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

12 de junio de 2023

ÍNDICE

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 y la impresión en un archivo del árbol resultante y sus códigos. Estos métodos se describen a continuación

El primer método es un auxiliar del radix sort, llamado get_max() que devuelve el número mayor de repeticiones dentro del solution array y que se usa para el siguiente método auxiliar, count_sort() que implementa toda la lógica de ordenamiento usando arreglos auxiliares y colocando el resultado dentro del solution_array, y luego está el método radix_sort() que se encarga de unir ambos métodos auxiliares. Después se crea el metodo reading_file() el cual es la tarea de los hilos lectores, aquí se hace uso del archivo que se indica al ejecutar el programa y se hace la lectura byte por byte, existen también varias verificaciones como si se llegó a final de archivo, se hace uso de bloqueos y desbloqueos a cada posición del solution_array por medio de MUTEX. Además se hacen declaraciones de métodos y un struct relacionados a la creación del árbol binario, el struct corresponde a un nodo que contiene el byte, la cantidad de repeticiones de este byte, el camino para llegar a este nodo, y punteros a sus hijos izquierdos y derechos, y los métodos son para crear un nodo que contiene un byte, otro para la creación de un nodo que contiene la suma de repeticiones o valor de dos nodos que contienen un byte, después uno que cuenta la cantidad de bytes que nunca aparecen dentro del archivo, es decir que su número de repetición es cero y el último que se encarga de crear todo el árbol. Luego, se declaran métodos para la creación del archivo del árbol binario resultante del algoritmo de Huffman, junto con todos sus datos, como las rutas, su altura y cuales nodos son hojas. Finalmente, dentro del main, como se mencionó anteriormente, existen verificaciones para la cantidad de hilos lectores, para la dirección del archivo y la correcta apertura del mismo, el nombre del archivo final, si se realizó correctamente la creación de los hilos y si estos se unieron correctamente al hilo principal, se realiza el ordenamiento radix sort y se destruyen los MUTEX y al final crea un archivo que contiene al árbol.

Finalmente, se realizaron varias pruebas de lectura con 7 archivos de distintos tamaños y de distintos tipos, con cuatro distintos casos de prueba, misma cantidad de hilos productores y consumidores, 10 productores 1 consumidor, 1 productor 10 consumidores, y 5 productores 8 consumidores. De estas se obtuvo la conclusión que el mejor caso es cuanto existe una cantidad igual de ambos tipos de hilos, y la peor cuando existen más lectores que consumidores, ya que estos consumidores van a quedar esperando hasta que el único productor haga la lectura de un byte y esto genera un proceso de “back and forth” donde los consumidores deben esperar al productor continuamente.

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 leguaje, 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 especifico de los 256 bytes que existen y así asignarles una ruta compuesta de unos y ceros, usando el algoritmo de Huffmann. 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 Huffmann creados al comprimir el 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 rápido.[**?**]

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. 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 cuatro 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 por último está filename que se usa para darle nombre al archivo comprimido. 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 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 el cual es 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() basandose 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() se utiliza para determinar la altura del árbol y colocarla como dato dentro del archivo que contiene el árbol, 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.[**?**][**?**][**?**]

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 posiciónes 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. Luego se hace la creación de los hilos lectores, tomando en cuenta la cantidad establecida por el usuario y se abre el documento para probar si se logra abrir correctamente, si no se termina la ejecución del programa, y luego se recorre todo el archivo para encontrar su tamaño, se devuelve el puntero que controla lectura y se cierra. Luego se hace la creación de lectores 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 imprime la cantidad de bytes en el archivo original, las repeticiones de cada byte, usando ambos arrays, el solution_aux para indicar el byte y solution_aux para indicar sus repeticiones, y luego se imprime la cantidad de bytes leídos y se destruyen los mutex, y para finalizar se hace la creación de los archivos del árbol haciendo uso de los métodos create_tree(), fill_path(), y haciendo una verificación de que si el nodo visitado es una hoja con is_leaf() entonces se termina de escribir esa ruta del árbol, y luego existe un ciclo que imprime el resto de nodos que no son hojas, usando la variable path_table que se describió anteriormente.

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 7 archivos de distintos tamaños y tipos, dos archivos txt, dos PDF, una foto de 24.5 megapíxeles y por último dos videos, uno 4K HDR y otro en 8K. Además, por cada archivo se realizaron cuatro casos de prueba, un hilo para productor y consumidor, 10 productores 1 consumidor, 10 consumidores 1 productor, y 5 productores 8 consumidores.

La primera prueba realizada fue la lectura del txt más pequeño de 33 bytes. Este se leyó con el programa en los siguientes tiempos 0.002s, 0.003s, 0.002s, y 0.003s para cada caso correspondiente, aquí se puede notar que al ser un archivo sumamente pequeño no hay mucha diferencia de tiempo entre cada caso. Seguidamente, se realiza la lectura de otro archivo txt de 7.4 kB, los tiempos respectivos de cada prueba fueron 0.011s, 0.018s, 0.129s, y 0.062s. Se logra percibir una diferencia en el caso donde existen 10 consumidores y 1 productor que tardo mas tiempo que el resto de las pruebas, seguido del caso de 5 productores 8 consumidores.

Luego se usó un PDF de 152.7 kB. Sus tiempos por cada caso fueron 0.193s, 0.276s, 2.371s, y 1.191s. Aquí ya se puede observar un patrón donde el caso de igual cantidad de productores y consumidores es el mejor y el peor es más consumidores que productores.

Se continua con la lectura de otro PDF de 4.2 MB. Sus tiempos respectivos fueron 5.253s, 16.084s, 2m 48.18s, y 1m 14.46s. Se confirma la observación realizada en las pruebas anteriores, siendo el mejor caso la cantidad igual

de productores y consumidores que produjo el menor tiempo.

A continuación, se realizó la prueba con la fotografía con un tamaño de 7.6 MB, dando los siguientes tiempos 12.317s, 32.694s, 5m 4.31s, y 2m 18.13s. Confirmando una vez más que el mejor tiempo se realizó con la misma cantidad de productores y consumidores, siendo el peor cuando hay 10 consumidores y 1 productor.

Por último, ya que el equipo ha demostrado durar bastante tiempo en correr los archivos más pesados de las pruebas anteriores, y, además, se logró confirmar que el mejor caso es cuando existe la misma cantidad de productores y consumidores, ambos videos se van a correr usando este caso para dar una idea del tiempo que podría durar en los peores casos. El tamaño es 437.8 MB para el video 4K HDR y 936,6 MB para el video 8K. El video 4K HDR se leyó en un tiempo de 17m 01.58s y el 8K con un tiempo de 38m 19.41s, si el promedio de diferencia entre el mejor y peor caso es un aumento del tiempo de 30 veces, entonces se podría aproximar que el peor caso para cada video duraría 8 horas para el video en 4K y 16 horas para el video en 8K.

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().