DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1: Threading

Grupo 6

Sistemas Operativos

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Leandro Rodriguez	521/17	leandro21890000@gmail.com
Guido Rodriguez Celma	374/19	guido.rodriguez@outlook.com.ar
Miguel Rodriguez	57/19	mmiguerodriguez@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (+54 11) 4576-3300

https://exactas.uba.ar

Índice

1. Introducción

En este informe vamos a explicar las decisiones tomadas, estructuras elegidas y consideraciones sobre nuestra implementación del trabajo práctico para la gestión de concurrencia.

Además habrá una sección de experimentación en la cual analizaremos las ventajas y desventajas del enfoque concurrente vs no concurrente. Para esto compararemos ejecuciones sobre varios datasets, con distintos niveles de paralelismo.

2. Desarrollo

Vamos a presentar las clases que modificamos y para cada una de ellas vamos a especificar los cambios que hicimos en sus funciones y la explicación de por que gestionan la contención sobre los recursos de manera correcta.

2.1. Lista atómica

Se nos pide implementar el método `insertar(T valor)` que inserta un valor en la cabeza de una lista atómica enlazada. Que la lista sea atómica quiere decir que, al tener procesos concurrentes que ejecuten diversas operaciones provistas sobre una misma instancia de la lista, no van a ocurrir condiciones de carrera sobre la misma.

Para que nuestra implementación de `insertar` cumpla la propiedad de atomicidad, definimos un mutex que proteje a la variable `_cabeza`. Esto obliga a que las inserciones se hagan de forma secuencial, es decir que una vez que se está en el medio de una inserción, no puede ocurrir que otra operación de inserción ejecutada por otro proceso modifique la lista antes de que termine de modificarla el proceso que tomó el mutex.

Que un programa utilice una lista atómica no significa que el mismo esté libre de incurrir en cualquier condición de carrera, sino solamente que al aplicar operaciones de la interfaz que provee la lista atómica, no va a provocar condiciones de carrera sobre los objetos que la conforman.

2.2. HashMap concurrente

2.2.1. Claves y valor

Las funciones `claves()` y `valor(string clave)` no tenían que garantizar que el resultado sea consistente si se ejecutaban con `incrementar` en simultáneo, por lo que la implementación de ambas fue la misma que se haría en un ambiente de ejecución normal secuencial ya que no son bloqueantes y únicamente tienen que recorrer la lista, guardar los resultados obtenidos en variables temporales y retornarlo.

2.2.2. Incrementar

La funcionalidad mas importante de esta clase a tener en cuenta en cuanto a las condiciones de carrera y problemas de sincronización es la función `incrementar(string clave)` que debe incrementar el valor de una clave si existe en la tabla o crearla en caso contrario. Esto puede traer problemas ya que por ejemplo, si ejecutamos `incrementar` sin contención podría ocurrir que, al querer incrementar dos claves iguales en simultáneo, en vez de insertar el par `<clave, 1>` una única vez e incrementarlo, este se inserte dos veces de forma que el bucket de esa clave quede de la forma `<..., <clave, 1>, <clave, 1>, ...>`.

Nuestra forma de evitar generar estas condiciones de carrera fué inicializar una lista de 26 mutex (uno por cada bucket) tal que cuando un thread llama a la función `incrementar`, al inicio de la ejecución se tomá el mutex correspondiente al bucket dado por la función de hash. Esto nos permite poder incrementar al mismo tiempo claves que no caigan en un mismo bucket (no colisionan) y a su vez, nos permite insertar de a uno por vez en cada bucket, por lo que no vamos a tener el problema de múltiples inserciones. Una vez realizado el incremento, el mutex es liberado.

Otra solución posible era utilizar un único mutex que sea tomado en cada llamado a `incrementar`, pero esa implementación generaría mayor contención que la permitida por el enunciado, ya que en ese caso no se van a poder incrementar concurrentemente claves que no colisionen sino que se van a incrementar de a una por vez.

2.2.3. Máximo

Si se ejecutasen concurrentemente `máximo` e `incrementar` sin ningún tipo de contención, podría ocurrir que el resultado obtenido por máximo, luego de una secuencia de operaciones concurrentes, no se corresponda con ningún orden de ejecución válido de las operaciones realizadas. Por ejemplo:

`Orden de ejecución`: inc(A, 1), inc (B, 1), maximo(), inc(B, 1), inc(A, 1).

`Resultado obtenido`: maximo(), inc(A, 1), inc (B, 1), inc(B, 1), inc(A, 1).

Para evitar inconsistencias de este estilo, decidimos que al iniciar un llamado a `máximo`, la función tome todos los mutex correspondientes a los buckets del hashmap. De esta forma se evita reportar un máximo anterior al comienzo de la ejecución, ya que para tomar los mutex se debe esperar a que termine la ejecución de cualquier `incrementar` que estuviera procesandose al momento del llamado a `maximo`.

Además se asegura que el máximo obtenido al terminar la ejecución sea el verdadero, ya que todos los buckets quedan bloqueados durante el recorrido del hashmap en busca del máximo.

2.2.4. Máximo paralelo

Al iniciar la ejecución de `maximoParalelo` tomamos los mutex correspondientes a los buckets del hashmap para prevenir inconsistencias con posibles llamados a `incrementar` realizados por otros procesos. Una vez que tomamos todos los mutex, creamos la cantidad de threads indicada.

Para repartir el trabajo entre los threads utilizamos dos variables compartidas, `proxFila` y `maximoGlobal`, cada una envuelta en un mutex propio.

Cada thread iterativamente consulta cual es su próximo número de fila a procesar mediante la variable `proxFila`, y la incrementa para indicar a los demás threads que esa fila ya fue considerada. Esto permite que todas las filas sean procesadas una única vez. En caso de que el número de fila obtenido sea mayor a la cantidad de buckets del hashMap, termina su ejecución ya que no quedan filas por procesar sin asignar. Esto obliga a que los threads siempre estén activos y su ejecución termine únicamente en este caso.

Al finalizar cada iteración, cada thread compara el máximo obtenido de la fila que procesó con el valor de `maximoGlobal`, en caso de que sea mayor lo reemplaza. Esta comparación (e intercambio) está libre de condición de carrera al estar contenida en una zona de exclusión mutua.

Una vez finalizada la ejecución de todos los threads, se liberan los mutex para los llamados a incrementar y se retorna el `maximoGlobal`.

2.3. Cargar archivos

Para la implementación de `cargarArchivo` no tuvimos que tomar ningún recaudo especial ya que todas las consideraciones requeridas para la ejecución concurrente están encapsuladas sobre `HashMapConcurrente`, que se va a encargar de incrementar las claves de manera correcta por lo visto en la sección 2.2.2.

Para `cargarMultiplesArchivos` el enfoque utilizado fue similar al de `maximoParalelo`, ya que cada thread que generamos, iterativamente obtiene qué índice del `vector<string>` de archivos debe leer mediante una variable compartida `proxArchivo` protegida por un mutex. Luego utiliza la función `cargarArchivo`, que va a cargar los archivos de forma correcta (por lo explicado anteriormente).

3. Experimentación

La experimentacion fue llevada a cabo en un entorno `Linux` con sistema operativo `Ubuntu 20.04 LTS`. La CPU utilizada fue un `Intel® Core™ i5-4670K CPU @ 3.40GHz × 4`.

3.1. Distintos niveles de paralelismo

Es este experimento analizaremos el tiempo de ejecución de la función `maximo` aplicada sobre distintos datasets y variando el número de threads dedicados a su procesamiento. Los datasets que usaremos son:

- `mejor-caso-concurrencia`: Todos los buckets del hashmap estarán en uso y en cada uno habrá una sola palabra repetida una cierta cantidad de veces.
- `peor-caso-concurrencia`: Solamente un bucket del hashmap será utilizado, el resto estarán vacíos.
- `caso-promedio`: El hashmap se llenará con palabras generadas de forma aleatoria.

Hipótesis:

Para el dataset `mejor-caso-concurrencia`, esperamos que al aumentar el número de threads se vea una mejora significativa en el tiempo de ejecución, ya que a cada thread le tocará hacer el mismo trabajo. Este dataset consta de 7.800 palabras.

Sobre el dataset `peor-caso-concurrencia`, creemos que el tiempo de ejecución no cambiara significativamente al aumentar el número de threads ya que solamente uno se encargará de procesar la fila no vacía, el resto no aportará a la ejecución. Este dataset consta de 1000 palabras.

Finalmente, para el dataset `caso-promedio`, esperamos que disminuya el tiempo de ejecución a medida que se aumente el número de threads, aunque no tan significativamente como para el dataset de `mejor-caso-concurrencia`. Al estar las palabras generadas de manera aleatoria, esperamos que la distribución $\sim$ uniforme en el hashmap beneficie a la ejecución concurrente, dado que cada thread tendrá que realizar un trabajo similar. Este dataset consta de 10.000 palabras.

Resultados:

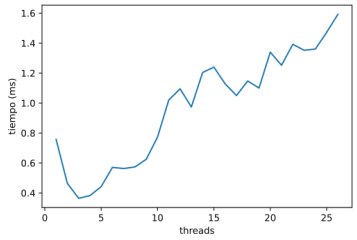

Figura 1: Tiempos de ejecución de máximo sobre mejor caso, para cant. de threads variable.

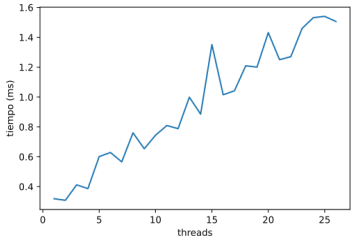

Figura 2: Tiempos de ejecución de máximo sobre peor caso, para cant. de threads variable.

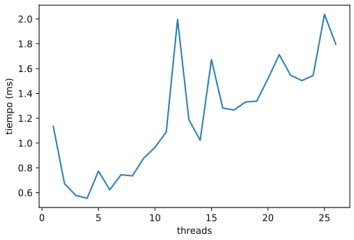

Figura 3: Tiempos de ejecución de máximo sobre caso promedio, para cant. de threads variable.

Cada experimento fue ejecutado 5 veces, el resultado presentado es la media de esas 5 ejecuciones.

Análisis:

Para el mejor caso y el caso promedio, observamos una mejora significativa cuando se ejecutan entre 2 y 10 threads concurrentemente, respecto a la ejecución secuencial. Sin embargo, a partir de $\sim$ 13 threads, vemos que el tiempo de ejecución crece de forma consistente a medida que se aumenta el número de threads. Creemos que esto se debe al overhead introducido por las estructuras de contención (los mutex) y la logica adicional.

En el peor caso, no solo el rendimiento no mejora, sino que se tiene una peor performance conforme aumenta el número de threads. Esto se explica en que el uso de threads añade complejidad a la ejecución y en este caso, no aporta nada al procesamiento ya que un solo thread procesará la fila no vacía mientras el resto se queda en estado de bloqueo o idle. Además al haber más procesos, por cada desalojo deben esperar más tiempo hasta que el scheduler vuelva a ponerlos a ejecutar.

3.2. Tiempo de trabajo por thread

En este experimento analizaremos el tiempo efectivo de procesamiento que realiza cada thread para calcular el máximo, variando el número de threads utilizados. Usaremos el dataset de caso promedio presentado anteriormente, con un archivo de 10.000 palabras.

Hipótesis:

Por la forma en que está escrita nuestra implementación, esperamos que los primeros threads creados se ocupen de una mayor parte del procesamiento que los últimos. Esto se debe a que los threads son enviados a ejecutar al momento de su creación, por lo que aquellos creados al final podrían iniciar su ejecución con una parte importante del procesamiento ya hecha.

Resultados:

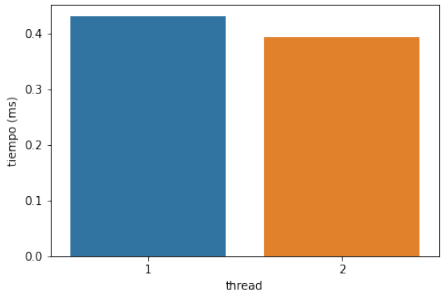

Figura 4: Tiempo de procesamiento 2 threads.

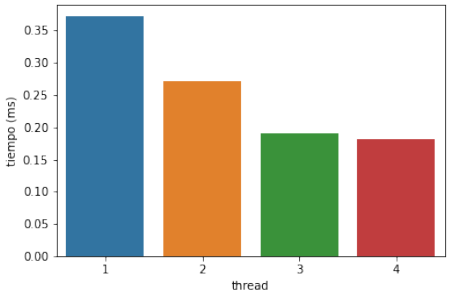

Figura 5: Tiempo de procesamiento 4 threads.

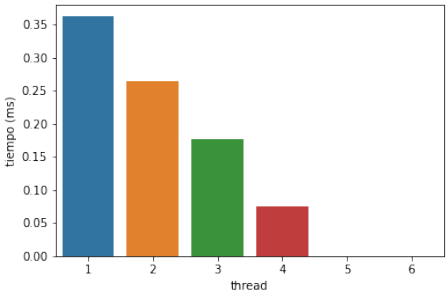

Figura 6: Tiempo de procesamiento 6 threads.

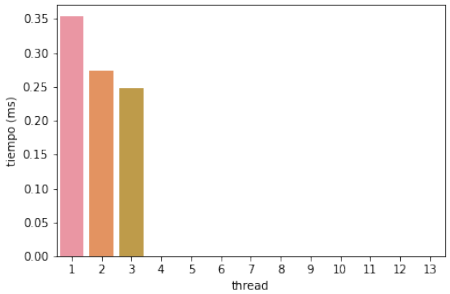

Figura 7: Tiempo de procesamiento 13 threads.

Análisis:

Observamos que, en forma similar a nuestra hipótesis, a medida que aumenta el número de threads el trabajo realizado por los últimos threads creados va disminuyendo considerablemente. Para nuestro dataset vemos que del 5to thread en adelante ninguno realiza nada de procesamiento.

Esto ayuda a explicar los resultados obtenidos en nuestro primer experimento, ya que vimos que tener más threads de los necesarios no aporta nada al procesamiento de la función máximo.

4. Conclusiones

Basándonos en los resultados presentados anteriormente, concluimos que una implementación concurrente puede ser efectiva, si se tienen en cuenta algunas consideraciones como son:

- La contención necesaria para evitar inconsistencias propias de la concurrencia no resulte en una ejecución prácticamente secuencial, es decir, que el problema a resolver sea paralelizable con zonas de exclusión pequeñas.
- El beneficio de tener varios threads procesando simultáneamente distintas partes del problema sea mayor al overhead introducido por las estructuras de contención y de manejo de threads.
- El costo de cómputo del problema a resolver sea lo suficientemente grande como para que se justifique el costo extra, en recursos, tiempo de procesamiento e implementación de la solución concurrente.
- La cantidad de threads utilizados no sea tal que muchos estén constantemente bloqueados o que no realicen una porción significativa del procesamiento.