

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



Proyecto de Unidad I

Mini-Shell

Asignatura

SISTEMAS OPERATIVOS

Docente

MSc. Hugo Manuel Barraza Vizcarra

Integrantes:

Josue David Poma Sucso 2022-119090

Hilder Elvis Robles Gonzales 2022-119014

Fecha

14/10/2025

Ciclo: 6to | **Turno:** Mañana

Tacna – Perú

1. Objetivos

El propósito de este proyecto es desarrollar un mini-shell o intérprete de comandos inspirado en las shells tradicionales de UNIX (como bash o sh), capaz de ejecutar programas del sistema y comandos personalizados dentro de un entorno controlado.

El objetivo general es recrear el comportamiento fundamental de una shell, comprendiendo a nivel práctico cómo el sistema operativo gestiona procesos, memoria, entradas y salidas, y la interacción entre el usuario y el kernel a través de llamadas al sistema (APIs POSIX).

De forma más específica, se buscó implementar las siguientes capacidades:

- Mostrar un prompt personalizado que permita al usuario ingresar comandos.
- Interpretar correctamente rutas absolutas y relativas al ejecutar programas.
- Ejecutar procesos mediante las funciones `fork()` y `exec()`, manejando la espera del proceso hijo con `waitpid()`.
- Incorporar mecanismos de redirección de entrada y salida, así como pipes para la comunicación entre procesos.
- Implementar un conjunto de comandos internos (built-ins) tales como `cd`, `pwd`, `help`, `alias`, `history`, `meminfo` y `parallel`.
- Añadir soporte para concurrencia mediante hilos, empleando la biblioteca POSIX Threads.
- Mostrar información sobre el uso de memoria del proceso actual como parte del monitoreo del sistema.

2. Alcance

El presente proyecto abarca el desarrollo completo de un intérprete de comandos funcional, implementado en C++ sobre un entorno Linux, que permite ejecutar programas externos, manejar la entrada y salida estándar, controlar procesos, ejecutar comandos en paralelo mediante hilos, y realizar operaciones de redirección y manejo de memoria.

El sistema desarrollado, denominado Mini-Shell, reproduce las operaciones más comunes de una terminal real, demostrando el uso práctico de conceptos fundamentales de los sistemas operativos, tales como la creación de procesos, la comunicación entre ellos, la sincronización de tareas concurrentes y la gestión eficiente de memoria dinámica.

A continuación, se detalla el alcance funcional del proyecto según los requisitos establecidos:

2.1. Prompt personalizado y lectura de comandos:

La mini-shell despliega un prompt propio en pantalla, indicando que está lista para recibir órdenes del usuario. El sistema lee una línea de comando completa y la analiza mediante un parser que separa los tokens para su posterior ejecución.

2.2. Resolución de rutas y ejecución de comandos externos:

El intérprete es capaz de ejecutar tanto rutas absolutas (por ejemplo, `/usr/bin/ls`) como comandos comunes del sistema, que se buscan automáticamente en el directorio `/bin`. En caso de que el ejecutable no exista o no tenga permisos de ejecución, el programa muestra mensajes de error claros utilizando `perror()` y el código de error del sistema.

(errno).

2.3. **Ejecución mediante procesos hijo:**

Cada comando ingresado por el usuario se ejecuta en un proceso hijo creado con la llamada al sistema `fork()`. Posteriormente, mediante `execvp()`, el hijo reemplaza su espacio de memoria con el programa solicitado.

El proceso padre, que actúa como intérprete, utiliza `wait()` o `waitpid()` para esperar la finalización del hijo antes de aceptar un nuevo comando, simulando el comportamiento de ejecución foreground típico de una shell.

2.4. **Redirección de salida estándar (>):**

Si el usuario agrega el operador `>` seguido del nombre de un archivo, la salida estándar (`stdout`) del proceso hijo se redirige al archivo indicado. Esto se logra mediante el uso de `dup2()` y `open()` antes de ejecutar el comando, permitiendo guardar resultados en archivos de texto.

2.5. **Comando de salida (salir):**

La shell finaliza su ejecución al ingresar el comando `salir`, cerrando todos los recursos abiertos y terminando el proceso padre de forma controlada.

2.6. **Comandos internos (built-ins):**

Además de los comandos externos, la mini-shell implementa varios comandos internos (built-ins), entre ellos:

- `cd`: Cambia el directorio de trabajo actual.
- `pwd`: Muestra la ruta actual.
- `help`: Presenta una lista con los comandos disponibles.
- `history`: Muestra el historial de comandos ejecutados en la sesión.
- `alias`: Permite crear alias simples de comandos.
- `parallel`: Ejecuta múltiples comandos en paralelo utilizando hilos (threads) de `pthread`.
- `meminfo`: Muestra el uso aproximado de memoria leyendo el archivo `/proc/self/status`.

2.7. **Ejecución en paralelo mediante hilos:**

Con el comando `parallel`, la mini-shell puede ejecutar varios comandos simultáneamente, separados por el delimitador `;;`.

Cada orden se ejecuta en un hilo independiente creado con `pthread_create()`, y el sistema espera la finalización de todos ellos mediante `pthread_join()`. Esta característica demuestra el manejo de concurrencia controlada y la sincronización entre hilos.

2.8. **Gestión de memoria y sincronización:**

El sistema utiliza estructuras estándar de C++ y sincroniza el acceso concurrente a recursos compartidos (como el historial y los alias) mediante mutexes, evitando condiciones de carrera.

Además, el comando `meminfo` ofrece una visión básica del consumo de memoria del proceso, mostrando valores como `VmSize` o `VmRSS`, tomados directamente del

sistema operativo.

2.9. Estructura modular y extensible:

El proyecto está dividido en múltiples archivos organizados en carpetas `src/` e `include/`, separando la lógica en módulos (`parser`, `executor`, `builtins`, `main`).

Esta modularidad facilita la lectura, mantenimiento y ampliación del código.

3. Arquitectura y diseño

La arquitectura del proyecto Mini-Shell se basa en un diseño modular, donde cada componente cumple una función específica dentro del flujo general de interpretación y ejecución de comandos. Esta estructura permite mantener un código limpio, escalable y fácil de depurar.

El sistema sigue una arquitectura cliente-controlador-ejecutor, donde el usuario (cliente) interactúa a través del prompt, el proceso principal actúa como intérprete o controlador, y los procesos hijo (o hilos) se encargan de la ejecución real de las órdenes.

3.1. Estructura General

La Mini-Shell está compuesta por los siguientes módulos:

- `main.cpp`
Es el punto de entrada del programa. Gestiona el ciclo principal de la shell: muestra el prompt, lee la línea de comandos, invoca al parser para analizarla y determina si se trata de un comando interno o externo.
También controla el comportamiento de espera (`waitpid`) y la captura de señales básicas (como `SIGINT`).
- `parser.cpp` / `parser.hpp`
Se encarga de dividir la línea de entrada del usuario en tokens separados por espacios. Esta fase es crucial porque define cómo se interpretan los argumentos, operadores de redirección (`>`) o tuberías (`|`), así como los separadores usados en comandos paralelos (`;;`).
- `executor.cpp` / `executor.hpp`
Contiene las funciones que crean procesos hijo mediante `fork()` y ejecutan los programas solicitados con `execvp()`.
Este módulo implementa también la redirección de salida estándar (`dup2 + open`) y la gestión de pipes simples entre comandos.
Se encarga de manejar errores de ejecución y sincronizar la espera del proceso hijo con el proceso padre.
- `builtins.cpp` / `builtins.hpp`
Contiene la implementación de los comandos internos (`cd`, `pwd`, `help`, `alias`, `history`, `meminfo`, `parallel`, etc.).
Algunos de estos comandos requieren sincronización de acceso a estructuras compartidas (como el historial), para lo cual se utiliza un `std::mutex`.

- Estructuras auxiliares
El proyecto utiliza estructuras como `std::vector` para almacenar tokens y `std::map` para registrar alias.
Estas estructuras facilitan la manipulación de datos dinámicos y la expansión del sistema en el futuro (por ejemplo, agregar historial persistente o autocompletado).

3.2. Flujo de Ejecución

El ciclo de ejecución de la mini-shell sigue los siguientes pasos:

- **Lectura y análisis del comando**
El proceso principal muestra el prompt (por ejemplo: `MiniShell>`) y espera la entrada del usuario.
Una vez ingresada la línea, se almacena en el historial y se envía al parser.
- **Tokenización y clasificación**
El parser separa la línea en tokens individuales y detecta si contiene operadores especiales como `>`, `|`, `;` o `&`.
- **Decisión de ejecución**
 - Si el comando es interno (built-in), se ejecuta directamente en el proceso principal.
 - Si es un comando externo, se llama a `fork()` para crear un proceso hijo que luego reemplaza su imagen con el ejecutable indicado mediante `execvp()`.
- **Redirección y manejo de archivos**
Si la línea contiene el operador `>`, se realiza una redirección de salida: el descriptor estándar `stdout` se cierra y se reemplaza por un descriptor de archivo mediante `dup2()`. De esta forma, la salida del comando no se muestra en pantalla sino que se guarda en un archivo.
- **Sincronización y espera**
El proceso padre espera al hijo con `wait()` o `waitpid()` (modo foreground).
En el caso de tareas paralelas (parallel), el proceso principal crea múltiples hilos de ejecución (pthread) y espera a que todos finalicen mediante `pthread_join()`.

3.3. Arquitectura de Procesos e Hilos

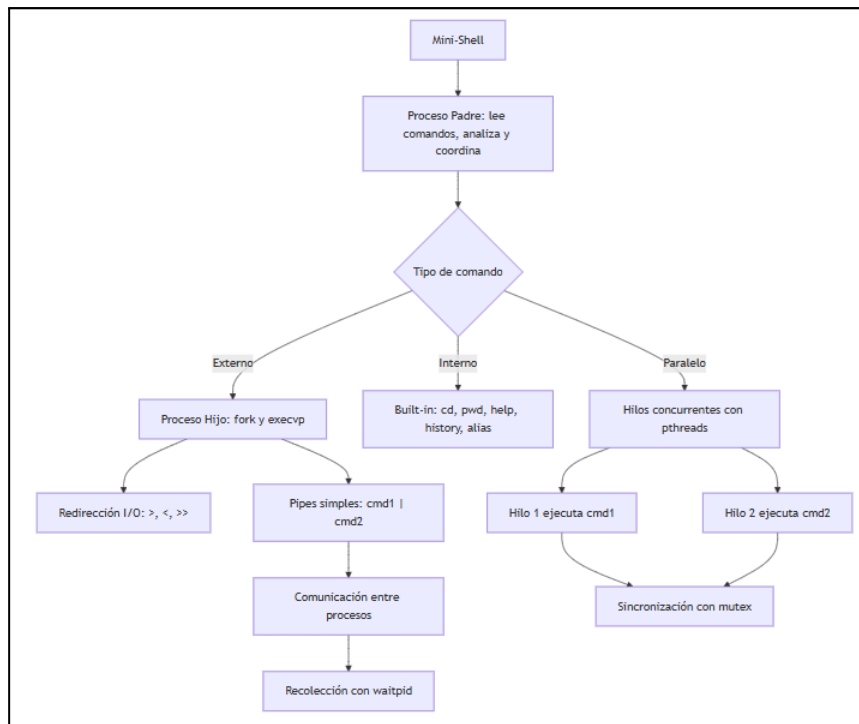


Figura 1. Representación de los procesos e hilos

4. Detalles de implementación (APIs POSIX usadas, decisiones clave)

La mini-shell fue desarrollada en C++17 utilizando las principales llamadas al sistema POSIX, que permiten manejar procesos, redirecciones, concurrencia y señales del sistema operativo Linux.

4.1. Principales APIs POSIX utilizadas

| Función | Descripción | Uso en la mini-shell |
|------------------|--|--|
| fork() | Crea un nuevo proceso hijo. | Usado para ejecutar comandos externos. |
| execvp() | Reemplaza el proceso actual por otro programa. | Ejecuta el binario solicitado (como /bin/lis). |
| waitpid() | Espera la finalización del proceso hijo. | Sincroniza la shell con los procesos foreground. |
| pipe() | Crea un canal de comunicación entre procesos. | Implementa los comandos con ` |
| dup2() | Duplica descriptores de archivo. | Redirige entradas/salidas en <, >, >>. |
| open() y close() | Abren y cierran archivos. | Manejan redirecciones de I/O. |

| | | |
|-----------------------------------|-------------------------------|--|
| pthread_create() / pthread_join() | Crea y sincroniza hilos. | Implementa el comando parallel. |
| signal() | Controla señales del sistema. | Ignora SIGINT en el proceso principal. |

4.2. Decisiones clave de diseño

- **Separación de responsabilidades:**

Se dividió el proyecto en módulos (parser, executor, builtins) para mantener un código limpio y modular.

- **Ejecución controlada:**

Solo el proceso hijo realiza redirecciones y llamadas a `execvp()`, asegurando que el proceso principal (la shell) mantenga su estado estable.

- **Concurrencia segura:**

Las operaciones compartidas (como historial y alias) se protegen con `std::mutex` para evitar condiciones de carrera.

- **Robustez ante errores:**

Todos los fallos del sistema (como ejecutables inexistentes o permisos denegados) se manejan con `perror()` y mensajes claros.

- **Simplicidad y compatibilidad:**

Se priorizó una arquitectura clara, centrada en cumplir los requisitos fundamentales sin recurrir a dependencias externas.

5. Concurrencia y sincronización:

La mini-shell ejecuta varios comandos al mismo tiempo mediante el comando `parallel`, que crea hilos (`pthread`) independientes para cada orden. Así se logra un paralelismo real, aprovechando los recursos del sistema.

Para evitar condiciones de carrera, se usa un `mutex` (`std::mutex`) que bloquea el acceso a datos compartidos (como historial o alias) cuando un hilo los modifica.

Además, el programa sincroniza los hilos con `pthread_join()` para esperar su finalización antes de continuar, y no usa múltiples bloqueos, evitando interbloqueos (deadlocks).

6. Gestión de memoria: estrategia y evidencias

La mini-shell sigue una estrategia basada en asignación y liberación controlada. Cada módulo del proyecto (lectura, parsing, ejecución) es responsable de manejar su propia memoria temporal, de manera que no existan dependencias innecesarias entre funciones.

- **Lectura de entrada**

La entrada del usuario se obtiene mediante funciones como `getline()` o `fgets()`, que asignan memoria dinámicamente para la cadena del comando. Una vez procesada la instrucción, la memoria es liberada inmediatamente con `free()` para evitar acumulación.

- **Tokenización y parsing**

Al separar el comando en argumentos, se utiliza una lista de punteros (`char **args`) generada con `malloc()` o `calloc()`. Estos arreglos permiten manipular los tokens de forma flexible y son liberados tras la ejecución del comando, ya sea interno o externo.

- **Ejecución de comandos**

En los comandos externos, el uso de `fork()` y `execvp()` crea un nuevo espacio de memoria aislado para el proceso hijo. Este proceso hereda únicamente la información esencial del padre, ejecuta el binario correspondiente y libera sus recursos automáticamente al finalizar.

El proceso padre, mientras tanto, utiliza `waitpid()` para sincronizar la finalización del hijo, asegurando que la memoria asociada al proceso hijo sea recuperada por el sistema operativo.

- **Comandos internos (built-in)**

Estos comandos se ejecutan dentro del proceso principal. En este caso, la mini-shell libera cualquier estructura temporal una vez completada la operación (por ejemplo, al cambiar de directorio o mostrar variables del entorno).

Evidencias de buenas prácticas

- Uso de punteros nulos después de liberar memoria (`ptr = NULL;`) para evitar accesos indebidos.
- Validación previa de punteros antes de liberar o reasignar memoria.
- Separación de responsabilidades: cada módulo libera la memoria que él mismo reserva.
- Evita duplicación de cadenas mediante referencias controladas cuando es posible.

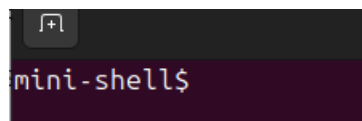
7. Pruebas y resultados:

El proceso de validación de la mini-shell se realizó mediante la ejecución práctica de casos de prueba diseñados para cubrir todas las especificaciones funcionales y extensiones solicitadas.

Cada caso fue probado en un entorno Linux (Ubuntu), utilizando tanto comandos del sistema como scripts propios, verificando que las salidas fueran correctas y los errores manejados adecuadamente.

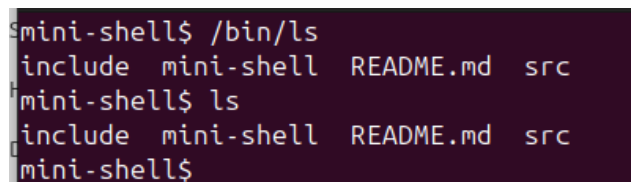
- **Prompt personalizado**

Objetivo: Mostrar un prompt propio que identifique la shell e indique disponibilidad para recibir comandos.

A screenshot of a terminal window with a dark background. The prompt 'mini-shell\$' is displayed in a light blue color. Above the prompt, there is a small icon of a terminal window with a plus sign.

- **Resolución de rutas**

Permite ejecutar programas tanto por su nombre como por su ruta absoluta.

A screenshot of a terminal window showing the execution of the 'ls' command. The prompt is 'mini-shell\$'. The first command is '/bin/ls', which outputs 'include mini-shell README.md src'. The second command is 'ls', which outputs the same result: 'include mini-shell README.md src'.

- **Ejecución mediante procesos (fork + exec)**

Crear un proceso hijo para ejecutar cada comando y sincronizar con el padre.


```
mini-shell$ date
Sun Oct 12 05:38:08 PM UTC 2025
mini-shell$ whoami
David
mini-shell$ ps
  PID TTY          TIME CMD
  3372 pts/0        00:00:00 bash
  3402 pts/0        00:00:00 mini-shell
  4411 pts/0        00:00:00 ps
mini-shell$
```

- Redirección de salida estándar (>)
Permitir redirigir la salida de un comando a un archivo.

```
mini-shell$ ls > salida.txt
mini-shell$ cat salida.txt
include
mini-shell
README.md
salida.txt
src
mini-shell$
```

- Comando de salida
Terminar la ejecución de la shell al ingresar salir.

```
mini-shell$ salir
David@David:~/Documents/mini-shell$
```

- Extensiones implementadas
- Pipes (|)
Permitir la conexión entre dos comandos mediante tubería.

```
mini-shell$ ls | grep .c
builtins.cpp
executor.cpp
main.cpp
parser.cpp
signals.cpp
mini-shell$
```

- Tareas en segundo plano (&)
Ejecutar comandos sin bloquear el prompt.

```
mini-shell$ sleep 2 &
[background pid 4527]
mini-shell$ ps
  PID TTY          TIME CMD
  4504 pts/0        00:00:00 bash
  4510 pts/0        00:00:00 mini-shell
  4518 pts/0        00:00:00 sleep
  4527 pts/0        00:00:00 sleep
  4528 pts/0        00:00:00 ps
[reaped pid 4527 exit 0]
```

- Redirección de entrada (<) y salida doble (>>)

```

mini-shell$ cat < entrada.txt
mini-shell$ echo "nuevo" >> salida.txt
mini-shell$ cat salida.txt
include
mini-shell
README.md
salida.txt
src
"nuevo"

```

- Comandos internos

| Comando | Descripción breve |
|----------|--------------------------------|
| salir | Termina la mini-shell |
| cd | Cambia de directorio |
| pwd | Muestra el directorio actual |
| help | Muestra la ayuda integrada |
| history | Lista los comandos ejecutados |
| alias | Crea o muestra alias |
| parallel | Ejecuta comandos en paralelo |
| meminfo | Muestra información de memoria |

8. Conclusiones:

A través de este proyecto, se pudo evidenciar la importancia de los procesos, los hilos y las llamadas al sistema POSIX en la ejecución de tareas y en la administración eficiente de los recursos del sistema.

Este trabajo permitió profundizar en el funcionamiento interno de un intérprete de comandos, entendiendo cómo se crean y controlan los procesos, cómo se gestionan las entradas y salidas, y cómo se mantiene la sincronización entre tareas concurrentes. Además, se demostró que una correcta gestión de memoria y la prevención de condiciones de carrera son esenciales para garantizar la estabilidad y confiabilidad de cualquier entorno de ejecución.

9. Anexos:

Comandos probados

Comandos Funcionales:

| Tipo | Comando / Operador | Descripción | Ejemplo de uso | Resultado esperado |
|---------|--------------------|--|----------------------------|--|
| Interno | cd | Cambia el directorio de trabajo actual. | cd /home | Cambia al directorio /home. |
| Interno | salir | Termina la ejecución de la MiniShell. | salir | Cierra la shell. |
| Interno | pwd | Muestra el directorio actual. | pwd | Imprime la ruta actual. |
| Interno | help | Muestra ayuda general con los comandos internos disponibles. | help | Lista de comandos con breve descripción. |
| Interno | history | Muestra el historial de comandos ejecutados. | history | Lista numerada de comandos anteriores. |
| Interno | alias | Crea alias para comandos o muestra los existentes. | alias ls="ls -l" | Define un alias para ls -l. |
| Interno | meminfo | Muestra información de memoria usada por la shell (malloc/free). | meminfo | Estadísticas de memoria dinámica. |
| Interno | parallel | Ejecuta varios comandos simultáneamente usando threads (pthread_create). | parallel ls;; pwd;;date | Ejecuta ls, pwd y date en paralelo. |
| Interno | clear | Limpia la consola. | clear | Pantalla limpia. |

Comandos Externos (usando execvp):

| Tipo de Comando | Comando | Descripción | Ejemplo de uso | Salida esperada |
|-----------------|---------|---|-----------------|---|
| Externo | ls | Lista los archivos del directorio actual. | ls | Muestra los nombres de archivos y carpetas. |
| Externo | pwd | Muestra el directorio de trabajo actual. | pwd | /home/usuario |
| Externo | cat | Muestra el contenido de un archivo. | cat archivo.txt | Imprime el contenido del archivo. |
| Externo | echo | Imprime texto o variables. | echo "Hola" | Hola |

| | | | | |
|---------|--------|---|-------------------------|--------------------------------------|
| Externo | grep | Busca texto dentro de archivos. | grep "hola" archivo.txt | Muestra líneas que contienen "hola". |
| Externo | ps | Muestra los procesos activos del sistema. | ps | Lista de procesos en ejecución. |
| Externo | sleep | Detiene la ejecución por unos segundos. | sleep 3 | Pausa durante 3 segundos. |
| Externo | date | Muestra la fecha y hora actual. | date | Ejemplo: Sat Oct 11 10:05:00 2025 |
| Externo | whoami | Muestra el usuario actual. | whoami | Ejemplo: david |