75.08 Sistemas Operativos

Lab Shell - Parte 1

Nombre y Apellido: Damián Cassinotti
Padrón: 96618
Fecha de Entrega: 20/04/2018

Índice

1. Invocación de comandos

El Lab de Shell consiste en implementar el comportamiento básico de un shell teniendo ya un esqueleto predefinido. El comportamiento pedido para esta entrega será el encargado de ejecutar comando básicos utilizando parámetros y variables de entorno.

En cada caso veremos una breve explicación de lo pedido junto con el código de la solución. No mostraremos el código del shell completo, sino en cada parte nos centraremos en el necesario para cumplir con lo pedido. Se aclara que los ejercicios son iterativos e incrementales. Es decir, en cada ejercicio se suponen hechos los anteriores, contando con el código necesario.

Al finalizar las explicaciones de cada punto se incluirá el código completo de los archivos modificados, sin incluir el esqueleto entero.

1.1. Búsqueda en $PATH

El primer objetivo del lab consiste en poder ejecutar programas que se encuentren en el path de nuestra computadora. Esto es, poder ejecutarlos solamente llamandolos por su nombre y no por su ruta completa. Lo que hace el sistema es buscar en cada una de los directorios ubicados en la variable de entorno PATH para luego ejecutarlo.

Para esto se utilizó la funcion *execvpe*, de la familia de funciones *exec*. De toda la familia de funciones se eligió *execvpe* por varios motivos. Primero, para cumplir con los buscado en este punto, que es poder indicarle al sistema que busque el archivo deseado en el path. A diferencia de otras funciones de la familia, las funciones que en su numbre contienen la letra **p** son aquellas que reciben el nombre del archivo para buscar en el path. También pueden recibir la ruta absoluta.

Los otros dos motivos por los cuales se eligió esta función es por la posibilidad de poder pasarle argumentos y variables de entorno externas. Esto se analizará en profundidad más adelante.

A continuación se presenta el código que se agregó o modificó del esqueleto del shell. Recordar que en caso de archivos ya existentes, se mostrará solamente la sección modificada.

```
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 void spawn_command(struct execcmd* cmd);
14
15 #endif // EXEC_H
```

exec.h

```
void spawn_command(struct execcmd* cmd) {
    execvpe(cmd->argv[0], cmd->argv, cmd->eargv);
}
void exec_cmd(struct cmd* cmd) {

    switch (cmd->type) {

        case EXEC:
            // spawns a command
            spawn_command((struct execcmd*)cmd);
            break;
    }
}
```

exec.c

¿Cuáles son las diferencias entre la syscall execve(2) y la familia de wrappers proporcionados por la librería estándar de C (libc) exec(3)? Como lo indica la pregunta, la principal diferencia entre execve(2) y exec(3) es su naturaleza: el primero es una syscall, mientras que el segundo es una familia de funciones. Las funciones de la familia funcionan como wrappers de execve, es decir, internamente ejecutan la syscall. Este empaquetado permite la ejecución de binarios que se encuentren en el PATH.

1.2. Argumentos del programa

En este caso, el objetivo es que los ejecutables puedan recibir argumentos. Esto se logra pasandole los argumentos deseados a la función *execvpe* utilizada en el punto anterior. Este fue uno de los elementos por los cuales se decidió usar dicha función, ya que permite el pasaje de parámetros.

De esta forma, la función cumple con dos de los objetivos pedidos. No se mostrará código en esa sección ya que el implementado en el punto anterior alcanza para resolver rambos problemas.

1.3. Expansión de variables de entorno

El último de los objetivos de este lab es realizar la expansión de variables de entorno. Para esto se necesita poder reconocerlas y encontrar su verdadero valor.

Las variables de entorno a reemplazar son indicadas mediante el caracter $. De esta forma, lo que necesitamos es buscar cuáles de los argumentos de nuestro programa comienza con el caracter $ para luego reemplazazrlos por su verdadero valor. Para obtener el valor verdadero se utilizará la función *getenv(3)*, la cual dado el nombre de una variable de entorno devuelve su valor. Una vez obtenido dicho valor, se reemplazará en los argumentos del programa.

Nuestor esqueleto ya cuenta con una función dedicada a la expansión de variables de entorno, la cual recibe todos los parámetros de la llamada. Simplemente debemos encontrar cuáles de esos parámetros debemos expandir y reemplazar su valor con el de la variable.

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

    if (block_contains(arg, '$') == 0) {
        char* env = getenv(&arg[1]);
        strcpy(arg, env);
    }

    return arg;
}
```

parsing.c

2. Apéndice

En esta sección se mostrará el código completo de los archivo modificados del esqueleto. Los archivos estarán ordenados por orden alfabético.

```
#ifndef EXEC_H
#define EXEC_H

#include "defs.h"
#include "types.h"
#include "utils.h"
#include "freecmd.h"

extern struct cmd* parsed_pipe;

void exec_cmd(struct cmd* c);

void spawn_command(struct execcmd* cmd);

#endif // EXEC_H
```

exec.h

```
#include "exec.h"

// sets the "key" argument with the key part of
// the "arg" argument and null-terminates it
static void get_environ_key(char* arg, char* key) {

    int i;
    for (i = 0; arg[i] != '='; i++)
        key[i] = arg[i];

    key[i] = END_STRING;
}

// sets the "value" argument with the value part of
// the "arg" argument and null-terminates it
static void get_environ_value(char* arg, char* value, int idx) {

    int i, j;
    for (i = (idx + 1), j = 0; i < strlen(arg); i++, j++)
        value[j] = arg[i];

    value[j] = END_STRING;
}

// sets the environment variables passed
// 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 void spawn_command(struct execcmd* cmd) {
54     execvpe(cmd->argv[0], cmd->argv, cmd->eargv);
55 }
56
57 // executes a command - does not return
58 //
59 // Hint:
60 // - check how the 'cmd' structs are defined
61 //   in types.h
62 void exec_cmd(struct cmd* cmd) {
63
64     switch (cmd->type) {
65
66         case EXEC:
67             // spawns a command
68             spawn_command((struct execcmd*)cmd);
69             break;
70
71         case BACK: {
72             // runs a command in background
73             //
74             // Your code here
75             printf("Background process are not yet implemented\n");
76             _exit(-1);
77             break;
78         }
```

```
79
80          case REDIR: {
81              // changes the input/output/stderr flow
82              //
83              // Your code here
84              printf("Redirections are not yet implemented\n");
85              _exit(-1);
86              break;
87          }
88
89          case PIPE: {
90              // pipes two commands
91              //
92              // Your code here
93              printf("Pipes are not yet implemented\n");
94
95              // free the memory allocated
96              // for the pipe tree structure
97              free_command(parsed_pipe);
98
99              break;
100         }
101     }
102 }
```

exec.c

```
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      if (block_contains(arg, '$') == 0) {
99          char* env = getenv(&arg[1]);
100         strcpy(arg, env);
101     }
102
103     return arg;
104 }
105
106 // parses one single command having into account:
107 // - the arguments passed to the program
108 // - stdin/stdout/stderr flow changes
109 // - environment variables (expand and set)
110 static struct cmd* parse_exec(char* buf_cmd) {
111
112     struct execcmd* c;
113     char* tok;
114     int idx = 0, argc = 0;
115
116     c = (struct execcmd*)exec_cmd_create(buf_cmd);
117
118     while (buf_cmd[idx] != END_STRING) {
119
120         tok = get_token(buf_cmd, idx);
121         idx = idx + strlen(tok);
122
123         if (buf_cmd[idx] != END_STRING)
124             idx++;
125
126         tok = expand_environ_var(tok);
127
```

```
128         if (parse_redir_flow(c, tok))
129             continue;
130
131         if (parse_environ_var(c, tok))
132             continue;
133
134         c->argv[argc++] = tok;
135     }
136
137     c->argv[argc] = (char*)NULL;
138     c->argc = argc;
139
140     return (struct cmd*)c;
141 }
142
143 // parses a command knowing that it contains
144 // the '&' char
145 static struct cmd* parse_back(char* buf_cmd) {
146
147     int i = 0;
148     struct cmd* e;
149
150     while (buf_cmd[i] != '&')
151         i++;
152
153     buf_cmd[i] = END_STRING;
154
155     e = parse_exec(buf_cmd);
156
157     return back_cmd_create(e);
158 }
159
160 // parses a command and checks if it contains
161 // the '&' (background process) character
162 static struct cmd* parse_cmd(char* buf_cmd) {
163
164     if (strlen(buf_cmd) == 0)
165         return NULL;
166
167     int idx;
168
169     // checks if the background symbol is after
170     // a redir symbol, in which case
171     // it does not have to run in in the 'back'
172     if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
173             buf_cmd[idx - 1] != '>')
174         return parse_back(buf_cmd);
175
176     return parse_exec(buf_cmd);
177 }
178
179 // parses the command line
```

```
180 // looking for the pipe character '|'
181 struct cmd* parse_line(char* buf) {
182
183     struct cmd *r, *l;
184
185     char* right = split_line(buf, '|');
186
187     l = parse_cmd(buf);
188     r = parse_cmd(right);
189
190     return pipe_cmd_create(l, r);
191 }
```

parsing.c