SISTEMAS OPERATIVOS

Memoria – Práctica 2

04 DE ABRIL DE 2020

Grupo 80
Iván Aguado Peralero - 100405871
Javier Cruz del Valle – 100383156
Jorge Serrano Pérez - 100405987

Índice

- Introducción …….…….…… 2
- Minishell …… 2
  - Batería de Pruebas ……..…..... 3
- Mandato interno: mycalc…….…… 5
  - Batería de Pruebas …… 6
- Mandato interno: mycp ……..... 7
  - Batería de Pruebas …… 8
- Conclusiones ……..... 9

Introducción

El objetivo de esta práctica es la familiarización del alumno con los servicios para la gestión de procesos que proporciona POSIX, además de adquirir el conocimiento del funcionamiento interno de un intérprete de mandatos (Shell) en Unix/Linux. Deberemos diseñar y codificar en leguaje C y sobre el sistema operativo Linux, un programa que actúe como intérprete de mandatos o Shell.

Para ello se nos ha proporcionado un parser que nos permite leer los mandatos introducidos por el usuario, además de un código fuente de apoyo que contiene el makefile que nos permite compilar el código, un probador que realiza una autocorrección y el fichero fuente de C que es donde deberemos codificar nuestras funciones.

Minishell

A continuación, se explicará el código presentado. Como se dice en el enunciado, se ha ido ascendiendo progresivamente empezando por ejecutar mandatos simples con éxito, luego mandatos simples en background, a continuación, mandatos conectados por pipes tanto en foreground como en background, y por último los mandatos internos requeridos: *mycalc* y *mycp*.

En primer lugar, se nos ha dado un código que nos almacena directamente los comandos introducidos en distintos arrays, y luego se ha comprobado que ese número de comandos no sea mayor que 8 (MAX_COMMANDS), dando error en caso contrario.

Una vez dicho esto, comenzamos creando una serie de variables auxiliares para ayudarnos, en las que almacenaremos las variables de entrada, salida y error estándar, que usaremos más adelante para reestablecer las variables del sistema a su estado original.

A continuación, comenzamos a leer los comandos introducidos utilizando para ello un bucle, y de esta manera no nos limitamos a leer solo tres.

Lo primero que hacemos es crear la tubería y almacenar su valor devuelto en una variable, para así poder comprobar que esta no da error, y si es así, informamos al usuario de ello y terminamos el programa.

El siguiente paso es comprobar si nos encontramos en el caso del último comando, que será entonces el último hijo. En ese caso deberemos cerrar el *fd* que hemos creado para guardar los descriptores (*fd* [1]). Si no lo es, entonces tendremos que almacenar la salida del anterior comando en la variable estándar correspondiente (salida) antes de cerrarla.

Seguimos, ahora ya creando el *fork* que nos creará a su vez un nuevo proceso: el que hemos escrito en la minishell. Primero controlamos que no haya ningún error al hacer el *fork*, que sería el caso en el que el valor del *pid* es -1, y luego ya simplemente redireccionamos la entrada, la salida y el error estándar a la dirección de *filev* que corresponda, ya que aquí es donde se encuentran las entradas, salidas y errores del comando anterior.

Continuamos ahora haciendo el *exec* del hijo en el que nos encontramos, aunque comprobamos también si este no da error (devuelve -1).

Y ahora llega el momento de codificar el caso en el que estamos en el proceso padre. Es donde comprobaremos si el mandato introducido se hace en background o no. Si es así, simplemente guardamos las entradas del hijo actual en la entrada estándar para que la recoja el siguiente hijo. Si no, tendremos que añadir el *wait* para que este espere el resultado de sus hijos.

Por último, restauramos con las variables auxiliares creadas al comienzo del código las entradas, salidas y error estándar, y luego las cerramos.

Como se puede ver, está toda la minishell programada y plenamente funcional, como se demuestra a continuación con esta batería de pruebas.

Batería de pruebas

Prueba	Resultado esperado	Resultado obtenido
Mandato simple	Función del mandato en concreto	Correcto
Mandato simple con redirección de entrada	Función del mandato en concreto que obtiene los datos de un fichero	Correcto
Mandato simple con redirección de salida	Función del mandato en concreto que almacena los datos en un fichero	Correcto

Mandato simple con redirección de error	Función del mandato en concreto que almacena los datos en un fichero	Correcto
Mandato simple en background	Función del mandato en concreto en background	Correcto
Mandato con un pipe	Función de los mandatos conectados en cada caso	Correcto
Mandato con un pipe con redirección de entrada	Función de los mandatos conectados en cada caso que obtienen los datos de un fichero	Correcto
Mandato con un pipe con redirección de salida	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con un pipe con redirección de error	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con un pipe en background	Función de los mandatos conectados en cada caso en background	Correcto
Mandato con dos pipes	Función de los mandatos conectados en cada caso	Correcto
Mandato con dos pipes con redirección de entrada	Función de los mandatos conectados en cada caso que obtienen los datos de un fichero	Correcto
Mandato con dos pipes con redirección de salida	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con dos pipes con redirección de error	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con dos pipes en background	Función de los mandatos conectados en cada caso en background	Correcto

Mandato con más de dos pipes	Función de los mandatos conectados en cada caso	Correcto
Mandato con más de dos pipes con redirección de entrada	Función de los mandatos conectados en cada caso que obtienen los datos de un fichero	Correcto
Mandato con más de dos pipes con redirección de salida	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con más de dos pipes con redirección de error	Función de los mandatos conectados en cada caso que almacenan los datos en un fichero	Correcto
Mandato con más de dos pipes en background	Función de los mandatos conectados en cada caso en background	Correcto

Mandato interno: mycalc

Se trata de una calculadora sencilla en la terminal. Se insertará una ecuación que siga el formato *operando operador operando*, con el operando como número entero y operador pudiendo ser solo la suma o el módulo. Para la suma se almacenará el valor en una variable entorno (*Acc*), y para el módulo se sacará el resultado por pantalla siguiendo el esquema *Dividendo = Divisor * Cociente + Resto*.

En primer lugar, tenemos que declarar la variable Acc de entorno de manera correcta. Para ello la declaramos dándole el nombre y el valor inicial correspondiente, y la inicializamos usando el método *setenev()*.

Para su ejecución, comenzamos comprobando el número de operandos que se han introducido. Cuando son cuatro, lo dividimos en el caso de suma (*add*) o de modulo (mod). Son cuatro refiriéndonos a *mycalc*, operando, operador y operando.

Si se trata de la suma, creamos la variable *accAux* para almacenar la variable de entorno usando *getenv*(). Como se trata de un String, la convertimos a entero para poder operar con ella, y una vez tenemos el valor de total de la suma, lo reconvertimos a *String (sprintf)* y lo actualizamos en la variable de entorno principal utilizando de nuevo setenv().

Además, almacenamos los operandos convertidos a enteros en variables auxiliares, para luego hacer la suma y sacarla por pantalla a través de la variable estándar de error (para ello hemos usado la función *fprintf*).

Por otro lado, si se trata del módulo, creamos las variables auxiliares necesarias para guardar el valor del resto y del cociente, convertimos los argumentos introducidos por el usuario en enteros, realizamos la operación y la imprimimos por pantalla de la misma manera que en el caso anterior.

Finalmente, en el caso de que no sean cuatro los argumentos introducidos, sacaremos el error indicado en el enunciado por pantalla, y haremos lo mismo en el caso en el que no se respete el formato de la operación de suma o módulo (Por ejemplo, si se introduce *mas* en vez de *add*).

Cabe destacar que este código se ha añadido en la parte anterior al *fork* de la Minishell, ya que, al ser un mandato interno, no tiene redirecciones de ficheros ni se ejecuta en background.

Batería de pruebas

Prueba	Resultado esperado	Resultado obtenido
mycalc con números enteros positivos	[OK] operando1+operando2 = suma; Acc sumaTotal	Correcto
mycalc con números enteros negativos	[OK] operando1+operando2 = suma; Acc sumaTotal	Correcto
mycalc con números enteros de distinto signo	[OK] operando1+operando2 = suma; Acc sumaTotal	Correcto
mycalc con números racionales	[OK] operando1(convertido a int) + operando2(convertido a int) = suma; Acc sumaTotal	Correcto
mycalc con letras en vez de números	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	[OK] 0+0 = 0; Acc sumaTotal

mycalc con símbolos en vez de números	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	[OK] 0+0 = 0; Acc sumaTotal
mycalc con operador distinto a add	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	Correcto
mycalc con operador distinto a mod	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	Correcto
mycalc sin argumentos	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	Correcto
mycalc con más argumentos de los indicados	[ERROR] La estructura del comando es <operando 1> <add/mod> <operando 2>	Correcto

Mandato interno: mycp

Este mandato coge un fichero y lo intenta copiar en el directorio pasado como argumento, poniéndole el mismo nombre que el del archivo original.

En este caso e igual que en el anterior, comprobamos que el numero de argumentos sea el adecuado, en este caso 3, ya que sigue la estructura *mycp fich_origen fich_dest*.

A continuación, guardamos el valor del argumento en unas variables para no perderlos. El siguiente paso es comprobar si existen errores al abrir el fichero origen o el fichero destino, en cuyo caso sacamos por pantalla el error indicado en el enunciado, esta vez por la salida estándar.

Luego creamos el buffer para guardar los bytes leídos y comprobamos que no existe error al leer el fichero, y reescribimos esos bytes en el fichero de destino.

Por otro lado, cerramos los ficheros y comprobamos que tampoco surge ningún error en ello, y por último mostramos el mensaje indicado en el enunciado si todo se realiza con éxito (*"[OK] Copiado con éxito el fichero <origen> a <destino>"*).

Sin embargo, si el número de argumentos no es el adecuado, mostramos el siguiente mensaje (*"[ERROR] La estructura del comando es mycp <fichero origen> <fichero destino>"*) como se nos ha sido indicado.

Cabe destacar que este código, de la misma manera que el anterior, se ha añadido en la parte anterior al *fork* de la Minishell. Esto se debe a que es un mandato interno, y por lo tanto no tiene redirecciones de ficheros ni se ejecuta en background.

Batería de pruebas

Prueba	Resultado esperado	Resultado obtenido
mycp con dos ficheros existentes	[OK] Copiado con éxito el fichero *fich_orig* a *fich_dest*	Correcto
mycp con fichero origen inexistente	[ERROR] Error al abrir el fichero origen	Correcto
mycp con fichero destino inexistente	[OK] Copiado con éxito el fichero *fich_orig* a *fich_dest*	Correcto
mycp con dos ficheros existentes y el destino con contenido	[OK] Copiado con éxito el fichero *fich_orig* a *fich_dest*. (Se sobrescribe el contenido)	Correcto
mycp con el mismo fichero existente	[OK] Copiado con éxito el fichero *fich_orig* a *fich_dest*. (Se sobrescribe el contenido, pero al ser el mismo se queda igual)	Correcto
mycp con fichero origen y el fichero destino el nombre de un directorio	[ERROR] Error al abrir el fichero destino	Correcto
mycp sin argumentos	[ERROR] La estructura del comando es mycp <fichero origen> <fichero destino>	Correcto

mycp con un argumento	[ERROR] La estructura del comando es mycp <fichero origen> <fichero destino>	Correcto
mycp con más argumentos	[ERROR] La estructura del comando es mycp <fichero origen> <fichero destino>	Correcto

Conclusiones

Para concluir este trabajo, añadimos aquí los problemas que hemos encontrado y nuestras opiniones.

En primer lugar, consideramos que el enunciado estaba menos esquematizado que la práctica anterior y no terminaba de quedar claro qué camino seguir para hacer la práctica, por lo que quizá pillar el hilo de la práctica ha sido más complicado que en la anterior, por ejemplo.

En segundo lugar, creemos que el código que nos ha sido proporcionado quizá podría haber estado mejor explicado, ya que nos llevó un tiempo comprender, por ejemplo, el funcionamiento del método *getCompleteCommand*.

Por último, hay que destacar la pesadez de tener que repetir cada vez el exportar la librería proporcionada, aunque en este caso entendemos que no hay otra opción.

Por lo tanto, nos ha parecido una práctica útil, que nos enseña con gran claridad la importancia y el trabajo que necesita un Sistema Operativo para comunicarse de manera óptima con el usuario. En cuanto a la complejidad, está claro que sobre todo al comienzo es difícil y que hay que dedicar trabajo y esfuerzo para comprender el código proporcionado y entender donde se almacena cada cosa. Consideramos que la parte más fácil ha sido los mandatos internos, ya que son muy parecido a la práctica uno que ya hicimos anteriormente.