TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Principios de Sistemas Operativos

Proyecto #1

Comunicación entre hilos

Integrantes:

Francisco Villanueva Quirós - 2021043887

Jarod Cervantes Gutiérrez - 2019243821

Semestre II

2024

Fecha de Entrega:

29/09/2024

Índice

Introducción.....3
Descripción del Problema (Enunciado).....3
Definición de estructuras de datos.....4
ReadDirectoryInfo..... 4
FileInfo.....4
FileInfoBuffer..... 5
Métodos importantes asociados a FileInfoBuffer..... 6
LogInfo..... 6
LogInfoBuffer.....6
Descripción detallada y explicación de los componentes principales del programa.....7
Main.....7
readDirectory.....8
copy.....8
writeLog.....8
Mecanismo de creación y comunicación de hilos..... 9
Pruebas de rendimiento.....10
Conclusiones..... 13

Introducción

Para esta primera entrega del curso, se solicitó poner en práctica conocimientos aprendidos en clase sobre la comunicación y desarrollo de hilos. Además, del uso de manejo de operaciones con directorios y archivos y crear variables compartidas con semáforos mutex.

Para ello, es necesario la creación de un programa en C que pueda manejar los hilos deseados por el usuario y poner a prueba la eficiencia de estos. Los hilos se van a encargar de copiar todo el contenido de un directorio elegido por el usuario y pegarlo en otro directorio vacío.

Descripción del Problema (Enunciado)

Como se comentó en la introducción, el objetivo del programa es la realización de una función “copy” que recibe el nombre del directorio origen y el directorio destino. La idea es visualizar el proceso de copiado del contenido del directorio. Por lo que, se debe tener la información del archivo que se está copiando y la cantidad de bytes que este contiene.

Para ello, es necesario crear varios componentes en el proyecto capaces de cumplir los requerimientos del proyecto. Estos componentes se van a explicar con mayor detalle en los siguientes apartados.

Primeramente, es necesario crear una estructura o función para manejar los archivos o componentes que pueden haber en el directorio. Es importante que cuando se lean los archivos el directorio se vayan asignando a los hilos para que realicen la copia correspondiente.

Esto lleva al siguiente componente, que es el manejo de pool de hilos. El programa debe contar con múltiples hilos que se encargan de copiar los archivos. Debido a que la copia se realiza sobre múltiples archivos no sería eficiente crear un único hilo. Por lo que, el usuario debe poder elegir la cantidad de hilos que quiera crear y

comparar el comportamiento del programa y analizar cuándo solución es más eficiente.

Por último, hay que crear un componente capaz de mostrar los resultados obtenidos por la función “copy”. Se va a crear una archivo de bitácora desde el hilo principal que escriba una entrada por cada archivo que se copia, indicando el nombre del archivo, el subhilo que lo copió, y el tiempo que se duró. Utilice preferiblemente un formato CSV para poder realizar el análisis

Definición de estructuras de datos

En este proyecto se utilizarán las siguientes estructuras de datos:

ReadDirectoryInfo

Esta estructura es utilizada como argumento en la función **readDirectory,** debido a que esta se llama vía **pthread_create.** Se necesitó definir una estructura que contuviera las rutas de origen y destino para copiar la información, además que se incluyó el número de hilo. Es usada para almacenar el directorio origen y destino extraídos del comando

```
typedef struct
{
   char *origin;
   char *destination;
   int threadNum;
} ReadDirectoryInfo;
```

FileInfo

Esta estructura es utilizada como base para poder encapsular cada archivo a copiar, por ello es que contiene un path de origen y de destino, además de incluir el tamaño de archivo a ser copiado. Existirá un **FileInfo** por cada archivo a copiar.

```
typedef struct
{
   char *origin;
```

```
    char *destination;
    size_t size;
} FileInfo;
```

FileInfoBuffer

Esta estructura es utilizada como un buffer que almacenará todas las estructuras **FileInfo** que sean procesadas por **readDirectory.** También de acá el proceso **copy** realiza lecturas para saber que archivo copiar. Además que está también contará con un mutex para controlar su manipulación. En el programa existirá una única instancia de está estructura con el fin que sea accedida por todos los hilos que la requieran. Esta estructura contiene los siguientes campos:

- mutex: es el mutex para controlar su acceso
- not_full: es una condicional utilizada, al momento de leer y escribir para indicar si el buffer está lleno. Y así controlar que el hilo espere alguna lectura o si el buffer ya está cerrado
- not_empty: es una condicional como la anterior. Controla el acceso del read para saber si tiene que esperar alguna escritura o si el buffer está cerrado.
- buffer: es una lista de **FileInfo** que se irá sobreescribiendo circularmente, de acuerdo a sus escritura o lecturas.
- readIndex: indica el índice sobre el que se tiene que leer.
- writeIndex: indica el índice sobre el que se tiene que escribir.
- keepCopying: indica si algún hilo seguirá escribiendo al buffer.

```
typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
    FileInfo *buffer; // Dynamic buffer for structs
    int readIndex;
    int writeIndex;
    int keepCopying;
} FileInfoBuffer;
```

Métodos importantes asociados a FileInfoBuffer

- void writeFileInfo(FileInfoBuffer *logInfoBuffer, FileInfo *logInfo): controla la escritura del buffer, utiliza el mutex para bloquear su acceso, además que espera alguna lectura exterior si el buffer está lleno para realizar su escritura.
- FileInfo *readFileInfo(FileInfoBuffer *logInfoBuffer): controla toda lectura sobre el buffer, utiliza el mutex para bloquear su acceso.

LogInfo

Esta estructura es utilizada como base para poder encapsular cada archivo copiado, por ello es que contiene el nombre del archivo, su tamaño y el tiempo que tardo en copiarlo. Existirá un **LogInfo** por cada archivo copiado.

```
typedef struct
{
   char *name;      // Field to store the name
   size_t size;     // In bytes
   double duration; // In miliseconds
} LogInfo;
```

LogInfoBuffer

Esta estructura es idéntica a **FileInfoBuffer,** sus diferencias son que el **buffer** es de **LogInfo,** y que la función **copy** es quien escribe sobre ella, pueden existir **n** hilos ejecutando **copy.** Y por último que la función **writeLog** lee de este buffer para construir su log.csv con las estadísticas de cada archivo copiado.

```
typedef struct
{
   pthread_mutex_t mutex;
   pthread_cond_t not_full;
   pthread_cond_t not_empty;

   LogInfo *buffer;
   int readIndex;
   int writeIndex;
   int keepLogging;

} LogInfoBuffer;
```

Descripción detallada y explicación de los componentes principales del programa

Como se mencionó anteriormente, se crearon diferentes componentes para dividir responsabilidades en el proyecto y hacerlo más escalable para futuras actualizaciones, además, de las buenas prácticas de repartir funcionalidades. Por ello fue que se dividió en 3 etapas principales:

- read: está etapa se encarga de procesar la ruta origen para extraer todos los archivos. Es ejecutada por la función **readDirectory,** además que se maneja por un hilo.
- copy: está etapa se encarga de copiar cada ítem extraído por **readDirectory,** es ejecutado por **n** hilos. La función que se encarga de esta etapa se llama **copy**
- log: está etapa se encarga de escribir en un archivo .csv las estadísticas de cada archivo copiado. Se ejecuta por la función **writeLog.** Se corre por un único hilo.

Para ello, se decidió crear la siguientes funciones:

Main

Se crea una función main, que es la encargada de llamar a todas las funciones del programa encargadas de realizar la función solicitada. Esta función recibe por parámetro dos variables de tipo char, que son las direcciones del directorio que se quiere copiar y la localización del nuevo directorio donde se van a copiar.

Además, en la función se establecen los hilos que se van a utilizar para la prueba que se va a realizar y una lista de ID’s para que cada hilo se pueda diferenciar. Una vez establecidos, se realiza un ciclo para crear los hilos y se les pasa por parámetros la función copy, que es la que van a realizar y su ID’s correspondiente.

readDirectory

La función readDirectory tiene como propósito leer el contenido de un directorio dado y manejar los archivos y subdirectorios que se encuentran en él. Utiliza dos parámetros, para el directorio de origen y para el destino donde se copiarán los archivos o directorios. Primero, intenta abrir el directorio de origen con opendir().

Luego, utiliza un bucle while que, con readdir(), itera sobre las entradas del directorio. Para cada entrada, la función genera las rutas completas del archivo o directorio en el origen y en el destino utilizando snprintf(), y luego llama a stat() para obtener información sobre la entrada.

Se utiliza la función S_ISREG() para validar si lo que se encuentra es un subdirectorio o un archivo normal. Se crea un objeto FileInfo, se establecen sus rutas de origen y destino, se almacena su tamaño, y se escribe la información en un buffer con writeFileInfo(), que se comentará más adelante.

copy

La función copy es ejecutada por varios hilos y es la que se encarga de copiar los elementos del repositorio origen al destino. Se trabaja un hilo por archivo. Está función obtiene la informacipón de archivo a copiar por medio buffer de FileInfo. En esta etapa se mide el tiempo que tarda en copiar un hilo y la cantidad de bytes. Cuando termina el copia se crea una estructura de log con las estadísticas extraídas, este se envía a la siguiente etapa por medio de un buffer

writeLog

Esta etapa consiste en un único hilo que escribe sobre un .csv con el fin de obtener las estadísticas de cada uno de los archivos copiados.

Mecanismo de creación y comunicación de hilos

En el programa se crean **N + 2** cantidad de hilos especificados en el archivo **macros.h,** su macro se llama **NUM_THREADS.** El **+ 2** se refiere a que se creara un hilo para la función **readDirectory,** y otro hilo para la función **writeLog**, en total dos hilos extra. Y también se crearán **NUM_THREADS** para la función copy.

Ahora que se especifica la cantidad de hilos creados se puede mencionar el orden de su creación. Primero se crea un hilo para **readDirectory** que estará llenado el **FILE_INFO_BUFFER.** Después se crearán los hilos de la función **copy,** los cuáles empezarán a consumir del **FILE_INFO_BUFFER,** y una vez terminan de copiar cada archivo escribirán a **LOG_INFO_BUFFER.** Despues de esta etapa se creará el hilo para **writeLog,** el cuál estará leyendo de **LOG_INFO_BUFFER.**

Una vez visto el orden de creación de hilos, se mencionará su comunicación. La etapa **read** se comunica con la etapa **copy** por medio de **FILE_INFO_BUFFER, read** escribe en el buffer, mientras que **copy** lee de él. Luego la etapa **copy** se comunica con la etapa **log** por medio del buffer **LOG_INFO_BUFFER.** La etapa **copy** escribe en el buffer mientras que **log** lee del mismo.

Los dos buffers mencionados, utilizan funciones para controlar su escritura y lectura por medio de hilos.

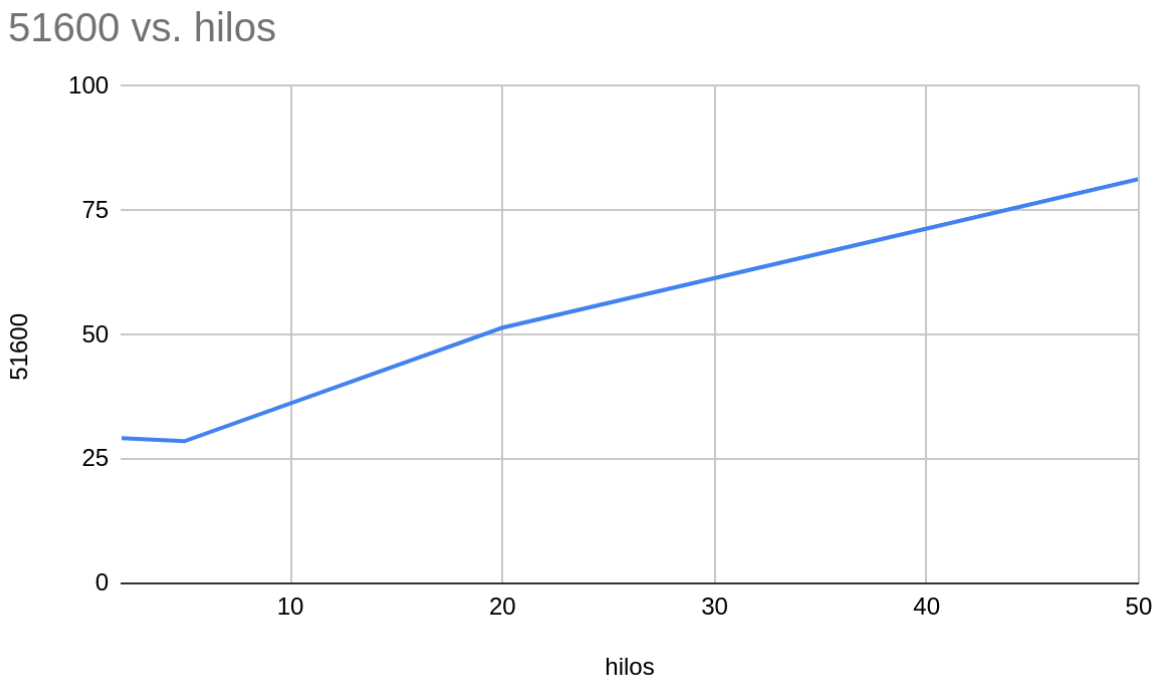
Por último, para detener los hilos, **readDirectory** se detiene cuando haya recorrido todos los directorios a copiar. Una vez termina asigna a la variable **FILE_INFO_BUFFER.keepCopying** un cero con el fin de indicarle a la etapa **copy** que ya no habrá nadie que escriba al buffer, por lo tanto la etapa **copy** estará próxima a detenerse y lo realizará hasta que haya leído todos los ítems dentro de **FILE_INFO_BUFFER.buffer.** Luego de esto la etapa **copy** se detendrá, en el **main** se esperará a que cada hilo de **copy** haya terminado, para así asignar a la variable **LOG_INFO_BUFFER.keepLogging** un cero, que le indicará a la etapa **log** que no ya no habrá algún hilo que escriba al buffer, a partir de aquí la etapa **log** estará ejecutándose mientras existan algún ítem dentro del buffer, y una vez haya procesado todos los ítems terminará su ejecución y notificará al **main** por medio de **pthread_join** que su ejecución a terminado, y concluirá el programa.

Pruebas de rendimiento

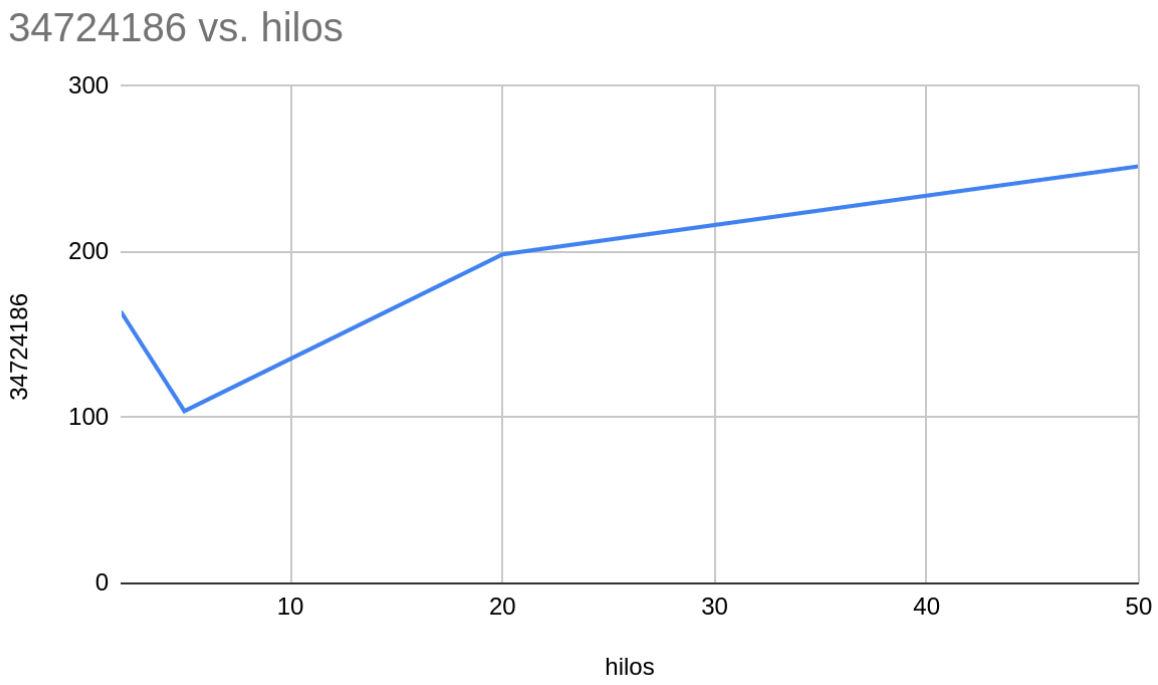
Las pruebas se realizan con buffers de capacidad para 10 elementos. Y el tiempo tomado es únicamente de la etapa de **copy**. La etapa de **read** o **write** no fueron medidas.

num	cantidad de hilos	tamaño de directorio (bytes)	tiempo total
1	2	51 600	29.2
2	5	51 600	28.61
3	20	51 600	51.43
4	50	51 600	81.33
5	2	34 724 186	164.1
6	5	34 724 186	103.99
7	20	34 724 186	198.58
8	50	34 724 186	251.66
9	2	335 589 715	917.82
10	5	335 589 715	598.57
11	20	335 589 715	592.83
12	50	335 589 715	647.19
13	2	551 800 000	30798.56
14	5	551 800 000	31631.03
15	20	551 800 000	34956.52
16	50	551 800 000	45279.28

Para las pruebas de 1, 2, 3, 4, que consistieron en la copia de un repositorio “pequeño”, de **51600 b.** Se puede visualizar que su punto óptimo fue cuando se efectuó sobre 5 hilos, un uso mayor de hilos (20 y 50) ralentizó su ejecución.

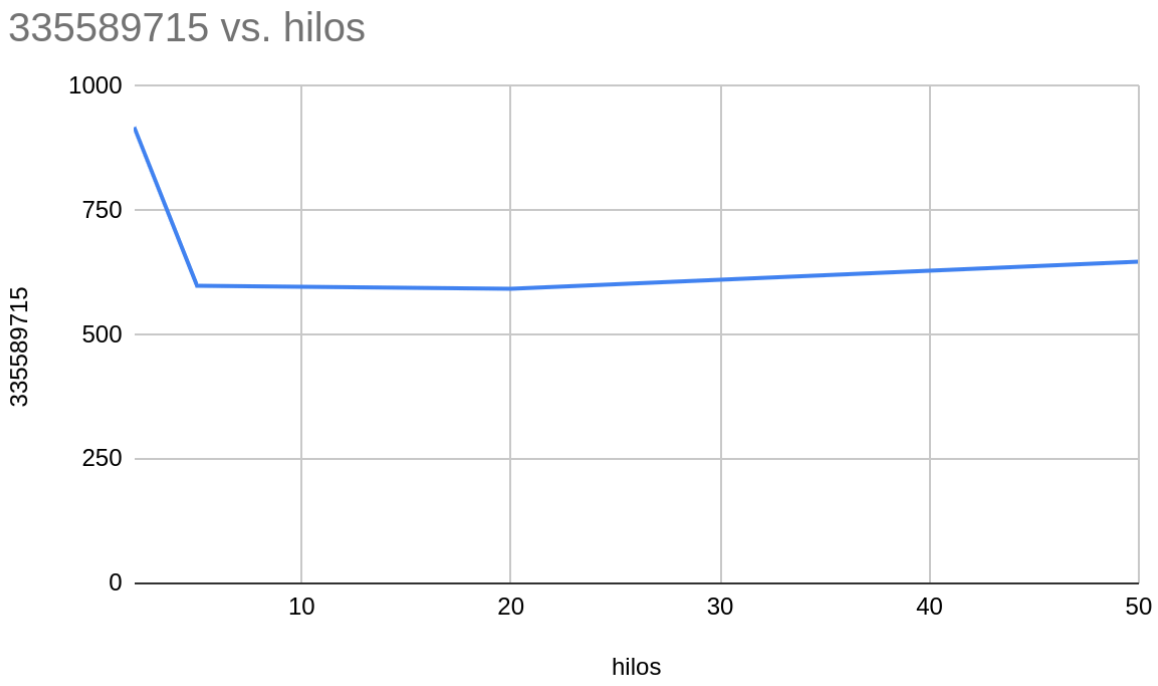


Luego para una repositorio de **34.724186 Mb** se obtuvo un comportamiento similar, pero este sí mostró una mejora importante con respecto a la ejecución de 2 hilos frente a la de 5. Con 20 y 50 tardó bastante más.

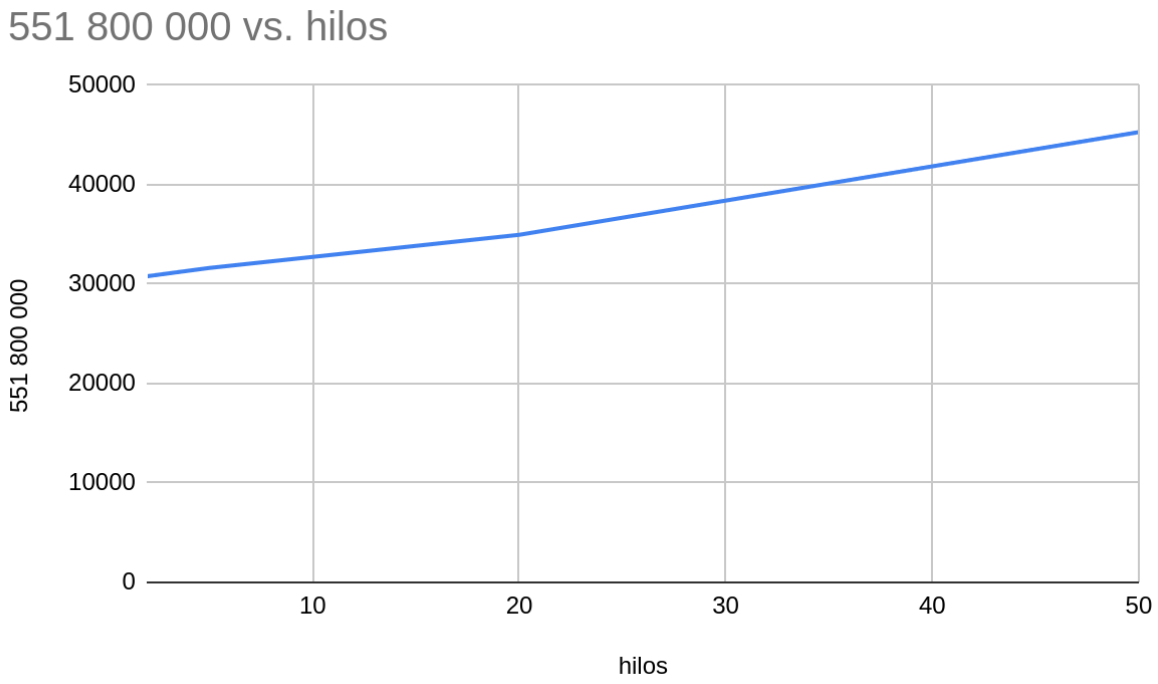


Luego con una ejecución sobre un repositorio de **335.589715 Mb,** se observa un comportamiento más homogéneo con respecto a las ejecuciones de 5, 20 y 50 hilos.

Pero en este caso volviendo a la tabla de pruebas, en el caso 11 se observa que la ejecución con 20 hilos mejoró con respecto a la de 5 hilos.



Por último, las pruebas realizadas sobre un repositorio “**grande**”, de **551 800 000 Mb,** se puede concluir que la cantidad de hilos óptima está entre un rango de 2-5 hilos. Al aumentar la cantidad de hilos, se puede observar como el tiempo de copiado aumenta considerablemente.



Conclusiones

- El uso de estructuras que manejan el buffer por medio del mutex y restringir su manipulación a funciones de escritura y lectura, permite una ágil comunicación entre hilos y etapas.
- Para poder debuggear se tiene un mejor manejo y entendimiento del código debido a su modularidad.
- Con respecto a las pruebas se evidencia que una cantidad de hilos es ineficiente para la copia masiva de archivos.
- También se demostró que un uso excesivo de hilos, (20 y 50) no dieron una mejora significativa con respecto a las copias.
- Se estima que un rango intermedio de hilos, entre 5 y 20, proporciona una copia de información más eficiente.
- Para hacer una mejor estimación se considera que se pueden realizar pruebas más diversas con repositorios que tengan las siguientes características:
    - pocos archivos y pequeños.
    - pocos archivos y grandes.
    - muchos archivos y pequeños.

    - muchos archivos y grandes.
- Se considera que para mejorar la eficiencia del proyecto se puede considerar el uso de hilos para copiar sobre segmentos de archivos en lugar de archivos completos por hilos.
- También cabe considerar que la ineficiencia encontrada en el uso de muchos hilos (20 y 50) puede ser debido al control interno de hilos.