

./logo-UNA blanco.png

# **Compresor distribuido y descompresor (secuencial)**

Proyecto # 2

Jorge Durán Campos  
Luis Antonio Montes de Oca Ruiz  
Diego Quirós Artiñano

EIF-212 Sistemas Operativos  
Profesor Eddy Ramirez Jiménez

14 de junio de 2023

---

## ÍNDICE

---

I.	Resumen Ejecutivo . . . . .	1
II.	Introducción . . . . .	2
III.	Marco teórico . . . . .	2
IV.	Descripción de la solución . . . . .	3
V.	Resultados de pruebas . . . . .	5
VI.	Conclusiones . . . . .	9
VII.	Aprendizajes . . . . .	9

## I. RESUMEN EJECUTIVO

El presente proyecto trata sobre un problema de compresión y descompresión de archivos usando el algoritmo de Huffman e hilos lectores. Estos hilos lectores van a leer un archivo byte por byte contando las repeticiones de cada byte y colocándola dentro de un array de solución llamado `solution_array`, en una posición `i`, donde `i` representa el byte, es decir si el byte 254 se repite 3 veces, en la posición del 254 del arreglo va a estar el entero 3. Para poder solucionar el problema planteado se implementaron varios mecanismos como los MUTEX (mutual exclusion) [?], algoritmo de Huffman[?][?], y threads, los cuales componen el objetivo principal del proyecto, el cual es aprender el uso del algoritmo.[?] [?] [?] [?]

Esta solución se implementa por medio de distintos métodos, uno para leer el archivo deseado byte por byte por cada hilo lector y que se coloque la cantidad de repeticiones en el arreglo de solución, como se mencionó anteriormente, además se usan métodos para la creación de un árbol binario y sus códigos respectivos, usando el algoritmo de Huffman, y el uso del algoritmo de ordenamiento radix sort[?] tomando en consideración la recomendación del profesor en el proyecto anterior. Finalmente, cada uno de estos métodos se llaman dentro del main, junto con las verificaciones necesarias y la ejecución y unión de los hilos lectores, y se termina con la creación del archivo comprimido, la impresión en un archivo del árbol resultante, sus códigos, y luego en otro código se realiza la descompresión haciendo uso de los códigos del archivo comprimido haciendo uso del algoritmo de Huffman. Estos métodos se mencionan a continuación con una descripción más detallada en la sección de descripción de la solución.

Estos métodos además se dividen en dos proyectos, el proyecto 2 que es el compresor y el proyecto 3 que sirve de descompresor. Dentro del proyecto 2 se encuentran los métodos del radix sort. Primero se tienen dos métodos auxiliares `get_max()` y `count_sort()` y los dos métodos se unen en uno solo llamado `radix_sort()`. También se tiene el método para leer el archivo llamado `reading_file()` junto con las verificaciones correspondientes. Luego se tiene la creación de un struct para los nodos del árbol de Huffman así como los métodos para la creación de este árbol, los cuales son `create_node()`, `create_node_not_byte()`, `amount_of_zeroes()` y `create_tree()`, así mismo los métodos para pasar este árbol a un archivo con sus datos (altura, anchura, frecuencia, entre otros), los cuales son `is_leaf()`, `print_tree()`, `get_tree_height()`, `get_tree_widths()`, `get_max1()`, `get_tree_frecuencias()`, `get_node_amount()` y `write_tree_data()`. Luego se tiene el método para la creación de tabla de códigos `fill_path()` y el método para imprimir esta tabla en un archivo `write_table_file()`. Finalmente está el método encargado de la compresión llamado `compression()`.

Después, el proyecto 3 es el encargado de la descompresión del archivo. Aquí se crea un árbol basado en el archivo del árbol que se generó al comprimir un archivo, existe un struct y varios métodos encargados de hacer este árbol, los cuales son los siguientes, el struct `node` y los métodos `create_node()`, `create_node_not_byte()`, `is_leaf()`, `create_tree()` y `get_tree_data()`, este último método es el encargado de tomar el archivo del árbol y generarlo de nuevo para poder descomprimir el archivo, haciendo uso de los códigos (rutas) asignados a cada hoja que representa un carácter del archivo original en ese mismo árbol[?] [?]. Por último, se tiene el método encargado de la descompresión del archivo llamado `decompression()`.

Finalmente, se realizaron varias pruebas de lectura con 6 archivos de distintos tamaños y de distintos tipos. Se ejecuto la compresion 3 veces para obtener los distintos tiempos. De estas se obtuvo la conclusión

## II. INTRODUCCIÓN

Este documento tiene como objetivo servir como documentación del segundo proyecto del curso de sistemas operativos. Este consta de un resumen ejecutivo en el cual se explica brevemente el proyecto junto a su solución y resultados, un marco teórico donde se describen aspectos relacionados al proyecto, como el lenguaje, las bibliotecas y una demostración del algoritmo de Huffman, una descripción de la solución implementada al problema planteado en el enunciado del proyecto, los resultados de las pruebas de la solución, las conclusiones donde se analizan los resultados y el proyecto en general, y por último se presentan los aprendizajes obtenidos al realizar el proyecto. Además, el proyecto se basa en realizar un compresor distribuido de cualquier tipo de archivo, sea un video, un texto o demás, este archivo es leído de manera binaria usando hilos que van byte por byte del archivo, este va a ir contando la cantidad de repeticiones de un byte específico de los 256 bytes que existen y así asignarles una ruta compuesta de unos y ceros, usando el algoritmo de Huffman. Además, también se implementa un descompresor, el cual utilizara la tabla de códigos del archivo a descomprimir para obtener el archivo en su forma original. Se entregarán tres archivos distintos, uno siendo los datos comprimidos, otro con los datos del árbol y otro que contiene la tabla de códigos de Huffman creados al comprimir el archivo y que se utiliza al descomprimir dicho archivo.

## III. MARCO TEÓRICO

El lenguaje utilizado en este proyecto es C, el cual se creó a principios de los años 1970 por Dennis M. Ritchie en Bell Laboratories. Este fue diseñado como un lenguaje minimalista para la creación de sistemas operativos para minicomputadoras, las cuales eran computadoras más baratas y menos potentes que una supercomputadora, pero más caras y potentes que una computadora personal. El principal motivo fue el deseo de migrar el kernel del sistema pronto a ser terminado, UNIX, a un lenguaje de alto nivel, teniendo las mismas funciones, pero con menos líneas de código. C se basó en CPL, o Combined Programming Language, por sus siglas en inglés, el cual a su vez sirvió de base para el lenguaje B. De este, Ritchie reescribió varias funciones de CPL para crear C y después

reescribió UNIX en este nuevo lenguaje.[?] [?]

Desde 1977 hasta 1979 ocurrieron distintos cambios en el lenguaje, y durante este tiempo se publicó un libro que sirve como manual para el lenguaje, titulado *The C Programming Language*, publicado en 1978 por Ritchie y Brian W. Kernighan. Cinco años después, se estandarizó C en el American National Standards Institute, y desde ese momento, al lenguaje se le refiere como ANSI Standard C. De C salieron varios lenguajes derivados, tales como Objective C y C++. Además, también surgió Java, el cual se creó como un lenguaje que simplifica C.[?]

Continuando con los recursos utilizados en este proyecto, se utilizó la librería pthread. Esta es una librería de POSIX la cual es un estándar para el uso de hilos (threads), en C y C++, los cuales permiten un flujo de procesos de manera concurrente. El uso de estos hilos es más efectivo en procesadores con múltiples núcleos, ya que se pueden asignar los procesos a distintos núcleos, haciendo la ejecución más rápida.[?]

Luego, un recurso esencial para la solución del proyecto es el algoritmo de Huffman. Este algoritmo fue presentado y descrito por David Huffman en 1952 en la publicación titulada *A Method for the Construction of Minimum-Redundancy Codes*. Este algoritmo se utiliza para la compresión de archivos, haciendo uso de códigos binarios, que se le asignan a cada carácter según su repetición dentro del archivo. Estos códigos se asignan haciendo uso de un árbol binario, donde sus hojas son los caracteres que menos repetición tienen, y los nodos más cercanos a la raíz tienen mayor repetición, al recorrer este árbol se asignan 1 a las aristas de la derecha de un nodo y 0 a las de la izquierda, siendo el código el resultado del recorrido para llegar a un nodo en específico desde la raíz. Como los nodos que tienen mayor repetición se encuentran más cerca a la raíz, su camino es más corto y por ende su código también. Estos códigos se utilizan para reemplazar cada carácter dentro del archivo, y aunque existan varios caracteres de baja repetición, estos teniendo códigos largos, se compensa con los códigos mucho más pequeños de los caracteres con mayor repetición. Finalmente, así logrando comprimir un archivo, ya que muchos códigos

quedan más pequeños que el código original del carácter, estos siendo de 8 bits.[?][?]

Además, se escogió CLion como el IDE preferido para este proyecto. Este IDE es de la empresa JetBrains y fue lanzado al mercado en 2014[?] con herramientas que ayudan a la creación de código, tales como refactoring, generación de código para sets/gets, terminación de código y arreglos rápidos del código escrito.[?]

Finalmente, ya que se mencionó tanto POSIX como UNIX se va a brindar información sobre estos. Portable Operating System Interface for UNIX, o POSIX, son estándares establecidos por la IEEE y publicados por la ANSI e ISO (International Organization for Standardization), estos estándares permiten el desarrollo de código universal, para que pueda correr en todos los sistemas operativos que implementen POSIX, tales como macOS o Ubuntu, la mayoría de los sistemas basados en UNIX cumplen con POSIX.[?] UNIX es un sistema operativo que fue creado para brindar a programadores funciones simples pero potentes, y que permitiera el uso de múltiples usuarios y de multi tarea, este se compone de tres partes, el kernel, los archivos de configuración de sistema y los programas.[?]

#### IV. DESCRIPCIÓN DE LA SOLUCIÓN

El primer paso de la solución fue crear las distintas variables y definiciones para el programa del compresor. Primero se crea un MUTEX que se va a utilizar para el arreglo de repeticiones de bits, el cual tiene 256 posiciones[?] [?]. Además, existen dos arreglos, uno llamado *solution\_array* que tiene 256 posiciones, las cuales representan los 256 bytes existentes, aquí se almacena la cantidad de repeticiones de un byte dado por la posición del arreglo, y el otro, llamado *solution\_aux*, tiene la misma cantidad de espacios y se utiliza como un auxiliar que va a contener en sus posiciones el número del byte, es decir en la posición 0 se coloca el número 0, para que cuando se ordene el arreglo principal, este se ordena al mismo tiempo y así se pueda identificar el byte con su repetición correspondiente. Luego, se crean varias variables, la primera llamada *readers\_num* se utiliza para definir la cantidad de hilos lectores por parte del usuario, la segunda, *filepath*, se utiliza como puntero

a la dirección donde se encuentra el archivo a comprimir, la tercera es *filelen* que indica el tamaño de dicho archivo, y luego se encuentran las variables para los nombres de los archivos, está *filename* que se usa para darle nombre al archivo comprimido, *filename\_with\_ext\_data* para el nombre del archivo del árbol, *filename\_with\_ext\_table* para la tabla de códigos y *filename\_with\_ext\_comp* para el nombre del archivo comprimido incluyendo la extensión solicitada por el profesor. Finalmente, se declara una tabla donde se almacena el camino resultante para cada carácter del archivo con el fin de poder usarla para descomprimir el archivo, llamada *path\_table*, esta matriz tiene 256 filas y dos columnas, donde cada fila son en representación de los 256 bytes que existen y las columnas para contener el byte y su código (ruta) correspondiente.

Se realiza un radix sort[?] en un método llamado *radix\_sort()* para poder ordenar los resultados de mayor a menor repeticiones por cada byte, este sort hace uso de dos métodos *get\_max()* y *count\_sort()*. El método *get\_max()* se encarga de devolver el número más grande del arreglo donde se guardan las repeticiones de cada byte. Luego este valor se usa dentro de *count\_sort()* el cual es el que contiene toda la lógica de ordenamiento, aquí se ordena primero por unidades, luego las decenas, y así hasta llegar al límite dado por *get\_max()*.

Se crea el método *reading\_file()* el cual es el que los hilos lectores se encargan de ejecutar. Este va a abrir el archivo indicado en la dirección dada en la ejecución del programa, así como determinar desde donde empezar y terminar la lectura tomando en cuenta la cantidad de hilos lectores determinados por el usuario.[?] [?] [?] Por mientras que no se haya llegado al final del archivo y la posición de lectura sea menor a la posición de lectura final, se realiza un bloqueo del mutex para el array de solución, para que ningún otro hilo lector toque este array, luego el hilo que está trabajando sube la cantidad de repeticiones del byte leído dentro del array de solución, y se desbloquea esta posición dentro del array para asegurarse que los demás hilos puedan lograr su trabajo, seguidamente la posición de lectura se aumenta en uno, y por último se hace que los hilos terminen su trabajo.[?]

Luego, se hace la creación de un struct llamado

nodo, el cual se usa para los nodos del árbol, este contiene el byte junto con sus repeticiones y su código, así como un apuntador a su nodo hijo de la izquierda y derecha. Seguidamente, se tienen dos métodos *create\_node()* y *create\_node\_not\_byte()*, que se usan para la creación de los nodos, el primero para los nodos que si contienen un byte en específico y el otro para los nodos que solo tienen la suma de la repeticiones o valor de un par de nodos juntos. Después está el método *amount\_of\_zeros()* que devuelve la cantidad de bytes que no aparecen nunca en el documento y por ende tienen un valor 0 en el array de solución. Y también está el método para la creación del árbol que es *create\_tree()* basándose en el algoritmo de Huffman y tomando el array de solución, aquí se hace toda la lógica de comparación entre la cantidad de repeticiones y así colocar los bytes en nodos con sus posiciones correspondientes.

Están después los métodos para la creación de archivos del árbol. Primero está *is\_leaf()* que se usa para determinar si el nodo que se acaba de visitar es una hoja. Luego el método *print\_tree()* es el que toma el árbol generado con el método *create\_tree()* y lo guarda en un archivo, *get\_tree\_height()* y *get\_tree\_widths()* que se utilizan para determinar la altura y anchura del árbol respectivamente y colocarla como dato dentro del archivo que contiene el árbol junto con los datos del carácter de mayor repetición usando *get\_maxl()*, *get\_tree\_frecuencias()* que genera una línea con la repetición de cada carácter y *get\_node\_amount()* para incluir la cantidad de nodos en total, el cual se coloca usando el método *write\_tree\_data()*, y el último método es *fill\_path()* que genera la ruta para cada una de las hojas del árbol.[?][?][?]

Así mismo, se tienen los métodos para la creación de la tabla de códigos y su archivo correspondiente. *fill\_path()* es el encargado de llenar la tabla *path\_table* con todos los códigos para cada uno de los caracteres del archivo, donde recursivamente se va llenando la tabla recorriendo el árbol y haciendo uso de una verificación si el nodo visitado es una hoja, usando el método *is\_leaf()*, y conforme se va moviendo a través del árbol, asigna un 1 a la ruta si se movió a la izquierda y 0 hacia la derecha, y después el método *write\_table\_file()* agarra esta

tabla y la imprime en un archivo.

Para terminar, dentro del main primero se inicializan todas las posiciones del array de solución con un número que es igual que su posición, junto con los mutex para las 256 posiciones del array de solución. Luego se realizan verificaciones para revisar si se da el número de hilos lectores deseados, la dirección del archivo a leer o el nombre deseado para el archivo final, así como una apertura preliminar del archivo y se verifica si se logró abrir, si es exitosa la apertura se toma la longitud del archivo, se hace un rewind del puntero lector y por último se cierra el archivo. Luego se hace la creación de los hilos lectores, tomando en cuenta la cantidad establecida por el usuario con su asignación de la tarea del método *reading\_file()*, si falla se detiene el programa. Por último, se hace join al hilo principal del programa a los lectores, e igual si falla se detiene el programa.[?] Seguidamente, se ejecuta el *radix\_sort()* para ordenar el solution array, y luego se destruyen todos los mutex para el array de solución, se crea el archivo del árbol usando el metodo *write\_tree\_data()*, se llena la tabla con los caminos del árbol y se imprime en el archivo correspondiente y por último se ejecuta el método *compression()* para realizar la compresión del archivo haciendo uso del árbol y sus caminos para cada hoja.

Posteriormente, para la creación del programa descompresor de archivos, se declaran varias variables globales que se utilizan en varias partes del programa, como nombres de archivo, longitudes de archivo, contadores y matrices de bytes, entre otros, los cuales se detallarán a continuación.

En la variable *filename* se almacena el nombre del archivo comprimido pasado como argumento de línea de comandos. A su vez, en *filename\_with\_ext\_data* se almacena la ruta de archivo completa para el archivo de datos comprimidos, y se forma concatenando *filename* con la extensión ".edy". Las variables *filename\_with\_ext\_comp* y *filename\_with\_ext\_table* son similares, excepto que concatenan la extensión ".una" y ".table", respectivamente.

Por su parte, *filename\_decomp* almacena el nombre del archivo descomprimido pasado como argumento de línea de comandos, mientras que

*ext\_decomp* almacena la extensión del archivo descomprimido pasado como argumento de línea de comandos. La variable *filelen* almacena la longitud del archivo original antes de la compresión. Además, la variable *bytes\_counter* almacena la cantidad de bytes diferentes en el archivo comprimido.

*non\_byte\_nodes\_amount* almacena la cantidad de nodos que no son bytes en el árbol de compresión, mientras que *bytes* Es una matriz de tamaño *bytes\_counter* que almacena los bytes y sus frecuencias. Cada fila contiene un byte y su frecuencia.

*struct Node* representa un nodo en el árbol de compresión. Tiene varios campos, como *byte* (almacena el valor del byte), *frequency* (almacena la frecuencia del byte), *left* (puntero al nodo hijo izquierdo) y *right* (puntero al nodo hijo derecho).

*create\_node()* es una función que crea un nodo en el árbol de compresión y devuelve un puntero a ese nodo. Toma como argumentos el byte y su frecuencia.

*create\_node\_not\_byte()* es una función que crea un nodo en el árbol de compresión que no representa un byte. Toma como argumento la frecuencia del nodo y devuelve un puntero a ese nodo.

*is\_leaf()* es una función que verifica si un nodo dado es una hoja en el árbol de compresión. Devuelve 1 si es una hoja y 0 si no lo es.

*create\_tree()* Es una función que construye el árbol de compresión utilizando los bytes y sus frecuencias cargados. Utiliza el algoritmo de Huffman para combinar nodos y crear un árbol óptimo. Devuelve un puntero al nodo raíz del árbol.

*get\_tree\_data()* es una función que lee los datos necesarios para construir el árbol de compresión desde el archivo de datos comprimidos. Lee información como la cantidad de bytes, las frecuencias de los bytes y la cantidad de nodos que no son bytes.

*decompression()* es una función que realiza la descompresión del archivo utilizando el árbol de compresión. Recorre el archivo comprimido bit a bit, siguiendo el camino correcto en el árbol para cada bit. Escribe los bytes descomprimidos en el archivo de salida.

Finalmente, *main()* es la función principal del programa. Se encarga de procesar los argumentos de línea de comandos, establecer los nombres de archivo y extensiones, llamar a *get\_tree\_data()* para obtener los datos necesarios del archivo comprimido, crear el árbol de compresión utilizando

*create\_tree()*, y finalmente llamar a *decompression()* para descomprimir el archivo.

## V. RESULTADOS DE PRUEBAS

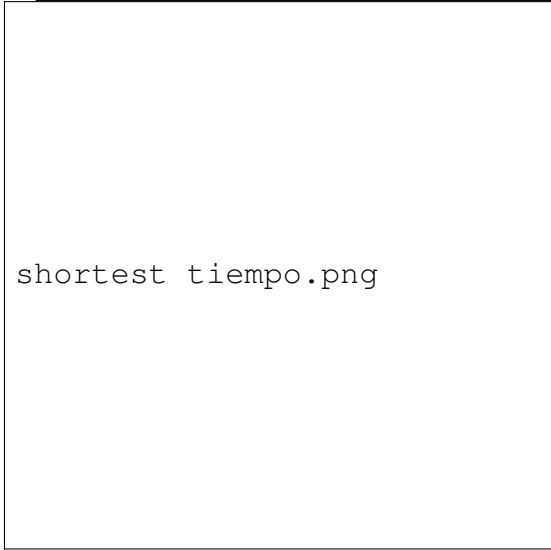
Para las pruebas se utilizó un equipo con un procesador Intel Core i7-8550U, el cual corre a una velocidad base de 1,8 GHz y un turbo hasta 4,00 GHz, con 4 cores y 8 threads, y cuenta con 8 GB de RAM.

Estas pruebas se realizaron con 5 archivos de distintos tamaños y tipos, tres archivos txt, un JPG, un .CR2 que es el formato de imagenes RAW de una DSLR Canon.

La primera prueba realizada fue la lectura del txt más pequeño de 33 bytes. Este se comprimió con el programa en los siguientes tiempos 0.003s, 0.002s y 0.002s y se obtuvo un archivo de 16 bytes.



```
shortest tamao.png
```



```
shortest tiempo.png
```

Seguidamente, se realiza la compresión de otro archivo txt de 7.47 kB, los tiempos respectivos de cada prueba fueron 0.009s, 0.001s y 0.009s. Se logra percibir una compresión hasta 4.3 kb.

```
cat tamao.png
```

```
ejemploclase tamao.png
```

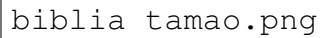
```
ejemploclase tiempo.png
```

```
cat tiempos.png
```

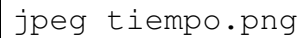
Luego se comprimió un txt de 92 kb a 43 kb, durando 0.004s, 0.002s y 0.002s.

Se corrió el último txt que originalmente pesaba 4120 kb y se obtuvo un archivo comprimido de 2500 kb, y se duró en comprimir 2.04s, 2.15s y 1.98s.

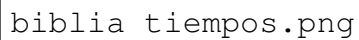




biblia\_tamao.png



jpeg\_tiempo.png



biblia\_tiempos.png

Por último, se tomó un archivo RAW .CR2 que pesaba 28614 kb, obteniendo uno comprimido de 28449



raw\_tamao.png

Luego se usó un JPEG de 7413 kB, sus tiempos por prueba fueron 3.74s, 4.44s y 6.12s y se obtuvo un archivo comprimido de 7313.



jpeg\_tamao.png




raw\_tiempo.png

Luego al descomprimir se obtuvieron los tiempos para los archivos, en el orden que se comprimieron. El primero 0.002s, 0.001s y 0.002s



```
shortest descomp.png
```



```
ejemploclase descomp.png
```

El último txt duró en descomprimir 0.88s, 1.33s y 0.79s

Segundo archivo se descomprimió en 0.004s en los tres intentos.



```
cat descomp.png
```



```
biblia descomp.png
```

Luego al descomprimir el tercer txt se obtuvieron 0.005s, 0.001s y 0.001s.

Luego para ambas imágenes, en JPEG se duró 3.36s, 3.35s y 3.3s.



jpeg descomp.png

Para el RAW .CR2 obtuvieron los tiempos 16.53s, 17.47s y 18.95s



jpeg descomp.png

## VI. CONCLUSIONES

Se puede concluir que, por medio de los resultados anteriores, lo mejor es manejar la misma cantidad de hilos productores y consumidores, ya que cualquier espera que pueda ocurrir por cualquiera de los hilos se reduce a casi cero, por que conforme los productores colocan bytes en el buffer, los consumidores los procesan, y, como se pudo observar en el peor caso, cuando hay más consumidores que productores, estos se quedan esperando por bytes para consumir ya que al ser mayor la cantidad de consumidores, el procesamiento de estos se hace mucho más rápido que la colocación de bytes en el buffer por parte

de los productores y esto genera varias esperas de los consumidores para poder hacer su trabajo.

Además, también se pudo observar que el manejo de hilos en un proyecto ayuda a que las tareas sean ejecutadas con mayor rapidez que si se manejara un único hilo principal, ya que a estos hilos se les asignan tareas para que sean ejecutadas en paralelo o de manera concurrente.[?]

Finalmente, los resultados anteriores se podrían mejorar haciendo uso de hardware más potente y moderno, principalmente utilizando un procesador con mayor cantidad de cores y threads, ya que este recurso sería mucho más rápido en el manejo de hilos paralelos y de ejecución de sus tareas.

## VII. APRENDIZAJES

En esta sección se muestran los aprendizajes de cada integrante del grupo de trabajo.

Jorge Durán Campos: Aprendí que usando toda la capacidad de procesamiento de un multiprocesador actual se pueden realizar procesos de manera más rápida haciendo uso de hilos, debido a que estos amortiguan la carga de un proceso entre los diferentes núcleos, y de esta forma conseguir una disminución de tiempo de ejecución proporcional a la cantidad de núcleos utilizados para este fin. También me ayudó a complementar, junto con la teoría, la importancia de la sincronización entre hilos.

Luis Antonio Montes de Oca Ruiz: De este proyecto aprendí sobre el manejo de hilos y como estos ayudan a mejorar la ejecución de un programa, dividiendo las tareas en cada hilo y así mejorando el tiempo de ejecución y el uso de recursos compartidos. También el manejo de exclusiones MUTEX que permiten un manejo correcto de la región crítica y recurso compartido entre los distintos hilos, siendo esto una parte sumamente importante de este tipo de ejecución, ya que ocurren errores importantes si no se maneja de manera correcta, como la perdida de datos o ciclos de espera de un hilo por otro.

Diego Quirós Artiñano: Aprendí como implementar la teoría que vimos en clase. El

proyecto me ayudó a entender en mayor detalle cómo es que los hilos corren durante su ejecución. Por ejemplo, a pesar de que son paralelos en teoría, en la práctica por la exclusión mutua entre los hilos el programa ejecuta de manera secuencial. Comprendí la importancia de los semáforos y la exclusión mutua. Además, entendí el uso de la función de `cond_wait()`, y la diferencia entre el `cond_signal()` y `cond_broadcast()`.