6      for (i = 0; arg[i] != '='; i++)
7          key[i] = arg[i];
8
9      key[i] = END_STRING;
10 }
11
12 // sets the "value" argument with the value part of
13 // the "arg" argument and null-terminates it
14 static void get_environ_value(char* arg, char* value, int idx) {
15
16     int i, j;
17     for (i = (idx + 1), j = 0; i < strlen(arg); i++, j++)
18         value[j] = arg[i];
19
20     value[j] = END_STRING;
21 }
22
23 // sets the environment variables passed
24 // in the command line
25 //
26 // Hints:
27 // - use 'block_contains()' to
28 //   get the index where the '=' is
29 // - 'get_environ_*()' can be useful here
30 static void set_environ_vars(char** eargv, int eargc) {
31
32     // Your code here
33     int p;
34         for(int i = 0; i < eargc; i++){
35             char* varEnv = eargv[i];
36             int idx = block_contains(varEnv,'=');
37             if(idx != -1){
38                 setEnvVar = 1;
39                 char* key = malloc(idx);
40                 char* value = malloc(strlen(varEnv) - (idx + 1));
41                 get_environ_key(varEnv,key);
42                 get_environ_value(varEnv,value,idx);
```

```
43                  int set = setenv(key,value,0);
44                  if(set < 0){
45                      perror("SetEnv");
46                  }
47                  free(key);
48                  free(value);
49              }
50          }
51 }
52
53
54 //Execute the command searching programs in PATH
55 //author: Martin Perez
56 void execute(struct cmd* cmd){
57     //Cast struct cmd* to struct execcmd* to use this interface
58     struct execcmd* execcmd = (struct execcmd*) cmd;
59
60     //Set Enviroment Variables
61     set_environ_vars(execcmd->eargv, execcmd->eargc);
62     //Si le paso el execcmd->scmd me va a guardar el comando con
    ↪ parametros y todo
63     //Debo pasarle execcmd->argv[0]
64     //Example ls -l -a -> execcmd->scmd = "ls -l -a"; execcmd->argv
    ↪ = {"ls","-l","-a",(char*)NULL}
65     execvp(execcmd->argv[0],execcmd->argv);
66     //En caso de que no haya programa que ejecutar salgo del proceso
67     _exit(-1);
68 }
```

env.c

Ejemplo:

```
1 $ ENTORNO=nada /usr/bin/env
2 ----todas las varialbes de entorno----
3 ENTORNO=nada
```

exampleEnv.txt

Pregunta: ¿por qué es necesario hacerlo luego de la llamada a fork(2) ? Respuesta: Es necesario ya que solo luego del fork, chequeando las condiciones necesarias, el proceso hijo al proceso que corre el shell, podrá setear las variables de entorno correctamente. El proceso hijo, es quien luego ejecutará el comando /usr/bin/env, por lo tanto si no es el quien configura sus variables de entorno, entonces no aparecera a la hora de imprimir todas las variables configuradas. En esta sección no se concidero que el padre también pueda imprimir dicha variable configurada por el hijo.

1.3. Procesos en segundo plano: Background

El último objetivo a cumplir para esta entrega, es proporcionar al shell, la capacidad de ejecutar programas en segundo plano. Esto permite seguir trabajando en el shell, aún si el ultimo comando invocado no finalizó. Para esto, lo propuesto en esta sección, es tomar la decisión de, si se invoca un comando en segundo plano, crear un proceso hijo que se encargue de ejecutarlo hasta su finalización y hacer que el proceso padre que ejecuta el shell no espere a que su hijo termine. Se propone chequear la condición de BACKGROUND, para determinar si se usa o no la función *waitpid()*. A continuación se muestran dos archivos, *runcmd.c* que contiene la condición a chequear y *back.c* que es el fragmento de código de *exec.c* el cual utiliza las funciones para los procesos background.

Código Fuente:

```
1 #include "runcmd.h"
2
3 int status = 0;
4 struct cmd* parsed_pipe;
5
6 // runs the command in 'cmd'
7 int run_cmd(char* cmd) {
8
9     pid_t p;
10    struct cmd *parsed;
11
12    // if the "enter" key is pressed
13    // just print the promt again
14    if (cmd[0] == END_STRING)
15        return 0;
16
17    // cd built-in call
18    if (cd(cmd))
19        return 0;
20
21    // exit built-in call
22    if (exit_shell(cmd))
23        return EXIT_SHELL;
24
25    // pwd buil-in call
26    if (pwd(cmd))
27        return 0;
28
29    // parses the command line
30    parsed = parse_line(cmd);
31
32    // forks and run the command
33    if ((p = fork()) == 0) {
```

```
34
35          // keep a reference
36          // to the parsed pipe cmd
37          // so it can be freed later
38          if (parsed->type == PIPE)
39              parsed_pipe = parsed;
40
41          exec_cmd(parsed);
42      }
43
44      // store the pid of the process
45      parsed->pid = p;
46
47      // background process special treatment
48      // Hint:
49      // - check if the process is
50      //   going to be run in the 'back'
51      // - print info about it with
52      //   'print_back_info()'
53      //
54      // Your code here
55      print_back_info(parsed);
56      if(parsed->type != BACK){
57          // waits for the process to finish
58          waitpid(p, &status, 0);
59          print_status_info(parsed);
60      }
61
62      free_command(parsed);
63
64      return 0;
65  }
```

runcmd.c

```
1  //Execute the command searching programs in PATH
2  //author: Martin Perez
3  void execute(struct cmd* cmd){
4      //Cast struct cmd* to struct execcmd* to use this interface
5      struct execcmd* execcmd = (struct execcmd*) cmd;
6
7      //Set Enviroment Variables
8      set_environ_vars(execcmd->eargv, execcmd->eargc);
9      //Si le paso el execcmd->scmd me va a guardar el comando con
       ↪ parametros y todo
10     //Debo pasarle execcmd->argv[0]
11     //Example ls -l -a -> execcmd->scmd = "ls -l -a"; execcmd->argv
       ↪ = {"ls","-l","-a",(char*)NULL}
12     execvp(execcmd->argv[0],execcmd->argv);
13     //En caso de que no haya programa que ejecutar salgo del proceso
```

```
14      _exit(-1);
15  }
16
17
18  //author: Martin Perez
19  void background(struct cmd* cmd){
20      //Cast struct cmd* to struct backmcmd*
21      struct backcmd* backcmd = (struct backcmd*) cmd;
22      //Call execute with struct cmd* contains in backcmd*
23      execute(backcmd->c);
24  }
25
26  // executes a command - does not return
27  //
28  // Hint:
29  // - check how the 'cmd' structs are defined
30  //   in types.h
31  void exec_cmd(struct cmd* cmd) {
32
33
34      switch (cmd->type) {
35
36          case EXEC:
37              // spawns a command
38              //
39              // Your code here
40              execute(cmd);
41              //printf("Commands are not yet implemented\n");
42              //_exit(-1);
43              break;
44
45          case BACK: {
46              // runs a command in background
47              //
48              // Your code here
49              background(cmd);
50              //printf("Background process are not yet implemented\n");
51              //_exit(-1);
52              break;
53          }
54
55          case REDIR: {
56              // changes the input/output/stderr flow
57              //
58              // Your code here
59              printf("Redirections are not yet implemented\n");
60              _exit(-1);
61              break;
62          }
63
64          case PIPE: {
65              // pipes two commands
```

```
66              //
67              // Your code here
68              printf("Pipes are not yet implemented\n");
69
70              // free the memory allocated
71              // for the pipe tree structure
72              free_command(parsed_pipe);
73
74              break;
75          }
76      }
77  }
```

back.c

Ejemplo:

```
1  Este ejemplo se concidera con notepadqq instalado previamente.
2
3  $ notepadqq &
4      [PID=2572]
5
6  $ ls
7    [PID=2606]
8  DesarrolloPersonal  Documentos  Plantillas  ticket.lib_files
9  Descargas           Escritorio
10     Program: [ls] exited, status: 0
11
12 --Luego cerrar el editor de texto
```

exampleBack.txt

2. Anexo

```
#ifndef BUILTIN_H
#define BUILTIN_H

#include "defs.h"
#include "utils.h"

extern char promt[PRMTLEN];

int cd(char* cmd);

int exit_shell(char* cmd);

int pwd(char* cmd);

#endif // BUILTIN_H
```

builtin.h

```
#include "builtin.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {

    //Your code here

    int ret = 0;
    char* aux = malloc(strlen(cmd));
    strcpy(aux,cmd);
    split_line(aux,SPACE);
    if(strcmp(aux,"exit") == 0){
        ret = 1;
    }

    free(aux);
    return ret;
}

// returns true if "chdir" was performed
// this means that if 'cmd' contains:
//  $ cd directory (change to 'directory')
//  $ cd (change to HOME)
// it has to be executed and then return true
int cd(char* cmd) {

    //Your code here

    char* aux = malloc(strlen(cmd));
    int ret = 0;

```

```
33      strcpy(aux, cmd);
34      char* directory;
35      directory = split_line(aux,SPACE);
36      if(strcmp(aux,"cd") == 0){
37          ret = 1;
38          int cdErr = 0;
39          if(strlen(directory) != 0 && strlen(aux) > 2){
40              cdErr = chdir(directory);
41          }else{
42              cdErr = chdir(getenv("HOME"));
43              strcpy(promt,getenv("HOME"));
44          }
45          if(cdErr < 0){
46              perror("Error");
47          }
48          char* pwd = get_current_dir_name();
49          strcpy(promt,pwd);
50          free(pwd);
51      }
52
53      free(aux);
54      return ret;
55  }
56
57  // returns true if 'pwd' was invoked
58  // in the command line
59  int pwd(char* cmd) {
60
61      // Your code here
62
63      int ret = 0;
64      char* aux = malloc(strlen(cmd));
65
66      strcpy(aux,cmd);
67      split_line(aux,SPACE);
68      if(strcmp(aux,"pwd") == 0){
69          ret = 1;
70          //Al invocar este medoto, el mismo hace un malloc para
        ↪ setear el buffer, despues de usarlo
71          //deberia hacer un free para limpiar la memoria.
72          char* buffer = get_current_dir_name();
73          printf("%s\n", buffer);
74          free(buffer);
75      }
76
77      free(aux);
78      return ret;
79  }
```

builtin.c

```
1 #ifndef CREATECMD_H
2 #define CREATECMD_H
```

```
3
4 #include "defs.h"
5 #include "types.h"
6
7 struct cmd* exec_cmd_create(char* cmd);
8
9 struct cmd* back_cmd_create(struct cmd* c);
10
11 struct cmd* pipe_cmd_create(struct cmd* l, struct cmd* r);
12
13 #endif // CREATECMD_H
```

createcmd.h

```
1 #include "createcmd.h"
2
3 // creates an execcmd struct to store
4 // the args and environ vars of the command
5 struct cmd* exec_cmd_create(char* buf_cmd) {
6
7     struct execcmd* e;
8
9     e = (struct execcmd*)calloc(sizeof(*e), sizeof(*e));
10
11     e->type = EXEC;
12     strcpy(e->scmd, buf_cmd);
13
14     return (struct cmd*)e;
15 }
16
17 // creates a backcmd struct to store the
18 // background command to be executed
19 struct cmd* back_cmd_create(struct cmd* c) {
20
21     struct backcmd* b;
22
23     b = (struct backcmd*)calloc(sizeof(*b), sizeof(*b));
24
25     b->type = BACK;
26     strcpy(b->scmd, c->scmd);
27     b->c = c;
28
29     return (struct cmd*)b;
30 }
31
32 // encapsulates two commands into one pipe struct
33 struct cmd* pipe_cmd_create(struct cmd* left, struct cmd* right) {
34
35     if (!right)
36         return left;
37
38     struct pipecmd* p;
39
```

```
40      p = (struct pipecmd*)calloc(sizeof(*p), sizeof(*p));
41
42      p->type = PIPE;
43      p->leftcmd = left;
44      p->rightcmd = right;
45
46      return (struct cmd*)p;
47  }
```

createcmd.c

```
 1  #ifndef DEFS_H
 2  #define DEFS_H
 3
 4  #define _GNU_SOURCE
 5
 6  #include <stdio.h>
 7  #include <stdbool.h>
 8  #include <stdlib.h>
 9  #include <string.h>
10  #include <errno.h>
11
12  #include <fcntl.h>
13  #include <unistd.h>
14  #include <signal.h>
15  #include <sys/wait.h>
16  #include <sys/types.h>
17
18  // color scape strings
19  #define COLOR_BLUE "\x1b[34m"
20  #define COLOR_RED "\x1b[31m"
21  #define COLOR_RESET "\x1b[0m"
22
23  #define END_STRING '\0'
24  #define END_LINE '\n'
25  #define SPACE ' '
26
27  #define BUFLEN 1024
28  #define PRMTLEN 1024
29  #define MAXARGS 20
30  #define ARGSIZE 1024
31  #define FNAMESIZE 200
32
33  // Command representation after parsed
34  #define EXEC 1
35  #define BACK 2
36  #define REDIR 3
37  #define PIPE 4
38
39  // Fd for pipes
40  #define READ 0
41  #define WRITE 1
42
```

```
43 #define EXIT_SHELL 1
44
45 #endif //DEFS_H
```

defs.h

```
1 #ifndef EXEC_H
2 #define EXEC_H
3
4 #include "defs.h"
5 #include "types.h"
6 #include "utils.h"
7 #include "freecmd.h"
8
9 extern struct cmd* parsed_pipe;
10
11 int setEnvVar;
12
13 void exec_cmd(struct cmd* c);
14
15 #endif // EXEC_H
```

exec.h

```
1 #include "exec.h"
2
3 // sets the "key" argument with the key part of
4 // the "arg" argument and null-terminates it
5 static void get_environ_key(char* arg, char* key) {
6
7     int i;
8     for (i = 0; arg[i] != '='; i++)
9         key[i] = arg[i];
10
11     key[i] = END_STRING;
12 }
13
14 // sets the "value" argument with the value part of
15 // the "arg" argument and null-terminates it
16 static void get_environ_value(char* arg, char* value, int idx) {
17
18     int i, j;
19     for (i = (idx + 1), j = 0; i < strlen(arg); i++, j++)
20         value[j] = arg[i];
21
22     value[j] = END_STRING;
23 }
24
25 // sets the environment variables passed
26 // in the command line
27 //
28 // Hints:
29 // - use 'block_contains()' to
```

```
30 //  get the index where the '=' is
31 // - 'get_environ_*()' can be useful here
32 static void set_environ_vars(char** eargv, int eargc) {
33
34     // Your code here
35     int p;
36         for(int i = 0; i < eargc; i++){
37             char* varEnv = eargv[i];
38             int idx = block_contains(varEnv,'=');
39             if(idx != -1){
40                 setEnvVar = 1;
41                 char* key = malloc(idx);
42                 char* value = malloc(strlen(varEnv) - (idx + 1));
43                 get_environ_key(varEnv,key);
44                 get_environ_value(varEnv,value,idx);
45                 int set = setenv(key,value,0);
46                 if(set < 0){
47                     perror("SetEnv");
48                 }
49                 free(key);
50                 free(value);
51             }
52         }
53 }
54
55 // opens the file in which the stdin/stdout or
56 // stderr flow will be redirected, and returns
57 // the file descriptor
58 //
59 // Find out what permissions it needs.
60 // Does it have to be closed after the execve(2) call?
61 //
62 // Hints:
63 // - if O_CREAT is used, add S_IWUSR and S_IRUSR
64 //  to make it a readable normal file
65 static int open_redir_fd(char* file) {
66
67     // Your code here
68     return -1;
69 }
70
71 //Execute the command searching programs in PATH
72 //author: Martin Perez
73 void execute(struct cmd* cmd){
74     //Cast struct cmd* to struct execcmd* to use this interface
75     struct execcmd* execcmd = (struct execcmd*) cmd;
76
77     //Set Enviroment Variables
78     set_environ_vars(execcmd->eargv, execcmd->eargc);
79     //Si le paso el execcmd->scmd me va a guardar el comando con
    ↪ parametros y todo
80     //Debo pasarle execcmd->argv[0]
```

```
81      //Example ls -l -a -> execcmd->scmd = "ls -l -a"; execcmd->argv
        ↪ = {"ls","-l","-a",(char*)NULL}
82      execvp(execcmd->argv[0],execcmd->argv);
83      //En caso de que no haya programa que ejecutar salgo del proceso
84      _exit(-1);
85  }
86
87  //Execute the command in background
88  //author: Martin Perez
89  void background(struct cmd* cmd){
90      //Cast struct cmd* to struct backmcmd*
91      struct backcmd* backcmd = (struct backcmd*) cmd;
92      //Call execute with struct cmd* contains in backcmd*
93      execute(backcmd->c);
94  }
95
96  // executes a command - does not return
97  //
98  // Hint:
99  // - check how the 'cmd' structs are defined
100 //   in types.h
101 void exec_cmd(struct cmd* cmd) {
102
103
104     switch (cmd->type) {
105
106         case EXEC:
107             // spawns a command
108             //
109             // Your code here
110             execute(cmd);
111             //printf("Commands are not yet implemented\n");
112             //_exit(-1);
113             break;
114
115         case BACK: {
116             // runs a command in background
117             //
118             // Your code here
119             background(cmd);
120             //printf("Background process are not yet implemented\n");
121             //_exit(-1);
122             break;
123         }
124
125         case REDIR: {
126             // changes the input/output/stderr flow
127             //
128             // Your code here
129             printf("Redirections are not yet implemented\n");
130             _exit(-1);
131             break;
```

```
132         }
133
134         case PIPE: {
135             // pipes two commands
136             //
137             // Your code here
138             printf("Pipes are not yet implemented\n");
139
140             // free the memory allocated
141             // for the pipe tree structure
142             free_command(parsed_pipe);
143
144             break;
145         }
146     }
147 }
```

exec.c

```
1 #ifndef FREECMD_H
2 #define FREECMD_H
3
4 #include "defs.h"
5 #include "types.h"
6
7 void free_command(struct cmd* c);
8
9 #endif // FREECMD_H
```

freecmd.h

```
1 #include "freecmd.h"
2
3 // frees the memory allocated
4 // for the tree structure command
5 void free_command(struct cmd* cmd) {
6
7     int i;
8     struct pipecmd* p;
9     struct execcmd* e;
10    struct backcmd* b;
11
12    if (cmd->type == PIPE) {
13
14        p = (struct pipecmd*)cmd;
15
16        free_command(p->leftcmd);
17        free_command(p->rightcmd);
18
19        free(p);
20        return;
21    }
22
```

```
23      if (cmd->type == BACK) {
24
25          b = (struct backcmd*)cmd;
26
27          free_command(b->c);
28          free(b);
29          return;
30      }
31
32      e = (struct execcmd*)cmd;
33
34      for (i = 0; i < e->argc; i++)
35          free(e->argv[i]);
36
37      for (i = 0; i < e->eargc; i++)
38          free(e->eargv[i]);
39
40      free(e);
41  }
```

freecmd.c

```
1 #ifndef PARSING_H
2 #define PARSING_H
3
4 #include "defs.h"
5 #include "types.h"
6 #include "createcmd.h"
7 #include "utils.h"
8
9 struct cmd* parse_line(char* b);
10
11 #endif // PARSING_H
```

parsing.h

```
1 #include "parsing.h"
2
3 // parses an argument of the command stream input
4 static char* get_token(char* buf, int idx) {
5
6     char* tok;
7     int i;
8
9     tok = (char*)calloc(ARGSIZE, sizeof(char));
10    i = 0;
11
12    while (buf[idx] != SPACE && buf[idx] != END_STRING) {
13        tok[i] = buf[idx];
14        i++; idx++;
15    }
16
17    return tok;
```

```
18 }
19
20 // parses and changes stdin/out/err if needed
21 static bool parse_redir_flow(struct execcmd* c, char* arg) {
22
23     int inIdx, outIdx;
24
25     // flow redirection for output
26     if ((outIdx = block_contains(arg, '>')) >= 0) {
27         switch (outIdx) {
28             // stdout redir
29             case 0: {
30                 strcpy(c->out_file, arg + 1);
31                 break;
32             }
33             // stderr redir
34             case 1: {
35                 strcpy(c->err_file, &arg[outIdx + 1]);
36                 break;
37             }
38         }
39
40         free(arg);
41         c->type = REDIR;
42
43         return true;
44     }
45
46     // flow redirection for input
47     if ((inIdx = block_contains(arg, '<')) >= 0) {
48         // stdin redir
49         strcpy(c->in_file, arg + 1);
50
51         c->type = REDIR;
52         free(arg);
53
54         return true;
55     }
56
57     return false;
58 }
59
60 // parses and sets a pair KEY=VALUE
61 // environment variable
62 static bool parse_environ_var(struct execcmd* c, char* arg) {
63
64     // sets environment variables apart from the
65     // ones defined in the global variable "environ"
66     if (block_contains(arg, '=') > 0) {
67
68         // checks if the KEY part of the pair
69         // does not contain a '-' char which means
```

```
70          // that it is not a environ var, but also
71          // an argument of the program to be executed
72          // (For example:
73          //  ./prog -arg=value
74          //  ./prog --arg=value
75          // )
76          if (block_contains(arg, '-') < 0) {
77              c->eargv[c->eargc++] = arg;
78              return true;
79          }
80      }
81
82      return false;
83  }
84
85  // this function will be called for every token, and it should
86  // expand environment variables. In other words, if the token
87  // happens to start with '$', the correct substitution with the
88  // environment value should be performed. Otherwise the same
89  // token is returned.
90  //
91  // Hints:
92  // - check if the first byte of the argument
93  //   contains the '$'
94  // - expand it and copy the value
95  //    to 'arg'
96  static char* expand_environ_var(char* arg) {
97
98      //Your code here
99      if(arg[0] == '$'){
100         //length: size of arg - 1
101         int length = strlen(arg);
102         char *aux = malloc(length);
103         for(int i = 1; i <= length; i++){
104             aux[i-1] = arg[i];
105         }
106         char *env = getenv(aux);
107         if(env != NULL){
108             strcpy(arg,env);
109         }else{
110             strcpy(arg," ");
111         }
112         free(aux);
113     }
114
115     return arg;
116 }
117
118 // parses one single command having into account:
119 // - the arguments passed to the program
120 // - stdin/stdout/stderr flow changes
121 // - environment variables (expand and set)
```

```
122 static struct cmd* parse_exec(char* buf_cmd) {
123 
124     struct execcmd* c;
125     char* tok;
126     int idx = 0, argc = 0;
127 
128     c = (struct execcmd*)exec_cmd_create(buf_cmd);
129 
130     while (buf_cmd[idx] != END_STRING) {
131 
132         tok = get_token(buf_cmd, idx);
133         idx = idx + strlen(tok);
134 
135         if (buf_cmd[idx] != END_STRING)
136             idx++;
137 
138         tok = expand_environ_var(tok);
139 
140         if (parse_redir_flow(c, tok))
141             continue;
142 
143         if (parse_environ_var(c, tok))
144             continue;
145 
146         c->argv[argc++] = tok;
147     }
148 
149     c->argv[argc] = (char*)NULL;
150     c->argc = argc;
151 
152     return (struct cmd*)c;
153 }
154 
155 // parses a command knowing that it contains
156 // the '&' char
157 static struct cmd* parse_back(char* buf_cmd) {
158 
159     int i = 0;
160     struct cmd* e;
161 
162     while (buf_cmd[i] != '&')
163         i++;
164 
165     buf_cmd[i] = END_STRING;
166 
167     e = parse_exec(buf_cmd);
168 
169     return back_cmd_create(e);
170 }
171 
172 // parses a command and checks if it contains
173 // the '&' (background process) character
```

```
174 static struct cmd* parse_cmd(char* buf_cmd) {
175
176     if (strlen(buf_cmd) == 0)
177         return NULL;
178
179     int idx;
180
181     // checks if the background symbol is after
182     // a redir symbol, in which case
183     // it does not have to run in in the 'back'
184     if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
185             buf_cmd[idx - 1] != '>')
186         return parse_back(buf_cmd);
187
188     return parse_exec(buf_cmd);
189 }
190
191 // parses the command line
192 // looking for the pipe character '|'
193 struct cmd* parse_line(char* buf) {
194
195     struct cmd *r, *l;
196
197     char* right = split_line(buf, '|');
198
199     l = parse_cmd(buf);
200     r = parse_cmd(right);
201
202     return pipe_cmd_create(l, r);
203 }
```

parsing.c

```
1 #ifndef PRINTSTATUS_H
2 #define PRINTSTATUS_H
3
4 #include "defs.h"
5 #include "types.h"
6
7 extern int status;
8
9 void print_status_info(struct cmd* cmd);
10
11 void print_back_info(struct cmd* back);
12
13 #endif // PRINTSTATUS_H
```

printstatus.h

```
1 #include "printstatus.h"
2
3 // prints information of process' status
4 void print_status_info(struct cmd* cmd) {
```

```
5
6       if (strlen(cmd->scmd) == 0
7           || cmd->type == PIPE)
8           return;
9
10      if (WIFEXITED(status)) {
11
12          fprintf(stdout, "%s Program: [%s] exited, status: %d %s\n",
13              COLOR_BLUE, cmd->scmd, WEXITSTATUS(status), COLOR_RESET);
14          status = WEXITSTATUS(status);
15
16      } else if (WIFSIGNALED(status)) {
17
18          fprintf(stdout, "%s Program: [%s] killed, status: %d %s\n",
19              COLOR_BLUE, cmd->scmd, -WTERMSIG(status), COLOR_RESET);
20          status = -WTERMSIG(status);
21
22      } else if (WTERMSIG(status)) {
23
24          fprintf(stdout, "%s Program: [%s] stopped, status: %d %s\n",
25              COLOR_BLUE, cmd->scmd, -WSTOPSIG(status), COLOR_RESET);
26          status = -WSTOPSIG(status);
27      }
28  }
29
30  // prints info when a background process is spawned
31  void print_back_info(struct cmd* back) {
32
33      fprintf(stdout, "%s  [PID=%d] %s\n",
34          COLOR_BLUE, back->pid, COLOR_RESET);
35  }
```

printstatus.c

```
1 #ifndef READLINE_H
2 #define READLINE_H
3
4 char* read_line(const char* promt);
5
6 #endif //READLINE_H
```

readline.h

```
1 #include "defs.h"
2 #include "readline.h"
3
4 static char buffer[BUFLEN];
5
6 // read a line from the standar input
7 // and prints the prompt
8 char* read_line(const char* promt) {
9
10      int i = 0,
```

```
11          c = 0;
12
13      fprintf(stdout, "%s %s %s\n", COLOR_RED, promt, COLOR_RESET);
14      fprintf(stdout, "%s", "$ ");
15
16      memset(buffer, 0, BUFLEN);
17
18      c = getchar();
19
20      while (c != END_LINE && c != EOF) {
21          buffer[i++] = c;
22          c = getchar();
23      }
24
25      // if the user press ctrl+D
26      // just exit normally
27      if (c == EOF)
28          return NULL;
29
30      buffer[i] = END_STRING;
31
32      return buffer;
33  }
```

readline.c

```
1 #ifndef RUNCMD_H
2 #define RUNCMD_H
3
4 #include "defs.h"
5 #include "parsing.h"
6 #include "exec.h"
7 #include "printstatus.h"
8 #include "freecmd.h"
9 #include "builtin.h"
10
11 int run_cmd(char* cmd);
12
13 #endif // RUNCMD_H
```

runcmd.h

```
1 #include "runcmd.h"
2
3 int status = 0;
4 struct cmd* parsed_pipe;
5
6 // runs the command in 'cmd'
7 int run_cmd(char* cmd) {
8
9     pid_t p;
10    struct cmd *parsed;
11
```

```
12      // if the "enter" key is pressed
13      // just print the promt again
14      if (cmd[0] == END_STRING)
15          return 0;
16
17      // cd built-in call
18      if (cd(cmd))
19          return 0;
20
21      // exit built-in call
22      if (exit_shell(cmd))
23          return EXIT_SHELL;
24
25      // pwd buil-in call
26      if (pwd(cmd))
27          return 0;
28
29      // parses the command line
30      parsed = parse_line(cmd);
31
32      // forks and run the command
33      if ((p = fork()) == 0) {
34
35          // keep a reference
36          // to the parsed pipe cmd
37          // so it can be freed later
38          if (parsed->type == PIPE)
39              parsed_pipe = parsed;
40
41          exec_cmd(parsed);
42      }
43
44      // store the pid of the process
45      parsed->pid = p;
46
47      // background process special treatment
48      // Hint:
49      // - check if the process is
50      //   going to be run in the 'back'
51      // - print info about it with
52      //   'print_back_info()'
53      //
54      // Your code here
55      print_back_info(parsed);
56      if(parsed->type != BACK){
57          // waits for the process to finish
58          waitpid(p, &status, 0);
59          print_status_info(parsed);
60      }
61
62      free_command(parsed);
63
```

```
64      return 0;
65  }
```

runcmd.c

```
1  #include "defs.h"
2  #include "types.h"
3  #include "readline.h"
4  #include "runcmd.h"
5
6  char promt[PRMTLEN] = {0};
7
8  // runs a shell command
9  static void run_shell() {
10
11      char* cmd;
12
13      while ((cmd = read_line(promt)) != NULL)
14          if (run_cmd(cmd) == EXIT_SHELL)
15              return;
16 }
17
18 // initialize the shell
19 // with the "HOME" directory
20 static void init_shell() {
21
22      char buf[BUFLEN] = {0};
23      char* home = getenv("HOME");
24
25      if (chdir(home) < 0) {
26          snprintf(buf, sizeof buf, "cannot cd to %s ", home);
27          perror(buf);
28      } else {
29          snprintf(promt, sizeof promt, "(%s)", home);
30      }
31 }
32
33 int main(void) {
34
35      init_shell();
36
37      run_shell();
38
39      return 0;
40 }
```

sh.c

```
1  #ifndef TYPES_H
2  #define TYPES_H
3
4  /* Commands definition types */
5
```

```
6  /*
7   cmd: Generic interface
8       that represents a single command.
9       All the other *cmd structs can be
10      casted to it, and they dont lose
11      information (for example the 'type' field).
12
13      - type: {EXEC, REDIR, BACK, PIPE}
14      - pid: the process id
15      - scmd: a string representing the command before being parsed
16  */
17  struct cmd {
18      int type;
19      pid_t pid;
20      char scmd[BUFLEN];
21  };
22
23  /*
24    execcmd: It contains all the relevant
25      information to execute a command.
26
27      - type: could be EXEC or REDIR
28      - argc: arguments quantity after parsed
29      - eargc: environ vars quantity after parsed
30      - argv: array of strings representig the arguments
31          of the form: {"binary/command", "arg0", "arg1", ... ,
    ↪ (char*)NULL}
32      - eargv: array of strings of the form: "KEY=VALUE"
33          representing the environ vars
34      - *_file: string that contains the name of the file
35          to be redirected to
36
37      IMPORTANT: an execcmd struct can have EXEC or REDIR type
38          depending on if the command to be executed
39          has at least one redirection symbol (<, >, >>, >&)
40  */
41  struct execcmd {
42      int type;
43      pid_t pid;
44      char scmd[BUFLEN];
45      int argc;
46      int eargc;
47      char* argv[MAXARGS];
48      char* eargv[MAXARGS];
49      char out_file[FNAMESIZE];
50      char in_file[FNAMESIZE];
51      char err_file[FNAMESIZE];
52  };
53
54  /*
55    pipecmd: It contains the same information as 'cmd'
56      plus two fields representing the left and right part
```

```
57      of a command of the form: "command1 arg1 arg2 | command2 arg3"
58      As they are of type 'struct cmd',
59      it means that they can be either an EXEC or a REDIR command.
60  */
61  struct pipecmd {
62      int type;
63      pid_t pid;
64      char scmd[BUFLEN];
65      struct cmd* leftcmd;
66      struct cmd* rightcmd;
67  };
68
69  /*
70    backcmd: It contains the same information as 'cmd'
71      plus one more field containing the command to be executed.
72      Take a look to the parsing.c file to understand it better.
73      Again, this extra field, can have type either EXEC or REDIR
74      depending on if the process to be executed in the background
75      contains redirection symbols.
76  */
77  struct backcmd {
78      int type;
79      pid_t pid;
80      char scmd[BUFLEN];
81      struct cmd* c;
82  };
83
84  #endif // TYPES_H
```

types.h

```
1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #include "defs.h"
5
6  char* split_line(char* buf, char splitter);
7
8  int block_contains(char* buf, char c);
9
10 #endif // UTILS_H
```

utils.h

```
1  #include "utils.h"
2
3  // splits a string line in two
4  // acording to the splitter character
5  char* split_line(char* buf, char splitter) {
6
7      int i = 0;
8
9      while (buf[i] != splitter &&
```

```
10          buf[i] != END_STRING)
11          i++;
12
13      buf[i++] = END_STRING;
14
15      while (buf[i] == SPACE)
16          i++;
17
18      return &buf[i];
19  }
20
21  // looks in a block for the 'c' character
22  // and returns the index in which it is, or -1
23  // in other case
24  int block_contains(char* buf, char c) {
25
26      for (int i = 0; i < strlen(buf); i++)
27          if (buf[i] == c)
28              return i;
29
30      return -1;
31  }
```

utils.c