Sistemas Operativos (75.08): Lab Shell

Matias Rozanec (#97404)
rozanecm@gmail.com

Abril 2018

Índice

Parte I
Resolución

1. Parte 1: Invocación de comandos

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: Invocación avanzada

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

```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.

3. Parte 3: Redirecciones

3.1. Flujo estándar

El esqueleto dado se encarga en su etapa de parsing de almacenar los nombres de archivo a los cuales habría que redireccionar el flujo de salida estándard y error, o de los cuales habría que tomar la entrada estándard. Gracias a eso, lo único que restaba hacer es detectar si se trataba de un comando de tipo REDIR, en cuyo caso se crea el archivo correspondiente (en caso de redireccionar salidas) o se abre el archivo solicitado para lectura. En cualquiera de los casos, se le cambia el file descriptor a estos archivos con los de las entradas y salidas estandar, mediante `dup2()`.

En el caso de la redirección `2>&1`, se chequea si el archivo al que se quiere redireccionar la salida de error tiene 'nombre' == '`&1`', en cuyo caso se redirecciona la salida de error a la salida std.

Investigar el significado de este tipo de redirección y explicar qué sucede con la salida de cat out.txt. Comparar con los resultados obtenidos anteriormente.
`2>&1` redirecciona la salida de error a la salida std. `>out.txt` redirecciona la salida std al archivo *out.txt*. Por lo tanto, todo lo que saldría por la salida std y por la salida de error, se guardará en el archivo *out.txt*.

Listing 6: exec.c: líneas correspondientes a manejo de redirecciones.

```
1         case REDIR: {
2             // changes the input/output/stderr flow
3             int in_fd, out_fd, err_fd;
4             if(strlen(full_cmd->in_file)){
5                 in_fd = open(full_cmd->in_file, O_RDONLY);
6                 if(in_fd == -1)
7                     perror("error changing fd");
8                 if(dup2(in_fd, STDIN_FILENO) == -1)
9                     perror("error in dup2");
10            }
11
12            if(strlen(full_cmd->out_file)){
13                out_fd = open(full_cmd->out_file, O_CREAT | O_WRONLY,
14                        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
15                if(out_fd == -1)
16                    perror("error changing fd");
17                if(dup2(out_fd, STDOUT_FILENO) == -1)
18                    perror("error in dup2");
19                close(out_fd);
20            }
21
22            if(strlen(full_cmd->err_file)){
23                if(strcmp(full_cmd->err_file, "&1") == 0){
24                    err_fd = 1;
25                }else{
26                    err_fd = open(full_cmd->err_file, O_CREAT | O_WRONLY,
27                            S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
28                }
29                if(err_fd == -1)
30                    perror("error changing out fd");
31                if(dup2(err_fd, STDERR_FILENO) == -1)
32                    perror("erro in dup2");
33            }
34
35            if((execvp(full_cmd->argv[0], full_cmd->argv)) == -1){
36                perror("Error with execvp: ");
37            }
38
39            _exit(-1);
40            break;
41        }
```

Challenge: *investigar, describir y agregar la funcionalidad del operador de redirección >> y &>*

El operador >> escribe en el archivo cuyo nombre le sucede a dicho operador, pero en lugar de crear un archivo nuevo y escribir desde el principio, en caso de que dicho archivo ya exista, lo

abre en modo *append*.

El operador `&>` redirige las salidas std y de error al archivo cuyo nombre le sucede al comando. En el código a continuación, se puede ver que en el caso de haber redireccionamiento de output, se chequea si se usó el operador `>>`, en cuyo caso se abre el archivo en modo append.

Listing 7: exec.c: líneas correspondientes a manejo de redirecciones.

```
        case REDIR: {
            // changes the input/output/stderr flow
            int in_fd, out_fd, err_fd;
            /* IN redir */
            if(strlen(full_cmd->in_file)){
                in_fd = open(full_cmd->in_file, O_RDONLY);
                if(in_fd == -1)
                    perror("error changing fd");
                if(dup2(in_fd, STDIN_FILENO) == -1)
                    perror("error in dup2");
            }

            /* OUT redir */
            if(strlen(full_cmd->out_file)){
                if(full_cmd->out_file[0] == '>'){
                    out_fd = open(full_cmd->out_file + 1,
                            O_WRONLY|O_CREAT|O_APPEND, 0666);
                }else{
                    out_fd = open(full_cmd->out_file, O_CREAT | O_WRONLY,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
                }
                if(out_fd == -1)
                    perror("error changing fd");
                if(dup2(out_fd, STDOUT_FILENO) == -1)
                    perror("error in dup2");
                close(out_fd);
            }

            /* ERR redir */
            if(strlen(full_cmd->err_file)){
                if(strcmp(full_cmd->err_file, "&1") == 0){
                    err_fd = 1;
                }else{
                    err_fd = open(full_cmd->err_file, O_CREAT | O_WRONLY,
                            S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
                }
                if(err_fd == -1)
                    perror("error changing out fd");
                if(dup2(err_fd, STDERR_FILENO) == -1)
                    perror("erro in dup2");
            }

            if((execvp(full_cmd->argv[0], full_cmd->argv)) == -1){
                perror("Error with execvp: ");
            }

            _exit(-1);
            break;
        }
```

Para la implementación del comando `&>` hubo que corregir primero el manejo de los comandos BACK:

Listing 8: exec.c: líneas corrgidas del manejo de comandos BACK.

```
        case BACK: {
            // runs a command in background
            struct backcmd* back_cmd = (struct backcmd*)cmd;
            exec_cmd(back_cmd->c);
            _exit(-1);
            break;
        }
```

Luego, en el archivo *parsing.c* se corrigió el código para que no se entienda como un comando BACK en caso de encontrar la secuencia `&>`.

Listing 9: parsing.c: se agrega caso de excepción, evitando malinterpretar comandos como BACK.

```
// parses a command and checks if it contains
// the '&' (background process) character
static struct cmd* parse_cmd(char* buf_cmd) {

    if (strlen(buf_cmd) == 0)
        return NULL;

    int idx;

    // checks if the background symbol is after
    // a redir symbol, in which case
    // it does not have to run in in the 'back'
    if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
            buf_cmd[idx - 1] != '>' &&
            buf_cmd[idx + 1] != '>')
        return parse_back(buf_cmd);

    return parse_exec(buf_cmd);
}
```

Por último, en `parse_redir_flow()` de *parsing.c* se consideró el caso de que se quiera redireccionar las salidas std y de error a un archivo (adentro del `case: 1`).

Listing 10: parsing.c: de detectrse la secuencia &>, se guarda el nombre de archivo tanto para la salida std como de error.

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

    int inIdx, outIdx;

    // flow redirection for output
    if ((outIdx = block_contains(arg, '>')) >= 0) {
        switch (outIdx) {
            // stdout redir
            case 0: {
                strcpy(c->out_file, arg + 1);
                break;
            }
            // stderr redir
            case 1: {
                if((block_contains(arg, '&')) == 0)
                    strcpy(c->out_file, &arg[outIdx + 1]);
                strcpy(c->err_file, &arg[outIdx + 1]);
                break;
            }
        }

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

        return true;
    }

    // flow redirection for input
    if ((inIdx = block_contains(arg, '<')) >= 0) {
        // stdin redir
        strcpy(c->in_file, arg + 1);

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

        return true;
    }

    return false;
}
```

3.2. Tuberías simples (pipes)

En este caso se completó el código correspondiente al caso de PIPE en la función *exec_cmd* del archivo *exec.c*. En primer lugar se llama a `pipe()` para obtener dos file descriptors. A continuación,

se hace un fork: el proceso hijo redirecciona el stdout al segundo file descriptor dado por `pipe()`, y luego llama a *exec_cmd()* con el comando de la izquierda del pipe. El roceso padre redirecciona el stdin al primer file descriptor dado por `pipe()` y llama a *exec_cmd* con el comando de la derecha del pipe.

Listing 11: exec.c: PIPE case

```
        case PIPE: {
            // pipes two commands
            struct pipecmd* pipe_cmd = (struct pipecmd*) cmd;
            int pipefd[2];
            if(pipe(pipefd) == -1)
               perror("error with pipe(2)");

            pid_t cpid = fork();
            if (cpid == -1) {
                perror("fork");
                exit(EXIT_FAILURE);
            }

            if (cpid == 0) {
                close(pipefd[0]);

                if((dup2(pipefd[1], STDOUT_FILENO)) == -1)
                    perror("error on dup2");

                exec_cmd(pipe_cmd->leftcmd);

            } else {
                close(pipefd[1]);

                if((dup2(pipefd[0], STDIN_FILENO)) == -1)
                    perror("error on dup2");

                exec_cmd(pipe_cmd->rightcmd);
            }

            // free the memory allocated
            // for the pipe tree structure
            free_command(parsed_pipe);

            break;
        }
    }
}
```