DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Threading

20 de Octubre de 2024 — Sistemas Operativos

Grupo 11

Integrante	LU	Correo electrónico
Melendez Rangel, Daniela Leilany del Carmen	102/20	byleilany@gmail.com
Murga, Valentino	1375/21	valenmurga23@gmail.com
Tiracchia, Thiago	1502/21	thiago.tiracchia@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
http://www.exactas.uba.ar

1 Introducción

La idea de este trabajo es implementar un Hash Map que pueda ser utilizado para cargar compras hechas a través de una tienda online y operaciones como calcular su promedio o su media. Ahora, para que se puedan hacer muchas compras a la vez vamos a necesitar que sea concurrente, lo cual va a traernos dificultades y va a llevarnos a usar los mecanismos de sincronización adecuados para evitar inconsistencias que podrían surgir a partir de condiciones de carrera. El Hash Map va a usar una Lista Atómica que va a prevenir que se pierdan elementos nuevos y luego en su propia interfaz va evitar también la pérdida de valores nuevos. Por último también vamos a tener ciertas funciones que se ocupen de la carga de archivos de compras de forma paralela para que se puedan hacer muchas compras en baches que se carguen de forma óptima.

2 Desarrollo

2.1 Lista Atómica

Llamar atómica a una Lista o una estructura de datos quiere decir que tiene operaciones las cuales no pueden ser interrumpidas por el scheduler. Por ejemplo: si hay dos o mas threads, digamos A, B; que se encuentran en simultaneo compartiendo una misma lista E y ocurre que el thread A requiere hacer una operación, en este caso, insertar un elemento en E. La ejecuta, pero antes de terminarla, el scheduler le da el procesador al thread B y este también quiere insertar en la lista E. No va a poder hacerlo hasta que el procesador vuelva a ejecutar A y termine su ejecución. Una vez terminada, el thread B (o otro thread al cual el scheduler decida darle el tiempo de ejecución) va a poder realizar la operación deseada.

Nosotros no vamos a seguir al pie de la letra esa definición ya que no es necesario y reduciría mucho los beneficios del paralelismo. Solo nos vamos a ocupar de la operación Insertar.

Algorithm 1 Insertar

```
valor
Nodo* nuevoNodo = new Nodo(valor);
nuevoNodo->siguiente = _cabeza.load();
while (!_cabeza.compare_exchange_weak(nuevoNodo->siguiente, nuevoNodo)) do
    continue
end while
```

Decidimos que la única parte que realmente sea atómica de la operación debería ser la atadura del elemento a la lista. Con esto, puede ocurrir que a medio de insertar otro thread inserte otro elemento sin el riesgo de que se pierdan nodos que es lo importante. Se podría pensar en que debería haber cierto orden entre threads pero, como se supone que los dos pidieron hacer la inserción al mismo tiempo, no debería tener importancia quien quede primero.

Para lograr esto usamos *CompareExchangeWeak*. Esta función se va a encargar de que, si alguien agregó un elemento a la lista después de que eligiéramos la cabeza vieja como la conexión con nuestro nuevo nodo, evitemos conectarlo ahí. Si hiciéramos esa conexión, el nodo que se agregó en el medio se perdería y tendríamos una condición de carrera. Con esta solución, simplemente cambiamos el nodo al que vamos a atar en el momento de forma atómica (cosa que nos asegura el lenguaje).

Algorithm 2 compare_exchange_weak (atómica)

```
cabeza, cabezaEsperada, nuevaCabeza (usamos nombres correspondientes a nuestra situación)
if cabeza == cabezaEsperada then
    cabeza = nuevaCabeza
    return true
else
    cabezaEsperada = cabeza
    return false
end if
```

2.2 Hash Map Concurrente

Queremos almacenar la cantidad de apariciones palabras en un Hash Map que va a tener una interfaz bastante común: incrementar() (que sería el equivalente a insertar() en un Hash Map común), valor() y claves(). Además también vamos a tener una operación para calcular el promedio de apariciones de las palabras pero de eso hablamos después. Nuestra estructura va a tener 26 listas (buckets), una por cada letra del abecedario (sin ñ). Una vez tenemos la interfaz, nos va a importar que se pueda usar concurrentemente, desde muchos threads a la vez. Sin embargo, eso puede traer problemas, así que vamos a tener que tomar ciertas medidas para evitar tener condiciones de carrera que nos lleven a resultados inválidos o estructuras inválidas, tratando siempre de lograr la mayor cantidad de concurrencia posible.

La operación que más problemas nos va a traer con respecto a esto va a ser incrementar() porque es la que modifica los valores y la estructura. Es por esto que vamos a querer evitar que se hagan dos incrementos en el mismo bucket al mismo tiempo. Aumentar el valor de la misma palabra al mismo tiempo podría llevar a que se pierda algún incremento, una condición de carrera normal. Para valor() y claves() decidimos no agregar ningún tipo de protección ya que son operaciones de lectura que además no condicionan ninguna escritura, por lo que no pueden tener ningún efecto negativo que lleve a condiciones de carrera.

2.3 Promedio

Ahora si, volvemos a la operación promedio(). Esta operación tiene que hacer cuentas que implican a todos los valores del mapa, los cuales pueden cambiar mientras se recorre para leerlos. En general esto no es un problema porque si se aumenta un valor por el que ya se pasó simplemente se devolvería un promedio un poco más viejo y si se aumenta uno al que todavía no se llegó el resultado va a ser un promedio más actualizado.

Sin embargo, existe una situación en la que se podría llegar a un valor que nunca fue el promedio del mapa y por lo tanto sería inválido. Ese caso es aquel en el que ocurren ambas cosas al mismo tiempo, un aumento en una clave que ya se recorrió y otro en una que todavía no fue contada. Esto nos lleva a que se cuentan algunos valores nuevos (los de las claves por las que todavía no pasamos) pero no todos (los de las claves que ya habíamos recorrido) y por lo tanto se devuelve un promedio inválido que puede no haber sido nunca un promedio real. Para evitar esto, vamos a bloquear la tabla completa en el momento que se llama a promedio y evitar que nos agreguen cosas adelante, devolviendo así el promedio del momento en el que se llamó la función. Como no queremos dejar la tabla bloqueada durante la operación entera, cosa que podría inhabilitar a nuestro sistema mucho

tiempo, vamos a querer ir destrabando cada bucket apenas podamos. Así, si bien se pueden agregar valores en las claves por las que ya pasamos, no los vamos a contar y simplemente devolveremos un promedio un poco desactualizado pero válido.

Luego, vamos a querer aprovechar el paralelismo lo máximo posible y tratar de calcular el promedio de a varios buckets a la vez con la intención de terminar más rápido, hacemos la operación promedioParalelo().

Algorithm 3 promedioParalelo

```
float sum = 0.0;
mutex mtxSum;
atomic_int count = 0;
atomic_int nextLetra = 0;
thread threads[cantThreads];
for (int i = 0; i < HashMapConcurrente::cantLetras; i++) do
    bucketMtx[i]->lock();
end for
for (unsigned int i = 0; i < cantThreads; i++) do
    threads[i] = thread(promedioThread(&sum, &mtxSum, &count, &nextLetra))
end for
for unsigned int i = 0; i < cantThreads; i++ do
    threads[i].join();
end for
if count > 0 then
    return sum / count;
else
    return 0;
end if
```

Algorithm 4 promedioThread

```
sum, mtxSum, count, nextLetra
int letra;
while ((letra = nextLetra++) < HashMapConcurrente::cantLetras) do
    for (const hashMapPair& par : *tabla[letra]) do
        mtxSum.lock();
        sum += par.second;
        mtxSum.unlock();
        count++;
    end for
    bucketMtx[letra]->unlock();
end while
```

La parte mas importante es la forma de repartir el trabajo entre los threads. Dado los N threads, van a compartir una variable atómica llamada *nextLetra* que es un entero que indica que bucket le toca a quien. De esta forma, cada thread se ocupa de un bucket a la vez y cuando termina pasa al próximo que no esté siendo procesado todavía. Van a ir sumando los valores que se encuentren en variables compartidas que van a tener que ser atómicas o protegidas con un mutex para evitar condiciones de carrera. Nosotros decidimos tener count como una variable atómica que cuenta la

cantidad de elementos distintos y sum como una variable regular que contiene la cantidad total de apariciones. Decidimos que sum no sea atómica ya que la interfaz de un float atómico es más incómoda y estamos más acostumbrados a trabajar con un mutex.

Entonces, la ejecución se ve así: el thread A va a procesar el bucket i que le tocó cuando leyó $nextLetra$. Lo primero que va a hacer es sumar un 1 a la variable $nextLetra$ para que el próximo thread B (o uno anterior que haya terminado de procesar una clave o letra) se ponga a procesar el bucket $i+1$ y así sucesivamente. Luego va a iterar sobre los elementos del bucket que van a sumar sus apariciones a la variable *sum* dentro de un mutex para evitar pérdidas. Para cada elemento distinto del bucket, aumenta la variable atómica *count* que cuenta los diferentes elementos que existen en la tabla. Cuando $nextLetra$ llega al último bucket y se termina de procesar, cada thread va a salir de su ciclo y terminar. El proceso padre que contiene a los threads espera a que todos terminen con la función $join()$ y, cuando eso ocurra, termina de calcular el promedio con una división y lo retorna.

2.4 Cargar Compras

Por último, queremos usar nuestra estructura con sus operaciones concurrentes para guardar las palabras de varios archivos a la vez. Almacenar las palabras de un solo archivo es relativamente sencillo, solo tenemos que iterar sobre el e ir haciendo incrementar() para cada palabra.

Luego, queremos hacerlo para muchos archivos. Vamos a querer aprovechar los recursos de nuestro sistema para procesar múltiples archivos al mismo tiempo y tratar de terminar más rápido que si hiciéramos uno atrás de otro. Una vez más, vamos a usar threads para esto y se van a dividir el trabajo de la misma manera que con $promedioParalelo()$ en HashMapConcurrente. Cada proceso se agarra un archivo para procesar y cuando termina pasa al próximo libre hasta que no queden más archivos sin procesar. Cuando un thread termina su archivo y no queda ningún otro para procesar, muere. La función en el proceso padre se va a quedar esperando a que todos los threads terminen para terminar. No se podría usar detach() porque si la función termina antes que los threads, la referencia al archivo que toca y a las direcciones de los archivos dejarían de ser válidas y los threads se romperían. Si no hubiera problema con las referencias, igual tendríamos el problema de que no podríamos estar seguros de cuando se terminaron de cargar todos los archivos.

2.5 Situaciones Hipotéticas

En una situación de estrés, es decir, de mucha carga y requests como por ejemplo un Black Friday, la implementación propuesta sería resistente a este caso ya que la función de $cargarMultiplesCompras$ puede elegir la cantidad de threads requeridos para poder dividir y repartir la tarea. Aun así esto aplica a cuando se quieran cargar las compras. En cambio, si se requieren hacer operaciones usando *promedio*, *claves* o *valor*, hay muchas probabilidades de que los resultados dados por estas operaciones sean desactualizados, aunque no inconsistentes ya que son resultados del estado del HashMap en un momento determinado.

Para el caso en el que hay solo dos productos, no sería tan útil tener concurrencia ya que todos los incrementos van a caer en las mismas dos variables y, para evitar condiciones de carrera, la ejecución terminaría siendo casi secuencial. Idealmente deberían poder modificarse ambas independientemente de la otra por lo que un poco de concurrecia si que estaría bien. La idea sería que aunque hayan muchos threads esperando para incrementar el valor de uno de los productos uno atrás del otro, algún thread pueda venir y aumentar el otro producto sin tener que hacer la misma fila que los

otros. Por otro lado, para CargarArchivos podría servir que se pueda hacer de a muchos threads pero, como cada thread va a tener que interactuar con esas variables, no va poder adelantarse mucho la cosa. La operación para calcular el promedio directamente perdería un poco el sentido o en todo caso su implementación se volvería trivial, sumando a ambas variables y dividiéndolas por 2.

Por último, si nos interesara calcular la mediana para nuestro Hash Map de forma concurrente, se nos ocurre que podríamos obtener los valores de todas las claves, ordenarlos y obtener el valor de la posición en el medio. Vamos a tratar de aprovechar la concurrencia en esos dos primeros pasos. Primero llegamos a una función que obtenga los valores de todas las claves de forma paralela con alguna cantidad de threads dada por parámetro. Va a hacer esto de la misma forma que con promedioParalelo, dejando que cada thread se ocupe de algún bucket libre hasta que se acaben. Una primera complicación va a ser que tenemos que almacenar los valores atómicamente en alguna estructura que luego pueda ser ordenada. Si bien en principio nuestra ListaAtómica no tiene ninguna forma de cambiar el orden de sus elementos, su propiedad de atomicidad nos conviene mucho para evitar condiciones de carrera, así que vamos a querer agregársela para después poder ordenarla. Y eso es lo que vamos a querer hacer una vez tengamos todos los valores guardados. Para agilizar el ordenamiento con concurrencia, podríamos hacer una idea como la de Merge Sort, donde la lista se parte en pedazos iguales para cada thread que los ordenan. Una vez están ordenados de forma aislada, los threads terminan y dejan al padre para que pegue los pedazos en el orden correcto. Solo faltaría quedarnos con el valor del medio pero eso es una división y el retorno. Con esta idea no debería haber ningún cuello de botella porque en ningún momento se quedan threads esperando a otros threads. El rendimiento va a depender mayoritariamente de la cantidad de threads que se elijan. Para conseguir los valores en el primer paso, cuantos más threads tengamos más rápido va a terminar, con el límite en la cantidad de buckets. Para el ordenamiento, en realidad no conviene pasarnos con los threads ya que, si quedan muchos pedazos demasiado chicos para que ordene el padre, no va a haber mucha diferencia con que ordene el vector el solo. Tampoco serviría tener muy pocos porque entonces ellos tienen que ordenar arreglos muy grandes y tampoco soluciona mucho. Habría que encontrar un punto ideal que balancee la cantidad de trabajo que hacen los threads y el padre al finalizar.

2.6 Conclusión

A lo largo del trabajo implementamos estructuras y sus operaciones de forma que puedan ser usadas simultáneamente sin tener inconsistencias en los datos que serían problemáticas. Nos encontramos con dificultades tratando de evitar dichas inconsistencias y pudimos resolverlas usando los mecanismos adecuados. Con eso logramos que los sistemas puedan funcionar paralelamente y así ahorrar mucho tiempo y aprovechar los recursos al máximo.