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 Primera 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. Buscando en PATH

El objetivo de la siguiente implementación es poder ejecutar un programa o comando por consola, sin la necesidad de buscar la ruta completa del archivo ejecutable. Para eso se hace uso de la función *execvp*, la cual nos ofrece ejecutar un comando, buscando los archivos ejecutables cuyas rutas esten especificadas en la variable de entorno PATH. La misma requiere de dos parametros para funcionar correctamente, el nombre del programa que deseamos ejecutar *scmd* y un array de parametros que se desee que el programa utilice *argv*, obligatoriamente finalizando ese array con un *(char *)NULL*. Para esta seccion supondremos que no se pasaran parametros al programa. A continuación, se expone el codigo fuente implementado para esta funcionalidad.

```
1  //Execute de comand searching programs in PATH
2  //author: Martin Perez
3  void execute(struct cmd* cmd){
4
5      //Cast struct cmd* to struct execcmd* to use this interface
6      struct execcmd* execcmd = (struct execcmd*) cmd;
7      //Si le paso el execcmd->scmd me va a guardar el comando con
       ↪ parametros y todo
8      //Debo pasarle execcmd->argv[0]
9      //Example ls -l -a -> execcmd->scmd = "ls -l -a"; execcmd->argv
       ↪ = {"ls","-l","-a",(char*)NULL}
10     execvp(execcmd->argv[0],execcmd->argv);
11 }
12
13 // executes a command - does not return
14 //
15 // Hint:
16 // - check how the 'cmd' structs are defined
17 //   in types.h
18 void exec_cmd(struct cmd* cmd) {
19
20
21     switch (cmd->type) {
22
```

```
23          case EXEC:
24              // spawns a command
25              //
26              // Your code here
27              execute(cmd);
28              //printf("Commands are not yet implemented\n");
29              _exit(-1);
30              break;
31
32          case BACK: {
33              // runs a command in background
34              //
35              // Your code here
36              printf("Background process are not yet implemented\n");
37              _exit(-1);
38              break;
39          }
40
41          case REDIR: {
42              // changes the input/output/stderr flow
43              //
44              // Your code here
45              printf("Redirections are not yet implemented\n");
46              _exit(-1);
47              break;
48          }
49
50          case PIPE: {
51              // pipes two commands
52              //
53              // Your code here
54              printf("Pipes are not yet implemented\n");
55
56              // free the memory allocated
57              // for the pipe tree structure
58              free_command(parsed_pipe);
59
60              break;
61          }
62      }
63  }
```

execvp.c

La función *execute* es la encargada de llamar a *execvp* recibiendo como parametro la estrucutra *cmd* (Ver en el apendice el archivo types.h) ya habiendo sido parseado correctamente y utilizar sus atributos correspondientes.

Ejemplo:

```
1 Ejecutando comando ls
```

```
2
3  (/home/martin)
4 $ ls
5   [PID=4234]
6 Descargas    eclipse FIUBA  Imagenes  Plantillas  ticket.lib_files
7 Documentos  Escritorio  IDEs  Musica  Publico  Videos
8     Program: [ls] exited, status: 0
```

examplePath.txt

Pregunta: ¿cuáles son las diferencias entre la syscall *execve(2)* y la familia de wrappers proporcionados por la librería estándar de C *exec(3)*?

Respuesta: En primer lugar y la mas obvia diferencia, es que execve es una syscall, mientras que las funciones exec() son funciones que llaman que utilizan la syscall.

La familia de funciones exec() reemplaza la imagen del proceso actual, con una nueva imagen de proceso. Para estas familias, el primer argumento es el nombre del archivo que se va a ejecutar.

Mientras que execve() puede recibir el path de un archivo binario ejecutable o bien un script que debe empezar con el siguiente formato de linea:

interpreter (argumento)

Para ambos casos las funciones solo retornan -1 si un error ocurrio durante la ejecucion del programa.

1.2. Argumentos del programa

En esta sección consideramos que si podran pasarle parametros a nuestros programas ingresandolos por consola. Para evitar la repetición de código, revisar el propuesto en la implementación anterior. Esta vez, percatarse del casteo previo a utilizar la función *execvp*, detallado en *execute*. El objetivo de esto es poder utilizar el parametro *cmd* de estructura ya conocida, implementando la interfaz de *execcmd* (Ver en el apendice el archivo types.h). De esta manera se pueden acceder a otros atributos que no se podrian obtener con *cmd*, por ejemplo *argv* que contienen los parametros que utilizara el programa invocado.

Ejemplo:

```
1  Ejecutando ls con parametros
2
3  (/home/martin)
4 $ ls -l
5   [PID=2588]
6 total 92
7 drwxr-xr-x 4 martin martin  4096 abr 13 13:23 Descargas
8 drwxr-xr-x 2 martin martin  4096 sep  4  2017 Documentos
```

```
drwxrwxr-x 3 martin martin  4096 sep  5  2017 eclipse
drwxr-xr-x 4 martin martin  4096 abr 12 19:52 Escritorio
drwxrwxr-x 7 martin martin  4096 abr  1 00:05 FIUBA
drwxrwxr-x 3 martin martin  4096 sep  5  2017 IDEs
drwxr-xr-x 2 martin martin  4096 dic 13 00:41 Imagenes
drwxr-xr-x 2 martin martin  4096 sep  4  2017 Musica
drwxr-xr-x 2 martin martin  4096 sep  4  2017 Plantillas
drwxr-xr-x 2 martin martin  4096 sep  4  2017 Publico
-rw-rw-r-- 1 martin martin   433 abr 20 13:25 Shell1.aux
-rw-rw-r-- 1 martin martin 39692 abr 20 13:25 Shell1.log
-rw-rw-r-- 1 martin martin     0 abr 20 13:25 Shell1.toc
drwx------ 2 martin martin  4096 dic  9 16:47 ticket.lib_files
drwxr-xr-x 2 martin martin  4096 dic  9 17:30 Videos
    Program: [ls -l] exited, status: 0
```

exampleArgs.txt

1.3. Imprimir Variables de Entorno

Ya que ahora podemos invocar comandos desde consola, podemos hacer uso del comando *echo* para imprimir texto por consola. Lo siguiente a realizar será implementar las funciones del shell, para que al recibir un parametro seguido de reconozca que es una variable de entorno. Si esta se encuentra en el sistema se la imprime por salida estandar. A continuación se muestra un fragmento del codigo en el archivo *parsing.c*, observar la funcion que expande el nombre de la variable de entorno, por ejemplo *'JAVA_HOME'* con la ruta absoluta que representa esa variable *'/usr/lib/jvm/java-8-oracle/jre/'*.

Código Fuente

```
static char* expand_environ_var(char* arg) {

    //Your code here
    if(arg[0] == '$'){
        //length: size of arg - 1
        int length = strlen(arg);
        char *aux = malloc(length);
        for(int i = 1; i <= length; i++){
            aux[i-1] = arg[i];
        }
        char *env = getenv(aux);
        if(env != NULL){
            strcpy(arg,env);
        }
        free(aux);
    }

```

```
18     return arg;
19 }
```

getenv.c

Ejemplo:

```
1 Ejecutando echo
2
3  (/home/martin)
4 $ echo $JAVA_HOME
5   [PID=3653]
6 /usr/lib/jvm/java-8-oracle/jre/
7     Program: [echo $JAVA_HOME] exited, status: 0
```

exampleEnv.txt

2. Anexo

```c
#ifndef BUILTIN_H
#define BUILTIN_H

#include "defs.h"

extern char promt[PRMTLEN];

int cd(char* cmd);

int exit_shell(char* cmd);

int pwd(char* cmd);

#endif // BUILTIN_H
```

builtin.h

```c
#include "builtin.h"

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

    // Your code here

    return 0;
}

// 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

    return 0;
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {

    // Your code here

    return 0;
}
```

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 void exec_cmd(struct cmd* c);
12
13 #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 }
36
37 // opens the file in which the stdin/stdout or
38 // stderr flow will be redirected, and returns
39 // the file descriptor
40 //
41 // Find out what permissions it needs.
42 // Does it have to be closed after the execve(2) call?
43 //
44 // Hints:
45 // - if O_CREAT is used, add S_IWUSR and S_IRUSR
46 //  to make it a readable normal file
47 static int open_redir_fd(char* file) {
48
49     // Your code here
50     return -1;
51 }
52
53 //Execute de comand searching programs in PATH
54 //author: Martin Perez
55 void execute(struct cmd* cmd){
56
57     //Cast struct cmd* to struct execcmd* to use this interface
58     struct execcmd* execcmd = (struct execcmd*) cmd;
59     //Si le paso el execcmd->scmd me va a guardar el comando con
    ↪ parametros y todo
60     //Debo pasarle execcmd->argv[0]
61     //Example ls -l -a -> execcmd->scmd = "ls -l -a"; execcmd->argv
    ↪ = {"ls","-l","-a",(char*)NULL}
62     execvp(execcmd->argv[0],execcmd->argv);
63 }
64
65 // executes a command - does not return
66 //
67 // Hint:
68 // - check how the 'cmd' structs are defined
69 //  in types.h
70 void exec_cmd(struct cmd* cmd) {
71
72
73     switch (cmd->type) {
74
75         case EXEC:
76             // spawns a command
77             //
78             // Your code here
79             execute(cmd);
```

```
80              //printf("Commands are not yet implemented\n");
81              _exit(-1);
82              break;
83
84          case BACK: {
85              // runs a command in background
86              //
87              // Your code here
88              printf("Background process are not yet implemented\n");
89              _exit(-1);
90              break;
91          }
92
93          case REDIR: {
94              // changes the input/output/stderr flow
95              //
96              // Your code here
97              printf("Redirections are not yet implemented\n");
98              _exit(-1);
99              break;
100         }
101
102         case PIPE: {
103             // pipes two commands
104             //
105             // Your code here
106             printf("Pipes are not yet implemented\n");
107
108             // free the memory allocated
109             // for the pipe tree structure
110             free_command(parsed_pipe);
111
112             break;
113         }
114     }
115 }
```

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         }
110         free(aux);
111     }
112 
113     return arg;
114 }
115 
116 // parses one single command having into account:
117 // - the arguments passed to the program
118 // - stdin/stdout/stderr flow changes
119 // - environment variables (expand and set)
120 static struct cmd* parse_exec(char* buf_cmd) {
121 
122     struct execcmd* c;
123     char* tok;
124     int idx = 0, argc = 0;
125 
126     c = (struct execcmd*)exec_cmd_create(buf_cmd);
127 
128     while (buf_cmd[idx] != END_STRING) {
129 
130         tok = get_token(buf_cmd, idx);
131         idx = idx + strlen(tok);
132 
133         if (buf_cmd[idx] != END_STRING)
134             idx++;
135 
136         tok = expand_environ_var(tok);
137 
138         if (parse_redir_flow(c, tok))
139             continue;
140 
141         if (parse_environ_var(c, tok))
142             continue;
143 
144         c->argv[argc++] = tok;
145     }
146 
147     c->argv[argc] = (char*)NULL;
148     c->argc = argc;
149 
150     return (struct cmd*)c;
151 }
152 
153 // parses a command knowing that it contains
```

```
154 // the '&' char
155 static struct cmd* parse_back(char* buf_cmd) {
156
157     int i = 0;
158     struct cmd* e;
159
160     while (buf_cmd[i] != '&')
161         i++;
162
163     buf_cmd[i] = END_STRING;
164
165     e = parse_exec(buf_cmd);
166
167     return back_cmd_create(e);
168 }
169
170 // parses a command and checks if it contains
171 // the '&' (background process) character
172 static struct cmd* parse_cmd(char* buf_cmd) {
173
174     if (strlen(buf_cmd) == 0)
175         return NULL;
176
177     int idx;
178
179     // checks if the background symbol is after
180     // a redir symbol, in which case
181     // it does not have to run in in the 'back'
182     if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
183             buf_cmd[idx - 1] != '>')
184         return parse_back(buf_cmd);
185
186     return parse_exec(buf_cmd);
187 }
188
189 // parses the command line
190 // looking for the pipe character '|'
191 struct cmd* parse_line(char* buf) {
192
193     struct cmd *r, *l;
194
195     char* right = split_line(buf, '|');
196
197     l = parse_cmd(buf);
198     r = parse_cmd(right);
199
200     return pipe_cmd_create(l, r);
201 }
```

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
57      // waits for the process to finish
58      waitpid(p, &status, 0);
59
60      print_status_info(parsed);
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