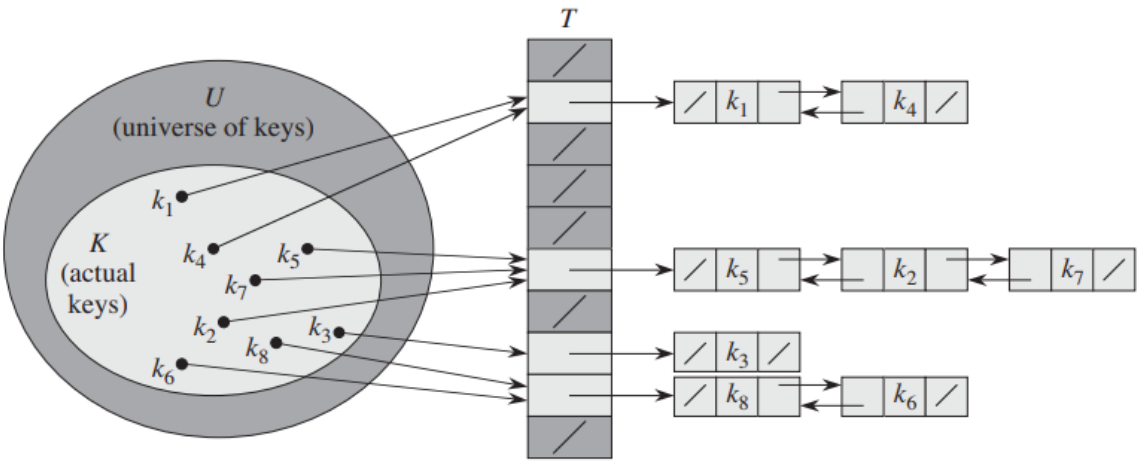
Introducción

En este Trabajo Práctico, buscamos implementar un hashmap que pueda ser usado de forma concurrente, en particular, utilizando threads. Un hashmap es simplemente un diccionario implementado sobre una tabla de hash. En nuestro caso queremos obtener el número de apariciones de una palabra en uno o varios archivos. Con eso presente, la parte más desafiante del trabajo, es evitar los errores que suele acarrear la concurrencia, como deadlocks, race conditions e inanición.

Para la tabla de hash usaremos un array de 26 posiciones, uno para cada letra del abecedario en orden de aparición ASCII. Para asignar una palabra i.e una clave (key) a una posición en la tabla, usamos una función "h" que mapea una palabra con su primera letra. Por ejemplo si una palabra empieza con la letra "a", se le asignará la posición 0 de la tabla, si comienza con "b", la posición 1 y así, hasta z, con posición 25. Para resolver las colisiones utilizaremos hashing abierto (channing), que consiste en encadenar en una lista enlazada (en nuestro caso de forma simple) todos los elementos a los que les corresponde el mismo slot i.e. los que tienen la misma primera letra.

11.2 Hash tables *257*



[1] En la figura, U es el universo de claves; K son las claves cargadas en el hashmap y T es la tabla de hash cuyos buckets apuntan a una lista enlazada. Tenemos que recalcar que si bien en el gráfico la lista es doblemente enlazada, para el trabajo usaremos una lista enlazada de forma simple.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein- *Introduction to Algorithms, 3rd Edition*, 2009. Capítulo 11.

Inserción concurrente

El primer problema que se nos presenta a la hora de armar el hashmap, es el de poder insertar elementos en las listas de conflictos de forma concurrente. El problema radica en que si dos threads quieren insertar un elemento a la misma lista, esto no dé un resultado inesperado y no deseable. Para esto vamos a usar una lista enlazada que tenga la propiedad de atomicidad, donde con esto nos referimos a que las operaciones que se realizan sobre ella, sean indivisibles, con lo cual varios procesos pueden acceder y modificarla sin que se produzcan race conditions, conservando la corrección del programa (la salida se va a corresponder a alguna serialización válida).

En el caso donde varios threads quieran insertar en la misma lista varios elementos, por definición, estos elementos se insertarán adelante, y lo que queremos garantizar con el método, es que el elemento, en efecto, sea insertado y que tenga todos los elementos hasta ese momento, detrás suyo. Cabe resaltar que, si dos o más threads quieren insertar un elemento, no vamos a garantizar un orden específico en la asignación. Esto se debe a que todos los threads van a intentar leer y escribir el valor de la cabeza de la lista, y no estamos imponiendo ningún tipo de exclusión mutua sobre este valor, sin embargo, que no se respete el orden de inserción, no significa que el programa sea incorrecto, ya que éste al estar libre de race conditions, siempre produce un resultado equivalente a alguna serialización válida.

Luego el comportamiento atómico está dado por la inserción exitosa del elemento y no el orden de estas, en el caso que hayan varios threads ejecutando este método.

Ahora veamos como nuestra implementación asegura que sea agregado adelante exitosamente en la lista.

1. Creamos un nuevo nodo con el constructor dado por la cátedra.
2. Cargamos atómicamente el valor de la cabeza. Notar que éste valor es conseguido atómicamente y por lo tanto es el valor de cabeza actual, luego se le asignará al valor siguiente del nuevo nodo.
3. Por último, estaremos chequeando si el valor de la cabeza y del siguiente del nodo, es el mismo con la función atómica `compare_exchange_weak` . Si los valores llegaran a diferir (esto se debe que otro thread ejecutando la función `incrementar` logró cambiar el valor de cabeza antes que éste) cargará de forma atómica el valor nuevo de cabeza en el valor siguiente del nodo creado y volverá a comparar. En el caso que los valores sean iguales, de forma atómica, se reemplaza el valor de cabeza por la dirección del nuevo nodo, así logrando que el nodo sea insertado más adelante con respecto a la lista actual.

Métodos de HashMap

En el segundo problema, queremos implementar un método para `incrementar` el valor en la tabla de una clave, si ésta ya existe. Si no existe, se agrega con valor 1. Para resolver problemas de concurrencia acarreados por una colisión (dos funciones intentando agregar dos palabras que comienzan con la misma letra), utilizamos 26 mutex.

Cuando un thread necesita acceso a una región crítica, llama a mutex_lock. *Si el mutex está desbloqueado, la llamada es exitosa y el thread llamador puede entrar a la región crítica. Por otro lado, si el mutex está bloqueado, el proceso llamador se bloquea hasta que el que está en la región crítica termine y llama a mutex_unlock. Si múltiples threads están bloqueados en un mutex, se elige uno de ellos al azar para que adquiera el lock*[2]. De esta forma un bucket puede ser modificado por una función `incrementar` a la vez. Una vez pedido el mutex, se busca si la clave ya tenía un número asignado, de ser así se lo incrementa en 1, si no, se concatena un nuevo nodo con la clave y el valor inicial 1, utilizando la función `insertar` del ejercicio 1. A su vez esta solución no empeora la concurrencia, porque una función `incrementar` solo restringe uno de los 26 buckets de ser modificado durante el tiempo que posee su mutex.

Por último, tanto `claves` como `valor` son no bloqueantes, pues en ningún momento de la ejecución esperamos a algún recurso para poder proceder. Esto es así porque como dijimos antes, no hace falta garantizar el correcto funcionamiento, en caso de que cualquiera de las dos funciones se ejecute en paralelo con `incrementar`, por lo que no es necesario evitar las posibles inconsistencias que ésto pueda generar en nuestros resultados. Por la misma razón tampoco presentan inanición, ya que además de suponer un scheduler fair, ambas funciones son independientes de todas las demás, pues no requieren esperar por un signal que proviene de otra función, para poder continuar con su ejecución, y por lo tanto ambas terminan en una cantidad acotada de pasos.

[2] Andrew S. Tanenbaum, Herbert Bos -*Modern Operating Systems*, *4th Edition, 2014.* Capítulo 2.3.

Máximo valor

En la función `maximo`, el desafío está en evitar resultados indeseados cuando se ejecuta concurrentemente con la función `incrementar`, ya que si no se toman medidas necesarias, el máximo podría estar cambiando todo el tiempo.

El requisito indispensable a la hora de programar la función, es evitar devolver como resultado, un máximo que nunca lo fue. Por ejemplo, imaginemos un escenario en el cual nuestra tabla solo dispone de dos claves, “arbol” y “buñuelo”, y nos piden calcular el máximo. Supongamos que antes de la llamada la primera tiene 3 apariciones y la segunda tiene 2. Chequeamos el bucket “a” y nos quedamos con “arbol” cómo el máximo con 3 apariciones. En ese momento, inoportunamente, el scheduler nos desaloja y se realizan 2 incrementos en “arbol” y 2 en “buñuelo”. Nos vuelve a dar el control, chequeamos “buñuelo” y vemos que tiene 4 apariciones (lo cual es mayor a la cantidad de apariciones de “arbol” al momento de desalojar, que era 3), por lo que lo convertimos en el nuevo máximo y lo retornamos ya que no quedan claves por recorrer. Sin embargo nunca fue el máximo. El problema fue que permitimos que se modifique un valor que ya habíamos revisado previamente.

En cambio, sí se puede devolver un máximo temporal, es decir, un valor que en algún momento fue el mayor pero al finalizar la ejecución ya no lo era. Sin embargo, nuestra implementación tampoco permite esto último, ya que siempre devuelve el máximo en tiempo real.

Para lograr este comportamiento tomamos varias medidas. A medida que recorremos el hashmap, vamos bloqueando el bucket a analizar, y si encontramos un nuevo valor máximo lo guardamos en una variable. De esta manera, los valores que ya revisamos y no volveremos a revisar, no van a sufrir cambios, por lo menos hasta que liberemos el mutex de su bucket correspondiente. En cambio, los valores que todavía no revisamos, si pueden seguir siendo incrementados, por lo tanto hay concurrencia. Es verdad que cuando estemos recorriendo el final de la estructura, el nivel de concurrencia va a ser mínimo ya que todas las filas de la tabla van a estar bloqueadas, pero es un tiempo muy corto y aceptable.

Al terminar de recorrer, encontramos el máximo (al momento en que fue solicitado) y luego se liberan todos los mutex. De esta manera, evitamos que potenciales llamados a `incrementar` hagan que devolvamos un máximo que nunca lo fue. Así podemos pensar que se está devolviendo un “máximo real”. Si liberamos los mutex antes de encontrar el máximo, en ese período de tiempo posterior al que los desbloqueamos, podrían llegar incrementos en los buckets que ya liberamos, lo cual podría alterar el máximo. En este último caso, el valor devuelto sería un máximo temporal (máximo al momento en que se pidió, no al momento de devolverlo).

Para poder realizar la función `maximoParalelo`, qué aprovecha el uso de threads para calcular el máximo valor, de manera concurrente y eficiente, adoptamos la siguiente estrategia:

- Declaramos una variable `fila`, en la que guardamos el número del siguiente bucket a recorrer, y un mutex para evitar race conditions alrededor de esta variable. También declaramos una variable `max` en donde guardamos los valores correspondientes a la clave cuyo valor sea el máximo temporal, y también un mutex para evitar race conditions. Estas 4 variables serán compartidas por todos los threads.
- Antes de crear los threads, verificamos que la variable `fila` esté en rango, para no crear threads innecesarios y que el algoritmo sea más eficiente. Esto lo hacemos sin pedir el mutex ya que sólo usaremos este valor cómo un primer filtro, y luego, más tarde, volveremos a asegurarnos de que realmente sea el valor real, sin race conditions. Si pidiéramos el mutex para hacer esto, estaríamos ralentizando la concurrencia de manera innecesaria.
- A partir de este momento, tenemos una variable `contador`, que cómo su nombre lo indica, contabiliza la cantidad de threads que creamos, para después realizar los joins correspondientes.
- Luego, creamos los threads, y estos lo primero que hacen es guardar el valor de `max` en otra variable `tmp1`. Tampoco pedimos el mutex en este caso, por la misma razón especificada anteriormente.
- Luego, pedimos el mutex de fila y chequeamos si está en rango. Si no lo está, significa que ya se recorrió toda la estructura, liberamos el mutex y retornamos.
- En caso que siga en rango, primero copiamos el valor de la variable `fila` en otra variable `tmp2`, incrementamos `fila` y liberamos el mutex (cómo se explica en el ejercicio 4, lo hacemos de esta manera para minimizar el tiempo de bloqueo del mutex).
- Luego pedimos el mutex del bucket correspondiente a la fila que tenemos en la variable `fila` y empezamos a recorrerla.
- Si encontramos una clave cuyo valor es mayor al guardado previamente en tmp1, pisamos el valor de tmp1 con el valor encontrado y luego seguimos recorriendo la fila.
- Al terminar de recorrer la fila, pedimos el mutex de `max` para chequear si el valor que tenemos en tmp1 es mayor al de `max`. Si es así, actualizamos el valor y liberamos el mutex. Si no, liberamos el mutex.
- Al finalizar este ciclo, comenzamos de nuevo, preguntando si el valor de fila sigue en rango, es decir, sigue habiendo filas para recorrer.

Todas estas medidas nos permitieron implementar la función sin condiciones de carrera, aprovechando al máximo los threads y terminando su ejecución solo en el caso en el que no queden más buckets por recorrer.

Carga de archivos

Al implementar `cargarArchivo` no hizo falta tomar recaudos desde el punto de vista de la sincronización, pues lo único que hacemos, es leer el archivo e incrementar el valor de la siguiente palabra (clave), mediante el uso de la función `incrementar`, que por como fue implementada, sabemos que no presenta race conditions, inanición, ni deadlocks. Esto posibilita que desde otros threads, podamos llamar a `incrementar` sin perjudicar la concurrencia y sin acarrear los problemas mencionados anteriormente, haciendo la implementación de `cargarArchivo` mucho más sencilla.

A la hora de desarrollar la función `cargarMultiplesArchivos`, hubo que tomar varios recaudos. Tuvimos que considerar cuál sería el próximo archivo a cargar. Ésta información la guardamos en la variable `archivo_a_cargar`, la cual es compartida y actualizada por todos los threads. La utilizamos tanto en la primera asignación de archivos para cada thread, como también cuando algún thread ya terminó de cargar el archivo que le correspondía, y debe continuar con otro (si es que hay).

Para garantizar el correcto funcionamiento y evitar race conditions, utilizamos un mutex a la hora de actualizar esta variable. Lo que hacemos en la sección crítica es: primero verificar que la variable sigue en rango. Si esta no lo estuviese, no tenemos más archivos por leer y por lo tanto liberamos el mutex. En el caso en que siga en rango, guardamos la variable en una copia, la incrementamos y liberamos el mutex. Si no hiciéramos una copia, tendríamos que esperar a que se termine de ejecutar la función `cargarArchivo` para incrementar la variable y luego liberar el mutex, lo que incrementaría considerablemente el tiempo que el mutex pasa bloqueado. De esta manera, maximizamos la concurrencia y evitamos los problemas por condiciones de carrera, como por ejemplo, dos o más threads distintos cargando el mismo archivo.

Queremos que cada archivo solo pueda ser leído por un thread a la vez. Por lo tanto, en el caso en el que tengamos más threads que archivos, decidimos ni siquiera inicializar el excedente de threads para mayor eficiencia. En consecuencia tuvimos que considerar el hecho de no hacer joins de threads que no fueron creados. Para ésto, utilizamos la variable contador, que justamente contabiliza la cantidad de threads creados hasta el momento, de forma que cuando finalizamos la carga de archivos, sabemos cuántos creamos y por lo tanto cuántos y cuáles debemos joinear.

Experimentación

Hipótesis

Con los conocimientos adquiridos sobre el tema, propusimos una serie de hipótesis, relacionadas con la performance de las versiones single-thread en comparación a las multi-thread, de las funciones implementadas en los ejercicios anteriores, junto con su respectiva experimentación.

Metodología de experimentación

Para comprobar el valor de verdad de las hipótesis, realizamos un archivo cpp de experimentación en el cual medimos los tiempos de ejecución al cargar archivos y buscar el máximo. Todos los tests fueron ejecutados veinte veces con la misma máquina, promediando los resultados obtenidos, y dejando dos segundos de tiempo muerto entre tests para que se enfríe el CPU. En el archivo readme.md están las instrucciones necesarias para correr los experimentos.

Especificaciones de la máquina:

- AMD Ryzen 5 5625U (six-core /12 Threads)
- 24 GB RAM
- Ubuntu 22.04.2 LTS

Hipótesis 1: a la hora de cargar un archivo grande, va a ser mejor en términos de performance, dividir el archivo en varios archivos y cargarlo concurrentemente, que cargarlo entero con la versión single-thread.

Experimentación:

Cantidad de archivos	Cantidad de palabras por archivo	Cantidad de palabras total	Cantidad de threads	Tiempo promedio de ejecución (segundos)
1	≈915,000	≈915,000	1	5.48825
10	≈91,500	≈915,000	4	2.32375
10	≈91,500	≈915,000	8	2.18097
10	≈91,500	≈915,000	10	1.97513
20	≈45,750	≈915,000	8	2.76741
20	≈45,750	≈915,000	16	2.59474
20	≈45,750	≈915,000	20	2.50313
30	≈30,500	≈915,000	30	2.34383
50	≈18,300	≈915,000	50	2.32547

Distribución: todos estos experimentos se realizaron sobre el mismo archivo, el cual está compuesto por alrededor de 915,000 palabras, donde 12,114 son distintas. La cantidad de claves está bien distribuida entre todos los buckets de la tabla de hash, exceptuando los que corresponden a las letras X y Z, que cuentan con una cantidad menor.

En las filas 2, 3 y 4 dividimos el archivo en 10 partes iguales para cargarlos de forma separada por distintos threads y aprovechar la concurrencia. En las filas 5, 6, y 7 hicimos lo mismo, pero en 20 partes iguales. En la fila 8 lo dividimos en 30. En la última lo dividimos en 50 partes.

Conclusión: con la evidencia empírica obtenida, podemos decir que nuestra hipótesis es verdadera. Al analizar un caso con muchas palabras, vemos que distribuirlas en varios archivos, y permitir la concurrencia entre threads, reduce considerablemente el tiempo de ejecución. Esto se debe a que a que como la carga de trabajo es tan grande, tener varios threads encargándose de la carga de distintos archivos, acelera el procedimiento, pues trabajan en conjunto, en comparación a la versión single-thread de la función, que opera de forma secuencial, sin poder repartir la carga de archivos.

Notemos que, por ejemplo en las pruebas con 10 threads, hay una mejora significativa respecto a single thread del 64%. Sin embargo, al realizar las pruebas con 20 threads, el tiempo utilizado fue 21% mayor. Intuimos que esta baja en el rendimiento no se debe a la cantidad de threads en sí que fue empleada, sino que la cantidad de archivos en la que partimos el dataset no es lo suficientemente grande como para sacarle un mayor provecho a los threads. Otra posible causa sería el costo adicional que implica traer una mayor cantidad de archivos de disco (20 en comparación a 10).
Finalmente, vale la pena mencionar que la diferencia más marcada en el rendimiento, ocurre cuando comparamos la versión single-thread contra la que utiliza 4 threads. En concreto es una mejora del 57,6%. A medida que incrementamos la cantidad de threads empleados, si bien en general se puede apreciar una mejora en el rendimiento, ese porcentaje de mejora va disminuyendo. Esto puede deberse a la capacidad de paralelización de nuestra máquina, como también a toda la complejidad estructural que implica tener muchos threads ejecutándose al mismo tiempo, como por ejemplo todos los mecanismos de sincronización necesarios para que no hayan race conditions, inanición, ni deadlocks.

Hipótesis 2: cuanto más grandes sean y más archivos haya, el rendimiento de la versión multi-thread de cargarArchivos va a ser mucho mejor en comparación a la single-thread.

Experimentación:

Cantidad de archivos	Cantidad de palabras por archivo	Cantidad de threads	Tiempo promedio de ejecución (segundos)
5	50,000	1	0.153936
5	50,000	5	0.118588
20	50,000	1	0.645397
20	50,000	8	0.386927
20	50,000	16	0.362592
20	50,000	20	0.362181
5	915,000	1	35.0709
5	915,000	5	10.7114

Distribución: en las primeras 6 filas trabajamos con un mismo archivo de 50,000 palabras. Es una lista de las 1000 palabras en inglés más usadas, copiada y pegada 50 veces (este cambio de dataset es arbitrario). Cada bucket de la tabla de hash cuenta con alrededor de 40 claves, exceptuando los que corresponden a la letra X y la letra Z, los cuales no cuentan con ninguna clave. En las últimas dos filas trabajamos con el mismo archivo de la hipótesis 1, el cual estaba compuesto por 915,000 palabras.

Conclusión: la evidencia empírica constata la hipótesis propuesta. En la tabla, vemos que manteniendo fija la cantidad de archivos y la cantidad de palabras que contienen, la versión multi-thread es mejor que la versión single-thread, e incluso la diferencia en rendimiento es mayor si se continúa incrementando la cantidad de archivos y threads que ejecutan la función. Si bien puede parecer que la mejora no es tan grande, hay que tener en cuenta que en general, el tiempo promedio de ejecución de la muestra analizada, es bastante chico.

Si por ejemplo comparamos la primera entrada de la tabla, con la segunda, hay una mejora del 22,9% en el rendimiento, y de la tercera a la cuarta, la mejora es del 40%, las cuales son considerables. La justificación de por qué sucede esto, es análoga a la que se proporcionó en la conclusión de la hipótesis 1.

Si analizamos la primera y la sexta fila de la tabla, vemos que ésta última tiene peor tiempo de ejecución. Uno creería que esto no debería pasar, ya que en la primera fila cargamos 5 veces el archivo sin concurrencia, y en la sexta estamos cargando 20 veces al archivo pero con 20 threads, pero una cosa a tener en cuenta, es que como se dijo en la hipótesis 1, el tiempo que lleva traer los archivos de disco a memoria puede estar perjudicando la performance (y más si consideramos que traer archivos de memoria secundaria es mucho más lento que si se tratase de memoria principal). Además es importante destacar, que como dijimos en la hipótesis 1, el procesador en el que realizamos los experimentos solo puede correr 12 threads al mismo tiempo. Por lo tanto los 20 archivos no se están cargando simultáneamente, sino que cada thread debe esperar su turno.

Por último, los mecanismos de sincronización para garantizar correctitud también podrían estar afectando al rendimiento, sobre todo porque estamos cargando 4 veces más archivos, habiendo el cuádruple de palabras y por consiguiente aumentando la lucha por los mutex.

Hipótesis 3: cuantas más palabras distintas haya en el archivo a cargar, peor va a ser el rendimiento de la versión multi-thread respecto a la versión single-thread de `cargarArchivo`.

Experimentación:

Cantidad de archivos	Cantidad de palabras por archivo	Cantidad de threads	Tiempo promedio de ejecución (segundos)
5	≈915,000	1	35.0709
5	≈915,000	5	10.7114
5	390,625	1	291.095
5	390,625	5	232.007

Distribución: en las primeras 2 filas trabajamos con el archivo de 915,000 palabras que venimos utilizando. En las últimas 2, usamos un archivo de 390,625 palabras, todas de ellas distintas, el cual creamos por medio de un script, insertando palabras de 4 letras de forma alfabética utilizando todas las letras de la tabla de hash. Luego mezclamos el archivo por medio de un comando para que estén distribuidas de forma aleatoria. Todos los buckets cuentan con la misma cantidad de claves.

Conclusión: los datos de la tabla muestran evidencia a favor de la hipótesis. Observemos que en las dos primeras entradas de la tabla, la cantidad de palabras total es 4,575,000, y la proporción de palabras distintas es del 0,26%. Entonces manteniendo fijos esos datos, y ejecutando la versión multi-thread con 5 threads, resulta en una mejora de tiempo de ejecución del 69,4% respecto de la versión single-thread.

Por otra parte, en las últimas dos entradas de la tabla, la cantidad de palabras total es 1,875,000 (menos de la mitad que antes), con una proporción de palabras distintas del 20%, y vemos que manteniendo fijos esos datos, y repitiendo el proceso (single-thread vs 5 threads), la mejora del rendimiento es tan solo del 20,2%. Más específicamente, la versión multi-thread sigue siendo mejor que la single-thread, pero no solo no incrementó el porcentaje de mejoría al aumentar la cantidad de palabras distintas, sino que se redujo sustancialmente.

Esta reducción en la ganancia de performance, y el hecho de que los tiempos de ejecución de las dos primeras entradas de la tabla sean mucho menor que el de las últimas dos, a pesar de trabajar con menos de la mitad de las palabras, se debe a lo siguiente: al tener una mayor proporción de palabras distintas en total, por cada bucket vamos a tener muchas claves, lo que implica que si bien accedemos en O(1) a las entradas de la tabla, después el recorrido de la lista enlazada tiene un costo más elevado. Ésto se realiza por cada palabra a cargar, afectando negativamente a la performance. Además, al tener una mayor proporción de palabras distintas, en general a cada archivo le va a corresponder una mayor cantidad de buckets, incrementando los intentos de acceso a un mismo bucket, que va a tener como

consecuencia a varios threads esperando que se libere el bucket en uso para cargar la clave que les corresponde a cada uno.

En el archivo de 390.625 palabras, hay la misma cantidad de claves para cada entrada de la tabla de hash, mientras que en el otro archivo esto no pasa. Hay buckets con muchas claves y otros con muy pocas o incluso ninguna. Por lo tanto los threads se van a pelear más seguido buscando el acceso a los buckets más llenos, mientras que en el otro caso, al estar mejor distribuido, no ocurre tan seguido.

Hipótesis 4: a la hora de cargar archivos de menor tamaño, la versión multi-thread va a seguir siendo más eficiente, pero en menor proporción.

Experimentación:

Cantidad de archivos	Cantidad de palabras por archivo	Cantidad de threads	Tiempo promedio de ejecución (segundos)
3	50,000	1	0.17545
3	50,000	2	0.127638
3	50,000	3	0.102540

Distribución: trabajamos con el mismo archivo de 50,000 palabras que utilizamos en la hipótesis número 2, el cual contiene 1000 palabras distintas.

Conclusión: los resultados de la experimentación comprueban la hipótesis enunciada.
En este análisis, la cantidad de archivos a cargar y las palabras que contienen, es menor en comparación a los experimentos de las hipótesis anteriores, e igualmente manteniendo fija la cantidad de archivos, vemos una mejora en el rendimiento del 27,25% al usar 2 threads en vez de la versión single-thread, y del 19,6% usando 3 threads en vez de 2. Con lo cual, al igual que en la hipótesis anterior, aumentar la cantidad de threads, siempre resulta en una mejora de rendimiento, aunque esta no es tan considerable como sí sucedía en la conclusión de la hipótesis 3.

Hipótesis 5: al cargar un archivo con una cantidad considerable de palabras distintas, la versión multi-thread de máximo, ejecutándose con 2 o más threads como mínimo y 26 como máximo, va a tener un mejor rendimiento que la versión single-thread.

Experimentación:

Cantidad de palabras distintas en total	Cantidad de threads	Tiempo promedio de ejecución (segundos)
390,625	1	$4{,}68 * 10^{-2}$
390,625	4	$2{,}07 * 10^{-2}$
390,625	8	$6{,}89* 10^{-3}$
390,625	16	$8{,}15 * 10^{-3}$
390,625	20	$8{,}67 * 10^{-3}$
390,625	26	$8{,}72 * 10^{-3}$

Distribución: trabajamos con un mismo archivo de 390,625 palabras. Se trata del archivo que contiene todas palabras distintas de 4 letras distribuidas aleatoriamente.

Conclusión: viendo los resultados de los experimentos podemos afirmar a favor de las hipótesis. Obtenemos una reducción significativa del tiempo al usar 4 threads y 8, y esto se debe que en la versión single thread, la tarea estaría viendo todas las listas secuencialmente y como los archivos provistos tienen una gran cantidad de palabras distintas, las listas generadas por `cargarMultiplesArchivos` tendrán un tamaño extenso y perdería mucho tiempo recorriéndolas.

Luego cuando usamos más de 8 threads notamos que empezamos a tener un incremento en el tiempo de ejecución.

El procesador que utilizamos para la experimentación solo cuenta con 12 threads en simultáneo, entonces la ventaja de tener tantos threads empieza a disminuir ya que tendríamos más threads a los cuales esperar a que terminen.

La idea de usar más threads resulta natural para reducir la carga del single thread y poder ir visitando la estructura en varias instancias diferentes y en listas diferentes y aunque tengamos un punto donde conviene dejar de incrementar los threads, en todos los casos el overhead generado no es suficiente como para elegir la versión single thread sobre la multithread.

Hipótesis 6: al cargar un archivo con una cantidad relativamente chica de palabras distintas, la versión single-thread de máximo va a ser mejor en cuanto a performance que la multi-thread.

Experimentación:

Cantidad de palabras distintas en total	Cantidad de threads	Tiempo promedio de ejecución (segundos)
1000	1	$3{,}40 * 10^{-5}$
1000	2	$4{,}07 * 10^{-5}$
1000	5	$4{,}89 * 10^{-5}$
1000	10	$5{,}19 * 10^{-5}$
1000	26	$5{,}21 * 10^{-5}$

Distribución: trabajamos con el archivo de 50,000 palabras que venimos utilizando, el cual es una lista de las 1000 palabras en inglés más usadas, copiada y pegada 50 veces.

Conclusión: la hipótesis planteada coincide con los resultados obtenidos en los experimentos. Vemos que manteniendo fija la cantidad de palabras, e incrementando la cantidad de threads, empeora sostenidamente el rendimiento. Esto se debe a que si bien, estamos incrementando los threads en un rango razonable, es decir, entre 2 y 26 (lo cual previamente resultó en una mejora de la performance, al tener tan pocas palabras distintas), si bien los threads se reparten el trabajo, este no es lo suficientemente grande como para que resulte beneficioso hacerlo.

Más aún, todos los mecanismos de sincronización implementados, como mutex, que si bien permiten obtener los resultados correctos, en un contexto en donde tenemos muchos threads, resulta complejo a nivel estructural mantener la concurrencia, lo cual perjudica el rendimiento de la versión multi-thread. Es decir, la simpleza que provee la versión single-thread, que en casos anteriores causaba que fuera peor, aquí resulta conveniente, tanto desde el punto de vista implementativo, como del rendimiento.

Conclusiones

En general no se puede decir que nuestros métodos que hacen uso de multi threading sean netamente superiores a su versión single thread, ni viceversa. La experimentación demostró que según distintos contextos se obtienen ventajas dramáticas entre ambos. Entre los factores más destacables están: la cantidad de archivos, el número de palabras y cuántas distintas hay.

También vimos las ventajas de implementar funciones de modo que no tengan problemas con la concurrencia, como por ejemplo poder llamar en otras funciones, sin importar si se pretende que sean concurrentes o no, a la función en cuestión, sin afectar a la correctitud de la misma, y por lo tanto teniendo el comportamiento esperado por parte del programador.

Además, observamos que a pesar de que nuestra implementación no cree más threads de los necesarios, igualmente el rendimiento puede empeorar.
Asimismo, nos resulta pertinente señalar que nuestra implementación, es más eficiente y concurrente con grandes volúmenes de datos. Esto lo evidenciamos en la sección de experimentación. En consecuencia, creemos que podría ser una solución factible para un entorno real con escala global, donde el tamaño de la entrada puede ser extremadamente grande.

Por último, consideramos que es muy importante, antes de empezar a escribir código, analizar bien los posibles escenarios para nuestras funciones, y contemplar el trade-off entre un código más sencillo, pero potencialmente (aunque no siempre) más lento, y un código más complejo, que requiere de implementar adecuadamente mecanismos de sincronización, no sólo para garantizar correctitud, si no una concurrencia aceptable, con la ventaja de que posiblemente (aunque no siempre) sea más eficiente.

Bibliografía

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein- *Introduction to Algorithms, 3rd Edition*, 2009. Capítulo 11.

2. Andrew S. Tanenbaum, Herbert Bos -*Modern Operating Systems*, *4th Edition, 2014.* Capítulo 2.3.

3. Linux Man Pages: https://man7.org/linux/man-pages/man2/futex.2.html