**GITHUB**

1. Actualiza tu repositorio local

Antes de hacer cualquier cambio, asegúrate de que tu repositorio local esté actualizado con los últimos cambios del repositorio remoto. Esto ayuda a evitar conflictos.

**git pull origin main**

****

Asumiendo que tu rama principal se llama main. Si se llama master (o cualquier otro nombre), reemplaza main por el nombre correspondiente.

2. Trabaja en una nueva rama

Para cada nueva característica, corrección de errores o tarea, crea una rama nueva. Esto mantiene tus cambios organizados y aislados del trabajo principal hasta que estén listos para ser fusionados.

**git checkout -b mi-nueva-rama**

****

Reemplaza mi-nueva-rama con un nombre descriptivo para tu rama.

3. Haz tus cambios

Realiza los cambios necesarios en tu código o en los archivos correspondientes.

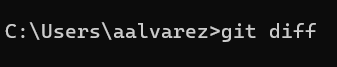
4. Revisa tus cambios

Antes de preparar tus cambios, es una buena práctica revisarlos. Esto te ayuda a asegurarte de que estás subiendo lo que realmente deseas.

**git status**

****

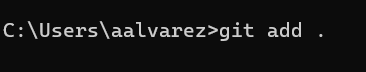
**git diff**

****

5. Prepara (stage) tus cambios

Una vez que hayas revisado tus cambios y estés satisfecho con ellos, añádelos al área de "staging" con git add.

**git add .**

****

O si prefieres añadir archivos específicos:

**git add mi\_archivo1 mi\_archivo2**

****

6. Haz un commit de tus cambios

Después de preparar tus cambios, haz un commit para guardarlos en tu historial de Git con un mensaje descriptivo de lo que has hecho.

**git commit -m "Descripción de los cambios realizados"**

****

7. Vuelve a actualizar tu rama

Antes de subir tus cambios, asegúrate nuevamente de que tu rama esté actualizada con la rama principal. Esto puede requerir un merge o rebase en tu rama local.

**git checkout main**

**git pull origin main**

**git checkout mi-nueva-rama**

**git merge main**

O, si prefieres rebase:

**git rebase main**

8. Resuelve conflictos, si los hay

Si hay conflictos entre tu rama y la rama principal, Git te pedirá que los resuelvas. Asegúrate de probar tus cambios después de resolver cualquier conflicto.

9. Haz push de tu rama

Una vez que tu rama esté actualizada y libre de conflictos, estás listo para subir tus cambios al repositorio remoto.

**git push origin mi-nueva-rama**

10. Crea una solicitud de pull/merge (PR/MR)

Desde la interfaz de GitHub, GitLab, Bitbucket u otra herramienta de alojamiento que estés utilizando, crea una solicitud de pull (o merge request) para tu rama. Asegúrate de describir los cambios y cualquier otra información relevante.

11. Revisión por pares

Espera a que al menos un compañero revise tu PR/MR y te dé retroalimentación. Sé abierto a las críticas constructivas y haz los ajustes necesarios basados en los comentarios.

12. Fusiona la PR/MR

Después de que tu PR/MR haya sido aprobada, puedes fusionar tus cambios a la rama principal. Dependiendo de la configuración de tu proyecto, quizás puedas hacerlo tú mismo o tal vez necesites que un mantenedor del proyecto lo haga.

Recuerda que la comunicación es clave. Mantén informados a tus compañeros de equipo sobre lo que estás haciendo y sé proactivo en la colaboración para mantener el flujo de trabajo lo más suave posible.

**Objetivo:**

El objetivo de Minishell es crear un shell sencillo, tu propio pequeño bash. Este proyecto te introduce a la gestión de procesos y file descriptors.

Instrucciones Generales:

* Lenguaje: El proyecto debe ser escrito en C, siguiendo la normativa de la escuela 42 sobre la codificación.
* Errores: No se permiten terminaciones inesperadas del programa, como segfaults. Cualquier uso indebido de memoria debe ser manejado adecuadamente sin memory leaks.
* Makefile: Debe incluirse un Makefile que gestione la compilación del proyecto con flags específicos y sin relinking. Se permiten bonuses bajo ciertas condiciones.

**Parte Obligatoria:**

Debes implementar funciones que permitan al shell:

* Mostrar una prompt y esperar por un nuevo comando.
* Ejecutar el ejecutable correspondiente, establecido en la variable PATH o usando rutas relativas o absolutas.
* Gestionar las señales ctrl-C, ctrl-D, y ctrl-\ de manera similar a Bash.
* Implementar built-ins como echo, cd, pwd, export, unset, env, y exit.
* Entre los detalles técnicos se incluye el manejo de pipes (|), redirecciones de entrada y salida (<, >, <<, >>), y la gestión de variables de entorno.

**Parte Extra:**

Como mejoras opcionales se sugieren:

* Implementar operadores lógicos (&&, ||) con manejo de prioridades mediante uso de paréntesis.
* Implementar wildcards (\*) para el directorio actual.

Los bonuses sólo se evalúan si la parte obligatoria está perfecta, sin errores y completamente funcional .

**En este proyecto que significaría tokenizar?**

En el contexto del proyecto Minishell, tokenizar significa dividir la cadena de entrada (es decir, el comando ingresado por el usuario en la terminal) en distintas partes (tokens) para su análisis y procesamiento. Este proceso es crucial para un shell, dado que necesita interpretar adecuadamente los comandos y sus argumentos para ejecutar las acciones solicitadas por el usuario.

Los tokens pueden incluir los nombres de los comandos, argumentos, operadores de redirección (>, <, >>, <<), conjuctores de comandos (||, &&), entre otros.

Aquí hay un ejemplo de cómo se podría tokenizar una cadena de entrada en un proyecto Minishell:

Entrada:

**ls -l | grep "minishell" > output.txt**

Tokens:

ls

-l

|

grep

"minishell"

>

output.txt

Tokenizar permite al shell:

Identificar y separar los comandos de sus argumentos y opciones.

Detectar y manejar adecuadamente las redirecciones de entrada y salida.

Procesar tuberías (|) que comunican la salida de un comando como entrada al siguiente.

Gestionar las características avanzadas como la expansión de variables, ejecución condicional (&&, ||), y agrupación de comandos.

Para implementar la tokenización, se puede utilizar una combinación de técnicas de análisis léxico, como máquinas de estados o funciones que identifiquen patrones en la cadena de entrada. Es importante manejar correctamente los casos especiales, como las comillas que agrupan espacios en blanco en un único argumento y las barras invertidas (\) que escapan caracteres especiales.

El resultado de la tokenización es una estructura de datos (por ejemplo, una lista de tokens) que representa de manera más manejable la secuencia de comandos y argumentos, lista para la etapa de análisis sintáctico y su posterior ejecución.

**Posible ROADMAP**

1. **Análisis Lexicográfico y Tokenización**
   1. Diseñar el analizador lexicográfico: Entiende cómo descomponer la cadena de comandos en tokens.
   2. Implementación: Crea funciones para identificar y separar comandos, argumentos, pipes y redirecciones.
2. **Análisis Sintáctico y Ejecución de Comandos**
   1. Parseo de comandos: Desarrolla una manera de interpretar la secuencia de tokens y construir una representación de los comandos a ejecutar.
   2. Ejecución básica de comandos: Implementa la capacidad de ejecutar comandos simples buscándolos en el PATH o mediante rutas.
3. **Gestión de Procesos y Tubería**.
   1. Manejo de pipes (|): Establece cómo comunicar la salida de un comando como entrada al siguiente.
   2. Implementar la ejecución de comandos en paralelo y el control de procesos.
4. **Implementación de Redirecciones.**
   1. Tratamiento de redirecciones: Codifica la funcionalidad para manejar >, <, >>, y <<.
5. **Implementar Built-ins.**
   1. Desarrollar comandos internos: Codifica funcionalidades para echo, cd, pwd, export, unset, env, y exit.
6. **Gestión de Señales**
   1. Implementar la gestión de señales: Asegura que ctrl-C, ctrl-D, y ctrl-\ sean manejados correctamente como en Bash.
7. **Pruebas y Depuración**
   1. Elaborar casos de prueba: Crea una suite de pruebas exhaustivas para cada característica.
   2. Depuración: Usa herramientas de depuración para solucionar problemas y asegurar un comportamiento estable.
8. **Mejoras y Características Bonus**
   1. Implementar características extra: Considera añadir operadores lógicos, mejoras en el parseo, y manejo de wildcards (\*) si el tiempo lo permite.
9. **Refinamiento y Entrega**
   1. Refactorización: Mejora la calidad del código y su organización.
   2. Documentación: Documenta el diseño e implementación de tu shell.
10. **Revisión de Normas y Entrega Final**
    1. Revisar cumplimiento de Norminette: Asegúrate de que todo el código cumple con las normas de codificación.
    2. Preparar la defensa: Organiza una explicación clara de las partes clave de tu proyecto para la evaluación final.

**Funciones permitidas:**

1. **chdir**
   1. **path:** Ruta hacia el nuevo directorio de trabajo.
   2. **Descripción:** Cambia el directorio de trabajo actual del proceso llamador al que se especifica en path. Devuelve 0 en caso de éxito y -1 en caso de error.
   3. **int** chdir(const **char** \*path);
2. **close**
   1. **fd:** Descriptor de archivo a cerrar.
   2. **Descripción:** Cierra el descriptor de archivo fd. Si tiene éxito, devuelve 0; si hay un error, devuelve -1.
   3. **int** close(**int** fd);
3. **dup, dup2**
   1. **oldfd:** Descriptor de archivo existente.
   2. **newfd:** (Solo dup2) Descriptor de archivo objetivo.
   3. **Descripción:** dup y dup2 duplican un descriptor de archivo. dup2 adicionalmente intenta copiarlo en newfd.
   4. **int** dup(**int** oldfd);
   5. **int** dup2(**int** oldfd, **int** newfd);
4. **execve**
   1. **pathname:** Ruta del archivo ejecutable.
   2. **argv:** Argumentos para el nuevo programa.
   3. **envp:** Variables de entorno.
   4. **Descripción:** Ejecuta el programa apuntado por pathname, pasando argv como argumentos y envp para el entorno.
   5. **int** execve(**const char** \*pathname, **char** \*const argv[], **char** \*const envp[]);
5. **exit**
   1. **status:** Código de salida.
   2. **Descripción:** Termina el proceso llamador inmediatamente.

* 1. **void** exit(**int** status);

1. **fork**
   1. **Descripción:** Crea un proceso hijo que es una copia del proceso padre.

* 1. **pid\_t** fork(**void**);

1. **free**
   1. **ptr:** Puntero a la memoria para liberar.
   2. **Descripción:** Libera el espacio de memoria previamente reservado por malloc, calloc, o realloc.
   3. **void** free(**void** \*ptr);
2. **getcwd**
   1. **Descripción:** Escribe el directorio de trabajo actual en buf.
   2. **buf:** Búfer donde se almacena el directorio actual.
   3. **size:** Tamaño del búfer.
   4. **char** \*getcwd(**char** \*buf, **size\_t** size);
3. **kill**
   1. **pid:** ID del proceso objetivo.
   2. **sig:** Señal a enviar.
   3. **Descripción:** Envía la señal sig al proceso especificado por pid.

* 1. **int** kill(**pid\_t** pid, **int** sig);

1. **malloc**
   1. **size:** Tamaño de la memoria a reservar.
   2. **Descripción:** Reserva un bloque de memoria del tamaño especificado y devuelve un puntero a la memoria reservada.

* 1. **void** \*malloc(**size\_t** size);

1. **open**
   1. **pathname:** Ruta al archivo.
   2. **flags:** Cómo abrir el archivo.
   3. **mode:** (Opcional) El modo de archivo cuando se crea.
   4. **Descripción:** Abre el archivo especificado y devuelve un descriptor de archivo.

* 1. **int** open(const **char** \*pathname, **int** flags, ... /\* mode\_t mode \*/ );

1. **opendir**
   1. **name:** Nombre del directorio.
   2. **Descripción:** Abre un flujo de directorio correspondiente al directorio name.

* 1. **DIR** \*opendir(**const** **char** \*name);

1. **pipe**
   1. **pipefd:** Un arreglo donde los dos descriptores de archivo se almacenan.
   2. **Descripción:** Crea una tubería, un canal unidireccional de datos que se puede usar para la comunicación entre procesos.

* 1. **int** pipe(**int** pipefd[2]);

1. **read**
   1. **fd:** Descriptor de archivo desde el cual leer.
   2. **buf:** Búfer donde almacenar los datos leídos.
   3. **count:** Número máximo de bytes a leer.
   4. **Descripción:** Lee hasta count bytes de fd en el búfer buf.

* 1. **ssize\_t** read(**int** fd, void \*buf, **size\_t** count);

1. **readdir**
   1. **dirp:** Puntero al flujo de directorio.
   2. **Descripción:** Lee la siguiente entrada en el directorio abierto.

* 1. **struct** dirent \*readdir(**DIR** \*dirp);

1. **signal**
   1. **signum:** Número de la señal.
   2. **handler:** Manejador de señal.
   3. **Descripción:** Establece un manejador para la señal signum.
   4. **typedef** **void** (\*sighandler\_t)(**int**);
   5. **sighandler\_t** signal(**int** signum, **sighandler\_t** handler);
2. **stat, fstat, lstat**
   1. **pathname , fd:** Ruta al archivo o descriptor de archivo.
   2. **statbuf:** Estructura donde se almacena el estado del archivo.
   3. **Descripción:** Obtiene información sobre el archivo. lstat es similar a stat, pero no sigue enlaces simbólicos.

* 1. **int** stat(**const** **char** \*pathname, **struct** stat \*statbuf);
  2. **int** fstat(**int** fd, **struct** stat \*statbuf);
  3. **int** lstat(**const** **char** \*pathname, **struct** stat \*statbuf);

1. **strerror**
   1. **errnum:** Número de error.
   2. **Descripción:** Devuelve una cadena que describe el error asociado con errnum.

* 1. **char** \*strerror(**int** errnum);

1. **wait**
   1. **status:** Donde almacenar el estado de salida.
   2. **Descripción:** Espera a que un proceso hijo cambie de estado.

* 1. **pid\_t** wait(**int** \*status);

1. **waitpid**
   1. **status:** Donde almacenar el estado de salida.
   2. **options:** Opciones para controlar la operación de waitpid.
   3. **Descripción:** Espera a que un proceso hijo específico cambie de estado, según pid.
   4. **pid:** PID del proceso hijo.

* 1. **pid\_t** waitpid(**pid\_t** pid, **int** \*status, **int** options);

1. **write**
   1. **fd:** Descriptor de archivo hacia el cual escribir.
   2. **buf:** Datos a escribir.
   3. **count:** Número de bytes a escribir.
   4. **Descripción:** Escribe count bytes desde buf hacia el descriptor de archivo fd.
   5. **ssize\_t** write(**int** fd, **const** **void** \*buf, **size\_t** count);

**Programa que queda a la espera y manejar ctrl+c ctrl+d ctrl+\**

**Ctrl+C**

**Señal Enviada:** SIGINT (Señal de INTerrupción).

**Comportamiento Predeterminado:** Termina el proceso.

**Descripción:** Ctrl+C se utiliza para interrumpir la ejecución de un programa desde la terminal. Envía la señal SIGINT al proceso activo en primer plano, lo que generalmente resulta en la terminación inmediata del programa. Los programas pueden capturar y manejar esta señal para realizar tareas específicas antes de terminar, como cerrar archivos o liberar recursos.

**Ctrl+D**

**Señal Enviada:** No envía una señal. Genera EOF (End Of File).

**Comportamiento Predeterminado:** Indica el fin de la entrada estándar (stdin).

**Descripción:** Ctrl+D se utiliza para indicar el final de la entrada cuando se toma de la entrada estándar (stdin). No termina el proceso, pero puede causar que el proceso termine si está esperando más entrada. Es como decir "ya no tengo más datos para proporcionarte". En contextos donde se lee entrada (por ejemplo, fgets o read en C), Ctrl+D hace que estas funciones devuelvan EOF (usualmente -1), lo que se puede usar para terminar bucles de lectura.

**Ctrl+\**

**Señal Enviada:** SIGQUIT.

**Comportamiento Predeterminado:** Termina el proceso y genera un archivo de volcado de memoria (core dump).

**Descripción:** Ctrl+\ se utiliza para enviar la señal SIGQUIT al proceso en primer plano. Al igual que SIGINT, puede ser capturado y manejado por el programa. Si no se maneja, el comportamiento predeterminado es terminar el proceso y crear un volcado de memoria (core dump). Esto es útil para fines de depuración porque el volcado de memoria puede ser analizado para entender qué estaba haciendo el proceso en el momento de la señal.

Estas combinaciones sirven como herramientas esenciales para interactuar con los procesos desde la terminal, permitiendo a los usuarios terminar programas, indicar el final de la entrada, y ayudar en la depuración de programas.

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <string.h>

void sigint\_handler(int sig) {

write(STDOUT\_FILENO, "\n", 1); // Simplemente empieza una nueva línea para simular la interrupción del comando actual.

}

void sigquit\_handler(int sig) {

// Para este ejemplo, no realizaremos ninguna acción para SIGQUIT, imitando el comportamiento de un terminal convencional.

}

int main() {

signal(SIGINT, sigint\_handler); // Establece sigint\_handler como el manejador de señales para SIGINT.

signal(SIGQUIT, SIG\_IGN); // Ignora la señal SIGQUIT completamente, sin terminar el shell.

char buffer[256];

while (1) {

printf("> ");

fflush(stdout); // Nos aseguramos de que el prompt se imprima antes de la entrada del usuario.

// fgets lee una línea de entrada de stdin hasta un máximo de sizeof(buffer)-1 caracteres o hasta que encuentre una nueva línea, lo que ocurra primero.

// Si fgets recibe EOF (que ocurre al presionar Ctrl+D), devuelve NULL. Esta condición se usa para salir del bucle y terminar el programa.

if (fgets(buffer, sizeof(buffer), stdin) == NULL) {

printf("Saliendo... (EOF detectado con Ctrl+D)\n");

break; // Sale del bucle y termina el programa si se detecta EOF.

}

// Aquí podrías agregar la lógica para procesar la entrada almacenada en buffer.

// Por ejemplo, podría simular o ejecutar un comando.

printf("Comando: %s", buffer);

}

return 0;

}

// Si te gustaría simular la ejecución de un comando, aquí es donde iría el código

printf("Comando: %s", buffer);

// Simular algún procesamiento aquí

}

return 0;

}



Este código hará lo siguiente en respuesta a las señales:

* **Ctrl+C (SIGINT):** Simplemente empieza una nueva línea y muestra el prompt nuevamente, imitando el comportamiento de ignorar la interrupción de la línea de comando actual sin salir del shell.
* **Ctrl+\ (SIGQUIT):** Se ignora completamente dentro del Minishell para este ejemplo, permitiendo que el shell continúe su ejecución normal.
* **Ctrl+D (EOF):** Detecta el EOF y cierra el Minishell, similar a cómo se comporta un terminal convencional al recibir EOF cuando no hay entrada pendiente.

¿Cómo debería ser el parseo?

1. **Lectura de la Entrada:**
   1. Comienza leyendo la entrada del usuario como una línea de texto completa. Ya has cubierto esta parte con fgets o una función similar.
2. **Trimming** (Eliminar Espacios en Blanco Iniciales y Finales):
   1. Antes del parseo, es útil eliminar espacios en blanco iniciales y finales para simplificar el análisis posterior.
   2. Puedes usar funciones como strtrim en librerías o implementar una sencilla tú mismo.
3. **División en Tokens:**
   1. Separa la línea de entrada en "tokens" basados en delimitadores, típicamente espacios en blanco (' '), pero también considera otros como tabulaciones ('\t') para comandos y argumentos.
   2. Usa funciones como strtok (o una variante segura como strtok\_r en POSIX) para dividir la entrada.
4. **Análisis de Citas y Caracteres Especiales:**
   1. Maneja adecuadamente las citas (comillas simples ' y dobles ") para permitir que los argumentos contengan espacios.
   2. Trata caracteres especiales como redireccionamientos (>, <), pipes (|), y otros que puedan alterar el flujo normal de ejecución de comandos.
   3. Considera la posibilidad de mantener un estado o un contexto mientras divides los tokens para manejar correctamente las subcadenas entre comillas.
5. **Construcción de Estructuras de Datos para Almacenar Comandos:**
   1. A medida que analizas la línea, organiza los tokens en estructuras de datos que representen comandos y sus argumentos. Una estructura con un array (o lista) para los argumentos es un buen punto de partida.
   2. Para las operaciones avanzadas como pipes y redirecciones, puede ser útil tener una estructura más compleja que también represente estos flujos.
6. **Manejo de Pipes y Redireccionamientos:**
   1. Implementa una manera de detectar y manejar pipes (|) y redireccionamientos (>, <, >>) durante el proceso de parseo.
   2. Considera la posibilidad de construir una lista enlazada (o un arreglo dinámico) de comandos cuando encuentres un pipe, donde cada nodo representa un comando en la secuencia.
7. **Expansión de Variables de Entorno:**
   1. Busca y expande variables de entorno. Por ejemplo, en la cadena "echo $HOME", sustituye $HOME por el valor real de la variable de entorno HOME.
   2. Puedes usar la función getenv para ayudarte a obtener valores de variables de entorno.
8. **Evaluación de Comandos Incorporados (Built-in Commands):**
   1. Antes de intentar ejecutar un comando, verifica si se trata de un comando incorporado (como cd, exit, export, unset, echo).
   2. Ejecuta el comando correspondiente directamente desde tu shell sin invocar un nuevo proceso si es un comando incorporado.
9. **Lógica de Ejecución:**
   1. Una vez realizado el parseo y la estructura del comando preparada, proceder con la lógica para ejecutar comandos externos usando fork, exec, y manejar pipes/redireccionamientos según sea necesario.
10. **Limpiar:**
    1. Asegúrate de liberar cualquier memoria dinámica reservada durante el parseo para evitar leaks.

**¿Qué debería saber a nivel teórico para hacer y defender el proyecto?**

1. **Conceptos de Sistemas Operativos:**
   1. Procesos y Ciclo de Vida de un Proceso: Entender cómo se crean, ejecutan y terminan los procesos, incluyendo los estados de un proceso.
   2. Gestión de Procesos: Conocimientos sobre fork(), exec(), y wait() son cruciales, ya que son la base para ejecutar comandos y controlar procesos hijos.
   3. Señales y Manejo de Señales: Comprender cómo funcionan las señales, cómo se envían (por ejemplo, kill), y cómo se manejan dentro de programas (signal, sigaction).
2. **Entrada/Salida y Redirección:**
   1. Entender cómo funcionan las redirecciones y los pipes, su propósito y cómo implementarlos (dup, dup2, pipe).
   2. Conocimiento sobre los descriptores de archivo estándar (stdin, stdout, stderr) y cómo gestionarlos.
3. **Manipulación de Texto y Cadenas en C:**
   1. Funciones de biblioteca para manejar cadenas (strcpy, strcat, strtok, strchr, etc.), y manejo de memoria (malloc, free).
   2. Security considerations in string manipulation (buffer overflows, using safer function variants like snprintf).
4. **Variables de Entorno:**
   1. Cómo las variables de entorno influyen en el comportamiento de los programas y cómo se gestionan (usando getenv, setenv, unsetenv).
5. **Interfaz de Línea de Comandos (CLI) y Shells:**
   1. Entender la estructura y el propósito de un shell. Conocimiento sobre cómo el usuario interactúa con el sistema a través del shell.
   2. Construcción y manejo de un intérprete de comandos (shell), incluyendo el parseo de comandos y argumentos.
6. **Programación Defensiva:**
   1. Técnicas para escribir código seguro y robusto que maneje correctamente los errores e invalidez de datos.
   2. Gestión de errores y recursos (memoria, descriptores de archivo).
7. **Depuración y Herramientas:**
   1. Uso de depuradores (gdb, valgrind) para identificar y resolver errores y fugas de memoria.