Para afrontar este trabajo práctico tomamos un enfoque de divide y vencerás, cada uno encargándose de completar un requisito específico. Dentro de la shell se han conseguido cumplir las funcionalidades solicitadas, aunque sí se han tenido unos pequeños problemas al momento de ejecutar y hacer que funcione dentro del entorno de clase de Rocky Linux.

**Paso 1:** El Bucle Principal (Main REPL)
Gracias a la ayuda de Google Gemini comenzó el desarrollo de la shell y acompañó todo el proceso gracias a las herramientas de desarrollo. Preguntamos por un piso de donde comenzar a construir, un main del cual se iba a comenzar la ejecución de la shell, pasándole todo el contexto necesario para tener una base sólida. Así también preguntamos por darle un enfoque minimalista al código proporcionado, es a partir de este punto que comienza el desarrollo en serio, fundando las bases de la shell.

**Paso 2:** Desarrollo e implementación de comandos.
El contexto que le proporcionamos a la inteligencia artificial cooperó a avanzar rápidamente, fuimos construyendo poco a poco el ecosistema completo para la shell. Primero fuimos consultando por las funciones básicas necesarias para la shell, manteniendo el flow de trabajo que sugería el asistente para evitar una “sobre-estimulación” de contextos al asistente.

Análisis del código y la explicación proporcionada por el asistente

- **Ls:** usamos opendir y readdir, así evitamos system(), que leen directamente la tabla de inodos del directorio. Esto es más eficiente y nos da control total sobre qué mostrar (por ejemplo, filtramos los archivos ocultos que empiezan con punto para mantener la salida limpia).
- **Cd:** Aquí tuvimos una serie de complicaciones dentro del desarrollo, ya que este se realizaba en un entorno de windows (luego solucionado por medio de instancias vm ssh) donde no funcionaba setenv, así que en su lugar usamos putenv, disponible tanto en windows como en linux.
- **Mkdir:** Implementamos mkdir usando directamente la syscall del sistema operativo en lugar del comando externo, lo que nos da más control y cumple con las reglas del TP. Para mantener compatibilidad entre Windows y Linux usamos #ifdef _WIN32: en Windows mkdir() recibe solo la ruta, mientras que en Linux requiere ruta y permisos (0755). El kernel se encarga de crear el nuevo inodo y validar permisos; si algo falla, mostramos el error con perror(). En caso de éxito, como en Unix, no imprimimos nada.
- **Rm:** Implementamos rm usando la syscall unlink, que en Linux elimina el enlace al inodo en lugar de “borrar” el archivo directamente. Esto evita usar el comando externo y cumple con el TP. La función reporta errores con perror() (archivo inexistente, permisos, intentar borrar un directorio) y, si funciona, no imprime nada. La única complicación fue la compatibilidad: en Windows unlink no siempre está disponible y puede requerir remove(), pero en Linux funciona sin problemas.
- **Cp:** Lo implementamos leyendo el archivo origen y escribiéndolo al destino con open, read y write, usando un buffer de 1KB para copiar por partes y no cargar todo en memoria. Hubo pequeñas complicaciones con la compatibilidad (O_BINARY en Windows) y con asegurar que write escriba exactamente lo que read leyó. Si falla algo lo mostramos con perror(), y si todo funciona, no imprimimos nada.
- **Cat:** Implementamos cat igual que cp pero escribiendo el contenido directamente en pantalla usando write(STDOUT_FILENO, ...), ya que en UNIX la salida estándar es

simplemente otro archivo (descriptor 1). Abrimos el archivo con open, leemos por partes con un buffer de 1KB y mostramos esos datos sin pasar por printf. Las complicaciones principales fueron la compatibilidad con O_BINARY en Windows y asegurar que write escriba exactamente lo leído. Si algo falla (archivo inexistente o permisos), mostramos perror(); si funciona, imprimimos el contenido sin mensajes extra.

- **Echo:** Lo implementamos reconstruyendo la frase que strtok separó, imprimiendo cada argumento desde args[1] y agregando espacios entre ellos, finalizando con un salto de línea. Es un comando simple pero requiere volver a unir las palabras manualmente para que la salida sea correcta. No hubo mayores complicaciones salvo manejar correctamente múltiples argumentos y evitar un espacio extra al final.
- **Redirección (> ):** La redirección funciona reemplazando temporalmente el descriptor 1 (STDOUT) con un archivo mediante dup2, de modo que todo lo que normalmente iría a la pantalla se escriba en él. Esto implica: detectar >, abrir el archivo con O_TRUNC (lo correcto para sobrescribir), guardar STDOUT, redirigirlo, ejecutar el comando y luego restaurarlo. El único detalle delicado fue el buffering: printf no escribe inmediatamente, así que sin fflush(stdout) los datos pueden terminar apareciendo en pantalla al restaurar STDOUT. Una vez añadido fflush antes de revertir la redirección, el comportamiento es correcto: no imprime nada en pantalla y sobrescribe el archivo tal como debe hacerlo >. Durante el desarrollo de la dirección nos encontramos con un problema, las librerías presentes en linux no funcionaban dentro del entorno Windows en el cual estábamos realizando el desarrollo, para solucionar esto nos conectamos por SSH a la máquina virtual utilizada en clase, manteniendo el entorno cómodo del desarrollo en Windows, pero con las funcionalidades de linux. A partir de este momento el desarrollo se centró solo en linux, evitando todo tipo de herramienta de desarrollo originaria de windows.

**Paso 3: Ejecución de Programas Externos**

Una vez completados todos los comandos internos, avanzamos hacia uno de los requisitos más importantes del trabajo práctico: permitir que la shell ejecute programas externos. Para esto implementamos el mecanismo clásico de UNIX basado en **fork() + execvp() + waitpid()**. El proceso padre crea un hijo con fork; dentro del hijo reemplazamos completamente la imagen del proceso usando execvp, que busca el binario en el PATH del sistema (por ejemplo, nano, vim, whoami, python). Si exec fallece mostramos perror, lo que permite detectar rutas inexistentes o permisos insuficientes. El padre, por su parte, usa waitpid para esperar la finalización del proceso hijo, manteniendo el control del ciclo REPL. Este enfoque reproduce el comportamiento estándar de las shells reales y permitió probar con éxito comandos externos complejos dentro del entorno Rocky Linux. La única dificultad consistió en manejar correctamente los casos donde el comando no existe, devolviendo errores claros sin colgar la shell y sin romper el bucle principal.

**Paso 4: Sistema de Logs y Auditoría**

Otro punto solicitado en el TP era llevar un registro persistente de la actividad del usuario dentro de la shell. Para cumplirlo, implementamos un sistema de logging **totalmente propio y portable**, evitando depender de permisos de administrador o de rutas restringidas como

/var/log. El enfoque elegido fue crear una carpeta oculta en el home del usuario, llamada ~/.flsh_logs, almacenando allí un archivo de texto con un log por cada sesión. Dentro del log registramos: comandos ejecutados, errores reportados por la shell, fecha y hora, y un mensaje especial cuando el usuario ejecuta exit. A diferencia de una shell profesional, donde el logging puede redirigirse a syslog o journald, nosotros lo implementamos directamente usando open, write y append, lo cual evita dependencias externas y funciona incluso en entornos muy limitados. Una consideración importante fue garantizar que el archivo se abra siempre en modo append y que el log se escriba incluso en caso de comandos fallidos, lo que permite auditar la sesión completa. Este sistema permitió cumplir con los requisitos del TP y, además, era especialmente útil para depuración durante el desarrollo remoto por SSH.

**Paso 5: Integración Final, Pruebas y Entorno de Desarrollo**

Tras completar la redirección, los comandos internos, la ejecución de programas externos y el sistema de auditoría, realizamos pruebas exhaustivas dentro del entorno Rocky Linux utilizado en clase. Aquí surgió una dificultad importante: muchas librerías estándar que funcionan en Linux no estaban disponibles dentro del entorno de desarrollo Windows en el que programábamos inicialmente. Esto complicaba la compilación de funciones clave (open, dup2, execvp, opendir), por lo que decidimos mantener el editor y herramientas de Windows, pero compilar y ejecutar exclusivamente dentro de la máquina virtual usando SSH. Con esta estrategia híbrida logramos conservar la comodidad del entorno local sin perder compatibilidad con Linux. A partir de ese punto, todo el desarrollo se centró definitivamente en Linux, y evitamos cualquier función o dependencia propia de Windows. Finalmente, validamos cada funcionalidad del TP: comandos internos, redirección, ejecución externa, logs, manejo de errores y robustez del ciclo REPL.

**Paso 6: Correcciones Finales y Consolidación del Sistema de Logs**

Durante la etapa final del desarrollo realizamos una revisión completa de la shell para asegurar que cumpliera con todos los requisitos del TP. Si bien la estructura base estaba correctamente implementada —bucle principal, comandos internos, redirección y ejecución de programas externos— detectamos inconsistencias en el sistema de logging. El código generaba archivos pero no garantizaba la creación automática del directorio correspondiente, la ruta dependía de manipular manualmente el path del ejecutable y no existía un mecanismo uniforme para registrar éxitos y errores de cada comando. Para resolverlo reescribimos completamente el sistema: ahora la shell obtiene dinámicamente su ruta mediante /proc/self/exe, crea la carpeta de logs sin necesidad de permisos especiales y unificamos todo el registro mediante la función log_shell(), que almacena fecha, usuario, comando ejecutado y resultado, separando los eventos normales en shell.log y los errores en sistema_error.log. Esta corrección estabilizó el comportamiento de la shell y aportó un nivel profesional de trazabilidad, facilitando tanto la auditoría como la depuración en el entorno Rocky Linux.

**Paso 7: Implementación del comando grep**

Finalmente, agregamos uno de los últimos requisitos del TP: la implementación del comando interno grep. Para mantener la coherencia con el resto del proyecto evitamos

invocar el binario externo y desarrollamos una versión propia basada en búsquedas en memoria usando fopen, fgets y strstr. El comando recorre el archivo línea por línea e imprime únicamente aquellas que contienen el patrón solicitado por el usuario. Aunque su comportamiento es minimalista, cumple estrictamente con lo pedido en el enunciado. La función se integró al sistema de logs, registrando en shell.log cuando encuentra coincidencias y en sistema_error.log cuando no existe el archivo, hay errores de lectura o no se encontraron coincidencias. La implementación no presentó mayores dificultades, y su estructura encaja naturalmente dentro del ecosistema ya construido para la shell.

**Paso 8: Introducción de la modalidad Sandbox (Jail del sistema)**

Como extensión opcional para aumentar la robustez del proyecto, incorporamos un mecanismo de **Jail/Sandbox** que restringe al usuario a su directorio personal, evitando que pueda navegar o modificar archivos fuera de ese entorno. Si bien no era un requisito obligatorio del TP, surgió durante la defensa técnica como una posible mejora, por lo que decidimos implementarlo. El sistema verifica que todas las rutas —incluyendo cd, cp, rm, mkdir, cat, grep e incluso la redirección “>”— permanezcan dentro del directorio permitido. Para ello se normalizan las rutas mediante realpath y luego se compara su prefijo con el home del usuario. El resultado es una protección sólida contra accesos indebidos, escapes mediante “..”, rutas absolutas e incluso redirecciones maliciosas. Agregar este módulo requirió modificar cada comando interno y también los programas externos que trabajan con rutas, pero una vez integrado el sistema demostró ser funcional y consistente, reforzando la seguridad general de la shell y simulando el comportamiento de entornos educativos o de práctica, donde se busca evitar daños al sistema real.

**Paso 9: Implementación de Seguridad Interactiva y Prevención de Accidentes**

Para reforzar la robustez del sistema y mitigar el riesgo de operaciones destructivas accidentales, implementamos una barrera de seguridad lógica mediante la función confirmar_accion. A nivel técnico, esta rutina evita el uso de scanf por sus vulnerabilidades de memoria, optando en su lugar por fgets para realizar una lectura de stdin con límites estrictos de buffer. Diseñamos su lógica bajo el principio de "denegación por defecto" (fail-safe), donde cualquier entrada distinta a 's' o 'S' resulta en una cancelación inmediata, garantizando que el usuario deba consentir explícitamente antes de proceder. Esta funcionalidad se integró transversalmente en los comandos rm y cp (al detectar conflictos de sobrescritura), vinculando cada decisión al sistema de logs para registrar tanto la ejecución exitosa como la cancelación preventiva, asegurando así la trazabilidad completa de las acciones críticas.

**Paso 10: Validación final, pruebas cruzadas y consolidación del proyecto**

Antes de dar por terminada la shell, realizamos una fase intensiva de pruebas combinadas, ejecutando cada comando interno con distintos tipos de entradas: rutas válidas, rutas inválidas, archivos inexistentes, redirecciones, programas externos y condiciones límite como archivos vacíos o directorios sin permisos. También sometimos el sistema a “pruebas de estrés” intencionalmente maliciosas dentro del sandbox para verificar que no existieran escapes posibles. Esta validación incluyó pruebas reales dentro de Rocky Linux —tanto en la VM del aula como en instancias creadas manualmente— para asegurar un

comportamiento consistente en todos los entornos. Los logs demostraron ser una herramienta clave durante esta etapa, al permitir identificar fácilmente desviaciones, errores o comportamientos inesperados del usuario o de la shell. Tras corregir los últimos detalles menores (principalmente relacionados con permisos, errores de lectura y pequeñas inconsistencias en las rutas), el proyecto quedó consolidado y listo para la entrega, cumpliendo satisfactoriamente con todos los requisitos solicitados en el trabajo práctico y extendiéndolos con mejoras de seguridad y robustez.

**Detalles del Parser de Comandos**

El parser fue uno de los componentes clave, porque todo lo que la shell ejecuta depende de interpretar correctamente la entrada. El comportamiento final quedó así:

- Se divide la línea ingresada usando strtok() como separador.
- Se normalizan espacios múltiples o finales para evitar argumentos vacíos.
- Se detecta el operador de redirección > buscando el token en la lista.
- Para comandos como grep, se detecta si el segundo parámetro es un patrón o parte del texto.
- Se arma el arreglo args[] listo para usar en comandos internos o externos.
- La shell diferencia comandos internos comparando args[0] con una lista de funciones implementadas.
- Si no coincide con ninguno, pasa automáticamente a ejecución externa.

No se implementaron comillas dobles, comillas simples ni escapes especiales para mantener la simplicidad del TP.

**Manejo de Errores y Robustez**

Para evitar caídas inesperadas, toda la shell utiliza un sistema centralizado de manejo de errores. Cada comando sigue el mismo patrón:

1. Se realiza la syscall correspondiente (open, mkdir, unlink, etc.)
2. Si hay error, se llama a perror(), lo que garantiza mensajes claros con la descripción exacta del fallo.
3. Los errores también se registran en el archivo .flsh_logs/error.log.

Este sistema permitió detectar rápidamente permisos incorrectos, rutas inválidas o intentos de usar comandos externos inexistentes. Además, garantizó que la shell **nunca se cierre de forma abrupta**, incluso ante errores graves.

**Gestión de Memoria**

Aunque la shell es relativamente pequeña, cuidamos especialmente la memoria. Decisiones importantes:

- Cada línea leída por el usuario se almacena en un buffer estático seguro.
- Se evita el uso excesivo de malloc(), y cuando se utiliza, se libera inmediatamente después.
- Todo buffer temporal es de tamaño fijo (1 KB) para prevenir sobrelecturas.
- No cargamos archivos completos en memoria, sino que los procesamos por partes (lectura/escritura en bloques de 1 KB).
- No se detectaron memory leaks.

Esto hizo que la shell sea estable incluso copiando o leyendo archivos grandes.

**Requisitos del Sistema**

La shell fue diseñada y probada principalmente en:

- **Rocky Linux 10**
- **Linux From Scratch 12.x**
- **Entorno de la clase (rocky-host-lfs por SSH)**

Requisitos mínimos:

- Kernel Linux con syscalls estándar POSIX
- Librerías glibc
- Acceso a /home para la sandbox
- Soporte para fork() y execvp()

La compatibilidad con Windows fue descartada deliberadamente una vez que migramos al entorno Linux del aula.

**Limitaciones Conocidas de la Shell**

Para mantener el enfoque minimalista y cumplir con el alcance del TP, hay funcionalidades que la shell **no implementa**:

- No soporta | (pipes)
- No soporta >> (redirección de append)
- No soporta ejecución en background (&)
- No soporta comillas dobles ni simples
- No maneja variables de entorno complejas
- No permite rutas absolutas fuera del HOME por la sandbox
- No implementa cd - ni cd ~
- grep no soporta expresiones regulares avanzadas

Estas limitaciones son deliberadas y coherentes con el objetivo del trabajo.

**Justificación del Diseño**

Durante el desarrollo tomamos decisiones conscientes:

- Usar dup2() en vez de freopen() para tener control completo sobre descriptores.
- Implementar grep de forma manual para respetar el enfoque minimalista del TP.
- Mantener ls basado en syscalls y no en system() para evitar procesos extra.
- Evitar dependencias externas para compilar fácilmente en el ambiente LFS.
- Implementar sandbox por software en lugar de un chroot para evitar privilegios root.

Estas decisiones permitieron un proyecto consistente, portable y alineado con las restricciones académicas.