

**Universidad Rey Juan Carlos**

**Escuela Técnica Superior Ingeniería  
Informática**

Sistemas Operativos

## **PRÁCTICA 2: MINISHELL**

GIS+GII Luis León Gámez

GIS Carlos Vázquez Sánchez

Móstoles - 14 de diciembre de 2014

# Índice general

<b>1. Código</b>	<b>3</b>
1.1. Funciones Auxiliares . . . . .	3
1.1.1. void senal (int s) . . . . .	3
1.1.2. void comandoCD(char* ruta) . . . . .	3
1.1.3. int** crearPipes (int n) . . . . .	3
1.1.4. void cerrarPipes(int** pipes,int n) . . . . .	3
1.1.5. void liberarPipes(int** pipes,int n) . . . . .	3
1.1.6. int redirecEntrada (tline* linea) . . . . .	4
1.1.7. int redirecSalida (tline* linea) . . . . .	4
1.1.8. int redirecError (tline* linea) . . . . .	4
1.1.9. void falloEntrada (int descriptor,char* fichero) . . . . .	4
1.1.10. void lectura (int* pipe) . . . . .	4
1.1.11. void escritura (int* pipe) . . . . .	4
1.2. Main . . . . .	5
<b>2. Comentarios personales</b>	<b>7</b>
2.1. Problemas encontrados . . . . .	7
2.2. Posibles mejoras . . . . .	7
<b>3. Algunas capturas</b>	<b>9</b>

# Capítulo 1

## Código

### 1.1. Funciones Auxiliares

#### 1.1.1. void senal (int s)

Función que se encarga de activar o desactivar las señales SIGINT y SIGQUIT, de tal modo que en caso de tratarse de un proceso en background o de la propia minishell, no respondan. Sin embargo no se activarán si es un proceso foreground.

Su funcionamiento es muy sencillo: usando la función *signal* si se le pasa el parámetro 0 las desactiva, y si es un 1 las activa.

#### 1.1.2. void comandoCD(char\* ruta)

Mediante la función *chdir* cambia el directorio de trabajo al especificado en "ruta". En caso de no especificar una ruta, accede a HOME. En caso de introducirse una ruta inexistente avisará del error.

#### 1.1.3. int\*\* crearPipes (int n)

Función que se encarga de crear las tuberías necesarias. Crea  $n - 1$  tuberías, puesto que para el último hijo no es necesario crear una tubería al ser éste el que ejecuta el último mandato.

#### 1.1.4. void cerrarPipes(int\*\* pipes,int n)

Método que cierra todas las tuberías usadas hasta ese momento.

#### 1.1.5. void liberarPipes(int\*\* pipes,int n)

Método que libera el espacio reservado de las tuberías usadas.

#### **1.1.6. int redirecEntrada (tline\* linea)**

En caso de que se haya introducido un comando con el símbolo < y se trate del primer mandato, se activará la redirección por entrada. Para ello abre el fichero con el nombre especificado por *linea->redirect-input* con permisos de solo lectura, y si no hay error al leerlo copiará el contenido del fichero a la entrada estándar.

#### **1.1.7. int redirecSalida (tline\* linea)**

En caso de que se haya introducido una orden con el símbolo > y se trate del último comando, se activará la redirección por salida. Para ello se creará un fichero con el nombre especificado por *linea->redirect-output* con permisos de lectura y escritura para todo el mundo, y si no hay error al crearlo copiará la salida estándar al fichero.

#### **1.1.8. int redirecError (tline\* linea)**

Como en la función anterior, si en el último comando se introduce una orden con el símbolo & se activará la redirección por error. Para ello se creará un fichero con el nombre especificado por *linea->redirect-error* con permisos de lectura y escritura para todo el mundo en el que guardará, en caso de error, la información sobre la misma.

#### **1.1.9. void falloEntrada (int descriptor,char\* fichero)**

Función que comprueba si se ha cometido un fallo en la apertura de un fichero, y en tal caso, muestra el mensaje de error y finaliza el proceso hijo.

#### **1.1.10. void lectura (int\* pipe)**

Este método recibe una tubería, y se encarga de cerrar el lado de escritura [1], duplicando el lado de lectura [0].

#### **1.1.11. void escritura (int\* pipe)**

Este método recibe una tubería, y se encarga de cerrar el lado de lectura [0], duplicando el lado de escritura [1].

## 1.2. Main

Explicamos muy brevemente su funcionamiento:

1. WHILE: inicia un bucle infinito que lee por entrada estándar los comandos a ejecutar.
2. CD: en el caso de que el primer comando introducido sea *cd*, ejecutará el comando *cd* realizando una llamada al método auxiliar *comandoCD*.
3. EXIT: en el caso de que el primer comando introducido sea *exit*, finalizará el programa.
4. OTRO: en el caso de que el primer comando no sea ninguno de los dos anteriores, inicializará una lista de tuberías (con tantas tuberías como comandos-1 que posea la línea), e iniciará un bucle FOR con tantas iteraciones como comandos se hayan introducido, en la que en cada una se hace la llamada a *fork()* bifurcando el proceso en proceso.
5. HIJO ÚNICO: en el caso de que solo se haya introducido un solo comando, se redireccionará la entrada y salida (tanto normal como error), en caso de que venga indicado en el comando.
6. VARIOS HIJOS: en el caso de que haya más de un comando, se procederá de la siguiente manera:
  - PRIMERO: si el proceso bifurcado es el primero, se redireccionará la entrada, en caso de que venga indicado en el comando, y pondrá en modo escritura la primera tubería.
  - ÚLTIMO: si el proceso bifurcado es el último, se redireccionará la salida (tanto la normal como error), en caso de que venga indicado en el comando, y se pondrá en modo lectura la última tubería.
  - OTRO: si no es el primer o último comando, se pondrá la tubería anterior en modo lectura y la actual en modo escritura.
7. EJECUTA: sea cual sea el proceso bifurcado, se ejecutará el comando correspondiente, y en caso de que no se ejecute, se mostrará por pantalla el error de que el mandato no se ha encontrado, finalizando el proceso.
8. PADRE: el proceso padre, en el caso de que el comando ejecutado se produzca en foreground, esperará a que terminen los comandos antes de proceder a mostrar el prompt. En el caso de que el comando ejecutado se produzca en background, desactivará para este las señales SIGINT y SIGQUIT, mostrará el pid del proceso que se está ejecutando en background, y volverá a mostrar el prompt.

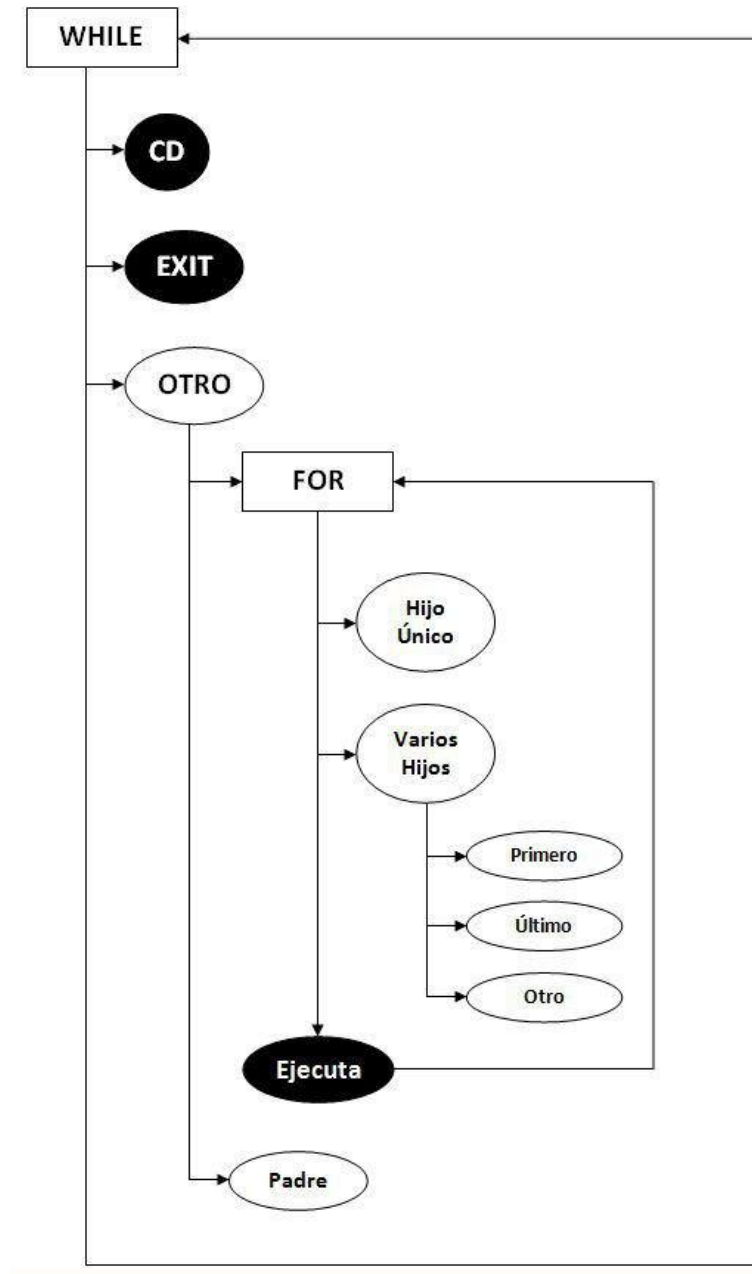


Figura 1.1: Esquema del main

## Capítulo 2

# Comentarios personales

### 2.1. Problemas encontrados

Al igual que la práctica anterior, hemos ido añadiendo funcionalidades de manera incremental. De esta manera, en un principio comprobamos el correcto funcionamiento al ejecutar un solo comando. A continuación probamos si es capaz de redireccionar la salida y la entrada correctamente, añadiendo posteriormente la funcionalidad para el comando "cd". Finalmente añadimos la comunicación entre procesos. Este último paso fue el que más problemas nos acarreó, ya que en un principio no poseíamos los conocimientos teóricos suficientemente claros y no sabíamos como debían crearse los procesos (si se crean hijos sucesivos, o se crean hijos hermanos). Además, nos dieron muchos problemas las tuberías, especialmente en el último proceso, ya que no se cerraba correctamente. Por ello tuvimos que añadir el método *cerrarPipes* que cierra todas las tuberías anteriores, asegurándonos así que no nos cejábamos alguno abierto.

Además para ayudarnos a comprender el funcionamiento de nuestro código, creamos un archivo paralelo que realizaba las mismas funciones, pero mostrando por pantalla información útil para nosotros (pid en ejecución, estado de las pipes...) de los procesos actuales. Manejábamos dichos procesos mediante mandatos *sleep*, de forma que podíamos ver de forma tranquila y detallada el estado de nuestro programa.

### 2.2. Posibles mejoras

Debido a la falta de tiempo no hemos podido comprobar las fugas de memoria, así que no sabemos cual es la calidad de nuestro código en este aspecto. Por otra parte, se podrían continuar añadiendo funcionalidades a la shell hasta conseguir una similar a la original. Podríamos por tanto añadir antes del prompt el usuario que ha ejecutado la terminal, o añadir la funcionalidad del autocompletado con la tecla tabulación, que facilita enor-

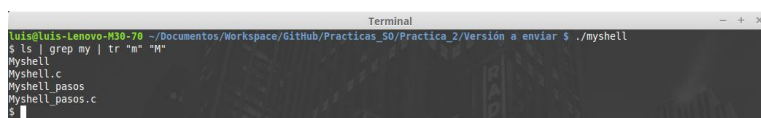
mamente el uso de la terminal cuando hay que escribir rutas largas o tratar con nombres de archivos largos. Siguiendo con ampliaciones que mejoran el manejo de la shell, también ofrecer la posibilidad de usar las teclas de arriba y abajo para acceder al historial de comandos introducidos.

Por otro lado, la estructura de nuestro método *main* no nos parece la más eficiente posible, ya que tiene un número muy alto de ramificaciones y, francamente, es complicado de entender hasta para nosotros. Por tanto nuestra primera prioridad si dispusiéramos de más tiempo sería cambiar la estructura del *main*, reduciendo todo lo posible las ramificaciones y añadiendo más métodos auxiliares que ayuden a la comprensión del código.



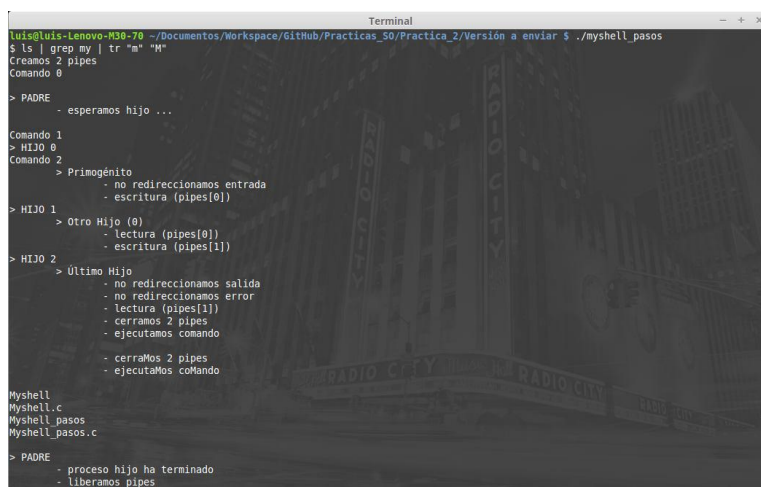
# Capítulo 3

## Algunas capturas



```
Terminal
luis@luis-Lenovo-M30-70 ~/Documentos/Workspace/GitHub/Practicas_S0/Practica_2/Version a enviar $ ./myshell
$ ls | grep my | tr "m" "M"
Myshell
Myshell.c
Myshell_pasos
Myshell_pasos.c
$
```

Figura 3.1: Muestra de ejemplo



```
Terminal
luis@luis-Lenovo-M30-70 ~/Documentos/Workspace/GitHub/Practicas_S0/Practica_2/Version a enviar $ ./myshell_pasos
$ ls | grep my | tr "m" "M"
Creamos 2 pipes
Comando 0
> PADRE
- esperamos hijo ...
Comando 1
> HIJO 0
Comando 2
> Primogénito
- no redireccionamos entrada
- escritura (pipes[0])
> HIJO 1
> Otro Hijo (0)
- lectura (pipes[0])
- escritura (pipes[1])
> HIJO 2
> Último Hijo
- no redireccionamos salida
- no redireccionamos error
- lectura (pipes[1])
- cerramos 2 pipes
- ejecutamos comando
- cerramos 2 pipes
- ejecutamos comando
Myshell
Myshell.c
Myshell_pasos
Myshell_pasos.c
> PADRE
- proceso hijo ha terminado
- liberamos pipes
```

Figura 3.2: Muestra del archivo auxiliar