PRÁCTICA 2: PROGAMACIÓN DE UN MINISHELL

SISTEMAS OPERATIVOS

PABLO AMOR MOLINA	100495855
SERGIO VERDE LLORENTE	100495899
HUGO CUEVAS ROMERA	100495962

16 DE MARZO DE 2024

UNIVERSIDAD CARLOS III DE MADRID

ÍNDICE

Descripción

Pipes y redirección

Para realizar la función de pipes, lo primero que se hace en la implementación es evaluar el número de comandos introducidos en la misma línea. En caso de ser más de 1, se crean tantos pipes como comandos haya menos 1 (ls | wc | grep 5 -> 3 comandos y 2 pipes).

Posteriormente, una vez se ha hecho el fork, para el caso del hijo, se dan 3 situaciones en función del número de comando que se esté procesando:

- Si es el primero de todos, se modifica el fichero de salida por el del pipe que se ha creado previamente. Además, si hay redirección del fichero de entrada este se modifica para cambiarlo por el introducido.

- Si es el último de todos, se modifica el fichero de entrada por el del pipe que se ha creado previamente. Además, si hay redirección del fichero de salida este se modifica para cambiarlo por el introducido.

- Si es un comando de posición genérica, se modifican tanto los ficheros de entrada y salida por los de los pipes que se han creado previamente.

En el caso del padre, en relación con los pipes, es importante decir que se cierra el pipe con el índice anterior (menos en el caso de que sea el primer comando (0)), para así evitar diversos errores.

Por otro lado, en caso de que se haya especificado también la redirección de la salida de error se ejecutará en primera instancia la redirección de este de forma análoga a la redirección de salida. Este descriptor de fichero siempre habrá que cerrarlo antes de devolver un código de error, para que no quede permanentemente abierto para el terminal, y se vea afectada la salida de error.

Mycalc

Para la función mycalc pasamos los argumentos del comando como parámetros, en donde tendremos los operandos y operador para realizar la operación, además de una variable de entorno Acc que usaremos en la opción de suma de la calculadora.

Primero declaramos dos variables auxiliares de tipo entero, ‘operando_1’ y ‘operando_2’, y las inicializamos con los datos guardados en el primer y tercer argumento respectivamente, todo ello tras haberlos cambiado de tipo string a entero con la función ‘atoi’ para realizar correctamente la operación.

Tras la declaración e inicialización de estas dos variables, declararemos una tercera que servirá como el operador a utilizar en la calculadora. Inicializamos dicha variable con el segundo argumento del comando y, finalmente, declaramos una cuarta variable que guardará el resultado de la operación realizada.

Una vez hayamos completado todos los pasos anteriores, comenzaremos a comparar el operador con las distintas opciones disponibles en la calculadora: *add*, *mul* o *div*; y actuar de las siguientes maneras según el caso que sea:

ADD: En el caso de la suma (o resta), primero obtenemos la variable Acc del entorno, modificamos su tipo a entero y la guardamos. Después hacemos la suma entre los dos operandos y el resultado se lo agregamos a la variable Acc. Declaramos el buffer de 100 bytes, imprimimos la suma, resultado y Acc por la salida estándar de error, volvemos a modificar el tipo de la variable Acc a tipo char y finalmente guardamos el valor guardado en Acc a su correspondiente variable de entorno del sistema.

MUL: En el caso de la multiplicación, tan solo realizamos la operación entre los dos operandos y guardamos el resultado en la variable correspondiente. Posteriormente, imprimimos la multiplicación y resultado por la salida estándar de error.

DIV: Al igual que en caso anterior, en la división tan solo realizamos la operación entre los dos operandos y guardamos el resultado en la variable correspondiente. Finalmente, imprimimos la división y resultado por la salida estándar de error.

Si se da el caso final en el que ninguna de las situaciones anteriores se haya dado por algún tipo de error en el operador, devolveremos un mensaje de error advirtiendo del fallo en la estructura del comando, todo ello por la salida estándar.

Myhistory

Para la función myhistory el primer pasa una vez hemos obtenido el myhistory como comando en la terminal es analizar su argumento en la posición argvv[i][1] ya que en esta posición se guardará el parámetro si lo tiene. Si observamos que en esa posición hay un NULL guardado tendremos que realizar el comando para imprimir el historial.

Para imprimir el historial tenemos que previamente haber guardado en una lista en este caso de 20 posiciones los comandos anteriormente realizados, para ello tenemos la lista history. Para añadir un elemento a la lista usaremos store_comand función proporcionada en el código del profesor y además tendremos dos punteros de pila para saber en qué posición de memoria estamos. Uno de estos punteros será el head que cuando haya menos de 20 elementos (pila parcialmente vacía) será estático y estará apuntando a el primer espacio de memoria reservado para la pila history. Además del puntero head encontraremos el puntero tail que ira apuntando a la posición del último elemento de la pila y así al sumarle 1 apuntara a la posición donde hay que almacenar el siguiente comando.

Este funcionamiento de los punteros se realiza para cuando la pila esta parcialmente vacía. Cuando la pila se llena totalmente el puntero head dejara de ser fijo e ira moviéndose a través de la pila ya que desde ahora en adelante tendremos que guardar los elementos en la esa posición porque será donde se encuentren los elementos más antiguos. Por otro lado, el puntero tail también avanzara por la pila para mantener la distancia de 20 posiciones con el head.

Una vez history está llena con los comandos ya realizados con anterioridad debemos imprimirlos para ello tendremos que recorrer la lista entera con varios bucles. Primero haremos un bucle que imprimirá la posición de la pila e iterará entre los distintos elementos o líneas de comando realizados. Posteriormente tendremos otro bucle que iterara en el

número de comandos de cada línea y además habrá otro que itere en el número de argumentos de cada comando. Dentro de estos bucles se realizará la impresión de las líneas de comandos de history.

El otro caso que necesitamos comprobar es si myhistory lleva un numero detrás ya que el comando realizara otra tarea distinta. En este caso el comando ejecutara la línea de comandos que se encuentre en la posición de history del número dado. Para ello primero comprobaremos si el numero esta entre 0 y 19 ya que son las posiciones de memoria que tenemos y además si en esa posición hay algo que no sea NULL. En el caso de que el número no cumpla esas condiciones lanzaremos un mensaje de error por la salida estándar.

Si el numero está dentro de los límites y tiene un comando dentro lo que haremos será sacar ese comando de history y meterlo en argvv. Esto lo haremos con dos bucles que iterarán en la posición history[i] siendo i el numero dado como argumento y dentro de este accederán a el argvv del comando.

Una vez el comando con todos sus argumentos este en el argvv es importante también cambiar la variable run_history proporcionada por el profesor a 1, ya que así podremos cuando acabemos el comando myhistory se ejecutará el comando que este en argvv sin pasar por el parser. De esta forma sea el comando que sea el que haya en la posición dada de history se ejecutara de forma normal y acabara su ejecución.

Batería de Pruebas

Pruebas ‘Comandos externos’

Prueba	Descripción	Salida	Salida modelo (terminal linux)
who	Ejecución de un comando único	pamor pts/1 2024-04-16 20:37	pamor pts/1 2024-04-16 20:37
who -l -b	Ejecución de un comando con varios argumentos	system boot 2024-04-16 20:37 LOGIN console 2024-04-16 20:37 235 id=cons LOGIN tty1 2024-04-16 20:37 237 id=tty1	system boot 2024-04-16 20:37 LOGIN console 2024-04-16 20:37 235 id=cons LOGIN tty1 2024-04-16 20:37 237 id=tty1
who &	Ejecución de un comando único con background	23099 MSH>>pamor pts/1 2024-04-16 20:37 (salida intercalada con el minishell)	[1] 23292 pamor pts/1 2024-04-16 20:37 [1]+ Done who
ls | wc	Ejecución de línea de comandos con 1 pipe	9 9 94	9 9 94
ls – l | sort | grep msh | wc	Ejecución de línea de comandos y argumentos con varios pipes	4 36 214	4 36 214
ls | wc &	Pipe con background	5702 MSH>> 9 9 94 (salida intercalada con el minishell)	6465 9 9 94
wc < autores.txt	Ejecución de mandato con redirección de entrada	2 12 94	2 12 94
wc < autores.txt > wcautores.txt	Ejecución de mandato con redirección de entrada y salida, creando el fichero	2 12 94 (dentro de wcautores.txt)	2 12 94 (dentro de wcautores.txt)
grep Pablo | wc < autores.txt > wcautores.txt	Ejecución de pipe con redirecciones, truncando sobre el fichero salida anterior	1 4 30 (dentro de wcautores.txt)	1 4 30 (dentro de wcautores.txt)

wc < ERROR !> error.txt	Prueba de la salida de error por fichero de entrada incorrecto	Error de entrada: No such file or directory (dentro de error.txt)	Error de entrada: No such file or directory (dentro de error.txt)
ERROOOR !> error.txt	Prueba de la salida de error por instrucción incorrecta	Error al ejecutar la instrucción: No such file or directory (dentro de error.txt)	Error al ejecutar la instrucción: No such file or directory (dentro de error.txt)
who !> error2.txt	Prueba de salida de error sin error (creando un nuevo fichero)	<fichero vacío> (dentro de error2.txt) pamor pts/1 2024-04-16 20:37 (salida estándar)	<fichero vacío> (dentro de error2.txt) pamor pts/1 2024-04-16 20:37 (salida estándar)

Pruebas ‘Mycalc’

Prueba	Descripción	Salida	Salida modelo (terminal linux)
mycalc 3 add 5	mycalc en caso de suma estándar	[OK] 3 + 5 = 8; Acc 8	8
mycalc 3 add -5	mycalc en caso de suma con número negativo	[OK] 3 + -5 = -2; Acc -2	-2
mycalc 7 mul 2	mycalc en caso de multiplicación estándar	[OK] 7 * 2 = 14	14
mycalc 8 div 2	mycalc en caso de división estándar sin resto	[OK] 8 / 2 = 4; Resto 0	4
mycalc 10 div 3	mycalc en caso de división estándar con resto	[OK] 10 / 3 = 3; Resto 1	~3.3333333333333333
mycalc 3 add 5 mycalc -4 add 2	Demostración de la funcionalidad del Acc en la suma	[OK] 3 + 5 = 8; Acc 8 [OK] -4 + 2 = -2; Acc 6	8 -2
mycalc a add 3 mycalc 5 mul a mycalc a div 6	mycalc en el caso de pasar un operador como letra	[OK] 0 + 3 = 3; Acc 3 [OK] 5 * 0 = 0 [OK] 0 / 6 = 0; Resto 0	"a" is undefined "a" is undefined "a" is undefined
mycalc 3 ¿ 1	Error de formato de operador	[ERROR] La estructura del comando es mycalc <operando_1> <add/mul/div> <operando_2>	Missing operator Error in coomands

Hemos encontrado varias situaciones donde el mycalc no funciona, como puede ser el caso en el que asignamos a un operador una letra en vez de un número, lo que trae como consecuencia que se tome el valor 0 en el lugar de la letra de forma automática. Hemos intentado arreglar dicho error con diferentes mecanismos, como por ejemplo usando la función “isdigit” para comprobar que el string del operador introducido correspondía con un número y en caso contrario devolver un error. Sin embargo, ninguno de estos mecanismos ha surtido efecto acerca de este error.

Pruebas ‘Myhistory’

Prueba	Descripción	Salida	Salida modelo (terminal linux)
myhistory	Utilizar el comando myhistory sin que haya ningún comando guardado	No devuelve nada	No devuelve nada
(ls) myhistory	Myhistory habiendo ejecutado un ls con anterioridad	0 ls	1 ls 2 history
(ls) Myhistory 0	Teniendo en la posición 0 un ls	Ejecutando el comando 0 *impresión ls*	No devuelve nada
Myhistory 17	No teniendo ningún comando guardado en esa posición	ERROR: Comando no encontrado	1 history 17
Myhistory 30	Numero fuera de rango de memoria	Error al ejecutar la instrucción: No such file or directory	1 history 30
(23 comandos) Myhistory	Haciendo 23 comandos para llenar la pila y ver que se borran los más antiguos	0 “comando20” 1 “comando21” 2 “comando22” 3 “comando23” 4 “comando1” ...	1 “comando1” 2 “comando2” 3 “comando3” ... 22 “comando22” 23 “comando23”
Myhistory 1	Teniendo en la posición 1 de una pipe	Violación de segmento	----------
Myhistory	Hacer myhistory después de ejecutar un comando con myhistory con un numero	Error ya que la impresión se realiza mal	----------

*la información entre paréntesis sirve para contextualizar la situación de los comandos

Hemos encontrado varias pruebas donde el myhistory no funciona.

- Prueba con pipes: Si en la posición dada en el myhistory hay guardado un comando con pipes el código da violación de segmento.
- Prueba con parse: Si realizamos un comando normal por ejemplo un ls y posteriormente lo ejecutamos con myhistory 0 esto funciona correctamente, el problema es si volvemos a realizar un myhistory ya que podremos observar que la impresión del primer comando es rara. Esto creemos que es un error con el parser.

Conclusiones de la práctica

Esta práctica ha sido útil a la hora de familiarizarse y conocer los servicios de gestión de procesos y el funcionamiento interno de los intérpretes de mandatos en Linux, los que le permiten al usuario comunicarse con el sistema operativo a la hora de ejecutar mandatos o comandos.

En cuanto a los problemas que han ido surgiendo a lo largo de la práctica, podemos destacar varios de ellos, principalmente vinculados al desarrollo de la propia implementación de nuestro código:

En primer lugar, uno de los mayores retos que se nos presentaron fue la creación de los pipes y el entendimiento de estos. A pesar del material de apoyo que se había proporcionado, teníamos la sensación de que nos quedábamos al borde de la solución definitiva, pero sin poder ver ningún resultado a lo largo de los 4 días que dedicamos a la implementación de dichas tuberías. A base de prueba y error, y con ayuda de material externo tal como videos de YouTube y diversos foros, pudimos resolverlo.

En cuanto a *my history,* el mayor número de problemas se encontró en la funcionalidad que posee el comando al introducirle un valor numérico, es decir, cuando tiene que ejecutar un comando previo al actual. Esto fue así debido a que, además de lo mencionado en las pruebas, tuvimos una gran dificultad para poder comprender el funcionamiento del parser y el run_history. Todos los problemas se consiguieron solventar, salvo las excepciones que se comentan en las pruebas.

La solución al resto de pequeños problemas se ha obtenido a partir de una serie de pruebas implícitas a lo largo del desarrollo de la práctica, además de las pruebas realizadas posteriormente a la escritura del código (aquellas redactadas con detalle en esta memoria) que nos han permitido depurar el propio código, para hacerlo mucho más eficiente y legible para los programadores.

Como opinión general del grupo hemos concluido que esta práctica sobre la programación de "*shells*" ha facilitado mucho la comprensión de su funcionamiento en el sistema operativo y del funcionamiento de los comandos simples o encadenados dentro del sistema operativo Linux. Esta práctica es bastante interesante para toda esta comprensión por el hecho de que es una práctica explicada con detalle en los documentos y que requiere de un nivel mínimo que el alumno ya ha adquirido gracias a la práctica y parciales anteriores.