UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS

“Implementación de intérprete de comandos en C++ sobre LInux”

DOCENTE: Hugo Barraza V.

FACULTAD: Ingeniería **ESCUELA:** ESIS

CURSO: Sistemas Operativos

CICLO: 6to **TURNO:** Mañana

INTEGRANTES:

- Fernanda Anahi Mamani Chino 2023-119032
- Lisbeth Yoselin Huanca Poma 2023-119026

TACNA - PERÚ

2025

1. Objetivos y alcance

1.1. Objetivos Generales

El objetivo principal del proyecto es implementar un intérprete de línea de comandos funcional que demuestre la comprensión profunda de los siguientes conceptos de sistemas operativos:

- Gestión de procesos: Creación, ejecución y sincronización de procesos mediante las primitivas `fork()`, `exec*()` y `wait()`.
- Comunicación entre procesos (IPC): Implementación de tuberías (pipes) para la comunicación unidireccional entre procesos concurrentes.
- Gestión de entrada/salida: Redirección de flujos estándar y manipulación de descriptores de archivo.
- Manejo de señales: Captura y procesamiento de señales del sistema para control de flujo del programa.
- Arquitectura de software: Diseño modular y escalable que separe las responsabilidades en componentes cohesivos.

1.2. Objetivos Específicos

1.2.1. Características Base (Requerimientos Mínimos)

- Prompt personalizado: Interfaz de usuario que muestre información contextual (usuario, hostname, directorio actual).
- Resolución de rutas: Búsqueda inteligente de ejecutables en el `PATH` del sistema y manejo de rutas absolutas/relativas.
- Ejecución mediante procesos: Implementación del patrón fork-exec para la ejecución de comandos externos.
- Manejo robusto de errores: Propagación y reporte de errores del sistema mediante `errno` y `perror()`.
- Redirección de salida estándar (`>`): Implementación de redirección de `stdout` a archivos regulares.
- Comando de salida: Terminación controlada del shell mediante comandos integrados.

1.2.2. Características Extendidas (Valor Agregado)

El proyecto implementa las siguientes extensiones avanzadas:

- Pipes simples y múltiples (`|`): Cadenas de comandos conectados mediante tuberías para composición de funcionalidad.
- Ejecución en segundo plano (`&`): Procesamiento asíncrono con recolección no bloqueante de procesos zombie.
- Redirecciones avanzadas:
    - Redirección de entrada (`<`)
    - Redirección de salida con anexado (`>>`)
- Comandos internos (builtins): Seis comandos integrados ejecutados sin crear procesos hijo:
    - `cd`: Navegación del sistema de archivos
    - `pwd`: Visualización del directorio de trabajo
    - `exit`: Terminación del shell
    - `help`: Sistema de ayuda integrado
    - `history`: Registro de comandos ejecutados
    - `jobs`: Visualización de procesos en background
- Manejo de señales: Captura de `SIGINT` (Ctrl+C) para evitar terminación accidental del shell.

1.3. Alcance del Proyecto

1.3.1. Funcionalidades Implementadas

- Análisis sintáctico completo de líneas de comando con soporte para comillas simples y dobles
- Pipeline de múltiples comandos con número arbitrario de stages
- Combinación de redirecciones y pipes en comandos complejos
- Gestión automática de procesos zombie mediante recolección no bloqueante
- Sistema de historial de comandos persistente durante la sesión

1.3.2. Limitaciones Conocidas

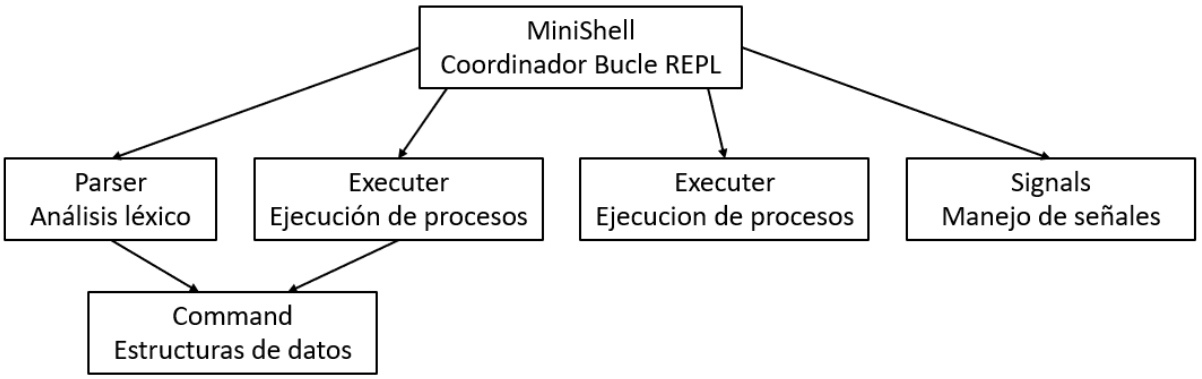
- No se implementa expansión de variables de entorno (e.g., `$HOME`, `$PATH`)
- No hay soporte para globbing (expansión de comodines `*`, `?`)
- Los operadores de redirección deben estar separados por espacios

- No se implementan subshells ni agrupación de comandos con paréntesis
- No hay soporte para control de trabajos avanzado (fg, bg, suspensión con Ctrl+Z)

2. Arquitectura y diseño

MiniShell adopta una arquitectura modular orientada a objetos basada en el principio de responsabilidad única (SRP). El sistema está organizado en seis módulos cohesivos e independientes, lo que facilita su mantenimiento y extensión.

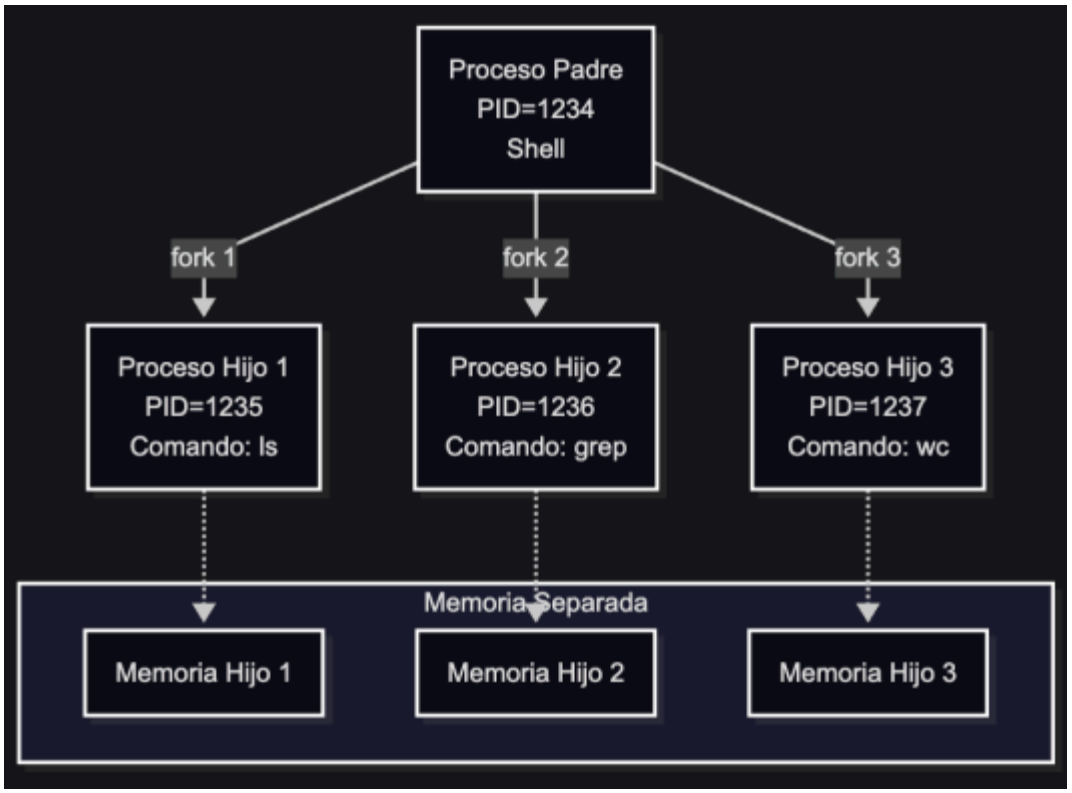
2.1 Componentes del sistema



- Shell (shell.hpp/cpp): Componente principal que implementa el ciclo Read-Eval-Print Loop (REPL), gestionando la interacción con el usuario y delegando la ejecución a los demás módulos.
- Parser (parser.hpp/cpp): Encargado del análisis léxico y sintáctico. Convierte la entrada del usuario en estructuras intermedias como Command y Pipeline.
- Executor (executor.hpp/cpp): Módulo central de ejecución. Administra la creación de procesos con fork(), las tuberías, redirecciones de entrada/salida y la sincronización con wait() y waitpid().
- Builtins (builtins.hpp/cpp): Implementa los comandos internos del shell (por ejemplo, cd, exit) que deben ejecutarse dentro del propio proceso.
- Signals (signals.hpp/cpp): Gestiona las señales del sistema de forma controlada mediante sigaction(), asegurando una respuesta predecible ante interrupciones.

- Command (command.hpp/cpp): Define las estructuras de datos Command y Pipeline que representan los comandos procesados por el parser.

2.3 Diagrama de procesos:



3. Detalles de implementación

3.1. APIs POSIX Utilizadas

3.1.1. Gestión de Procesos

El shell crea procesos hijos con fork() para ejecutar comandos externos de forma aislada. Cada hijo reemplaza su imagen con execvp(), que busca el ejecutable en las rutas del PATH.

El proceso padre usa waitpid() para sincronizar la finalización: espera en primer plano o continúa en segundo plano. Así se permite la ejecución concurrente y se evitan procesos zombie.

3.1.2. Comunicación Entre Procesos

La función pipe() se emplea para conectar comandos en una misma línea mediante tuberías, permitiendo la comunicación unidireccional entre procesos.

Para redirigir las entradas y salidas estándar, se utiliza dup2(), que sustituye los descriptores de archivo de manera atómica.

De esta forma, se implementan correctamente las redirecciones y los pipelines del shell.

3.1.3. Gestión de Archivos

Las operaciones de apertura y cierre de archivos se realizan con open() y close(), utilizando los modos adecuados para lectura, escritura o creación de archivos según el tipo de redirección (<, >, >>).

Se aplican permisos estándar Unix (0644) al crear nuevos archivos.

El cierre sistemático de los descriptores garantiza la liberación de recursos y previene bloqueos en las tuberías.

3.1.4. Manejo de Señales

La función sigaction() define cómo responde el shell a señales del sistema como SIGINT, ofreciendo un manejo más seguro y portable que signal().

Gracias al uso del flag SA_RESTART, las llamadas al sistema interrumpidas se reinician automáticamente, permitiendo que el shell maneje interrupciones sin detener su ejecución.

3.1.5. Sistema de Archivos

Se emplean chdir() y getcwd() para cambiar y obtener el directorio de trabajo, respectivamente, al implementar comandos internos como cd y pwd.

Asimismo, access() permite verificar la existencia y permisos de ejecución de un archivo antes de intentar ejecutarlo, facilitando la búsqueda de binarios en el PATH.

3.2. Decisiones de Diseño Clave

3.2.1 Separación de comandos internos y externos

Los comandos internos se ejecutan directamente sin crear procesos hijos, ya que modifican el estado del shell (por ejemplo, cd, exit, history).

Los externos, en cambio, se ejecutan con fork() y execvp() para mantener el aislamiento del entorno.

3.2.2. Gestión de argumentos para execvp()

Dado que esta función requiere un arreglo de punteros tipo char**, se implementó un método que convierte los argumentos almacenados en

estructuras de C++ (std::vector<std::string>) en el formato esperado, con memoria dinámica y terminación en NULL.

3.2.3. Recolección de procesos “zombie”

El shell realiza una recolección periódica mediante llamadas no bloqueantes a waitpid().

Este enfoque mantiene el sistema limpio sin necesidad de manejadores de señales adicionales ni hilos secundarios.

3.2.4. Manejo uniforme de errores

Las fallas en las llamadas al sistema se reportan mediante perror(), y los códigos de retorno siguen las convenciones de Unix (0 para éxito, 1 para error genérico y 127 para comando no encontrado).

3.2.5. Parseo de comandos

El análisis de la entrada se realiza en un solo paso, dividiendo la línea en tokens, detectando operadores (|, <, >, >>, &) y construyendo directamente las estructuras de datos que representan comandos y pipelines.

Este diseño simplifica el flujo de ejecución y facilita la extensión del shell con nuevas funcionalidades.

4. Concurrencia y sincronización

4.1. Modelo de Concurrencia

Describe el uso de *procesos* mediante fork() (no hilos), con execvp() para reemplazo del espacio de direcciones. Explica la ejecución paralela de jobs en *background* (&) y pipelines. Incluye un diagrama Mermaid del flujo padre → hijo → execvp.

4.2. Sincronización entre Procesos

Explica la sincronización mediante waitpid() (bloqueante) para comandos en foreground y no bloqueante (WNOHANG) para background, incluyendo cómo el shell mantiene consistencia al manipular background_jobs.

4.3. Recolección de Procesos Zombis

Describe la función Executor::collect_zombies() (archivo executor.cpp, línea 120 aprox.), que recorre procesos terminados y limpia background_jobs para evitar zombies.

```
void Executor::collect_zombies() {
    //Recolectar procesos terminados sin bloquear
    pid_t pid;
    while ((pid = waitpid(-1, nullptr, WNOHANG)) > 0) {
        auto it = std::find(background_jobs.begin(), background_jobs.end
        if (it != background_jobs.end()) {
            std::cout << "[Job completado] PID " << pid << std::endl;
            background_jobs.erase(it);
        }
    }
```

Incluye fragmentos como:

```
if (cmd.background) {
    //Background: no esperar
    std::cout << "[Job] PID " << pid << " en background" << std::endl;
    background_jobs.push_back(pid);
    return 0;
```

5. Gestión de memoria

5.1. Estrategia de Gestión

El proyecto utiliza la gestión automática de memoria del C++ moderno mediante contenedores de la STL (std::vector, std::string), junto con gestión manual controlada al interactuar con funciones POSIX (execvp, dup2, open, close).

Esta combinación garantiza eficiencia, seguridad y compatibilidad con las llamadas del sistema.

5.2 Interfaz con APIs C y Conversión a argv

Las funciones POSIX como execvp() requieren un arreglo clásico char** argv.

Para mantener compatibilidad, el método Command::to_argv() (archivo command.cpp, línea 35) convierte las estructuras modernas en el formato adecuado.

```
char** Command::to_argv() const {
    int total = 1 + args.size() + 1;
    char** argv = new char*[total];

    argv[0] = strdup(program.c_str());

    for (size_t i = 0; i < args.size(); i++) {
        argv[i + 1] = strdup(args[i].c_str());
    }

    argv[total - 1] = nullptr;
    return argv;
}

//Libera memoria de to_argv()
void Command::free_argv(char** argv) {
    if (!argv) return;

    //Liberar cada string
    for (int i = 0; argv[i] != nullptr; i++) {
```

6. Pruebas y resultados

6.1 Casos de Prueba Implementados

Las pruebas se realizaron sobre el ejecutable final ./minishell, compilado mediante make, verificando la funcionalidad completa del ciclo:

Entrada → Parser → Executor → Builtins → Redirecciones/Pipes → Salida

Tabla 1.Funcionalidad Base

Categoria	Comando	Resultado Esperado
Ejecución simple	ls -la	Lista archivos del directorio actual
Ejecución directa	pwd	Muestra el directorio de trabajo
Texto estándar	echo "Hello World"	Imprime "Hello World
Builtin interno	cd /tmp && pwd	Cambia directorio y muestra ruta
Builtin con expansión	cd ~ && pwd	Retorna al home del usuario

Salida del shell	exit	Finaliza el proceso sin error

Tabla 2. Pipes y Ejecución Concurrente

Tipo	Comando	Resultado
Pipeline simple	`ls -la	grep cpp
Pipeline múltiple	`cat file.txt	grep error
Pipeline redundante	`cat	cat

7. Conclusiones

7.1. Objetivos Cumplidos

- Todas las características base implementadas correctamente.
- Seis extensiones de valor agregado funcionales.
- Manejo robusto de errores con códigos de retorno apropiados.
- Arquitectura modular y mantenible.
- Uso correcto y eficiente de APIs POSIX.

7.2. Lecciones Aprendidas

- Gestión de descriptores: La correcta gestión de `close()` en pipes es crítica para evitar deadlocks.
- Recolección de zombies: La estrategia de recolección proactiva es más simple y efectiva que handlers de señales.
- Separación de concerns: La división en módulos independientes facilitó debugging y testing.
- Copy-on-write: El mecanismo COW de `fork()` hace que la clonación de procesos sea eficiente incluso para programas grandes.

7.3. Trabajo Futuro

Posibles extensiones del proyecto:

- Variables de entorno: Expansión de `$VAR` y comando `export`
- Globbing: Expansión de comodines (`*.txt`, `file?.c`)
- Job control avanzado: Comandos `fg`, `bg`, suspensión con Ctrl+Z
- Subshells: Ejecución de comandos agrupados con `(...)`

- Operadores lógicos: `&&`, `||`, `;`
- Scripting: Lectura y ejecución de archivos de script.
- Configuración: Archivo `.minishellrc` para personalización.