Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3825 - Sistemas de Operación
Profs. Fernando Torre y Fernando Lovera

Proyecto 2: Terminal Esencial

Elaborado por:
Kauze B., Jesús E. 12-10273
Faria R., Manuel A. 15-10463
Oropeza C., Juan A. 15-11041

Sartenejas, junio de 2019

Introducción

Las estructuras para controlar el sistema de archivos son realmente importantes para la administración del mismo en el computador. En este proyecto empleamos los conceptos aprendidos en la parte teórica de este curso para la administración de memoria y el almacenamiento de los archivos en el disco.

Para esto, desarrollamos una aplicación en el lenguaje de programación C con la finalidad de emular el funcionamiento de un intérprete de comandos reducido. Entre las funciones del mismo se encuentran versiones simplificadas de los comandos ls, grep y chmod. Para ello, manipulamos las estructuras de control de archivos en el sistema y con ellas, realizamos los cambios necesarios en dichas estructuras y utilizamos la amplia variedad de llamadas al sistema que proporciona el lenguaje de programación empleado.

Gran parte del comportamiento de los comandos creados en este intérprete fueron hechos a semejanza de las implementaciones de UNIX, tal como fue indicado por los instructores del curso.

Cómo correr el programa

Para la ejecución del intérprete se creó un programa principal. Para ejecutar dichos programa, primero deberá compilarlo. Para ello, ejecute en el directorio principal el siguiente comando:

$ *make*

Una vez compilado tendrá disponible un archivo ejecutable para iniciar el intérprete de comandos. Existen dos maneras de correr el intérprete:

1. Archivo de Entrada: Para ejecutar el intérprete en este modo deberá ejecutar el siguiente comando:

$ *tesh script*

Donde *script* es un archivo que contiene una secuencia de comandos correctos a ser ejecutados. De esta forma, el intérprete imprimirá en pantalla el resultado de la ejecución de los comandos siempre que dichos comandos tengan un *output*.

2. Prompt de Comandos: Para ejecutar el intérprete en este modo solo deberá ejecutar el siguiente comando:

$ *tesh*

De esta manera se abrirá un *prompt* de comandos donde el usuario deberá ingresar las acciones que desee que sean ejecutadas.

Diseño y Estructura

El diseño de este proyecto se basó en las casos presentados en el enunciado. Esta implementación consta de dos carpetas:

1. **Main:** En ella se encuentra el archivo *tesh* con el cual se ejecutará el intérprete desarrollado.
2. **Commands:** En esta carpeta se encuentran los archivos ejecutables de los comandos desarrollados. Si en un futuro se desean agregar comandos al intérprete, deberán agregarse en esta carpeta.

A continuación, se enuncia la estructura de los archivos, y cómo desarrollan cada una de las funcionalidades para las cuales fueron creados.

1. **tesh.c:** En este archivo se ejecuta la función que da vida a la implementación, la cual decide en función de la cantidad de parámetros suministrados en la corrida del comando. Si se escoge correr el comando sin argumentos, se ejecuta la función $loop()$ que genera el prompt de comandos para la entrada del usuario. En caso contrario se ejecuta la función $readFromFile(filename)$, que abre el archivo pasado como argumento y comienza a leer las líneas en el. Una vez tomada la línea bien sea desde el prompt o desde el archivo, el programa se comporta del mismo modo. Lo primero que hace es revisar si en la expresión existe un ***pipe***, de no contenerlo, separar el string leido para obtener los parámetros, para que luego ejecute el comando, creando un proceso nuevo y utilizando la instrucción $execv()$ En caso de contener un ***pipe***, divide las instrucción en dos partes. La primera parte es ejecutada y su salida es redirigida a un *pipe* para que luego sea leída por el segundo comando. Para ejecutar cada comando se crea un proceso hijo distinto.
2. **chmod.c:** Para la implementación de este comando, se obtiene la estructura de permisología de los archivos con el ***syscall*** $stat()$, con él se genera la máscara de bits apropiada según sea el caso de las opciones con la cuales fue ejecutado el comando. Una vez modificada la máscara de bits se guarda la estructura con el ***syscall*** $chmod()$. Las redirecciones se implementan tanto en este comando como en los otros que fueron implementados, escribiendo la salida en un archivo o recibiendo como input el contenido de un archivo.
3. **ls.c:** El comando ls se implementó haciendo uso de las librerías ***stat.h*** y ***dirent.h*****.** Para obtener las propiedades de cada archivo y obtener los archivos según la ruta especificada por el usuario. La implementación

hace uso de 2 grandes casos que se dividen en múltiples subcasos, estos mismo son: cuando el archivo es un directorio y cuando no lo es. Para verificar si un archivo (*recordar que todo en unix es un archivo*) se utilizó la salida de *opendir().* Si dicha salida arroja NULL estamos ante la presencia de un parámetro que puede ser un: *archivo/flag/error/*

En caso contrario estamos ante la presencia de un directorio, gracias a la estructura DIR aportada por *dirent.h.* El reto de *ls* consistió en los múltiples subcasos de la implementación del mismo, donde se evalúa si se pasa por parámetro un archivo o no, combinación de flags, comando sin argumentos, entre otros. Para el redireccionamiento se implementó un *int* tal que haga función de llave para filtrar los datos que serán impresos en el archivo de salida.

4. **grep.c:** La implementación de este comando al igual que para ls, se divide en dos grandes casos. El primer caso, en el que se implementa el prompt de grep, donde mediante un ciclo se obtiene input del usuario y para cada línea ingresada, se verifica si existe alguna coincidencia con el patrón usado y el segundo caso, donde grep busca coincidencias del patrón usado en uno o más archivos, para ello se usa un for que itera sobre los argumentos referidos a archivos y se corre el programa de la misma forma que para el input de usuario pero sobre cada archivo, buscando de la misma manera las coincidencias sobre cada una de las líneas, además para este caso se toma en cuenta la presencia del argumento **“>”**, y haciendo uso de condicionales una vez conseguida las coincidencias, en vez de imprimir en la salida, se escribe en el archivo referido en el último argumento. La implementación de los flags de grep fue sencilla, una vez identificado los flags utilizados, mediante verificaciones con if se corre la función adecuada para conseguir la coincidencia deseada.

Dificultades encontradas

La mayor dificultad encontrada fue la redirección entre dos procesos cuando existía la presencia de un *pipe.* Dado que nuestra solución fue redirigir la salida a *stdin* para que fuera leída por el siguiente comando, la implementación se solapaba con la ejecución de grep con un solo parámetro. Solucionamos este problema con la implementación de un flag especial para el comando *ls* que le indica cuando debe redirigir su salida hacia un pipe, imprimiendo un caracter especial para indicarle al siguiente comando cuando debe terminar su ejecución. Invertimos numerosas horas de investigación navegando a través de internet, implementando variadas soluciones que nos permitieron escoger la más ajustada a nuestro problema. Esta dificultad se presentó debido a la versatilidad de los comandos implementados, ya que en el caso del comando *grep* ejecuta su propio *prompt* que espera entrada del usuario por un tiempo indefinido.

Conclusiones

1. Estado de la implementación

El estado de la implementación realizada es completamente funcional para los casos definidos en el enunciado del proyecto. Ejecuta extensos archivos de comandos con diversas combinaciones y imprime su salida en la pantalla. Si se ejecuta como en el modo *prompt* pueden realizarse todas las opciones solicitadas.

2. Mejoras a la solución

Para mejorar nuestra solución principalmente podríamos generalizar secciones de código que están repetidas pero que por cuestiones de tiempo no pudieron ser generalizadas para que funcionaran en las zonas del código donde eran necesarias.

También creemos conveniente que como aplicación a los requisitos del proyecto podríamos implementar una función recursiva para leer cuantos *pipes* sean necesarios en una sola expresión y que nuestro intérprete no esté limitado a expresiones de un solo *pipe*

De igual manera utilizaríamos una herramienta de *linting* para mejorar el estilo del código y la estandarización del espaciado, definición de variables, etc.

Fuentes Consultadas

- StackOverFlow.com
- GeeksForGeeks.org
- Manpages de los distintos comandos implementados (chmod, ls, grep)
- TutorialsPoint.com
- Aula Virtual del Curso