75.08 Sistemas Operativos

Lab Shell - Parte 2

Nombre y Apellido: Damián Cassinotti
Padrón: 96618
Fecha de Entrega: 04/05/2018

Índice

1. Invocación avanzada

En esta segunda parte del lab veremos opciones avanzadas en la invicación de comandos. Para eso nos basaremos en el esqueleto provisto y la resolución de la parte anterior.

Para esta parte se codificarán soluciones para la utilización de comandos built-in, la adición de variables de entorno y la ejecución de procesos en segundo plano.

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. Comandos built-in

El primer objetivo del lab consiste en la ejecución de comandos built-in dentro de nuestra shell. Estos comandos son aquellos que no se pueden ejecutar en un proceso separado pues carecerían de sentido. En nuestro ejemplo se programarán los comandos *cd, exit y pwd*.

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.

```
int run_cmd(char* cmd) {
    if (cd(cmd)) {
        exec_cd(cmd);
        return 0;
    }

    // exit built-in call
    if (exit_shell(cmd))
        exit(EXIT_SUCCESS);

    // pwd buil-in call
    if (pwd(cmd)) {
        exec_pwd();
        return 0;
    }
```

```
}
```

../runcmd.c

```
#include "builtin.h"

static int starts_with(char* original_str, char* compare_str) {
    if (strncmp(original_str, compare_str, strlen(compare_str)) == 0)
        return 1;
    return 0;
}

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {
    return starts_with(cmd, "exit");
}

// 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) {
    return starts_with(cmd, "cd");
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {
    return starts_with(cmd, "pwd");
}
```

../builtin.c

```
void exec_cd(char* cmd) {
    char* nwd = split_line(cmd, ' ');
    if (chdir(nwd) != 0) {
        char buf[BUFLEN] = {0};
        snprintf(buf, sizeof buf, "cannot cd to %s", nwd);
        perror(buf);
    }
}

void exec_pwd() {
    char* cwd = getcwd(NULL, 0);
    printf("%s\n", cwd);
```

```
    free(cwd);
}
```

../exec.c

¿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? El comando pwd podría implementarse en un proceso nuevo. El motivo de hacerlo como built-in es que al realizar un fork no solo se crea un nuevo proceso, sino que se realizan distintas acciones como copia de memoria, etc. Para evitar todo esto, ya que pwd se trata solo de imprimir por pantalla la ruta actual, se decide hacerlo built-in para que sea más eficiente.
Por otro lado, cd es obligatorio hacerlo como built-in ya que sino no tendría sentido, pues cambiaría el working directory del proceso hijo (que luego de esto, moriría) y no del proceso padre (o sea, la shell).

1.2. Variables de entorno adicionales

Este objetivo del lab consta de incluir variables de entorno adicionales, específicas para el proceso hijo. Para esto, luego del fork, se setean cada una de las variables de entorno que se reciben en el comando mediante la invocación a la función setenv.

A continuación se muestra el código de la solución:

```
static void set_environ_vars(char** eargv, int eargc) {
    for (int i = 0; i < eargc; i++) {
        char key[ARGSIZE] = {0};
        char value[ARGSIZE] = {0};
        get_environ_key(eargv[i], key);
        get_environ_value(eargv[i], value, strlen(key));
        setenv(key, value, 1);
    }
}
void spawn_command(struct execcmd* cmd) {
    set_environ_vars(cmd->eargv, cmd->eargc);
    execvp(cmd->argv[0], cmd->argv);
}
```

../exec.c

¿por qué es necesario hacer las llamadas a setenv(3) luego de la llamada a fork(2)? Esto es necesario ya que las nuevas variables de entorno se deben agregar solamente en el proceso hijo, pues solo deben existir para la ejecución del proceso que se está llamando. En caso de hacerlo antes del fork, las nuevas variables estarán seteadas para el proceso padre (la shell) y perdurarán en el tiempo.

Supongamos, que en vez de utilizar setenv(3) por cada una de las variables, se guardan en un array y se lo coloca en el tercer argumento

de una de las funciones de exec(3). ¿El comportamiento es el mismo que en el primer caso? Explicar qué sucede y por qué. El comportamiento no será el mismo. Las funciones de la familia exec(3) que permiten el pasaje de variables de entorno tomarán como válidas sólo las variables que se encuentren en el array pasado. Es decir, si en la llamada se le pasan dos variables de entorno que quisimos agregar, esas serán todas las variables disonibles en la ejecución. Análogamente, si el array se encuentra vació, no existinrán variables de entorno válidas.
Para emular el comportamiento deseado, se le debe pasar el array de cadenas que encontramos en la variable *environ* que se encuentra definida en *unistd.h*. Dicha variable posee todas las variables de entorno disponibles. Entonces, para poder agregar variables de entorno mediante la función exec(3) debemos obtener el array de todas las variables, agregarle las adicionales que queramos, y recién ahí lo podremos pasar como parámetro.

1.3. Procesos en segundo plano

El último de los objetivos de este lab es realizar una llamada a procesos que se ejecuten en segundo plano.

Para esta resolución se tuvo en cuenta, luego de hacer el fork, qué tipo de ejecucón se estaba realizando. En todos los casos se crea un proceso hijo sobre el cual se ejecutará el comando deseado. La diferencia radica en el proceso padre: si la ejecución es por background, no se espera que el proceso hijo termine de realizar su ejecución. Caso contrario, se ejecuta la función *waitpid*.

A continuación veremos el código que se utilizó para esta solución:

```
int run_cmd(char* cmd) {
    if (parsed->type == BACK)
        print_back_info(parsed);
    else
        waitpid(p, &status, 0);
}
```

../runcmd.c

```
void exec_cmd(struct cmd* cmd) {
        case BACK: {
            spawn_background_command((struct backcmd*)cmd);
            break;
        }
}
```

../exec.c

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 BUILTIN_H
2 #define BUILTIN_H
3
4 #include "defs.h"
5 #include "parsing.h"
6
7 extern char promt[PRMTLEN];
8
9 int cd(char* cmd);
10
11 int exit_shell(char* cmd);
12
13 int pwd(char* cmd);
14
15 #endif // BUILTIN_H
```

../builtin.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     for (int i = 0; i < eargc; i++) {
34         char key[ARGSIZE] = {0};
35         char value[ARGSIZE] = {0};
36         get_environ_key(eargv[i], key);
37         get_environ_value(eargv[i], value, strlen(key));
38         setenv(key, value, 1);
39     }
40 }
41
42 // opens the file in which the stdin/stdout or
43 // stderr flow will be redirected, and returns
44 // the file descriptor
45 //
46 // Find out what permissions it needs.
47 // Does it have to be closed after the execve(2) call?
48 //
49 // Hints:
50 // - if O_CREAT is used, add S_IWUSR and S_IRUSR
51 //   to make it a readable normal file
52 static int open_redir_fd(char* file) {
53
54     // Your code here
55     return -1;
56 }
57
58 void spawn_command(struct execcmd* cmd) {
59     set_environ_vars(cmd->eargv, cmd->eargc);
60     execvp(cmd->argv[0], cmd->argv);
61 }
62
63 void spawn_background_command(struct backcmd* cmd) {
64     exec_cmd(cmd->c);
65 }
66
67 // executes a command - does not return
68 //
69 // Hint:
70 // - check how the 'cmd' structs are defined
71 //   in types.h
72 void exec_cmd(struct cmd* cmd) {
73
74     switch (cmd->type) {
75
76         case EXEC:
77             spawn_command((struct execcmd*)cmd);
78             break;
```

```
79
80          case BACK: {
81              spawn_background_command((struct backcmd*)cmd);
82              break;
83          }
84
85          case REDIR: {
86              // changes the input/output/stderr flow
87              //
88              // Your code here
89              printf("Redirections are not yet implemented\n");
90              _exit(-1);
91              break;
92          }
93
94          case PIPE: {
95              // pipes two commands
96              //
97              // Your code here
98              printf("Pipes are not yet implemented\n");
99
100             // free the memory allocated
101             // for the pipe tree structure
102             free_command(parsed_pipe);
103
104             break;
105         }
106     }
107 }
108
109 void exec_cd(char* cmd) {
110     char* nwd = split_line(cmd, ' ');
111     if (chdir(nwd) != 0) {
112         char buf[BUFLEN] = {0};
113         snprintf(buf, sizeof buf, "cannot cd to %s", nwd);
114         perror(buf);
115     }
116 }
117
118 void exec_pwd() {
119     char* cwd = getcwd(NULL, 0);
120     printf("%s\n", cwd);
121     free(cwd);
122 }
```

../exec.c

```
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         exec_cd(cmd);
20         return 0;
21     }
22
23     // exit built-in call
24     if (exit_shell(cmd))
25         exit(EXIT_SUCCESS);
26
27     // pwd buil-in call
28     if (pwd(cmd)) {
29         exec_pwd();
30         return 0;
31     }
32
33     // parses the command line
34     parsed = parse_line(cmd);
35
36     // forks and run the command
37     if ((p = fork()) == 0) {
38
39         // keep a reference
40         // to the parsed pipe cmd
41         // so it can be freed later
42         if (parsed->type == PIPE)
43             parsed_pipe = parsed;
44
45         exec_cmd(parsed);
46     }
47
48     // store the pid of the process
49     parsed->pid = p;
50
51     // background process special treatment
52     if (parsed->type == BACK)
53         print_back_info(parsed);
54     else
55         waitpid(p, &status, 0);
```

```
56
57      print_status_info(parsed);
58
59      free_command(parsed);
60
61      return 0;
62  }
```

../runcmd.c