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: 8vo ciclo

Turno: Mañana

Integrantes:

Nicolás Arturo Marin Gonzales (2022-119082)

Roger Huanca

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

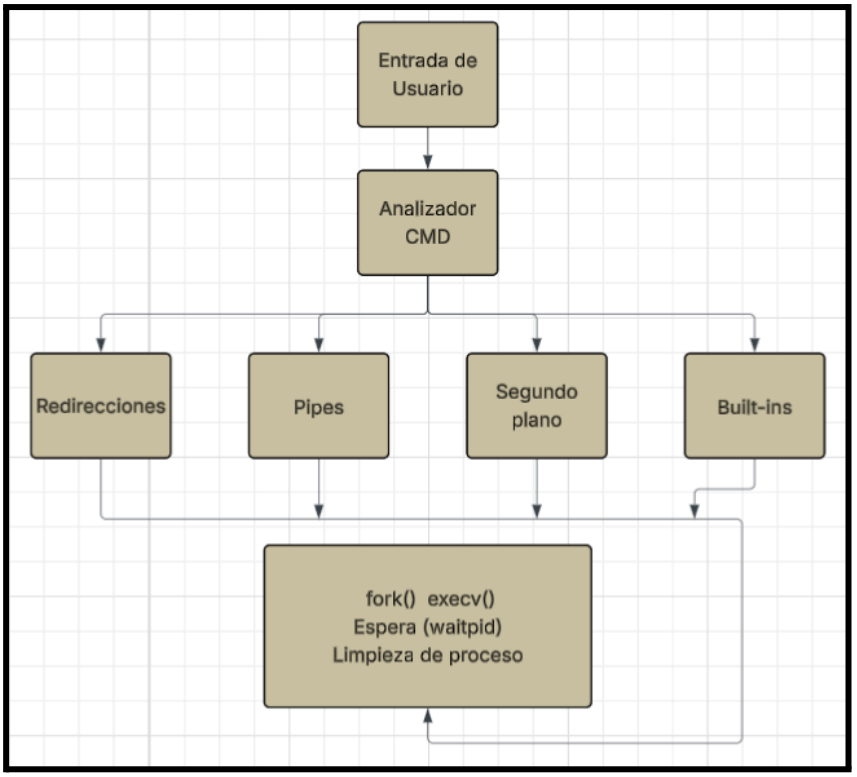

Imagen 1. Elaboración propia de diagrama general interno del funcionamiento de la mini-Shell.

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 MiniShell 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.

8. Anexos