

UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN
FACULTAD DE INGENIERÍA
ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS



PROYECTO:

Intérprete de comandos en C++ sobre Linux

CURSO Y SECCIÓN:

Sistemas Operativos "A"

DOCENTE:

MSc. Hugo Manuel Barraza Vizcarra

INTEGRANTES:

Arratia Paz, Russell Jhean Paul 2023-119004

Carita Pinchi, Nelly Adriana 2023-119007

FECHA DE PRESENTACIÓN:

15/10/2025

TACNA - PERÚ

2025

1. OBJETIVOS Y ALCANCE

1.1. OBJETIVO PRINCIPAL

El objetivo general de nuestro proyecto es desarrollar un intérprete de comandos en C++ que pueda reproducir las funcionalidades esenciales de una Shell Unix, logrando la ejecución de programas externos, también el manejo de redirecciones y de la gestión de procesos. Lo que se busca es consolidar los conocimientos previos sobre la creación y control de procesos, la concurrencia e hilos, así como también la gestión eficiente de memoria en sistemas Linux, aplicando las llamadas al sistema POSIX.

1.2. OBJETIVOS ESPECÍFICOS

- Ejecutar comandos externos mediante `fork()` y `exec()`, controlando los procesos con `wait()` o `waitpid()`.
- Gestionar redirecciones de entrada y salida usando `dup2()`, `open()` y `close()`.
- Implementar comandos internos (`cd`, `pwd`, `help`, `salir`) sin crear nuevos procesos.
- Incorporar extensiones como `pipes()` o comandos internos para ampliar funcionalidades.
- Aplicar buenas prácticas de programación: modularidad, control de versiones y manejo adecuado de memoria.

1.3. ALCANCE DEL PROYECTO

Funcionalidades principales:

- Prompt personalizado con información del usuario o directorio actual.
- Ejecución de comandos externos mediante `fork()` y `exec()`.
- Comandos internos: `cd`, `pwd`, `help`, `history`, `alias` y `salir`.
- Redirección de salida (`>`) y manejo de errores claros.

Extensiones implementadas:

- `Pipes()` para comunicación entre procesos.
- Comandos internos (`built-ins`) para ejecutar ciertas operaciones dentro del mismo intérprete, sin crear un proceso nuevo.

Fuera del alcance:

No se incluyen múltiples pipes, redirecciones dobles (`>>`), historial persistente ni interfaz gráfica. La gestión de memoria se limita al uso básico de estructuras internas de C++.

2. ARQUITECTURA Y DISEÑO

El intérprete de comandos desarrollado se estructura bajo una arquitectura modular, compuesta por componentes que se encargan de gestionar de manera independiente las operaciones principales: lectura e interpretación de comandos, creación y control de procesos, manejo de redirecciones, concurrencia con hilos y gestión de memoria.

2.1. ESTRUCTURA GENERAL

El sistema sigue un flujo secuencial basado en un bucle principal (loop del intérprete) que espera la entrada del usuario, interpreta los comandos ingresados y ejecuta las acciones correspondientes. Cada comando se analiza para determinar si corresponde a:

- Un comando interno (ejecutado directamente dentro de la shell).
- Un comando externo, que requiere la creación de un nuevo proceso hijo mediante `fork()` y su reemplazo con `exec()`.

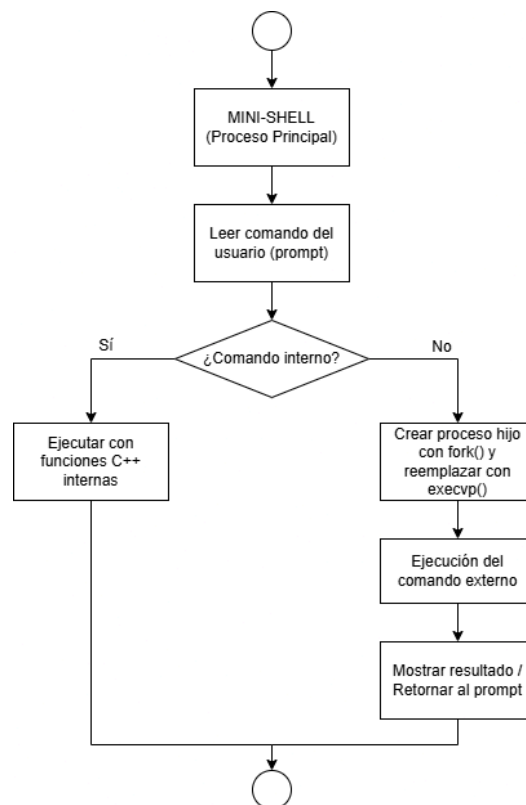
El proceso padre se mantiene en espera hasta que el hijo finalice, a menos que se trate de una tarea en segundo plano, en cuyo caso continúa aceptando nuevos comandos.

2.2. COMPONENTES PRINCIPALES

- **Módulo de lectura e interpretación:** Encargado de mostrar el prompt, recibir la entrada del usuario y dividirla en tokens. Implementa un parser simple que detecta operadores especiales como `>`, `|` y `&`.

- **Módulo de ejecución de comandos:** Utiliza las llamadas al sistema `fork()`, `execvp()` y `waitpid()` para crear y controlar procesos. En el caso de redirecciones, se aplican `dup2()`, `open()` y `close()` sobre los descriptores de archivo estándar.
- **Módulo de comandos internos:** Incluye funciones implementadas dentro de la shell (sin `fork`), como:
 - `cd`: cambia el directorio de trabajo usando `chdir()`.
 - `pwd`: obtiene el directorio actual con `getcwd()`.
 - `help`: muestra una guía básica de uso.
 - `salir`: finaliza la ejecución del intérprete.
 - `alias`: crea atajos personalizados.
 - `history`: muestra el historial de comandos ejecutados.

2.3. DIAGRAMA GENERAL DEL FLUJO



Representación de la ejecución del mini-shell

3. DETALLES DE IMPLEMENTACIÓN

La mini-shell fue desarrollada en C++ bajo entorno Linux, utilizando principalmente llamadas al sistema POSIX para la gestión de procesos, redirecciones, concurrencia y memoria. Su implementación se basa en una estructura modular que separa las funciones de lectura, análisis, ejecución y manejo de errores.

3.1. LLAMADAS AL SISTEMA UTILIZADAS

- `fork()`: crea un nuevo proceso hijo.
- `execvp()`: reemplaza el proceso hijo con el comando solicitado.
- `waitpid()`: sincroniza el proceso padre con el hijo.
- `pipe()`: establece un canal de comunicación entre procesos.
- `dup2()`, `open()`, `close()`: permiten la redirección de flujos estándar.
- `chdir()`, `getcwd()`: implementan los comandos internos `cd` y `pwd`.

3.2. ESTRUCTURA DEL CÓDIGO

El programa se organiza en archivos separados (`parser.cpp`, `executor.cpp`, `builtins.cpp`, etc.) que interactúan a través de cabeceras comunes (`.h`). Esto facilita la legibilidad y el mantenimiento del código.

3.3. DECISIONES CLAVES

Algunas decisiones claves que hemos tomado para el proyecto:

- Los comandos internos se ejecutan directamente en el proceso principal, evitando la creación innecesaria de procesos hijo.
- Se utilizó `execvp()` por su capacidad para manejar listas de argumentos dinámicas.
- El control de flujo entre padre e hijo se realizó con `waitpid()`, garantizando sincronización y evitando procesos huérfanos.
- La gestión de errores y recursos se priorizó mediante verificaciones de retorno en

todas las llamadas al sistema.

- No se implementó concurrencia con hilos para mantener una ejecución secuencial y simplificar la depuración.

4. CONCURRENCIA Y SINCRONIZACIÓN

Aunque no se implementó concurrencia con hilos (`pthread_*`), el programa presenta comportamientos concurrentes naturales derivados del uso de procesos mediante `fork()`.

4.1. SINCRONIZACIÓN ENTRE LOS PROCESOS

- Se usa `waitpid()` para que el proceso padre espere la finalización del hijo antes de continuar, evitando conflictos o terminaciones prematuras.
- La ejecución de `pipes (|)` permite que los procesos trabajen de forma encadenada, compartiendo datos sin interferencias.

4.2. PREVENCIÓN DE BLOQUEOS

- No existen secciones críticas compartidas entre hilos ni variables globales modificadas simultáneamente.
- El manejo correcto de descriptores de archivo y el cierre de pipes evita fugas de recursos y bloqueos.
- La comunicación entre procesos se realiza de forma controlada, asegurando que los flujos se cierren correctamente tras su uso.

5. GESTIÓN DE MEMORIA

La mini-shell hace uso de memoria dinámica principalmente para el almacenamiento temporal de argumentos y tokens.

5.1. ESTRATEGIA DE GESTIÓN

- Uso de arreglos dinámicos de tipo `char* []` y estructuras auxiliares para el manejo de argumentos.

- Liberación de memoria con `free()` y `delete[]` tras la ejecución de cada comando.
- Verificación de punteros nulos antes de liberar memoria para evitar errores de segmentación.

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

Se realizaron diversas pruebas funcionales para verificar el correcto desempeño de la mini-shell, evaluando la ejecución de comandos internos, externos, redirecciones, pipes y tareas concurrentes.

6.1. CASOS PROBADOS

- **Comandos internos:** Se comprobó el correcto funcionamiento de `cd`, `pwd`, `help`, `alias`, `history` y `salir`, verificando que las acciones se ejecuten sin crear procesos adicionales.
- **Comandos externos:** Se probaron instrucciones del sistema como `ls`, `cat`, `echo` y `ps`, asegurando que la shell creara procesos hijo y esperara su finalización con `waitpid()`.
- **Redirecciones:** Se validó la redirección de salida (`>`) hacia archivos, confirmando la escritura correcta y la preservación de los datos previos.
- **Pipes (|):** Se probó la comunicación entre procesos con comandos encadenados, verificando que la salida del primer proceso se dirigiera correctamente al segundo.

6.2. RESULTADOS

- Las pruebas se realizaron desde terminales de Ubuntu.
- El sistema mostró un manejo correcto de errores al ejecutar comandos inexistentes.
- Los pipes y redirecciones funcionaron correctamente, transfiriendo datos entre procesos sin pérdida.
- Se documentaron capturas de pantalla y logs de ejecución para evidenciar el correcto funcionamiento.

7. CONCLUSIONES Y TRABAJOS FUTUROS

La mini-shell desarrollada cumple correctamente con los requisitos funcionales establecidos, demostrando el uso adecuado 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.

Como trabajos futuros estamos considerando:

- Añadir concurrencia mediante hilos (`pthread_*`) para permitir la ejecución paralela de tareas.
- Incluir manejo de señales (`SIGINT`, `SIGCHLD`) para una finalización más controlada de procesos.
- Optimizar la gestión de memoria y recursos, además de mejorar la interfaz de usuario para una experiencia más fluida.

8. ANEXOS

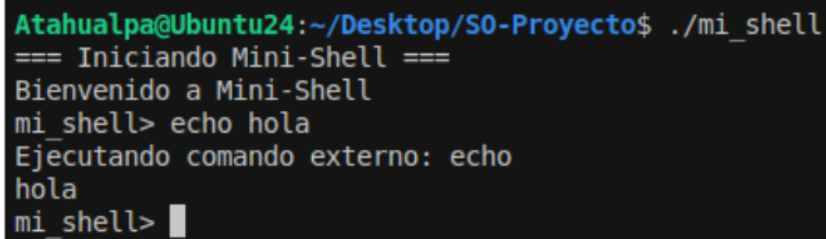
8.1. COMANDOS PROBADOS

Se validaron los siguientes comandos y combinaciones:

- **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:**
 - lszzz → muestra error de comando no encontrado.
 - > archivo → detecta falta de comando previo.

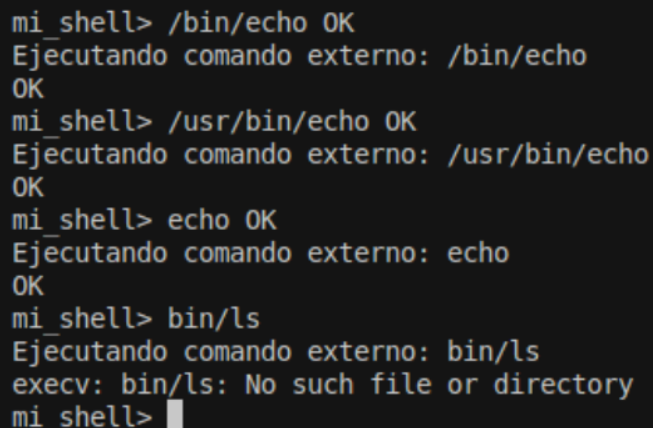
8.2. SCRIPTS DE PRUEBA

- Prompt personalizado



```
Atahualpa@Ubuntu24:~/Desktop/S0-Proyecto$ ./mi_shell
=== Iniciando Mini-Shell ===
Bienvenido a Mini-Shell
mi_shell> echo hola
Ejecutando comando externo: echo
hola
mi_shell> 
```

- Resolución de rutas



```
mi_shell> /bin/echo OK
Ejecutando comando externo: /bin/echo
OK
mi_shell> /usr/bin/echo OK
Ejecutando comando externo: /usr/bin/echo
OK
mi_shell> echo OK
Ejecutando comando externo: echo
OK
mi_shell> bin/ls
Ejecutando comando externo: bin/ls
execv: bin/ls: No such file or directory
mi_shell> 
```

- Ejecución mediante procesos

```
mi_shell> date
Ejecutando comando externo: date
Tue Oct 14 01:06:34 AM UTC 2025
mi_shell> sleep 2
Ejecutando comando externo: sleep
datemi_shell> date
Ejecutando comando externo: date
Tue Oct 14 01:06:46 AM UTC 2025
```

- Manejo de errores

```
mi_shell> foobarbaz123
Ejecutando comando externo: foobarbaz123
no se puede ejecutar /bin/foobarbaz123: No such file or directory
mi_shell> /no/existe/cmd
Ejecutando comando externo: /no/existe/cmd
execv: /no/existe/cmd: No such file or directory
mi_shell> noexec.sh
Ejecutando comando externo: noexec.sh
no se puede ejecutar /bin/noexec.sh: No such file or directory
mi_shell> /bin/
Ejecutando comando externo: /bin/
execv: /bin/: Permission denied
```

- Redirección de salida estándar (>)

```
=== Iniciando Mini-Shell ===
Bienvenido a Mini-Shell
mi_shell> echo hola > out.txt
Ejecutando comando externo: echo
mi_shell> cat out.txt
Ejecutando comando externo: cat
hola
mi_shell> echo linea1 > out.txt
Ejecutando comando externo: echo
mi_shell> echo linea2 > out.txt
Ejecutando comando externo: echo
mi_shell> cat out.txt
Ejecutando comando externo: cat
linea2
```

- Comando de salida

```
Ejecutando comando externo: echo
mi_shell> cat out.txt
Ejecutando comando externo: cat
linea2
mi_shell> salir
¡Hasta luego!
=== Shell finalizada ===
```

- Pipes

```
mi_shell> ls | grep .cpp
builtins.cpp
executor.cpp
line_reader.cpp
main.cpp
parser.cpp
pipe_simple.cpp
redirection.cpp
shell.cpp
mi_shell> /bin/printf "uno\n dos\n tres\n" | wc -l
3
mi_shell> █
```

- Comandos internos (built-ins)

```
Atahualpa@ubuntu: ~/Desktop/S0-Proyecto$ ./mi_shell
=== Iniciando Mini-Shell ===
Bienvenido a Mini-Shell
mi_shell> pwd
/home/Atahualpa/Desktop/S0-Proyecto
mi_shell> cd ..
mi_shell> pwd
/home/Atahualpa/Desktop
mi_shell> history
1: pwd
2: cd ..
3: pwd
4: history
mi_shell> alias ll='ls -la'
mi_shell> ll
Ejecutando comando externo: ls
total 20
drwxr-xr-x  4 Atahualpa Atahualpa 4096 Oct 12 03:24 .
drwxr-x--- 20 Atahualpa Atahualpa 4096 Oct 13 23:33 ..
drwxrwxr-x  5 Atahualpa Atahualpa 4096 Sep 23 01:47 02-procesos-LARMD
-rw-rw-r--  1 Atahualpa Atahualpa  173 Sep 10 03:57 EJ1.ASM
drwxrwxr-x  7 Atahualpa Atahualpa 4096 Oct 14 01:20 S0-Proyecto
mi_shell> alias
alias ll='ls -la'
mi_shell> help
MINI-SHELL - COMANDOS DISPONIBLES
=====
COMANDOS INTERNOS:
  cd [directorio]      - Cambiar directorio
  pwd                  - Mostrar directorio actual
  help                 - Mostrar esta ayuda
  history              - Mostrar historial de comandos
  alias nombre=valor   - Definir alias (valor puede ser varios tokens)
  exit o salir         - Salir del shell

COMANDOS EXTERNOS:
  ls, cat, echo, etc.  - Cualquier comando del sistema

REDIRECCIONES:
  comando > archivo    - Redirigir salida a archivo

mi_shell> █
```