

Práctica 2 SSOO. Programación de un intérprete  
de mandatos (Minishell)



Nombre: Diego Fuentes Antúnez

NIA: 100405879

Correo electrónico: [100405879@alumnos.uc3m.es](mailto:100405879@alumnos.uc3m.es)

# Índice

1. Descripción del código.....	3
1.1 - Mandatos internos (mycalc y mycp).....	3
1.2 - Minishell.....	4
2. Batería de pruebas.....	5
2.1 - Mandatos simples y secuencias de mandatos.....	5
2.2 - Mandatos y secuencias de mandatos con redirecciones.....	6
2.3 Modo background.....	9
2.4 - Mandatos internos.....	10
2.4.1 - Mycalc.....	10
2.4.2 - Mycp.....	11
3. Conclusiones.....	12

# Descripción del código

A continuación, voy a comentar las principales funcionalidades del código.

Para empezar voy a describir cómo se han implementado los mandatos internos “mycalc” y “mycp”.

## Mandatos internos

### Mycalc

Lo primero que se hace es un string compare para comprobar si el argumento 0 es “mycalc”.

Se ha hecho uso de una variable int check a modo de booleano de tal manera que se comprueba que la sintaxis sea correcta, y de no serlo esta variable se ponga a 0.

Se realizan 4 comprobaciones básicas:

- Que haya 3 argumentos y que ninguno de estos sea NULL (es decir operando1, operador, operando2).
- Que el argumento 1 y el argumento 3 sean números, esto se hará mediante la función isdigit. También se comprueba si el primer carácter del string es un “-” para comprobar si es un número negativo.
- Que el segundo argumento tenga la palabra “add” o la palabra “mod” para saber si hay que hacer la suma o el módulo

Si alguna de las condiciones no se cumple, la variable check se pone a 0 e imprime por la salida estándar de error el mensaje indicado por la práctica

En el caso contrario, hará un string compare para saber si tiene que hacer la suma o el módulo:

- Si tiene que hacer la suma, se aplica la función atoi para transformar el string en un int y así poder hacer la suma, se almacena el resultado en la variable “resultado” y también se va acumulando dicho valor en “acc”. Para terminar imprime por la salida estándar de error el mensaje de éxito.

### mycp

Al igual que con mycalc, se comprueba que argvv[0][0] sea “mycp” con un strcmp para poder acceder al bloque de código. Comprueba que hay mínimo dos argumentos que van a corresponder al fichero de origen y al fichero destino y después procede a codificar el mycp en sí. Abre el fichero origen y destino y comprueba que se han abierto correctamente.

Después en el bucle lee del buffer asociado al descriptor de fichero origen, y escribe en el fichero destino lo que se ha leído en el buffer. El bucle terminará cuando el número de bytes leídos sea 0.

Después se cierran los descriptores de ficheros y en caso de que la variable check no se haya puesto a 0, imprime por la salida estándar de error un mensaje de éxito.

## Minishell

Primero comprueba que lo que se quiere ejecutar no es un mandato interno. Si se trata de un mandato interno no entra en este bloque de código.

Hay un bucle que se ejecuta por el número total de procesos (para ir creando procesos y pipes según sea conveniente en cada iteración del bucle).

Se creará un pipe para todas las iteraciones del bucle menos para la última (puesto que el último proceso imprimirá su salida por la salida estándar).

En cada iteración se hará un fork() para crear un proceso que se corresponderá con un mandato de la secuencia y entramos en el if (pid==0) para hacer las modificaciones pertinentes: Aquí se hará lo siguiente:

- Hacer las redirecciones de entrada y salida para ficheros
- Hacer las redirecciones de entrada y salida entre procesos mediante los pipes.

La redirección de salida estándar de error a fichero se hace en cada iteración del bucle puesto que todos los procesos tienen que imprimir los mensajes de error en dicho fichero.

La redirección de la entrada estándar a fichero se hace solo para el primer mandato de la secuencia, de tal manera que la entrada del primer mandato sea dicho fichero.

Después, para todas las iteraciones que no son la 0 se hace la debida redirección para los pipes (las redirecciones de entrada) y para todas las iteraciones que no son la última se hacen las redirecciones de salida.

Para el último proceso se comprueba si este tiene redirección de salida y en ese caso redirige la salida al fichero indicado por filev[1].

Después se cambia la imagen del proceso mediante un execvp y cuando sale del código del proceso hijo, hace las limpiezas de redirecciones de pipes del proceso padre.

Por último tenemos la condición que hará que podamos ejecutar los mandatos o no en background. Si no indicamos que el mandato se ejecute en background, el padre hará un wait por el hijo y no podremos ejecutar nada más.

## Batería de pruebas

### Mandatos simples

Primero vamos a comprobar la correcta ejecución de mandatos simples:

```
diego@diego-B450-AORUS-ELITE:~/Escritorio/practica 2/mm/p2_minishell_2022$ make
gcc -Wno-implicit-function-declaration -c -o msh.o msh.c -I.
gcc -Wno-implicit-function-declaration -L -o msh msh.o -lparser -L.,-rpath=.
diego@diego-B450-AORUS-ELITE:~/Escritorio/practica 2/mm/p2_minishell_2022$ ./msh
MSH>>ls
autores.txt destino libparser.so Makefile msh msh.c msh.o origen probador_ssso_p2.sh ssso_p2_100405879.zip
MSH>>ls -l
total 120
-rw-rw-r-- 1 diego diego 35 abr 8 00:15 autores.txt
-rw-rw-r-- 1 diego diego 106 abr 8 03:02 destino
-rwxr-xr-x 1 diego diego 21136 feb 28 13:47 libparser.so
-rw-r--r-- 1 diego diego 241 mar 1 13:16 Makefile
-rwxrwxr-x 1 diego diego 22096 abr 8 03:57 msh
-rw-r--r-- 1 diego diego 18714 abr 8 03:48 msh.c
-rw-rw-r-- 1 diego diego 10688 abr 8 03:57 msh.o
-rw-rw-r-- 1 diego diego 106 abr 7 18:32 origen
-rwrxr-xr-x 1 diego diego 12700 mar 1 10:32 probador_ssso_p2.sh
-rw-rw-r-- 1 diego diego 5459 abr 8 03:03 ssso_p2_100405879.zip
MSH>>pwd
/home/diego/Escritorio/practica 2/mm/p2_minishell_2022
MSH>>
```

```
MSH>>cat hola
hola esto
es
una
pruebaaaaMSH>>
```

Como podemos comprobar con la ejecución de ls, ls -l, pwd y cat por probar unos cuantos mandatos simples, no dan ningún tipo de problemas y funcionan como esperado.

### Secuencias de mandatos

Vamos a comprobar ahora la correcta ejecución de secuencias de mandatos conectadas por pipes:

```
diego@diego-B450-AORUS-ELITE:~/Escritorio/practica 2/mm/p2_minishell_2022$ ./msh
MSH>>ls | wc
   11      11     112
MSH>>ls -l | wc
   12     101     606
MSH>>ls -l | wc | wc
      1      3     24
MSH>>ls -l | sort | grep a
-rw-r--r-- 1 diego diego 241 mar 1 13:16 Makefile
-rw-r--r-- 1 diego diego 18714 abr 8 03:48 msh.c
-rw-rw-r-- 1 diego diego 26 abr 8 04:00 hola
-rw-rw-r-- 1 diego diego 35 abr 8 00:15 autores.txt
-rw-rw-r-- 1 diego diego 106 abr 7 18:32 origen
-rw-rw-r-- 1 diego diego 106 abr 8 03:02 destino
-rw-rw-r-- 1 diego diego 5459 abr 8 03:03 ssso_p2_100405879.zip
-rw-rw-r-- 1 diego diego 10688 abr 8 03:57 msh.o
-rwxrwxr-x 1 diego diego 22096 abr 8 03:57 msh
-rwxr-xr-x 1 diego diego 12700 mar 1 10:32 probador_ssso_p2.sh
-rwxr-xr-x 1 diego diego 21136 feb 28 13:47 libparser.so
total 124
MSH>>ls -l | sort | grep a | wc
   12     101     606
MSH>>
```

Como podemos comprobar con estas sencillas pruebas, las secuencias de mandatos también funcionan correctamente y no están limitadas a ningún número de pipes.

## Mandatos simples con redirecciones

Ejecución de mandatos simples con redirecciones:

```
MSH>>ls
autores.txt destino hola libparser.so Makefile msh msh.c msh.o origen probador_sssoo_p2.sh sssoo_p2_100405879.zip
MSH>>ls > ejemplo
MSH>>cat < ejemplo
autores.txt
destino
ejemplo
hola
libparser.so
Makefile
msh
msh.c
msh.o
origen
probador_sssoo_p2.sh
sssoo_p2_100405879.zip
MSH>>
```

Como podemos observar en la captura, vemos qué archivos hay en la carpeta actual mediante un “ls” y nos damos cuenta de que no existe ningún fichero llamado “ejemplo”.

Después al ejecutar “ls > ejemplo”, estamos redirigiendo la salida ls a un fichero llamado “ejemplo” que se crea al ejecutar el mandato, y podemos comprobar que esto es así volviendo a ver el contenido del fichero mediante un cat y redirigiendo la entrada del proceso de tal modo que la entrada de dicho proceso sea ahora el fichero “ejemplo”, y como esperábamos, imprime el resultado de la consulta “ls”

También podemos comprobar que podemos hacer dos redirecciones a la vez:

```
MSH>>cat < ejemplo > ejemplo2
MSH>>cat ejemplo2
autores.txt
destino
ejemplo
hola
libparser.so
Makefile
msh
msh.c
msh.o
origen
probador_sssoo_p2.sh
sssoo_p2_100405879.zip
MSH>>
```

Vemos que se redirige la entrada de cat al fichero “ejemplo”, y la salida del mismo a ejemplo2. Entonces ahora ejemplo 2 debería tener el contenido de “ejemplo” que es el contenido de ls.

Haciendo un cat de ejemplo2 nos damos cuenta de que ha funcionado correctamente y la salida es la que esperábamos.

### Redirección de salida de error

Por último, vamos a comprobar que las redirecciones de salida de error funcionan:

```
MSH>>gcc -g sdfcoifd.c
gcc: error: sdfcoifd.c: No existe el archivo o el directorio
gcc: fatal error: no input files
compilation terminated.
MSH>>gcc -g sdcoifd.c !> ejemploerror
MSH>>cat ejemploerror
gcc: error: sdcoifd.c: No existe el archivo o el directorio
gcc: fatal error: no input files
compilation terminated.
MSH>>
```

Para comprobarlo se me ha ocurrido intentar compilar un archivo que no existe. Primero, no hacemos ninguna redirección de la salida para ver cuál es el error que nos saca. Después ejecutamos ese mismo mandato redirigiendo al fichero de salida de error, y cuando hacemos un cat de ese fichero vemos que se nos imprime exactamente el mensaje de error que nos salía antes de hacer la redirección.

## Secuencias de mandatos con redirecciones

### Salida

El siguiente ejemplo muestra el correcto funcionamiento de una redirección de salida de una secuencia de mandatos. Mandamos la salida de la secuencia a "ejemplopipes" y después cuando mostramos el contenido del mismo tenemos la salida que esperábamos.

```
diego@diego-B450-AORUS-ELITE: ~/Escritorio/practica 2/mm/p2_minishell_2022
MSH>>ls -l | sort | grep ad > ejemplopipes
MSH>>cat ejemplopipes
-rwxr--r-- 1 diego diego 26 abr 8 04:20 adios
-rwxr-xr-x 1 diego diego 12700 mar 1 10:32 probador_ss00_p2.sh
MSH>>
```

## Entrada

Como podemos comprobar también funciona para redirecciones de entrada:

```
MSH>>ls -l > ejemploentrada
MSH>>cat | sort | grep ad < ejemploentrada
-rwxr--r-- 1 diego diego    0 abr  8 04:27 ejemploentrada
-rwxr--r-- 1 diego diego   26 abr  8 04:20 adios
-rwxr-xr-x 1 diego diego 12700 mar  1 10:32 probador_ssoo_p2.sh
MSH>>[]
```

Pasamos el mandato ls -l al fichero “ejemplo entrada”

Luego al ejecutar cat | sort | grep ad < ejemploentrada redirigiendo la entrada

del cat con dicho fichero debería ejecutarse lo mismo que si ejecutaramos:

ls -l | sort | grep ad, y como podemos comprobar así es.

## Salida de error en secuencias de mandatos:

```
MSH>>cat noexiste | grep a
cat: noexiste: No existe el archivo o el directorio
MSH>>cat noexiste | grep a !> pruebaerror
MSH>>cat pruebaerror
cat: noexiste: No existe el archivo o el directorio
MSH>>[]
```

Como podemos observar en la redirección, el mensaje de error de que no existe el fichero al que está intentando acceder el cat se almacena en el fichero de error que le indicamos

Por otro lado también podemos comprobar que el segundo mandato de la secuencia puede redirigir su salida de error:

```
MSH>>cat pruebaerror | grep
Usage: grep [OPTION]... PATTERN [FILE]...
Pruebe 'grep --help' para más información.
MSH>>cat pruebaerror | grep !> salidaaerror
MSH>>cat salidaaerror
Usage: grep [OPTION]... PATTERN [FILE]...
Pruebe 'grep --help' para más información.
MSH>>[]
```

Al no poner qué cadena estamos buscando al grep, este imprime un mensaje de error. Si ejecutamos esa misma secuencia redirigiendo la salida de error vemos que dicho mensaje se almacena correctamente en el fichero .

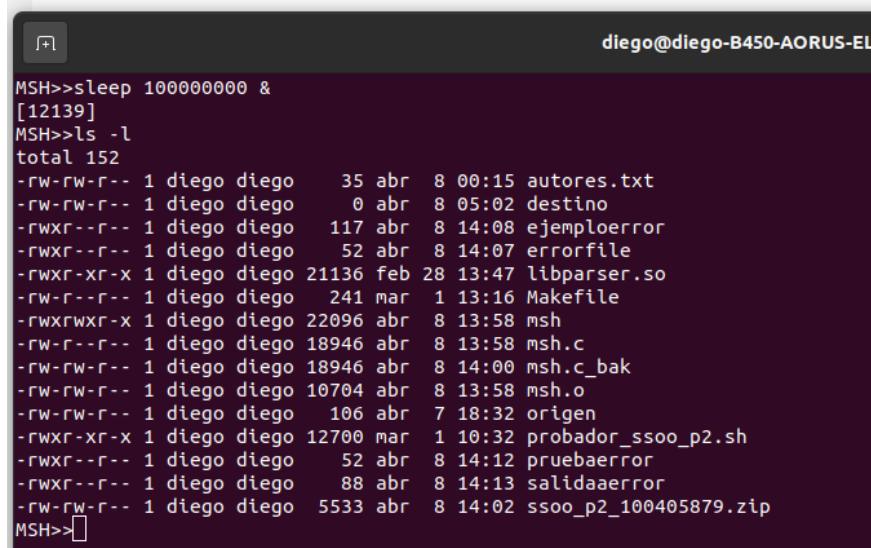
## Modo background (“&”)

Primero ejecutamos un sleep 100000000 &, donde se ve que el proceso por el que estamos esperando es el 12139. He puesto la hora a la que ejecuto el mandato para demostrar que luego he hecho un ls -l sin tener que esperar todo ese tiempo, aunque es evidente que no he esperado todo ese tiempo, por eso se ha elegido un número tan grande.



diego@diego-B450-AORUS-ELIT

```
MSH>>sleep 100000000 &
[12139]
MSH>>
```



diego@diego-B450-AORUS-EL

```
MSH>>sleep 100000000 &
[12139]
MSH>>ls -l
total 152
-rw-rw-r-- 1 diego diego    35 abr  8 00:15 autores.txt
-rw-rw-r-- 1 diego diego     0 abr  8 05:02 destino
-rwxr--r-- 1 diego diego   117 abr  8 14:08 ejemploerror
-rwxr--r-- 1 diego diego    52 abr  8 14:07 errorfile
-rwxr-xr-x 1 diego diego 21136 feb 28 13:47 libparser.so
-rw-r--r-- 1 diego diego   241 mar  1 13:16 Makefile
-rwxrwxr-x 1 diego diego 22096 abr  8 13:58 msh
-rw-r--r-- 1 diego diego 18946 abr  8 13:58 msh.c
-rw-rw-r-- 1 diego diego 18946 abr  8 14:00 msh.c_bak
-rw-rw-r-- 1 diego diego 10704 abr  8 13:58 msh.o
-rw-rw-r-- 1 diego diego   106 abr  7 18:32 origen
-rwxr-xr-x 1 diego diego 12700 mar  1 10:32 probador_ssoo_p2.sh
-rwxr--r-- 1 diego diego    52 abr  8 14:12 pruebaerror
-rwxr--r-- 1 diego diego    88 abr  8 14:13 salidaaerror
-rw-rw-r-- 1 diego diego  5533 abr  8 14:02 ssoo_p2_100405879.zip
MSH>>
```

Como podemos comprobar, el mandato sleep se ejecuta en background de tal modo que no hace falta esperar por él, y podemos ejecutar otros mandatos mientras, como es el caso del ls -l.

## Comprobación de la ejecución de mandatos internos

### mycalc

```
| MSH>>mycalc 3 add 4  
| [OK] 3 + 4 = 7; Acc 7  
MSH>>mycalc 4 add 10  
[OK] 4 + 10 = 14; Acc 21  
MSH>>mycalc -1 add 3  
[OK] -1 + 3 = 2; Acc 23  
MSH>>mycalc 3 add -5  
[OK] 3 + -5 = -2; Acc 21  
MSH>>mycalc -8 add 200  
[OK] -8 + 200 = 192; Acc 213  
MSH>>mycalc -13 add -200  
[OK] -13 + -200 = -213; Acc 0  
MSH>>|
```

Se puede hacer cualquier suma con cualquier combinación de números negativos y esta se va almacenando en la variable acc al mismo tiempo que da por pantalla un mensaje de éxito.

Si intentamos poner mal los argumentos:

```
| MSH>>mycalc  
| [ERROR] La estructura del comando es mycalc <operando_1> <add/mod> <operando_2>  
MSH>>mycalc 3  
| [ERROR] La estructura del comando es mycalc <operando_1> <add/mod> <operando_2>  
MSH>>mycalc 4 suma 4  
| [ERROR] La estructura del comando es mycalc <operando_1> <add/mod> <operando_2>  
MSH>>mycalc 3 add  
| [ERROR] La estructura del comando es mycalc <operando_1> <add/mod> <operando_2>  
MSH>>mycalc 8 mod sd  
| [ERROR] La estructura del comando es mycalc <operando_1> <add/mod> <operando_2>  
MSH>>|
```

Siempre se detecta que estos se han puesto mal, ya sea porque faltan argumentos o porque la sintaxis no es exactamente la que se requiere

mycalc con modulo:

```
| MSH>>mycalc 8 mod 2  
| [OK] 8 % 2 = 0; Cociente 4  
MSH>>mycalc 9 mod 4  
| [OK] 9 % 4 = 1; Cociente 2  
MSH>>|
```

## Mycp

Por último comprobamos el correcto funcionamiento del comando interno mycp:

```
MSH>>cat origen
ergesagrzergvdfgsat
wtgae3rg
eafbvs
eagergergsdfgdfgst
tw4
65w
tesa
gfs
defgse5rytws45y
dfhg
dser
hwe45
uyMSH>>cat destino
MSH>>mycp origen destino
[OK] Copiado con éxito el fichero origen a destinoMSH>>cat destino
ergesagrzergvdfgsat
wtgae3rg
eafbvs
eagergergsdfgdfgst
tw4
65w
tesa
gfs
defgse5rytws45y
dfhg
dser
hwe45
uyMSH>>
```

Vemos el contenido del fichero origen con cat. Después comprobamos el contenido del fichero destino con cat y vemos que está vacío.

Hacemos mycp origen destino. Volvemos a hacer cat de destino y vemos que se ha copiado el contenido del fichero origen en el fichero destino

Si metemos mal los argumentos o no existen los ficheros:

```
MSH>>mycp
[ERROR] La estructura del comando es mycp <fichero_origen><fichero_destino>
MSH>>mycp origen
[ERROR] La estructura del comando es mycp <fichero_origen><fichero_destino>
MSH>>mycp eaodfa aefa
[ERROR] Error al abrir el fichero origen
[ERROR] Error al abrir el fichero destino
MSH>>mycp origen eafaf
[ERROR] Error al abrir el fichero destino
MSH>>mycp aedfa destino
[ERROR] Error al abrir el fichero origen
MSH>>
```

Se detecta perfectamente cual es el problema, ya sea que falte algún argumento o haya errores al abrir cualquiera de los dos ficheros.

Por último aunque sea una prueba muy tonta, podemos comprobar que si no reconoce el mandato a ejecutar, no se ejecuta el exec y se imprime por la salida estándar de error el mensaje indicado por perror, para después hacer un exit(-1) y así finalizar la ejecución del proceso hijo

```
MSH>>ergedrg
Error en el exec: No such file or directory
MSH>>
```

## Conclusiones

El principal problema a la hora de realizar la práctica ha sido entender bien cómo funcionan las redirecciones mediante los descriptores de ficheros y pipes, y cómo crear secuencias no limitadas a ningún tipo de pipes.

El haber realizado la práctica yo solo me ha requerido mucho trabajo y no ha sido fácil, pero lo bueno es que he profundizado mucho en los conocimientos vistos en clase y no me ha requerido de tanto esfuerzo teniendo en cuenta que he disfrutado haciéndola al parecerme muy interesante el tener una idea más clara de cómo funciona un intérprete de mandatos.

Este trabajo lo veo muy acertado para aprender realmente en la práctica los conocimientos vistos en la clase de teoría.