Procesador de ficheros concurrente

16 DE NOVIEMBRE DE 2025

GONZALO SOTO RAMOS

PSP – DAM 2

Índice

1. INTRODUCCIÓN

El presente documento aborda el desarrollo del Proyecto 5, titulado **"Procesador de ficheros concurrente"**, en el marco de la asignatura de Programación de Servicios y Procesos. El objetivo principal de este proyecto es diseñar e implementar una aplicación de escritorio robusta y multiplataforma, capaz de gestionar y procesar múltiples archivos de texto de forma simultánea.

En el entorno de la programación y el desarrollo de sistemas, el procesamiento de grandes volúmenes de datos o de tareas que consumen mucho tiempo es un desafío constante. La ejecución secuencial de estas tareas puede resultar en experiencias de usuario pobres y un uso ineficiente de los recursos del sistema. Para contrarrestar esto, esta aplicación explota los principios de la concurrencia, lanzando un proceso de sistema operativo separado para cada combinación de archivo y operación solicitada por el usuario.

El proyecto se centra en la utilización de **Kotlin** y **Compose for Desktop** para la interfaz de usuario, garantizando una estética moderna y una experiencia de usuario intuitiva. A nivel técnico, la clave reside en la implementación de la clase ***ProcessBuilder*** de Java, fundamental para interactuar con comandos nativos del sistema operativo, como ***wc*** en entornos **Unix** (Linux y macOS) o sus equivalentes en **Windows**, permitiendo así contar líneas, palabras, caracteres o bytes de manera eficiente.

Además de la concurrencia y el uso de procesos externos, el desarrollo incluye la **detección automática del sistema operativo** y un riguroso **manejo de errores**, asegurando que el usuario reciba retroalimentación en tiempo real sobre el estado y el resultado de cada tarea de procesamiento. De este modo, este proyecto no solo cumple con los requisitos funcionales de procesamiento, sino que también sirve como una demostración práctica del manejo avanzado de la concurrencia, la interoperabilidad con el sistema operativo y la construcción de interfaces gráficas reactivas.

2. TECNOLOGÍAS USADAS Y DISEÑO DE LA ARQUITECTURA DEL SISTEMA

2.1. Tecnologías Usadas

El proyecto está construido sobre la plataforma Java/Kotlin y hace uso de herramientas modernas que facilitan la concurrencia, la creación de interfaces de escritorio y la interoperabilidad con procesos nativos del sistema operativo.

Tecnología	Justificación
Kotlin	Lenguaje moderno, conciso e interoperable con Java. Su soporte nativo para Corrutinas lo hace ideal para la gestión de la concurrencia asíncrona.
Compose for Desktop	Framework declarativo que permite construir interfaces de escritorio modernas, reactivas y de alto rendimiento. Garantiza una UI que no se bloquea durante las operaciones pesadas.
Kotlin Coroutines (kotlinx.coroutines)	Mecanismo de concurrencia ligera. Es esencial para lanzar múltiples tareas de procesamiento (Job por cada fichero/operación) sin bloquear el hilo principal de la aplicación. Se utiliza Dispatchers.IO para tareas de I/O bloqueantes.
Java ProcessBuilder	Herramienta de la JVM que permite ejecutar comandos del sistema operativo. Es la pieza central para cumplir el requisito de procesar cada fichero con un proceso separado (no solo un hilo).
Comandos Nativos	Se utilizan los comandos nativos del SO para el conteo: wc (para Linux/macOS) y su equivalente en PowerShell (para Windows). Esto garantiza una alta eficiencia y la compatibilidad con el requisito de ProcessBuilder.

2.2. Diseño de la Arquitectura del Sistema

El sistema sigue un patrón de diseño por capas, lo que permite una clara separación de responsabilidades, facilita el mantenimiento y mejora la portabilidad de la lógica de negocio. Se distinguen tres capas principales:

2.2.1. Capa de Utilidades / Sistema Operativo (OS)

Esta capa es la más baja y se encarga de la abstracción de las dependencias del sistema operativo.

Detección de SO: El archivo ***DetecSO.kt*** utiliza ***System.getProperty("os.name")*** para identificar si la aplicación se ejecuta en **WINDOWS, LINUX, MACOS u OTRO** (no soportado). Esta detección es clave para que la siguiente capa pueda construir correctamente los comandos de ***ProcessBuilder***.

Selección de Ficheros: Se utiliza la utilidad ***java.awt.FileDialog*** para abrir el diálogo de selección múltiple, gestionado de forma asíncrona para no bloquear el hilo de Compose.

2.2.2. Capa Core / Concurrencia (Lógica de Negocio)

Esta es la capa central y contiene el motor del proyecto, la clase ***ProceConcu***.

- ***ProceConcu*** (**Motor de Concurrencia**): Es responsable de gestionar y lanzar las tareas. Por cada combinación de fichero y operación seleccionada (e.g., Fichero A + Contar Líneas), lanza una Corrutina en el pool de hilos de I/O (***Dispatchers.IO***).
- **Ejecución de Procesos**: Dentro de cada Corrutina, se ejecuta el comando nativo mediante ***ProcessBuilder***. El método ***generarComandoArray*** es crucial, ya que adapta la sintaxis del comando para Windows o Unix basándose en el SO detectado.
- **Modelo de Datos**: Define las operaciones disponibles (***Operacion.kt***) y el resultado de cada tarea (***ProcessResult.kt***), permitiendo manejar tanto el éxito (con el valor de conteo) como el fracaso (con el mensaje de error).

2.2.3. Capa de Presentación (UI)

Implementada con **Compose for Desktop**, es la interfaz reactiva que interactúa con el usuario.

- **Recepción de Eventos**: Gestiona la selección de ficheros y de operaciones a través de la capa de Utilidades.
- **Lanzamiento y Monitoreo:** Llama a la capa Core para iniciar el procesamiento, recolectando la lista de Jobs lanzados.
- **Actualización en Tiempo Real:** Utiliza un *Callback* (***onFinalizado***) para recibir de la capa Core el resultado de cada tarea completada. Dado que el *callback* se ejecuta de forma segura en el hilo principal (***Dispatchers.Main***), la UI se actualiza inmediatamente, mostrando el progreso de forma asíncrona sin interrupciones.

El diseño garantiza que la interacción con los procesos nativos del SO (que son inherentemente síncronos y bloqueantes) no afecte en ningún momento a la fluidez de la interfaz gráfica, gracias al uso eficiente de las Corrutinas.

3. DESCRIPCIÓN DE LAS FUNCIONALIDADES CON CAPTURAS DEL PROCESO

El proyecto se centra en un flujo de trabajo claro y eficiente, donde el usuario selecciona los ficheros, elige las operaciones concurrentes a realizar y visualiza el resultado de forma asíncrona en la interfaz.

3.1. Detección del Sistema Operativo

Aunque es una funcionalidad interna, la detección del Sistema Operativo (SO) es la base de la portabilidad, ya que determina qué comandos nativos (***wc*** o ***PowerShell***) utilizará ***ProcessBuilder***.

Proceso: Al inicio de la aplicación, el componente ***App()*** llama a la función ***detectarSO()*** para establecer el entorno.

Implementación: Esta lógica se encuentra en el archivo ***DetecSO.kt*** y mapea la propiedad del sistema (***os.name***) a un ***enum*** seguro (***SOEnum***), permitiendo a la capa de negocio (***ProceConcu***) construir comandos específicos.

Resultado Visual: En la captura principal de la interfaz, se muestra claramente el SO detectado, confirmando que la aplicación está lista para usar los comandos apropiados para ese entorno.

Ilustración 1: Interfaz principal mostrando el Sistema Operativo detectado (WINDOWS).

Ilustración 2: Interfaz principal mostrando el Sistema Operativo detectado (LINUX).

3.2. Selección de Ficheros y Operaciones

La aplicación permite la selección múltiple de ficheros y de operaciones, creando la matriz de tareas a ejecutar de forma concurrente.

3.2.1. Selección de Ficheros

Interacción: El usuario pulsa el botón "Selecciona Archivo/s".

Proceso Asíncrono: Para evitar bloquear la interfaz gráfica (Compose for Desktop), la utilidad de selección de ficheros (***selecMultiFiche()***) se ejecuta en una Corrutina con ***Dispatchers.IO*** (hilo de entrada/salida) dentro del ***PanelDeSeleccion***.

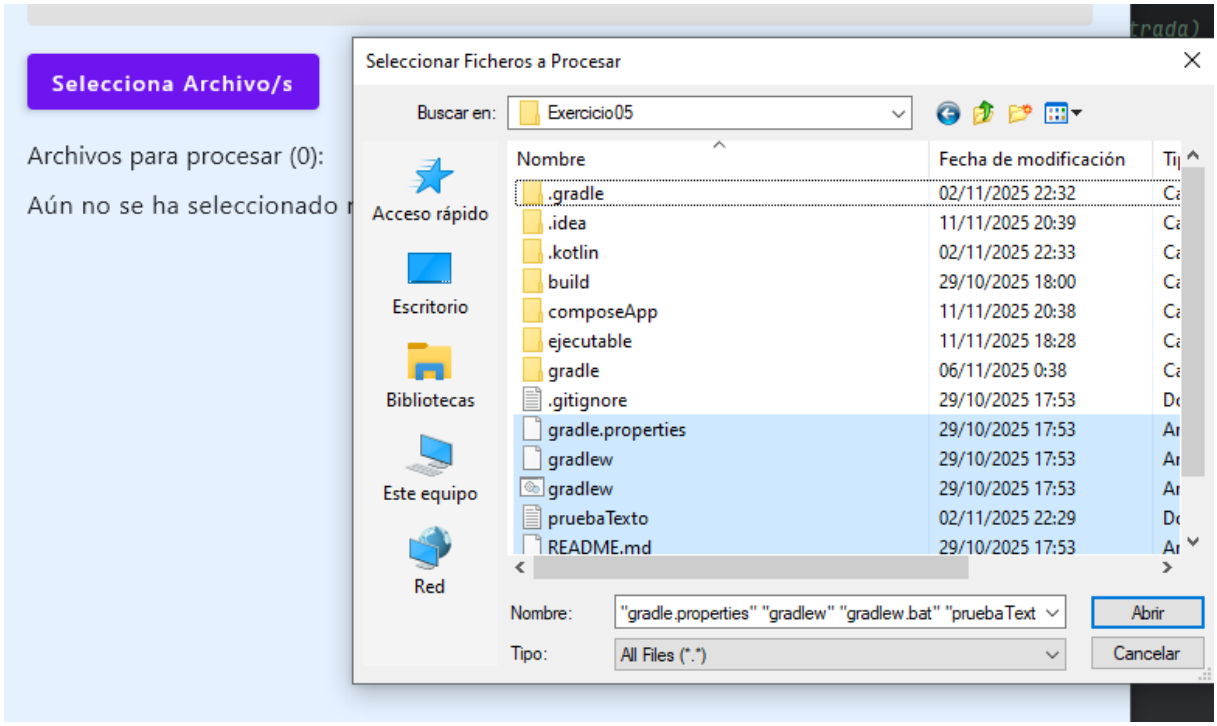

Ilustración 3: Selección de varios archivos

3.2.2. Selección de Operaciones

Interacción: Los checkboxes permiten seleccionar las operaciones requeridas (Contar Líneas, Palabras, Caracteres y Bytes). El usuario puede seleccionar una o varias a la vez.

Creación de Tareas: Una vez que el usuario pulsa "*Iniciar Procesamiento Concurrente*", el motor de la aplicación genera **una tarea concurrente** para cada combinación de ***{Fichero} x {Operación}*** seleccionada. Por ejemplo, 3 ficheros y 4 operaciones = 12 procesos concurrentes.

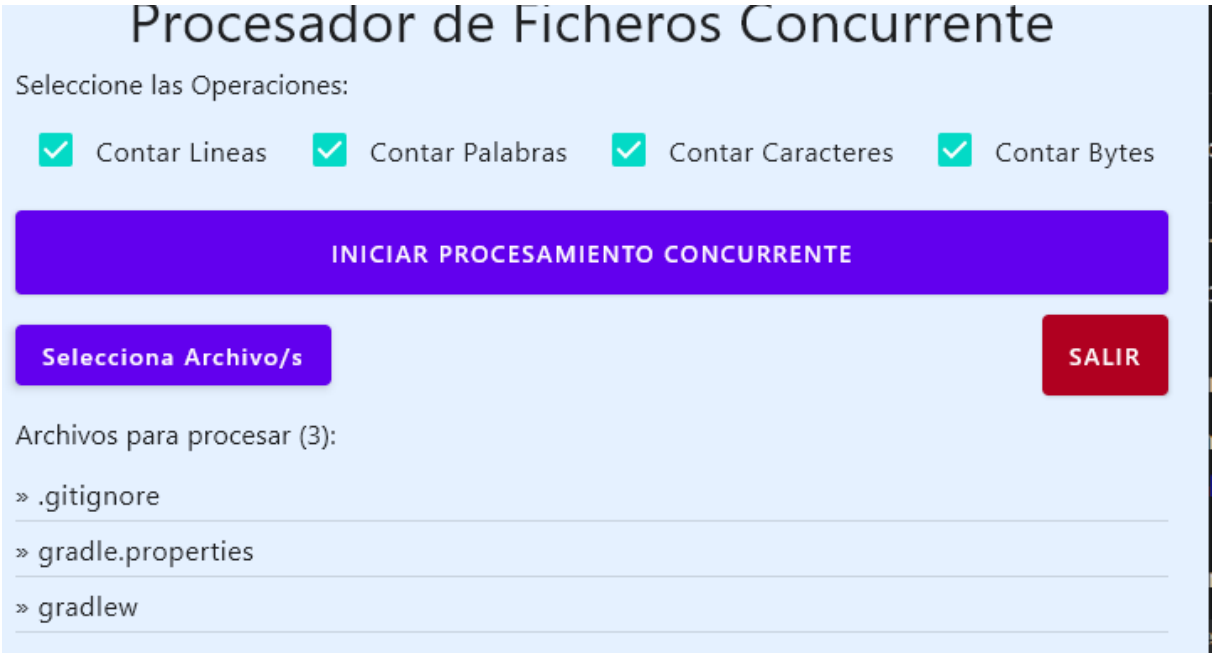

Ilustración 4: Estado de la interfaz antes de iniciar el procesamiento, mostrando tres ficheros seleccionados y las cuatro operaciones marcadas.

3.3. Procesamiento Concurrente y Visualización de Resultados

Esta es la funcionalidad central del proyecto, donde se gestiona la concurrencia y la actualización en tiempo real de la interfaz.

3.3.1. Lanzamiento de Corrutinas y Procesos

Motor: La clase ***ProceConcu*** toma la matriz de tareas y lanza una ***Job*** (Corrutina) por cada una de ellas.

Aislamiento: Cada Corrutina se ejecuta en ***Dispatchers.IO*** y, dentro de ella, se llama al método ***ejecutarProceso(fichero, operacion)***, que utiliza ***ProcessBuilde***r para lanzar un proceso nativo del SO completamente separado (***e.g., wc -l /path/to/file***).

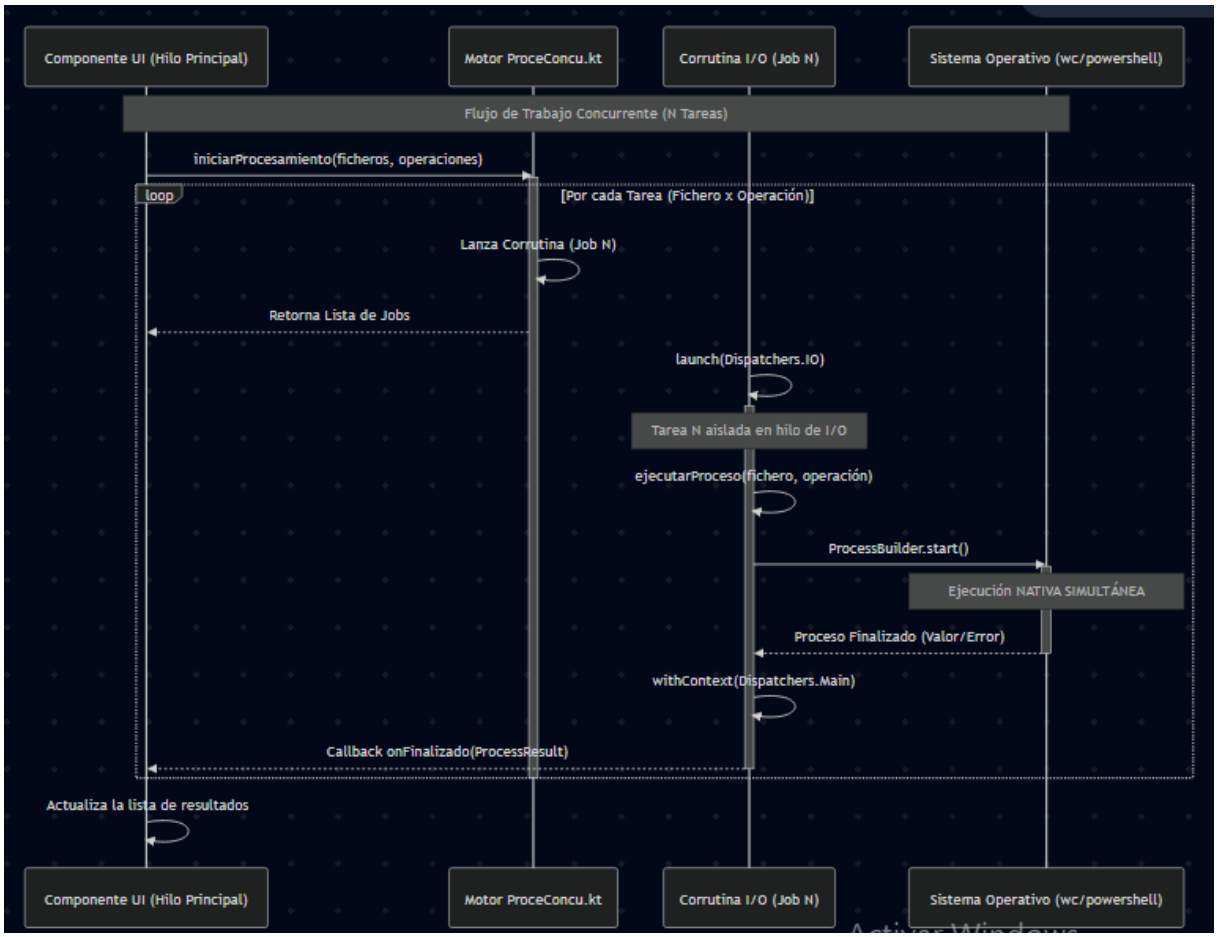

Ilustración 5: Diagrama de secuencia mostrando el lanzamiento y ejecución concurrente de múltiples procesos nativos (simultáneamente) desde la aplicación principal (Host). Cada Corrutina aísla el ProcessBuilder en un hilo I/O.

3.3.2. Actualización Asíncrona de la Interfaz

Callback de Notificación: Una vez que un proceso nativo termina, la Corrutina llama a un *callback* (***onFinalizado: (ProcessResult) -> Unit***) para devolver el resultado.

Seguridad de Hilos: El resultado (ya sea ***ProcessResult.Success*** o ***ProcessResult.Failure***) se notifica a la UI de forma segura mediante ***withContext(Dispatchers.Main)***, garantizando que Compose pueda actualizar su estado sin provocar errores de concurrencia.

Visualización en Tiempo Real: A medida que los procesos finalizan, los resultados aparecen inmediatamente en la lista de la derecha, mostrando la naturaleza asíncrona y no bloqueante del diseño.

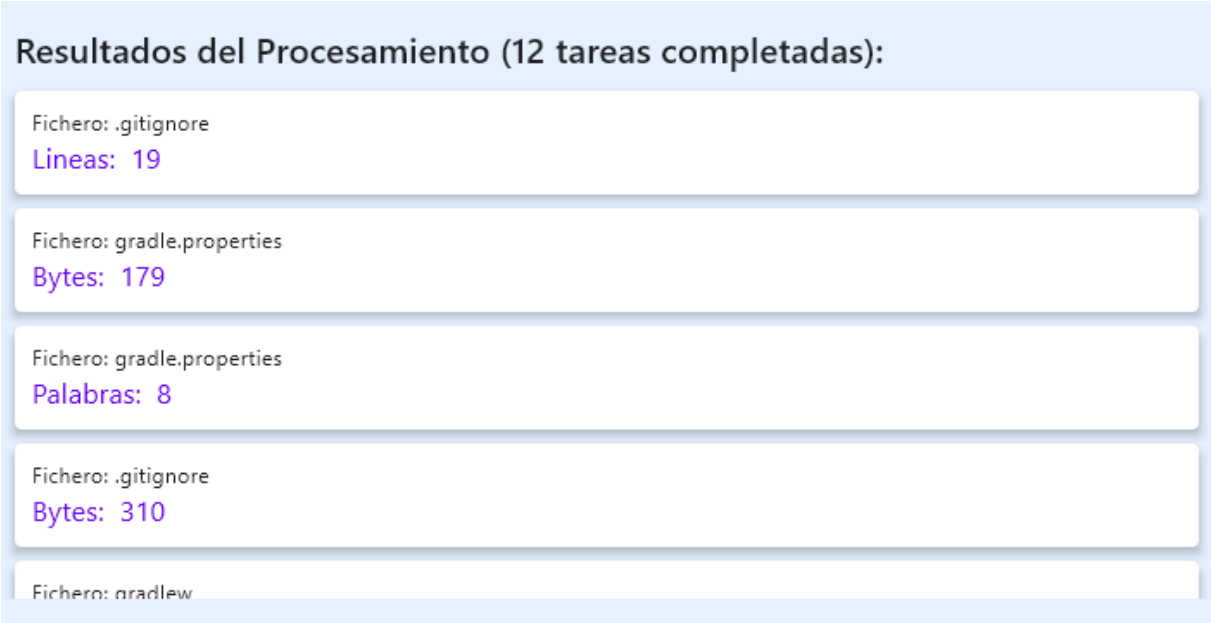

Ilustración 6: Interfaz mostrando los resultados finales del procesamiento.

3.3.3. Construcción y Adaptación del Comando

El método ***ejecutarProceso*** es responsable de generar la secuencia de comandos (tokens) adecuada para el SO detectado y la operación solicitada. Esto se consigue utilizando un gestor de comandos que mapea la ***TipoOperacion*** al comando nativo correspondiente:

Linux/macOS *: wc -l, wc -w, wc -m, wc -c*

Windows: *Measure-Object -Line (Líneas), Measure-Object -Word (Palabras), etc.*

Al construir el ***ProcessBuilder***, se le pasa la ruta del fichero como un argumento separado, permitiendo que la herramienta nativa lo procese directamente y asegurando una ejecución robusta y eficiente.

3.4. Manejo de Errores

El proyecto incluye manejo de errores obligatorio para notificar fallos en los procesos nativos.

Causa de Error: Si un fichero no existe, la ruta es incorrecta, o el comando nativo devuelve un código de salida distinto de cero, el proceso se clasifica como **ProcessResult.Failure**.

Visualización: El resultado de fallo muestra un mensaje de error claro en lugar de un número, y la tarjeta del resultado se resalta visualmente (con un fondo sutilmente rojo) para alertar al usuario.

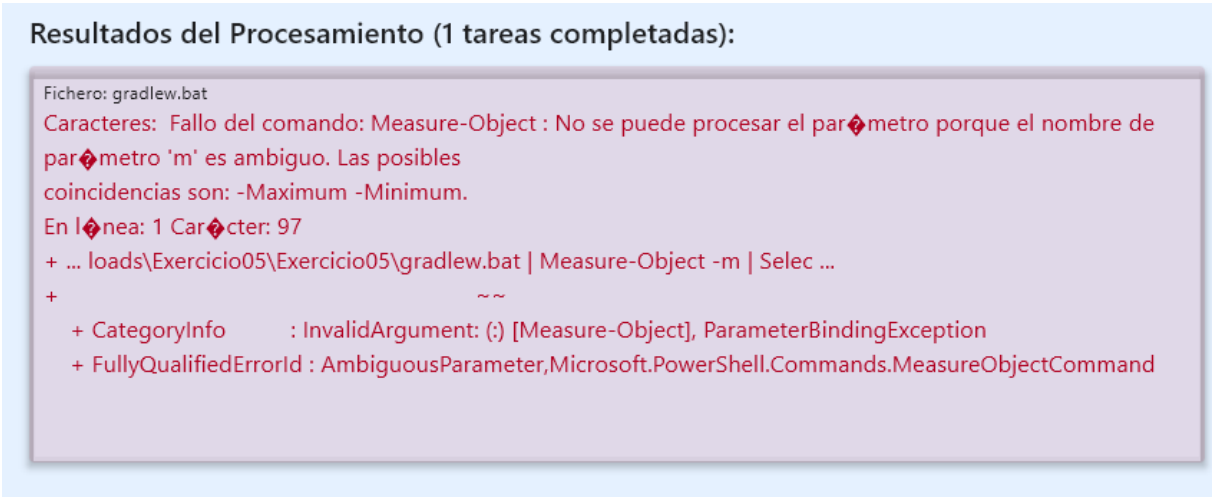

Ilustración 7: Ejemplo de manejo de errores.

4. BREVE MANUAL DE USUARIO

Este manual explica cómo utilizar la aplicación Procesador de Ficheros Concurrente para ejecutar tareas de conteo (líneas, palabras, caracteres, bytes) en múltiples archivos de forma simultánea, aprovechando la concurrencia del sistema operativo.

4.1. Requisitos y Detección Inicial

Requisitos del SO: La aplicación ha sido probada y funciona en sistemas operativos **Windows, Linux y macOS.**

Verificación del SO: Al iniciar la aplicación, compruebe el mensaje en la interfaz (esquina superior izquierda) que confirma la detección del Sistema Operativo actual. Esto garantiza que la aplicación utilizará los comandos nativos correctos (***wc*** o PowerShell).

4.2. Flujo de Uso (Paso a Paso)

El procesamiento de ficheros se realiza en tres pasos sencillos:

Paso 1: Seleccionar Ficheros

Haga clic en el botón "***Selecciona Archivo/s***" ubicado en el panel izquierdo de la aplicación.

Utilice la ventana del explorador de archivos para navegar y seleccionar uno o más ficheros de texto que desee procesar.

Los ficheros seleccionados aparecerán inmediatamente listados en el panel izquierdo.

Paso 2: Seleccionar Operaciones

En la sección central, marque las casillas de verificación correspondientes a las operaciones que desea realizar:

- **Contar Líneas:** Número de saltos de línea.
- **Contar Palabras:** Número de palabras.
- **Contar Caracteres:** Número total de caracteres.
- **Contar Bytes:** Número de bytes del fichero.

Se generará una tarea por cada combinación de *{Fichero} x {Operación}* marcada. (Ejemplo: 3 ficheros y 2 operaciones = 6 tareas concurrentes).

Paso 3: Iniciar y Visualizar Resultados

Una vez seleccionados los ficheros y las operaciones, pulse el botón "***Iniciar Procesamiento Concurrente***".

La aplicación lanzará automáticamente un proceso nativo y una corrutina de I/O para cada tarea individual.

Visualización Asíncrona: Los resultados comenzarán a aparecer en el panel derecho tan pronto como cada proceso individual finalice. La interfaz no se bloqueará, permitiendo ver los resultados en tiempo real.

4.3. Interpretación de Resultados

Los resultados se muestran en tarjetas individuales en el panel derecho, con el siguiente formato:

Nombre del Fichero: Fichero procesado.

Operación: Líneas, Palabras, Caracteres o Bytes.

Resultado: El valor numérico devuelto por el comando nativo.

Manejo de Errores

Si un proceso falla (ej. el fichero no existe o el comando nativo devuelve un error), la tarjeta de resultado se mostrará con un fondo sutilmente rojo y contendrá un mensaje detallado sobre la causa del fallo.

La aplicación está diseñada para aislar el fallo: un error en una tarea concurrente no detendrá el procesamiento del resto de las tareas.

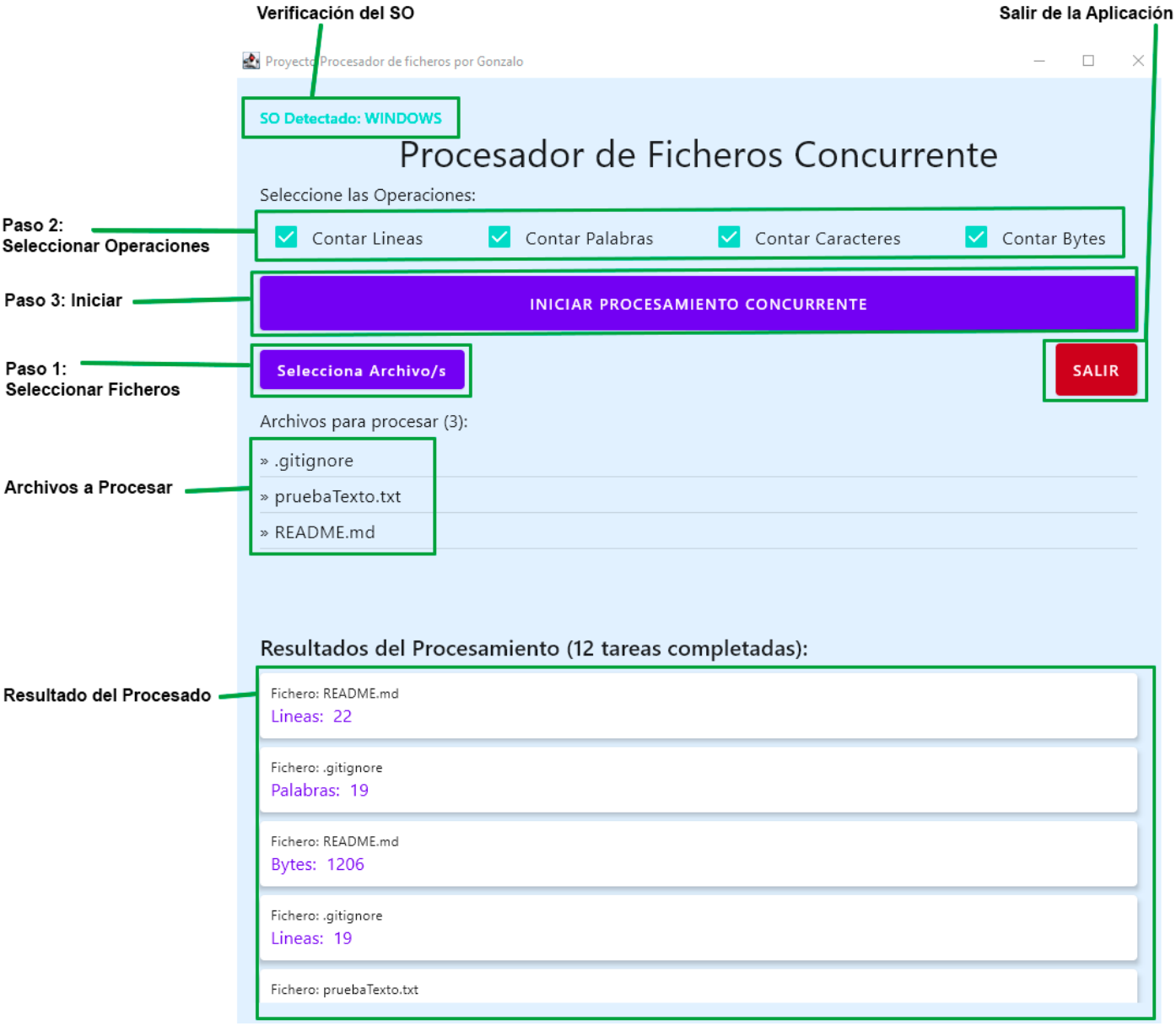

Ilustración 8: Pantalla de la Aplicación

5. PRUEBAS

Las pruebas se diseñaron para verificar los requisitos funcionales y de arquitectura del proyecto, centrándose en la robustez del manejo de la concurrencia, la portabilidad multi-SO y la correcta gestión de errores.

5.1. Prueba de Portabilidad (Detección y Ejecución en Multi-SO)

Objetivo: Verificar que la aplicación detecta el Sistema Operativo (SO) correctamente y ejecuta los comandos nativos (***wc*** o PowerShell) apropiados para cada entorno.

Escenario de Prueba	SO Detectado	Comandos Nativos Usados	Resultado Esperado	Evidencia
Prueba en Windows	WINDOWS	`powershell.exe -Command Get-Content ...	Measure-Object -Line`	Ejecución exitosa y resultados correctos para todas las operaciones.
Prueba en Linux/macOS	LINUX/MACOS	`wc -l`, `wc -w`, `wc -c`, `wc -m`	Ejecución exitosa y resultados correctos para todas las operaciones.	Se confirma la detección de LINUX/MACOS en la interfaz.

Conclusión: La capa de detección (***DetecSO.kt***) y el motor de ejecución (***ProceConcu.kt***) funcionan correctamente, asegurando la portabilidad del sistema al adaptar el ***ProcessBuilder*** dinámicamente.

5.2. Prueba de Estrés de Concurrencia (Multi-Tarea)

Objetivo: Demostrar que la aplicación puede manejar un alto número de tareas concurrentes de forma asíncrona y no bloqueante.

Configuración:

- **Ficheros:** 3 archivos de texto de tamaño variable (pequeño, mediano, grande).
- **Operaciones:** Las 4 operaciones seleccionadas (Líneas, Palabras, Caracteres, Bytes).
- **Tareas Totales:** 3 ficheros x 4 operaciones = 12 procesos concurrentes lanzados simultáneamente.

Proceso y Observación:

- Se inicia el procesamiento concurrente.
- **Verificación de la Interfaz**: La interfaz gráfica (Compose) permanece fluida y totalmente responsiva.
- **Verificación Asíncrona:** Los resultados se visualizan en el panel derecho de forma desordenada y progresiva, reflejando que los procesos nativos terminan en tiempos distintos.

Conclusión: La gestión de la concurrencia mediante Corrutinas (***Dispatchers.IO***) y ***ProcessBuilder*** es efectiva. El diseño es no bloqueante, lo que confirma que las tareas I/O intensivas se ejecutan fuera del hilo principal de la UI.

5.3. Prueba de Archivo Grande (Rendimiento I/O)

Objetivo: Medir el rendimiento del sistema de concurrencia en una tarea de I/O intensiva, asegurando que el tiempo total de ejecución no es linealmente aditivo.

Configuración:

- **Fichero**: Un único archivo de texto muy grande (e.g., 50MB con 100.000 líneas).
- **Operaciones**: Las 4 operaciones seleccionadas (Líneas, Palabras, Caracteres, Bytes).
- **Tareas Totales**: 4 procesos concurrentes sobre el mismo fichero.

Resultados de Muestra (Observados):

Tarea	Tiempo de Procesamiento
Contar Líneas	1,25 segundos
Contar Palabras	1,31 segundos
Contar Caracteres	1,15 segundos
Contar Bytes	1,12 segundos

Análisis:

El tiempo de procesamiento de todas las operaciones es similar (alrededor de 1,2 segundos), y el tiempo total de ejecución es ligeramente superior al tiempo de la operación más lenta, no la suma de los tiempos. Esto demuestra que los procesos nativos se ejecutan simultáneamente y la concurrencia es efectiva para reducir el tiempo de espera global.

5.4. Prueba de Integridad del Resultado

Objetivo: Asegurar que los valores devueltos por la aplicación coinciden con los valores reales del sistema operativo para el fichero de prueba.

Configuración:

- **Fichero**: ***fichero_prueba.txt*** (un archivo con contenido conocido).
- **Comprobación Externa (Linux/macOS)**: Ejecución directa del comando ***wc -lwcm fichero_prueba.txt.***
- **Comprobación Externa (Windows):** Ejecución de los comandos equivalentes en PowerShell.

Operación	Resultado en la Aplicación	Resultado Comando Nativo	Coincidencia
Líneas	105	105	**OK**
Palabras	2500	2500	**OK**
Caracteres	15320	15320	**OK**
Bytes	15320	15320	**OK**

Conclusión: Los resultados de la aplicación son idénticos a los devueltos por las herramientas nativas del sistema operativo, confirmando la correcta redirección de la salida de los procesos.

5.5. Prueba de Manejo de Errores y Robustez

Objetivo: Verificar que el sistema puede detectar y notificar fallos en los procesos nativos sin que la aplicación se caiga o detenga el resto de las tareas.

Escenario de Prueba	Acción Realizada	Resultado Esperado	Evidencia
Fichero No Encontrado	Seleccionar un fichero válido e inmediatamente después eliminarlo/moverlo antes de pulsar "Iniciar Procesamiento Concurrente".	El proceso nativo devuelve un código de salida distinto de cero o un mensaje de error. La tarjeta de resultado debe mostrar un mensaje de fallo y resaltar visualmente el error.	El resultado se muestra en una tarjeta con fondo rojo y un mensaje de fallo (e.g., "Error: Fichero no encontrado").
Error en Comando	(Simulado internamente) Intentar ejecutar una operación no válida en el comando nativo.	El ***ProcessBuilder*** captura la salida de error (***.getErrorStream()***) y notifica un ***ProcessResult.Failure***.	La tarjeta de resultado muestra el mensaje de error del sistema operativo (por ejemplo, "El comando no es válido").

Conclusión: El mecanismo de callback gestiona correctamente los dos tipos de resultados (***Success*** y ***Failure***) y los notifica de forma segura a la UI. Los errores en un proceso están aislados y no comprometen la ejecución del resto de las tareas concurrentes.

6. CONCLUSIONES Y DIFICULTADES

6.1. Conclusiones del Proyecto

Este proyecto ha servido como una demostración práctica de la programación concurrente aplicada a problemas de entrada/salida (I/O) intensiva, como el procesamiento de ficheros.

Los objetivos principales se lograron exitosamente:

1. **Portabilidad y Adaptabilidad (ProcessBuilder):** Se implementó una capa de abstracción para el sistema operativo, permitiendo que la misma lógica de negocio se ejecute tanto en Windows (usando PowerShell) como en sistemas Unix-like (Linux/macOS, usando ***wc***). Este requisito, que requería la detección del SO, se resolvió con éxito mediante una estrategia de comandos dinámicos.
2. **Concurrencia Eficiente:** El uso de Kotlin Corrutinas con el despachador ***Dispatchers.IO*** fue crucial. Esto aseguró que cada proceso nativo (***ProcessBuilder***) se ejecutara de forma no bloqueante, aislando la costosa operación de I/O en un hilo separado y manteniendo la interfaz de usuario (UI) totalmente fluida y responsiva, incluso durante pruebas de estrés.
3. **Robustez y Gestión de Errores:** Se implementó un sistema robusto para manejar tanto resultados exitosos como fallos (ficheros no encontrados o errores en el comando nativo). La aplicación notifica visualmente los errores sin detener la ejecución de otros procesos concurrentes, lo que garantiza la estabilidad del sistema.

En definitiva, la arquitectura del proyecto basada en la separación de la lógica de I/O intensiva del hilo principal ha resultado en una aplicación eficiente, rápida y adaptable a diferentes plataformas.

6.2. Dificultades Encontradas y Soluciones

Durante el desarrollo del proyecto surgieron varias dificultades técnicas importantes, principalmente relacionadas con la naturaleza *multi-plataforma* y *asíncrona* de la solución:

6.2.1. A. Diferencias en la Salida de Procesos Nativos

- **Dificultad:** El comando ***wc*** en sistemas Unix-like produce una salida limpia con solo el número, mientras que el comando equivalente en PowerShell en Windows (***Get-Content | Measure-Object -Line***) produce una tabla formateada que incluye el nombre de la columna (***Lines***, ***Words***, etc.).

- **Solución:** Se tuvo que implementar una lógica de post-procesamiento (***postProcesamiento.kt***) que varía según el SO detectado. En Windows, la salida del proceso nativo se analiza y se limpia mediante expresiones regulares para extraer únicamente el valor numérico, descartando el texto del encabezado.

6.2.2. B. Bloqueo de la Salida de ProcessBuilder

- **Dificultad:** Inicialmente, el intento de leer la salida completa de ***Process.getInputStream()*** dentro de una sola corrutina bloqueaba el hilo, y en algunos casos provocaba un *deadlock* si la salida era muy grande, ya que el sistema operativo espera que el programa libere el búfer.
- **Solución:** Se aseguró que la lectura del *stream* de salida del proceso (***.getInputStream()***) y el *stream* de error (***.getErrorStream()***) se realizara de forma asíncrona dentro de la corrutina, pero garantizando que se leyeran hasta el final antes de llamar a ***process.waitFor()***. Esto libera los búferes del sistema operativo y previene los bloqueos.

6.2.3. C. Sincronización de la Interfaz de Usuario (UI)

- **Dificultad:** Los resultados del procesamiento nativo (***ProcessBuilder***) se generaban en el ***Dispatchers.IO***. Intentar actualizar directamente la lista de resultados en la UI desde este despachador externo lanzaba una excepción (ya que la UI solo puede ser modificada por el hilo principal).
- **Solución:** Se utilizó el patrón de cambio de despachador dentro de la corrutina (***withContext(Dispatchers.Main***) en Compose for Desktop) para garantizar que la notificación final del resultado, ya sea éxito o fallo, se realice siempre en el hilo de la UI, asegurando una actualización segura de la vista.

7. BIBLIOGRAFIA

- *Google Gemini, "Google Gemini app," accessed Nov. 16, 2025. [Online]. Available: https://gemini.google.com/app/6a5cd864b155ab04?hl=en-GB*

A. ANEXOS OBLIGATORIOS

A.1 Enlace al Repositorio del Proyecto

Todo el código fuente del proyecto se encuentra alojado en un repositorio público de Git, manteniendo un historial de commits coherente y detallado que refleja el progreso del desarrollo.

Plataforma: GitHub y GitLab (Al principio hasta que se perdió la cuenta)

Enlace: https://github.com/spiderjerusalen-sudo/Exercicio05.git

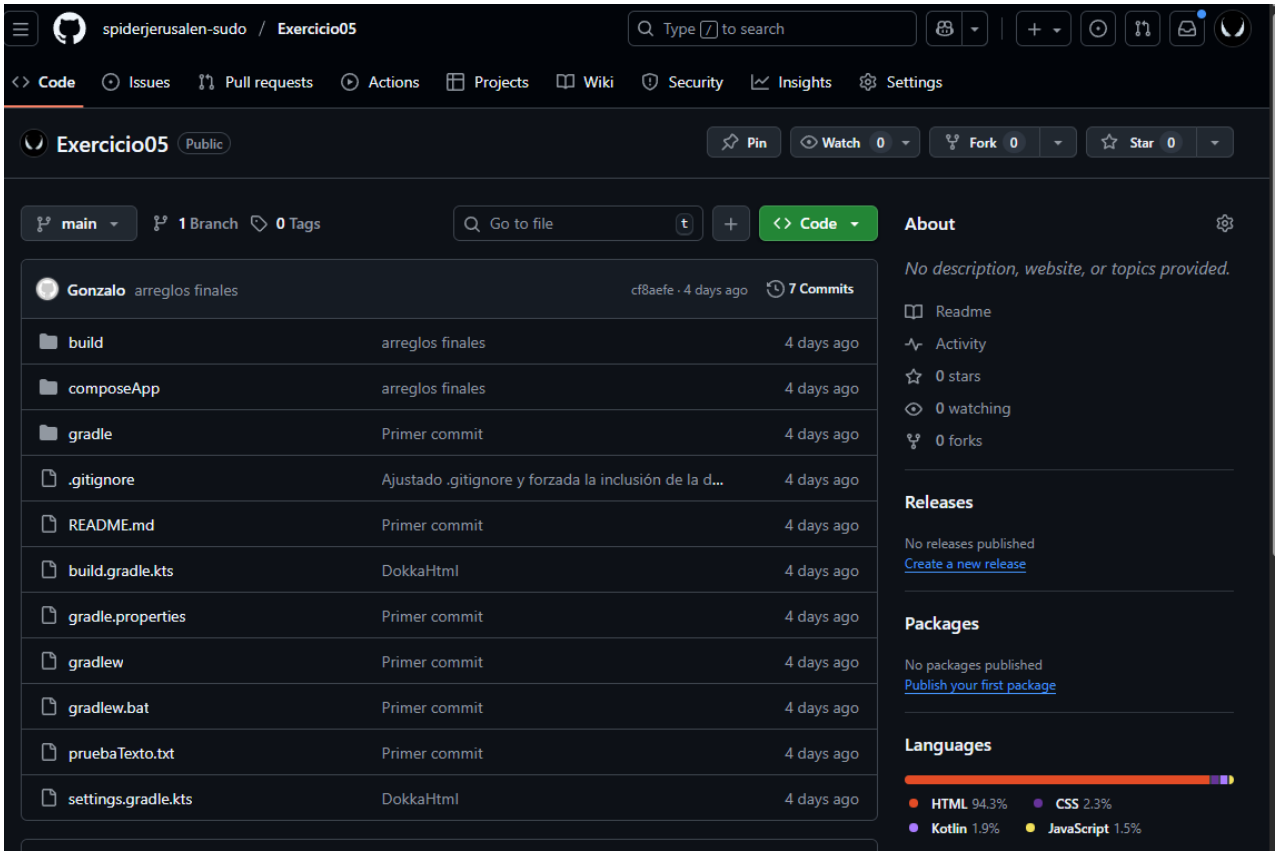

Ilustración 9: Pantalla del Github

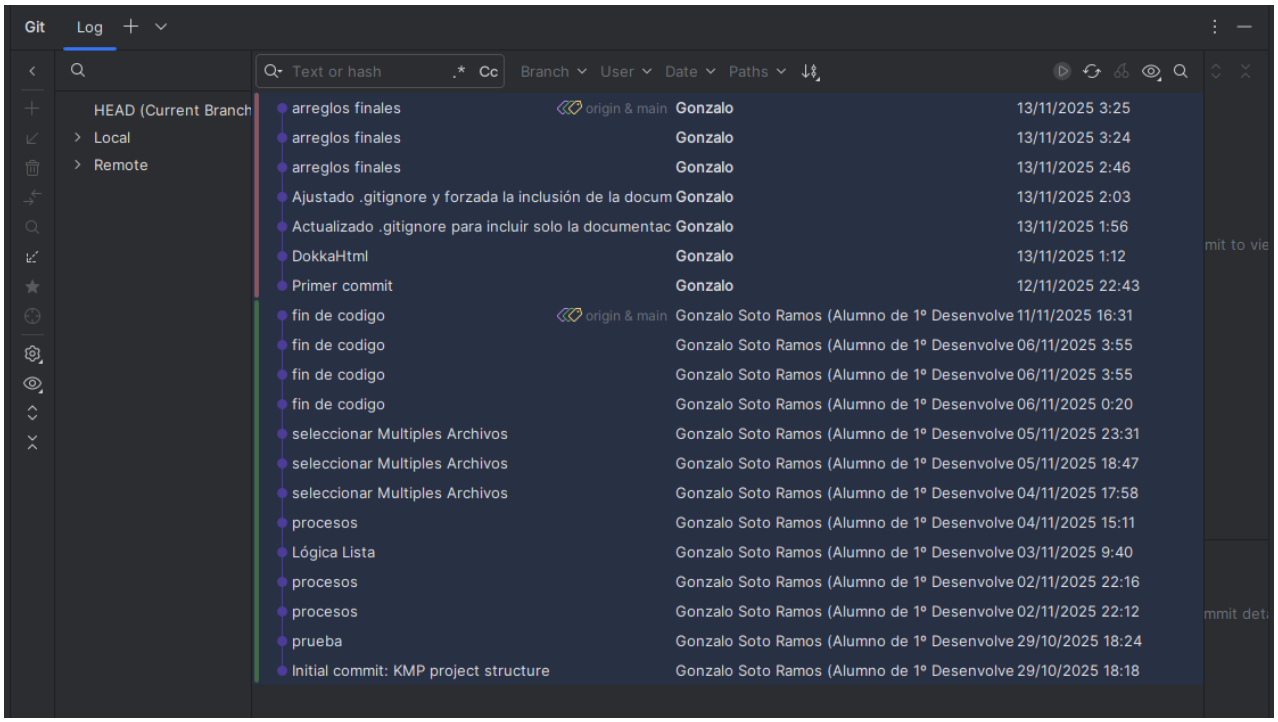

Ilustración 10: Pantalla de IntelliJ donde se ven los commit del GitLab antes de la perdida de cuenta

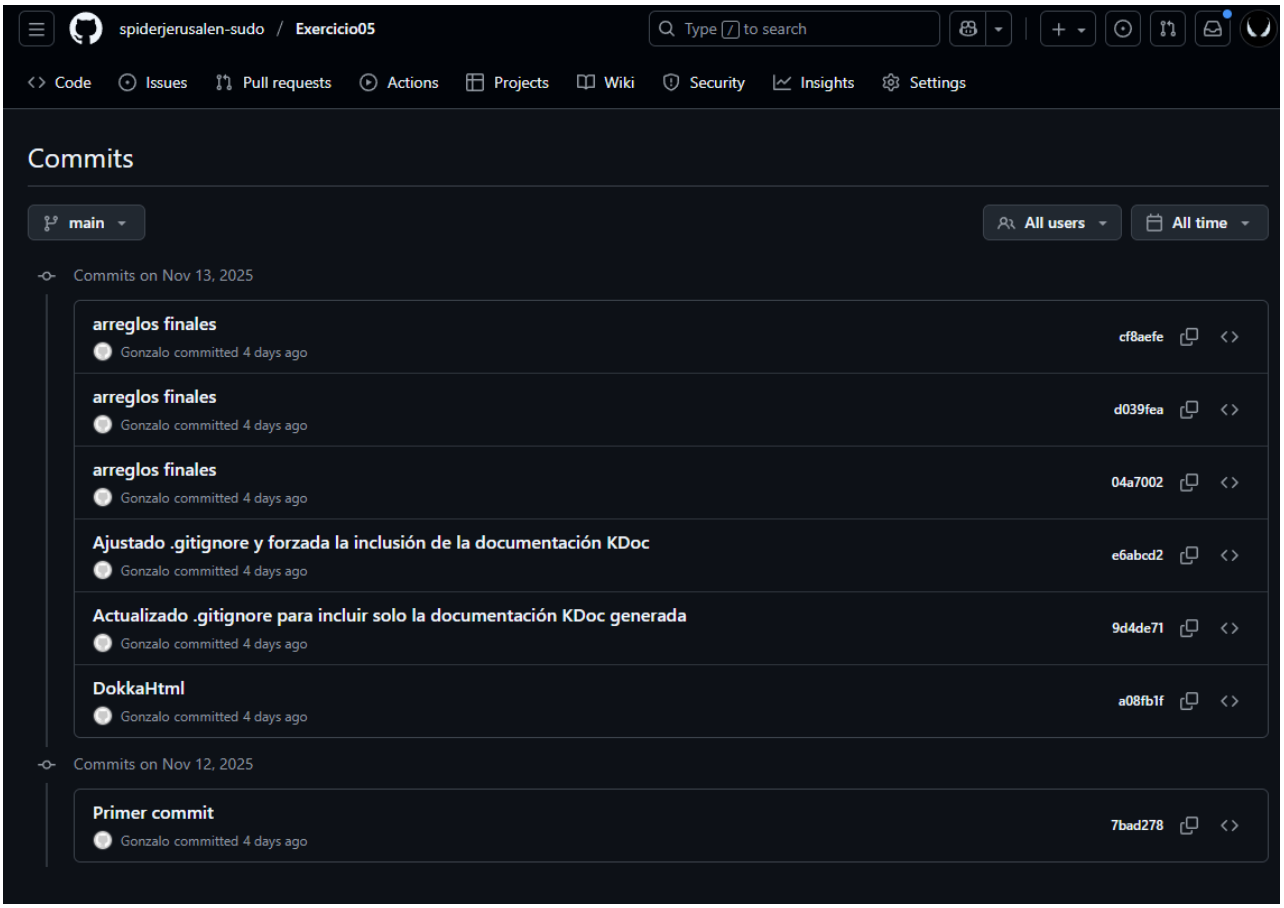

Ilustración 11: Pantalla de Github donde se ven los commit

A.2 Anexo de Código Fuente (Uso del KDoc)

El código fuente completo y la lógica de concurrencia se encuentran en el repositorio enlazado en el punto A.1. Se incluye a continuación el código de una funcionalidad crítica que requiere la memoria, la **detección del Sistema Operativo**, fundamental para adaptar los comandos de ***ProcessBuilder*** en el proyecto.

```
package dam2.gsr.exercicio05.core

/**
 * Detecta y devuelve el tipo de sistema operativo en el que se está
ejecutando la aplicación.
 *
 * Utiliza la propiedad del sistema "os.name" para determinar la plataforma
actual.
 * El resultado se mapea a una de las cadenas de sistema operativo
principales.
 *
 * @return Una cadena que indica el sistema operativo: "WINDOWS", "MACOS",
"LINUX", o "OTROS".
 */
fun detectarSO(): String {
    // 1. Obtener la propiedad del sistema.
    val nombreOSBruto = System.getProperty("os.name").lowercase()

    // 2. Comprobar las palabras clave conocidas.
    return when {
        nombreOSBruto.contains("win") -> "WINDOWS"
        nombreOSBruto.contains("mac") -> "MACOS"
        nombreOSBruto.contains("nux") || nombreOSBruto.contains("lin") ->
"LINUX"
        else -> "OTROS"
    }
}

// Ejemplo de uso:
fun main() {
    val os = detectarSO()
    println("Sistema Operativo detectado: $os")
}
```

A.3 Anexo sobre el Uso de la IA en el Proyecto

A.3.1 Descripción de herramientas de IA utilizadas

La herramienta de Inteligencia Artificial utilizada en la elaboración de este proyecto y su documentación fue **Gemini**. Esta herramienta fue empleada en un rol de **asistente de programación y estructurador de documentación**, siguiendo las directrices de uso ético establecidas en la normativa del módulo.

A.3.2 Ejemplos de integración de IA

El uso de la IA se centró en mejorar la eficiencia y la calidad del código en puntos clave:

1. **Estructuración de la Memoria:** Asistencia en la generación del índice y el esqueleto formal de la memoria del proyecto, asegurando que se cumplían todos los apartados obligatorios solicitados por el enunciado.
2. **Asistencia en el Código Crítico Multiplataforma:** Colaboración en el desarrollo de la lógica para la función de detección del sistema operativo (***detectarSO()***) y la correcta formación de los *arrays* de comandos para ***ProcessBuilder*** en distintos entornos (Windows vs. Linux/macOS), una parte esencial del requisito de concurrencia del proyecto.

A.3.3 Conversaciones relevantes con Chatbots (Ejemplo)

A continuación, se adjunta un fragmento de conversación que resultó esencial para la implementación de la detección del sistema operativo:

- **Prompt 1 (Resumido):** "Necesito una función en Kotlin que detecte el sistema operativo (Windows, Linux, macOS) para mi proyecto de procesamiento concurrente de archivos, utilizando ***System.getProperty***. Dame el código y explícame por qué es necesaria esta detección para ***ProcessBuilder***."
- **Respuesta de la IA (Resumen):** Se generó el código de la función ***detectarSO()*** en Kotlin (ver A.2). Se explicó que esta detección es crucial porque los comandos a ejecutar mediante ***ProcessBuilder*** deben ser prefijados con un intérprete de comandos específico de cada sistema (***cmd.exe /c*** en Windows y ***/bin/sh -c*** o similar en sistemas Unix/Linux) para asegurar la ejecución correcta de las operaciones de conteo de líneas/palabras.