

75.08 Sistemas Operativos

# Lab Shell - Challenges promocionales

**Nombre y Apellido:** Damián Cassinotti

**Padrón:** 96618

**Fecha de Entrega:** 29/06/2018

# Índice

<b>1. Challenges promocionales</b>	<b>2</b>
1.1. Pseudo-variables . . . . .	2
1.1.1. Entrada y Salida estándares a archivos . . . . .	2
1.1.2. Error estándar a archivo . . . . .	3
1.1.3. Combinar salida y errores . . . . .	4
1.1.4. Challenges . . . . .	4
1.2. Tuberías simples (pipes) . . . . .	6
<b>2. Apéndice</b>	<b>8</b>

# 1. Challenges promocionales

## 1.1. Pseudo-variables

El objetivo de este ejercicio es implementar la pseudo-variable `$?`. El propósito de esta variable es expandirse al estado de salida del último proceso ejecutado en primer plano.

Otras dos variables mágicas pueden ser:

- `$ ($$)`: Esta variable mágica se expande al pid del shell. En un subshell se expande al pid del shell padre, no al del subshell.
- `0 ($0)`: Se expande al nombre del shell. Esta variable se setea al inicializar el shell.

Para la resolución del ejercicio se utilizó la variable global `status`, y se modificó la función de expansión de variables:

```
extern int status;
```

../parsing.h

```
static char* expand_envIRON_var(char* arg) {  
    if (block_contains(arg, '$') == 0) {  
        if (!strcmp(&arg[1], "?")) {  
            char env[4] = {0};  
            sprintf(env, "%d", status);  
            strcpy(arg, env);  
        } else {  
            char *env = getenv(&arg[1]);  
            if (env == NULL)  
                env = "";  
            if (strlen(env) > ARGSize)  
                arg = realloc(arg, strlen(env));  
            strcpy(arg, env);  
        }  
    }  
    return arg;  
}
```

../parsing.c

## 1.2. Tuberías múltiples

El diseño que se eligió para la realización de tuberías múltiples es una forma recursiva de armar los comandos. En el esqueleto del shell se dividía el comando

por el caracter "|", y, en el caso de utilizarse pipes, se trataban los dos lados del pipe como comandos simples. De esta forma, al usar tuberías múltiples, el comando de la derecha es el que tendría los distintos pipes. Entonces lo que se decidió hacer es tratar al comando de la derecha como una línea nueva, de forma que no importe cuantos pipes existan, se generarán tantos comandos como sean necesarios.

```
struct cmd* parse_line(char* buf) {  
    if (strlen(buf) == 0)  
        return NULL;  
  
    struct cmd *r, *l;  
  
    char* right = split_line(buf, '|');  
  
    l = parse_cmd(buf);  
    r = parse_line(right);  
  
    return pipe_cmd_create(l, r);  
}
```

../parsing.c

### 1.3. Segundo plano avanzado

## 2. Apéndice

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

```
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 extern int status;
10
11 struct cmd* parse_line(char* b);
12
13 #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     if ((outIdx = block_contains(arg, '&')) >= 0 && arg[outIdx + 1]
26 ↪ == '>') {
27         strcpy(c->out_file, &arg[outIdx + 2]);
28         strcpy(c->err_file, &arg[outIdx + 2]);
29     }
30 }
```

```

28     free(arg);
29     c->type = REDIR;
30
31     return true;
32 }
33
34 // flow redirection for output
35 if ((outIdx = block_contains(arg, '>')) >= 0) {
36     switch (outIdx) {
37         // stdout redir
38         case 0: {
39             strcpy(c->out_file, arg + 1);
40             break;
41         }
42         // stderr redir
43         case 1: {
44             strcpy(c->err_file, &arg[outIdx + 1]);
45             break;
46         }
47     }
48 }
49
50 free(arg);
51 c->type = REDIR;
52
53 return true;
54 }
55
56 // flow redirection for input
57 if ((inIdx = block_contains(arg, '<')) >= 0) {
58     // stdin redir
59     strcpy(c->in_file, arg + 1);
60
61     c->type = REDIR;
62     free(arg);
63
64     return true;
65 }
66
67 return false;
68 }
69
70 // parses and sets a pair KEY=VALUE
71 // environment variable
72 static bool parse_envIRON_var(struct execcmd* c, char* arg) {
73
74     // sets environment variables apart from the
75     // ones defined in the global variable "environ"
76     if (block_contains(arg, '=') > 0) {
77
78         // checks if the KEY part of the pair
79         // does not contain a '-' char which means

```

```

80         // that it is not a environ var, but also
81         // an argument of the program to be executed
82         // (For example:
83         // ./prog -arg=value
84         // ./prog --arg=value
85         // )
86         if (block_contains(arg, '-') < 0) {
87             c->eargv[c->eargc++] = arg;
88             return true;
89         }
90     }
91
92     return false;
93 }
94
95 // this function will be called for every token, and it should
96 // expand environment variables. In other words, if the token
97 // happens to start with '$', the correct substitution with the
98 // environment value should be performed. Otherwise the same
99 // token is returned.
100 //
101 // Hints:
102 // - check if the first byte of the argument
103 //   contains the '$'
104 // - expand it and copy the value
105 //   to 'arg'
106 static char* expand_environ_var(char* arg) {
107
108     if (block_contains(arg, '$') == 0) {
109         if (!strcmp(&arg[1], "?")) {
110             char env[4] = {0};
111             sprintf(env, "%d", status);
112             strcpy(arg, env);
113         } else {
114             char *env = getenv(&arg[1]);
115             if (env == NULL)
116                 env = "";
117             if (strlen(env) > ARGSIZE)
118                 arg = realloc(arg, strlen(env));
119             strcpy(arg, env);
120         }
121     }
122
123     return arg;
124 }
125
126 // parses one single command having into account:
127 // - the arguments passed to the program
128 // - stdin/stdout/stderr flow changes
129 // - environment variables (expand and set)
130 static struct cmd* parse_exec(char* buf_cmd) {
131

```

```

132     struct execcmd* c;
133     char* tok;
134     int idx = 0, argc = 0;
135
136     c = (struct execcmd*)exec_cmd_create(buf_cmd);
137
138     while (buf_cmd[idx] != END_STRING) {
139
140         tok = get_token(buf_cmd, idx);
141         idx = idx + strlen(tok);
142
143         if (buf_cmd[idx] != END_STRING)
144             idx++;
145
146         if (parse_redir_flow(c, tok))
147             continue;
148
149         if (parse_envIRON_var(c, tok))
150             continue;
151
152         tok = expand_envIRON_var(tok);
153
154         c->argv[argc++] = tok;
155     }
156
157     c->argv[argc] = (char*)NULL;
158     c->argc = argc;
159
160     return (struct cmd*)c;
161 }
162
163 // parses a command knowing that it contains
164 // the '&' char
165 static struct cmd* parse_back(char* buf_cmd) {
166
167     int i = 0;
168     struct cmd* e;
169
170     while (buf_cmd[i] != '&')
171         i++;
172
173     buf_cmd[i] = END_STRING;
174
175     e = parse_exec(buf_cmd);
176
177     return back_cmd_create(e);
178 }
179
180 // parses a command and checks if it contains
181 // the '&' (background process) character
182 static struct cmd* parse_cmd(char* buf_cmd) {
183

```



```

184     if (strlen(buf_cmd) == 0)
185         return NULL;
186
187     int idx;
188
189     // checks if the background symbol is after
190     // a redir symbol, in which case
191     // it does not have to run in in the 'back'
192     if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
193         (buf_cmd[idx - 1] != '>' && buf_cmd[idx + 1] != '>'))
194         return parse_back(buf_cmd);
195
196     return parse_exec(buf_cmd);
197 }
198
199 // parses the command line
200 // looking for the pipe character '|'
201 struct cmd* parse_line(char* buf) {
202     if (strlen(buf) == 0)
203         return NULL;
204
205     struct cmd *r, *l;
206
207     char* right = split_line(buf, '|');
208
209     l = parse_cmd(buf);
210     r = parse_line(right);
211
212     return pipe_cmd_create(l, r);
213 }

```

../parsing.c