MINISHELL

Práctica 1 – Sistemas Operativos

2018-2019

Mikayel Mardanyan Petrosyan
3º GII

Índice de contenidos

1. Introducción

El objetivo de esta práctica es la adquisición de conocimientos sobre el funcionamiento de un intérprete de mandatos o *shell*, para lo cual se debe comprender qué son los procesos y cómo estos se comunican entre ellos mediante tuberías o *pipes*.

El funcionamiento del intérprete de mandatos implementado, llamado *myshell*, es el siguiente:

1. Muestra el *prompt* por pantalla.
2. Lee una línea del teclado.
3. Analiza la línea usando la librería *parser*.
4. Ejecuta todos los mandatos a la vez y vuelve al paso 1.

2. Autoría

El autor de la práctica y de esta memoria es únicamente Mikayel Mardanyan Petrosyan.

3. Descripción del código

El código implementado para esta práctica se encuentra en el archivo *myshell.c*, pero primero se describirá la librería *parser*, ya que la *minishell* hace uso de esta.

En la librería *parser*, el tipo de datos *tline* organiza los datos extraídos de la línea de comandos. Almacena el número de mandatos que hay en la línea, los distintos mandatos que aparecen en variables de tipo *tcommand*, un entero que marca si el mandato se ejecuta en *foreground* o *background* y los nombres de fichero para las distintas redirecciones, en el caso de que estas existan. El tipo de datos *tcommand* guarda la ruta del mandato, o NULL si el mandato no existe, y el número y contenido de los argumentos, en caso de que estos existan. El propósito de la función *tokenize()* es organizar la información de la línea de comandos en una variable de tipo *tline*.

Al principio del archivo *myshell.c* nos encontramos con las directivas del preprocesador y la declaración de las funciones implementadas. A continuación, empieza el bloque *main()*. Las variables de este bloque son:

- Una variable que guarda el *pid* del proceso padre.
- Un buffer que guarda la información introducida por teclado.
- Una cadena de caracteres que almacena el directorio para el mandato *cd*.
- Una variable del tipo *tline*.
- Un array con los *pipes* que se vayan a usar durante la ejecución del programa.
- Enteros para guardar la posición de los descriptores de fichero en el caso de que haya redirecciones.
- Enteros que servirán de contador.
- Un entero para el control de errores y ejecución.

Antes de mostrar el *prompt*, se modifica el comportamiento del programa ante las señales SIGINT y SIGQUIT, haciendo que estas se ignoren. Para parar el programa se debe usar la señal *Ctrl+Z*. Se imprime el *prompt* por pantalla y comienza un bucle infinito del que no se podrá salir hasta usar la señal indicada antes.

Dentro del bucle, se comprueba si el usuario ha introducido un salto de línea. Si es así, se imprime el *prompt* de nuevo y se espera a una nueva línea. Si la línea introducida contiene información, se tokeniza y organiza el contenido en una variable de tipo *tline*. Se diferencian dos tipos de mandatos, los propios de la *minishell* (mandato *cd*) y los que son programas del sistema. Ahora se debe comprobar qué tipo de mandato ha introducido el usuario.

- Si se quiere ejecutar el mandato *cd*, se comprueba que no lo está ejecutando con *pipes* y que no introduce más de un argumento. En cualquiera de estos casos, se muestra el error por pantalla y termina la iteración, mostrándose de nuevo el *prompt*. Si lo hace correctamente, se comprueba que el argumento es un directorio y se cambia a él, mostrándolo por pantalla. En el caso de que no tuviera ningún argumento, intenta acceder al directorio almacenado en la variable del sistema *HOME*. Si lo consigue, se cambia el directorio a este; si no, se muestra el error.
- Si se quiere ejecutar un mandato del sistema, primero se recorre la línea introducida comprobando que todos los mandatos existen y que el mandato *cd* no se ejecuta con *pipes*. En cualquiera de estos casos, se notifica el error y se aborta la ejecución, acabando así la iteración.

  Si todo está correcto, se pasa a reservar la memoria para los *pipes*. El número de *pipes* necesarios es el número de mandatos menos uno. A continuación, hay un bucle *for* para crear los procesos hijo y los *pipes* que estos van a usar. Se crean las tuberías y para cada mandato se llama a *fork()* para crear un proceso hijo.

  - Si el *fork* salió mal, se notifica por pantalla y se aborta la ejecución.
  - Para los procesos hijo, se reestablece el comportamiento frente a las señales SIGINT y SIGQUIT. Después, se redirecciona la entrada y la salida en caso de que se especifique.
    - Si se quiere redireccionar la salida, se abre el archivo en un *file descriptor* y se cambia la salida del último mandato a este archivo. En caso de que el fichero no exista, se crea. Para la redirección de error el procedimiento es similar.
    - Para la redirección de entrada, se comprueba que el fichero existe y se cambia la entrada del primer mandato a este archivo, del mismo modo que antes.

    Si el fichero de entrada no existe o hay un fallo en cualquiera de las otras dos redirecciones, se notifica por pantalla y se aborta la ejecución del mandato, finalizando la iteración.

    Para el resto de procesos intermedios, se redirige su entrada y salida a través de las tuberías y se cierran las que no va a usar. Por último, se ejecuta el mandato. Si dicha ejecución sale bien, el proceso termina y no avanza a la siguiente línea de código.
  - Para los procesos padre, se cierran los *pipes* que no usa y se espera a que termine la ejecución de los hijos.

  Para finalizar, se libera el espacio de memoria reservado para los *pipes* y se imprime el *prompt* por pantalla.

4. Objetivos cumplidos

a. Ser capaz de reconocer y ejecutar en *foreground* líneas con un solo mandato y 0 o más argumentos.
b. Ser capaz de reconocer y ejecutar en *foreground* líneas con un solo mandato y 0 o más argumentos, redirección de entrada estándar desde archivo y redirección de salida a archivo.
c. Ser capaz de reconocer y ejecutar en *foreground* líneas con dos mandatos con sus respectivos argumentos, enlazados con '|', y posible redirección de entrada estándar desde archivo y redirección de salida a archivo.
d. Ser capaz de reconocer y ejecutar en *foreground* líneas con más de dos mandatos con sus respectivos argumentos, enlazados con '|', redirección de entrada estándar desde archivo y redirección de salida a archivo.
e. Ser capaz de ejecutar el mandato *cd*.
f. Evitar que ~~los comandos lanzados en *background* y~~ el *minishell* mueran al enviar las señales desde el teclado SIGINT y SIGQUIT, mientras los procesos en *foreground* respondan ante ellas.

5. Objetivos no cumplidos

a. Ser capaz de reconocer y ejecutar tanto en *foreground* como en *background* líneas con más de dos mandatos con sus respectivos argumentos, enlazados con '|', redirección de entrada estándar desde archivo y redirección de salida a archivo. Para su correcta demostración, se deben realizar los mandatos internos *jobs* y *fg*.

6. Comentarios personales

Los problemas encontrados durante la realización de la práctica fueron los siguientes. Muchos de ellos son los indicados en la corrección de la convocatoria ordinaria.

Problema	Causa	Solución
Los *pipes* funcionan con algunos mandatos y con otros no.	No cerrar los *pipes* en los procesos que no los usan.	Cerrar los *pipes* en el proceso padre.
Al introducir únicamente saltos de línea (\n) en la línea de mandatos da *Segment violation*.	El programa continua la ejecución de un mandato inexistente.	Controlar que al introducir una línea vacía termine la iteración y se vuelva a mostrar el *prompt*.
No se muestran mensajes de error cuando no existe el mandato.	Se esperaba que al llamar a *execvp()* con un mandato inexistente, este diera fallo y pasara a la siguiente línea de código, en la que se muestra el mensaje de error.	Para comprobar si un mandato existe se debe acceder, dentro de la línea introducida, al *tcommand* del mandato en cuestión y después a la variable *filename* que guarda en su interior. Si esta variable es NULL, el mandato no existe.
En las redirecciones de salida y error no se crea el archivo si este no existe.	El programa no estaba preparado para estos casos.	Modificar los *flags* de la función *open()*.

En la redirección de entrada, el programa deja de funcionar si el archivo no existe.	El programa no estaba preparado para estos casos.	Controlar este error y detener la ejecución del mandato.
El mandato *cd* se ejecuta cuando es llamado con *pipes*.	Se comprobaba si el primer mandato de la línea era *cd* y se ejecutaba, independientemente de lo que hubiera a continuación.	Comprobar si en las llamadas con *pipes* alguno de los mandatos es *cd*. Si es así, detener la ejecución y terminar la iteración.

En cuanto al tiempo dedicado, la mayor parte de la práctica ya estaba hecha, corregir los fallos no me ha llevado mucho tiempo. Un aspecto a mejorar sería incluir en el enunciado ejemplos de los distintos apartados, como tres mandatos que se puedan ejecutar a la vez, con el fin de comprobar el correcto funcionamiento de la *minishell*.