DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Sincronización

Sistemas Operativos

Integrante	LU	Correo electrónico
Florencia Rosenzuaig	118/21	f.rosenzuaig@gmail.com
Gonzalo Ariel Meyoyan	514/20	gonzalo@meyoyan.com
Lucas Ariel Sotomayor	240/20	lasotomayor20@gmail.com
Pablo Segre Maturano	601/18	pablosegre@hotmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

1. Introducción

En este trabajo práctico, hemos explorado la gestión de la concurrencia en los sistemas operativos, centrándonos en la ejecución concurrente de programas y las técnicas para evitar condiciones de carrera y gestionar la contención de recursos. Para ello, hemos utilizado *threads*, una herramienta proporcionada por los sistemas operativos que nos permite tener múltiples hilos de ejecución concurrentes dentro de un mismo programa, utilizando la interfaz `pthreads` del estándar `POSIX`.

Como parte de este trabajo, hemos realizado la implementación de una estructura de datos llamada **HashMapConcurrente**. Esta estructura es una tabla de hash abierta que utiliza listas enlazadas para gestionar las colisiones. Su interfaz de uso es similar a la de un mapa o diccionario, donde las claves son `strings` y los valores son enteros no negativos. El propósito principal de esta estructura es procesar archivos de texto y contar la cantidad de apariciones de palabras. En este caso, las palabras se utilizan como claves y el número de veces que aparecen se almacena como valor correspondiente.

Durante el desarrollo del mismo, hemos aplicado conceptos clave de la gestión de la concurrencia para garantizar la integridad de los datos en entornos concurrentes. Hemos utilizado técnicas como bloqueo de secciones críticas, sincronización de hilos y control de acceso a recursos compartidos para evitar condiciones de carrera y garantizar la consistencia de los resultados.

2. Lista atómica

Para guardar los elementos dentro de cada bucket de nuestro hash vamos a usar una lista enlazada. Nuestra implementación de lista debe tener en cuenta que el hash va a ser usado de forma concurrente, es decir, que varios *threads* van a estar accediento y modificando listas en simultáneo, y puede suceder que varios *threads* quieran acceder o modificar una misma lista a la vez.

Si no tenemos cuidado en nuestra implementación, estamos en riesgo de que haya condiciones de carrera.

Una solución para éste problema es hacer que nuestra lista sea atómica, es decir, que sus operaciones sean indivisibles. En particular, queremos que `insertar` al inicio de la lista sea indivisible, ya que es lo que va a suceder concurrentemente y lo que puede causarnos conflicto.

2.1. Insertar

```
void insertar(const T valor)
```

Para insertar en una lista enlazada se deben hacer 3 cosas:

1. Crear un nodo nuevo (llamemoslo `nodoNuevo`)
2. Asociar que su siguiente es la cabeza actual de la lista (es decir, `nodoNuevo.siguiente = _cabeza}`
3. Definir `nuevoNodo` como la nueva cabeza (`_cabeza = nuevoNodo`)

Vamos a usar la función atómica `exchange` [*] de la siguiente forma.

```
nuevoNodo->_siguiente = _cabeza.exchange(nuevoNodo):
```

Esta función lo que realiza es las siguientes asignaciones de forma atómica:

[*] https://en.cppreference.com/w/cpp/atomic/atomic/exchange

```
nuevoNodo->_siguiente = _cabeza;
_cabeza = nuevoNodo;
```

No hay conflicto con realizar el paso 1 en simultáneo con otros threads. Dado que pudimos lograr hacer el paso 2 y 3 de forma atómica, la inserción de nodos no genera algún tipo de conflicto con la concurrencia.

3. HashMapConcurrente

Para este punto se nos proveyó una implementación parcial de **HashMapConcurrente** y se nos requirió completar la misma con tres funciones.

3.1. Incrementar

```
void HashMapConcurrente::incrementar(std::string clave)
```

La primera a implementar es `incrementar`, que dado por parametro una `clave` de tipo `String`, incrementará el valor asociado a la misma si es que esta se encuentra en la lista correspondiente dentro de la tabla; o en caso contrario la definirá e insertará en la lista debida asociandola con el valor 1.

Para la implementación de la función se pensó la utilización de una lista de mutex llamada `totalKeys` y su tamaño es la cantidad total de letras a utilizar en la tabla de hash. Esta decision es debido a que la tabla tiene la idea explotar al máximo la idea de concurrencia mediante los *Threads*, asi que es necesario tener ciertos cuidados al manipularla.
Con esto sabido, se puede explicar la funcion. Dada la clave por parámetro, la dicha comienza tomando el `hashIndex` de la misma para poder saber en que lista de la tabla la clave debe ser posicionada. Pero antes de esto, lockeamos el semáforo `totalKeys[hashIndex]` para, en dado caso que el *thread* pueda pasar, señalar que pasó y bloquear al siguiente *thread* que necesite el mismo index; o esperar a que el *thread* que esté manipulando la zona crítica termine. Esta decisión es para evitar el posible caso de que en el intento de manipular la misma lista de la tabla, dos *threads* crean que la misma clave no estaba y la definan, entre otros posibles errores.

Para lo anterior y ya dentro de la zona crítica, se crea un iterador para la lista correspondiente y verifico si está la clave o no, si está incremento el valor asociado a esta, sino defino a la clave y la inserto a la lista con el valor asociado 1. Luego de manipular la zona crítica, el *thread* manda la señal `unlock` para que el siguiente en espera pueda realizar su tarea. Por último añadimos la clave a la lista general `Keys`.

3.2. Claves

```
std::vector<std::string>HashMapConcurrente::claves()
```

La segunda función a implementar es `claves`, que simplemente nos devuelve todas las claves definidas en la tabla. Su implementación es bastante sencilla, itera en los 26 buckets de la tabla de hash y sobre cada lista; asi obteniendo la clave y colocándola en un vector a parte. Luego de realizar

todas las iteraciones, retornamos este vector. Al necesitar que no sea bloqueante y libre de inanición, no colocamos ningun tipo de semaforo para que se pueda explorar la lista libremente.

3.3. Valor

```
unsigned int HashMapConcurrente::valor(std::string clave)
```

La tercer función a implementar es `valor`, que dada una `clave` como parámetro esta devolverá el valor asociado a la clave, y si no se encuentra en la tabla devolverá 0. Como se solicita que no sea bloqueante y libre de inanición, simplemente lo que hacemos es crear un iterador y buscar la clave en la tabla. Al no modificar la tabla en ningun aspecto, y al querer encuadrarnos a los requerimientos; no se usaron semaforos de ningun tipo en este caso.

3.4. Maximo

```
hashMapPair HashMapConcurrente::maximo()
```

Dentro de la implementación de **HashMapConcurrente** se nos proveyó la función `máximo`, que devuelve el par `<clave,valor>` mas alto de la tabla. Esta funcion otorgada lo que hacía de base es iterar en toda la tabla y comparando todos los elementos con el máximo valor encontrado anteriormente. Al terminar la iteración, devolvía la variable `máximo` que guardaba el valor mas grande obtenido.

En primera instancia se nos solicitó que se modifique la misma para que la ejecución de la misma junto a la función de incrementar no provoque problemas en la ejecución paralela; como la modificación de la misma lista que estabamos revisando el posible nuevo máximo. Esta primer solución se llevo a cabo mediante el arreglo de mutex `totalKeys`; bloqueando el bucket/lista que estabamos revisando así no era posible modificarla mientras nosotros ejecutabamos la iteración en la misma. Al terminar de revisarla se daba rienda suelta a poder modificarla nuevamente.

Sin embargo, y a pesar de que soluciona gran parte de los problemas de ejecución en paralelo, sigue sin atrapar todas los inconvenientes que nos podrían surgir. Una de ellas, que podría ser considerada las más grave, es que puede darse el caso de que la función `maximo` devuelva un valor que jamas tuvo esa propiedad en la tabla. ¿Cómo? A continuación se prooverá un ejemplo.

1. Supongamos que tenemos en la tabla 4 palabras definidas: Albahaca, Almidón, Carabobo y Pepe.
2. Ahora veamos que orden tienen en base a sus valores, en este ejemplo tendremos lo siguiente `<Almidón,3>, <Barba,4> ,<Albahaca,5>, <Carabobo,8>, <Pepe,14>`.
3. En la primer lista `<Albahaca,5>` es máximo y se lo toma temporalmente como tal. Luego de iterar sobre el bucket de **a**, esta se desbloquea para su modificación.
4. Al ser posible modificarla nuevamente, entonces podría modificarse el valor hasta un valor tal que `<Almidón,15>`.
5. Ahora la tabla en orden de valores tiene la siguiente pinta: `<Barba,4>,<Albahaca,5>,<Carabobo,8>, <Pepe,14>,<Almidón,17>`.
6. Ahora mientras máximo bloqueo el bucket de B, entonces es posible modificar el bucket C, e incrementar a `Carabobo` para que supere a `Pepe`.

7. Ahora la tabla en orden de valores tiene la siguiente pinta: `<Barba,4>,<Albahaca,5>, <Pepe,14>, <Carabobo,15>,<Almidón,17>.`

8. Entro al bucket C y tomo a `Carabobo` como máximo.

9. Finalizo la iteración de buckets en la tabla y devuelvo a `<Carabobo,15>`.

Nótese que `Carabobo` fue tomado como máximo por tener la propiedad circunstancial de que superaba al máximo anterior que era `Alabahaca` antes de incrementar a `Almidón`. Pero como es posible a este último incrementarlo luego, entonces este pasa a ser el nuevo máximo superando a todos. Y como a `Carabobo` se lo pudo incrementar para que supere a `Pepe` pero que no supere a `Almidón`, entonces este es visto como un máximo, aunque realmente jamás lo fue en ningun momento.

Esto sucede ya que solo bloqueo un solo bucket, y pueden ser incrementados no solo los posteriores al que estoy revisando, sino también los anteriores; dando este problema.

3.5. MaximoParalelo

`hashMapPair HashMapConcurrente::maximoParalelo(unsigned int cantThreads)`

Para solucionar el problema anterior, se nos requirió implementar el método maximoParalelo. Este lo que realiza es, dado por parametro un entero; reparte el análisis del máximo en esa cantidad de *threads* indicada.

La implementación del método consiste en, primero, en la creación de una variable atómica llamada `sigIndex` cual es el siguiente index por recorrer. También se crea un arreglo llamado `maxThreads` con `cantThreads` elementos, donde cada thread va a guardar el máximo que encontró. Luego iteraremos desde 0 hasta la `cantThreads`, realizando por cada uno la función `maximoPorThread` dandole por referencia `sigIndex` y el vector `maxThreads`. Vease que se debió hacer un arreglo en el código para que este mismo sea compatible con los requisitos de funcionamiento de C++.

```
for(unsigned int i=0; i<cantThreads; i++
    threads.emplace_back(maximoPorThread, this, ref(sigIndex), ref(maxThread[i]))
}
```

Para esto, se necesito definira `maximoPorThread` como `static`. Esto nos traería la limitación de no poder acceder a las variables privadas de la clase **HashMapConcurrente**. Para poder suplir esta necesidad, tambien damos por parámetro al *this*, que es la misma instancia de la clase que estamos usando. Luego en la función se llama a la clase como data mediante un puntero:
`void HashMapConcurrente::maximoPorThread(HashMapConcurrente *data, atomic<unsigned int>sigIndex, hashMapPair max)`. En `maximoPorThread` lo que realizamos en primera instancia es guardarnos en `index` el valor `sigIndex` y luego sumarla en 1 atómicamente mediante `fetch_add`. Esto para marcar el indice al bucket siguiente disponible para que otro thread lo tenga en cuenta.

Luego hasta que el `index` no supere a el máximo indice de la tabla que es dado por `cantLetras`, buscaremos el máximo del bucket indicado por `index` mediante la funcion `maximoPorBucket`; realizamos el cambio al máximo global dado por `max` si es que el del bucket lo supera; y luego volvemos a hacer:

```
index = sigIndex.fetch_add(1);
```

Como nuestros threads no acaban de terminar su rutina hasta que todos juntos acabador de revisar toda la lista, ninguno queda inactivo en el trabajo de búsqueda del máximo.

La estrategia para repartir los *Threads* fue hacer un arreglo de estos llamado `Threads`, reservamos la memoria necesaria que estará indicada por la cantidad de *threads* que tenemos. Luego hacemos un `for` que iterará tantas veces como cantidad de `threads` tengamos, realizando el `maximoPorThread` de cada uno. Finalmente, esperamos a que todos finalicen, chequeamos el máximo en la lista `maxThread` y lo retornamos.

¿Qué recursos comparten los *threads*? Bueno, estos necesitan compartir si o si el índice `sigIndex` para que todos los threads estén en un bucket de la tabla distinto; y la lista `maxThread`, que guarda el máximo que obtuvo cada *thread*. ¿Cómo los protegemos de las condiciones de carrera? Bueno, para esto definimos a `sigIndex` como un `atomic<int>` lo que nos permite atómicamente guardar su valor en otra variable por copia e incrementar su valor en uno, lo que nos salvaguarda de que dos *threads* tomen dos valores *index* iguales. Y segundo para `maxThreads` cada *thread* edita su posición unicamente, por lo que ningún otro podría tomar o editar de forma errónea esa posición cambiando el máximo equívocamente.

4. Cargar archivos

Para cargarlos archivos implementamos las siguientes funciones.

4.1. cargarArchivos

```
int cargarArchivo(HashMapConcurrente &hashMap, std::string filePath)
```

Esta función carga el archivo en`filePath`, y por cada palabra en el archivo, la guarda en `hashMap` usando la operación `incrementar`.

4.2. cargarMultiplesArchivos

```
void cargarMultiplesArchivos(
    HashMapConcurrente &hashMap,
    unsigned int cantThreads,
    std::vector<std::string> filePaths
)
```

Esta función recibe un vector de *filePaths* y carga todos los archivos indicados en un mismo *hashMap*. Para ello, creamos `cantThreads` threads (usando las mismas precauciones que para `maximoParalelo`).

5. Experimentación

5.1. Introducción

En esta sección comparamos los resultados de ejecutar el programa sobre un conjunto de tests variando los parámetros iniciales.

5.2. Hipótesis

La lectura de archivos se va a beneficiar de una cantidad incremental de threads. El tiempo de lectura disminuirá a medida que el numero de threads se acerca al numero de archivos a ser leídos.

Por otro lado, la búsqueda de máximos también se vera beneficiada por una cantidad incremental de thread. En este caso, el tiempo de calculo disminuirá a medida que el numero de threads se acerquen a la cantidad de buckets que tenemos en nuestro hashmap (26).

Cabe destacar que el tiempo de lectura o cómputo, no debería verse afectado positivamente por un numero de threads mayor al soportado.

5.3. Metodología

- En nuestro test utilizamos, para cargar el hashmap, una `lista de strings` [**] de aproximadamente 370000 entradas, dividida en 8 archivos.
- Procedemos a ejecutar el programa múltiples veces, variando la cantidad de threads utilizados en cada ejecución (entre 1 y 16). Recopilamos alrededor de 150 data-points para cada posible variación, lo cual consideramos proporciona un conjunto significativo de resultados para poder realizar un análisis estadísticamente sólido.

5.4. Características del experimento

5.4.1. Información de Hardware

Para correr la experimentación, utilizamos una maquina virtual (encima de Windows 10) con las siguientes características:

- CPU: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz (4 Núcleos asignados)
- Memoria: 8GB RAM DDR4
- Sistema: Ubuntu 22.04.2 LTS

5.5. Resultados

Threads Utilizados	Tiempo de Lectura (Segundos)	Tiempo de Computo (Milisegundos)
1	216.4	6.65882
2	118.051	3.60820
3	79.9727	2.67589
4	67.2432	2.39418
8	64.3049	2.65464
16	64.5778	2.65206

Tabla 1: Mediana de Tiempos de Ejecución

En esta tabla figuran las medianas de los tiempos de lectura y computo medidos sobre los resultados obtenidos (n=150). Observamos que duplicar la cantidad de threads disponibles de 1 a 2 resulta en

[**] https://github.com/dwyl/english-words/blob/master/words_alpha.txt

grandes mejoras sobre el tiempo de ejecución. Notamos que las mejoras resultan ser rendimientos decrecientes cuando $\#threads > 2$, y no resultan en mejoras significativas cuando $\#threads > 4$.

Continuamos entonces, realizando un análisis en detalle de los casos mas relevantes (1-4 threads):

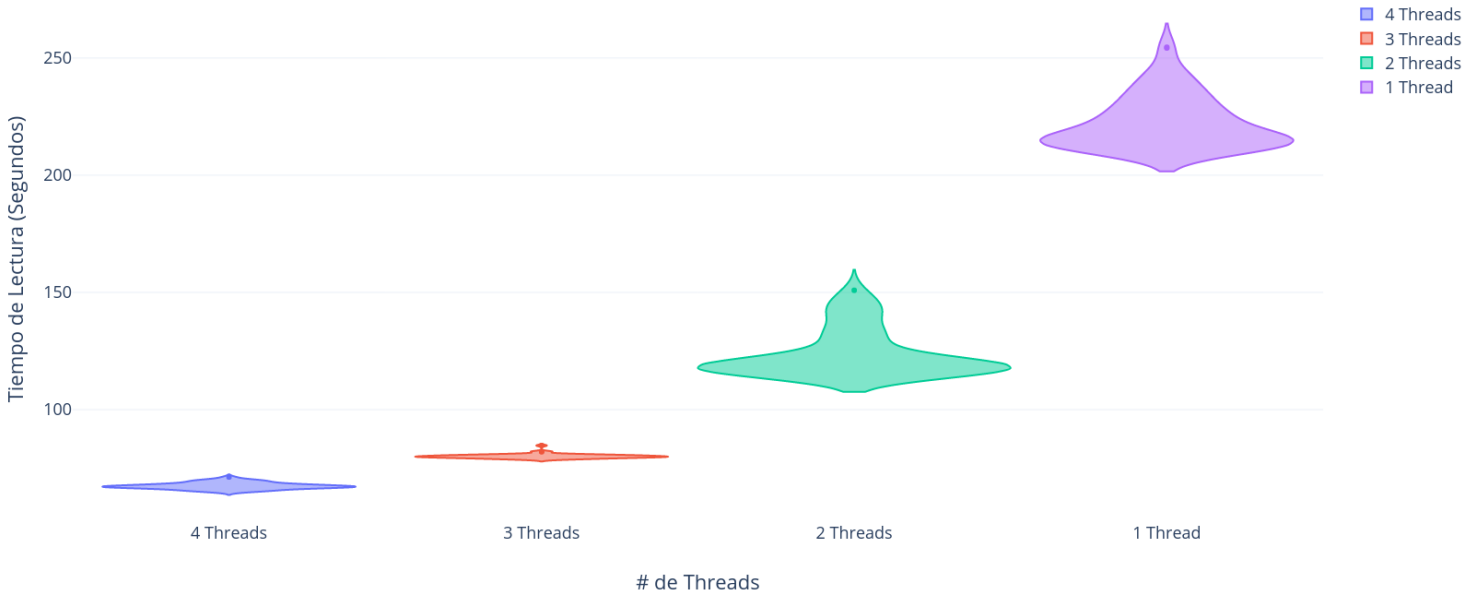

Figura 1: Comparacion de Tiempos de Lectura

Figura 2: Comparacion de Tiempos de Computo

Los *violin plot* representados en la figura 1 y 2, comparan el tiempo total de lectura y computo del programa con distintos números de threads. El eje x representa el numero de threads utilizados, mientras que el eje y representa el tiempo de lectura/computo para cada iteración (n=150).

Los gráficos ilustran claramente como el aumento del numero asignado de threads impacta positivamente el tiempo de lectura/computo, y en particular resalta la importante mejora que se presenta al pasar de single-threading (#threads=1) a multi-threading (#threads$\geq$ 2).

Además, mediante el uso de un *violin plot*, podemos observar que no hay una cantidad significativa de *outliers* que pudieran estar afectando las métricas utilizadas para evaluar el rendimiento de cada experimento. Este tipo de gráfico también nos permite notar que la desviación estándar de cada violín esta asociada inversamente a la cantidad de threads utilizados. Esto es un indicador de que los tiempos de computo varían mas frecuentemente a medida que se reduce el número de threads involucrados.

5.6. Conclusión

Los resultados obtenidos durante la experimentación previa, demuestran que un numero incremental de threads conlleva significativas mejoras en el rendimiento del programa.

Al aumentar sistemáticamente el número de threads y al evaluar las medianas de ejecución, observamos una clara tendencia de mejoras de velocidad en ambas operaciones de lectura, y búsqueda de máximo. Adicionalmente, concluimos que aumentar los threads utilizados resulta en tiempos de ejecución mas consistentes.