75.08 Sistemas Operativos

Lab Shell - Parte 3

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

Índice

1. Redirecciones **2**
1.1. Flujo estándar 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 3
1.1.4. Challenges 4
1.2. Variables de entorno adicionales 6

2. Apéndice **7**

1. Redirecciones

La tercera parte del shell consiste en implementar las distintas redirecciones que podemos encontrar en unix. Estas son las redirecciones a archivos y entre procesos mediante pipes.

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. Flujo estándar

Las redirecciones de flujo estándar consiste en redirigir tanto la entrada y salida estándar como la salida estándar de errores a distintos archivos. Esto permite analizar la salida de un programa, o también automatizar la entrada. Veremos a continuación las tres formas de redirección de flujo estándar.

1.1.1. Entrada y Salida estándares a archivos

Esta primera forma es la clásica forma de redirección de entrada y salida estándar. Esta redirección consiste en utilizar los operadores <y >, seguidos del nombre del archivo, para entrada y salida estándar respectivamente.

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.

```
static int open_redir_fd(char* file, int truncate) {
    int flags = O_CREAT|O_RDWR;
    if (truncate)
        flags |= O_TRUNC;
    else
        flags |= O_APPEND;
    int fd = open(file, flags,
    ↪ S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
    if (fd < 0) {
        char buf[BUFLEN] = {0};
        snprintf(buf, sizeof buf, "cannot open %s", file);
        perror(buf);
    }
    return fd;
```

```
}
void spawn_redir_command(struct execcmd* cmd) {
    if (strlen(cmd->in_file) > 0)
        dup2(open_redir_fd(cmd->in_file, 0), STDIN_FILENO);
    if (strlen(cmd->out_file) > 0) {
            dup2(open_redir_fd(cmd->out_file, 1), STDOUT_FILENO);
    }
    spawn_command(cmd);
}
void exec_cmd(struct cmd* cmd) {

    switch (cmd->type) {
        case REDIR: {
            spawn_redir_command((struct execcmd*)cmd);
            break;
        }
    }
}
```

../exec.c

1.1.2. Error estándar a archivo

La segunda forma que veremos es la salida estándar de errores. El caso es análogo a los anteriores, sólo que se antepone un 2 delante del comando de redirección (el 2 es el file descriptor del error estándar). De esta forma, el operador queda como 2>seguido del nombre de archivo.

El código utilizado para la solución es el siguiente:

```
void spawn_redir_command(struct execcmd* cmd) {
    if (strlen(cmd->err_file) > 0) {
            dup2(open_redir_fd(cmd->err_file, 1), STDERR_FILENO);
    }
    spawn_command(cmd);
}
```

../exec.c

1.1.3. Combinar salida y errores

La última forma que veremos es la redirección de error estándar en salida estándar. Dicha redirección se logra mediante el operador 2>&1. Como generalización se puede ver que también funciona con cualquier file descriptor. Es decir, la expresión general es 2>&fd.

Habiendo visto eso, la solución planteada que se complementa con la solución del punto anterior, es la siguiente:

```
void spawn_redir_command(struct execcmd* cmd) {
    if (strlen(cmd->err_file) > 0) {
        if (cmd->err_file[0] == '&')
            dup2(atoi(&cmd->err_file[1]), STDERR_FILENO);
        else
            dup2(open_redir_fd(cmd->err_file, 1), STDERR_FILENO);
    }
    spawn_command(cmd);
}
```

../exec.c

1.1.4. Challenges

Por último contamos con dos challenges en los cuales se deberá implementar la solución para los operadores >> y &>.

El primero consiste en realizar un append del archivo, a diferencia de los casos anteriores que por defecto truncaban el archivo en caso de que ya esté creado. Entonces, si el archivo existe y tenemos un doble operador redirección, el archivo no se reemplazará sino que el nuevo contenido se situará al final del archivo.
Para esto se debe contemplar el caso especial en la apertura del archivo. A continuación veremos el código correspondiente a la solución:

```
void spawn_redir_command(struct execcmd* cmd) {
    if (strlen(cmd->out_file) > 0) {
        if (cmd->out_file[0] == '>')
            dup2(open_redir_fd(&cmd->out_file[1], 0), STDOUT_FILENO);
        else
            dup2(open_redir_fd(cmd->out_file, 1), STDOUT_FILENO);
    }
    if (strlen(cmd->err_file) > 0) {
        if (cmd->err_file[0] == '&')
            dup2(atoi(&cmd->err_file[1]), STDERR_FILENO);
        else if (cmd->err_file[0] == '>')
            dup2(open_redir_fd(&cmd->err_file[1], 0), STDERR_FILENO);
        else
            dup2(open_redir_fd(cmd->err_file, 1), STDERR_FILENO);
    }
    spawn_command(cmd);
}
```

../exec.c

El segundo operador tiene un comportamiento análogo a 2>&1, es decir, redirige la salida de error estándar a la slaida estándar. A su vez, esta es redirigida a

un archivo, pues luego de &>se espera el nombre de un archivo.
Para este caso se requirió modificar el parser del shell, ya que en caso de encontrar un & y no tener un >antes, se toma como un comando que se debe ejecutar en background. Para esto, se estableció que tampoco seria un comando de background si el & estaba seguido de un >.
Por lo tanto, el código utilizado es el siguiente:

```
static bool parse_redir_flow(struct execcmd* c, char* arg) {
    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;
    }
    return false;
}
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);
}
```

../parsing.c

```
void spawn_redir_command(struct execcmd* cmd) {
    if (strlen(cmd->out_file) > 0 && strlen(cmd->err_file) > 0 &&
    ↪ strcmp(cmd->out_file, cmd->err_file) == 0)
        dup2(STDOUT_FILENO, STDERR_FILENO);
    spawn_command(cmd);
}
```

../exec.c

1.2. Tuberías simples (pipes)

El último objetivo del lab consiste en implementar las tuberías para relacionar dos comandos. Para esto se utilizó la syscall pipe(2), la cual crea una "tubería"mediante dos file descriptors. Uno de los file descriptors funciona como la punta de lectura, mientras que el otro como la punta de escritura.
Toda la información que se escriba en la punta de escritura se guardará en un buffer por el kernel hasta que se tenga que leer desde la punta de lectura.

Para implementar la solución se tuvieron en cuenta varias cosas: primero, se realizan dos fork, uno por cada proceso a ejecutar. Ahora, teniendo tres procesos (el padre y sus dos hijos), se debe trabajar con los file descriptors que que abrieron para la tubería.
El proceso padre no debe utilizar ninguno de los file descriptors, por lo cual su proceso cierra ambos.
En el caso del primer hijo, este sólo debe utilizar la punta de escritura, por lo que cierra la de lectura y duplica el file descriptor de su salida estándar en el de la punta de escritura, de forma de redireccionar todo su contenido a la misma.
El segundo hijo tiene un comportamiento contrario al del primero. Éste solamente necesita la punta de lectura, por lo que cierra la de escritura y duplica su entrada estándar a la punta de lectura.
Luego de realizar los cierres de file descriptors necesarios, cada proceso se ejecuta o, en el caso del padre, espera que sus hijos terminen la ejecución.

A continuación se muestra el código de la solución:

```
void spawn_pipe_command(struct pipecmd* cmd) {
    int pipefd[2];
    int result = pipe(pipefd);
    if (result < 0) {
        char buf[BUFLEN] = {0};
        snprintf(buf, sizeof buf, "error piping commands %s",
    ↪ cmd->scmd);
        perror(buf);
        return;
    }
    pid_t pid_a, pid_b;
    if ((pid_a = fork()) == 0) {
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        exec_cmd(cmd->leftcmd);
    } else if ((pid_b = fork()) == 0) {
        close(pipefd[1]);
        dup2(pipefd[0], STDIN_FILENO);
        exec_cmd(cmd->rightcmd);
    } else {
        close(pipefd[0]);
        close(pipefd[1]);
```

```
    }
    waitpid(pid_a, NULL, 0);
    waitpid(pid_b, NULL, 0);
}
void exec_cmd(struct cmd* cmd) {

    switch (cmd->type) {
        case PIPE: {
            spawn_pipe_command((struct pipecmd*)cmd);
            free_command(parsed_pipe);
            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 #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 static void set_environ_vars(char** eargv, int eargc) {
28    for (int i = 0; i < eargc; i++) {
29        char key[ARGSIZE] = {0};
30        char value[ARGSIZE] = {0};
31        get_environ_key(eargv[i], key);
32        get_environ_value(eargv[i], value, strlen(key));
33        setenv(key, value, 1);
34    }
35 }
36
37 // opens the file in which the stdin/stdout or
38 // stderr flow will be redirected, and returns
39 // the file descriptor
40 static int open_redir_fd(char* file, int truncate) {
41    int flags = O_CREAT|O_RDWR;
42    if (truncate)
43        flags |= O_TRUNC;
44    else
45        flags |= O_APPEND;
```

```
46      int fd = open(file, flags,
    ↪ S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
47      if (fd < 0) {
48          char buf[BUFLEN] = {0};
49          snprintf(buf, sizeof buf, "cannot open %s", file);
50          perror(buf);
51      }
52      return fd;
53  }
54
55  void spawn_command(struct execcmd* cmd) {
56      set_environ_vars(cmd->eargv, cmd->eargc);
57      int result = execvp(cmd->argv[0], cmd->argv);
58      if (result < 0) {
59          char buf[BUFLEN] = {0};
60          snprintf(buf, sizeof buf, "error executing %s", cmd->scmd);
61          perror(buf);
62      }
63  }
64
65  void spawn_background_command(struct backcmd* cmd) {
66      exec_cmd(cmd->c);
67  }
68
69  void spawn_redir_command(struct execcmd* cmd) {
70      if (strlen(cmd->in_file) > 0)
71          dup2(open_redir_fd(cmd->in_file, 0), STDIN_FILENO);
72      if (strlen(cmd->out_file) > 0) {
73          if (cmd->out_file[0] == '>')
74              dup2(open_redir_fd(&cmd->out_file[1], 0), STDOUT_FILENO);
75          else
76              dup2(open_redir_fd(cmd->out_file, 1), STDOUT_FILENO);
77      }
78      if (strlen(cmd->err_file) > 0) {
79          if (cmd->err_file[0] == '&')
80              dup2(atoi(&cmd->err_file[1]), STDERR_FILENO);
81          else if (cmd->err_file[0] == '>')
82              dup2(open_redir_fd(&cmd->err_file[1], 0), STDERR_FILENO);
83          else
84              dup2(open_redir_fd(cmd->err_file, 1), STDERR_FILENO);
85      }
86      if (strlen(cmd->out_file) > 0 && strlen(cmd->err_file) > 0 &&
    ↪ strcmp(cmd->out_file, cmd->err_file) == 0)
87          dup2(STDOUT_FILENO, STDERR_FILENO);
88      spawn_command(cmd);
89  }
90
91  void spawn_pipe_command(struct pipecmd* cmd) {
92      int pipefd[2];
93      int result = pipe(pipefd);
94      if (result < 0) {
95          char buf[BUFLEN] = {0};
```

```
96          snprintf(buf, sizeof buf, "error piping commands %s",
    ↪ cmd->scmd);
97          perror(buf);
98          return;
99      }
100     pid_t pid_a, pid_b;
101     if ((pid_a = fork()) == 0) {
102         close(pipefd[0]);
103         dup2(pipefd[1], STDOUT_FILENO);
104         exec_cmd(cmd->leftcmd);
105     } else if ((pid_b = fork()) == 0) {
106         close(pipefd[1]);
107         dup2(pipefd[0], STDIN_FILENO);
108         exec_cmd(cmd->rightcmd);
109     } else {
110         close(pipefd[0]);
111         close(pipefd[1]);
112     }
113     waitpid(pid_a, NULL, 0);
114     waitpid(pid_b, NULL, 0);
115 }
116
117 // executes a command - does not return
118 void exec_cmd(struct cmd* cmd) {
119
120     switch (cmd->type) {
121
122         case EXEC: {
123             spawn_command((struct execcmd*)cmd);
124             break;
125         }
126
127         case BACK: {
128             spawn_background_command((struct backcmd*)cmd);
129             break;
130         }
131
132         case REDIR: {
133             spawn_redir_command((struct execcmd*)cmd);
134             break;
135         }
136
137         case PIPE: {
138             spawn_pipe_command((struct pipecmd*)cmd);
139             free_command(parsed_pipe);
140             break;
141         }
142     }
143 }
144
145 void exec_cd(char* cmd) {
146     char* nwd = split_line(cmd, ' ');
```

```
147     if (chdir(nwd) != 0) {
148         char buf[BUFLEN] = {0};
149         snprintf(buf, sizeof buf, "cannot cd to %s", nwd);
150         perror(buf);
151     }
152 }
153
154 void exec_pwd() {
155     char* cwd = getcwd(NULL, 0);
156     printf("%s\n", cwd);
157     free(cwd);
158 }
```

../exec.c

```
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        strcpy(c->out_file, &arg[outIdx + 2]);
27        strcpy(c->err_file, &arg[outIdx + 2]);
28
29        free(arg);
30        c->type = REDIR;
31
32        return true;
33    }
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         char* env = getenv(&arg[1]);
110         if (env == NULL)
111             env = "";
112         if (strlen(env) > ARGSIZE)
113             arg = realloc(arg, strlen(env));
114         strcpy(arg, env);
115     }
116
117     return arg;
118 }
119
120 // parses one single command having into account:
121 // - the arguments passed to the program
122 // - stdin/stdout/stderr flow changes
123 // - environment variables (expand and set)
124 static struct cmd* parse_exec(char* buf_cmd) {
125
126     struct execcmd* c;
127     char* tok;
128     int idx = 0, argc = 0;
129
130     c = (struct execcmd*)exec_cmd_create(buf_cmd);
131
132     while (buf_cmd[idx] != END_STRING) {
133
134         tok = get_token(buf_cmd, idx);
135         idx = idx + strlen(tok);
136
137         if (buf_cmd[idx] != END_STRING)
138             idx++;
```

```
139
140         if (parse_redir_flow(c, tok))
141             continue;
142
143         if (parse_environ_var(c, tok))
144             continue;
145
146         tok = expand_environ_var(tok);
147
148         c->argv[argc++] = tok;
149     }
150
151     c->argv[argc] = (char*)NULL;
152     c->argc = argc;
153
154     return (struct cmd*)c;
155 }
156
157 // parses a command knowing that it contains
158 // the '&' char
159 static struct cmd* parse_back(char* buf_cmd) {
160
161     int i = 0;
162     struct cmd* e;
163
164     while (buf_cmd[i] != '&')
165         i++;
166
167     buf_cmd[i] = END_STRING;
168
169     e = parse_exec(buf_cmd);
170
171     return back_cmd_create(e);
172 }
173
174 // parses a command and checks if it contains
175 // the '&' (background process) character
176 static struct cmd* parse_cmd(char* buf_cmd) {
177
178     if (strlen(buf_cmd) == 0)
179         return NULL;
180
181     int idx;
182
183     // checks if the background symbol is after
184     // a redir symbol, in which case
185     // it does not have to run in in the 'back'
186     if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
187             (buf_cmd[idx - 1] != '>' && buf_cmd[idx + 1] != '>'))
188         return parse_back(buf_cmd);
189
190     return parse_exec(buf_cmd);
```

```
191 }
192
193 // parses the command line
194 // looking for the pipe character '|'
195 struct cmd* parse_line(char* buf) {
196
197     struct cmd *r, *l;
198
199     char* right = split_line(buf, '|');
200
201     l = parse_cmd(buf);
202     r = parse_cmd(right);
203
204     return pipe_cmd_create(l, r);
205 }
```

../parsing.c