Sistemas Operativos (75.08): Lab Shell

Matias Rozanec (#97404)
rozanecm@gmail.com

Abril 2018

Índice

Parte I
Resolución

1. Parte 1

Para resolver la primera parte de este Lab, se alteraron las líneas correspondientes al caso de ejecución de comandos en el switch de la función `void exec_cmd(struct cmd* cmd)` del archivo `exec.c` (líneas 6 a 16 del fragmento de código recuadrado).

Para lograr invocar programas y permitir pasarles argumentos, lo primero que se hizo fue castear el `struct cmd* cmd` a `(struct execcmd*) full_cmd`; de esta forma logramos tener acceso de forma directa a todos los datos necesarios del comando recibido. Por último, invocamos `execvp` pasándole como primer argumento `argv[0]`, o sea el nombre del programa a ejecutar, y como segundo argumento `argv`, o sea toda la lista de argumentos.

Para poder expandir variables de entorno, lo que primero que se hizo fue trabajar sobre la detección de las mismas. Esto se implementó leyendo el primer caracter de cada uno de los argumentos. En caso de que el caracter coincida con `$`, se trata de una variable de entorno, por lo que se procede a su expansión mediante `getenv()`. A dicha función hay que pasarle el argumento correspondiente pero sin el caracter `$`.

Listing 1: exec.c

```
1  void exec_cmd(struct cmd* cmd) {
2
3      switch (cmd->type) {
4
5          case EXEC: {
6              // spawns a command
7              struct execcmd* full_cmd = (struct execcmd*)cmd;
8              execvp(full_cmd->argv[0], full_cmd->argv);
9              break;
10         }
11
12         case BACK: {
13             // runs a command in background
14             //
15             // Your code here
16             printf("Background process are not yet implemented\n");
17             _exit(-1);
18             break;
19         }
20
21         case REDIR: {
22             // changes the input/output/stderr flow
23             //
24             // Your code here
25             printf("Redirections are not yet implemented\n");
26             _exit(-1);
27             break;
28         }
29
30         case PIPE: {
31             // pipes two commands
32             //
33             // Your code here
34             printf("Pipes are not yet implemented\n");
35
36             // free the memory allocated
37             // for the pipe tree structure
38             free_command(parsed_pipe);
39
40             break;
41         }
42     }
43 }
```

Listing 2: parsing.c

```
// this function will be called for every token, and it should
// expand environment variables. In other words, if the token
// happens to start with '$', the correct substitution with the
// environment value should be performed. Otherwise the same
// token is returned.
//
// Hints:
// - check if the first byte of the argument
//      contains the '$'
// - expand it and copy the value
//       to 'arg'
static char* expand_environ_var(char* arg) {
    if(arg[0] == '$'){
        char* temp = getenv(arg+1);
        char* temp_argv = realloc(arg, strlen(temp));
        if(temp_argv == NULL){
            perror("error in realloc.\n");
        }

        arg = temp_argv;
        strcpy(arg, temp);
    }
    return arg;
}
```

2. Parte 2

2.1. Comandos *built-in*

Pregunta: ¿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? (para esta última pregunta pensar en los built-in como true y false)

Respuesta: Ambos comandos podrían ser implementados sin necesidad de ser built-in. La razón por la que no se opta por esa opción es que son comandos muy simples que no requieren el tratamiento más comlejo que otros comandos sí necesitan, y por ende se espera una ejecución rápida sin procesamiento innecesario.

Listing 3: builtin.c

```
#include "builtin.h"
#include "utils.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {
    if(strcmp(cmd, "exit") == 0)
        exit(0);

        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) {
    /* lets assume no command name will take more than 100 bytes */
    char cmd_name[100];
    sscanf(cmd, "%s", cmd_name);

    if(strcmp(cmd_name, "cd") == 0) {
        char* args = split_line(cmd, SPACE);

        if (strlen(args) > 0) {
            chdir(args);
        } else {
            chdir(getenv("HOME"));
        }

        char* current_path = get_current_dir_name();
        snprintf(promt, sizeof promt, "(%s)", current_path);
        free(current_path);
        return 1;
    }

        return 0;
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {
    if(strcmp(cmd, "pwd") == 0){
        char* current_path = get_current_dir_name();
        printf("%s\n", current_path);
        return 1;
    }

        return 0;
}
```

2.2. Variables de entorno adicionales

Pregunta: ¿por qué es necesario hacerlo luego de la llamada a `fork(2)`?

Es necesario hacerlo luego de la llamada a `fork(2)` porque de esa forma la incorporación de las nuevas variables sucede en el proceso hijo, que es lo que nos interesa.

Responder (opcional):

- *¿el comportamiento es el mismo que en el primer caso? Explicar qué sucede y por qué.*
  El comportamiento no será el mismo, ya que en el primer caso la variable se agrega al conjunto de variables de entorno, y luego el comando a ejecutar se busca en todo el entorno. En el segundo caso, en cambio, se busca el comando únicamente en el entorno descripto por el array que se le pasa como tercer argumento.
- *Describir brevemente una posible implementación para que el comportamiento sea el mismo.*
  Una posible implementación sería armar un array con todo el env default, agregándole nuestras nuevas variables, y pasar eso como tercer argumento.

Listing 4: exec.c: funcion set_environ_vars

```
// sets the environment variables passed
// in the command line
//
// Hints:
// - use 'block_contains()' to
//  get the index where the '=' is
// - 'get_environ_*()' can be useful here
static void set_environ_vars(char** eargv, int eargc) {
    for(int i = 0; i < eargc; i++){
        int idx = block_contains(eargv[i], '=');
        char* value = &eargv[i][idx+1];
        eargv[i][idx] = '\0';

        if((setenv(eargv[i], value, 0)) == -1){
            perror("Error when setting env: ");
        }
    }
}
```

2.3. Procesos en segundo plano

Detallar cúal es el mecanismo utilizado.
En caso de detectar un proceso que deba ser corrido en *background*, no se espera a que el hijo termine. Nótese que se diversifica el print de estado de acuerdo al tipo de proceso que se está corriendo. El free del comando se hace en cualquier caso, de lo contrario habrá leaks. de memoria.

Listing 5: runcmd.c: últimas líneas

```
    // store the pid of the process
    parsed->pid = p;

    // background process special treatment
    // Hint:
    // - check if the process is
    //  going to be run in the 'back'
    // - print info about it with
    //  'print_back_info()'
    //
    // Your code here
    if(parsed->type != BACK){
        // waits for the process to finish
        waitpid(p, &status, 0);
        print_status_info(parsed);
    }else{
        print_back_info(parsed);
    }

    free_command(parsed);

    return 0;
}
```

2.4. Pregunta de parte 1

Respondo a continuación la pregunta de la parte 1 que para la primer entrega pasé por alto.

Pregunta: ¿cuáles son las diferencias entre la syscall execve(2) y la familia de wrappers proporcionados por la librería estándar de C (libc) exec(3)?
La syscall recibe como argumentos el nombre de programa a ejecutar, un array de argumentos que se le pasarán al nuevo programa, y array especificando el ambiente del programa. La familia de wrappers llama internamente a la syscall, pero hacen que sea más simple el manejo de la misma, contemplando los diversos escenarios posibles.