PRÁCTICA 2 –

Programación de un intérprete de mandatos (minishell)

Autores:

- Guillermo Rubio Bolger (NIA: 100487574).
- Jaime Rabazo Fernández (NIA: 100499019).
- Daniele Laporta (NIA: 100502703).

Grupo de clase: Grupo 801.

Titulación cursada: Doble Grado en Ingeniería Informática y Administración de Empresas.

Tabla de contenido

1. Introducción 3

2. Descripción del código 3

2.1 Redirección de error 3

2.2 Mandatos Internos 3

2.3 Pipes y mandatos externos 5

3. Batería de pruebas 6

msh.c 6

mycalc 6

mytime 9

4. Conclusiones 12

1. Introducción

El objetivo principal de esta práctica es que nos familiaricemos con el funcionamiento de un pequeño intérprete de mandatos. Para ello, se nos han propuesto pruebas de ejecución; redireccionamiento de entrada, salida y de error; y comunicación entre procesos. En este documento se van a exponer y explicar las maneras de resolución implementadas por los integrantes del grupo para los enunciados propuestos en la práctica.

2. Descripción del código

En este apartado se van a detallar el funcionamiento de la minishell, se describirá de forma lineal el código, es decir, de arriba abajo, siguiendo el tiempo de ejecución.

2.1 Redirección de error.

Lo primero que hace nuestro código es comprobar si se ha especificado una redirección de error, revisando el array *filev* proporcionado por *read_command*. Si es así, se cierra el descriptor de fichero de salida de error estándar y se sustituye por el especificado. Este es el primer paso de nuestro código, ya que, si se produce algún error, va a ser después de aplicar la salida de error señalada, con el objetivo de escribir los posibles errores futuros de la minishell, en la salida especificada.

2.2 Mandatos Internos

En siguiente lugar, tras comprobar que el número de comandos introducidos es válido, se comprueba si el mandato recibido es uno interno de la minishell. Se distinguen dos mandatos internos:

- Si se solicita mycalc, se crea la variable de entorno Acc y se le asigna el valor 0. Se comprueba el argumento de operación, que debería estar en *argvv[0][2]*, y el número de argumentos pasados, que debería ser tres. Si se pasan estas comprobaciones, se distingue mediante un *if-else* cada operación: *"add", "mul" y "div".*

  En cada una de estas operaciones, se leen los argumentos pasados en forma de *string*, se transforman a enteros, se operan cómo especificados, se vuelven a transformar a *string*, y se devuelven en un *string*, con el formato especificado en el enunciado de la práctica, en la salida estándar de error. Además de esto, la operación *add* extrae la variable de entorno *"Acc"*, le suma el resultado de la operación, y la vuelve a guardar asegurándose de almacenar el resultado acumulado, también la imprime tras la operación siguiendo las instrucciones del enunciado.
- Si se solicita *mytime*, se crea un string resultado de 23 bytes, ya que el formato de impresión indicado es: *XX:YY:ZZ\0*, donde las variables son *longs*. Se declaran variables *long* sin signo para guardar los segundos, minutos, y horas que el programa lleva ejecutándose, y se hacen las operaciones correspondientes para pasar de milisegundos dados por la variable mytime, a segundos, minutos y horas, asegurándose de que minutos y horas se mantengan en módulo 60.

  Finalmente, se llevan a cabo varias sentencias condicionales para asegurarse de que el formato de impresión se mantiene con dos cifras, incluso cuando el resultado de segundos o minutos presenta una sola cifra, y se imprime por la salida de error, cómo se indicó en el enunciado.

Si no se pasan las pruebas de formato de mandato, es decir, entra un mandato inválido, se imprime un mensaje de error indicando el formato adecuado para el uso de estos mandatos internos, mediante una sentencia condicional.

Es importante denotar que no se ha utilizado memoria dinámica para las variables *String* de retorno de los mandatos internos. En su lugar, se ha determinado el tamaño máximo que estas variables podrían ocupar y se ha reservado esa cantidad de memoria para almacenar los resultados de las cadenas de texto.

2.3 Pipes y mandatos externos

En el caso de que no se haya solicitado ningún mandato interno a la minishell, y no se haya violado ninguna condición de formato hasta ahora (número de mandatos), llegamos a la sección dedicada a los *pipes* y *mandatos externos*. Tras la declaración de las variables necesarias comienza una sentencia iterativa que recorre todos los mandatos de la tubería de procesos. Para cada mandato, hace lo siguiente:

1. Si no es el último mandato utiliza un array previamente declarado para crear un nuevo pipe.
2. Se hace un fork.

Hijo

3. Si no es el primero mandato de la tubería de procesos, redirige su entrada al descriptor de fichero de lectura del pipe creado por el proceso del mandato anterior, que esta guardado en una variable compartida por los procesos.

4. Si es el primero, comprueba a través de los datos modificados por *read_command*, si se ha indicado una redirección de entrada. Si es así, redirige su entrada estándar al descriptor de fichero proporcionado.

5. Si no es el último redirige su salida al descriptor de fichero de escritura del pipe creado por este mismo proceso, y tras ello lleva a cabo la limpieza de los descriptores de fichero abiertos por el procedimiento *pipe()*.

6. Si es el último, comprueba a través de los datos modificados por *read_command*, si se ha indicado una redirección de salida. Si es así, redirige su salida estándar al descriptor de fichero proporcionado.

7. Tras las redirecciones se ejecuta el mandato correspondiente.

Padre

3. Si el *fork* no es del primer mandato de la tubería, cierra el descriptor de lectura del pipe anterior, guardado en una variable compartida por los procesos.

4. Si el *fork* no es del último mandato de la tubería, se cierra el descriptor de escritura del mandato actual, y se cierra el descriptor de lectura del mandato anterior guardado en una variable compartida por los procesos.

5. Si no es el último mandato, itera.

6. Tras iterar con todos los mandatos del pipe, si las variables editadas por *read_command* no indican que se ha de ejecutar en *background*. Se espera a finalizar el último proceso.

Es importante denotar que, en esta sección del código, los pipes están funcionando cómo variable compartida para asegurar la concurrencia correcta de los procesos, y que, en cada llamada a funciones con retorno, se comprueba si ha habido un error, al igual que con *execvp*.

3. Batería de pruebas

msh.c				
ID	**Datos para introducir:**	**Objetivos de la prueba:**	**Resultados esperados:**	**Resultados obtenidos:**
01	Se introducen más mandatos permitidos por la variable MAX_COMMANDS (8 en el caso de esta práctica).	Levantamiento del error del número máximo de comandos.	Levantamiento e impresión por pantalla de “`Error: Número máximo de comandos es %d \n`”	Levantamiento e impresión por pantalla de “`Error: Número máximo de comandos es %d \n`”
02	Se introduce el siguiente comando: “`ls -l | grep 7 | grep feb |`”	Se quiere comprobar el funcionamiento del programa con más de tres pipes. Además, se quiere escribir en un fichero la	La escritura en un fichero de salida especificado el resultado de haber aplicado todos los mandatos del	Se crea un fichero con el nombre “`output.txt`” en el que se escribe la salida de los anteriores

	`grep a | grep .txt | grep 2 > output.txt"`	salida de dicho comando.	comando en el orden especificado gracias al funcionamiento de las tuberías.	mandatos de manera correcta.
03	Se introduce el siguiente comando: "`ls | sort < input.txt`"	Se quiere comprobar el funcionamiento del redireccionamiento de entrada. Para ello, creamos un fichero `input.txt` con una lista de números desordenados.	Se espera la impresión por pantalla de los números ordenados y la lista de directorios del directorio actual.	Se imprimen por pantalla los números ordenados y la lista de directorios del directorio actual.
04	Se introduce el siguiente comando "`<cualquier cadena de caracteres inválida> !> error.txt`"	Se quiere comprobar el funcionamiento del redireccionamiento de salida de error. Para ello, introducimos un comando no válido y redirigimos la salida de error a un fichero llamado `error.txt` .	Se espera la creación de un fichero llamado `error.txt` con el error provocado en la minishell.	Se crea un fichero llamado `error.txt` con el error provocado en la minishell.
05	Se introduce el siguiente comando "`ls -l | grep a | gep c`"	Se quiere comprobar que se lanza el error "`Error al ejecutar`" cuando se alcanza el mandato `gep c` dado que dicho	Se espera el levantamiento del error cuando se alcanza la ejecución en el hijo del mandato `gep c`, además de la terminación	Se levanta el error cuando se alcanza la ejecución en el hijo del mandato `gep c`, además de la terminación del comando actual.

		mandato no es válido.	del comando actual.	
06	Se introduce el siguiente comando “`ls -l | grep a &`”	Se quiere comprobar que funciona la ejecución de mandatos en background. Una manera de comprobar que esto se cumple es listando la lista de directorios de un directorio con un filtrado. Si no sale de manera ordenada, es decir que el prompt sale antes de que termine la ejecución del comando, el mandato ha sido ejecutado en background.	Se espera un listado de directorios que contengan la letra ‘a’, pero el prompt debe salir antes de la terminación de la ejecución del mandato.	Se listan los directorios que contienen la letra ‘a’, pero el prompt sale antes de la terminación de la ejecución del mandato.
07	Se introduce el siguiente comando “`ls -l | grep 7 | grep abr | grep a | grep .pdf | grep 2 > output.txt &`”	Se quiere comprobar que funciona la ejecución de mandatos en background con más de tres pipes. Si no sale de manera ordenada, es decir que el prompt sale antes de que termine la ejecución del comando, el mandato ha sido ejecutado en background.	La escritura en un fichero de salida especificado el resultado de haber aplicado todos los mandatos del comando en background.	Se crea un fichero con el nombre “`output.txt`” en el que se escribe la salida de los anteriores mandatos de manera correcta.

mycalc				
ID	**Datos para introducir:**	**Objetivos de la prueba:**	**Resultados esperados:**	**Resultados obtenidos:**
08	Se introduce el comando `mycalc <op1> add <op2>`	Se quiere comprobar que ambos operandos se suman. Se necesita comprobar, también, que en la variable de entorno "Acc" se almacena el resultado de la suma actual más el último valor almacenado en dicha variable, resultado de posibles anteriores sumas. El resultado se debe imprimir por pantalla a través de la salida estándar del error, con un formato específico indicado en el enunciado.	Un string con el formato "`[OK] %s + %s = %d; Acc %s\n`" mostrando el resultado de la suma y el valor almacenado en la variable de entorno "Acc".	Un string con el formato "`[OK] %s + %s = %d; Acc %s\n`" mostrando el resultado de la suma y el valor almacenado en la variable de entorno "Acc".
09	Se introduce el comando `mycalc <op1> mul <op2>`	Se quiere comprobar que ambos operandos se multiplican. El resultado se debe imprimir por pantalla a través de la salida estándar del error, con un formato específico	Un string con el formato "`[OK] %s * %s = %d\n`" mostrando el resultado de la multiplicación.	Un string con el formato "`[OK] %s * %s = %d\n`" mostrando el resultado de la multiplicación.

		indicado en el enunciado. El resultado se debe imprimir por pantalla a través de la salida estándar del error, con un formato específico indicado en el enunciado.		
10	Se introduce el comando `mycalc <op1> div <op2>`	Se quiere comprobar que ambos operandos se dividen. Se necesita comprobar, también, que en la se muestra el resultado de la división correctamente. El resultado se debe imprimir por pantalla a través de la salida estándar del error, con un formato específico indicado en el enunciado.	Un string con el formato "`[OK] %s / %s = %d; Resto %d\n`" mostrando el resultado de la división.	Un string con el formato "`[OK] %s / %s = %d; Resto %d\n`" mostrando el resultado de la división.
11	Se introduce el comando `mycalc <cualquier cadena de caracteres que no siga el formato indicado>`	Se quiere comprobar que se levanta el error en el que se indica el formato correcto del uso del mandato interno. . El resultado se debe imprimir por pantalla a través	Un string con el formato "`[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>`"	Un string con el formato "`[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>`"

		de la salida estándar, con un formato específico indicado en el enunciado.		
mytime				
Id:	**Datos para introducir:**	**Objetivos de la prueba:**	**Resultados esperados:**	**Resultados obtenidos:**
12	Se introduce el comando `mytime`	Se quiere comprobar que se imprime por pantalla el tiempo que lleva ejecutando la minishell, en el formato HH:MM:SS, por la salida estándar de error.	Se imprime por pantalla el tiempo que lleva ejecutando la minishell en el formato HH:MM:SS.	Se imprime por pantalla el tiempo que lleva ejecutando la minishell en el formato HH:MM:SS.

4. Conclusiones

A diferencia de la primera práctica, teníamos la sensación inicial de que esta práctica iba a ser realmente desafiante. Sin embargo, una vez que nos pusimos a programar, y con el apoyo de los apuntes de teoría, nos dimos cuenta de que la implementación de esta minishell podía ser medianamente sencilla con un poco de orden y entendimiento de la teoría.

Comenzamos la implementación tal y como se nos recomendó en el enunciado:

En primer lugar, nos centramos en la ejecución de mandatos simples, como un ls -l. Queríamos entender primeramente la comunicación y el funcionamiento del proceso padre e hijo. En el siguiente paso, nos centramos en la ejecución de mandatos simples en background, lo cual no supuso mucha dificultad.

Posteriormente, nos centramos en los pipes. Entendimos la teoría tras varias revisiones, elaborando nuestros propios diagramas de funcionamiento de los pipes y los procedimientos de redireccionamiento y limpieza. Comenzamos implementando una tubería para dos procesos, y posteriormente, dos tuberías para tres. Pero queríamos retarnos y decidimos implementar la creación de tuberías para un número indeterminado de procesos hijos (realmente, para tantos comandos se escriban en la minishell). Estuvimos varias horas desarrollando esta parte, sobre todo porque había ocasiones en las que funcionaba, y otras, que no. Acabó siendo un fallo de orden en el código.

Finalmente, desarrollamos los mandatos internos mycalc y mytime. Ninguno de los dos nos supuso mayor dificultad, aunque el comando mytime sí que nos dio varios problemas en un principio, pero fue por nuestro vago recuerdo de la aritmética básica de secundaria.

En definitiva, esta práctica nos ha dotado de un firme entendimiento práctico del funcionamiento de las tuberías, las redirecciones y la ejecución de procesos en background, además de una profundización en el lenguaje C. La hemos considerado de gran utilidad para entender el contenido teórico de la asignatura.