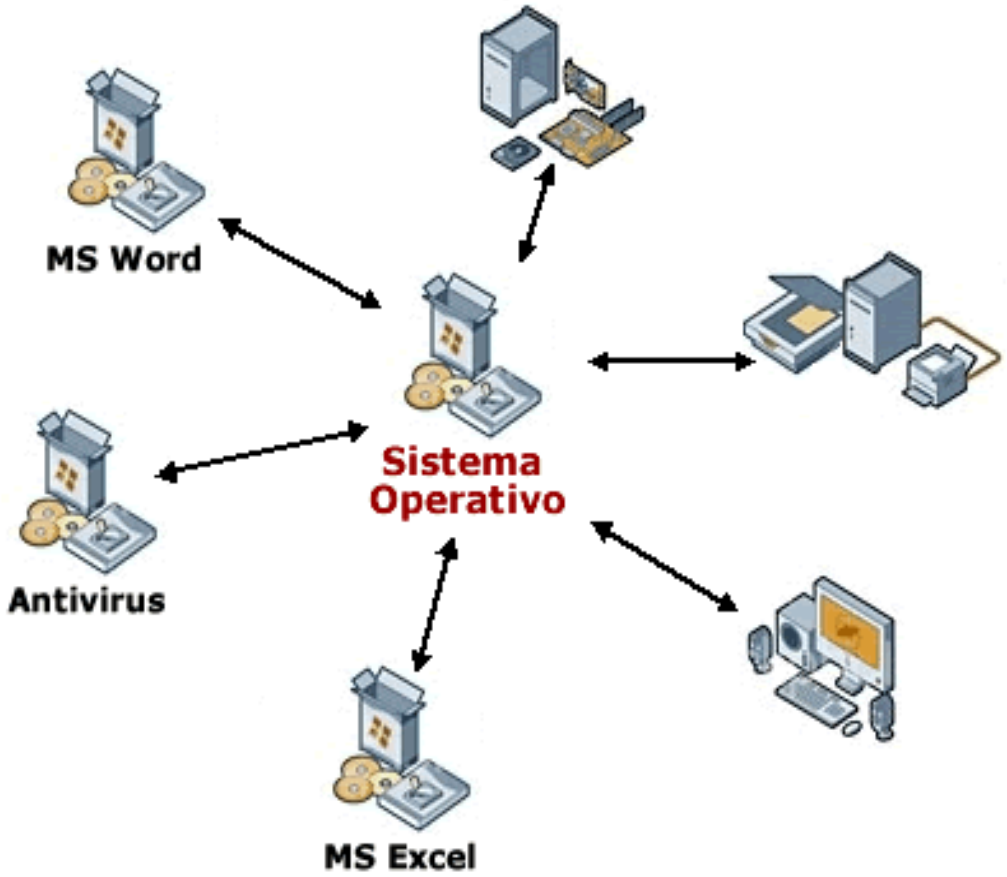
Titulación: GRADO INGENIERÍA INFORMÁTICA Y ADE
Año Académico: 2023/24 – 2º cuatrimestre
Asignatura: Sistemas Operativos (2º curso)
Memoria de Prácticas 2: Programación de un intérprete de mandatos (Minishell)

uc3m | Universidad Carlos III de Madrid

Alumno/a:	**Juan Marín Olloqui**	**NIA:**	**100475063**
Alumno/a:	**David Sierra Fernández**	**NIA:**	**100471913**
Alumno/a:	**Paulo Álvarez Da Costa**	**NIA:**	**100475757**



Titulación: GRADO INGENIERÍA INFORMÁTICA Y ADE
Año Académico: 2023/24 – 2º cuatrimestre
Asignatura: Sistemas Operativos (2º curso)
Memoria de Prácticas 2: Programación de un intérprete de mandatos (Minishell)

Índice de Contenidos

1. Descripción del código

1.1 Minishell

La minishell comienza con un bucle que recorre desde el primer proceso hasta el número total de procesos, almacenado en command_counter, consiguiendo así una implementación que no se limite a un número definido de procesos ni de pipes. Para cada proceso, se reserva memoria para su PID (identificador de proceso) y se crea un pipe para la comunicación entre dos procesos, es decir el número de pipes = número de procesos -1. Luego, se genera un proceso hijo con fork(). Si el proceso es el hijo, se comprueba si hay redirecciones de la salida estándar de error. Si las hay, se abre el fichero de error, se redirige la salida estándar de error al fichero filev[2] y se cierra el fichero. Si el proceso es el primero, se comprueba si hay redirecciones de entrada. Si las hay, se abre el fichero de entrada, se redirige la entrada estándar al fichero filev[0] y se cierra el fichero. Si el proceso es intermedio o final, se redirige la entrada al pipe anterior, se cierra la escritura del pipe anterior y se espera a que el proceso anterior termine usando waitpid(). Esta implementación permite un número ilimitado de procesos y pipes, y maneja correctamente las redirecciones de entrada y salida.

El código continúa con la gestión de los pipes y las redirecciones de salida estándar. Se cierra la lectura del pipe actual, ya que no se va a usar. Si el proceso es el último, se cierran ambos extremos del pipe actual, ya que el último comando no lo usa. Si hay una redirección de salida, se abre el fichero de salida, se redirige la salida estándar al fichero y se cierra el fichero. Si el proceso no es el último, se redirige la salida al pipe actual. Una vez que se han gestionado las redirecciones de los pipes y los ficheros, se ejecuta el comando con execvp(). Si hay algún error al ejecutar el comando, se imprime un mensaje de error y se termina el proceso con exit(-1).

Si el proceso es el padre, se guarda el PID del proceso hijo en el array de pids y se cierran los pipes que no se van a usar, que son los del proceso anterior. Si el proceso es el último, se cierran ambos extremos del pipe actual, ya que no se va a usar. Si la ejecución es en primer plano (in_background == 0), se espera a que termine el proceso hijo con wait(), y luego se espera a que terminen los procesos en segundo plano de secuencias de comandos anteriores con la función auxiliar wait_bg_processes(), que los recoge si han terminado para que no queden en estado de zombie y en caso contrario los espera. Si la ejecución es en segundo plano, se imprime el PID del proceso y se reasigna memoria para el array de procesos en segundo plano con realloc(), incrementando el número de procesos en segundo plano. Este código gestiona eficientemente las redirecciones de entrada y salida, así como la ejecución de comandos en primer y segundo plano.

El código finaliza con la gestión de los procesos en segundo plano. Si la ejecución es en segundo plano, se añaden los PIDs de esta secuencia a la lista de procesos en segundo plano empezando por el primer proceso. Esto se hace con un bucle `for` que recorre desde `j = 0` hasta `command_counter`, que es el número total de procesos. Para cada proceso, se añade su PID a la lista de procesos en segundo plano en la posición correspondiente. Este código asegura que todos los procesos en segundo plano se gestionan correctamente y se mantienen en la lista de procesos en segundo plano para su seguimiento.

Finalmente, después de todas las operaciones de la minishell, el programa retorna 0, indicando que ha terminado su ejecución de manera exitosa. Esta es una práctica común en programación para indicar que un programa ha terminado sin errores. En resumen, este código es una implementación completa y eficiente de una minishell en C, con soporte para un número ilimitado de procesos y pipes, redirecciones de entrada y salida, y ejecución de comandos en primer y segundo plano.

1.2 Mandatos internos

1.2.1 myCalc

La función `myCalc` es una calculadora básica que se ejecuta en la terminal. Esta función toma un array de cadenas de caracteres como argumento, que representa los argumentos proporcionados en la línea de comandos.

La función comienza inicializando varias variables locales. `acc` almacena el valor de la variable de entorno 'Acc', `acc_str` almacena el valor de 'Acc' en formato de cadena, `resultado` almacena el resultado de la operación y `resto` almacena el resto de la operación de división.

Luego, la función verifica que el comando tenga exactamente tres argumentos: operando_1, operador, operando_2. Si no es así, imprime un mensaje de error y retorna -1.

Después, verifica que los operandos sean números válidos. Si no lo son, imprime un mensaje de error y retorna -1.

A continuación, verifica que el operador sea una de las operaciones permitidas: add, mul, div. Si no lo es, imprime un mensaje de error y retorna -1.

Dependiendo del primer carácter del operador, selecciona la operación a realizar. Si el operador es 'add', realiza una suma. Si la variable de entorno 'Acc' no existe, se crea con valor 0. Luego, calcula el resultado de la suma y modifica 'Acc' sumándole el resultado. Finalmente, muestra el resultado por pantalla.

Si el operador es ‘mul’, realiza una multiplicación y muestra el resultado por pantalla.

Si el operador es ‘div’, realiza una división. Antes de hacerlo, verifica que el divisor no sea 0. Si lo es, imprime un mensaje de error y retorna -1. Luego, calcula el resultado y el resto de la división y muestra ambos por pantalla.

La función retorna 0 si la operación se realizó con éxito.

Además, hay una función auxiliar llamada `int_digits` que hemos creado para calcular el número de dígitos de un número entero. Esta función toma un número entero como argumento y retorna el número de dígitos en ese número. Esta función es útil para convertir el número ‘acc’ a una cadena en la operación de suma.

1.2.2 myhistory

La función myhistory no la hemos sacado como nos hubiera gustado, pero este es el objetivo que hemos tratado de alcanzar. Aunque no se ha visto reflejado, esta era la idea de nuestro codigo :

La función `myhistory` es una función que maneja el historial de comandos ingresados por el usuario. Esta función se activa cuando el primer argumento del array `argvv` es igual a “myhistory”.

Si no se proporciona ningún argumento adicional (es decir, `argvv[0][1]` es NULL), la función muestra el historial completo de comandos. Para cada comando en el historial, imprime el índice del comando, seguido de los argumentos del comando. Si el comando tiene múltiples partes (indicado por el número de comandos), separa cada parte con un “|”.

Si se proporciona un argumento adicional, la función intenta interpretarlo como un índice en el historial de comandos. Primero, convierte el argumento a un número entero largo con la función `strtol`. Si la conversión es exitosa y el índice está dentro del rango válido (es decir, mayor o igual a 0 y menor que `current_size`), procede a ejecutar el comando en ese índice del historial.

Si el comando en el historial es “mycalc”, lo ejecuta usando la función dedicada `mycalc`. Si el comando no es “mycalc”, lo ejecuta usando la función `aux_myhistory`. Antes de ejecutar `aux_myhistory`, prepara las tuberías necesarias para la comunicación entre los procesos y muestra un mensaje indicando que está ejecutando el comando.

Si el argumento adicional no es un índice válido en el historial de comandos, imprime un mensaje de error.

Después de manejar el comando “myhistory”, la función omite el resto del bucle principal para este comando con la instrucción `continue`.

La función `aux_myhistory` es una función auxiliar que se utiliza para procesar y ejecutar comandos. Esta función toma cinco argumentos: un puntero a un array de cadenas de caracteres `argvv` que representa los argumentos del comando, un contador de comandos `command_counter`, un array de descriptores de archivo `pipefds` para las tuberías, un número máximo de tuberías `max_pipes`, y un indicador `in_background` que determina si el comando debe ejecutarse en segundo plano.

La función comienza con un bucle que recorre cada comando en `argvv`. Para cada comando, crea un nuevo proceso utilizando la función `fork`.

Si el proceso es un proceso hijo (es decir, `fork` retorna 0), realiza varias tareas:

1. Redirige la entrada si hay una tubería anterior, o si hay un archivo de entrada para el primer comando.
2. Redirige la salida a la siguiente tubería, o al archivo de salida si es el último comando.
3. Cierra todas las tuberías en el proceso hijo.
4. Ejecuta el comando utilizando la función `execvp`.

Si el proceso es un proceso padre (es decir, `fork` retorna un número positivo), cierra todas las tuberías en el proceso padre.

Finalmente, si el comando no se está ejecutando en segundo plano (es decir, `in_background` es falso), la función espera a que todos los procesos hijos terminen utilizando la función `wait`.

En resumen, `aux_myhistory` es una función que maneja la ejecución de comandos, incluyendo la redirección de entrada/salida y la ejecución en segundo plano. Esta función es esencial para el funcionamiento de la shell.

En nuestro código hemos tratado de implementar esta estrategia pero por falta de tiempo, problemas de compilación y mejora de la nota del probador hemos optado por poner otro similar.

2. Batería de Pruebas

2.1 Minishell

2.1.1 Mandatos simples

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Lista el contenido del directorio actual	**ls**	autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh	autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh
Muestra el contenido del directorio actual en formato detallado	**ls -l**	total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh	total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh

Muestra el directorio actual	**pwd**	/home/loorezz/Escritorio/practica_2/p2_minishell_2024	/home/loorezz/Escritorio/practica_2/p2_minishell_2024
Muestra información sobre los procesos en ejecución	**ps**	PID TTY TIME CMD 4061 pts/0 00:00:00 bash 5752 pts/0 00:00:00 msh 5754 pts/0 00:00:00 ps	PID TTY TIME CMD 4061 pts/0 00:00:00 bash 5752 pts/0 00:00:00 msh 5754 pts/0 00:00:00 ps

Busca patrones en archivos	**grep myCalc msh.c**	int myCalc(char *argv[]);
		myCalc(argvv[0]);
		int myCalc(char *argv[]){
			int myCalc(char *argv[]);
			myCalc(argvv[0]);
			int myCalc(char *argv[]){

Muestra el contenido de un archivo	**cat msh.c | head**	//P2-SSOO-23/24
		// MSH main file
		// Write your msh source code here
		//#include <parser.h>
		#include <stddef.h>
		/* NULL */
		#include <sys/types.h>
		#include <sys/stat.h>
		#include <fcntl.h>
			//P2-SSOO-23/24
			// MSH main file
			// Write your msh source code here
			//#include <parser.h>
			#include <stddef.h>
			/* NULL */
			#include <sys/types.h>
			#include <sys/stat.h>
			#include <fcntl.h>

2.1.2 Mandatos simples en background

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Ver el proceso por el que estamos esperando. Se ejecuta en background por lo que podemos ejecutar otros mandatos mientras tanto	**sleep 50 &**	Proceso en background con pid [5796]	Proceso en background con pid [5796]
Ahora probamos a ejecutar un comando en segundo plano (background)	**sleep 50 &** **ls -l**	Proceso en background con pid [5948] MSH>>ls -l total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh	Proceso en background con pid [5948] MSH>>ls -l total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh

Aquí comprobamos que después de usar un sleep en background, y ejecutar tres comandos después, se espera a a la finalización de cada proceso para ejecutar el siguiente	**sleep 50 &** **ls** **who** **ls -l**	Proceso en background con pid [37416] MSH>>ls autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh ls who ls -l MSH>>autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh MSH>>loorezz tty2 2024-04-16 13:21 (tty2) loorezz pts/1 2024-04-16 19:07 MSH>>total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh	Proceso en background con pid [37416] MSH>>ls autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh ls who ls -l MSH>>autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh MSH>>loorezz tty2 2024-04-16 13:21 (tty2) loorezz pts/1 2024-04-16 19:07 MSH>>total 136 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rwxrwxr-x 1 loorezz loorezz 30192 abr 16 13:37 msh -rw-rw-r-- 1 loorezz loorezz 21531 abr 16 13:37 msh.c -rw-rw-r-- 1 loorezz loorezz 29016 abr 16 13:37 msh.o -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh

2.1.3 Secuencias de mandatos conectados por pipes

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Busca archivos en un directorio y los ordena por tamaño	**ls -l | sort -k 5 -n**	total 156 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 68 abr 16 22:18 ex1 -rw-rw-r-- 1 loorezz loorezz 73 abr 16 21:43 ex -rw-rw-r-- 1 loorezz loorezz 73 abr 16 22:15 ej -rw-rw-r-- 1 loorezz loorezz 86 abr 16 21:55 error.txt -rw-rw-r-- 1 loorezz loorezz 101 abr 16 22:25 error -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rw-rw-r-- 1 loorezz loorezz 680 abr 16 22:20 ex2 -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh -rw-rw-r-- 1 loorezz loorezz 19872 abr 16 21:37 msh.c	total 156 -rw-rw-r-- 1 loorezz loorezz 65 feb 21 16:27 autores.txt -rw-rw-r-- 1 loorezz loorezz 68 abr 16 22:18 ex1 -rw-rw-r-- 1 loorezz loorezz 73 abr 16 21:43 ex -rw-rw-r-- 1 loorezz loorezz 73 abr 16 22:15 ej -rw-rw-r-- 1 loorezz loorezz 86 abr 16 21:55 error.txt -rw-rw-r-- 1 loorezz loorezz 101 abr 16 22:25 error -rw-rw-r-- 1 loorezz loorezz 244 feb 21 16:27 Makefile -rw-rw-r-- 1 loorezz loorezz 680 abr 16 22:20 ex2 -rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh -rw-rw-r-- 1 loorezz loorezz 19872 abr 16 21:37 msh.c -rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 29464 abr 16 21:37 msh.o -rwxrwxr-x 1 loorezz loorezz 30344 abr 16 21:37 msh

		-rw-rw-r-- 1 loorezz loorezz 20696 feb 21 16:27 libparser.so -rw-rw-r-- 1 loorezz loorezz 29464 abr 16 21:37 msh.o -rwxrwxr-x 1 loorezz loorezz 30344 abr 16 21:37 msh	
Cuenta el número total de archivos y directorios en el directorio actual mostrando la lista detallada	**ls -l | wc**	14 119 734	14 119 734
Cuenta el número total de archivos y directorios en el directorio actual ordenándolos alfabéticamente antes de contarlos.	**ls -l | sort | wc**	14 119 734	14 119 734
Prueba complementaria	**ls | sort | wc | wc -l**	1	1

2.1.4 mandatos simples y secuencias de mandatos con redirecciones (entrada, salida y de error) y en background

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Mandatos simples con redirecciones	**MSH>> ls > ex** **MSH >> cat < ex**	autores.txt ex libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh	autores.txt ex libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh
Redirección de salida de error	**MSH >> gcc -g xyz.c !> error.txt** **MSH >> cat error.txt**	fatal error: xyz.c: No existe el archivo o el directorio compilation terminated.	fatal error: xyz.c: No existe el archivo o el directorio compilation terminated.
Dos redirecciones a la vez	**MSH >> cat < ex > ej** **MSH >> cat ej**	autores.txt ex libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh	autores.txt ex libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh
Secuencias de mandatos con redirecciones: SALIDA → ex1	**MSH>> ls -l | sort | grep ad > ex1** **MSH >> cat ex1**	-rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh	-rw-rw-r-- 1 loorezz loorezz 12527 feb 22 13:28 probador_ssoo_p2.sh
Secuencias de mandatos con redirecciones:	**MSH>> ls -l > ex2**	-rw-rw-r-- 1 loorezz loorezz 12527 feb 22	-rw-rw-r-- 1 loorezz loorezz 12527 feb 22

ls -l → ENTRADA	**MSH >> cat | sort | grep ad < ex2**	13:28 probador_ssoo_p2.sh	13:28 probador_ssoo_p2.sh
Secuencias de mandatos con redirecciones: ERROR	**MSH >> cat ex3 | grep a** **MSH >> cat ex3 | grep a !> error** **MSH >>cat error** **Nota: ex3 no exite**	cat: ex3: No existe el archivo o el directorio cat: ex3: No existe el archivo o el directorio	cat: ex3: No existe el archivo o el directorio cat: ex3: No existe el archivo o el directorio
Secuencias de mandatos con redirecciones: REDIRECCIÓN SALIDA ERROR	**MSH >> cat error | grep** **MSH >> cat error | grep !> error1** **MSH >>cat error1**	Modo de empleo: grep [OPCIÓN]... PATRONES [FICHERO]... Pruebe 'grep --help' para más información.	Modo de empleo: grep [OPCIÓN]... PATRONES [FICHERO]... Pruebe 'grep --help' para más información.

2.2 Mandatos internos

2.2.1 myCalc

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Prueba de que mycalc funciona I	**mycalc 5 add 3**	[OK] 5 + 3 = 8; Acc 8	[OK] 5 + 3 = 8; Acc 8
Prueba de que mycalc funciona II. Prueba de que la variable acumuladora funciona	**mycalc 3 add 3**	[OK] 3 + 3 = 6; Acc 14	[OK] 3 + 3 = 6; Acc 14

Prueba de que multiplicación funciona	**mycalc 2 mul 7**	[OK] 2 * 7 = 14	[OK] 2 * 7 = 14
División entre 0	**mycalc 7 div 0**	[ERROR] Division por 0	[ERROR] Division por 0
Prueba de que la división funciona	**mycalc 7 div 3**	[OK] 7 / 3 = 2; Resto 1	[OK] 7 / 3 = 2; Resto 1
Prueba que va a devolver ERROR por formato erróneo	**mycalc 1 mul a**	[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>	[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>

2.2.2 myHistory

Nota: Asumiendo que nuestro historial de comandos es: ls, ls | grep a, ls | grep b &, ls | ls -l

Descripción	Uso	Resultado esperado (descripción)	Resultado obtenido
Prueba myhistory sin argumentos	**myhistory**	0 ls 1 ls | grep a 2 ls | grep b & 3 ls | ls -l	

Ejecuta el comando 0 del historial	**myhistory 0**	Ejecutando el comando 0 autores.txt libparser.so Makefile msh msh.c msh.o probador_ssoo_p2.sh	ERROR: Comando no encontrado
Comando fuera de rango	**myhistory 27**	ERROR: Comando no encontrado	ERROR: Comando no encontrado
Prueba ERROR por formato erróneo	**myhistory a**	[ERROR] La estructura del comando es myhistory o myhistory <nº comando>	[ERROR] La estructura del comando es myhistory o myhistory <nº comando>

3. Conclusiones

Está práctica ha sido exageradamente extensa. Por otra parte, la cantidad de ayudas que nos han proporcionado junto con las clases teórico-prácticas hacen la práctica mucho más llevadera. La cantidad de información que hay que comprender para la minishell es lo que la hace complicada pero una vez asimilados los conceptos no nos ha resultado tan complicada. En cuanto a los mandatos internos, mycalc ha sido sin duda, la parte más asequible, y el myhistory un auténtico quebradero de cabeza. En nuestra opinión, hemos comprendido al 100% el funcionamiento del historial de comandos pero no lo hemos sabido plasmar con código.
Estamos bastante contentos con el resultado. Creemos que hemos entendido la práctica y el probador, aunque no del todo, nos ha dado la razón. A pesar de haber pasado unos días muy duros, creemos que el sudor y el tiempo empleado han merecido totalmente la pena.