

A dark blue vertical bar runs along the left edge of the page. A blue arrow-shaped banner points to the right from this bar, containing the date '1-6-2020'. In the bottom-left corner, there are several thin, curved, light blue lines that sweep upwards and to the right.

1-6-2020

Práctica 2. MiniShell

Sistemas Operativos

Javier Barrio

GRADO EN INGENIERÍA DE SOFTWARE

Práctica 2. MiniShell

Descripción y entendimiento del Problema:	2
Etapa 0: Planteamiento del Main	2
Etapa 1: Función executeLine()	3
Etapa 2: Pipes	4
Etapa 3: Redirecciones	5
Etapa 4: Cd	6
Etapa 5: Background	9
Problemas Encontrados	10
Evaluación del Tiempo Dedicado	10

Práctica 2. MiniShell

Descripción y entendimiento del Problema:

En esta práctica se abordará el problema de implementar un programa que actúe como intérprete de mandatos. El minishell en cuestión, debe interpretar y ejecutar los mandatos introducidos por la entrada estándar, es decir, el teclado.

Debe ser capaz de ejecutar una secuencia de uno o varios mandatos separados por '|', además, se deben permitir las redirecciones, para ello, deberemos poder redireccionar la entrada o salida estándar a un fichero, incluyendo la salida de error.

Por otro lado, se debe permitir la ejecución en segundo plano o background, con se ejecute el minishell debe mostrar el pid del proceso por el que está esperando entre corchetes, y no bloquearse por la ejecución de dicho mandato.

El minishell deberá mostrar el prompt, leer una línea introducida por el teclado, analizarla mediante la librería parser y ejecutar todos los mandatos de la línea creando los procesos hijos correspondientes, estos, deben estar bien comunicados entre sí, para ello, se utilizarán pipes.

Por cada línea que leamos mediante tokenize vamos a analizar una array de mandatos que almacenaremos en la estructura de datos suministrada por el profesorado en el archivo parser.h.

En cada posición de nuestro Array tendremos un registro, con un campo fileName con la ruta del primer mandato introducido, el segundo campo, argc, almacenará el número de argumentos que tendrá mi programa y, el último campo, argv, será un array con los mandatos introducidos.

Debido a la extensión de la práctica la dividiré en etapas para elaborarla poco a poco e ir probándola.

Etapas 0: Planteamiento del Main

En la fase 0 vamos a declarar las variables principales, line de tipo tline y un buffer, al mismo tiempo vamos a plantear un bucle infinito que mostrará el prompt, lee una línea de la entrada estándar, rellenará el buffer, y a través de tokenize rellenaremos nuestra estructura de datos planteada en la librería parser.h, en el caso de que los datos de la entrada estándar sean NULL se romperá el bucle infinito, en caso contrario se volverá a mostrar el prompt.

Ilustración

```
#include <stdio.h>
#include "parser.h"

int
main(void) {
    //Sección de Variables

    tline * line; //Variable de tipo line
    char buffer[1024]; //Buffer

    //Fin de la Sección de Variables

    for (;;) { //Bucle Infinito
        printf("Shell ==>");
        if (fgets(buffer, 1024, stdin) == NULL) { //Leo la línea y si es igual a Null salgo del bucle
            break;
        } else {
            line = tokenize(buffer); //Tokenizo, es decir, relleno la estructura de datos
        }
    }

    return 0;
}
```

Ilustración 1: Código Etapa 0 – Función Main

Etapa 1: Función executeLine()

En la etapa uno vamos a implementar la ejecución de los mandatos almacenados por tokenize. Lo primero de todo es implementar la función executeLine(), en ella mediante un bucle recorreremos el número de mandatos almacenados en la variable line.

Por cada mandato, crearemos un proceso hijo mediante fork(), y en este proceso hijo ejecutaremos el mandato correspondiente en el bucle for a través de execvp. Finalmente, crearemos un bucle donde el padre esperará mediante un Wait a cada proceso hijo creado.

Ilustración

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <unistd.h>
#include "parser.h"

//Sección de Variables Globales
tline * line; //Variable de tipo line
// Fin Sección de Variables Globales

void
executeLine(void){
    //Sección de Variables Locales
    int i; //Variable contadora
    pid_t PID; //Variable para guardar el PID del hijo

    //Fin de la Sección de Variables

    for (i = 0; i < line->ncommands; i++){
        PID = fork(); //Creamos un proceso hijo
        if (PID < 0){
            fprintf(stderr, "Error al crear proceso hijo"); //Notificación del error por la salida de error
            exit(1); //A partir de ahora el valor de error 1, se utilizará para el error al crear un proceso hijo
        } else if (PID == 0){ //Proceso Hijo
            execvp(line->commands[i].filename, line->commands[i].argv); //Ejecutamos el mandato i
            exit(2); //En caso de no ejecutar correctamente execvp abortamos ejecución
        }
    }

    for (i = 0; i < line->ncommands; i++){
        wait(NULL);
    }
}

int
main(void) {
    //Sección de Variables Locales
    char buffer[1024]; //Buffer
    //Fin de la Sección de Variables

    for (;;){ //Bucle Infinito
        printf("Shell ==>");
        if (fgets(buffer, 1024, stdin) == NULL){ //Leo la linea y si es igual a Null salgo del bucle
            break;
        } else {
            line = tokenize(buffer); //Tokenizo, es decir, relleno la estructura de datos
            executeLine();
        }
    }

    return 0;
}
```

Ilustración 2: Código Etapa 1 – Función ExecuteLine y Main

Etapa 2: Pipes

En la etapa dos vamos a implementar el uso de los pipes, para ello deberemos analizar la diferencia entre la situación del número de procesos, pongamos el ejemplo de tres mandatos, el primero deberá redireccionar su salida estándar a un pipe, el segundo deberá redireccionar su entrada estándar al pipe utilizado previamente, y a la vez su salida estándar se redireccionará a otro pipe, que, finalmente redireccionará el último mandato como su entrada estándar.

Vamos a modificar nuestra función `executeLine()`, como queremos que nuestros pipes sean dinámicos declararemos la variable `**p`, y mediante `malloc`, asignaremos la memoria necesaria, en esta variable es donde guardaremos los descriptores de fichero que abren los pipes, y condicionaremos que si hay más de un mandato introducido, se analice cada uno particularmente.

Además, crearemos una función `closePipes`, que dado un pipe, nos cierre ambos descriptores de fichero. Esta función la utilizaremos al finalizar el tratamiento de los mandatos mencionados anteriormente y nada más comenzar el proceso padre.

Ilustración

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <unistd.h>
#include "parser.h"

//Sección de Variables Globales

tline * line; //Variable de tipo line

// Fin Sección de Variables Globales

void
closePipes(int **p){
    for (int i = 0; i < (line->ncommands-1); i++){
        close(p[i][0]); //Cerramos el descriptor de lectura de fichero del pipe
        close(p[i][1]); //Cerramos el descriptor de escritura de fichero del pipe
    }
}

void
executeLine(void){
    //Sección de Variables Locales

    int i; //Variable contadora
    int PID; //Variable para guardar el PID del hijo
    int **p; //Variable para las pipes mediante memoria dinámica

    //Fin de la Sección de Variables

    p = malloc(sizeof(int*)*line->ncommands-1);
    //Reservamos la memoria de un entero tantas veces como número de mandatos - 1
    for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
        p[i] = malloc(sizeof(int)*2); //Reservamos la memoria que ocupan dos enteros
    }

    for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
        pipe(p[i]); //Abrimos dos descriptores de fichero por cada comando introducido
    }

    for (i = 0; i < line->ncommands; i++){ //Bucle de recorrido del número de mandatos
        PID = fork(); //Creamos un proceso hijo
        if (PID < 0){ //Error al crear proceso hijo
            fprintf(stderr, "Error al crear proceso hijo"); //Notificación del error por la salida de error
            exit(1); //A partir de ahora el valor de error 1, se utilizará para el error al crear un proceso hijo
        }

        if (PID == 0){ //Proceso Hijo

            if (line->ncommands > 1){ //Si el numero de comandos es > 1
                if (i == 0){ //Si el mandato es el primero
                    dup2(p[i][1], 1); //Nuestra salida estándar sera el pipe
```

Práctica 2. MiniShell

```
        } else if (i == (line->ncommands-1)){ //Si el mandato es el último
            dup2(p[i-1][0], 0); //Nuestra entrada estándar sera el pipe donde ha escrito el anterior

        } else { //Si el mandato está entre el primero y el ultimo
            dup2(p[i-1][0], 0); //Nuestra entrada estándar sera el pipe donde ha escrito el anterior
            dup2(p[i][1], 1); //Nuestra salida estándar sera el pipe*/
        }
        closePipes(p);
    }

    execvp(line->commands[i].filename, line->commands[i].argv); //Ejecutamos el mandato i
    exit(2); //En caso de no ejecutar correctamente execvp abortamos ejecución
}

closePipes(p); //Cerramos los pipes
for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
    free(p[i]); //Liberamos la memoria
}

free(p); //Liberamos memoria
for (i = 0; i < line->ncommands; i++){ //Esperamos a los hijos
    wait(NULL);
}
}

int
main(void) {
    //Sección de Variables Locales
    char buffer[1024]; //Buffer
    //Fin de la Sección de Variables

    for (;;) { //Bucle Infinito
        printf("Shell ==> ");
        if (fgets(buffer, 1024, stdin) == NULL) { //Leo la línea y si es igual a Null salgo del bucle
            break;
        } else {
            line = tokenize(buffer); //Tokenizo, es decir, relleno la estructura de datos
            printf("\n");
            executeLine();
        }
        printf("\n");
    }

    return 0;
}
```

Ilustración 3: Código Etapa 2 – Función ClosePipes, ExecuteLine y Main

Etapa 3: Redirecciones

En la etapa tres vamos a implementar las redirecciones, volveremos a modificar nuestra función `executeLine()`. Nuestra modificación deberá incluir tres condicionales, el primero, si estamos en el primer mandato y tiene redirección de entrada, abrimos el fichero y, mediante `dup2`, establecemos el fichero como la nueva entrada estándar.

Para el caso de la salida estándar y la salida de error el proceso es el mismo, solo que no abrimos un fichero, si no, que lo creamos, y que cambiará la salida correspondiente.

Ilustración

```
if ((i == 0) && (line->redirect input)){ //Si es el primer mandato y tiene redirección de entrada
    fd = open(line->redirect input, O_RDONLY); //Abrimos el fichero

    if (fd < 0){
        fprintf(stderr, "Error al abrir fichero"); //Notificación del error por la salida de error
        exit(3); //A partir de ahora el valor de error 3, se utilizará para el error al abrir un fichero
    } else {
        dup2(fd, 0); //Nuestra entrada estandar ahora es el fichero
        close(fd); //Cerramos el fichero
    }
}
```

Práctica 2. MiniShell

```
if ((i == line->ncommands-1) && (line->redirect output)){ //Si es el último y tiene redirección de entrada
    fd = creat(line->redirect output, 0644); //Creamos el fichero
    if (fd < 0){
        fprintf(stderr, "Error al crear fichero"); //Notificación del error por la salida de error
        exit(4); //A partir de ahora el valor de error 4, se utilizará para el error al crear un fichero
    } else {
        dup2(fd, 1); //Nuestra salida estandar ahora es el fichero
        close(fd); //Cerramos el fichero
    }
}

if ((i == line->ncommands-1) && (line->redirect error)){ //Si tiene redirección de Error
    fd = creat(line->redirect error, 0644); //Creamos el fichero
    if (fd < 0){
        fprintf(stderr, "Error al crear fichero"); //Notificación del error por la salida de error
        exit(4); //A partir de ahora el valor de error 4, se utilizará para el error al crear un fichero
    } else {
        dup2(fd, 2); //Nuestra salida de error ahora es el fichero
        close(fd); //Cerramos el fichero
    }
}
```

Ilustración 4: Código Etapa 3 – Modificación de Executeline - Redirecciones

Etapa 4: Cd

En la etapa cuatro vamos a implementar el mandato `cd`, `changeDirectory`, para implementarlo, creamos una función `isChangeDirectory`; para comprobar si el mandato que nos han introducido en la minishell es `cd`, si es así, la función devolverá `true`.

Ahora nos centramos en una nueva función llamada `executeCd()`, en ella, mediante la llamada al sistema `chdir` cambiaremos el directorio de trabajo, pero tendremos que tener en cuenta si el mandato tiene un argumento o más de uno. Esto lo podremos comprobar mediante `argc`, que nos devuelve el número de argumentos.

En el caso de ser uno, únicamente habrán escrito `cd`, y por lo tanto nos moveremos mediante `chdir` a la variable de entorno `HOME`, en caso de ser más de uno, nos moveremos a la ruta o directorio establecido.

También he tenido en cuenta que si en la minishell no introducen nada, se reinicia el bucle y nos vuelva a mostrar el prompt.

Para finalizar esta etapa, he querido darle una apariencia a la Shell real, por lo tanto, mediante `getcwd` obtengo el path actual y lo muestro de forma similar a como sucede en la terminal de Linux.

Ilustración

```
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include "parser.h"

//Sección de Variables Globales

tline * line; //Variable de tipo line
const char changeDirectory[] = "cd";

// Fin Sección de Variables Globales
```

Práctica 2. MiniShell

```
void
executeCd (tline *line){
    //Sección de Variables Locales

    char *home;
    int error;

    //Fin de la Sección de Variables

    if ((line->commands[0].argc == 1){ //Si solo han introducido cd, un argumento
        home = getenv("HOME"); //Capturamos la variable del entorno HOME
        error = chdir(home); //Cambiamos el directorio a home
        if (error < 0){ //Si ese directorio no existe
            fprintf(stderr, "%s No existe \n", home);
        }
    } else { //Si nos dan el nombre del directorio al que queremos ir
        error = chdir(line->commands[0].argv[1]); //Cambiamos el directorio al que nos dan
        if (error < 0){ //Si ese directorio no existe
            fprintf(stderr, "%s No existe \n", line->commands[0].argv[1]);
        }
    }
}

}

bool
isChangeDirectory(tline *line){
    if (strcmp(line->commands[0].argv[0], changeDirectory) == 0){ //Si introducimos cd como argumento
        return true; //Devolvemos TRUE
    } else {
        return false; //Si no, FALSE
    }
}

void
closePipes(int **p){
    for (int i = 0; i < (line->ncommands-1); i++){
        close(p[i][0]); //Cerramos el descriptor de lectura de fichero del pipe
        close(p[i][1]); //Cerramos el descriptor de escritura de fichero del pipe
    }
}

void
executeLine(void){
    //Sección de Variables Locales

    int i; //Variable contadora
    int PID; //Variable para guardar el PID del hijo
    int **p; //Variable para las pipes mediante memoria dinámica
    int fd;

    //Fin de la Sección de Variables

    p = malloc(sizeof(int*)*line->ncommands-1);
    //Reservamos la memoria de un entero tantas veces como número de mandatos - 1
    for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
        p[i] = malloc(sizeof(int)*2); //Reservamos la memoria que ocupan dos enteros
    }

    for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
        pipe(p[i]); //Abrimos dos descriptors de fichero por cada comando introducido
    }

    for (i = 0; i < line->ncommands; i++){ //Bucle de recorrido del número de mandatos
        PID = fork(); //Creamos un proceso hijo
        if (PID < 0){ //Error al crear proceso hijo
            fprintf(stderr, "Error al crear proceso hijo"); //Notificación del error por la salida de error
            exit(1); //A partir de ahora el valor de error 1, se utilizará para el error al crear un proceso hijo
        }

        if (PID == 0){ //Proceso Hijo
            //Redirecciones

            if ((i == 0) && (line->redirect input)){ //Si es el primer mandato y tiene redirección de entrada
                fd = open(line->redirect input, O_RDONLY); //Abrimos el fichero

                if (fd < 0){
                    fprintf(stderr, "Error al abrir fichero"); //Notificación del error por la salida de error
                    exit(3); //A partir de ahora el valor de error 3, se utilizará para el error al abrir un fichero
                } else {
                    dup2(fd, 0); //Nuestra entrada estandar ahora es el fichero
                    close(fd); //Cerramos el fichero
                }
            }
        }
    }
}
```


Práctica 2. MiniShell

```
if ((i == line->ncommands-1) && (line->redirect output)){ //Si es el último y tiene redirección de entrada
    fd = creat(line->redirect output, 0644); //Creamos el fichero

    if (fd < 0){
        fprintf(stderr, "Error al crear fichero"); //Notificación del error por la salida de error
        exit(4); //A partir de ahora el valor de error 4, se utilizará para el error al crear un fichero
    } else {
        dup2(fd, 1); //Nuestra salida estandar ahora es el fichero
        close(fd); //Cerramos el fichero
    }
}

if ((i == line->ncommands-1) && (line->redirect error)){ //Si tiene redirección de Error
    fd = creat(line->redirect error, 0644); //Creamos el fichero

    if (fd < 0){
        fprintf(stderr, "Error al crear fichero"); //Notificación del error por la salida de error
        exit(4); //A partir de ahora el valor de error 4, se utilizará para el error al crear un fichero
    } else {
        dup2(fd, 2); //Nuestra salida de error ahora es el fichero
        close(fd); //Cerramos el fichero
    }
}

//Fin de Redirecciones

if (line->ncommands > 1){ //Si el numero de comandos es > 1
    if (i == 0){ //Si el mandato es el primero
        dup2(p[i][1], 1); //Nuestra salida estándar sera el pipe

    } else if (i == (line->ncommands-1)){ //Si el mandato es el último
        dup2(p[i-1][0], 0); //Nuestra entrada estándar sera el pipe donde ha escrito el anterior

    } else { //Si el mandato está entre el primero y el ultimo
        dup2(p[i-1][0], 0); //Nuestra entrada estándar sera el pipe donde ha escrito el anterior
        dup2(p[i][1], 1); //Nuestra salida estándar sera el pipe*/
    }

    closePipes(p);
}

execvp(line->commands[i].filename, line->commands[i].argv); //Ejecutamos el mandato i
exit(2); //En caso de no ejecutar correctamente execvp abortamos ejecución
}

closePipes(p); //Cerramos los pipes
for (i = 0; i < (line->ncommands-1); i++){ //Bucle de recorrido del número de mandatos
    free(p[i]); //Liberamos la memoria
}

free(p); //Liberamos memoria
for (i = 0; i < line->ncommands; i++){ //Esperamos a los hijos
    wait(NULL);
}

int
main(void) {
    //Sección de Variables Locales
    char buffer[1024]; //Buffer
    char* path;

    //Fin de la Sección de Variables

    for (;;) { //Bucle Infinito
        if ((path = getcwd(NULL, 0)) == NULL) { //Obtenemos la ruta actual
            exit(5); //A partir de ahora el valor de error 5, se producirá cuando no se pueda obtener la ruta actual
        } else {
            printf("\n-%s$ | Shell ==> ", path); //Mostramos el prompt con nuestra ruta actual
        }
        if (fgets(buffer, 1024, stdin) == NULL) { //Leo la línea y si es igual a Null salgo del bucle
            break;
        }
        if (line = tokenize(buffer); //Tokenizo, es decir, relleno la estructura de datos
            if ((line == NULL) || (strlen(buffer) == 1)) { //Si la entrada estandar está vacía o su longitud es 1
                continue; //Reinicia el bucle
            }
            printf("\n");
            if (isChangeDirectory(line)) { //Si introducen cd
                executeCd(line); //Ejecutar la función executeCd
            } else {
                executeLine(); //En caso contrario ejecutar la función executeLine
            }
        }
        printf("\n");
    }
    return 0;
}
```

Ilustración 5: Código Etapa 4 – Actualización código implementación de Cd

Práctica 2. MiniShell

Etapa 5: Background

En la última etapa vamos a implementar el campo background, vamos a modificar nuestro main, deshabilitando las señales, ahora en nuestra función executeLine vamos a guardar los PID de los hijos en memoria dinámica y vamos a comprobar si hay background, si no lo hay, volvemos a habilitar las señales.

Por último, modificamos el bucle de Wait por un bucle de waitpid con el PID específico del hijo.

Ilustración

```
int
main(void) {
    //Sección de Variables Locales

    char buffer[1024]; //Buffer
    char* path;

    //Fin de la Sección de Variables

    signal(SIGINT, SIG_IGN); //Deshabilitamos la señal SIGINT
    signal(SIGQUIT, SIG_IGN); //Deshabilitamos la señal SIGQUIT
```

Ilustración 6: Código Etapa 5 – Modificación de Main

Ilustración

```
for (i = 0; i < line->ncommands; i++){ //Bucle de recorrido del número de mandatos
    PIDS = malloc(sizeof(int)*line->ncommands);
    PIDS[i] = fork(); //Creamos un proceso hijo

    if (PIDS[i] < 0){ //Error al crear proceso hijo
        fprintf(stderr, "Error al crear proceso hijo"); //Notificación del error por la salida de error
        exit(1); //A partir de ahora el valor de error 1, se utilizará para el error al crear un proceso hijo
    }

    if (PIDS[i] == 0){ //Proceso Hijo
        //Background

        if (!line->background){ //Si no tenemos background
            signal(SIGINT, SIG_DFL); //Habilitamos la señal SIGINT
            signal(SIGQUIT, SIG_DFL); //Habilitamos la señal SIGQUIT
        }

        //Fin Background
    }
}
```

Ilustración 7: Código Etapa 5 – Modificación de ExecuteLine - Background

Ilustración

```
for (i = 0; i < line->ncommands; i++){ //Esperamos a los hijos
    waitpid(PIDS[i], NULL, 0);
}
```

Ilustración 8: Código Etapa 5 – Modificación de ExecuteLine - WaitPid

Práctica 2. MiniShell

Problemas Encontrados

Uno de los mayores problemas fue el desarrollo total de la práctica, para su resolución se fue implementando por etapas solventando el problema total en pequeños problemas, este método es más conocido como 'divide y vencerás', explicado dentro del grado en asignaturas posteriores.

Evaluación del Tiempo Dedicado

El tiempo dedicado a la práctica es elevado, pero acaba siendo muy útil para entender el funcionamiento de la Shell de Linux, y entender el funcionamiento del lenguaje C.