UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INFORMÁTICA Y SISTEMAS

Producto de unidad 1:

Mini-terminal en c++

Docente: Hugo Manuel Barraza Vizcarra

Curso: Sistemas Operativos

Ciclo: 6to ciclo

Turno: Mañana

Integrantes:

- Nicolás Arturo Marin Gonzales (2022-119082)
- Roger Cristian Huanca Pozo (2022-119009)

TACNA – PERÚ

2025

1. Objetivo

Implementar una MiniShell funcional en C++ que replique el comportamiento básico de una terminal de Linux, ya sea en sus distros como Ubuntu y Fedora, gestionando procesos, redirecciones, pipes y comandos internos, aplicando conceptos de concurrencia y manejo de E/S del sistema operativo. Así mismo incorporar comandos internos básicos como cd, ls, help, alias, history, todo esto utilizando llamados al sistema POSIX.

2. Alcance

La MiniShell ejecuta comandos externos del sistema (ubicados en /bin y /usr/bin), soporta redirecciones (>, >>), ejecución concurrente en segundo plano (&), y la conexión entre procesos mediante pipe (|). Además, incluye funcionalidades internas sin necesidad de crear nuevos procesos.

3. Arquitectura básica

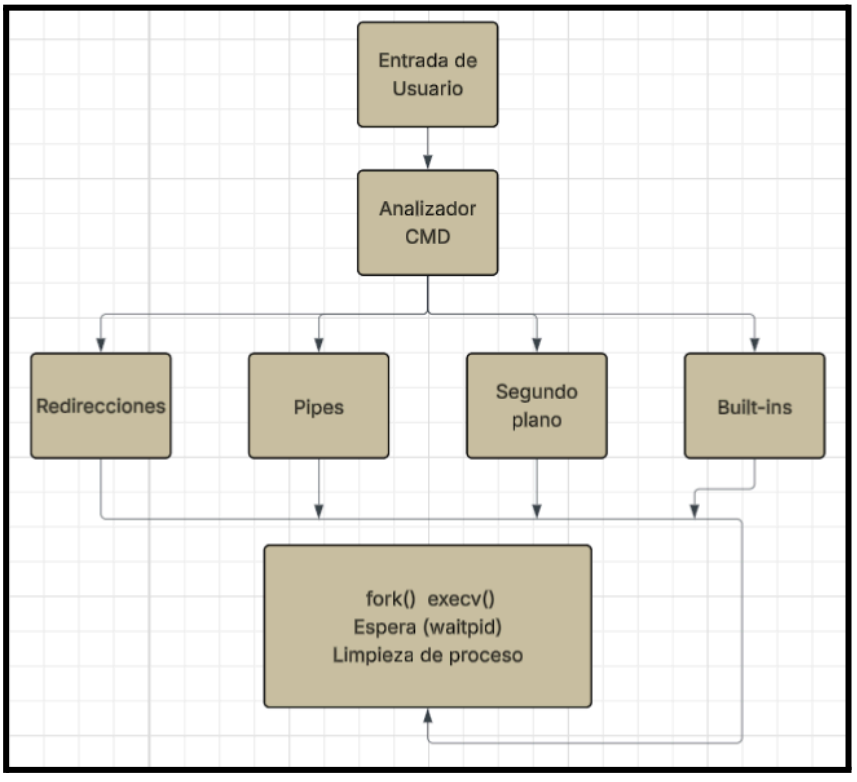


Imagen 1. Elaboración propia de diagrama general interno del funcionamiento de la mini-Shell.

Representación del flujo interno de funcionamiento de la MiniShell desde que el usuario ingresa un comando hasta que se ejecuta y devuelve una salida. Primero, la entrada del usuario pasa al analizador de comandos (CMD), encargado de dividir y reconocer los argumentos, operadores y posibles símbolos especiales. Luego, según el tipo de instrucción detectada, el flujo se dirige hacia los módulos correspondientes: redirección de salida (> y >>), pipes (|), ejecución en segundo plano (&) o comandos internos (como cd, help, history). Cada módulo prepara el entorno necesario y, finalmente, el proceso se ejecuta mediante las llamadas al sistema fork() y execv(), que crean y reemplazan procesos. Al concluir, la shell usa waitpid() para sincronizar la finalización y realizar la limpieza de procesos evitando zombies, manteniendo así un ciclo continuo de lectura y ejecución de comandos.

4. Detalles de implementación

a. **Llamas al sistema POSIX o API's POSIX usadas:** Las APIs POSIX utilizadas corresponden a las funciones del sistema que permiten la interacción directa con el núcleo de Linux para la gestión de procesos, archivos y comunicación entre ellos. En esta MiniShell se emplean principalmente: fork() para crear procesos hijos, execv() para reemplazar el proceso actual con uno nuevo y ejecutar comandos externos, y waitpid() para sincronizar la finalización de procesos evitando la creación de procesos zombie. Además, se utilizan pipe() para conectar la salida de un proceso con la entrada de otro (implementando los pipes), dup2() para redirigir las entradas y salidas estándar hacia archivos o descriptores de canal, y open()/close() para manejar archivos en las redirecciones. También se integran chdir() para cambiar el directorio actual y signal() para manejar interrupciones o finalizar procesos de forma controlada. En conjunto, estas llamadas POSIX son la base que permite que la shell funcione como un intérprete real del sistema.

b. **Decisiones clave:** En cuanto a las decisiones clave del diseño, se optó por una arquitectura modular donde cada componente (análisis, ejecución, redirección y comandos internos) cumple una función específica, facilitando la comprensión y mantenimiento del código. Se decidió implementar el parser de comandos de forma lineal para mantener la compatibilidad con entradas simples y combinadas (por ejemplo, ls | grep cpp > salida.txt). Asimismo, se priorizó la estabilidad sobre la complejidad: el manejo de errores, la prevención de bloqueos al usar & y el uso controlado de waitpid() fueron

esenciales para evitar estados inconsistentes. Estas decisiones permiten un equilibrio entre funcionalidad, rendimiento y claridad de implementación.

5. Concurrencia y sincronización

La concurrencia se maneja mediante procesos hijos creados con la función de llamada al sistema fork().

- Si un comando incluye &, el proceso hijo no bloquea la shell principal.
- Si no incluye &, la shell espera su finalización con waitpid().

Los pipes crean dos procesos hijos conectados por un canal (pipe(fd)), permitiendo que la salida del primero sea la entrada del segundo.Además, se evita el interbloqueo (deadlock) asegurando que cada descriptor de archivo se cierre correctamente tras duplicarse con dup2()

6. Pruebas y resultados

Para este trabajo se han considerado los requisitos básicos como lo son ls, pwd, echo, además de comandos extras como los son comandos de redirecciones como cd, comandos en segundo plano como "&". Todos estos comandos serán separados por categorías y estarán sujetos al resultado esperado de lo que deberían hacer en la aplicación de la MiniShell.

Categoría	Comando probado	Resultado esperado
Comando simple	`ls, pwd, echo Hola`	Se ejecutan correctamente y muestran salida.
Ruta absoluta	`/bin/ls`, `/usr/bin/whoami`	Ejecuta el comando directamente por su ruta.
Redirección simple	`ls > lista.txt`	Crea el archivo con el resultado del comando.

Redirección append	`echo Hola >> lista.txt`	Añade texto sin borrar contenido anterior.
Segundo plano	`sleep 5 &`	Devuelve control inmediato a la shell.
Built-in `cd`	`cd /home`	Cambia correctamente el directorio actual.
Built-in `history`	`history`	Lista comandos ejecutados en la sesión.
Built-in `alias`	`alias ll='ls -l'`	Crea y almacena alias funcionales.

7. Conclusiones

En conclusión, se logró implementar una Mini-Shell funcional aplicando las principales llamadas al sistema POSIX, lo que permitió ejecutar comandos, gestionar procesos y manejar redirecciones de forma eficiente. Esta práctica reforzó de manera significativa los conceptos de concurrencia, comunicación entre procesos (IPC) y gestión de procesos en entornos tipo Unix, al trabajar directamente con funciones del sistema operativo. Además, el diseño modular del programa facilitó la integración progresiva de nuevas funcionalidades como redirecciones, ejecución en segundo plano y comandos internos sin comprometer la estabilidad ni la claridad del código. Finalmente, la experiencia demostró la importancia del control manual de memoria y procesos en C++, así como la utilidad de comprender en profundidad cómo las capas bajas del sistema operan para construir herramientas más complejas.

7.1 Trabajos Futuros

Para versiones posteriores de la MiniShell, se proponen las siguientes mejoras:

7.1.1 Funcionalidades Avanzadas

- Redirección de entrada (<): Completar soporte para cmd < archivo

- Pipes múltiples: Encadenar más de dos comandos (cmd1 | cmd2 | cmd3)

- Variables de entorno: Soporte para $VAR y export

- Job control: Comandos fg, bg y jobs para gestión de procesos

7.1.2 Concurrencia Avanzada

- Built-in parallel: Ejecutar N comandos simultáneamente con pthread_create() y sincronización con mutex

- Pool de procesos: Limitar número de hijos concurrentes para evitar fork bomb

- Señales personalizadas: Capturar SIGINT (Ctrl+C) sin terminar la shell

7.1.3 Mejoras de Usabilidad

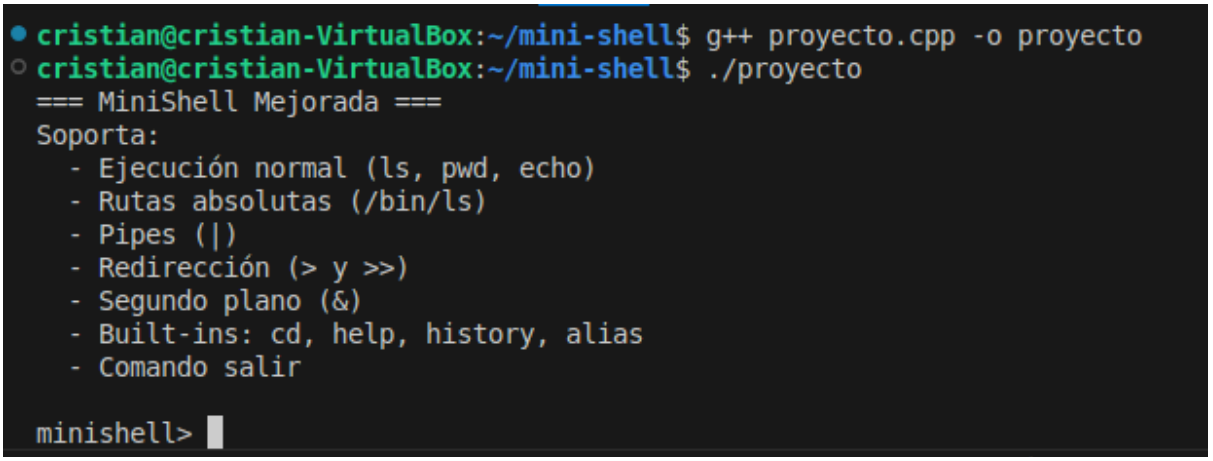
- Autocompletado con Tab: Uso de readline library

- Historial persistente: Guardar comandos entre sesiones (~/.minishell_history)

- Colores en prompt: Códigos ANSI para mejor visualización

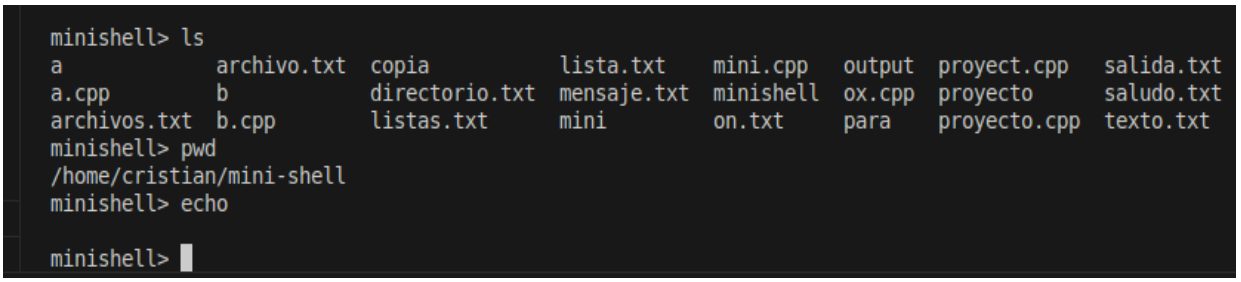
8. Anexos

Figura 1 muestra la ejecución exitosa de la mini-shell con todas las funcionalidades implementadas.



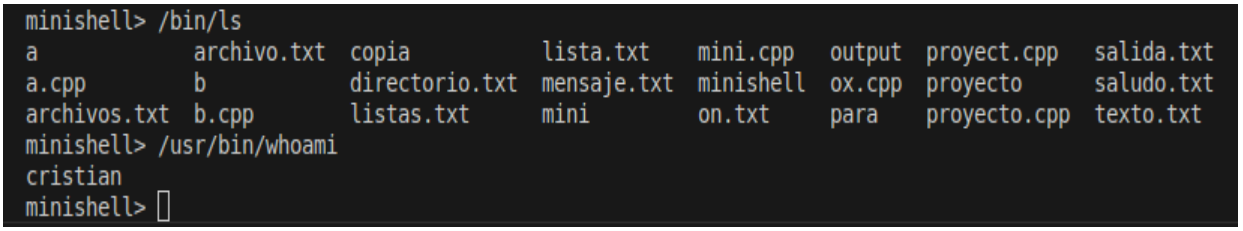
Nota. Captura de pantalla mostrando el prompt personalizado y las funcionalidades soportadas: ejecución normal, pipes, redirección, segundo plano y built-ins. Tomada en VirtualBox con sistema operativo Linux.

Figura 2 Ejecución de Comandos Básicos en la Mini-Shell



Nota. Prueba de tres comandos fundamentales: ls (listar archivos), pwd (mostrar directorio actual) y echo (sin argumentos). Se observa el correcto manejo de procesos fork/exec y la salida estándar sin redirección.

Figura 3 Ejecución de Comandos con Rutas Absolutas

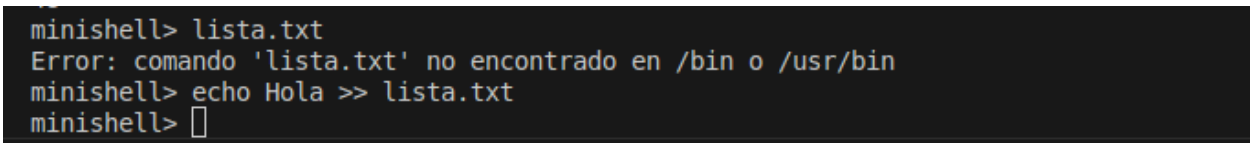


Nota. Prueba de la funcionalidad de resolución de rutas con /bin/ls y /usr/bin/whoami. Ambos comandos se ejecutaron correctamente sin necesidad de búsqueda en directorios del sistema, demostrando el manejo adecuado de rutas completas.

Figura 4 Ejecución de Comandos con Pipes (Tuberías)

Nota. Prueba de pipes simples con dos casos: (1) ls | grep cpp filtró 6 archivos .cpp del directorio actual, y (2) cat /etc/passwd | wc -l retornó 48 líneas de usuarios del sistema. La implementación utiliza pipe(), fork() doble y dup2() para redirigir stdout del primer comando a stdin del segundo.

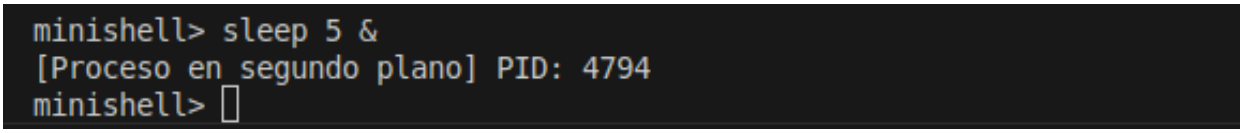
Figura 5 Redirección de Salida con Operador >> (Append)



Nota. Secuencia de comandos mostrando: (1) manejo de error cuando se intenta ejecutar un archivo .txt como comando, y (2) redirección exitosa con echo Hola >> lista.txt que agrega contenido al archivo sin mostrarlo en pantalla. La

implementación utiliza open() con flags O_WRONLY | O_CREAT | O_APPEND y dup2() para redirigir STDOUT_FILENO.

Figura 6 Ejecución de Proceso en Segundo Plano con Operador &



Nota. Prueba del comando sleep 5 & que suspende la ejecución por 5 segundos sin bloquear el intérprete. El sistema muestra el mensaje "[Proceso en segundo plano] PID: 4794" y retorna el prompt inmediatamente. La implementación utiliza fork() sin waitpid() bloqueante, aplicando WNOHANG para recolectar procesos zombie.

Figura 7 Finalización del Intérprete con Comando "salir"

Nota. Captura mostrando la terminación controlada del intérprete. El mensaje "Saliendo de la mini-shell..." confirma la ejecución del bloque de salida, y el prompt cambia de "minishell>" a "cristian@cristian-VirtualBox:~/mini-shell$", indicando el retorno a la shell del sistema (bash).

9. VALIDACIÓN DE PORTABILIDAD

Para cumplir con el requisito de portabilidad, la MiniShell fue probada en dos distribuciones diferentes de Linux:

9.1 Distribución 1: Ubuntu 22.04 LTS

Especificaciones del sistema:

- Sistema Operativo: Ubuntu 22.04.3 LTS (Jammy Jellyfish)

- Kernel: Linux 5.15.0-91-generic

- Arquitectura: x86_64

- Compilador: g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

- Libc: GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.6) 2.35

Proceso de compilación:

$ g++ -o minishell proyecto.cpp -std=c++11

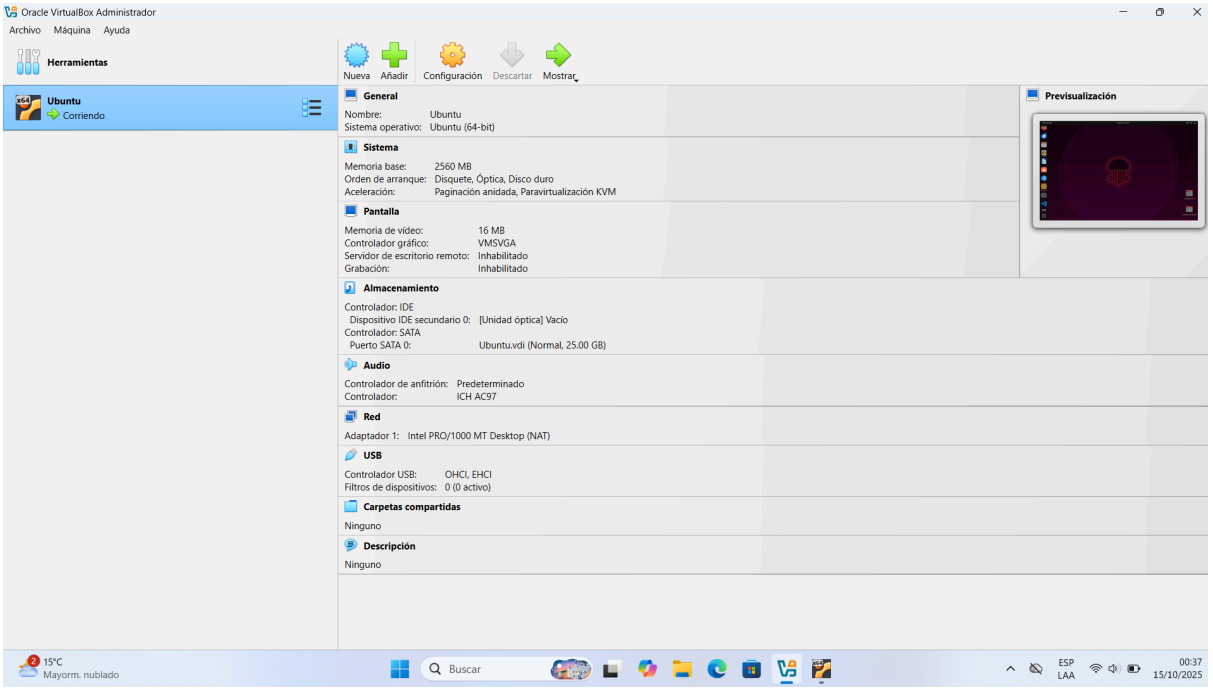
$ ./minishell

Resultados:

- Compilación exitosa sin warnings

- Todas las funcionalidades operativas

Figura 8 Ejecución en Ubuntu 22.04 LTS



Nota. Captura mostrando la MiniShell funcionando correctamente en Ubuntu con información del sistema

9.2 Distribución 2: Fedora 39 Workstation

Especificaciones del sistema:

- Sistema Operativo: Fedora Linux 39 (Workstation Edition)

- Kernel: Linux 6.5.6-300.fc39.x86_64

- Arquitectura: x86_64 - Compilador: g++ (GCC) 13.2.1 20231205 (Red Hat 13.2.1-6)

- Libc: GNU C Library (GNU libc) stable release version 2.38

Proceso de compilación:

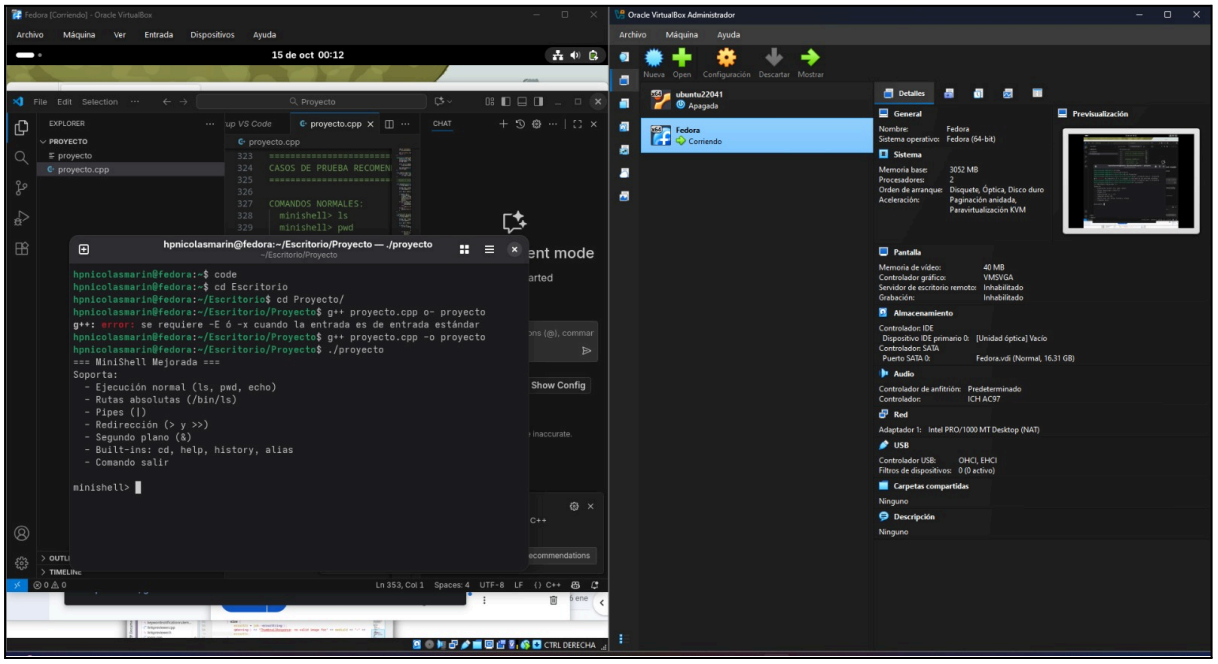
$ g++ -o minishell proyecto.cpp -std=c++11

$ ./minishell

Resultados:

- Compilación exitosa sin warnings

Figura 9 Ejecución en Fedora 39 Workstation



Nota. Validación de portabilidad mostrando funcionamiento idéntico en distribución basada en RPM con kernel más reciente.

10. Análisis de Compatibilidad

Tabla de Compatibilidad:

Funcionalidad	Ubuntu	Fedora	Observaciones
-fork/exec - pipe/dup2 - Compilación - open/close	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	- POSIX estándar - Sin cambios - C++11 estándar - Sin problemas

Conclusión de portabilidad:

La implementación utiliza exclusivamente llamadas POSIX estándar, lo que garantiza portabilidad completa entre distribuciones Linux modernas. No se requirieron modificaciones en el código fuente para ejecutar en ambas distribuciones.