

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



“Proyecto: Mini Shell POSIX en C++”

ASIGNATURA:

Sistemas Operativos

DOCENTE:

Hugo Manuel Barraza Vizcarra

INTEGRANTES:

Edgar Jampier Leyva Garcia 2023-119028

Henry Mark Maquera Mamani 2023-119038

TACNA - PERÚ

2025

Objetivo

Desarrollar un intérprete de comandos funcional (mini-shell) en C++ sobre un entorno POSIX (Linux), aplicando de manera práctica los conceptos fundamentales de la gestión de procesos, la comunicación entre procesos y el manejo de la entrada/salida a bajo nivel para demostrar el dominio de las primitivas de un sistema operativo.

Alcance

Ejecución de Comandos Externos: La mini-shell será capaz de ejecutar cualquier comando externo disponible en el PATH del sistema (ej. ls, grep, cat, gcc).

Tuberías (Pipes) Simples: Se implementará el encadenamiento de dos o más comandos a través de tuberías (|), permitiendo que la salida estándar de un proceso se convierta en la entrada estándar del siguiente.

Redirección de Entrada/Salida: La shell soportará la redirección de la salida estándar a un archivo, tanto en modo de truncamiento (>) como de adición (>>), y la redirección de la entrada estándar desde un archivo (<).

Manejo de Errores Básico: El sistema proveerá retroalimentación clara al usuario cuando un comando no pueda ser encontrado o cuando una llamada al sistema falle, utilizando perror.

Arquitectura y diseño

Imagen 1

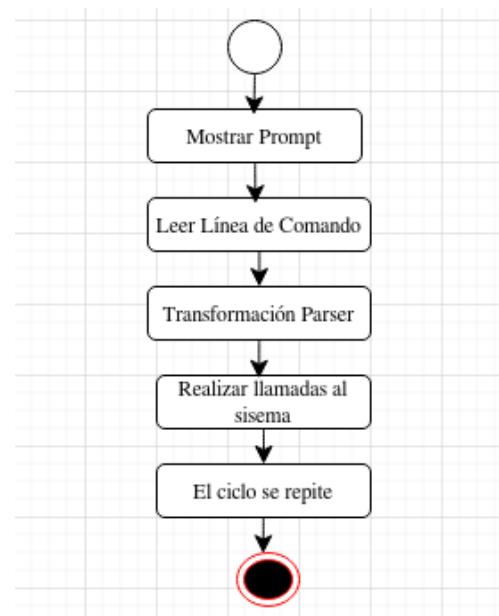


Imagen 2

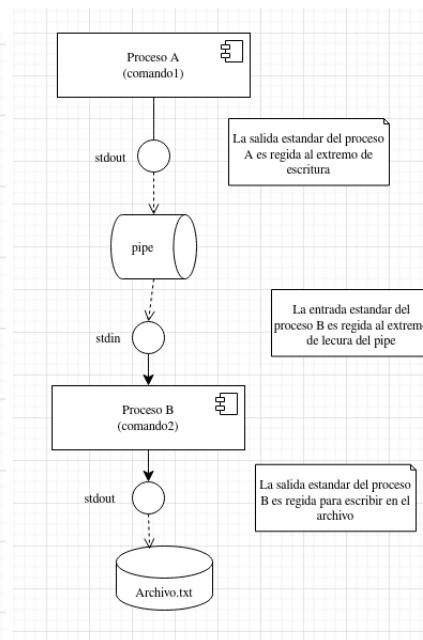
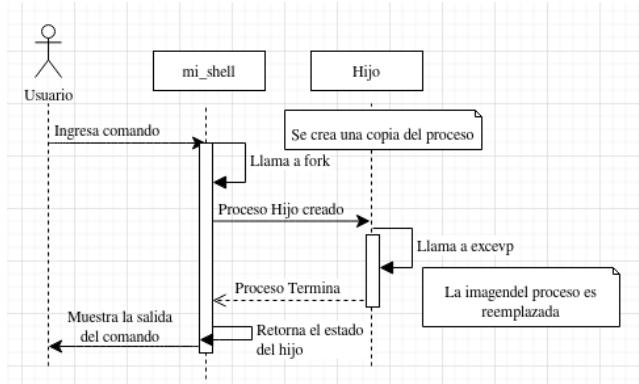


Imagen 3



Detalles de implementación

En esta sección detallaremos las llamadas al sistema APIs del estándar POSIX que es el núcleo fundamental.

APIs POSIX

La funcionalidad de la mini-shell se basa en un conjunto de potentes llamadas al sistema POSIX que permiten una interacción a bajo nivel con el kernel de Linux.

fork(): Crear un nuevo proceso (proceso hijo) que es una copia exacta del proceso que lo llama (proceso padre).

execvp(): Reemplazar la imagen del proceso actual con un nuevo programa.

waitpid(): Suspender la ejecución del proceso actual hasta que un proceso hijo específico cambie de estado (generalmente, hasta que termine).

pipe(): Crear una tubería anónima, un canal de comunicación unidireccional que proporciona dos descriptores de archivo: uno para leer (pipefd[0]) y otro para escribir (pipefd[1]).

dup2(): Duplicar un descriptor de archivo, haciendo que un descriptor apunte a la misma entrada de la tabla de archivos que otro.

open() / close(): open() abre o crea un archivo y devuelve un descriptor de archivo para su uso. close() libera un descriptor de archivo.

chdir() / getcwd(): chdir() cambia el directorio de trabajo actual del proceso. getcwd() obtiene la ruta del directorio de trabajo actual.

Concurrencia y sincronización

Se basa en procesos gestionados por el kernel, mientras que el paralelismo y la sincronización se logra mediante el uso de las APIs de POSIX.

¿Qué se Paralleliza?

Comandos Externos: Cada comando se ejecuta en un proceso hijo (fork()) que corre en paralelo a la shell.

Tareas en Segundo Plano (&): El padre lanza un proceso hijo y continúa su ejecución sin esperar (waitpid()), permitiendo paralelismo real entre la shell y múltiples comandos.

Tuberías (|): Múltiples procesos hijos (uno por cada comando en la cadena) se ejecutan de forma concurrente, sincronizados automáticamente por el kernel a través de los búferes de los pipes. Un proceso se bloquea si intenta leer de un pipe vacío o escribir en uno lleno.

¿Cómo se Evitan Problemas de Concurrencia?

Prevención de Condiciones de Carrera (Race Conditions):

Aislamiento de Memoria: La causa principal se elimina porque cada proceso tiene su propio espacio de memoria. No hay datos compartidos (variables, heap) entre la shell y los comandos que ejecuta, por lo que no pueden interferir entre sí.

Ejecución Secuencial de Built-ins: Comandos que modifican el estado de la shell (como cd) se ejecutan directamente en el proceso padre, de forma secuencial y sin crear concurrencia, garantizando la integridad del estado.

Gestión de memoria

La estrategia adoptada en este proyecto se centra en minimizar la gestión manual de memoria y aprovechar las abstracciones de alto nivel de C++ para prevenir errores comunes como fugas de memoria y punteros colgantes.

Estrategias de gestión de memoria

Preferencia por la Memoria en la Pila (Stack):

Todos los objetos principales, como la struct Command y las variables locales dentro de las funciones (pid_t, contadores, etc.), se declaran en la pila. La memoria de la pila es gestionada automáticamente por el compilador: se reserva al entrar en un ámbito (una función) y se libera automáticamente al salir. Esta es la forma más segura y eficiente de gestionar la memoria, ya que elimina por completo la posibilidad de fugas para estos objetos.

Gestión Manual en Puntos Críticos:

El único punto del código donde se requiere explícitamente memoria dinámica con new es en la función de utilidad convert_to_c_array. Esto es un requisito inevitable de la API de execvp, que espera un array de punteros a char terminado en NULL, un formato de C.

Sin embargo, esta asignación de memoria se realiza dentro del proceso hijo, justo antes de la llamada a execvp. Cuando execvp tiene éxito, reemplaza por completo el espacio de memoria del proceso hijo con el del nuevo programa. Esto significa que toda la memoria del proceso hijo, incluyendo cualquier bloque reservado en el montículo que no se haya liberado, es reclamada por el sistema operativo. Por lo tanto, aunque técnicamente no hay una llamada a delete[], esta asignación no produce una fuga de memoria persistente en el sistema.

Pruebas y Resultados

ID	Característica	Caso de Prueba (Comando a Ingresar)	Resultado Esperado	Método de Validation
01	Prompt y Lectura	Iniciar el programa: ./mi_shell	La shell muestra el prompt mi_shell> y el cursor espera la entrada del usuario.	Inspección visual directa en la terminal.
02	Ejecución Simple	ls	La shell muestra una lista de los archivos y directorios en la carpeta actual.	Comparar la salida con el resultado de ejecutar ls en una shell estándar (ej. bash).
03	Ejecución con Argumentos	echo "Sistemas Operativos"	La shell imprime la cadena "Sistemas Operativos" en la siguiente línea y vuelve a mostrar el prompt.	Inspección visual. El texto mostrado debe coincidir exactamente con el argumento.
04	Manejo de Errores	comandoquenoexiste	La shell muestra un mensaje de error claro, como Error en execvp(): No such file or directory, y vuelve a mostrar el prompt sin cerrarse.	Verificación de que la shell no termina y que el mensaje de error es informativo.
05	Redirección de Salida (>)	ls -l > listado.txt	No se muestra ninguna salida en la terminal. Se crea (o sobrescribe) un archivo listado.txt en el directorio actual.	1. Verificar la ausencia de salida. 2. Comprobar la existencia del archivo listado.txt con ls. 3. Verificar su contenido con cat listado.txt.
06	Comando de Salida	salir	El proceso de la mini-shell termina y el control regresa a la terminal del sistema operativo.	El prompt mi_shell> desaparece y es reemplazado por el prompt de la terminal original.

Conclusiones

Se logró aplicar con éxito los conceptos teóricos de la gestión de procesos en un sistema POSIX funcional. El proyecto cumplió su objetivo principal al demostrar el dominio práctico del ciclo de vida de los procesos mediante el uso correcto y coordinado de las llamadas al sistema fork(), execvp() y waitpid(). La implementación de la mini-shell sirvió como un puente tangible entre la teoría de los sistemas operativos y su aplicación real a bajo nivel, validando la capacidad del equipo para controlar la creación, ejecución y sincronización de procesos en un entorno Linux.

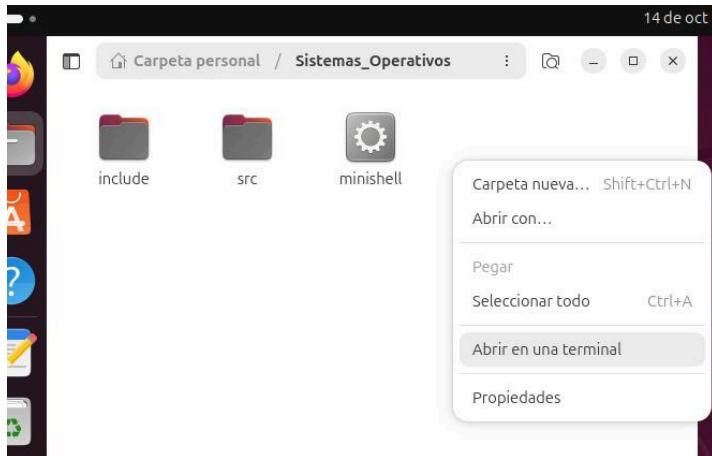
La adopción de una arquitectura modular y el uso de abstracciones de C++ fueron decisiones clave para la robustez y seguridad del proyecto. La separación de la lógica en un módulo Parser y un Executor simplificó el desarrollo y la depuración. Más importante aún, la estrategia de gestión de memoria, basada en el uso de contenedores de la STL (vector, string) y la preferencia por la memoria de pila, eliminó eficazmente la necesidad de gestión manual con new/delete o malloc/free en el proceso principal, previniendo fugas de memoria y demostrando un enfoque de ingeniería moderno y seguro.

Anexos

Sistema Operativo Ubuntu Linux

Descripción: Primero ingresamos a la carpeta donde se encuentra nuestro proyecto y abrimos el terminal con el click derecho (como se muestra en la imagen).

Anexo I



Una vez compilado se vera asi en el Ubuntu Terminal:

Anexo II

```
toni@toni-VirtualBox:~/Sistemas_Operativos$ ./minishell
mi_shell>> 
```

Se hace un prueba de comandos simples:

Se usa el comando ls (Lista los archivos del proyecto).

Anexo III

```
mi_shell>> ls
include minishell src
```

Se usa el comando pwd (Muestra la ruta de la carpeta)

Anexo IV

IV

```
mi_shell>> pwd
/home/toni/Sistemas_Operativos
```

Se usa el comando un comando más avanzado ls -l (muestra la lista de archivos en formato largo).

Anexo V

```
mi_shell>> ls -l
total 80
drwxrwxrwx 2 toni toni 4096 oct 14 16:33 include
-rwxrwxr-x 1 toni toni 70184 oct 14 16:35 minishell
drwxrwxrwx 2 toni toni 4096 oct 14 16:34 src
```

Se usa el comando echo “ ” (imprime una frase en la pantalla , como se verá en la imagen siguiente).

Anexo VI

```
mi_shell>> echo "hola,mi shell funciona?"  
"hola,mi shell funciona?"  
mi_shell>>
```

Usando este comando cd src (no se mostrará nada pero cambiaremos a otro directorio)

Anexo VII.

```
[+]  
toni@toni-VirtualBox:~/Sistemas_Operativos  
toni@toni-VirtualBox:~/Sistemas_Operativos$ ./minishell  
mi_shell>> cd src
```

Siguiendo con el comando pwd (ahora podre ver que estamos dentro de la carpeta src).

Anexo VIII

```
mi_shell>> pwd  
/home/toni/Sistemas_Operativos/src
```

Para poder volver a la carpeta principal se usará cd .. y para confirmar que he vuelto se usará pwd .

Anexo IX

```
mi_shell>> cd ..  
mi_shell>> pwd  
/home/toni/Sistemas_Operativos
```

Prueba de Redirección de salida se usará el comando ls-l > mis_archivos.txt y con ls se podrá visualizar el nuevo archivo en la terminal.

Anexo**X**

```
mi_shell>> ls -l > mis_archivos.txt  
mi_shell>> ls  
include minishell mis_archivos.txt src
```

Mostrar el contenido que antes se había impreso con el comando cat mis_archivos.txt

Anexo**XI**

```
mi_shell>> cat mis_archivos.txt  
total 80  
drwxrwxrwx 2 toni toni 4096 oct 14 16:33 include  
-rwxrwxr-x 1 toni toni 70184 oct 14 16:35 minishell  
-rw-rw-r-- 1 toni toni 0 oct 14 17:02 mis_archivos.txt  
drwxrwxrwx 2 toni toni 4096 oct 14 16:34 src
```

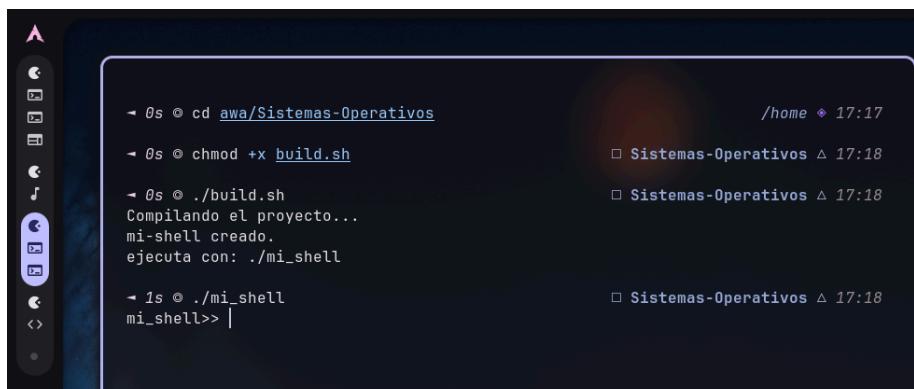
Para terminar la prueba de salida se usará salir .

Anexo**XII**

```
mi_shell>> salir  
toni@toni-VirtualBox:~/Sistemas_Operativos$
```

Sistema Operativo Arch Linux

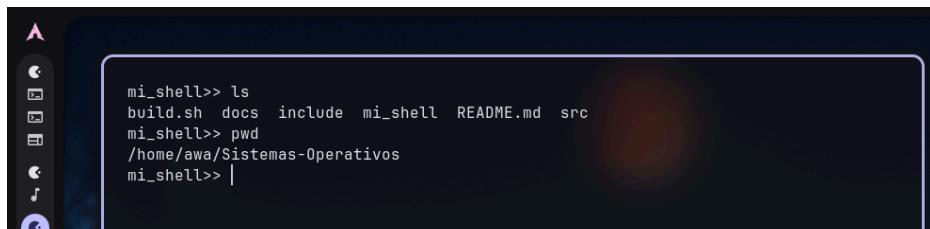
Anexo XIII



```
- 0s @ cd awa/Sistemas-Operativos
- 0s @ chmod +x build.sh
- 0s @ ./build.sh
Compilando el proyecto...
mi-shell creado.
ejecuta con: ./mi_shell
- ls @ ./mi_shell
mi_shell>> |
```

Descripción: Primero ingresamos a la carpeta donde se encuentra nuestro proyecto, seguidamente utilizamos el comando chmod +x para brindar los permisos de ejecución y finalmente ./mi_shell para abrir nuestra shell.

Anexo XIV



```
mi_shell>> ls
build.sh docs include mi_shell README.md src
mi_shell>> pwd
/home/awa/Sistemas-Operativos
mi_shell>> |
```

Descripción: En esta imagen podemos observar los comandos básicos como por ejemplo ls para visualizar los archivos que tenemos en la ruta actual. El comando pwd para revisar la ruta actual

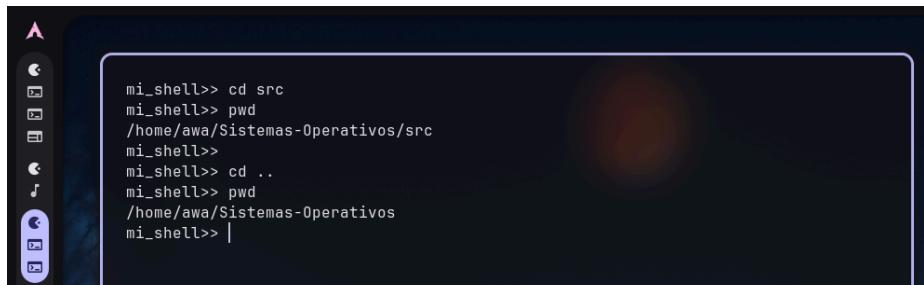
Anexo XV



```
mi_shell>> ls -l
total 96
-rwxr-xr-x 1 awa awa 270 oct 14 16:40 build.sh
drwxr-xr-x 2 awa awa 4096 oct 14 15:59 docs
drwxr-xr-x 2 awa awa 4096 oct 14 15:59 include
-rwxr-xr-x 1 awa awa 76032 oct 14 16:47 mi_shell
-rw-r--r-- 1 awa awa 280 oct 14 15:59 README.md
drwxr-xr-x 2 awa awa 4096 oct 14 16:23 src
mi_shell>> echo "hola"
"holo"
mi_shell>> |
```

Descripción: En esta imagen ls -l funciona para mostrar la lista de archivos en formato largo y el comando echo para imprimir en pantalla

Anexo XVI



```
mi_shell>> cd src
mi_shell>> pwd
/home/awa/Sistemas-Operativos/src
mi_shell>> cd ..
mi_shell>> pwd
/home/awa/Sistemas-Operativos
mi_shell>> |
```

Descripción: En esta parte estamos corroborando la funcionalidad de los comandos internos como por ejemplo cd donde podemos ingresar a otras carpetas o retroceder con cd .. al anterior directorio, para corroborar las operaciones con pwd nos muestra la dirección donde nos encontramos

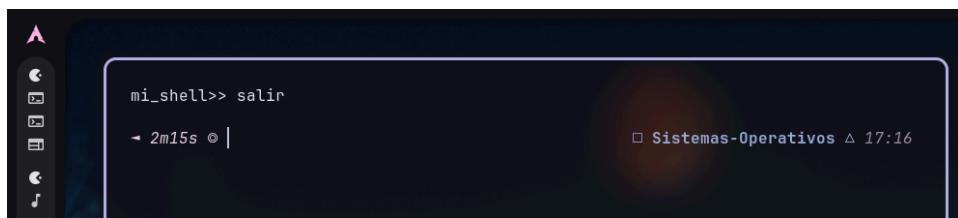
Anexo XVII



```
mi_shell>> ls | grep .cpp
mi_shell>> ls
build.sh docs include mi_shell README.md src
mi_shell>> cd src
mi_shell>> ls | grep .cpp
executor.cpp
main.cpp
parser.cpp
mi_shell>> |
```

Descripción: pruebas de PIPES

Anexo XVIII



```
mi_shell>> salir
- 2m15s @ |
□ Sistemas-Operativos △ 17:16
```

Descripción: Finalmente probamos el comando salir para poder salir de la minishell