UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS

IMPLEMENTACIÓN DE UN INTÉRPRETE DE COMANDOS

ASIGNATURA:
Sistemas operativos

SECCIÓN:
Grupo A

INTEGRANTES:
Maykol David Espinoza Kquerare 2025-11902c
Luz Marina Flores Carbajal 2025-11901c

FECHA:
Octubre del 2025

1. Objetivos y Alcance

1.1 Objetivo general

Implementar una mini shell en C++ que cumpla con los principios de los sistemas operativos tipo UNIX, utilizando llamadas al sistema POSIX para la creación y control de procesos, manejo de señales, redirección de entrada/salida, concurrencia mediante hilos y gestión de memoria dinámica instrumentada.

1.2. Objetivos específicos

- Comprender y aplicar el uso de las funciones POSIX: fork, execvp, waitpid, pipe, dup2, open, close, sigaction, pthread_create y pthread_join.
- Desarrollar un programa modular en C++ que simule el comportamiento básico de una shell UNIX.
- Implementar redirecciones, pipes, ejecución en segundo plano, alias, historial y comandos internos (cd, pwd, echo, etc.).
- Medir y mostrar estadísticas del uso de memoria dinámica mediante instrumentación (new/delete sobrescritos).
- Validar la correcta gestión de procesos e hilos mediante pruebas funcionales.

1.3. Alcance

El proyecto cubre:

- Ejecución de comandos externos y built-ins.
- Redirección de I/O (>, <, >>), tuberías (|) y ejecución en segundo plano (&).
- Múltiples hilos (parallel) y manejo de señales.
- Medición del uso de memoria (meminfo).
- No se incluyen funciones avanzadas como pipes múltiples o scripting interactivo.

2. Arquitectura y diseño

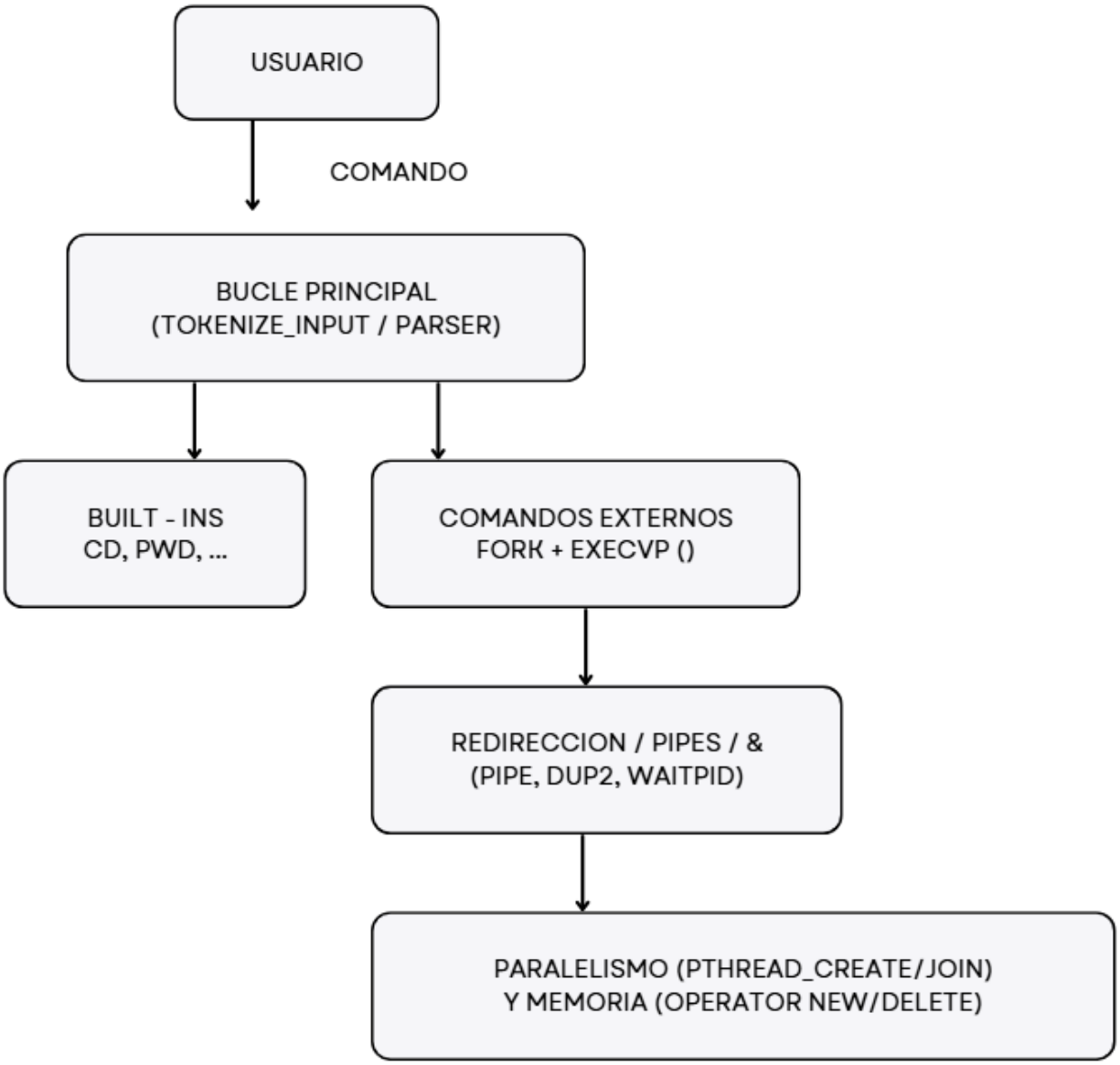
El diseño de la mini-shell se basó en un enfoque modular para separar las responsabilidades.

2.1 Estructura general

Archivo	Función
main.cpp	Bucle principal del shell y manejo de señales.
ShellCore.cpp/h	Ejecución de comandos, pipes, redirecciones y built-ins.
MemoryManager.cpp/h	Instrumentación de memoria.
Globals.cpp/h	Variables globales y manejadores de señales.

El sistema se organiza alrededor de un bucle principal interactivo que constituye el núcleo de la shell. Este bucle muestra en pantalla un prompt dinámico con el directorio de trabajo actual, recibe y tokeniza el comando ingresado por el usuario, determina si se trata de una instrucción interna o un programa externo, y posteriormente gestiona su ejecución. Durante este proceso, el sistema maneja la sincronización de procesos e hilos, garantizando la correcta coordinación entre tareas concurrentes y la estabilidad general del entorno de ejecución.

2.2. Diagrama de arquitectura



3.Detalles de implementación

3.1. POSIX usadas y su función

POSIX	Módulo principal	Propósito en la Shell
fork()	ShellCore	Crea un proceso hijo idéntico para ejecutar comandos externos en un entorno aislado.
execv()/execvp()	ShellCore	Reemplaza la imagen del proceso hijo con el programa a ejecutar.
waitpid()	ShellCore, main	Utilizado en el padre para esperar la finalización del hijo foreground o para recolección no bloqueante de tareas en segundo plano.
pipe()	ShellCore	Crea un canal de comunicación para conectar stdout de cmd1 a stdin de cmd2.
dup2()	ShellCore	Redirige descriptores de archivo.
open()/close()	ShellCore	Apertura de archivos con banderas de truncamiento, adjuntar o lectura para las redirecciones.
sigaction()	main	Configura los signal handlers SIGCHLD y SIGINT.
pthread_create()	ShellCore	Crea y espera hilos para la ejecución concurrente del built-in paralelo

3.2. Decisiones clave

- Uso de std::filesystem para cambiar directorio (cd) y obtener la ruta (pwd).
- Sobrecarga global de new/delete con contadores atómicos para medir memoria.
- Implementación modular para facilitar mantenimiento y escalabilidad.

4. Concurrencia y sincronización

El comando paralelo permite ejecutar múltiples procesos simultáneamente, cada uno desde un hilo POSIX independiente.

Cada hilo crea un proceso hijo con fork() y espera su finalización.

4.1. Estrategias para evitar condiciones de carrera

- Uso de std::atomic para contadores globales de memoria (g_total_alloc, g_total_free).
- Separación de contexto: cada proceso o hilo trabaja sobre copias de sus vectores de argumentos, evitando colisiones.

5. Gestión de memoria

La gestión de memoria fue abordada con una doble estrategia: adoptar prácticas seguras de C++ moderno y proporcionar una instrumentación explícita para el análisis del heap.

5.1. Estrategia de uso

- Sobrecarga global de los operadores new y delete.
- Cada asignación incrementa g_total_alloc, y cada liberación incrementa g_total_free.
- Estos contadores son atómicos, evitando condiciones de carrera en entornos concurrentes.

5.2. Evidencia de Uso

El comando meminfo muestra en tiempo real:

Asignaciones: 131

Liberaciones: 124

Esto permite verificar fugas o uso excesivo de memoria en ejecución.

6. Pruebas y resultados

Comando	Descripción	Resultado
pwd	Muestra el directorio actual	Correcto
echo hola > prueba.txt	Redirección de salida	Crea archivo
cat < prueba.txt	Redirección de entrada	Muestra 'hola'
sleep 2 &	Generea proceso en segundo plano	Prompt no bloqueado
alias ll = ls -l	Alias funcional	Correcto
paralelo "sleep 2" "echo termino"	Hilos concurrentes	Ejecución en paralelo
meminfo	Muestra métricas de memoria	Correcto

7. Conclusiones y trabajos futuros

7.1. Conclusiones

El desarrollo de la shell basada en POSIX permitió consolidar los conocimientos sobre gestión de procesos, señales, concurrencia y memoria dinámica en entornos tipo UNIX. La implementación demostró un correcto uso de las llamadas al sistema, logrando un funcionamiento estable y eficiente. Gracias a su arquitectura modular, fue posible integrar funcionalidades adicionales como alias, historial y monitoreo de memoria sin comprometer la el sistema. Por lo tanto, podemos decir que el proyecto cumple con los requisitos establecidos.

7.2. Trabajos Futuros

Entre las posibles mejoras se plantea la incorporación de soporte para múltiples pipes encadenados, la persistencia de historial y alias mediante archivos de configuración, y la inclusión de colores o formatos personalizados en el prompt para mejorar la interacción con el usuario. También, se busca implementar el manejo avanzado de comillas y variables de entorno, junto con mecanismos de sincronización más seguros que utilicen bloqueos (locks) sobre estructuras compartidas.