75.08 Sistemas Operativos

Lab Shell - Challenges promocionales

Nombre y Apellido: Damián Cassinotti
Padrón: 96618
Fecha de Entrega: 29/06/2018

Índice

1. Challenges promocionales

1.1. Pseudo-variables

El objetivo de este ejercicio es implementar la psudo-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 ele squeleto 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 linea 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

Para la resolución de este ejercicio se utilizaron las señales del kernel, en este caso la señal SIGCHLD. De esta forma, el kernel le avisa al proceso padre que el hijo terminó su ejecución. Esto es muy importante, como se explicará más adelante, para determinar la finalización de un proceso que corre en segundo plano.

Para realizar esta solución se implementó el manejador de señales, sobre el cual se deberá imprimir lo pedido, en caso de ser necesario. Para esto, primero se declaró una varible global *last_background_cmd*, la cual guardará el último comando de background ejecutado. Dicha variable se seteará al finalizar la ejecución del comando dentro de la función run_cmd:

```
    if (parsed->type == BACK) {
        print_back_info(parsed);
        last_background_cmd = malloc(sizeof(struct backcmd));
        memcpy(last_background_cmd, parsed, sizeof(struct backcmd));
        free_command(parsed);
        return 0;
    } else {
        waitpid(p, &status, 0);
    }
```

```
    print_back_finish_info();
```

../runcmd.c

Por otro lado, también se debe setear el manejador de la señal. Esto se realiza una única vez al iniciar la consola.

```
int main(void) {

    set_action_background();

    init_shell();

    run_shell();

    return 0;
}
```

../sh.c

En donde el seteo de la acción con su manejador se realiza de la siguiente manera:

```
#include "handlers.h"

char back_msg[BUFLEN];

void background_handler(int signum) {
    if (last_background_cmd == NULL)
        return;
    pid_t exited_pid = wait(NULL);
    if (exited_pid != last_background_cmd->pid)
        return;
    snprintf(back_msg, BUFLEN, "==> terminado: PID=%d (%s)",
    ↪ last_background_cmd->pid, last_background_cmd->scmd);
    free(last_background_cmd);
    last_background_cmd = NULL;
}

void set_action_background() {
    struct sigaction act;
    act.sa_handler = background_handler;
    act.sa_flags = SA_RESTART;
    sigaction(SIGCHLD, &act, NULL);
}
```

../handlers.c

En este archivo vemos que se creó una nueva variable global: *back_msg*. En dicha variable se guardará el mensaje a imprimir antes de imprimir el nuevo prompt. Esta variable se imprime, de ser necesario, utilizando la siguiente función:

```
void print_back_finish_info() {
    if (strlen(back_msg) > 0) {
        fprintf(stdout, "%s %s %s\n",
            COLOR_BLUE, back_msg, COLOR_RESET);
        back_msg[0] = END_STRING;
    }
}
```

../printstatus.c

De esta forma es como queda definido el manejo de las señales. Estas señales son necesarias ya que es la única forma de que el padre sepa que un hijo terminó su ejecución, ya que para correr los procesos en segundo plano no se utiliza el wait_pid. Pero, de todas formas, el sistema operativo le envía la señal por cada uno de los hijos que terminaron, sin importar si corrían en background o no (es decir, si el padre esperaba a que el hijo termine o no).

Para finalizar se realizan dos aclaraciones:

- Al finalizar el manejo de la señal se realiza simplemente un free del comando, en vez de un free_command. Esto se debe a que todos los atributos del comando de background que deben liberarse, se liberan al finalizar la ejecución del comando.

- La solución propuesta es válida solamente si se corre de a un proceso de background a la vez. Esto se debe a que solo se guarda la información del último que corrió.

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 EXEC_H
2 #define EXEC_H
3
4 #include "defs.h"
5 #include "types.h"
6 #include "freecmd.h"
7 #include <signal.h>
8
9 extern struct backcmd* last_background_cmd;
10
11 void background_handler(int signum);
12
13 void set_action_background();
14
15 #endif // EXEC_H
```

../handlers.h

```
1 #include "handlers.h"
2
3 char back_msg[BUFLEN];
4
5 void background_handler(int signum) {
6     if (last_background_cmd == NULL)
7         return;
8     pid_t exited_pid = wait(NULL);
9     if (exited_pid != last_background_cmd->pid)
10        return;
11    snprintf(back_msg, BUFLEN, "==> terminado: PID=%d (%s)",
      ↪ last_background_cmd->pid, last_background_cmd->scmd);
12    free(last_background_cmd);
13    last_background_cmd = NULL;
14 }
15
16 void set_action_background() {
17    struct sigaction act;
18    act.sa_handler = background_handler;
19    act.sa_flags = SA_RESTART;
20    sigaction(SIGCHLD, &act, NULL);
21 }
```

../handlers.c

```h
#ifndef PARSING_H
#define PARSING_H

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

extern int status;

struct cmd* parse_line(char* b);

#endif // PARSING_H
```

../parsing.h

```c
#include "parsing.h"

// parses an argument of the command stream input
static char* get_token(char* buf, int idx) {

    char* tok;
    int i;

    tok = (char*)calloc(ARGSIZE, sizeof(char));
    i = 0;

    while (buf[idx] != SPACE && buf[idx] != END_STRING) {
        tok[i] = buf[idx];
        i++; idx++;
    }

    return tok;
}

// parses and changes stdin/out/err if needed
static bool parse_redir_flow(struct execcmd* c, char* arg) {

    int inIdx, outIdx;

    if ((outIdx = block_contains(arg, '&')) >= 0 && arg[outIdx + 1]
    ↪ == '>') {
        strcpy(c->out_file, &arg[outIdx + 2]);
        strcpy(c->err_file, &arg[outIdx + 2]);

        free(arg);
        c->type = REDIR;

        return true;
    }
```

```
34
35      // flow redirection for output
36      if ((outIdx = block_contains(arg, '>')) >= 0) {
37          switch (outIdx) {
38              // stdout redir
39              case 0: {
40                  strcpy(c->out_file, arg + 1);
41                  break;
42              }
43              // stderr redir
44              case 1: {
45                  strcpy(c->err_file, &arg[outIdx + 1]);
46                  break;
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

```
1 #ifndef PRINTSTATUS_H
2 #define PRINTSTATUS_H
3 
4 #include "defs.h"
5 #include "types.h"
6 
7 extern int status;
8 
9 extern char back_msg[BUFLEN];
10 
11 void print_status_info(struct cmd* cmd);
12 
13 void print_back_info(struct cmd* back);
14 
15 void print_back_finish_info();
16 
17 #endif // PRINTSTATUS_H
```

../printstatus.h

```c
#include "printstatus.h"

// prints information of process' status
void print_status_info(struct cmd* cmd) {

    if (strlen(cmd->scmd) == 0
        || cmd->type == PIPE)
        return;

    if (WIFEXITED(status)) {

        fprintf(stdout, "%s Program: [%s] exited, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, WEXITSTATUS(status), COLOR_RESET);
        status = WEXITSTATUS(status);

    } else if (WIFSIGNALED(status)) {

        fprintf(stdout, "%s Program: [%s] killed, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, -WTERMSIG(status), COLOR_RESET);
        status = -WTERMSIG(status);

    } else if (WTERMSIG(status)) {

        fprintf(stdout, "%s Program: [%s] stopped, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, -WSTOPSIG(status), COLOR_RESET);
        status = -WSTOPSIG(status);
    }
}

// prints info when a background process is spawned
void print_back_info(struct cmd* back) {

    fprintf(stdout, "%s  [PID=%d] %s\n",
        COLOR_BLUE, back->pid, COLOR_RESET);
}

void print_back_finish_info() {
    if (strlen(back_msg) > 0) {
        fprintf(stdout, "%s %s %s\n",
            COLOR_BLUE, back_msg, COLOR_RESET);
        back_msg[0] = END_STRING;
    }
}
```

../printstatus.c

```c
#include "defs.h"
#include "types.h"
#include "readline.h"
#include "runcmd.h"
#include "handlers.h"

char promt[PRMTLEN] = {0};
```

```
 8
 9 // runs a shell command
10 static void run_shell() {
11
12     char* cmd;
13
14     while ((cmd = read_line(promt)) != NULL)
15         if (run_cmd(cmd) == EXIT_SHELL)
16             return;
17 }
18
19 // initialize the shell
20 // with the "HOME" directory
21 static void init_shell() {
22
23     char buf[BUFLEN] = {0};
24     char* home = getenv("HOME");
25
26     if (chdir(home) < 0) {
27         snprintf(buf, sizeof buf, "cannot cd to %s ", home);
28         perror(buf);
29     } else {
30         snprintf(promt, sizeof promt, "(%s)", home);
31     }
32 }
33
34 int main(void) {
35
36     set_action_background();
37
38     init_shell();
39
40     run_shell();
41
42     return 0;
43 }
```

../sh.c