UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INFORMÁTICA Y SISTEMAS

“Implementación de intérprete de comandos en C++ sobre LInux”

ASIGNATURA: Sistemas Operativos

DOCENTE: MSc. Hugo Manuel Barraza Vizcarra

SECCIÓN: A

FECHA: 15 de Octubre del 2025

INTEGRANTES:

Wilbert Raúl Copaja Huayta

2023-119016

Cristhian Angel Flores Miranda

2023-119018

Tacna – Perú

2025

1. Objetivos y Alcance

   1.1 Objetivos generales:

   - Implementar un intérprete de comandos funcional
   - Demostrar el uso correcto de llamadas POSIX
   - Crear código modular y bien documentado

   1.2 Objetivos específicos:

   - Ejecutar comandos externos con fork() + execvp()
   - Implementar pipes para conectar comandos
   - Soportar redirecciones de entrada/salida
   - Ejecutar procesos en segundo plano
   - Manejar señales correctamente

   1.3 Alcance:

   El sistema implementa las siguientes funcionalidades:

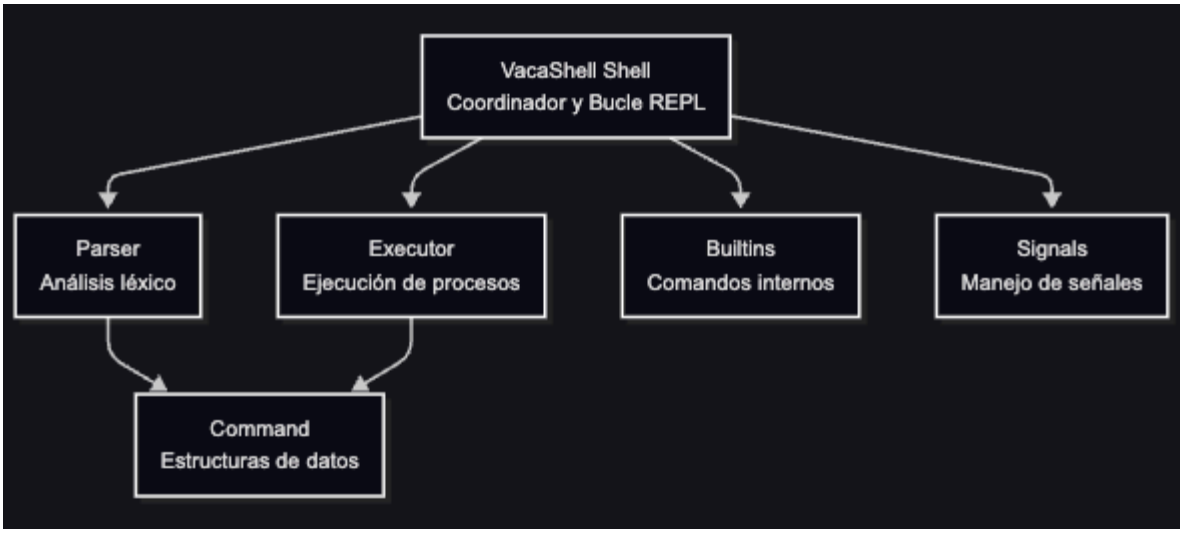
   - Ejecución de comandos externos con resolución automática de rutas
   - Pipelines para conectar múltiples comandos (cmd1 | cmd2 | cmd3)
   - Redirecciones de I/O: entrada (<), salida (>, >>)
   - Ejecución en segundo plano con operador &
   - Comandos internos (builtins): cd, pwd, exit, help, jobs
   - Manejo de señales: SIGINT (Ctrl+C), SIGCHLD
   - Prompt personalizado con información contextual

2. Arquitectura y diseño

   2.1 Vaca-Shell implementa una arquitectura modular orientada a objetos fundamentada en el principio de separación de responsabilidades (Single Responsibility Principle, SRP). El sistema se descompone en seis módulos cohesivos e independientes.

   2.2 Componentes del sistema

   Descripción de módulos:



2.3 Flujo de control:

El sistema opera mediante un ciclo iterativo. Anexo A

3. Ejecución de Comandos y Mecanismos del Shell

   3.1 Ejecución de Comandos Simples

   Para ejecutar un comando externo, como ls -la, la shell utiliza el patrón fork-and-exec.

   - fork(): La shell crea un proceso hijo, que es una copia exacta de sí misma.
   - Proceso hijo: Este proceso configura las redirecciones de entrada/salida si es necesario y luego usa execvp() para reemplazar su propio código por el del comando a ejecutar (por ejemplo, ls).
   - Proceso padre: La shell (proceso padre) utiliza waitpid() para esperar a que el proceso hijo termine.
     - Si el comando se ejecuta en segundo plano (usando &), el padre no espera al hijo y muestra el prompt inmediatamente.

   3.2 Implementación de Tuberías (Pipes)

   Para conectar comandos mediante tuberías (por ejemplo, cat archivo | grep "texto"), la shell crea canales de comunicación entre procesos.

   - Creación: Para *n* comandos encadenados, se crean *n-1* tuberías (pipe()).
   - Conexión: Se genera un proceso hijo para cada comando. Mediante dup2(), la salida estándar (STDOUT) del primer comando se conecta a la entrada estándar (STDIN) del siguiente, y así sucesivamente.

- Cierre de descriptores: Es esencial que cada proceso (tanto padre como hijos) cierre las copias de los descriptores de las tuberías que no utiliza. Esto evita bloqueos (deadlocks) y fugas de recursos.

3.3 Redirecciones de Entrada y Salida

La shell maneja las redirecciones de entrada (<) y salida (> o >>) manipulando los descriptores de archivo del proceso.

- Redirección de entrada (<):
  Ejemplo: wc -l < archivo.txt
  Se abre el archivo y se utiliza dup2() para redirigir la entrada estándar (STDIN) al archivo, reemplazando el teclado como fuente de entrada.
- Redirección de salida (> o >>):
  Ejemplo: ls > salida.txt
  Se abre el archivo (creándolo, truncándolo o añadiendo contenido) y se utiliza dup2() para redirigir la salida estándar (STDOUT) al archivo, reemplazando la terminal como destino.

3.4 Manejo de Señales

La shell gestiona señales asíncronas, como las generadas por Ctrl+C, mediante el uso de sigaction(), asegurando un comportamiento controlado y estable.

SIGINT (Ctrl+C):

a. Si hay un comando en primer plano, la shell captura la señal y la reenvía al proceso hijo, permitiendo que este se interrumpa correctamente.
b. Si no hay ningún comando ejecutándose, la shell simplemente muestra una nueva línea del prompt.

SIGCHLD:
Esta señal se activa cuando un proceso hijo (especialmente uno en segundo plano) termina.
La shell la utiliza para “limpiar” los procesos terminados y evitar la acumulación de procesos zombi.

4. Concurrencia y sincronización: qué se paraleliza y cómo se evita la condición de carrera/interbloqueo

   4.1. ¿Qué se paraleliza?

   VacaShell ejecuta procesos en paralelo en tres situaciones clave:

   1. Comandos Externos: Cada comando (como ls, grep, etc.) se ejecuta en un proceso hijo creado con fork(). La shell principal espera a que termine usando waitpid() antes de continuar.
   2. Procesos en Segundo Plano (&): Si un comando termina con &, la shell crea un proceso hijo pero no espera a que termine. Esto permite que la shell y el comando se ejecuten de forma paralela, dejando el prompt disponible para el usuario inmediatamente.
   3. Tuberías (|): Para un comando como cat archivo | grep "texto", VacaShell crea un proceso para cada parte (cat y grep). Estos procesos se ejecutan al mismo tiempo, y el sistema operativo gestiona el flujo de datos entre ellos a través de las tuberías.

   4.2. ¿Cómo se evita la condición de carrera?

   Una condición de carrera ocurre cuando el resultado de una operación depende del orden incontrolado en que varios procesos acceden a un recurso compartido. VacaShell lo previene así:

   1. Aislamiento de Memoria (Mecanismo Principal): Al usar fork(), cada proceso hijo obtiene una copia de la memoria de la shell principal. Cualquier cambio que haga un hijo a sus variables no afecta a la shell padre, eliminando el riesgo de que los comandos externos corrompan datos internos como el historial o los alias.
   2. Gestión de la Lista de Trabajos (jobs): Para evitar conflictos al actualizar la lista de procesos en segundo plano, el manejador de señales SIGCHLD está diseñado para ser muy rápido y no bloqueante, "cosechando" procesos terminados sin interferir con otros comandos.
   3. Acceso a la Terminal: La salida de un proceso en segundo plano puede mezclarse con lo que el usuario está escribiendo. Esto no se evita, ya que es el comportamiento estándar en las shells POSIX y no corrompe el estado interno del programa.

   4.3. ¿Cómo se evita la condición de interbloqueo?

Un interbloqueo (deadlock) sucede cuando dos o más procesos se quedan bloqueados, esperándose mutuamente. VacaShell lo evita con dos estrategias:

1. Jerarquía de Espera Clara: El flujo es unidireccional. La shell padre espera a los hijos, pero los hijos (execvp()) nunca esperan al padre. Esto rompe cualquier posible ciclo de espera.
2. Manejo Correcto de Tuberías: Este es el punto más crítico. Para evitar que los procesos se queden esperando datos que nunca llegarán, VacaShell se asegura de que:
    - Cada proceso hijo cierre los extremos de la tubería que no necesita.
    - La shell padre cierre ambos extremos de la tubería después de crear a los hijos. Esto es crucial para que, cuando el proceso escritor termine, el proceso lector reciba la señal de fin de fichero (EOF) y no se quede bloqueado.

5. Gestión de memoria: estrategias y evidencias

5.1. Estrategias

Estrategias/mecanismos	Propósito e Implementación
RAII (Resource Acquisition Is Initialization)	Utiliza contenedores de la STL (std::string, std::vector) que gestionan su memoria de forma automática. Elimina el new/delete manual, previniendo fugas de memoria por diseño.
Memoria de Pila (Stack) vs. Montón (Heap)	Prioriza el uso de la memoria de Pila (Stack) para objetos y variables, garantizando una liberación automática y rápida. La memoria del Montón (Heap) se usa de forma segura solo a través de los contenedores de la STL.
Aislamiento de Memoria por fork()	fork() crea un proceso hijo con una copia del espacio de memoria del padre. Esto aísla a la shell de cualquier corrupción de memoria que pueda

	causar el comando ejecutado.
Limpieza de Memoria por execvp()	execvp() reemplaza por completo la memoria del proceso hijo con la del nuevo programa. Esto libera automáticamente toda la memoria copiada durante el fork, garantizando que no haya fugas en los hijos.

5.1. Evidencias

Para demostrar que la gestión de memoria es correcta y no existen fugas, se utilizó la herramienta de análisis de memoria Valgrind.

1. Ejecución del Caso de Prueba

Se ejecutó VacaShell bajo Valgrind, se interactuó con la shell ejecutando varios comandos (internos, externos, pipes) y finalmente se salió limpiamente.

valgrind --leak-check=full --show-leak-kinds=all ./vacashell

2. Análisis de Resultados

Al finalizar la ejecución, Valgrind generó el siguiente resumen:

```
==91414== HEAP SUMMARY:
==91414==     in use at exit: 160 bytes in 2 blocks
==91414==   total heap usage: 8 allocs, 6 frees, 76,047 bytes allocated
==91414==
==91414== LEAK SUMMARY:
==91414==    definitely lost: 0 bytes in 0 blocks
==91414==    indirectly lost: 0 bytes in 0 blocks
==91414==      possibly lost: 0 bytes in 0 blocks
==91414==    still reachable: 160 bytes in 2 blocks
==91414==         suppressed: 0 bytes in 0 blocks
==91414==
==91414== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

3. Interpretación de la Evidencia

- definitely lost: 0 bytes in 0 blocks: Esta es la línea más importante y la conclusión clave. Confirma que VacaShell no tiene ninguna fuga de memoria

(memory leak). Cada byte de memoria que fue asignado fue correctamente liberado o sigue siendo accesible.

- still reachable: 160 bytes in 2 blocks: Este mensaje no indica un error. Se refiere a memoria que todavía era accesible al momento de terminar el programa (por ejemplo, la memoria usada por el historial de comandos). Es una práctica común y eficiente no liberar esta memoria explícitamente, ya que el sistema operativo la reclama por completo e instantáneamente al finalizar el proceso.
- ERROR SUMMARY: 0 errors: Confirma que durante la ejecución no ocurrieron errores de memoria, como escribir o leer fuera de los límites de un bloque de memoria asignado.

6. Pruebas y resultados

6.1 Metodología de Pruebas

6.2 Casos de Prueba

6.2.1 Comandos Simples

- T1.1: Comando ls -la
  Resultado esperado: Listar los archivos con detalles.
  Estado: PASS
- T1.2: Comando pwd
  Resultado esperado: Mostrar el directorio actual.
  Estado: PASS
- T1.3: Comando echo "test"
  Resultado esperado: Imprimir el texto “test”.
  Estado: PASS
- T1.4: Comando comando_inexistente
  Resultado esperado: Mostrar el error “comando no encontrado”.
  Estado: PASS

6.2.2 Redirecciones

- T2.1: Comando ls > out.txt
  Resultado esperado: Crear un archivo con el listado de archivos.
  Estado: PASS
- T2.2: Comando echo "test" >> out.txt
  Resultado esperado: Anexar el texto al archivo existente.
  Estado: PASS

- T2.3: Comando wc -l < out.txt
  Resultado esperado: Contar las líneas del archivo de entrada.
  Estado: PASS
- T2.4: Comando cat < noexiste.txt
  Resultado esperado: Mostrar error indicando que el archivo no existe.
  Estado: PASS

6.2.3 Pipes

- T3.1: Comando ls | wc -l
  Resultado esperado: Contar la cantidad de archivos listados.
  Estado: PASS
- T3.2: Comando cat file | grep pattern
  Resultado esperado: Filtrar las líneas que contienen el patrón especificado.
  Estado: PASS
- T3.3: Comando ps aux | grep bash | wc -l
  Resultado esperado: Ejecutar una cadena de tres comandos en pipeline.
  Estado: PASS
- T3.4: Comando ls | grep txt | sort | uniq
  Resultado esperado: Filtrar, ordenar y eliminar duplicados en una secuencia de cuatro comandos.
  Estado: PASS

6.2.4 Segundo Plano

- T4.1: Comando sleep 5 &
  Resultado esperado: Ejecutar el proceso en segundo plano.
  Estado: PASS
- T4.2: Comando jobs
  Resultado esperado: Listar los procesos en ejecución en segundo plano.
  Estado: PASS
- T4.3: Comando ls | wc &
  Resultado esperado: Ejecutar un pipeline en segundo plano.
  Estado: PASS

6.2.5 Señales

- T5.1: Acción sleep 30 seguida de Ctrl+C
  Resultado esperado: Interrumpir la ejecución del comando.
  Estado: PASS

- T5.2: Acción Ctrl+C sin un comando activo
  Resultado esperado: Mostrar solo una nueva línea del shell.
  Estado: PASS
- T5.3: Acción sleep 30 & seguida de Ctrl+C
  Resultado esperado: No interrumpir el proceso en segundo plano.
  Estado: PASS

6.2.6 Comandos Internos

- T6.1: Comando cd /tmp
  Resultado esperado: Cambiar al directorio /tmp.
  Estado: PASS
- T6.2: Comando cd
  Resultado esperado: Ir al directorio HOME.
  Estado: PASS

7. Conclusiones y trabajos futuros

7.1 Conclusiones:

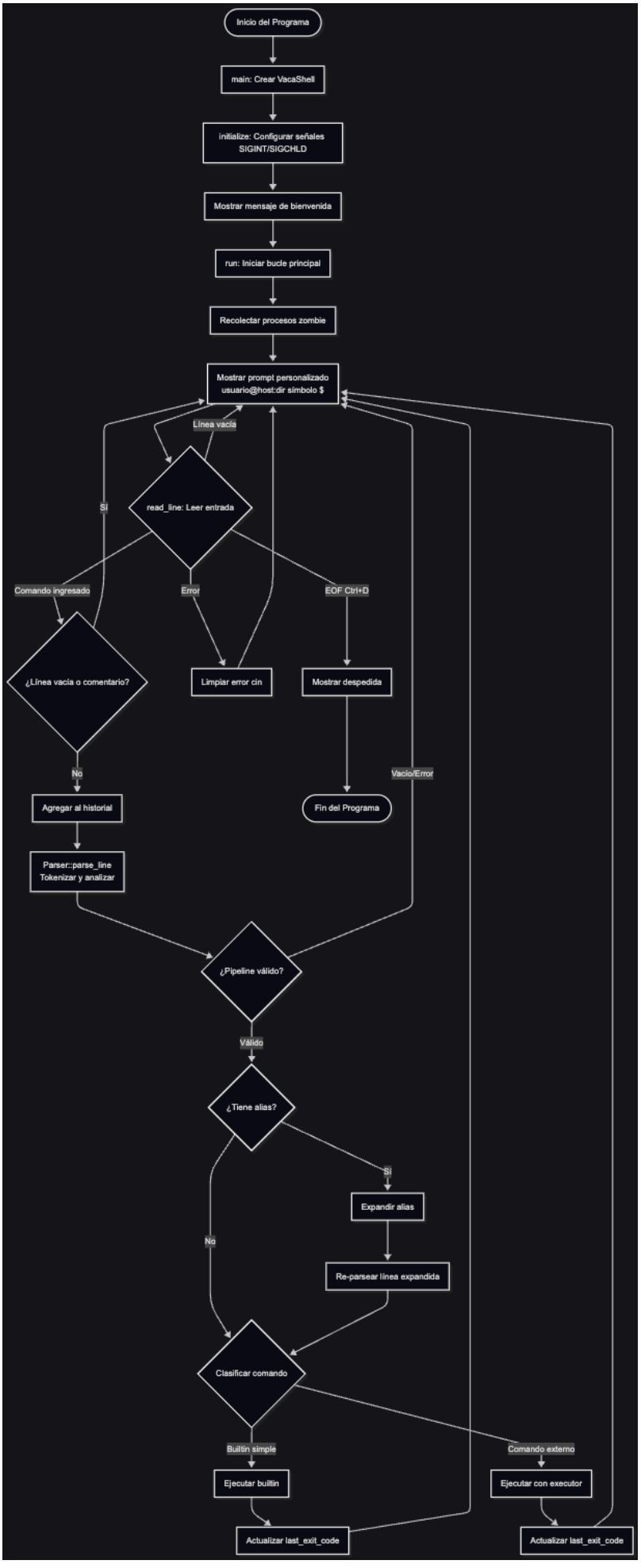
El proyecto VacaShell logró cumplir con sus objetivos académicos, demostrando una correcta aplicación de los principios de programación de sistemas. Se implementó exitosamente la gestión de procesos mediante el modelo *fork-exec*, la comunicación interproceso con *pipes*, y la gestión de I/O a través de descriptores de archivo y redirecciones. Además, se manejaron adecuadamente la concurrencia y las señales, garantizando estabilidad y control de ejecución. La arquitectura modular del sistema facilita su mantenimiento y futuras extensiones.

7.2 Trabajos futuros

Se propone ampliar el proyecto con funciones como control de trabajos (fg, bg, Ctrl+Z), soporte para variables de entorno, expansión y globbing, ejecución de scripts con estructuras de control, historial de comandos persistente y autocompletado mediante *readline*, para mejorar la funcionalidad y la experiencia de uso de la shell.

8. Anexos

Anexo A:


Inicio del Programa
main: Crear VacaShell
initialize: Configurar señales SIGINT/SIGCHLD
Mostrar mensaje de bienvenida
run: Iniciar bucle principal
Recolectar procesos zombie
Mostrar prompt personalizado usuario@host:dir símbolo $
Línea vacía
read_line: Leer entrada
Sí
Comando ingresado
Error
EOF Ctrl+D
¿Línea vacía o comentario?
Limpiar error cin
Mostrar despedida
No
Vacío/Error
Agregar al historial
Fin del Programa
Parser::parse_line Tokenizar y analizar
¿Pipeline válido?
Válido
¿Tiene alias?
Sí
Expandir alias
No
Re-parsear línea expandida
Clasificar comando
Builtin simple
Comando externo
Ejecutar builtin
Ejecutar con executor
Actualizar last_exit_code
Actualizar last_exit_code