

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



INFORME TÉCNICO:

“Intérprete de comandos en C++ para Linux”

Docente: Hugo Manuel Barraza Vizcarra

Integrantes:

Alex Yasmani Huaracha Bellido 2023 - 119027

Jheral Jhosue Maquera Laque 2023 - 119037

Ciclo académico : 6 ciclo

Asignatura: Sistemas Operativos

TACNA - PERÚ

2023

1. OBJETIVOS Y ALCANCE

1.1. Objetivo general

El objetivo del proyecto es diseñar e implementar un intérprete de comandos tipo Mini Shell utilizando el lenguaje C + +, que permita ejecutar programas del sistema operativo Linux mediante la gestión de procesos, redirecciones, tuberías y concurrencia con hilos.

1.2. Objetivos específicos

- Comprender el funcionamiento interno de un shell y su interacción con el sistema operativo.
- Aplicar los conceptos de procesos, llamadas al sistema POSIX y hilos.
- Implementar mecanismos de entrada y salida estándar como redirección y tuberías.
- Permitir la ejecución paralela de comandos empleando hilos POSIX.
- Fomentar buenas prácticas de programación modular y manejo de errores.

1.3. Alcance

a. Prompt Personalizado

El intérprete muestra un prompt propio y gestiona la lectura de la línea de comando..

b. Resolución de Rutas

Se maneja la ejecución directa de rutas absolutas (ej. /usr/bin/ls). Para comandos sin ruta, se asume el directorio predefinido /bin (ej. ls → /bin/ls si existe).

c. Ejecución de Procesos

La invocación de programas externos se realiza mediante fork() en el proceso padre, seguido de exec*() en el proceso hijo, asegurando que el padre espere al hijo con wait()/waitpid() (comportamiento foreground por defecto).

d. Manejo de Errores

El sistema incluye manejo de errores para notificar al usuario cuando un comando no exista, la ruta no sea válida, o si las llamadas a exec fallan, utilizando mensajes claros y errno/perror cuando sea aplicable.

e. Redirección de Salida (>)

Se soporta la sintaxis `nombrePrograma arg2 > archivo`. Se utiliza `dup2()` para redirigir el `stdout` del hijo al archivo especificado, el cual es creado o truncado.

f. Comando de Salida

El intérprete finaliza su ejecución de manera controlada al ingresar la palabra interna `salir`.

2. ARQUITECTURA Y DISEÑO

2.1. Descripción general

La mini shell Linux se diseñó con una arquitectura modular, compuesta por funciones separadas para el análisis, ejecución y control de los comandos. El programa principal (`main`) actúa como intérprete central, que lee las instrucciones del usuario y decide el flujo de ejecución según el tipo de comando ingresado.

La estructura principal del programa es el siguiente:

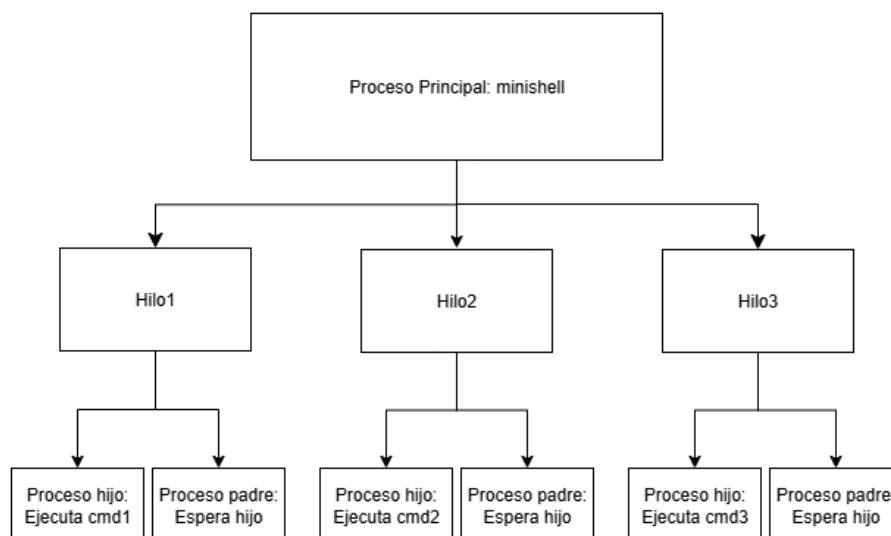
Carpeta / Archivo	Descripción
<code>docs/</code>	Carpeta destinada a la documentación e informes del proyecto.
<code>docs/documentos/</code>	Contiene archivos complementarios del informe o evidencias de pruebas.
<code>include/</code>	Directorio de encabezados (<code>.h</code>) con las declaraciones de funciones y estructuras compartidas entre los módulos.
<code>include/ejecutor.h</code>	Archivo principal de cabecera que declara las funciones de ejecución de comandos.
<code>src/</code>	Carpeta que contiene todo el código fuente de la mini-shell.
<code>src/main.cpp</code>	Punto de entrada del programa. Lee los comandos y controla el flujo general de la shell.

src/ejecutador.cpp	Implementa la creación de procesos mediante fork() y ejecución con execv().
src/paralelo.cpp	Implementa el comando paralelo, utilizando hilos (pthread_create) para ejecutar múltiples procesos concurrentemente.
src/redireccion.cpp	Implementa la redirección de salida estándar (>), usando open(), dup2() y close().
src/rutas.cpp	Resuelve las rutas absolutas o relativas para ejecutar comandos válidos.
src/tuberia.cpp	Implementa el pipe simple (`cmd1
README.md	Documento informativo general sobre el uso, compilación y características del proyecto.

2.2. Diagrama de procesos e hilos

El siguiente diagrama representa la estructura de procesos e hilos de la mini-shell durante la ejecución de comandos normales y en modo paralelo.

En el caso del comando paralelo, el proceso principal crea varios hilos, y cada hilo lanza su propio proceso hijo mediante fork() para ejecutar un comando independiente.



2.3. Manejo de Entrada/Salida (I/O)

La mini-shell gestiona la entrada y salida estándar a través de los mecanismos de redirección (>) y tuberías (|), haciendo uso de las funciones del sistema POSIX. Estos mecanismos permiten controlar el flujo de datos entre procesos o hacia archivos de manera eficiente y controlada.

La redirección de salida (>) posibilita enviar la salida de un comando a un archivo en lugar de mostrarla en pantalla. Para ello, se emplea la función `open()` para crear o abrir el archivo de destino y posteriormente `dup2()` para redirigir la salida estándar (`STDOUT_FILENO`) hacia dicho archivo. Por ejemplo, al ejecutar `ls > salida.txt`, el resultado del comando `ls` se almacena en el archivo *salida.txt* en lugar de mostrarse por consola.

De manera similar, las tuberías (|) permiten conectar la salida de un comando con la entrada del siguiente, facilitando la comunicación entre procesos. Esto se implementa mediante las funciones `pipe()`, `fork()` y `dup2()`, de modo que la salida estándar del primer proceso se convierte en la entrada estándar del segundo. Por ejemplo, el comando `ls | grep cpp` muestra únicamente los archivos que contienen la palabra “cpp”.

En conjunto, estas operaciones de entrada y salida garantizan un manejo flexible del flujo de datos dentro de la mini-shell, permitiendo tanto la conexión entre procesos como la escritura directa en archivos, sin afectar la ejecución del intérprete principal.

3. DETALLES DE IMPLEMENTACIÓN

3.1. APIs POSIX utilizadas

El shell utiliza llamadas POSIX para interactuar directamente con el sistema operativo.

- Creación de procesos: `fork()` y `waitpid()` controlan la ejecución de comandos externos.
- Ejecución de programas: `execvp()` reemplaza el proceso hijo por el programa indicado.
- Redirección: `dup2()` y `open()` permiten redirigir salidas hacia archivos.
- Tuberías: `pipe()` conecta comandos para transmitir datos entre ellos.
- Concurrencia: `pthread_create()` y `pthread_join()` permiten la ejecución paralela.
- Control de errores: `stat()` y `perm()` detectan errores en rutas o permisos.

El manejo modular facilita extender el sistema con nuevas funciones, como historial o ejecución en segundo plano.

3.2. Decisiones clave

Durante el desarrollo del Mini Shell, se tomaron varias decisiones de diseño con el objetivo de equilibrar la simplicidad, la eficiencia y la estabilidad del sistema:

a) Uso de hilos para la ejecución paralela

Se decidió implementar el comando paralelo utilizando hilos (threads) en lugar de procesos separados, debido a su menor costo de creación y conmutación de contexto.

Los hilos permiten compartir el mismo espacio de memoria y recursos del proceso principal, reduciendo la sobrecarga del sistema y permitiendo ejecutar múltiples comandos de forma más fluida.

b) Limitación del uso de tuberías

Se implementó un modelo de tuberías simples (una sola "|") en lugar de tuberías anidadas.

Esta decisión se basó en criterios de claridad y control de errores, evitando la complejidad de manejar múltiples procesos encadenados y flujos de datos simultáneos.

c) Manejo explícito de errores

Cada módulo verifica el estado de las operaciones críticas (como apertura de archivos, permisos o ejecución de procesos). En caso de error, se genera un mensaje descriptivo utilizando perror() junto con información adicional (errno y strerror()), lo que facilita la depuración.

d) Elección de una arquitectura modular

Se optó por separar las funcionalidades en distintos archivos (rutas.cpp, tuberia.cpp, paralelo.cpp, etc.), lo que mejora la legibilidad, permite pruebas unitarias independientes y favorece el mantenimiento.

e) Control del flujo principal

El bucle principal fue diseñado para mantener una estructura simple: leer, interpretar, ejecutar y esperar la finalización del comando. Esta decisión favorece la claridad lógica, manteniendo el comportamiento similar al de un shell real sin añadir complejidad excesiva.

4. CONCURRENCIA Y SINCRONIZACIÓN

4.1. Qué se paraleliza

El comando paralelo permite ejecutar varios comandos de forma simultánea.

Cuando el usuario ingresa algo como:

paralelo ls ; pwd ; whoami

La shell crea un hilo (pthread) por cada comando, y dentro de cada hilo se ejecuta un proceso hijo mediante fork() y execv(). Así, todos los comandos se ejecutan al mismo tiempo, aprovechando la concurrencia de los hilos.

4.2. Cómo se evita la condición de carrera

- Cada hilo trabaja con una copia independiente de su cadena de comando (string comando), por lo que no hay variables compartidas modificadas al mismo tiempo.
- Los hilos no comparten estructuras de datos globales ni escriben en los mismos archivos o buffers simultáneamente.
- Se utiliza pthread_join() al final para sincronizar y asegurar que todos los hilos terminen antes de continuar la ejecución del programa principal.

De esta forma, no hay riesgo de condiciones de carrera en el manejo de comandos.

4.3. Evitación de interbloqueos

En la implementación no se presentan interbloqueos, ya que no existen recursos compartidos bloqueantes entre los hilos. Cada hilo creado mediante pthread_create() se limita a ejecutar un comando distinto, y dentro de cada hilo se genera un proceso hijo independiente mediante fork(). Estos procesos hijos ejecutan sus comandos de manera completamente aislada, sin compartir memoria ni archivos entre sí, lo que elimina cualquier dependencia entre ellos.

Además, los hilos no compiten por recursos comunes ni requieren mecanismos de sincronización como *mutexes* o semáforos, ya que cada uno opera sobre datos locales. La única sincronización ocurre al final del proceso, cuando el hilo principal utiliza pthread_join() para esperar la finalización ordenada de todos los hilos antes de continuar. Por esta razón, no existe posibilidad de interbloqueo, puesto que no hay dependencias circulares ni recursos en espera entre hilos o procesos.

5. PRUEBAS Y RESULTADOS

5.1. Casos de prueba principales

La fase de pruebas se centró en validar el correcto funcionamiento de las características implementadas del Mini Shell.

Se diseñaron casos de prueba para evaluar la ejecución de comandos, la redirección de salida, las tuberías, la concurrencia mediante hilos y el manejo de errores.

Todas las pruebas se realizaron en un entorno Linux Ubuntu 22.04 con compilador `g++` y soporte POSIX.

Comando	Descripción de la prueba	Resultado esperado	Resultado obtenido
<code>ls</code>	Ejecutar un comando simple del sistema.	Lista los archivos del directorio actual.	Correcto: muestra el contenido del directorio.
<code>pwd</code>	Verificar ejecución de un comando interno de ubicación.	Muestra la ruta del directorio actual.	Correcto: imprime la ruta completa.
<code>ls > salida.txt</code>	Probar redirección de salida hacia un archivo.	Crea o reemplaza <code>salida.txt</code> con el resultado de <code>ls</code> .	Correcto: se genera el archivo con la salida esperada.
<code>`ls`</code>	<code>grep .cpp`</code>	Validar funcionamiento de la tubería.	Muestra solo archivos con extensión <code>.cpp</code> .
<code>paralelo date; whoami; pwd</code>	Ejecutar varios comandos en paralelo con hilos.	Los tres comandos se ejecutan de manera simultánea.	Correcto: las salidas se intercalan, confirmando concurrencia.
<code>/fake/path</code>	Verificar manejo de errores en rutas inexistentes.	Muestra el mensaje "ruta no encontrada" sin detener el shell.	Correcto: se muestra el error con <code>perror()</code> .
<code>paralelo</code>	Comando <code>paralelo</code> sin argumentos.	Debe mostrar un mensaje de error de uso.	Correcto: muestra "no se proporcionó ningún comando".
<code>echo Hola > resultado.txt</code>	Redirección de salida con texto literal.	Crea archivo con la palabra "Hola".	Correcto: contenido verificado.

5.2. Validación de requisitos

Requisito	Módulo	Validación realizada	Estado
Ejecución de comandos externos.	ejecutador.cpp	Se probaron comandos como ls, pwd, cat.	Cumplido
Redirección de salida (>).	redireccion.cpp	Se redirigieron salidas a archivos (ls > salida.txt).	Cumplido
Manejo de tuberías ().	tuberia.cpp	Se conectaron procesos ('ls wc')	Cumplido
Ejecución paralela con hilos.	paralelo.cpp	Se ejecutaron varios comandos simultáneamente (paralelo date; pwd; whoami).	Cumplido
Manejo de errores y validaciones.	Todos los módulos	Se probaron rutas inválidas, permisos y comandos vacíos.	Cumplido
Finalización del shell.	main.cpp	Se validó el comando salir para finalizar el ciclo principal.	Cumplido

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Conclusiones

El desarrollo del Mini Shell en C++ permitió comprender en profundidad cómo el sistema operativo Linux gestiona los procesos, la entrada/salida y la concurrencia.

A través del uso de llamadas al sistema POSIX como fork(), execvp(), pipe(), dup2() y las librerías de hilos (pthread), se logró implementar un entorno funcional que reproduce el comportamiento básico de un shell real.

El proyecto también demostró la importancia de un diseño modular y estructurado, que facilita el mantenimiento, la comprensión y la escalabilidad del código.

En términos generales, se cumplieron los objetivos planteados, consolidando habilidades tanto teóricas como prácticas en programación de sistemas operativos.

6.2. Trabajos futuros

Si bien el Mini Shell cumple con las funcionalidades básicas de ejecución de comandos, redirección, tuberías y concurrencia, existen múltiples áreas que podrían desarrollarse en futuras versiones para ampliar su alcance y realismo. Entre ellas destacan:

- **Historial de comandos:** Implementar un registro que permita al usuario volver a ejecutar comandos anteriores, mejorando la usabilidad.
- **Soporte para múltiples tuberías:** Extender la implementación actual para admitir más de una conexión (cmd1 | cmd2 | cmd3 | ...), simulando el comportamiento de shells avanzados.
- **Implementación del comando cd:** Agregar soporte para cambiar el directorio de trabajo dentro del mismo proceso, lo cual requiere manejar rutas relativas y absolutas.
- **Ejecución en segundo plano (&):** Permitir que los comandos se ejecuten sin bloquear el shell principal, gestionando procesos activos de manera asíncrona.
- **Gestión de señales:** Incorporar el manejo de interrupciones (por ejemplo, Ctrl+C), permitiendo pausar o finalizar procesos controladamente.
- **Optimización del manejo de errores:** Ampliar la detección de errores de sintaxis y mejorar la retroalimentación al usuario mediante mensajes más descriptivos.

Estas mejoras permitirían que el Mini Shell evolucione hacia un entorno más robusto, interactivo y cercano a las funcionalidades de un intérprete de comandos real

7. Anexos

7.1. Comandos probados

Ejemplos de los comandos ejecutados para validar el funcionamiento de la mini-shell.

```
# Ejecución de comandos básicos

ls -l
pwd
echo "Hola Mundo"
```

```
# Ejemplo con ruta absoluta

/bin/ls -l /home/alex/proyecto

# Prueba de redirección

ls -l > salida.txt
cat < entrada.txt

# Prueba de pipe

ls -l | grep cpp
cat archivo.txt | wc -l

# Prueba de concurrencia

parallel ls pwd date whoami
parallel "ls -l" "echo Hola" "uname -r"
```

7.2. Fragmentos de código relevantes

- Implementación de pipes

```
int fd[2];
pipe(fd);
pid_t pid1 = fork();

if (pid1 == 0) {
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]);
    close(fd[1]);
    execvp(cmd1[0], cmd1);
} else {
    pid_t pid2 = fork();
    if (pid2 == 0) {
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execvp(cmd2[0], cmd2);
    }
}
```

```

        close(fd[0]);
        close(fd[1]);
        waitpid(pid1, nullptr, 0);
        waitpid(pid2, nullptr, 0);
    }

```

- **Implementación de concurrencia con hilos**

```

void* ejecutar_comando(void* arg) {
    char** comando = (char**)arg;
    execvp(comando[0], comando);
    pthread_exit(nullptr);
}

for (int i = 0; i < n_comandos; ++i) {
    pthread_create(&hilos[i], nullptr,
        ejecutar_comando, (void*)comandos[i]);
}

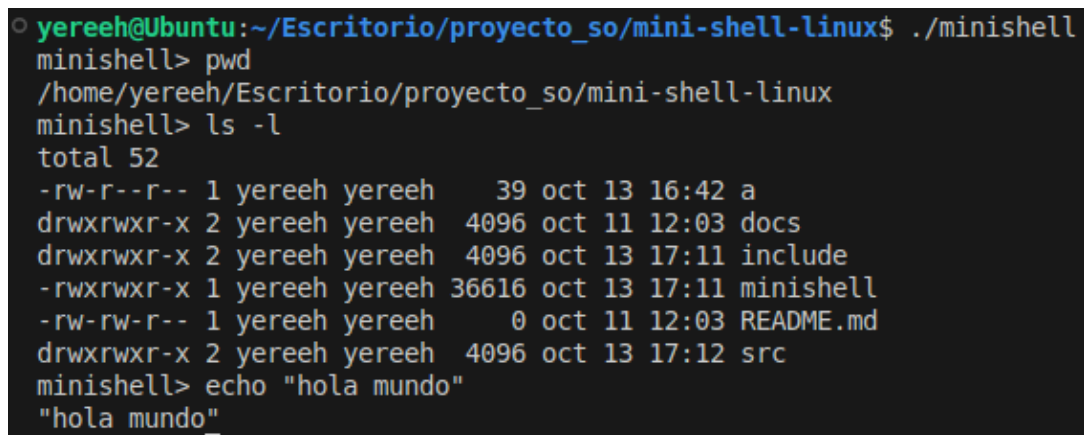
for (int i = 0; i < n_comandos; ++i) {
    pthread_join(hilos[i], nullptr);
}

```

7.3. Capturas de pantalla

Figura 1

Ejecución de comandos básicos en la mini-shell.



```

yereeh@Ubuntu:~/Escritorio/proyecto_so/mini-shell-linux$ ./minishell
minishell> pwd
/home/yereeh/Escritorio/proyecto_so/mini-shell-linux
minishell> ls -l
total 52
-rw-r--r-- 1 yereeh yereeh   39 oct 13 16:42 a
drwxrwxr-x 2 yereeh yereeh 4096 oct 11 12:03 docs
drwxrwxr-x 2 yereeh yereeh 4096 oct 13 17:11 include
-rwxrwxr-x 1 yereeh yereeh 36616 oct 13 17:11 minishell
-rw-rw-r-- 1 yereeh yereeh    0 oct 11 12:03 README.md
drwxrwxr-x 2 yereeh yereeh 4096 oct 13 17:12 src
minishell> echo "hola mundo"
"hola mundo"

```

Nota. Se observa la correcta ejecución de los comandos ls, pwd y echo dentro del intérprete. *Elaboración propia.*

Figura 2

Ejecución de un comando utilizando una ruta absoluta.

```
minishell> /bin/ls -l /home/yereeh
total 56
drwxr-xr-x 2 yereeh yereeh 4096 oct 11 12:00 Descargas
drwxr-xr-x 2 yereeh yereeh 4096 sep 17 01:15 Documentos
drwxr-xr-x 4 yereeh yereeh 4096 oct 11 16:39 Escritorio
-rwxrwxr-x 1 yereeh yereeh 16496 sep 17 09:54 hilos
drwxr-xr-x 3 yereeh yereeh 4096 sep 24 10:18 Imágenes
drwxr-xr-x 2 yereeh yereeh 4096 sep 17 01:15 Música
drwxr-xr-x 2 yereeh yereeh 4096 sep 17 01:15 Plantillas
drwxr-xr-x 2 yereeh yereeh 4096 sep 17 01:15 Público
drwx----- 6 yereeh yereeh 4096 sep 23 21:00 snap
drwxr-xr-x 2 yereeh yereeh 4096 sep 17 01:15 Vídeos
```

Nota. Se demuestra la ejecución correcta del binario /bin/ls sobre un directorio específico. Elaboración propia.

Figura 3

Redirección de entrada y salida estándar.

```
minishell> ls -l > salida.txt
≡ salida.txt
1 total 52
2 -rw-r--r-- 1 yereeh yereeh 39 oct 13 16:42 a
3 drwxrwxr-x 2 yereeh yereeh 4096 oct 11 12:03 docs
4 drwxrwxr-x 2 yereeh yereeh 4096 oct 13 17:11 include
5 -rwxrwxr-x 1 yereeh yereeh 36616 oct 13 17:11 minishell
6 -rw-rw-r-- 1 yereeh yereeh 0 oct 11 12:03 README.md
7 -rw-r--r-- 1 yereeh yereeh 0 oct 13 22:26 salida.txt
8 drwxrwxr-x 2 yereeh yereeh 4096 oct 13 17:12 src
```

Nota. Se muestra cómo la salida del comando ls se guarda en un archivo (> salida.txt). Elaboración propia.

Figura 4

Uso de tuberías (pipes) entre procesos.

```
minishell> ls /home/yereeh/Escritorio/proyecto_so/mini-shell-linux/src | grep cpp
ejecutador.cpp
main.cpp
paralelo.cpp
redireccion.cpp
rutas.cpp
tuberia.cpp
```

Nota. El ejemplo ls -l | grep cpp conecta la salida de un comando con la entrada de otro. Elaboración propia.

Figura 5

Ejecución concurrente de comandos con el built-in parallel.

```
minishell> parallel ls ; pwd ; date ; whoami  
a docs include minishell README.md salida.txt src  
/home/yereeh/Escritorio/proyecto_so/mini-shell-linux  
yereeh  
lun 13 oct 2025 22:33:22 -05  
Todos los comandos en paralelo han finalizado.
```

Nota. Se aprecia la ejecución simultánea de varios comandos mediante hilos POSIX. Elaboración propia.

Figura 6

Manejo de error por comando inexistente.

```
minishell> prueba  
Error: no se encontró el comando 'prueba': No such file or directory  
(errno 2: No such file or directory)
```

Nota. La mini-shell detecta correctamente comandos no válidos y muestra un mensaje de error. Elaboración propia.

7.4. Instrucciones de compilación y ejecución

Incluye cómo compilar y ejecutar tu programa, así el profesor puede probarlo:

```
# Compilación  
  
g++ src/*.cpp -Iinclude -o minishell  
  
# Ejecución  
  
./minishell
```