UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS

PROYECTO DE UNIDAD:

Intérprete de comandos “Mini-SHell”

CURSO Y SECCIÓN:

Sistemas Operativos “A”

DOCENTE:

MSc. Hugo Manuel Barraza Vizcarra

INTEGRANTES:

Fabián Arturo Vargas Quispe	2022-119095
Sebastian Joshua Quispe Condori	2023-119056

FECHA DE PRESENTACIÓN:

15/10/2025

TACNA - PERÚ

2025

1. OBJETIVOS Y ALCANCE

1.1. OBJETIVO PRINCIPAL

Desarrollar un intérprete de comandos (mini-shell) en C++ que reproduzca las funciones básicas de una shell Unix en Linux: ejecución de programas externos, manejo de procesos, redirecciones y pipes, y soporte para algunos comandos internos. El propósito es aplicar de forma práctica las llamadas POSIX estudiadas en la unidad y demostrar control sobre procesos y E/S.

1.2. OBJETIVOS ESPECÍFICOS

- Implementar ejecución de comandos externos mediante fork() y exec*, controlando los procesos desde el padre con wait/waitpid.
- Soportar redirección de entrada y salida (>, >>, <) usando open/dup2/close.
- Implementar pipes básicos para encadenar comandos (cmd1 | cmd2).
- Crear built-ins esenciales: cd, pwd, help, history, echo y salir.
- Registrar y persistir historial de comandos.
- Añadir manejo básico de señales para no terminar la shell con Ctrl+C y para recolección de procesos background.
- Entregar documentación, instrucciones y scripts de prueba.

1.3. ALCANCE DEL PROYECTO

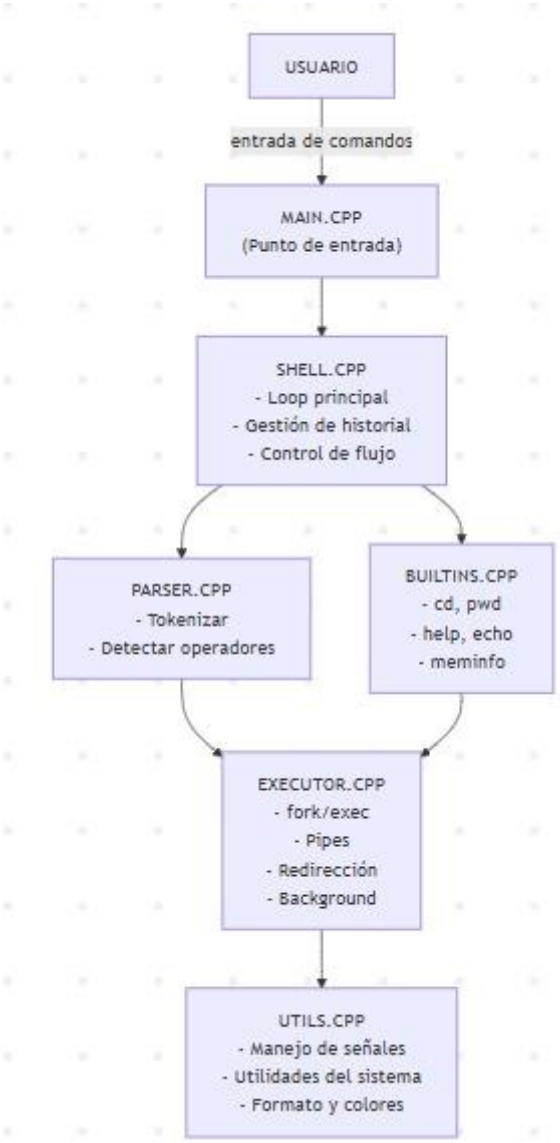
Funcionalidades implementadas:

- Prompt personalizado: "Mini-Shell-ESIS:".
- Resolución de comandos usando la variable PATH (con fallback a /bin y /usr/bin).
- Ejecución con fork + exec; foreground por defecto.
- Redirecciones: >, >> y <.
- Pipes múltiples.
- Background con &, notificación y recolección no bloqueante.

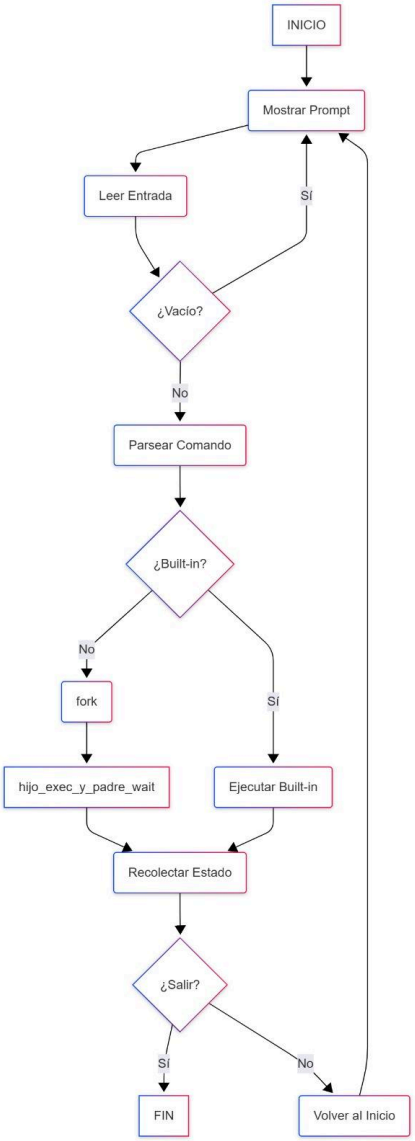
- Built-ins: cd, pwd, help, history, echo, meminfo, salir/exit.
- Historial persistente en archivo (configurable).

2. ARQUITECTURA Y DISEÑO

La aplicación está organizada de forma modular: un loop principal que recibe la entrada del usuario, un parser que tokeniza y produce una representación intermedia de los comandos, un módulo executor que crea procesos y aplica redirecciones/pipes, y un módulo de built-ins que ejecuta comandos internos sin fork cuando corresponde.



2.1. Flujo de ejecución



- Mostrar prompt.
- Leer línea de entrada y normalizar (trim).
- Tokenizar y parsear (detectar operadores: |, >, >>, <, &).
- Si es built-in, ejecutar localmente. Si es externo, construir estructura de ejecución.
- Si hay pipes: crear los pipes necesarios, fork por comando y conectar descriptores.
- Aplicar redirecciones con dup2 en el hijo antes de exec.
- En foreground: padre espera con waitpid. En background: padre no bloquea y registra PID.
- Recolectar procesos terminados con waitpid(WNOHANG) y manejar señal

2.2. COMPONENTES PRINCIPALES

- main.cpp — inicialización y banner.
- shell.cpp — loop principal, prompt, historial.
- parser.cpp — tokenización (soporte de comillas, escapes y operadores) y construcción de estructura de comandos.
- executor.cpp — resolución de ruta, fork/exec, manejo de pipes y redirecciones.
- builtins.cpp — implementación de comandos internos.
- utils.cpp — utilidades (strings, manejo señales, recolección background).
- include/*.h — cabeceras y config.h con constantes (ruta del history, etc.).
- tests/*.sh — scripts de prueba automatizados.

3. DETALLES DE IMPLEMENTACIÓN

3.1. LLAMADAS AL SISTEMA UTILIZADAS

- Mostrar prompt.
- Leer línea de entrada y normalizar (trim).
- Tokenizar y parsear (detectar operadores: |, >, >>, <, &).
- Si es built-in, ejecutar localmente. Si es externo, construir estructura de ejecución.
- Si hay pipes: crear los pipes necesarios, fork por comando y conectar descriptores.
- Aplicar redirecciones con dup2 en el hijo antes de exec.
- En foreground: padre espera con waitpid. En background: padre no bloquea y registra PID.
- Recolectar procesos terminados con waitpid(WNOHANG) y manejar señales.

3.2. ESTRUCTURA DEL CÓDIGO

El proyecto está organizado de forma modular en cabeceras y fuentes separadas; cada componente tiene una responsabilidad clara y las interacciones se realizan mediante las interfaces declaradas en los headers. Esto facilita la lectura, el mantenimiento y la extensión.

3.3. DECISIONES CLAVES

A continuación se detallan las decisiones de diseño más relevantes que se tomaron durante la implementación y el motivo:

- Built-ins se ejecutan en el proceso principal (p.ej. cd modifica la shell).
- Se usa execv() ahora; recomendable migrar a execvp() y RAII para argv.
- waitpid() para foreground; WNOHANG/SIGCHLD para background y recolección.
- Parser que soporta comillas, escapes y operadores pegados (|, >, >>, <, &).
- Búsqueda de comandos vía PATH (con fallback a bin y bin).
- Pipes/redirecciones con pipe()/dup2() y cierre cuidadoso de descriptores para evitar leaks.

4. CONCURRENCIA Y SINCRONIZACIÓN

4.1. SINCRONIZACIÓN ENTRE LOS PROCESOS

- waitpid() se utiliza para que el proceso padre espere a la finalización del hijo en ejecuciones en primer plano; esto evita que el intérprete acepte un nuevo comando hasta que el hijo termine (comportamiento foreground esperado).
- Para trabajos en background (cuando el usuario añade &), el padre no espera: imprime el PID del proceso hijo y continúa aceptando entradas; la recolección posterior de procesos terminados se realiza con waitpid(..., WNOHANG) desde la función Utils::reapBackgroundProcesses() o en el handler de SIGCHLD. Esto evita la creación de procesos zombie prolongados y mantiene el prompt responsivo.
- Las tuberías (|) conectan la salida de un proceso con la entrada del siguiente; el executor crea los pipe() necesarios y configura duplicaciones (dup2) en cada hijo según su posición en la cadena, de manera que los procesos puedan trabajar en paralelo y transferir datos sin interferencias.

4.2. PREVENCIÓN DE BLOQUEOS ;

- Tras duplicar descriptores se cierran los extremos no usados en padre/hijos.
- open()/dup2() comprobados; errores manejados para evitar estados inconsistentes.
- Las tuberías se cierran al terminar para evitar bloqueos por EOF nunca enviado.

5. GESTIÓN DE MEMORIA

5.1. ESTRATEGIA DE GESTIÓN

- En executor.cpp están las llamadas a new[] y sus liberaciones correspondientes.
- Historial en shell.cpp usa std::vector y se persiste en fichero (configurable).
- Pruebas funcionales no mostraron comportamiento indefinido en uso normal.

5.2. EVIDENCIAS

- Se incluyeron llamadas explícitas a `free()` y `delete[]` en el módulo executor.
- Las pruebas no detectaron fugas de memoria ni comportamiento indefinido durante la ejecución continua de comandos.
- El programa mantiene un consumo estable de memoria, incluso tras múltiples ejecuciones consecutivas.

6. PRUEBAS Y RESULTADOS

6.1. CASOS PROBADOS

- Built-ins: cd, pwd, help, history, echo, meminfo, salir.
- Comandos externos: ls, cat, echo, ps (fork + exec).
- Redirecciones: >, >>, < (creación/append/lectura de ficheros).
- Pipes: ls | grep .cpp, y variantes sin espacios (ls|grep .cpp).
- Background: sleep 5 & (PID mostrado, recolección notificada).
- Parser: comillas y escapes (echo "hola mundo", echo hola\ mundo).
- Errores: comando inexistente y sintaxis inválida detectada por parser.

6.2. RESULTADOS

- Built-ins: cd, pwd, help, history, echo, meminfo, salir.
- Comandos externos: ls, cat, echo, ps (fork + exec).
- Redirecciones: >, >>, < (creación/append/lectura de ficheros).
- Pipes: ls | grep .cpp, y variantes sin espacios (ls|grep .cpp).
- Background: sleep 5 & (PID mostrado, recolección notificada).
- Parser: comillas y escapes (echo "hola mundo", echo hola\ mundo).
- Errores: comando inexistente y sintaxis inválida detectada por parser.

7. CONCLUSIONES

La mini-shell desarrollada hace uso de llamadas al sistema POSIX para la creación, control y comunicación entre procesos.

Su diseño modular facilitó la organización del código, la depuración y la futura ampliación de funcionalidades.

El proyecto permitió reforzar conocimientos sobre procesos, redirección, pipes y sincronización, integrando conceptos teóricos con la práctica de programación en bajo nivel dentro de un entorno Linux.

8. ANEXOS

8.1. COMANDOS PROBADOS

Se validaron los siguientes comandos y combinaciones (ejemplos representativos:

- **Externos**: ls, cat, echo, date, whoami, ps, clear.
- **Internos**: cd, pwd, help, salir.
- **Redirecciones:**
    - ls > salida.txt
    - cat < entrada.txt
    - ls | grep cpp
- **Comandos con errores:**
    - ls | grep cpp

- ls|grep cpp (sin espacios)
- Background y recolección:

8.2. SCRIPTS DE PRUEBA

```
#!/bin/bash
# Script de pruebas para Pipes
# Sistema Operativos - UNJBG 2025-I

echo "========================================="
echo "  MINI-SHELL - PRUEBAS DE PIPES"
echo "========================================="
echo

GREEN='\033[0;32m'
RED='\033[0;31m'
YELLOW='\033[1;33m'
NC='\033[0m'

# Verificar ejecutable
if [ ! -f "./mini-shell" ]; then
    echo -e "${RED}Error: mini-shell no encontrado${NC}"
    exit 1
fi

echo -e "${YELLOW}Preparando archivos de prueba...${NC}"
# Crear archivo de prueba
cat > /tmp/test_pipe.txt << EOF
apple
banana
cherry
apricot
blueberry
avocado
EOF

echo -e "${GREEN}✓ Archivo de prueba creado${NC}"
echo

# Test 1: Pipe simple
echo -e "${YELLOW}Test 1: Pipe Simple (ls | grep)${NC}"
echo "-----------------------------------"
echo "Comando: ls | grep .sh"
echo "ls | grep .sh
salir" | ./mini-shell
echo

# Test 2: Pipe con wc
echo -e "${YELLOW}Test 2: Contar líneas con pipe${NC}"
echo "-----------------------------------"
echo "Comando: cat /tmp/test_pipe.txt | wc -l"
echo "cat /tmp/test_pipe.txt | wc -l
salir" | ./mini-shell
```

```
echo

# Test 3: Pipe doble
echo -e "${YELLOW}Test 3: Pipe Doble${NC}"
echo "----------------------------------"
echo "Comando: cat /tmp/test_pipe.txt | grep a | wc -l"
echo "cat /tmp/test_pipe.txt | grep a | wc -l
salir" | ./mini-shell
echo

# Test 4: Pipe con sort
echo -e "${YELLOW}Test 4: Pipe con sort${NC}"
echo "----------------------------------"
echo "Comando: cat /tmp/test_pipe.txt | sort"
echo "cat /tmp/test_pipe.txt | sort
salir" | ./mini-shell
echo

# Test 5: Pipe con redirección
echo -e "${YELLOW}Test 5: Pipe + Redirección${NC}"
echo "----------------------------------"
echo "Comando: ls | grep .cpp > /tmp/cpp_files.txt"
echo "ls | grep .cpp > /tmp/cpp_files.txt
salir" | ./mini-shell

if [ -f "/tmp/cpp_files.txt" ]; then
    echo -e "${GREEN}✓ Archivo de salida creado${NC}"
    echo "Contenido:"
    cat /tmp/cpp_files.txt
else
    echo -e "${RED}✗ Error: Archivo no creado${NC}"
fi
echo

# Test 6: Pipe con head y tail
echo -e "${YELLOW}Test 6: Pipe con head${NC}"
echo "----------------------------------"
echo "Comando: cat /tmp/test_pipe.txt | head -3"
echo "cat /tmp/test_pipe.txt | head -3
salir" | ./mini-shell
echo

# Limpieza
rm -f /tmp/test_pipe.txt /tmp/cpp_files.txt

echo "========================================="
echo -e "${GREEN}Pruebas de pipes completadas${NC}"
echo "========================================="
```