18 DE DICIEMBRE DE 2022

MINI-SHELL

2ª PRÁCTICA

JOSÉ LUIS MEZQUITA JIMÉNEZ Y MIGUEL ÁNGEL VILLANUEVA
GRADO EN IGENIERÍA DE LA CIBERSEGURIDAD
ASIGNATURA: SISTEMAS OPERATIVOS

En esta segunda práctica de la asignatura de Sistemas Operativos, hemos tenido que crearnos nuestra propia minishell. Para ello hemos tenido que consultar las diapositivas como el propio internet cuando nos surgían dudas.

El esquema que vamos a seguir en esta memoria es, empezando a comentar el código mediante capturas realizadas de este (aunque en el propio código ya hay comentarios) y su explicación y al final haremos una conclusión con los problemas que nos han surgido en esta práctica y lo que hemos conseguido hacer como lo que más nos ha costado.

Ahora vamos a empezar comentando todo nuestro código mediante una serie de capturas:

Estas son todas las librerías que hemos tenido que usar para la resolución de esta práctica.

myshell.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include "parser.h"
```

Ahora vamos a explicar en que consiste el contenido del #include “parser.h”.

myshell.c | parser.h

```c
typedef struct {
    char * filename;
    int argc;
    char ** argv;
} tcommand;

typedef struct {
    int ncommands;
    tcommand * commands;
    char * redirect_input;
    char * redirect_output;
    char * redirect_error;
    int background;
} tline;

extern tline * tokenize(char *str);
```

La librería “parser.h” es una librería que nos ayuda en la gestión de los mandatos recibidos, es decir, nos dice cuantos mandatos hay en la instrucción que escribimos por teclado, detalla cada mandato con sus argumentos, nos indica si hay una o varias redirecciones de fichero y, por último, también nos indica su un mandato va a ser ejecutado en segundo plano.

Estas son las dos funciones que nos hemos tenido que crear para el desarrollo de la práctica, una es “void redirect ()” que nos va a redireccionar (en caso de que el mandato lo indique) y “void cd ()” que nos va a realizar el mandato cd.

```c
void redirect(){
    if (line->redirect_input != NULL){ // si hay redireccion de entrada
        int entrada = open(line->redirect_input, O_RDONLY); // abrimos el fichero de redireccion
        if (entrada == -1){
            fprintf(stderr, "%s: Error %s\n", line->redirect_input, strerror(errno));
            exit(1);
        }else{
            dup2(entrada,fileno(stdin));
        }
    }

    if (line->redirect_output != NULL){ // si hay redireccion de salida
        int salida = creat(line->redirect_output, S_IWUSR | S_IRUSR); // creamos o sobreescribimos el fichero de redireccion
        if (salida == -1){
            fprintf(stderr, "%s: Error %s\n", line->redirect_output, strerror(errno));
            exit(1);
        }else{
            dup2(salida,fileno(stdout));
        }
    }

    if (line->redirect_error != NULL){ // si hay redireccion de error
        int error = creat(line->redirect_error, S_IWUSR | S_IRUSR);
        if (error == -1){
            fprintf(stderr, "%s: Error %s\n", line->redirect_error, strerror(errno));
            exit(1);
        }else{
```

En la función de redirección, usado la función “dup2(variable1, variable2)” para controlar que tanto variable 1 como variable 2 “contengan lo mismo”.

La otra función que nos hemos creado es la que ejecuta el mandato “cd”.

En la siguiente captura vemos el código de esta, en donde, para empezar, nos hemos creado un puntero *dir que va a contener los directorios y un buffer que va a almacenar lo que es el mandato completo. Como sabemos que el mandato cd puede tener únicamente 1 o 2 argumentos, ya que, si solo tiene uno, significa que estamos haciendo cd a secas, mientras que, si pasamos dos argumentos, quiere decir que le estamos pasando cd + ubicación.

Esto lo comprobamos en el primer if, si le pasamos mas de dos argumentos, nos salta un mensaje de error.

En el siguiente if, comprobamos si estamos pasando uno o dos argumentos. Aquí es donde entra en juego el puntero *dir que nos hemos creado ya que va a almacenar el directorio al que nos vamos a dirigir.

Si el número de argumentos es uno, es que solo estamos poniendo cd, por lo tanto, dir va a contener la ruta de la variable HOME, aunque es raro que suceda, a continuación, vamos a comprobar que la variable HOME existe.

Si el numero de argumentos es dos, asignamos a *dir el valor del segundo argumento, es decir, la ruta que le hemos pasado (siempre rutas completas, es decir, /home/…). Después de esto, nos aseguramos que el directorio que le hemos pasado es de verdad un directorio y no otra cosa.

```
194
195  void cd (){
196
197      char *dir; // variable de directorios
198      char buffer[512];
199
200      if(line->commands[0].argc > 2)  {
201          fprintf(stderr,"Uso: %s directorio\n", line->commands[0].argv[0]);
202      }
203      if (line->commands[0].argc == 1) { // si ejecutamos solo cd
204          dir = getenv("HOME");
205          if(dir == NULL){
206              fprintf(stderr,"No existe la variable $HOME\n");
207          }
208      }else { // si ejecutamos cd con un directorio
209          dir = line->commands[0].argv[1];
210      }
211
212      if (chdir(dir) != 0) { // comprobamos si es un directorio.
213          fprintf(stderr,"Error al cambiar de directorio: %s\n", strerror(errno));
214      }
215      printf( "El directorio actual es: %s\n", getcwd(buffer,-1));
216  }
```

Una vez ya explicadas las dos funciones que vamos a usar, en el main, nos hemos creado una variable buffer que va a contener la entrada, es decir, el mandato que le hemos pasado por teclado a nuestra minishell. Al final de esta captura, hemos ignorado las señales que se nos indica en el enunciado de la práctica.

```c
tline *line;

void redirect(); // función para la redirección
void cd (); // función para el mandato cd

int main(void)
{
    char buf[1024]; // almacenamos los input en un buffer
    int pid;
    printf("msh >> ");

    // ignoramos las señales SIGINT y SIGQUIT para la minishell
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
```

En esta captura, nos centramos en el momento cuando nos pasan un único mandato, es decir, sin pipes.

Para ello, tokenizamos, es decir, dividimos el mandato pasado por teclado, en distintas cadenas de caracteres.

Al empezar volvemos a ignorar las señales, después, comprobamos que es un cd el mandato pasado, si es así, llamamos a la función cd(), si no, ejecutamos el mandato pasado, siempre llamando a la función redirect() para redireccionar en caso que el mandato nos lo indique.

```c
line = tokenize(buf); // tokenizamos lo que hay en el buffer y lo guardamos en line
 if (line->ncommands == 1){ //solo un mandato

    //los procesos en primer plano no ignoran las señales SIGINT y SIGQUIT
    signal(SIGINT, SIG_DFL);
    signal(SIGQUIT, SIG_DFL);

    if (strcmp(line->commands[0].argv[0],"cd")==0){ // si el mandato es cd
        cd();
    }else{ // ejecutamos el mandato correspondiente
        pid = fork();
        if (pid < 0){ // error en el fork
            fprintf(stderr, "Error en el fork %s\n", strerror(errno));
        }else if (pid == 0){ // proceso hijo
            redirect(); // comprobamos si hay redirecciones
            execvp(line->commands[0].argv[0],line->commands[0].argv); // ejecutamos mandato
            fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]); // imprir error en caso de no exec
            exit(1);
        }else{ // proceso padre
            wait(NULL);
        }
    }
```

Ahora nos vamos a centrar en 2 mandatos, por lo tanto, 1 pipe.

```
}else if (line->ncommands == 2){// 2 mandatos 1 pipe
    int pipe_des[2];
    pipe(pipe_des);

    for (int i = 0; i < 2; ++i){

        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);

        pid = fork();
        if (i == 0){ // primer mandato
            if (pid < 0){
                fprintf(stderr, "Error en el fork %s\n", strerror(errno));
            }else if (pid == 0){
                close(pipe_des[0]); // cerramos la parte del pipe que no se usa
                redirect();
                dup2(pipe_des[1],fileno(stdout)); // escritura del pipe es la salida estandar
                close(pipe_des[1]);
                execvp(line->commands[i].filename, line->commands[i].argv);
                fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]);
                exit(1);
            }else{
                close(pipe_des[1]);
                wait(NULL);
            }
        }else if (i == 1){ // segundo mandato
            if (pid < 0){
                fprintf(stderr, "Error en el fork %s\n", strerror(errno));
            }else if (pid == 0){
                close(pipe_des[1]);
                redirect();
                dup2(pipe_des[0],fileno(stdin));
                close(pipe_des[0]);
                execvp(line->commands[i].filename, line->commands[i].argv);
                fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]);
                exit(2);
            }else{
                close(pipe_des[0]);
                wait(NULL);
            }
        }
    }
```

Ahora vamos a tratar cuando pasamos por teclado mas de dos mandatos, es decir, más de un pipe. Para ello nos vamos a crear una matriz donde se almacenen todos los pipes (al final se muestra como liberamos la matriz creada de forma dinámica. Para resolverlo, hemos creado un bucle for que recorra de 0 al numero de mandatos que hay.

```
}else{ // varios mandatos con varios pipes
    int npipes = line->ncommands - 1; // numero de pipes

    int **pipes = (int**) malloc(npipes*sizeof(int*)); // reservo en mem. dinamica una matriz con los pipes
    for (int i = 0; i < npipes; ++i){
        pipes[i] = (int *) malloc(2 * sizeof(int));
        pipe(pipes[i]);
    }

    for (int i = 0; i < line->ncommands; ++i){

        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);

        pid = fork();

        if (pid < 0){
            fprintf(stderr, "Error en el fork %s\n", strerror(errno));
        }else if (pid == 0){ // proceso hijo
            if (i == 0){ // primer mandato
                close(pipes[i][0]);
                redirect();
                dup2(pipes[i][1],fileno(stdout));
                close(pipes[i][1]);
                execvp(line->commands[i].filename, line->commands[i].argv);
                fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]);
                exit(1);
            }else if (i == line->ncommands - 1){// ultimo mandato
                close(pipes[i-1][1]);
                redirect();
                dup2(pipes[i-1][0],fileno(stdin));
                close(pipes[i-1][0]);
                execvp(line->commands[i].filename, line->commands[i].argv);
                fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]);
                exit(1);
```

```
                }else{ // resto de mandatos
                    close(pipes[i-1][1]);
                    close(pipes[i][0]);
                    dup2(pipes[i-1][0],fileno(stdin));
                    dup2(pipes[i][1],fileno(stdout));
                    close(pipes[i-1][0]);
                    close(pipes[i][1]);
                    execvp(line->commands[i].filename, line->commands[i].argv);
                    fprintf(stderr, "%s No se encuentra el mandato\n", line->commands[0].argv[0]);
                    exit(1);
                }
            }else{ // proceso padre
                if(!(i==(line->ncommands-1))){
                        close(pipes[i][1]);
                }
                wait(NULL);
            }
        }

        // liberamos mem. dinamica
        for (int i = 0; i < npipes; ++i){
            free(pipes[i]);
        }
        free(pipes);
    }
    printf("msh >> ");
```

CONCLUSIONES

Esta práctica nos ha ayudado bastante a entender mejor el concepto de los procesos padre e hijo y el manejo de las señales; pero sobre todo nos ha servido para comprender el funcionamiento de los pipes, aunque era una parte complicada, hemos conseguido resolver los problemas que se nos han presentado. En resumen, esta practica ha sido más complicada que la anterior, hemos tenido que indagar y buscar información, pero nos ha ayudado mucho a entender este tema.