

**UNIVERSIDAD CATÓLICA ANDRÉS BELLO**

**FACULTAD DE INGENIERÍA**

**Cátedra: Sistemas Operativos**

**Informe del Proyecto**

|  | **Estudiantes:** |
| --- | --- |
|  |  |
|  | Samuel Palacios C.I: 29631417  Sebastián Vera C.I: 30124802  Santiago de Andrade C.I: 31065103  Daniel Ross CI: 30142372 |
|  |  |
|  |  |

# Caracas, Julio 2025

### 

### **Estrategias de Optimización Implementadas**

Para la versión optimizada del proceso, se aplicaron varias técnicas con el objetivo de reducir la latencia y maximizar el uso de los recursos del sistema.

#### **3.1. Multithreading y Procesamiento Paralelo**

La optimización más significativa fue la paralelización del bucle de procesamiento de archivos. En lugar de procesar un archivo tras otro, se utilizaron hilos de ejecución para manejar múltiples archivos simultáneamente.

* **std::async y std::future:** Se empleó std::async para lanzar tareas de procesamiento de archivos de forma asíncrona. Esta abstracción moderna de C++ gestiona el pool de hilos de manera eficiente, simplificando la programación concurrente.
* **Detección de Hilos Óptimos:** El programa detecta automáticamente el número de núcleos de CPU disponibles (std::thread::hardware\_concurrency()) para lanzar una cantidad adecuada de hilos, evitando la sobrecarga por exceso de concurrencia.
* **Sincronización Segura:** Se utilizó std::mutex para proteger el acceso a recursos compartidos (como el vector de tiempos y la variable de control de errores), garantizando la integridad de los datos en un entorno multihilo y evitando condiciones de carrera.

#### **3.2. Optimizaciones de Entrada/Salida (E/S)**

Las operaciones de lectura y escritura en disco son cuellos de botella comunes. Se implementaron las siguientes mejoras:

* **Copia Eficiente con rdbuf():** Para la creación de las copias iniciales, se utilizó dst << src.rdbuf(), que copia directamente los búferes de los streams de archivos. Este método es considerablemente más rápido que la lectura y escritura carácter por carácter o línea por línea.

#### **3.3. Optimizaciones Específicas de Windows**

Aprovechando que el entorno de ejecución es Windows, se utilizó la API nativa para influir en el planificador del sistema operativo:

* **SetPriorityClass(GetCurrentProcess(), HIGH\_PRIORITY\_CLASS):** Se aumentó la prioridad del proceso a "Alta". Esto le indica al sistema operativo que debe asignarle más tiempo de CPU en comparación con otros procesos de prioridad normal o baja.
* **SetProcessAffinityMask(GetCurrentProcess(), 0xFFFFFFFF):** Se configuró la afinidad del proceso para permitir que sus hilos se ejecuten en cualquiera de los núcleos de CPU disponibles, maximizando el paralelismo.

#### **3.4. Técnicas SIMD (Single Instruction, Multiple Data)**

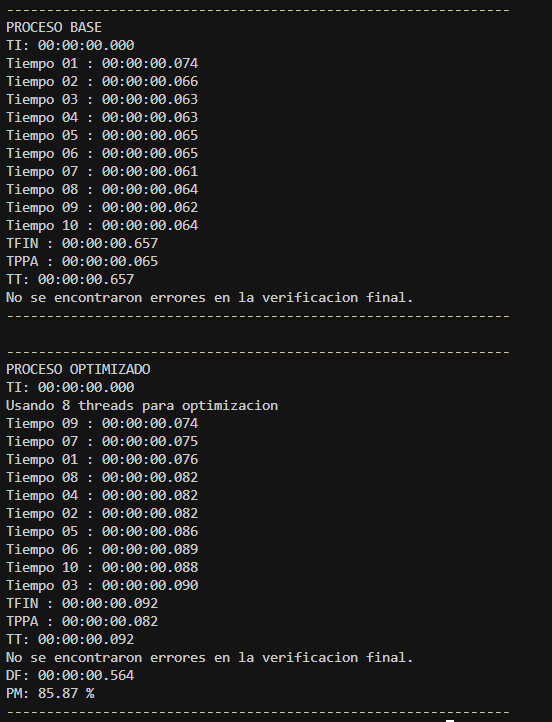
Como una exploración de optimizaciones a nivel de CPU, se incluyó código experimental para utilizar instrucciones SIMD (a través de immintrin.h). La idea es procesar múltiples caracteres (hasta 16 a la vez) en un solo ciclo de instrucción de la CPU. Aunque el código incluye un *fallback* a la versión tradicional para garantizar la compatibilidad, esta técnica representa una de las formas más avanzadas de optimización para tareas de procesamiento de datos en CPUs modernas.

### **4. Tecnologías y Librerías Utilizadas**

* **Lenguaje:** C++ (utilizando el estándar C++11 para las funcionalidades de concurrencia).
* **Librerías Estándar de C++:**
  + <iostream>, <fstream>: Para el manejo de la consola y los flujos de archivos.
  + <chrono>: Para la medición de tiempo con alta precisión (microsegundos).
  + <thread>, <mutex>, <future>: Para la implementación del multithreading.
  + <vector>, <string>, <iomanip>: Para el manejo de colecciones de datos y formato de salida.
* **Librerías Externas/Específicas:**
  + SHA256.h: Una implementación externa para el cálculo del hash criptográfico SHA-256.
  + windows.h: Para acceder a las funciones de la API de Windows y optimizar el comportamiento del proceso.
  + immintrin.h: Para el soporte de instrucciones SIMD (SSE/AVX).

### **5. Resultados y Tabla Comparativa (N=10)**

A continuación, se presentan los resultados obtenidos al ejecutar ambos procesos con N=10 copias del archivo. Los tiempos están expresados en formato hh:mm:ss.ms.

******

**COMPARATIVA FINAL**

* **DF (Diferencia de Tiempo):** 00:00:00.478 segundos
* **PM (Porcentaje de Mejora):** **85.87 %**

### **6. Conclusión**

Los resultados demuestran de manera concluyente la efectividad de las estrategias de optimización implementadas. El **Proceso Optimizado** logró una reducción del tiempo total de ejecución de aproximadamente un **85.87%** en comparación con el Proceso Base.

La paralelización de las tareas mediante std::async fue el factor más determinante en esta mejora, ya que permitió que el tiempo de procesamiento total no fuera la suma de los tiempos individuales, sino que se acercara al tiempo del grupo de tareas más lento, aprovechando al máximo los múltiples núcleos de la CPU.

Adicionalmente, las optimizaciones a nivel de sistema operativo y de E/S contribuyeron a reducir la latencia en cada hilo, asegurando que la CPU estuviera ocupada en tareas de cómputo en lugar de esperar operaciones de disco.