

Proyecto DWIMSH

Semana: 10

Estudiante:

Alberth Godoy

12111345

Sede de Estudio:

UNITEC TGU

Docente:

Doc. Roman Arturo Pineda Soto

Clase:

Sistemas Operativos I

Sección:

517

Fecha de entrega:

viernes 28 de marzo de 2025

Indice

Introducción	4
Descripción del Proyecto	4
Corrección de errores usando distancia de Levenshtein	4
Sugerencia de comandos alternativos	6
Manejo de procesos	7
Animación ASCII	7
Manejo de SIGINT	8
Uso del Proyecto	8
Sistema Operativo	8
Dependencias	8
Compilación	9
Ejemplos de uso	9
Descripción de Rutinas y Procedimiento	11
Setup():	11
Levenshtein:	11
find_closet_command()	11
show_welcome_banner() y show_goodbye_banner()	12
handle_SIGINT()	12
load_animation() y show_animation()	12

load_commands().....	12
Comentarios de Implementación	13
¿Porque se uso el algoritmo levenshtein?	13
Desafíos y Soluciones	13
Manejo de Señales:	13
Manejo de Duplicado de comandos	13
Mejoras en la búsqueda de comandos.....	13
Conclusión	14

Introducción

En el mundo de los sistemas operativos *nix, la interfaz de usuario, o shell, juega un papel crucial al permitir a los usuarios interactuar con el kernel. La capacidad de personalizar y programar estos shells ofrece un entorno operativo flexible y adaptable a las necesidades individuales. Este proyecto se enfoca en la creación de un shell personalizado, "dwimsh", para el sistema operativo Minix, con el objetivo de mejorar la experiencia del usuario mediante la corrección de errores de comandos y la sugerencia de alternativas viables. A través de este proyecto, exploraremos las llamadas al sistema POSIX y nos adentraremos en la programación de entornos *nix, aplicando conceptos como la búsqueda de similitudes entre palabras y la gestión eficiente de comandos.

Descripción del Proyecto

Corrección de errores usando distancia de Levenshtein

La shell implementa un sistema de corrección de errores utilizando el algoritmo de distancia de Levenshtein, que mide la similitud entre dos cadenas de texto. Los componentes principales son:

```
/**
 * @brief
 * Calcula la distancia de Levenshtein entre dos cadenas
 * @param s1 La primera cadena
 * @param s2 La segunda cadena
 * @return
 * La distancia de Levenshtein entre las dos cadenas
 */
int levenshtein(const char* s1, const char* s2)
{
    int len1 = strlen(s1);
    int len2 = strlen(s2);
    int matrix[len1 + 1][len2 + 1];

    for (int i = 0; i <= len1; i++) {
        matrix[i][0] = i;
    }
    for (int j = 0; j <= len2; j++) {
        matrix[0][j] = j;
    }

    for (int i = 1; i <= len1; i++) {
        for (int j = 1; j <= len2; j++) {
            int cost = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
            matrix[i][j] = fmin(fmin(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1), matrix[i - 1][j - 1] + cost);
        }
    }
    return matrix[len1][len2];
}
```

Figure 1

```

char* find_closest_command(const char* input) {
    char* closest_command = NULL;
    int min_distance = INT_MAX;

    if (strlen(input) < 2) {
        return NULL;
    }

    for (int i = 0; i < num_commands; i++) {
        int len_diff = abs(strlen(input) -
        strlen(command_list[i]));
        if (len_diff > 2) continue;

        int distance = levenshtein(input,
        command_list[i]);
        if (distance < min_distance && distance
        <= (strlen(input) / 3 + 1)) {
            min_distance = distance;
            free(closest_command);
            closest_command = strdup(
        command_list[i]);
        }
    }
    return closest_command;
}

```

Figure 2

Sugerencia de comandos alternativos

Cuando un comando no se encuentra, el sistema:

- Busca comandos similares en la lista de comandos disponibles
- Pregunta al usuario si quiere ejecutar el comando sugerido
- Permite confirmar con 's/S' o 'y/Y'

```

void load_commands() {
    DIR* dir;
    struct dirent* entry;

    for (int i = 0; i < num_commands; i++) {
        free(command_list[i]);
    }
    num_commands = 0;

    if ((dir = opendir("/usr/bin")) != NULL) {
        while ((entry = readdir(dir)) != NULL &&
            num_commands < MAX_COMMANDS) {
            if (entry->d_name[0] != '.') {
                int exists = 0;
                for (int i = 0; i <
                    num_commands; i++) {
                    if (strcmp(command_list[i],
                        entry->d_name) == 0) {
                        exists = 1;
                        break;
                    }
                }
                if (!exists) {
                    command_list[num_commands] =
                        strdup(entry->d_name);
                    num_commands++;
                }
            }
        }
        closedir(dir);
    }
}

```

Figure 3

Manejo de procesos

La shell maneja procesos en:

- Primer plano (foreground):
- El shell espera a que el proceso termine
- Se implementa usando wait(NULL)

Animación ASCII

El sistema tiene dos animaciones creadas con ASCII, son animaciones al ejecutar la Shell y al salir de esta. Los frames son cargados desde un json.

Manejo de SIGINT


La Shell implemente un manejador personalizado para la señal SIGINT que es usando el atajo de teclas “Control + C”

Uso del Proyecto

Sistema Operativo

- UNIX/Linux (basado en las librerías incluidas)
- Compatible con sistemas tipo UNIX

Dependencias



```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <dirent.h>
#include <limits.h>
#include <math.h>
#include <ctype.h>
#include <signal.h>
#include <errno.h>
#include <bits/sigaction.h>
```

Figure 4

Archivos necesarios:

- ascii-frames.json (para animación de bienvenida)
- ascii-frames-end.json (para animación de despedida)

Compilación

Comando de compilación:

```
gcc -o dwimsh shell.c -lm
```

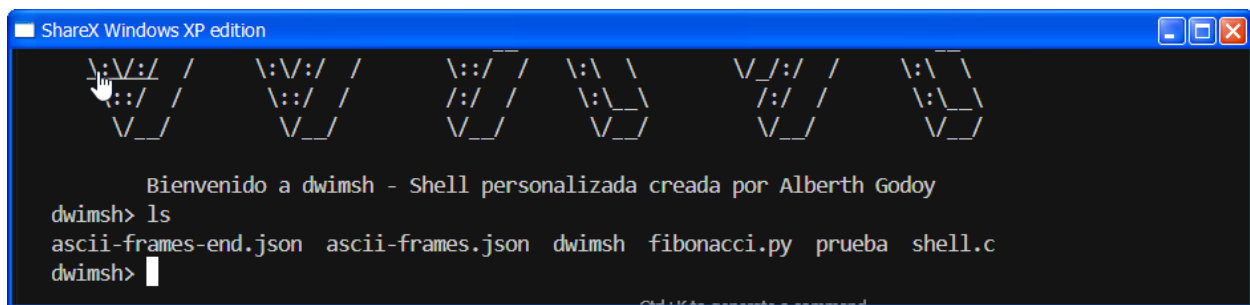
Luego inicial la shell:

```
./dwimsh
```

Puedes terminar el programa con:

```
Exit o Control + C
```

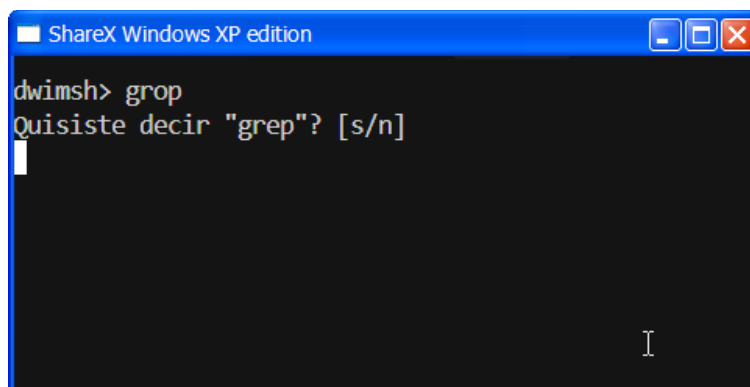
Ejemplos de uso



```
ShareX Windows XP edition

\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /
\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /
\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /
\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /
\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /
\:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ / \:\V:\ /

Bienvenido a dwimsh - Shell personalizada creada por Alberth Godoy
dwimsh> ls
ascii-frames-end.json  ascii-frames.json  dwimsh  fibonacci.py  prueba  shell.c
dwimsh> 
```



```
ShareX Windows XP edition

dwimsh> grop
Quisiste decir "grep"? [s/n]

```

```

ShareX Windows XP edition
dwimsh> comandoQuenoExiste
Comando no encontrado
dwimsh>

```

```

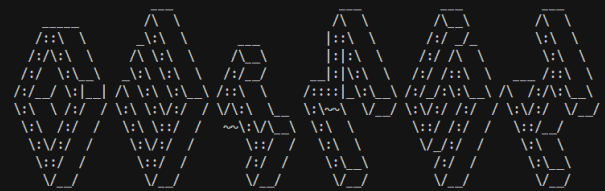
ShareX Windows XP edition
¡Gracias por usar dwimsh! ¡Hasta la próxima!
¡Que tengas un excelente día!

```

```

ShareX Windows XP edition
alexvillag@LAP:UP~$

```



```

Bienvenido a dwimsh - Shell personalizada creada por Alberth Godoy
dwimsh> cat fibonacci.py

def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

if __name__ == "__main__":
    n = int(input("Ingrese un número: "))
    print(f"El número {n} en la secuencia de Fibonacci es: {fibonacci(n)}")
dwimsh>

```

Descripción de Rutinas y Procedimiento

Setup():

El código analiza la línea de comando ingresada por el usuario, separando los argumentos mediante espacios y tabuladores como delimitadores. Identifica el inicio y fin de cada argumento, convirtiendo la entrada en un array de strings (`args[]`). Además, , añade terminadores nulos (`\0`) para garantizar strings válidos en C, y limita el número de argumentos a $\text{MAX_LINE}/2 + 1$ para evitar desbordamientos de memoria.

Levenshtein:

Este código implementa la distancia de Levenshtein, calculando el número mínimo de operaciones requeridas para transformar una cadena en otra. Considera tres operaciones fundamentales: inserción, eliminación y sustitución de caracteres. Para lograr una solución óptima, utiliza programación dinámica mediante una matriz de estados que almacena resultados intermedios, evitando así recálculos redundantes. Finalmente, el algoritmo retorna el valor numérico que representa la distancia mínima entre las cadenas comparadas, siendo 0 cuando las cadenas son idénticas.

find_closest_command()

Esta función busca el comando más similar al input del usuario mediante dos criterios principales: primero una diferencia máxima de longitud de 2 caracteres y segundo una distancia de Levenshtein que no exceda $(\text{longitud_input}/3 + 1)$. El algoritmo compara sistemáticamente la entrada del usuario con la lista de comandos disponibles, evaluando primero la longitud para

filtrar opciones claramente distintas, y luego calculando la distancia de edición para las candidatas restantes. Retorna el comando más cercano que cumpla ambos criterios de similitud, o NULL cuando no encuentra sugerencias válidas

show_welcome_banner() y show_goodbye_banner()

Funciones que cargan los llamados para la animación al ejecutar la aplicación y al salirse de ella.

handle_SIGINT()

Maeja para poder salir con atajo de teclas, en este caso Control + C

load_animation() y show_animation()

Funciones donde load_animation, cargo por frame el formato json, siendo que cada “[“ es el comienzo de un frame y “]” es el cierre de un frame, luego show_animation() da un espacio en la terminal para ir borrando y poniendo cada frame cada cierto tiempo para simular una animación de video con arte ASCII.

load_commands()

Este módulo realiza un escaneo completo del directorio /usr/bin para construir una lista actualizada de comandos disponibles. Inicia limpiando cualquier contenido previo en command_list[] para garantizar integridad de datos. Durante el proceso, lee sistemáticamente cada entrada del directorio, aplicando los siguientes filtros: primero excluye archivos ocultos (que comienzan con '.'), segundo evita duplicados mediante verificación de existencia previa, y (3) almacena únicamente nombres válidos en el array command_list[]

Comentarios de Implementación

¿Porque se uso el algoritmo levenshtein?

El sistema de sugerencias implementado en "dwimsh" se basa en el algoritmo de Levenshtein, elegido por su capacidad para ofrecer una mayor precisión en la identificación de comandos similares en comparación con métodos de comparación simples. Este algoritmo proporciona un equilibrio óptimo entre rendimiento y exactitud, lo que resulta crucial para una experiencia de usuario fluida y eficiente.

Desafíos y Soluciones

Manejo de Señales:

Para manejar interrupciones durante la lectura de comandos en "dwimsh", se implementa un mecanismo que verifica si la función `read()` fue interrumpida por una señal (EINTR). En tal caso, el shell reintenta la lectura del comando, asegurando una operación continua y robusta.

Manejo de Duplicado de comandos

Evitar duplicados y comandos inválidos, para esto se añadió una verificación para la existencia del comando y la gestión dinámica en memoria.

Mejoras en la búsqueda de comandos

Se pudo haber mejorado el sistema de búsqueda de comando por comandos mas usados dado que al intentar algún comando, que es de mucho uso tipo "ls", al escribirlo mal como "sl", recomendando "nl", al escribir "lss" recomienda "less", así que seria adaptar al código y darle prioridad aquellos comandos mas usados.

Conclusión

El desarrollo de "dwimsh" ha proporcionado una valiosa experiencia en la programación de shells y la manipulación de llamadas al sistema en entornos *nix. A lo largo del proyecto, hemos abordado desafíos como la corrección de errores de comandos, la sugerencia de alternativas y la gestión eficiente de la ejecución de programas. La implementación de algoritmos para la búsqueda de similitudes entre palabras y la creación de una tabla de comandos en memoria han sido fundamentales para lograr un shell funcional y eficiente. Este proyecto no solo demuestra la capacidad de crear herramientas personalizadas para mejorar la interacción con el sistema operativo, sino que también resalta la importancia de comprender a fondo los fundamentos de la programación en entornos *nix.