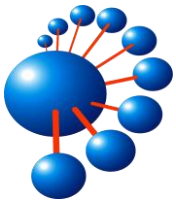


Departamento Ingeniería Informática y Ciencias de la Computación

Ingeniería Civil Informática

Sistemas Operativos

Desarrollo de un Intérprete de Comandos en Linux

Integrantes

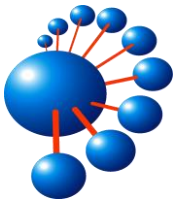
Rodrigo Bascuñán León

Jesús Guevara Salcedo

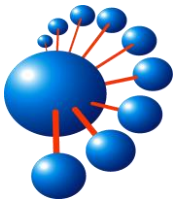
Marcos Martínez Rojas

Gabriel Castillo Castillo

Concepción, 09 de septiembre del 2024



Contenido



Introducción

Una Shell es una herramienta del sistema operativo que nos permite interactuar con él, como mediador entre el usuario y el núcleo del sistema operativo, provee comandos o instrucciones con parámetros específicos que permiten actuar sobre los recursos del sistema, como archivos, procesos y dispositivos.

El objetivo de esta tarea es desarrollar una Shell funcional para Linux que permita la ejecución de comandos de manera eficiente y que incorpore funcionalidades adicionales específicas como el guardado de los comandos mediante un comando personalizado llamado *favs*, y la creación de recordatorios tipo alarma con el comando set alarma.

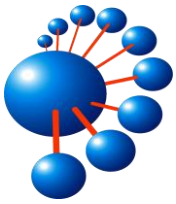
Metodología de trabajo

Para el trabajo, hemos seguido una ruta de planificación para crear e implementar todas las funcionalidades solicitadas por el docente. Dentro de los recursos que hemos utilizado para lo anterior se encuentran las transparencias entregadas por los docentes del curso para comprender la teoría, los códigos entregados en los laboratorios para la comprensión de uso de las diversas llamadas a sistema utilizadas en el proyecto, y la herramienta *Valgrind* para la detección y corrección de algunos errores imprevistos dentro de la programación.

Para acceder a nuestro proyecto acceda a: https://github.com/StaCKm29/Shell

En el repositorio encontrará más instrucciones sobre su uso.

Universidad de Concepción



Diseño del Sistema

Shell

La *shell* desarrollada, denominada *"OhMyShell"*, presenta un *prompt* que muestra al usuario la ruta actual del directorio en el que se encuentra. Para lograr esta funcionalidad, se utilizó la función *getcwd()*, la cual obtiene la ruta del directorio de trabajo y la imprime como parte del *prompt*, proporcionando una referencia visual clara.

El programa está controlado por un bucle *while*, que permite que la *shell* esté siempre en ejecución continua, manteniéndose lista para recibir y procesar comandos en cualquier momento. Esto garantiza que la *shell* no finalice hasta que el usuario decida salir explícitamente con el comando *exit.*

Para la lectura de los comandos, se utiliza la función *fgets()*, que permite capturar la entrada del usuario desde el teclado, almacenarla en un buffer y eliminar el salto de línea al final, lo que facilita el procesamiento correcto de los comandos ingresados. Este input es luego enviado a la función *tokenPipes()*, donde se procesa la entrada para identificar y separar los comandos que están conectados por pipes (|).

Manejo de pipes

Para el manejo de pipes implementamos una función llamada *tokenPipes()* la cual se encarga de segmentar la entrada de la terminal ingresada por el usuario y subdividirla en comandos.

Manejo de espacios

Una vez listos los comandos, se utiliza la función *tokenEspacios()* para buscar los espacios dentro del comando y así subdividirlo en argumentos para ser manejados e interpretados más tarde.

Exploración y manejo de archivos

Como toda shell, esta incorpora la funcionalidad de poder recorrer directorios con el comando cd (mediante el uso de *chdir()*), a su vez como el uso de otros comandos útiles como el *ls* o el *wc* (estos últimos gracias al uso de *execvp()*).

Función Favoritos

La función *favs* permite gestionar los comandos favoritos del usuario.

Para el uso e implementación de la función fue necesario crear un *struct* que guardase los comandos favoritos junto con un contador de comandos.

Al momento de ejecutarse un comando, si el primer argumento del comando es *"favs"*, se llama a la función *elegirFavs()*, la cual toma el segundo argumento ingresado por el usuario y, con base en este, determina cuál función específica debe ejecutar. Dependiendo de los subcomandos que acompañen a *favs*, esta función invocará el comportamiento adecuado.

Dentro de las principales funcionalidades está *favs* crear, que crea un archivo en un directorio proporcionado en donde se guardarán todos los comandos que son detectados como favoritos, esta funcionalidad está implementada en la función *crearArchivo()*, la cual recibe como argumento un puntero al *struct favs* y una ruta. La función abre o crea un archivo en la ubicación proporcionada por el usuario, y dentro de él guarda todos los comandos favoritos almacenados en la estructura *favs.*

También se encuentra *favs eliminar*, lo que permite la eliminación de un comando en cuestión. Para implementar esta funcionalidad se verifican el 3er argumento ingresado (dos índices de los comandos a eliminar). Una vez eliminado el comando se reajusta el arreglo de favoritos y se libera la memoria usada para el comando eliminado, para después decrementar el contador de comandos. Otra forma de eliminar comandos de la lista de favoritos es *favs borrar*, pero al ejecutar este comando, borra todos los comandos a diferencia de *favs eliminar* que solo tienes la opción de borrar dos, esto se hace sobrescribiendo el archivo.

Adicionalmente a *favs eliminar*, se encuentran otros comandos encargados del manejo del archivo de guardado el primero de ellos es *favs cargar*, el cual en base a una ruta de direccionamiento buscará el archivo con los comandos favoritos guardados y lo cargará en caso de existir. Y el segundo comando es *favs guardar*. Este comando se encarga de abrir el archivo donde se almacenan los comandos en modo escritura y procede con el guardado de los comandos, una vez finalizado el proceso se despliega la ruta de guardado de los comandos favoritos.

Otro de los subcomandos de *favs* es *favs [num] ejecutar*, este comando ejecutará el comando favorito con el enumerado correspondiente y la función encargada de esta es, *ejecutarComando()*, esta función utiliza un proceso *fork* para crear un proceso hijo, que será el encargado de ejecutar el comando seleccionado. Dentro del proceso hijo, se utiliza *execvp()* para reemplazar el código del proceso con el comando favorito correspondiente. Si la ejecución del comando tiene éxito, el hijo ejecuta el comando en cuestión; si no, imprime un mensaje de error y termina su ejecución. Mientras tanto, el proceso padre utiliza *wait()* para esperar a que el proceso hijo finalice, asegurando que no se continúe con otras tareas hasta que el comando se haya ejecutado completamente.

Adicionalmente se encuentra el comando *"favs buscar"*, el cual recibe un *string*, para así desplegar todos aquellos comandos que contengan el *string* ingresado.

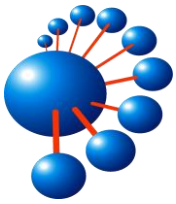
Otro comando que posee gran utilidad es *favs mostrar*, cuya función es recorrer la lista de comandos e imprimirlos de forma ordenada con su número correspondiente.

Función Alarma

La función alarma recibe como mínimo 4 argumentos, los dos primeros deben de ser *"set" "alarma"* para hacer el uso de la función, el 3ro debe de ser un

número que será el tiempo en segundos que esperará la alarma y el 4to argumento en adelante serán considerados como mensajes.

Para la creación de esta función primero implementamos una validación de los primeros 3 argumentos, una vez eso listo se procede con el armado del mensaje a desplegar (para ello utilizamos memoria dinámica para evitar errores en mensajes largos) concatenando con espacios entremedio a todos los argumentos desde el 4 en adelante. Una vez listo esto se llama a la función *setAlarma* (dicha función recibe el tiempo y el *string* con el mensaje a desplegar) la cual junto a un manejador de señales y la llamada a sistema *"alarm()"* logran el objetivo de implementar una alarma con mensaje y temporizador.



Conclusión

El proyecto dado ha sido un gran desafío para implementarlo correctamente, sin embargo, este proceso ha contribuido al aprendizaje de manera excepcional. Poder implementar funciones como *fork()*, *execvp()*, *wait()*, además de implementar un *pipe* en un proyecto real ha conllevado a un mejor entendimiento de las llamadas a sistema y de cómo trabaja un intérprete de comandos en un sistema operativo.