DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

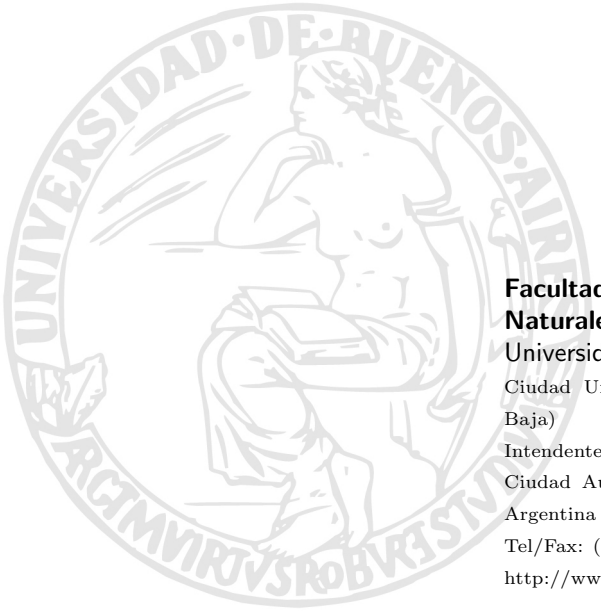
Trabajo Práctico

Threading

22 de Octubre de 2024 — Sistemas Operativos

Grupo 11

Integrante	LU	Correo electrónico
Valencia, Juan Segundo	705/22	juansegundovalencia@gmail.com
Melli, Tomás Felipe	371/2	tomas.melli1@gmail.com
Loria, Ezequiel	111/16	c03iff@hotmail.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

http://www.exactas.uba.ar

Índice

1 Introducción

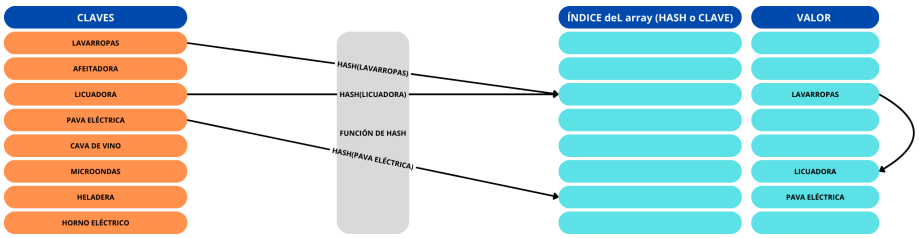
En este informe se describirá el conjunto de soluciones logradas para implementar un **HashMapConcurrente**, una estructura de datos que consiste en una tabla de hash abierta que maneja las colisiones mediante listas enlazadas. La idea de esta implementación es gestionar la concurrencia y las condiciones de carrera en un contexto de órdenes de compra de una tienda.
El principal desafío entonces será, desarrollar una estructura capaz de mantener la consistencia de los datos aún en un contexto de alta concurrencia. Esto lo podremos garantizar con el uso de herramientas de sincronización como los semáforos y las variables atómicas.
En cuanto a la lista enlazada que gestiona la colisión de los hashes, podemos decir que se modelará de manera que no haya **RaceCondition** y la denominaremos **ListaAtómica**. Esta lista resguardará cada producto que se inserte de forma que, incluso en un contexto de alta concurrencia, los datos guardados respondan a la información real sobre ellos.
Finalmente, luego de satisfactoriamente implementar la estructura de datos antes presentada, haremos uso de ella para cargarle toda la información que devenga de los productos de la tienda.

2 Desarrollo

2.1 Operaciones Lista Atómica

2.1.1 Insertar

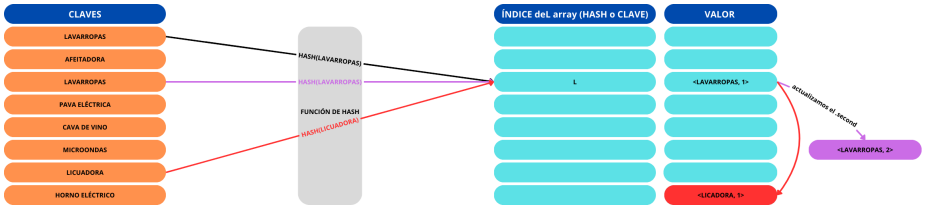
En el primer ejercicio del TP nos piden completar el método insertar(T valor) de la clase ListaAtómica. Esta clase se utiliza para modelar una lista enlazada que se encargará de resolver las colisiones dentro del HashMap. Frente a muchas operaciones a la hora de querer agregar o incrementar elementos dentro de la lista, podría surgir que haya race condition, es decir, salidas que no correspondan a una serialización válida. Es por ello que se decide construir una lista que cumpla con la propiedad de atomicidad. Esta es la propiedad que tienen ciertas variables que al ser modificadas por alguna tarea, esta última no perderá el procesador durante su ejecución garantizando una correcta modificación de la misma.
Para cumplir con la propiedad de atomicidad, nuestra función insertar hace uso de la función compare exchange weak(expected , desired) la cuál recibe como parametro el valor esperado de cabeza "el valor al cuál queremos enlazar al nuevo nodo" y el valor desired que es el nuevo nodo el cuál queremos que sea la nueva cabeza en la lista atómica. Para lograrlo la función en cada iteración verifica que el valor de nuevo.siguiente coincida con el desired (esta comparación es atómica y es weak por lo tanto podría ser desalojada) y en caso de devolver verdadero se efectua el cambio de cabeza. Por el otro lado, en caso de devolver false la función actualiza el valor de nuevo siguiente por el actual de cabeza (en el caso en que se inserte una nueva cabeza) y continua hasta que devuelva true y se inserte el nodo que queríamos insertar.



2.2 Operaciones Hash Map Concurrente

2.2.1 Incrementar

En este ejercicio se nos pide implementar la función incrementar. Esta función recibe como paramétro una clave que deberá ser incrementada en caso de existir en la tabla, en caso de no existir, será agregada. Es evidente que se pueden dar condiciones de carrera cuando varios threads quieren modificar la misma lista (al existir una colision de hashes en la tabla). Para resolver esto se adopto el uso de semáforos binarios que garantizan un control sobre el acceso a la modificación de cada lista atómica. Dado que en la consigna se pide que sólo haya contención en caso de colisión de hash, optamos por no bloquear toda la tabla, sino sólamente la lista atómica correspondiente al hash reduciendo también el tiempo de espera de procesos que quieran ejecutar incrementar sobre otras claves que no colisionen entre sí.



2.2.2 Claves

La función claves devuelve un vector de todas las claves existentes en la tabla. Dado que en la consigna nos piden que sea no bloqueante y libre de inanición decidimos implementar esta función de manera que no se bloqueé toda la tabla por completo sino que únicamente la lista en la cuál se buscarán nuestras claves. Como consecuencia, si hay otros procesos intentando acceder a la tabla, solamente deberán esperar a que se libere la lista correspondiente.
*Durante la resolución del ejercicio tuvimos un debate sobre la consistencia del resultado de esta función. Dado que se podría dar la situación en que se agreguen claves en listas atómicas ya visitadas durante la ejecución de claves, al momento de retornar la función, los datos no serían exactamente los mismos que aquellos contenidos en la tabla en ese momento dado. Como respuesta a este problema se nos ocurrió bloquear toda la tabla, recorrer todas las listas y retornar todas las claves. Sin embargo esta forma de resolverlo no cumple con lo exigido en la consigna: NO BLOQUEANTE Y LIBRE DE INANICIÓN. Ya que bloquearía toda la tabla y en caso de tener muchos threads queriendo acceder a la tabla, se incurriría en largas esperas y como consecuencia en inanición.

2.2.3 Valor

La función valor recibe como parametro una clave. La función deberá devolver el valor de dicha clave en la tabla si existe, sino 0. Como condición deberemos evitar que sea bloqueante y que haya inanición. Dicho esto, solamente estamos

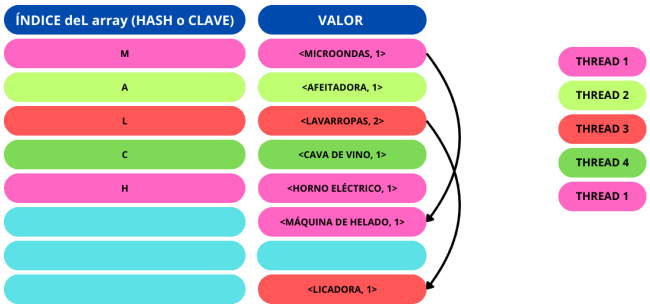
bloqueando la lista correspondiente a su hash y no toda la tabla. Esto garantizará que otros threads puedan acceder a la tabla (a excepción de la lista en la que sucede la búsqueda) sin incurrir en largas esperas.

2.2.4 Promedio

En el ejercicio 3 se pide devolver el promedio de la cantidad que tenemos de cierto producto. Esta operación requiere recorrer el HashMap en su totalidad. En el caso de querer ejecutar concurrentemente con incrementar, se podría dar el caso en que se quiera incrementar una clave en una lista de la cual promedio debe ir capturando los valores, generando cierta condición de carrera. En nuestra implementación, esto no sucede dado que incrementar deberá esperar a que se libere el mutex correspondiente a dicho hash. Sí podría suceder que se incremente una clave en una lista diferente a la que promedio está recorriendo, pero sólo en listas ya recorridas (dado que nosotros bloqueamos al inicio toda la tabla y luego se va liberando). Esto último garantiza que al momento de llamar a promedio() el valor de retorno sea consistente con el momento en el que se llamó. La idea de ir liberando garantiza que no haya tanto tiempo de espera por parte de incrementar.
Respecto al escenario mencionado en la consigna, nosotros logramos evitar modificaciones en listas aún recorridas y por tanto, un escenario de un resultado que no fue, no podría suceder. Sí podría suceder que el promedio devuelto no sea en el momento exacto en que retorna, pero se pide que incrementar pueda ejecutarse concurrentemente y por tanto no podremos bloquear completamente la tabla para garantizar dicha consistencia.

2.2.5 PromedioParalelo

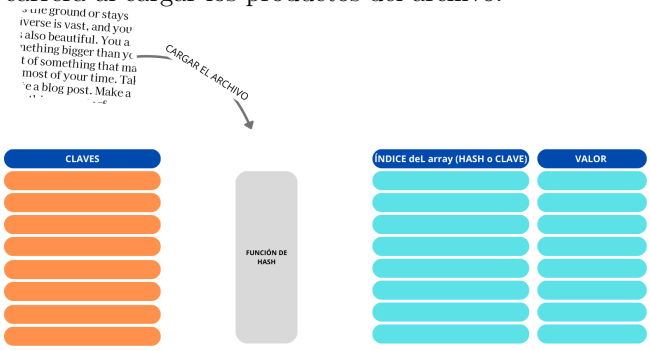
En la función PromedioParalelo se buscará obtener el Promedio con la ayuda de distintos hilos de ejecución. La idea conceptual es definir una variable de tipo int atómica que defina en qué hash deberá buscar el thread lanzado. Esto garantiza que no haya dos threads buscando en la misma lista atómica y también que aquellos lanzados tendrán algo para hacer mientras tengan el proce en su poseción. Aquellos que lean un valor por fuera del rango de la tabla, terminarán su ejecución inmediatamente. Se define una función auxiliar en la que la tarea que deben cumplir es, capturar la cantidad total de elementos distintos y la cantidad total de elementos por lista asignada. Como valor de retorno, deberán insertar un par de enteros en el vector de pares que la función promedioParalelo procesará una vez que todos los threads hayan terminado de capturar la información anteriormente descripta. Los recursos que comparten los threads son : el hashmap, el vector de resultados y el atomic int. Con la estrategia mencionada, no se modificó el tipo de dato que devuelve promedioParalelo pero sí se le exige recibir como parámetro cierta cantidad de threads para distribuir el trabajo.



3 Operaciones de Carga de Archivos

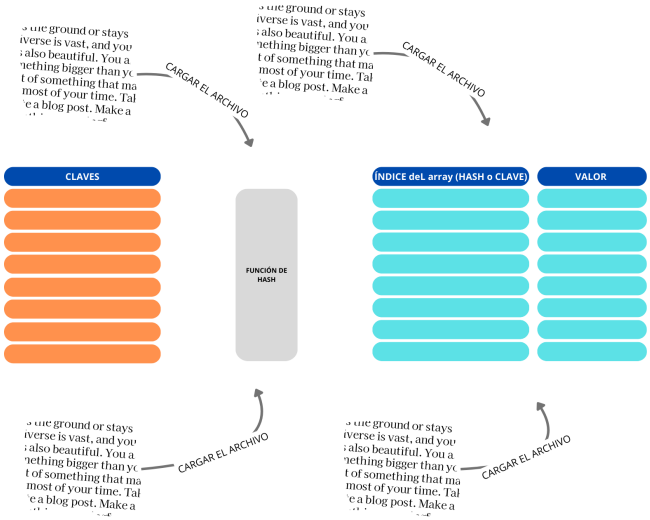
3.0.1 Cargar Archivos

En el ejercicio 4 se pide completar la implementación parcial de una función que dado un archivo carga en un hashmap los productos. Para completar la implementación simplemente hacemos uso de la función implementada en el ejercicio 2, incrementar. Como en esta función esta contemplada la concurrencia , el uso de la misma mantiene en nuestra funcion que no haya condiciones de carrera al cargar los productos del archivo.



3.0.2 Cargar Múltiples Archivos

En el caso de la función cargar múltiples archivos, hacemos uso de una función auxiliar cargar que ejecutaran nuestros threads en la que se le de adjudicara un archivo a cada uno siempre que haya archivos disponibles. Para asegurar que no haya dos threads cargando el mismo archivo hacemos uso de la variable atómica archivo que denotara el archivo a cargar.



4 Últimas consideraciones

En el contexto de un black friday donde se reciben millones de compras y la cantidad de productos sufre modificaciones constantemente se deberá garantizar que el comprador de producto disponga de su producto. Problemas de consistencia generaria ventas sin stock, con este sistema ya vimos que la función incrementar/decrementar a pesar de ejecutarse muchas veces no se perderá información. Sin embargo se podría incurrir en cierta espera. De todos modos estamos satisfechos con nuestra implementación.
Tuvimos en consideración la posibilidad de implementar un mutex para cada producto de manera de no bloquear toda la lista atómica. Dado que la cantidad de hashes es baja y la cantidad de colisiones será muy alta.
En el contexto de una tienda que vende sólo dos productos la concurrecia es vital al momento de que lleguen pedidos de compra o de incremento del stock. Por tanto, la contención será más pronunciada. Es vital implementar un modelo que evite race condition.
En el caso de querer implementar la función calcularMedianaParalela, se optaría por implementar un algoritmo de sorting en el que cierta cantidad de threads tomarían la labor de ordenar los valores. Es decir, se capturarían todos los datos en un vector compartido, en el que los threads pushearían los valores y luego cada uno tomaría cierta partición para realizar el ordenamiento. MergeSortParalelo para el ordenamiento y finalmente, mirar el dato medio del vector ordenado (en caso de ser par, sorted[medio - 1] + sorted[medio] / 2).

5 Tests

5.1 Test : PromedioParalelo

La implementación de **PromedioParalelo** explota el uso de Threads para calcular el promedio. Esta es una clara función de programación paralela que para evitar conflictos de consistencia en los valores utiliza una variable atómica. El test está pensado para cumplir con las especificaciones de la consigna en las que se pide : "Cada thread procesará una fila de la tabla a la vez, en ningún momento debe haber threads inactivos y los mismos deben terminar su ejecución si y solo si ya no quedan filas por procesar". Esto se testea al pasar una cantidad muy alta en comparación a la cantidad de listas a explorar por thread. Resultando en :

```
Running test (16): PromedioParaleloCorrecto
- Time spent during "PromedioParaleloCorrecto": 0.282959 ms
```

5.2 Test : CargarMúltipleArchivos

La implementación de **CargarMúltiplesArchivos** también implementa varios hilos de ejecución para cargarle a la tabla información contenida en archivos. La idea es análoga al caso de PromedioParalelo. En particular, los test apuntan a observar si el comportamiento que resulta de la concurrencia no concluye en condiciones de carrera.

```
Running test (18): CargarMultiplesArchivosFuncionaUnThread
- Time spent during "CargarMultiplesArchivosFuncionaUnThread": 0.596924 ms
Running test (19): CargarMultiplesArchivosFuncionaDosThreads
- Time spent during "CargarMultiplesArchivosFuncionaDosThreads": 0.134033 ms
```

6 Conclusiones

Como conclusión de este informe, se exploraron distintas técnicas para asegurar los datos en ejecuciones concurrentes, como por ejemplo el uso de mutexes y el uso de listas atómicas que también ayudan a prevenir race conditions y deadlocks. Adicionalmente, se puede apreciar un mejor rendimiento al utilizar varios threads para procesar archivos en la función cargarMultiplesArchivos optimizando el tiempo de carga de los mismos. Se pudo satisfactoriamente utilizar el HashMapConcurrente como estructura de datos de una tienda que recibe gran volumen de pedidos garantizando que la información cargada en él sea consistente y eficiente. Se garantizó en cada paso proteger la información de condiciones de carrera y simultaneamente evitar largar esperas para realizar cargas y consultas.