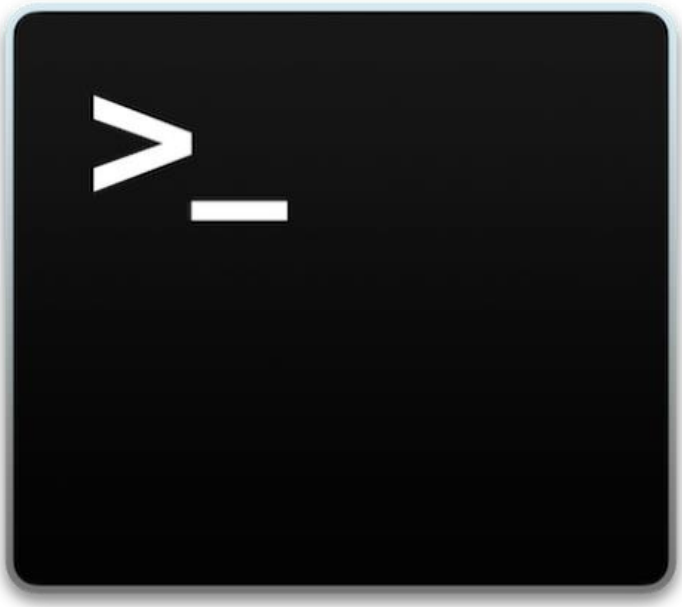

SISTEMAS OPERATIVOS

Práctica MiniShell

Autores: **Patricia Tarazaga Cozas**
Álvaro Martínez Quiroga

24/04/2020

GRADO EN INGENIERÍA DE COMPUTADORES

Universidad Rey Juan Carlos

Contenido

Tablas

Ilustraciones

INTRODUCCIÓN

Esta práctica va a desarrollar una Shell o consola orientada a satisfacer los comandos implementados en el sistema operativo Linux junto con algunos comandos denominados *comandos internos* que proporcionará la propia Shell con motivo de gestionar la ubicación dentro del sistema de directorios y la gestión de los procesos lanzados por el usuario.

El nombre de la práctica, minishell, viene dado porque pese a tener una consola bastante completa no implementaremos todas sus funcionalidades ni tampoco estará diseña como una Shell real (programa o proceso especial con dependencias dentro del sistema operativo en cuestión), si no que será un programa escrito en lenguaje C que compilaremos y ejecutaremos desde el interprete de comandos real.

La programación e implementación de la minishell estará basada en los conocimientos adquiridos durante la asignatura Sistemas Operativos. Las técnicas de creación de procesos, gestión de los mismos y utilización de los comandos Scripts serán la base de la construcción y finalización de la práctica.

La documentación aportada para la entrega de la práctica será la solicitada.

Archivos adjuntos:

- Memoria en formato PDF
- Comprimido RAR con el código fuente minishell.c y el ejecutable del programa (*minishell*)

PLANTEAMIENTO

Ya que la practica abarca un amplio tema de diferentes puntos a desarrollar se ha decidido dividirla en varias secciones, siguiendo un orden de más simple a más complejo. También se ha tenido en cuenta pasos que no serían posibles de realizar si algunos de los otros no estuviesen implementados.

El esquema el que se ha seguido es el de la Tabla 1

PASO	UTILIDAD	DESCRIPCION Y DETALLES
1	▪ Comando cd. ▪ Ejecución de un comando.	El comando cd tendrá que ejecutarse siempre solo.
2	▪ Un comando ejecutado en segundo plano (background, &). ▪ Ejecución de varios comandos (pipes).	Si se ejecuta un comando en segundo plano este no debe parar (bloquear) la ejecución de la minishell o ningún otro proceso. Pipes: ! Input: redireccionamiento solo del primer comando. ! Output: redireccionamiento solo del ultimo comando.
3	Señales: ▪ SIGINT ▪ SIGQUIT	Que ninguna de estas dos señales sea capaz de cerrar la ejecución de la minishell. Tener algún comando para salir de la minishell (exit).
4	▪ jobs. ▪ fg (traer un comando de segundo plano a primer plano).	
5	Personalización de la miniShell (no necesario)	La minishell puede imprimir, dependiendo del mensaje, por pantalla de un color u otro.

Tabla 1: guía esquemática

Como puede observarse en la Tabla 1 se ha distinguido 5 pasos a seguir. La división de cada paso se ha hecho analizando el enunciado y estudiando las posibles complicaciones que se tendría al implementar cada parte del código. Por decisiones de complejidad o de altas dependencias, se ha programado en el orden mostrado.
La columna *UTILIDAD* está rellena con los comandos que debían programarse en cada paso, cumpliendo el correcto funcionamiento antes de continuar a la siguiente sección.
Por último, la columna de *DESCRIPCIÓN Y DETALLES* es una incorporación personal a nivel de equipo para focalizarnos en algún matiz imprescindible para cumplir los objetivos de funcionamiento o recordatorio de cómo debe implementarse algo concreto.

Se ha usado la tabla de guía durante todo el proceso, guardando así en distintas versiones cada paso finalizado, obteniendo de este modo un resguardo o backup de cada cambio grande y poder tener siempre código de apoyo y la estructura anterior en caso de *crasheo* de programa.

DIAGRAMA DE FLUJO DE LA SHELL

A continuación, en la Ilustración 1 se presenta el diagrama de flujo que sigue la minishell.

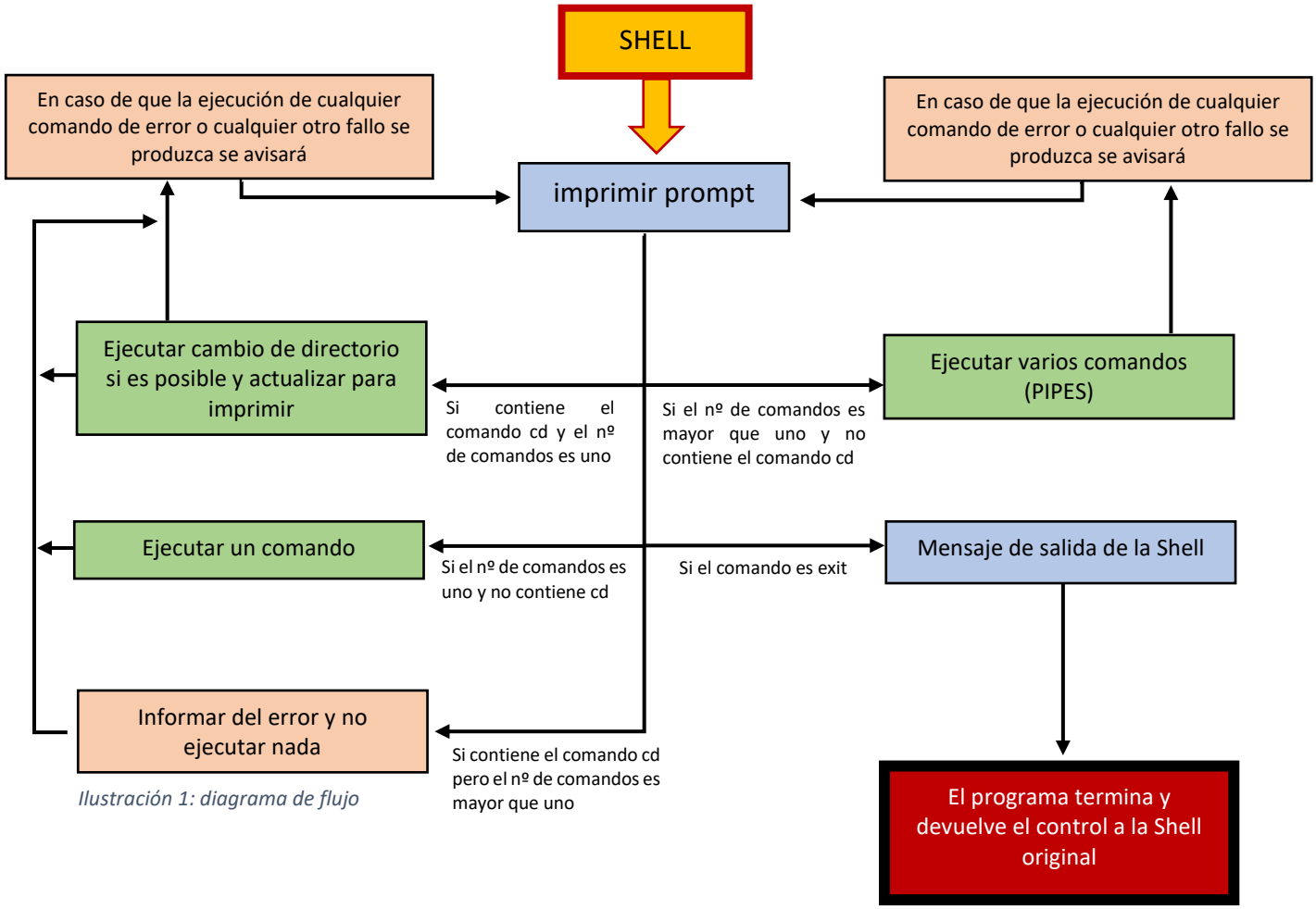

Ilustración 1: diagrama de flujo

Ilustración 2: leyenda del diagrama de flujo

Notas sobre el diagrama de flujo:

- Nótese que no se plantean las posibilidades de los comandos “jobs” o “fg”, sin embargo, entran dentro del ciclo de ejecución de un único comando, aunque en el código pertenezcan a secciones diferentes.

Comenzamos la programación del código incluyendo librerías necesarias y definiendo estructuras de datos, variables globales y funciones, seguido de la función principal **main().**

Después esperamos a que el usuario introduzca una línea por teclado para analizarla y comenzamos a hacer comprobaciones para centrar una ruta de ejecución del diagrama de la Ilustración 1.

INSTRUCCIONES DE USO

Partiendo de la base de que el usuario sabe manejar una Shell y entiende su funcionamiento se procede a explicar algunos detalles del programa minishell propio:

- **Comando fg:** el comando funciona tanto con el número de índice proporcionado por el comando jobs como por el pid real del proceso ejecutado en segundo plano (también proporcionado por el comando jobs). Es posible utilizar solo el comando sin ningún argumento y pasará a primer plano el ultimo comando mandado a background. Por lo tanto, el uso de fg es:
  fg <pid> o <número de índice>
  fg (pasará a primer plano el último comando).

- **Comando exit:** comando interno usado para salir de la minishell. Imprimirá por pantalla un mensaje y matará el proceso. Está contemplado en segundo plano. Su uso es:
  exit

DESARROLLO

En esta sección se pasará a explicar las distintas partes del código y el pensamiento lógico detrás cada línea programada. No se copiará en ningún caso el código fuente, pero se referenciarán las sentencias de selección (los *if* que podemos encontrar) que distinguen las situaciones que pueden darse dentro de la minishell.
Por otra parte, se han incorporado los comentarios oportunos en el código fuente para facilitar su entendimiento, por lo que puede concebirse como una ampliación de la explicación en este apartado. Se considerará así que la explicación técnica estará contenida en el archivo con el código fuente y la explicación teórica o analítica se hará en este documento.

La forma de referenciar las sentencias de selección se hará en formato de números ordinales. Empezaremos el conteo a la entrada del primer *else*, puesto que la primera sentencia de selección es un simple controlador para el prompt (en caso de pulsar un *enter,* sacar el prompt de nuevo).

La **primera** comprobación es para el comando "cd". En este caso pueden ocurrir 3 escenarios diferentes:

- Si el comando se ha introducido solo: la minishell cambiará el directorio de trabajo al directorio raíz del sistema.
- Si el comando se ha introducido con un parámetro: se comprueba que este directorio existe y se puede acceder al mismo y se procede al cambio de directorio.
- Si el comando se ha introducido con más de un parámetro: se informará del error y no se cambiará el directorio de trabajo.

La **segunda** comprobación es para el comando "jobs". Este comando simplemente accede a la estructura de datos del programa, una lista enlazada simple (con puntero al siguiente elemento) e imprime todos los procesos que se han creado en segundo plano y están en ejecución aún.

La estructura es la siguiente:

TIPO DE DATO	NOMBRE	SIGNIFICADO
int	idx	Número de orden de creación
int	id	PID del proceso
char	nombre	Nombre del proceso
char	status	En "ejecución" o "hecho"
Estado* siguiente	next	Puntero al siguiente elemento

Tabla 2: struct de lista enlazada

La **tercera** comprobación es para el comando "fg". Este comando sigue el siguiente esquema de ejecución:

1. Comprueba que el comando se haya introducido con un único parámetro, si no, informara del error.
2. Comprueba que el valor introducido exista en la estructura de la lista, esto es, que exista un proceso en ejecución con ese pid o número de índice de la lista. Seguido de esto, si el pid no existe se informará y si el pid es el mismo pid que el de la minishell, esta informará que ya está en ejecución en primer plano.

3. Por último y en caso de que no se haya ejecutado ninguno de los anteriores errores, el proceso seleccionado se traerá a primer plano, imprimiendo su nombre por consola y la minishell esperara a hasta el fin de su ejecución.

La **cuarta** comprobación se relaciona con la ejecución de un solo comando, en este caso puede ocurrir lo siguiente:

- El comando introducido es igual al comando "exit", se terminará la ejecución del programa.
- El comando introducido es interno de la propia Shell, se procede a su ejecución.
- El comando introducido no existe, la Shell no ejecutará nada e imprimirá una nueva línea en el "prompt".

La **quinta** comprobación pertenece a varios comandos (pipes) y su ejecución es la siguiente:

- Si todos los comandos están bien introducidos, se redirigen las salidas para cada proceso en sus respectivos pipes. Nota: primero tendrán que crearse los pipes y después ya podremos ejecutar "fork()" por cada uno de los hijos, si no los hijos conocerían una copia de estos pipes, no los originales. Finalmente se procede a cerrar los pipes y liberar la memoria que estos ocupan, al igual que los hijos.

La **sexta** comprobación pertenece al error de lectura de comandos el cual simplemente imprime por pantalla un mensaje de error y vuelve a imprimir el prompt del programa.

Finalmente, y basándonos en una ejecución de una Shell como la de Linux se ha decidido cambiar el color de algunos mensajes que se imprimen por pantalla para que sea más visual cuando, por ejemplo, se produce un error o se sale de la minishell. Lo conseguimos de la siguiente manera mostrada en la Tabla 2.

EJEMPLO	CÓDIGO

1	fprintf(stderr, "\033[1;31m");
	fprintf(stderr, "Error en la lectura de comandos.\n");
	fprintf(stderr, "\033[0;0m");

2	fprintf(stderr, "\033[1;35m");
	fprintf(stderr, "\nSaliendo de la MiniShell.\n\n");
	fprintf(stderr, "\033[0;0m");

Tabla 3: ejemplos de mensajes coloridos

Lo único que realizan estos comandos es cambiar el color de la salida estándar en el primer caso a rojo y en el segundo caso a magenta y después de imprimir el mensaje deseado devuelven el color blanco a la minishell.

FUNCIONES

En esta sección se hará una breve explicación sobre el motivo de cada función utilizada en el código. Se recuerda que el archivo con el código fuente está comentando y que los nombres de las variables y funciones son muy descriptivos.
Las líneas en **negrita** son los prototipos de las funciones y las líneas en *cursiva* su explicación.

void imprimirPrompt();
Imprime el prompt con su correspondiente ruta y a color.
void cerrarPipes(int primero, int medio, int ultimo, int pipes, int i, int n);**
Cierra correctamente las pipes dependiendo si la función es llamada desde un primer comando, un comando intermedio o el último comando de una pipe.
void redireccionEntrada(tline* line);
Abre archivos e informa de errores.
void redireccionSalida(tline* line);
Guarda archivos e informa de errores.
void redireccionError(tline* line);
Saca por la salida de error estándar datos
struct Estado* crearEstado(char *nombreComando, int id);
Crea un nodo en la lista enlazada inicializado con los datos proporcionados.
void insertarEstado(char *nombreComando,int id);
Inserta el nodo en el orden correcto.
void imprimirEstados();
Imprime los nodos de la lista enlazada.
int borrarNodo(struct Estado *nodo);
Borra un nodo de forma correcta en la lista enlazada.
void fgUltimoNodo();
Borra el último nodo de la lista enlazada, imprime el nombre del comando y se espera por él.
void imprimirErrores(int error, tline* line);
Mediante un código de error muestra un mensaje por pantalla para informar al usuario.
void validarComandos(tline* line);
Comprueba que los comandos introducidos existan, en otro caso, informa del error.
int comprobarComando(char* comando);
Comprueba si es un comando interno de la Shell. Devuelve verdadero en ese caso.
int comprobarBackground(tline* line, pid_t pidbg);
Se encarga de dar una entrada nueva a la lista enlazada en caso de que el comando esté en background.
void manejador_background(int sig);
Cuando un comando en background acaba, se informa de su finalización y se borra su nodo.
void liberaCabeza();
Libera la memoria del primer nodo de la lista enlazada.
int existeEnBackground(tline* line);
Comprobación de si un comando se encuentra en la lista enlazada.
char* devuelveNombre(int numero);
Recogemos el nombre de un comando que se encuentre en la lista enlazada.
void borrarNodoID(int idBorrar);
Libera un nodo de la lista enlazada.
void manejador_sigint(int sig);
Controla la señal Ctrl+C y mata procesos en caso de estar en primer plano.

CÓDIGOS DE ERROR

Los códigos de error o control de errores se han gestionado mediante una función que detecta el error e imprime el mensaje correspondiente por pantalla.
Se ha conseguido diferenciar una gran variedad de errores concretando qué lo ha podido ocasionar o cómo corregirlo.
Gracias a esta función de gestión de errores se ha estructurado bastante código y se ha evitado la duplicidad del mismo, evitando también fallos a la hora de informar al usuario final sobre lo sucedido.

Se puede encontrar el prototipo de esta función en la sección del código de Funciones llamada **imprimirErrores( ),** y se puede ver su implementación completa debajo de la función **main( )** donde residen las demás funciones del programa.

En la Tabla 3 se muestra los errores contemplados y su mensaje correspondiente.

CÓDIGO DE ERROR	MENSAJE
1	El comando cd debe utilizarse solo.
2	Demasiados parámetros recibidos.
3	No se ha podido cambiar el directorio de trabajo. El directorio introducido [COMANDO] no existe.
4	[COMANDO]: No se encuentra el comando.
5	Ha ocurrido un error con la creación del proceso hijo.
6	Error en la lectura de comandos.
7	Saliendo de la MiniShell.
9	No se ha encontrado el proceso.

Tabla 4: mensajes de error

CONTROL DE SEÑALES

En los requisitos del proyecto se pide controlar las señales encargadas de finalizar procesos con el objetivo de no poder matar la minishell mientras está en uso.

Para ello, se ha modificado las reacciones de SIGQUIT y SIGINT.

La señal SIGQUIT será ignorada durante todo el proceso, sin embargo, la señal SIGINT será activada y desactivada a conveniencia, pues se deben poder matar los propios procesos que crea nuestro programa.

En un primer momento se dejará la señal SIGINT desactivada (para no matar el programa), y en caso de la ejecución de algún comando en primer plano, se activará llevándola a un manejador propio. Cuando se produzca un comando en segundo plano (explicado al final de esta sección) se desactivará la señal para no poder matar ese proceso.
Si se quisiera matar dicho proceso en segundo plano, primero deberemos traerlo al primer plano con el comando interno *fg* y por último escribir Ctrl+C.

En cuanto a los procesos en segundo plano, se ha conseguido su comprobación de estado reescribiendo la señal SIGCHLD. Se ha creado un manejador propio que proporciona entradas a una estructura de lista enlazada con la que se controlan los procesos en segundo plano y en caso de existir éste aún en la estructura, sacarlo ordenadamente, liberar la memoria e informar.

COMENTARIOS PERSONALES

En cuanto a la organización de la práctica ha sido muy fácil dado que hemos funcionado muy bien en equipo y ha habido mucha comunicación durante todo el proceso. Hemos dividido las tareas y se ha ido trabajando en paralelo a la vez que nos íbamos actualizando los cambios hechos.
Hemos trabajado siempre partiendo de la misma versión del código (ya que hemos llegado a tener hasta 7 versiones debido a ir aumentando las funcionalidades de la Shell).
Hemos sido lo más ordenados posible y dejando siempre comentarios explicativos para que el compañero pueda entender el nuevo código.

Un gran punto de inflexión fue las pipes, ya que estuvimos varios días atascados viendo cómo podríamos hacerlo. Tuvimos un gran problema con el tema de liberar la memoria y cerrar todas las pipes al final del proceso padre, pues es algo que no habíamos incluido y nos estaba dando bastantes problemas. Al final nos dimos cuenta de ese fallo y lo corregimos y pudimos avanzar.

Otro problema con el que nos topamos fue el comando Jobs. Estuvimos los dos miembros del grupo trabajando en ello porque no sabíamos en un principio usar el atributo WNOHANG. Sabíamos de su existencia por las clases pero no dónde colocarlo o cómo utilizarlo. Estuvimos también varios días dándole vueltas a todo el tema de background y Jobs.

Al final conseguimos resolver todos los problemas bastante antes de la entrega y hemos estado desde entonces probando la minishell y buscando bugs.

CONCLUSIONES

La práctica de la minishell es dura, pues el lenguaje C tiene sus complicaciones y más aún sin poder hacer debug y ver dónde se atasca el código o qué contiene cada variable.
Lo que más se ha echado de menos es la comodidad de un compilador, pues se tarda mucho en detectar y corregir errores de código o funcionamiento de algoritmos.

Por otra parte, el hecho de contar con la librería proporcionada y con las estructuras dadas ha hecho mucho más asequible la práctica, pues ha sido crucial a la hora de definir el código. Ha sido de gran ayuda contar con ello.

Finalmente, la realización de la práctica te da un dominio sobre los temas estudiados de creación y gestión de procesos y comunicación entre ellos, un conocimiento totalmente necesario para aprobar la asignatura y entender lo básico sobre Sistemas Operativos.