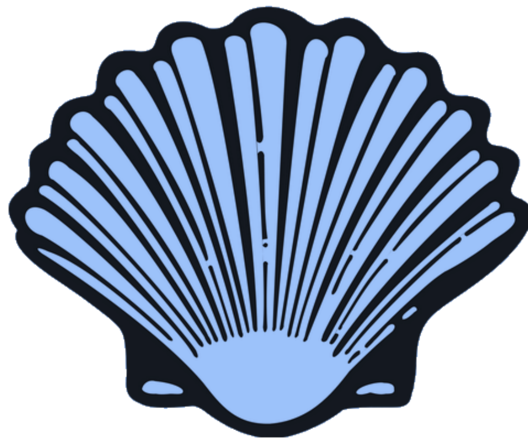
PL2 ~ 2024-25:

PROYECTO MINISHELL

PRIMER CUATRIMESTRE



Sistemas Operativos – Julia María Clemente Párraga 2/12/2024 – Grupo 4

Luis Miguel Herrá Alpuente – luis.herra@edu.uah.es – 06612001F

Miguel de Sande Moreno - miguel.sande@edu.uah.es - 09102482W

Matías Vázquez Herboso - matias.vasquez@edu.uah.es - Y4507803R

Marcos Bonilla de las Morenas – marcos.bonilla@edu.uah.es -   09105928K

Índice

Consideraciones previas 3

Fase 1 4

Fase 2.. 5

Fase 3 7

Fase 4.. 9

Fase 5.. 11

Fase 6.. 14

Fase 7.. 17

Bibliografía, Añadidos, valoración 20

Consideraciones Previas

El comportamiento de la Minishell y su implementación sigue lo establecido en el enunciado de la PL2. Las funcionalidades prestadas por la Minishell serán las siguientes:

- Ejecución de órdenes internas
- Ejecución de órdenes externas
- Múltiples órdenes en una única línea.
- Redirecciones (<, >, <<)
- Tuberías
- Creación de variables de entorno
- Manejo de señales (Ctr+C,Ctr+Z,Ctr+D)

Ante comandos no disponibles la Shell mostrará este comando como no inexistente.

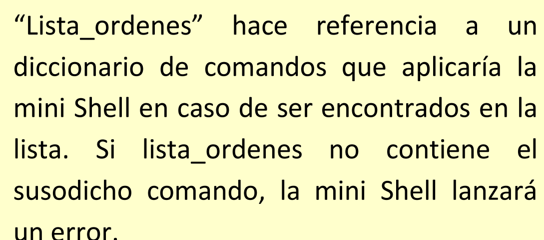
El {comando} no existe. Compatible únicamente con sistemas Linux.

Los roles ejercidos por cada miembro: Líder (L), Editor (E), Documentalista (D), Programador (P)

Fase 1: L: Luis D: Marcos E: Miguel P: Matías

El equipo dedicó gran parte de la jornada a investigar los comandos POSIX e indagar en los contenidos que el profesorado ofrecía para llevar a cabo la práctica. Antes de ver las funciones que nos aportaban, tras hacer un repositorio común en GitHub y administrar las tareas, diseñamos este pseudocódigo:

```
"Ordenes Minishell:
mientras 1:
    imprime_prompt()
    x = lee_orden()
    si x es "exit" entonces
        break
    si no:
        si x en lista_ordenes:
            ejecuta_orden
        fin si
    fin si
fin mientras"
```



Finalmente el código que se implementó para concluir esta primera parte es el siguiente

```
int main(int argc, char *argv[])
{
    char buf[BUFSIZ];
    while (1)
    {
        imprimir_prompt();
        leer_linea_ordenes(buf);
        if (strcmp(buf, "exit") == 0)
        {
            break;
        }
        else
        {
            if (es_ord_interna(buf))
            {
                ejecutar_ord_interna(buf);
            }
            else
            {
                ejecutar_linea_ordenes(buf);
            }
        }
    }
    return 0;
}
```

Funcionalidad:

Se reserva espacio en memoria para una variable cadena de texto que almacenará la orden ingresada por el usuario, leída por leer_linea_ordenes. En caso de que la orden sea "exit" se finalizará la ejecución de la mini Shell. Mediante es_ord_interna verificamos si la orden es interna o externa, es decir, está definida en nuestro propio código (son más rápidas) o recurre a contenido externo.

Órdenes internas: cd, echo, export, exit, set. *Órdenes externas*: ls, grep, cat, cp…

Conclusiones:

Tras probar el código, se ha llegado a la conclusión de que funciona de manera fructífera y ya es capaz de ejecutar órdenes como "**pwd**", "**cd**", "**clean**" (ya que estas se tratan de órdenes internas) y, también "**exit**" para parar la ejecución.

Fase 2: L: Marcos D: Miguel E: Matías P: Luis

Hemos editado el archivo "ejecutar.c" para completar las funciones ejecutar_orden() y ejecutar_linea_ordenes(). En las órdenes internas no se crean procesos hijos.

```
pid_t ejecutar_orden(const char *orden, int *backgr) {
    char **args;
    pid_t pid;
    int ind_entrada = -1, ind_salida = -1;  // Para después

    args = parser_orden(orden, &ind_entrada, &ind_salida, backgr);
    if (args == NULL) {
        return -1;
    }

    // Crear un proceso hijo.
    pid = fork();
    if (pid == -1) {
        printf("Error al crear proceso hijo.\n");
        free_argumentos(args);
        return -1;
    }

    if (pid == 0) {
        // En el hijo: Ejecutar la orden.
        if (execvp(args[0], args) == -1) {
            printf("Error al ejecutar la orden.\n");
            exit(EXIT_FAILURE);  // Salir si exec falla.
        }
    }

    free_argumentos(args);
    return pid;
}
```

Funcionamiento: Esta función recibe dos parámetros: una cadena que contiene el comando a ejecutar y un puntero a un entero que indica si el proceso debe ejecutarse en segundo plano. Inicialmente, llama a la función parser_orden() pasando como argumento la orden recibida; esta función devuelve un array con los argumentos separados y gestiona las redirecciones correspondientes. Tras el análisis, se verifica si el parsing fue exitoso; en caso contrario, la función retorna -1 para indicar un error. Con el array args correctamente procesado, se procede a crear un proceso hijo utilizando el servicio fork(), cuyo PID se almacena en una variable. Si ocurre un error durante la creación del proceso, se retorna -1 y se libera la memoria utilizada. Finalmente, si el proceso hijo se genera correctamente, se ejecuta la orden verificando si el PID del proceso es igual a cero.

Una vez sabiendo cómo funciona ejecutar_orden() vemos ejecutar_linea_ordenes():

```
void ejecutar_linea_ordenes(const char *orden)
{
    int backgr = 0;
    pid_t pid = ejecutar_orden(orden, &backgr);

    if (pid > 0 && !backgr) {
        // Si no está en segundo plano, espera al hijo.
        if (waitpid(pid, NULL, 0) == -1) {
            printf("Error al esperar el proceso hijo\n");
        }
    }
}
```

Funcionamiento:

Esta función es la que recibe la cadena de caracteres que componen la orden y se la pasa a ejecutar_orden. Después, comprueba si se quiere ejecutar el proceso en segundo plano. Si es así, llama al servicio waitpid() pasándole el pid del proceso hijo, para que el padre espere a que este termine para continuar. En caso de que ocurra un error al esperar al proceso hijo se imprimirá en pantalla un mensaje indicándolo.

Conclusiones:

Tras probar el código, se concluye que funciona correctamente, ya que es capaz de crear procesos hijos que heredan las características del padre y ejecutan comandos externos de manera exitosa.

Pseudocódigo:

fun ejecutar_orden(orden, backgr):

args = procesar_orden(orden) // Procesar la orden y separar los argumentos

Si args es NULL: dev -1 // Error en el procesamiento fin si

pid = crear_proceso_hijo()

si pid = -1: Liberar memoria; dev -1 // Error al crear el proceso fin si

Si pid = 0: //estamos en proceso hijo

Ejecutar la orden

Si la ejecución dev -1; salir; fin si

Liberar memoria; Retornar pid // Retornar el pid del hijo

Fase 3: L: Miguel D: Matías E: Marcos P: Luis

En esta fase, el objetivo principal fue evitar que los procesos en segundo plano quedaran en estado *zombie*. Un proceso hijo entra en este estado cuando termina su ejecución, pero el proceso padre no recoge su estado, lo que deja recursos del sistema ocupados innecesariamente. Para solucionar este problema, implementamos un manejador de señales que captura la señal SIGCHLD enviada al proceso padre cuando un hijo termina su ejecución.

Se creó la función manejar_sigchild(), que se activa al recibir la señal SIGCHLD, la cual utiliza la función waitpid(-1, &estado, WNOHANG) para recoger el estado de todos los procesos hijos terminados. El parámetro, -1, indica que deben manejarse todos los hijos que hayan terminado, y la bandera WNOHANG asegura que el proceso padre no se bloquee si no hay hijos terminados evitando que el proceso padre se detenga esperando indefinidamente a que los procesos hijos terminen.

```c
static void manejar_sigchild(int signo)
{
    int estado;
    waitpid(-1, &estado, WNOHANG); // Recoger el estado de un hijo terminado
}
```

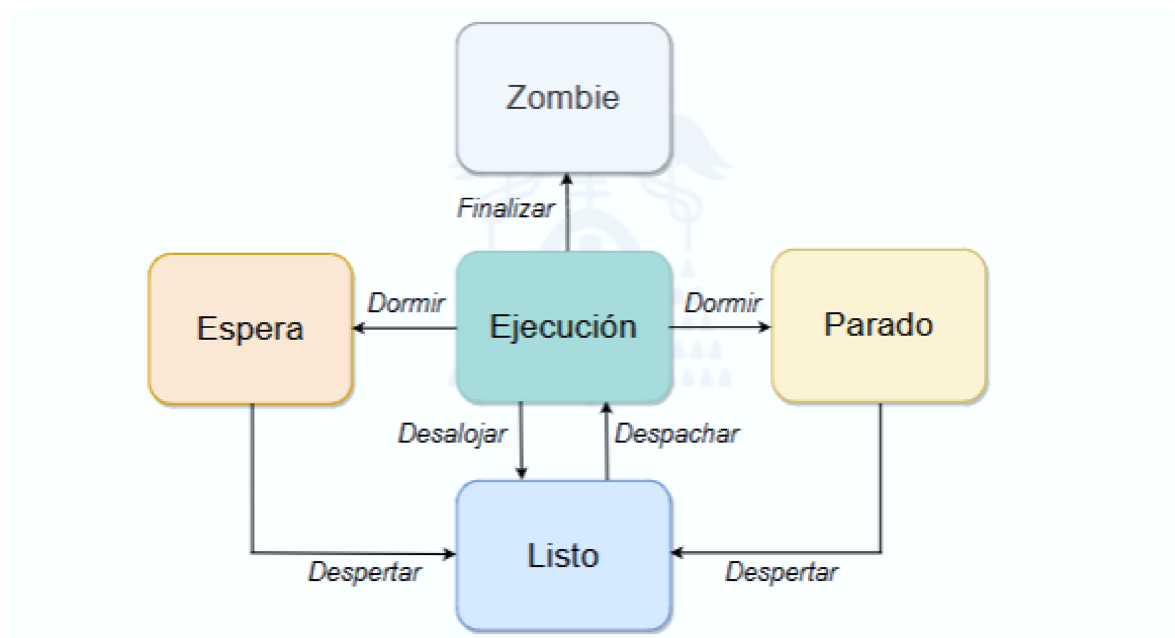
Utilizamos la función sigaction() dentro del main, para instalar el manejador de la señal, ya que ofrece un control más detallado que signal(). En primer lugar, configuramos el manejador asignando la función manejar_sigchild al campo sa_handler encargado de manejar la señal, estableciendo la bandera SA_NOCLDSTOP para evitar que se generen señales de hijos que se detienen sin terminar y SA_RESTART para reanudar las llamadas bloqueantes interrumpidas por señales.

```c
struct sigaction sa;
memset(&sa, 0, sizeof(sa)); // Configuración del manejador para la señal SIGCHLD, inicializado en 0
    sa.sa_handler = manejar_sigchild; // Asignada la función manejadora de la señal
    sa.sa_flags = SA_NOCLDSTOP | SA_RESTART; // No se generan señales cuando un hijo para o reanuda su ejecución
    sigaction(SIGCHLD, &sa, NULL); // Instalar el manejador de señales
```

Además, agregamos #define _POSIX_C_SOURCE 200809L para habilitar las funciones de sigaction() necesarias para el manejo de señales.

Conclusiones:

Con esta implementación, evitamos los procesos zombies, optimizando recursos y manteniendo la Minishell operativa sin bloqueos. Usamos sigaction() para un manejo adecuado de señales, cumpliendo con POSIX, y comprobamos que la minishell seguía funcionando correctamente sin procesos zombies, verificando con el comando "ps" que los procesos en segundo plano se gestionaban adecuadamente y desaparecían después de finalizar.



Fase 4: L: Matías D: Miguel E: Luis P: Marcos

En esta fase el objetivo fue crear un archivo makefile para compilar el proyecto. En él se escriben las reglas donde se le indican las dependencias necesarias para la compilación y así automatizar y facilitar el proceso de compilación.

```
CC = gcc
CFLAGS = -Wall -Wextra -Werror -std=c99 -D_POSIX_C_SOURCE=200809L #Añade los flags: Warnings y la versión del compilador
#Ruta de los archivos .c
RUTA_C = p2/archivosAdicionales_P2/archivos_c_h_apoyo
#Ruta de los archivos .o
RUTA_O =
p2/archivosAdicionales_P2/archivos_objeto_arquitecturas/archivos_objeto_Linu
x_64bits_fPIC

all: debug
debug: CFLAGS+= -g      #Añade el flag de debug
debug: bin/minishell    #Crea el ejecutable "minishell" en la carpeta bin
release: CFLAGS+= -O3   #Añade el flag de optimización (se pone O3 al esta
algo más optimizado que el O2)
release: bin/minishell  #Crea el ejecutable "minishell" en la carpeta bin

.PHONY: clean
clean: #Borra el ejecutable
    rm -rf bin/minishell build/*

build: #Crea la carpeta build
    mkdir -p build

bin: #Crea la carpeta bin
    mkdir -p bin

bin/minishell: bin build build/minishell.o build/libmemoria.o
build/ejecutar.o build/entrada_minishell.o build/redirecciones.o
build/parser.o build/internas.o #Crea el ejecutable minishell
    $(CC) $(CFLAGS) -o bin/minishell build/minishell.o build/libmemoria.o
build/ejecutar.o build/entrada_minishell.o build/redirecciones.o
build/parser.o build/internas.o

build/libmemoria.o: $(RUTA_C)/libmemoria.c #Compila el archivo libmemoria.c
    $(CC) $(CFLAGS) -c $(RUTA_C)/libmemoria.c -o build/libmemoria.o

build/ejecutar.o: $(RUTA_C)/ejecutar.c #Compila el archivo ejecutar.c
    $(CC) $(CFLAGS) -c $(RUTA_C)/ejecutar.c -o build/ejecutar.o

build/entrada_minishell.o: $(RUTA_C)/entrada_minishell.c #Compila el archivo
entra-da_minishell.c
```

```
    $(CC) $(CFLAGS) -c $(RUTA_C)/entrada_minishell.c -o
build/entrada_minishell.o

build/redirecciones.o: $(RUTA_C)/redirecciones.c #Compila el archivo
redirecciones.c
    $(CC) $(CFLAGS) -c $(RUTA_C)/redirecciones.c -o build/redirecciones.o

build/minishell.o: $(RUTA_C)/minishell.c #Compila el archivo minishell.c
    $(CC) $(CFLAGS) -c $(RUTA_C)/minishell.c -o build/minishell.o

build/internas.o: $(RUTA_O)/internas.o
    cp $(RUTA_O)/internas.o build/internas.o

build/parser.o: $(RUTA_O)/parser.o
    cp $(RUTA_O)/parser.o build/parser.o
```

Primero, se definen las siguientes variables: **CC**, que representa el compilador que utilizaremos, en nuestro caso **gcc**; **CFLAGS**, que almacenará las flags que se añaden al comando **gcc** para compilar (en este caso, hemos añadido la flag **"-D_POSIX_C_SOURCE=200809L"**, necesaria para el manejo de procesos hijos, y hemos eliminado la definición **#define "_POSIX_C_SOURCE=200809L"** de **minishell.c**); **RUTA_C**, que guarda la ruta donde se encuentran los archivos **.c**; y **RUTA_O**, que guarda la ruta donde se almacenan los archivos **.o** proporcionados en el trabajo.

A continuación, se definen las instrucciones **all**, **debug** y **release**, que sirven para compilar el proyecto añadiendo flags adicionales, por si se desea compilar de manera especial para realizar debugging o para optimizar una versión final. También se define la instrucción **clear**, que elimina el ejecutable y todos los archivos generados durante la compilación. Además, se crean las instrucciones **bin** y **build**, que se encargan de crear las carpetas necesarias para generar los archivos, en caso de que no existan previamente.

Por último, se definen las instrucciones que generan el ejecutable y las que crean los objetos. Se utilizan las variables **RUTA_C** y **RUTA_O** para simplificar las instrucciones de compilación y facilitar la edición del archivo **Makefile** en caso de que se cambie la ruta donde se encuentran estos archivos.

Fase 5: L: Miguel D: Matías E: Luis P: Marcos

En esta fase, el objetivo principal fue implementar la lectura de varias órdenes en una misma línea, las cuales están delimitadas por el carácter ;. Las modificaciones realizadas afectaron únicamente al archivo minishell.c, y el resultado final fue el siguiente:

```
while (1) {
        imprimir_prompt();
        leer_linea_ordenes(buf);

        char *token;
        const char delim[] = ";";

        token = strtok(buf, delim); // Dividir en órdenes usando ';' como
delimitador
        while (token != NULL) {

            if (strcmp(token, "exit") == 0) {
                exit(EXIT_SUCCESS);
            }
            // Procesar la orden actual
            else if (es_ord_interna(token)) {
                ejecutar_ord_interna(token);
            } else {
                ejecutar_linea_ordenes(token);
            }

            // Obtener el siguiente token
            token = strtok(NULL, delim);
        }
    }
```

Funcionalidad:

En primer lugar, se define un puntero a la variable token, el cual es un nombre simbólico utilizado por la función strtok. Este puntero hace referencia a un "elemento" o "ficha" dentro de la cadena de texto. Luego, establecemos el carácter delimitador ; como una constante, ya que no se espera que este valor cambie. La función strtok se encarga de separar la cadena en diferentes elementos utilizando este delimitador. La función continuará dividiendo la cadena hasta que no haya más elementos. Si el token obtenido corresponde a una orden (interna o externa), se ejecuta. Luego, con strtok(NULL, delim), se obtiene el siguiente token. La sentencia NULL indica que la función debe continuar procesando desde el último punto de separación. Este método se puede visualizar como un puntero que apunta a los distintos elementos separados por el delimitador.

¿Por qué strtok?

En el grupo se debatió cuál era la opción más adecuada para cumplir con nuestros objetivos. Por un lado, la función strsep (separator de cadenas) fue considerada debido a su mayor legibilidad y su capacidad para manejar multihilos, según algunas fuentes como Stack Overflow. Sin embargo, strtok se mostró como la mejor opción debido a su portabilidad (aunque este aspecto no fue determinante en nuestro caso) y su mayor control sobre el flujo del programa. Uno de los factores clave que influyó en nuestra decisión fue el siguiente ejemplo:

Si el usuario introduce una cadena con dos punto y coma consecutivos.

Por ejemplo: clean;;clean

- Usando strtok, los tokens serían: 'clean' y 'clean'.
- Usando strsep, los tokens serían: 'clean', "" (vacío) y 'clean'.

El uso de strtok evita que se ejecute innecesariamente una orden vacía entre los dos punto y coma (;;), lo que mejora el comportamiento de la shell.

Por otro lado, también se consideraron otras alternativas como memchr, strcspn y memcpy. Sin embargo, estas opciones fueron descartadas rápidamente, ya que no requeríamos un nivel de control tan fino para esta operación en particular. Además, el uso continuo de strtok a lo largo del proyecto hizo que fuera la opción más comprensible y adecuada para el grupo.

Al probar su funcionamiento, detectamos errores relacionados con espacios en blanco al inicio y al final de las cadenas. El programa interpretaba los espacios iniciales o finales como caracteres nulos (NULL), lo que provocaba que entrara en la condición de parada del bucle, deteniendo la ejecución.

Ejemplo: clean; ls

Para solucionar este inconveniente, desarrollamos la función auxiliar recorta_espacios, que elimina los espacios en blanco al principio y al final de los comandos, sin afectar a los espacios intermedios. Por ejemplo, un comando como: \echo 'Hola Mundo' funcionará correctamente preservando los espacios intermedios.

```
char *recorta_espacios(char *str) {
    // Esta función elimina los espacios iniciales
    while (isspace((unsigned char)*str)) {
        str++;
    }

// Comprobamos si el string está vacío una vez hayamos eliminados los espacios iniciales
    if (*str == 0) {
        return str;
    }

    // Eliminamos los espacios finales
    char *end = str + strlen(str) - 1;
    while (end > str && isspace((unsigned char)*end)) {
        end--;
    }

    // Colocamos el "terminador nulo" al final del string
    *(end + 1) = '\0';
    return str;
}
```

El proceso es sencillo y se basa en la función estándar isspace() de C, que verifica si un carácter es un espacio en blanco. La función opera de la siguiente manera:

Eliminación de espacios iniciales:

Se utiliza un bucle que avanza el puntero str mientras apunte a un carácter considerado como espacio en blanco por isspace().

Verificación de cadena vacía:

Una vez eliminados los espacios iniciales, se comprueba si la cadena está vacía. Si es así, la función devuelve el puntero actualizado.

Eliminación de espacios finales:

Se posiciona un puntero end al final de la cadena (antes del terminador nulo) y retrocede mientras encuentre espacios en blanco.

Marcado del final de la cadena:

Una vez identificada la última posición con un carácter válido, se coloca el terminador nulo (\0) para delimitar correctamente el final de la cadena.

Fase 6: L: Luis D: Marcos E: Miguel P: Matías

En esta fase, el objetivo fue llevar a cabo la implementación de las redirecciones de entrada y salida en la Minishell. Para ello, editamos el código del archivo "redirecciones.c", y se completaron las funciones "Redireccionar entrada" y "Redireccionar salida".

```
void redirec_entrada(char **args, int indice_entrada) {
    int fd = open(args[indice_entrada + 1], O_RDONLY);
    if (fd == -1) {
        printf("Error al abrir archivo para entrada");
        exit(EXIT_FAILURE);
    }
    if (dup2(fd, STDIN_FILENO) == -1) {
        printf("Error al redirigir entrada");
        close(fd);
        exit(EXIT_FAILURE);
    }
    close(fd);
    args[indice_entrada] = NULL;
    args[indice_entrada + 1] = NULL;
}
```

Funcionalidad:

En el caso de "Redireccionar entrada", el funcionamiento es el siguiente: Tras pasarle a la función los argumentos del programa que está ejecutando y el índice en el que se encuentra el operador que indica la redirección, "open()" abre el archivo situado en la posición indice_entrada + 1 con la flag "O_RDONLY" activa (ya que solo permite la lectura) y devuelve un descriptor de archivo "fd" (en el caso de que fd sea -1, se interpretará como un dato anómalo y lanzará un error).

El operador POSIX "dup2()" se ocupa de redirigir el descriptor de archivo "fd" al descriptor estándar de entrada (STDIN_FILENO), y en caso de fallar imprimirá un mensaje de error y cerrará el programa. Una vez se ha concluido la redirección, no se necesita más el descriptor y por lo tanto se procede a cerrar con "close()".

```
void redirec_salida(char **args, int indice_salida) {
    int fd;

    if (strcmp(args[indice_salida], ">>") == 0) {
        // Abrir archivo en modo de agregar (append).
        fd = open(args[indice_salida + 1], O_WRONLY | O_CREAT | O_APPEND,
0644);
    } else if (strcmp(args[indice_salida], ">&") == 0) {
        // Redirección de errores estándar y salida a un archivo
        fd = open(args[indice_salida + 1], O_WRONLY | O_CREAT | O_TRUNC,
0644);
        if (fd == -1) {
            perror("Error al abrir archivo para redirección de errores");
            exit(EXIT_FAILURE);
        }

        // Redirigir tanto la salida estándar (stdout) como los errores
(stderr)
        if (dup2(fd, STDOUT_FILENO) == -1 || dup2(fd, STDERR_FILENO) == -1)
{
            perror("Error al redirigir salida y errores");
            close(fd);
            exit(EXIT_FAILURE);
        }
        close(fd);
        return;  // No continuar con las otras verificaciones de redirección
    } else if (strcmp(args[indice_salida], ">2") == 0) {
        // Abrir archivo para redirección de errores.
        fd = open(args[indice_salida + 1], O_WRONLY | O_CREAT | O_TRUNC,
0644);
    } else {
        // Comportamiento por defecto: sobrescribir salida estándar.
        fd = open(args[indice_salida + 1], O_WRONLY | O_CREAT | O_TRUNC,
0644);
    }

    if (fd == -1) {
        perror("Error al abrir archivo para redirección");
        exit(EXIT_FAILURE);
    }

    // Redirigir la salida estándar (si no se usó `>&` para ambos)
    if (strcmp(args[indice_salida], ">2") == 0) {
        if (dup2(fd, STDERR_FILENO) == -1) {
            perror("Error al redirigir errores");
            close(fd);
            exit(EXIT_FAILURE);
        }
    } else {
```

```
        if (dup2(fd, STDOUT_FILENO) == -1) {
            perror("Error al redirigir salida");
            close(fd);
            exit(EXIT_FAILURE);
        }
    }

    close(fd);

    // Eliminar la redirección de los argumentos.
    args[indice_salida] = NULL;
    args[indice_salida + 1] = NULL;
}
```

Funcionalidad:

En el caso de "Redireccionar salida", el funcionamiento es el siguiente: Al igual que en "Redireccionar entrada", se le pasan a la función los argumentos del programa que está ejecutando y el índice en el que se encuentra el operador que indica la redirección, y tras esto, según el operador especificado, se decide como dirigir la salida:

- Se usa ">>" para agregar contenido a un archivo.

- Se usa ">2" para redirigir los errores a un archivo y sobrescribir su contenido.

- Se usa ">" para redirigir la salida estándar a un archivo y sobrescribirlo.

El operador POSIX "open()" abre el archivo situado en la posición indice_entrada + 1 y con las flags "O_WRONLY" (ya que solo permite la escritura) y "O_CREAT" (para que cree el archivo en caso de no existir este) activas. En el caso de ">>", también activa "O_APPEND" para que escriba el contenido al final del archivo sin sobrescribir el contenido existente, y en el caso de ">2", activa "O_TRUNC" para sobrescribir los archivos. En ambos casos, los permisos del archivo son 0644 ("read" y "write" para el usuario, solo "read" para otros).

En caso de usar ">2", mediante el operador POSIX "dup2()" se redirigen los errores, y abre el archivo en modo sobrescribir. Si se usa ">", redirige solo la salida estándar al archivo indicado.

Si "open()" no puede abrir el archivo, lanzará un error y finalizará la ejecución. Una vez se ha concluido la redirección, no se necesita más el descriptor y por lo tanto se procede a cerrar con "close()".

Fase 7: L: Marcos D: Luis E: Matías P: Miguel

En esta fase, el objetivo fue llevar a cabo la implementación de las tuberías(pipes) en nuestra Minishell. Para ello, editamos el código del archivo "ejecutar.c" modificando las funciones ejecutar_orden, ejecutar_linea_ordenes y añadiendo la función crear_pipes.

En la función ejecutar_orden modificamos la condición para pid == 0.

```
    if (pid == 0) {
        // En el hijo: Configurar redirección de entrada si aplica.
        if (entrada != STDIN_FILENO) {
            dup2(entrada, STDIN_FILENO);
            close(entrada);
        }
        // Configurar redirección de salida si aplica.
        if (salida != STDOUT_FILENO) {
            dup2(salida, STDOUT_FILENO);
            close(salida);
        }
        // En el hijo: Configurar redirecciones, si es necesario.
        if (ind_entrada != -1) {
            redirec_entrada(args, ind_entrada);
        }
        if (ind_salida != -1) {
            redirec_salida(args, ind_salida);
        }

        // Ejecutar la orden.
        if (execvp(args[0], args) == -1) {
            printf("Error al ejecutar la orden.\n");
            exit(EXIT_FAILURE); // Salir si exec falla.
        }
    }
```

Funcionalidad:

A partir del condicional, pip == 0, verificamos estar en el proceso hijo. Posteriormente, verificamos si existe una tubería o una redirección. Entrada, ahora formará parte del argumento de la función ejecutar_orden, por tanto, se modificará también el archivo ejecutar.h. El atributo entrada, es un descriptor de archivo que apunta a un extremo de la tubería. En caso de que la entrada, STDIN, no esté ya ocupada duplica el descriptor entrada para que STDIN apunte al mismo archivo. Ahora redirige la salida estándar al recurso cuyo descriptor de archivo sea entrada. Este funcionamiento se realiza de forma análoga en caso de salida. Posteriormente, verificamos la presencia de redirecciones, es decir, la existencia de los caracteres "<" o ">"; de existir se llamará a los métodos definidos durante la parte 6.

```c
int **crear_pipes(int nordenes) {
    int **pipes = NULL;
    int i;

    // Reservar memoria para las tuberías (nordenes - 1)
    pipes = (int **)malloc(sizeof(int *) * (nordenes - 1));
    if (pipes == NULL) {
        perror("Error al reservar memoria para las tuberías");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < nordenes - 1; i++) {
        pipes[i] = (int *)malloc(sizeof(int) * 2);
        if (pipes[i] == NULL) {
            perror("Error al reservar memoria para una tubería");
            exit(EXIT_FAILURE);
        }

        // Crear cada tubería con pipe()
        if (pipe(pipes[i]) == -1) {
            perror("Error al crear la tubería");
            exit(EXIT_FAILURE);
        }
    }

    return pipes;
}
```

Funcionalidad:

El propósito del método crear_pipes, como su nombre indica, es la creación del número de tuberías necesarias dado el número de órdenes, es decir, el número de procesos a conectar. El atributo **pipes, representa una matriz bidimensional y el entero i el identificador de tubería. Primero, reservamos la memoria necesaria para cada una de las tuberías, n-1. En caso de error en la función malloc, es decir, no se reserve correctamente la memoria para las tuberías notificará un error. Posteriormente, por medio de un bucle for, reservamos dos posiciones (entrada y salida) para cada tubería. Finalmente, asociamos los descriptores a las tuberías mediante el servicio pipe y devolvemos el resultado. Internamente, se verá una matriz bidimensional: pipes[0][0],pipes[0],[1],pipes[1][0], pipes[1][1], pipes[2][0]…

La primera columna representa el índice del bucle que fue empleado para asignar memoria a la tubería y la segunda el descriptor de archivo, lectura o escritura, en función de la entrada o salida. Es decir, pipes[i][0/1].

```
void ejecutar_linea_ordenes(const char *orden) {
    char **ordenes;
    int nordenes;
    pid_t *pids;
    int **pipes;
    int backgr = 0;

    // Dividir la línea de órdenes en órdenes individuales.
    ordenes = parser_pipes(orden, &nordenes);
    if (ordenes == NULL) {
        perror("Error al dividir la línea de órdenes.");
        return;
    }
    // Reservar memoria para los PIDs.
    pids = (pid_t *)malloc(sizeof(pid_t) * nordenes);
    if (pids == NULL) {
        perror("Error al reservar memoria para los PIDs.");
        free_ordenes_pipes(ordenes, NULL, 0);
        return;
    }
    // Crear las tuberías.
    pipes = crear_pipes(nordenes);

    // Bucle para ejecutar cada orden.
    for (int i = 0; i < nordenes; i++) {
        int entrada, salida;

        // Configurar entrada y salida según la posición de la orden.
        if (i == 0) {
            // Primera orden.
            entrada = STDIN_FILENO;
            if (nordenes > 1) {
                salida = pipes[0][1];  // Descriptor de escritura del primer
pipe.
            } else {
                salida = STDOUT_FILENO;  // Salida estándar.
            }
        } else if (i == nordenes - 1) {
            // Última orden.
            entrada = pipes[i - 1][0];
            salida = STDOUT_FILENO;
        } else {
            // Orden intermedia.
            entrada = pipes[i - 1][0];
            salida = pipes[i][1];
        }

        // Ejecutar la orden.
        pids[i] = ejecutar_orden(ordenes[i], entrada, salida, &backgr);
```

```c
        // Cerrar descriptores ya utilizados en el padre.
        if (i > 0) close(pipes[i - 1][0]);
        if (i < nordenes - 1) close(pipes[i][1]);
    }
    // Esperar a los hijos (si no están en segundo plano).
    if (backgr == 0 && pids[nordenes - 1] > 0) {
        for (int i = 0; i < nordenes; i++) {
            if (waitpid(pids[i], NULL, 0) == -1) {
                perror("Error al esperar a los hijos.");
            }
        }
    }

    // Liberar memoria.
    free(pids);
    free_ordenes_pipes(ordenes, pipes, nordenes);
}
```

Funcionalidad:

Trabajaremos con las siguientes variables: ordenes: Lista de cadenas de texto, cada una representando un comando extraído de la línea de órdenes. nordenes: Número total de comandos separados por |. pids: usado con pid_t para almacenar los identificadores de procesos creados. pipes: Matriz de descriptores de archivo para las tuberías que conectan los comandos. backgr: Indicador de si alguno de los comandos debe ejecutarse en segundo plano.

Primero, la función parser_pipes separa la línea de órdenes en comandos individuales con | como delimitador. Esta función nos devuelve una serie de cadenas (ordenes) y el número de órdenes. A continuación, reservamos memoria para los respectivos id de los procesos de cada comando. En caso de error, liberamos lo reservado hasta ese momento para correctas futuras ejecuciones. Acudimos a la función definida previamente crear_pipes, que permitirá comunicar los procesos. Posteriormente, definimos la entrada y salida de las diferentes tuberías, destacando 3 casos; inicial, final e intermedia, su diferencia reside primordialmente, en que en las tuberías intermedias su entrada y salida es otra tubería y no la salida o entrada estándar. Se ejecutan los respectivos comando s en procesos hijos, los fd ya utilizados se cierran en el proceso padre para optimizar los recursos. El proceso padre espera la finalización de los procesos hijos con waitpid a no ser que se ejecuten en segundo plano. Por último, se libera la memoria asignada a cada recurso.

Añadidos:

```c
void mostrar_bienvenida() {
    printf("Bienvenido a nuestra Minishell\n");
    printf("        _.---._\n");
    printf("    .'''.'/|\\`.'''.\n");
    printf("   :  .' / | \\ `.  :\n");
    printf("   '.'  /  |  \\  `.'\n");
    printf("    `. /   |   \\ .'\n");
    printf("      `-.__|__.-'\n");
    printf("\n");
}
```

Hemos agregado la función mostrar bienvenida a entrada_minishell.c, que muestra una concha en formato ASCII. Esta función se ejecutará la primera vez para dar la bienvenida al usuario a la MiniShell. Este cambio también supone la modificación de entrada_minishell.h.

```c
void guardar_historial(char *cad, char *ruta_binario) {

    // Obtenemos el directorio del binario
    char ruta_completa[1024];

    // Obtengo la ruta absoluta del binario
    realpath(ruta_binario, ruta_completa);

    // Obtenemos el directorio raíz del proyecto
    char *directorio_binario = dirname(dirname(ruta_completa));

    // Construimos la ruta al archivo de historial
    char ruta_historial[1064]; // Suficiente para directorio + nombre de
archivo
    snprintf(ruta_historial, sizeof(ruta_historial), "%s/historial.txt",
directorio_binario);

    // Abrimos el archivo en modo append
    FILE *historial_file = fopen(ruta_historial, "a");

    if (historial_file != NULL) {
        fprintf(historial_file, "%s\n",
                cad);           // Guardamos la línea en el archivo
        fclose(historial_file); // Cerramos el archivo
    } else {
        perror("Error al guardar el historial");
    }
}
```

Hemos agregado la función guardar_historial, que guarda en un documento de texto los comandos realizados por el usuario. Esto supone la modificación paralela de entrada_minishell.h. Además de añadir la sentencia guardar_historial a Minishell.c:

```
guardar_historial(token, ruta_binario);
```

Donde token son los diferentes comandos, divididos por el carácter; como se ha visto en anteriores fases del desarrollo del trabajo. Su funcionamiento es el siguiente:

Primero, determina el directorio raíz del proyecto donde se encuentra el binario. Luego, construye la ruta completa al archivo de historial, el cual se crea o abre en modo append para agregar nueva información al final. Finalmente, escribe la cadena de texto proporcionada en el archivo de historial y cierra el archivo. Si ocurre algún error durante el proceso, se muestra un mensaje de error.

Análisis del Trabajo:

Complicaciones:

El comprender exactamente y manejar las señales en procesos zombies y la combinación de tuberías fueron conceptualmente lo más complicado del trabajo. Técnicamente:

La implementación de *trim* fue una solución bastante útil aunque un poco compleja, ya que resultaba poco intuitivo que al concatenar órdenes fuera necesario eliminar los espacios para evitar errores. Lo más complicado fue entender cómo está estructurado el código. Por ejemplo, nos sorprendió que muchas funciones devuelvan los parámetros en ubicaciones de memoria específicas en lugar de hacerlo mediante un `return`. Además, el `return` se emplea principalmente para indicar si la operación se ha realizado con éxito o no. Algo muy diferente a lo que estamos acostumbrados a ver en el resto de asignaturas.

Igualmente, valoramos el trabajo muy positivamente por la satisfacción que te recompensa el entender el código, consolidar los conceptos vistos en clase y poder sentir que has realizado una adaptación bastante cercana a la consola de comandos.

Bibliografía:

https://stackoverflow.com/questions/122616/how-do-i-trim-leading-trailing-whitespace-in-a-standard-way

https://www.w3schools.com/