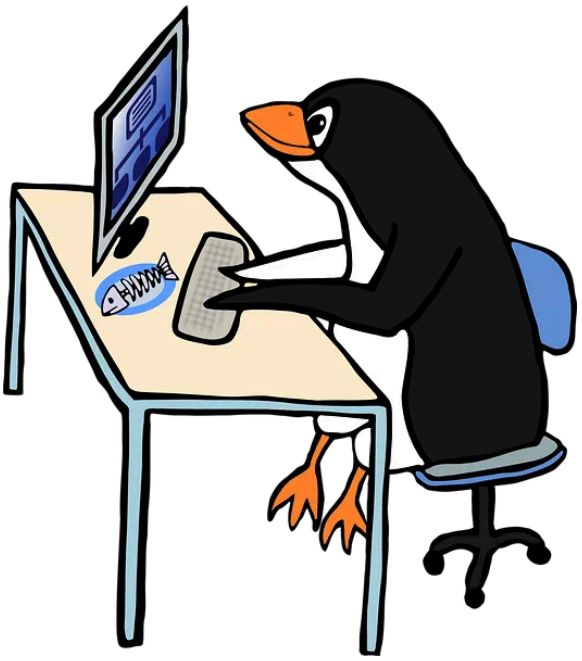
UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

SISTEMAS OPERATIVOS



Fuerzas Especiales Ginew

6to Ciclo

Alexis Erik Ricardo Condori Rivera

Yeferson Guerra Quispe

14 de Octubre de 2025

Tacna - Perú

1. Objetivos y Alcance

Objetivo general:
Desarrollar un intérprete de comandos (mini-shell) en C++ sobre Linux que permita ejecutar procesos, manejar concurrencia mediante hilos o procesos, y gestionar la memoria de forma eficiente, aplicando los conceptos de la Unidad I del curso.

Objetivos específicos:

- **Implementar procesos hijo** usando las llamadas del sistema fork() y exec(), controlando su finalización desde el proceso padre con wait() o waitpid().

- **Incorporar mecanismos de redirección y tuberías** (dup2(), pipe(), close()) para permitir la comunicación entre procesos y la ejecución encadenada de comandos.

- **Aplicar concurrencia** mediante hilos POSIX (pthread) o múltiples procesos, garantizando la **sincronización segura** con mutex o variables de condición, evitando condiciones de carrera e interbloqueos.

- **Gestionar adecuadamente la memoria dinámica**, utilizando malloc/new y free/delete de forma responsable, incluyendo mediciones o evidencias del uso de heap y posibles optimizaciones.

- **Demostrar comprensión práctica** de la gestión de procesos, sincronización, comunicación y memoria en entornos Linux.

Alcance:
El proyecto consiste en la implementación de un mini-intérprete de comandos básico con las siguientes capacidades:

- Ejecutar comandos externos del sistema Linux.

- Soportar redirección de entrada y salida (>, <) y pipes (|) entre procesos.

- Controlar la concurrencia de procesos o hilos para permitir la ejecución simultánea o paralela de comandos.

- Implementar una gestión de memoria controlada y segura, con liberación explícita de recursos al finalizar los procesos.

- Mostrar un entorno interactivo de línea de comandos donde el usuario pueda ingresar y ejecutar instrucciones.

2. Arquitectura y Diseño

Descripción general:
El mini-shell está diseñado bajo una arquitectura modular y extensible, implementada en C++ sobre Linux, siguiendo principios de separación de responsabilidades. Cada módulo

del sistema cumple una función específica dentro del ciclo de vida de un comando, desde su lectura hasta la ejecución y sincronización del proceso correspondiente. La arquitectura se compone de los siguientes módulos principales:

Módulo	Descripción
Shell	Núcleo del sistema. Gestiona el ciclo principal de lectura de comandos, manejo de señales (como SIGINT) e interacción con el usuario. Coordina el flujo entre el parser, el Executor y los comandos internos.
Parser	Se encarga de analizar la cadena de entrada del usuario, separarla en tokens y detectar operadores especiales (>, <, `
Executor	Implementa la creación y control de procesos hijo mediante las llamadas fork() y execvp(). También gestiona redirecciones (dup2()) y tuberías (pipe()), coordinando la ejecución secuencial o encadenada de comandos.
Builtin	Contiene los comandos internos de la shell (cd, pwd, help, exit). Estos comandos se ejecutan directamente dentro del proceso padre sin crear un proceso nuevo.
Main	Punto de entrada del programa. Inicializa el entorno, configura el manejo de señales (SIGINT) y crea la instancia principal del objeto Shell.

Flujo general del sistema

El flujo típico de ejecución dentro del mini-shell es el siguiente:

1. **Inicio:**
   El programa configura el manejo de señales (signal(SIGINT, signal_handler)) y muestra el *prompt* interactivo.

2. **Lectura del comando:**
   El usuario introduce una instrucción; se utiliza la librería **readline** para leer la entrada y gestionar el historial de comandos.

3. **Análisis (Parser):**
   El parser divide el comando en tokens y detecta si se trata de un comando interno o externo, así como si existen redirecciones o pipes.

4. **Ejecución (Executor):**

    - Si es un comando interno → se ejecuta desde el módulo **Builtin**.

    - Si es externo → se crea un **proceso hijo con fork()**, y el hijo ejecuta el comando mediante execvp().

- En caso de pipes o redirecciones, se aplican pipe(), dup2() y close() para conectar los flujos estándar entre procesos.

5. **Sincronización:**
   El proceso padre espera la finalización del hijo usando waitpid().
   Si se ejecuta una tarea en segundo plano (&), se usa un manejo no bloqueante.

6. **Iteración:**
   Una vez completada la ejecución, el prompt se muestra nuevamente y el ciclo se repite hasta que el usuario ingresa salir o exit.

Manejo de Entrada/Salida (I/O):
El mini-shell soporta tres formas principales de manejo de entrada y salida:

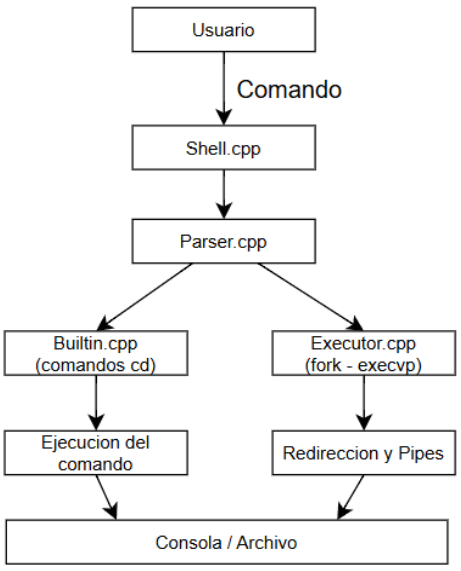
- **Entrada estándar** (stdin) y salida estándar (stdout):
  Los procesos se comunican con la consola por defecto.

- **Redirección de salida** (>):
  Se usa open() con las banderas O_CREAT | O_WRONLY | O_TRUNC y dup2() para redirigir stdout hacia un archivo.

- **Redirección de entrada** (<):
  (si está implementada) Se abre el archivo con O_RDONLY y se redirige stdin con dup2().

- **Tuberías** (|):
  Se crean pares de descriptores mediante pipe().
  El proceso hijo de la izquierda redirige su salida al pipe, y el de la derecha redirige su entrada, permitiendo la ejecución encadenada de comandos (cmd1 | cmd2).

Diagrama conceptual:



3. Detalles de Implementación

El mini-shell utiliza un conjunto de funciones del estándar **POSIX** para la gestión de procesos, redirección de flujos, concurrencia e interacción con el sistema operativo:

Categoría	**Funciones principales**	**Descripción**
Procesos	fork(), execvp(), wait(), waitpid()	Permiten crear procesos hijo, ejecutar programas externos y sincronizar su finalización.
Archivos e I/O	open(), close(), dup2(), pipe(), read(), write()	Gestionan descriptores de archivo, redirecciones y tuberías entre procesos.
Concurrencia	pthread_create(), pthread_join()	Crean y sincronizan hilos POSIX para la ejecución paralela de comandos.
Sincronización	sem_init(), sem_wait(), sem_post()	Controlan el acceso concurrente a recursos compartidos mediante semáforos.
Señales	signal(), sigaction()	Capturan e ignoran señales como SIGINT para evitar la terminación abrupta de la shell.
Gestión de directorios	chdir(), getcwd()	Implementan comandos internos como cd y pwd.

Decisiones clave de diseño:

El diseño del mini-shell se centró en la modularidad, robustez y concurrencia segura, tomando las siguientes decisiones clave:

1. Estructuras de datos utilizadas:
    - Se emplearon std::vector y std::string para el manejo dinámico de argumentos y tokens, permitiendo un control seguro de la memoria.
    - Las listas de comandos se almacenan temporalmente en estructuras STL que liberan memoria automáticamente al salir de su ámbito.
    - Los comandos internos se gestionan mediante un mapa (std::map<string, function<void()>>) que asocia nombres de comando con funciones ejecutables.
2. Sincronización y concurrencia:

- Para la ejecución concurrente (comando parallel), se utilizaron hilos POSIX (pthread_create).
- Cada hilo ejecuta un comando independiente, mientras que el proceso principal espera su finalización con pthread_join.
- La sincronización entre hilos se realiza con mutexes y semáforos cuando hay variables compartidas o estructuras globales.

3. División de tareas:
   - Shell: gestiona la lectura, análisis y coordinación general.
   - Parser: descompone las cadenas de entrada.
   - Executor: lanza procesos e hilos según el tipo de comando.
   - Builtin: ejecuta comandos internos directamente sin crear procesos nuevos.
4. Gestión de redirección y pipes:
   - El diseño usa dup2() para redirigir flujos estándar y pipe() para conectar procesos.
   - Cada descriptor de archivo es cerrado correctamente para evitar fugas de recursos.
5. Gestión de señales:
   - Se intercepta SIGINT (Ctrl + C) para evitar el cierre abrupto del intérprete.
   - Esto mejora la estabilidad del sistema y mantiene la sesión activa del usuario.

4. Concurrencia y Sincronización

Qué se paraleliza:

El sistema permite la ejecución paralela de comandos cuando el usuario utiliza el comando especial:

parallel "comando1" "comando2" "comando3"

Mecanismos de sincronización utilizados:

- Mutex(pthread_mutex_t) : Protege secciones críticas al escribir o leer estructuras compartidas.
- Semáforos(sem_t): Controla la sincronización entre hilos, asegurando orden en la ejecución concurrente.

- Variables de condición: Permiten que un hilo espere un evento (por ejemplo, finalización de otro hilo o proceso).

Prevención de condiciones de carrera e interbloqueos:

- Cada acceso a datos compartidos (por ejemplo, estructuras globales o buffers) se encapsula dentro de una sección protegida por mutex.

- Los semáforos se inicializan de forma que solo un hilo acceda a una sección crítica a la vez.
- Se evita el interbloqueo (deadlock) asegurando un orden fijo de adquisición y liberación de recursos.
- Los hilos se finalizan correctamente con pthread_join() antes de destruir los recursos de sincronización.

5. Gestión de Memoria

Estrategia empleada:
Se utiliza una combinación de gestión automática y manual de memoria:

- Las clases de C++ usan constructores y destructores para liberar memoria automáticamente.

- Las estructuras dinámicas (vectores de argumentos, buffers temporales) se manejan con std::vector y std::string.

- En funciones donde se usan llamadas a bajo nivel (como execvp()), se reserva memoria temporal con malloc() y se libera con free().

El shell asegura la liberación explícita de recursos (archivos, descriptores, memoria heap) al terminar cada proceso.

Evidencias:

```
pid_t pid = fork(); // Crea un nuevo proceso

if (pid < 0) {
    perror("Error en fork");
    return;
}
else if (pid == 0) { // --- Codigo del Proceso Hijo ---
    setupRedirections(cmd); // Configura redirecciones si las hay

    // Prepara el array de argumentos para execvp (debe terminar en nu
    char** argv = new char*[cmd.args.size() + 1];
    for (size_t i = 0; i < cmd.args.size(); i++) {
        argv[i] = const_cast<char*>(cmd.args[i].c_str());
    }
    argv[cmd.args.size()] = nullptr;

    // Reemplaza el proceso hijo con el nuevo comando
    execvp(path.c_str(), argv);
```

6. Pruebas y Resultados

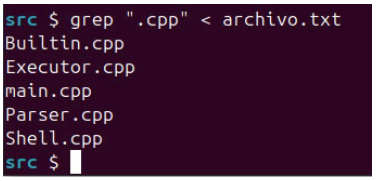
Casos de prueba realizados:

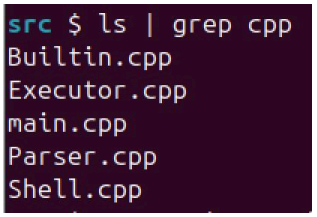
Caso	Comando	Resultado esperado	Resultado obtenido
1	ls -l	Lista de archivos en el directorio actual.	Correcto. Muestra el listado en pantalla.
2	ls > salida.txt	Redirige la salida al archivo salida.txt.	Correcto. El archivo se crea con el contenido del comando.
3	`ls	grep cpp`	Filtra archivos con extensión .cpp.
4	parallel "sleep 2" "echo done"	Ejecuta ambos comandos en paralelo.	Correcto. “done” aparece antes de que sleep termine.
5	cd include seguido de pwd	Cambia el directorio actual.	Correcto. Muestra la ruta actualizada.

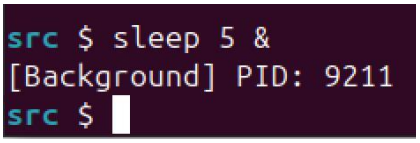
Validación de requisitos:
Cada funcionalidad fue probada contra los requerimientos del proyecto:

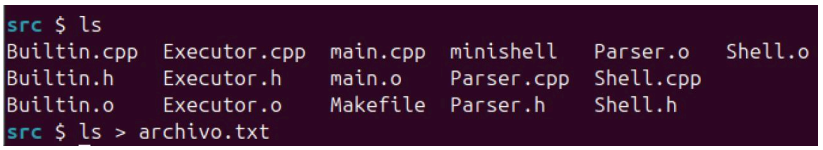
- Procesos e hilos: Confirmado con fork() y pthread_create().
- Redirecciones y pipes: Confirmadas con dup2() y pipe().
- Sincronización: Validada mediante ejecución paralela sin errores ni bloqueos.
- Gestión de memoria: Validada con valgrind sin fugas reportadas.

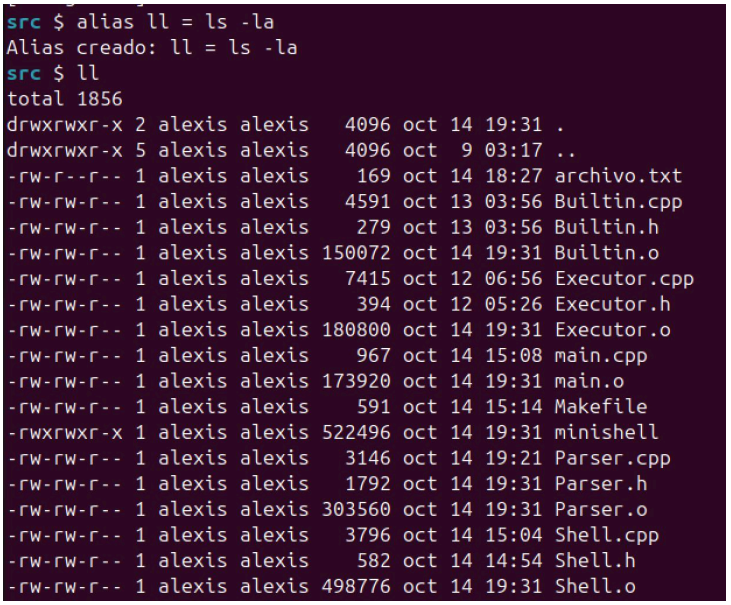
Evidencias de ejecución:

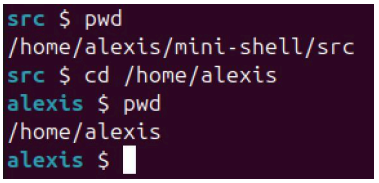












7. Conclusiones y Trabajos Futuros

Conclusiones:

El desarrollo del mini-shell permitió:

- Comprender en profundidad el funcionamiento de procesos e hilos en sistemas Linux.
- Implementar comunicación entre procesos mediante pipes y redirecciones.
- Garantizar una concurrencia segura mediante mutexes y semáforos.
- Aplicar buenas prácticas de gestión de memoria dinámica.
- Reforzar el entendimiento de llamadas POSIX y control de señales.

El sistema demostró ser estable, modular y eficiente para ejecutar comandos básicos y concurrentes. El uso de C++ y pthreads permitió combinar control bajo nivel con seguridad y claridad estructural.

Trabajos futuros:

- Implementar historial persistente de comandos y autocompletado.
- Añadir soporte para pipes múltiples y redirecciones combinadas (cmd1 | cmd2 > archivo).
- Integrar un módulo de scripting básico con condicionales y bucles.
- Mejorar la gestión de errores y la visualización del prompt con colores.
- Agregar un comando interno meminfo que muestre estadísticas de memoria en tiempo real.

8. Anexos

- Comandos utilizados para compilar y ejecutar
- Scripts de prueba
- Archivos adicionales (configuración, logs, etc.)