DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Sistemas Operativos
Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Rodrigo Laconte	193/18	rola1475@gmail.com
Julia Rabinowicz	48/18	julirabinowicz@gmail.com
Amalia Sorondo	281/18	sorondo.amalia@gmail.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

Índice

1. Introducción

En este trabajo implementaremos la estructura $HashMapConcurrente$, la cual se trata de una tabla de hash abierta que implementa un diccionario cuyas claves son palabras y los valores la cantidad de apariciones de éstas en un archivo de texto. Cada posición de la tabla representa entonces una letra del abecedario y almacena un puntero a lista enlazada de tuplas. La estructura provee además diferentes funciones para consultar y modificar su contenido. Para operar de forma concurrente sobre la tabla utilizaremos threads. La utilización de éstos permite la paralelización de las operaciones proveídas por la estructura y por lo tanto un uso más rápido y eficiente de la misma. Sin embargo, es necesario tomar recaudos en cuanto a la sincronización de los hilos ya que se pueden generar condiciones de carrera al tener éstos acceso a recursos compartidos. A lo largo de este informe estudiaremos entonces los escenarios donde pueden ocurrir conflictos entre los threads al implementar las distintas operaciones así como las soluciones consideradas. También evaluaremos cómo afecta la concurrencia al desempeño de las operaciones realizando experimentos variando la cantidad de threads utilizados y el tamaño de las instancias pasadas por parámetro.

2. Desarrrollo

2.1. Ejercicio 1

La lista es atómica, lo que significa que todas las operaciones que se le aplican no le permiten al sistema distinguir sus estados intermedios. Si por ejemplo estamos insertando un nodo, la lista pasa de no tener el elemento a poseerlo, es decir que no existe un estado de transición. Un programa que utiliza una lista atómica no presentará condiciones de carrera en cuanto a los resultados que extraiga de consultar la lista, o modificarla si los valores contenidos en los nodos son de tipo atómico. En este caso, si un proceso está modificando la lista, éste no puede ser interrumpido dejándola en un estado intermedio, luego si otro proceso consultó o modificó la lista al mismo tiempo ambos obtendrán resultados correctos. En nuestro caso, los nodos contienen un *hashMapPair* compuesto de un string y un entero, por lo que si distintos threads llaman a funciones que modifican el valor de un mismo nodo se pueden producir condiciones de carrera. Esto se puede ver en el caso de que dos threads quieran incrementar el valor de una misma clave, accediendo en consecuencia al mismo nodo. Sumarle 1 al entero de la tupla implica que se va a traer el valor de memoria, guardándolo en un registro. Luego se va a incrementar en 1 lo que haya en el registro y finalmente se va a volver a guardar en su lugar correspondiente en memoria. Puede darse el caso de que el scheduler desaloje al thread que estaba ejecutando esta operación justo antes de que llegue a actualizar el valor en memoria, y ponga a correr el otro thread que también va a sumarle 1. En este caso, al leer el valor de la clave, todavía va a ver el valor antes del incremento, por lo que cuando lo incremente y lo guarde en memoria el número original se verá aumentado en 1. Cuando vuelva a correr el primer thread, y finalmente guarde el valor que quería en memoria, este valor será el del número incrementado en 1. En consecuencia, luego de correr dos veces operaciones para incrementar en 1 un valor, el mismo se verá aumentado en 1 en lugar de 2.

Nuestra implementación de *insertar* en un principio crea un nuevo nodo y le asigna como siguiente a la cabeza de la lista. Para evitar problemas de concurrencia, el siguiente paso no podía ser simplemente poner el nuevo nodo como la cabeza de la lista, ya que podría ocurrir que otro proceso haya insertado un nodo entre esta operación y la anterior. Por eso utilizamos la función *atomic_compare_exchange*. Lo que hace es tomar la cabeza de la lista en el momento y compararla con el siguiente del nodo que queremos insertar. En caso de que sean iguales, pone el nuevo nodo como la cabeza de la lista y devuelve *true*. Si no, pone la cabeza actual de la lista como el siguiente del nodo en cuestión y devuelve *false*. La forma en que utilizamos esta función es comparar hasta que devuelva *true*, es decir, hasta que efectivamente se produzca el cambio de la cabeza de la lista por el nodo nuevo. Como la comparación con el intercambio es de manera atómica, esta función cumple con la propiedad de atomicidad.

2.2. Ejercicio 2

Al implementar las funciones *incrementar*, *claves* y *valor* consideramos distintas situaciones en las que se podrían obtener resultados inesperados y determinamos cuáles de éstas inconsistencias admitiríamos con el objetivo de limitar la contención.

En primer lugar, para evitar que en caso de colisión en la tabla creamos un arreglo donde en cada posición se almacena un mutex, cada lista enlazada de la tabla tiene entonces su acceso protegido. Al llamar a la función *incrementar* se toma el mutex de la letra correspondiente, impidiento que más de un proceso accedan de forma simultánea. De esta manera, si dos o más threads intentan incrementar concurrentemente claves que no colisionan no va a haber contención.

Por otro lado, pensamos qué situaciones podrían darse al ejecutar *incrementar* y *claves*, e *incrementar* y *maximo* de manera concurrente y vimos que no había problemas de condiciones de carrera. Analicemos el escenario en que mientras el thread que está recorriendo las listas buscando todas las claves de la tabla, otro ejecuta *incrementar* con una clave no existente aún. Si esta clave se inserta en una lista que ya había sido recorrida por el primer thread, o en la lista siendo recorrida actualmente pero en una posición anterior, esta clave no aparecerá en el resultado. En cambio si la clave se inserta en un lugar posterior que todavía no fue recorrido, entonces *sí* formará parte del conjunto solución de la función *claves*.

Consideremos ahora el caso en que un thread invoca la función *incrementar* al mismo tiempo que otro ejecuta *valor*. El valor de la clave que retorna el segundo thread puede haberse visto alterado por el primer thread, por lo que el resultado no seguirá siendo válido.

Decidimos admitir estos casos ya que no se trata de una condición de carrera sino de una inconsistencia surgida del paralelismo de las operaciones. Es decir, el resultado obtenido sería uno de los posibles resultados de la ejecución secuencial de las funciones mencionadas.

2.3. Ejercicio 3

Si se permite que las funciones *máximo* e *incrementar* se ejecuten de forma concurrente, puede pasar que el resultado de la primera se vea afectado. Esto puede ocurrir si, por ejemplo, mientras *máximo* está recorriendo una lista, en particular, está por leer el valor de una clave, el scheduler desaloja al thread que llamó a dicha función. Supongamos que la tabla de hash se encontraba vacía hasta el momento, es decir, todas las listas estaban vacías. Entonces imaginemos que los threads que pasan a ejecutarse ahora corren lo siguiente: el primero y el segundo llaman a *incrementar* de una misma clave que se agrega a una lista por la que el thread original ya pasó mientras ejecutaba *máximo*. Luego el tercero llama a *incrementar* de una clave que se agrega a una lista que el thread original todavía no recorrió. Si inmediatamente después de está última modificación a la tabla el scheduler vuelve a correr el thread que ejecutaba *máximo*, va a seguir en donde se había quedado y eventualmente leerá el valor de la clave que fue incrementada una única vez, mientras que nunca leerá el valor de la que aumentó su valor en 2. La función en este caso devolverá a la clave con valor 1 como el máximo, pero en realidad nunca ocurrió que este lo fuera, ya que el valor mayor siempre lo tuvo la clave que se incrementó dos veces.

La función *maximoParalelo* se encarga de crear una cierta cantidad de threads recibidos por parámetro, para que se ocupen de recorrer las listas de forma paralela y luego devolver la clave con la máxima cantidad de apariciones. Para ello creamos una serie de variables antes de crear los threads, las cuales son compartidas entre ellos: *n* es un `atomic<int>` que tiene la cantidad de listas a recorrer, es decir, la cantidad de letras presentes en la tabla de hash y va a representar la cantidad de listas que faltan recorrer; *marcas* es un arreglo de n posiciones en el que cada booleano va a representar si la lista fue recorrida por algún thread o no (*true* si fue recorrida, *false* si no); por úlitmo cada thread va a guardar el valor máximo encontrado para cada lista en la posición correspondiente del arreglo *maximos*. Lo que hace la función en primer lugar es bloquear todos los mutex mencionados en la sección 2.2 para no permitir la concurrencia

con *incrementar*, tal como hicimos en la función *maximo*. Luego crea la cantidad de threads recibida por parámetro, guardando los id's respectivos en el arreglo *tid*. Cuando termina de crearlos espera a que todos finalicen y hayan guardado los máximos de cada lista en el arreglo *maximos*. Por último busca el máximo entre todos los elementos del mismo y libera los mutex.

La función que corre cada thread toma como parámetros *n*, *marcas*, *maximos* y la tabla de hash. La misma va a fijarse si quedan listas por recorrer utilizando la función atómica *atomic_fetch_sub*, para evitar problemas si otro thread quiere hacer el mismo chequeo. Si no hay finaliza, y si no se fija en el arreglo *marcas* qué lista falta recorrer, utilizando la función *atomic_compare_exchange* para avisar que se va a ocupar de la lista y evitando así que otro thread recorra la misma. Luego busca el máximo de la lista de la misma manera que la función *maximo* y lo guarda en el lugar correspondiente. El arreglo *maximos* no puede sufrir de condiciones de carrera ya que cuando un thread busca qué lista recorrer lo hace con el arreglo de booleanos atómicos obteniendo un índice y luego guarda el máximo en la posición correspondiente al mismo, por lo que dos threads nunca accederán a la misma posición de dicho arreglo.

2.4. Ejercicio 4

Como en el ejercicio anterior,con el objetivo de evitar conflictos de concurrencia la función *cargarMultiplesArchivos* utiliza un entero atómico *n* que representa la cantidad de archivos a cargar y un arreglo de booleanos atómicos. Cada thread consultará la cantidad de archivos restantes y si ésta es mayor a cero, recorrerá el arreglo de booleanos buscando un archivo que no haya sido cargado. Si encuentra alguno, cambia su valor en el arreglo utilizando `atomic_compare_exchange_strong` para indicarle a otros threads que ese archivo ya no requiere ser tratado. Luego, el thread llama a la función *cargarArchivo*. No fue entonces necesario tomar recaudos especiales al implementar la última ya que al estar el estado de cada archivo protegido por un booleano atómico, ningún thread va a cargar el mismo archivo que otro. Además, como ya resolvimos el caso de la concurrencia en la función *incrementar*, que es la que van a utilizar los threads al cargar el archivo, no fue necesario tomar precauciones respecto de la sincronización en este sentido.

3. Experimentación

Para la experimentación analizamos las variaciones en el costo temporal de ejecutar *cargarMultplesArchivos* y *maximoParalelo* con diferentes cantidades de threads, y con distintas cantidades de archivos en el caso de la primera función. Los experimentos fueron corridos en las computadora con las siguientes características: versión del sistema operativo Ubuntu 18.04.4 LTS, con 12GB RAM y procesador Intel core i5-8265U CPU @1.60GHz x8.

Al momento de elaborar las hipótesis y las implementación de nuestros experimentos hubo ciertos elementos a tener en cuenta. Por un lado, como la computadora donde ejecutamos el experimento cuenta con ocho núcleos, nuestra hipótesis es que la performance de las funciones mejorará al incrementar la cantidad de threads pero al llegar a una cierta cantidad, probablemente no mucho mayor a ocho, el desempeño comenzará a disminuir. Suponemos esto ya que sólo se podrán ejecutar a la vez a lo sumo ocho threads por lo que no ahorraremos tiempo de ejecución. Igualmente, si contamos con ocho threads no podemos asegurar que el scheduler los reparta uno por núcleo. Asumimos entonces que luego de una cierta cantidad de threads estos estarán distribuidos uniformemente entre los procesadores y en este caso aumentar la cantidad de threads no supondrá una mejora sino un costo creciente de cambios de contexto.

Otra cuestión a tener en cuenta es que al ejecutar las funciones la computadora se encuentra ejecutando otros procesos por lo que el tiempo de ejecución se puede ver alterado. Con el objetivo de obtener resultados que reflejen mejor el desempeño de las funciones corremos los experimentos varias veces y luego tomamos el promedio de éstos.

3.1. Experimento cargarMultiplesArchivos

Hipótesis

Suponemos que cuando la cantidad de threads iguale la cantidad de archivos y este número sea suficiente como para que los threads se distribuyan por todos los núcleos, suponemos que el desempeño será óptimo. Es decir, vimos que no tenía sentido correr más threads que la cantidad de archivos, ya que debido a nuestra implementación de la función en cuestión los threads comienzan fijándose cuántos archivos no están a cargo de ningún otro y si no hay un archivo disponible finalizarán. Entonces, en este caso la performance de la función no mejorará en comparación al caso en que la cantidad de archivos es igual a la cantidad de threads. Si la cantidad de archivos es superior a la cantidad de threads, cada thread deberá cargar varios archivos por lo que el paralelismo mejorará el tiempo de ejecución considerablemente.

Experimento

Comprobamos si nuestras suposiciones son correctas utilizando de 1 a 12 threads para cargar archivos, y en cada caso pasamos diferentes cantidades de archivos (de 1 a 20). Esto lo hicimos 10 veces promediando los tiempos obtenidos, por las razones ya explicadas. El tamaño de los archivos lo mantuvimos fijo en todos los casos.

Análisis de resultados y conclusiones

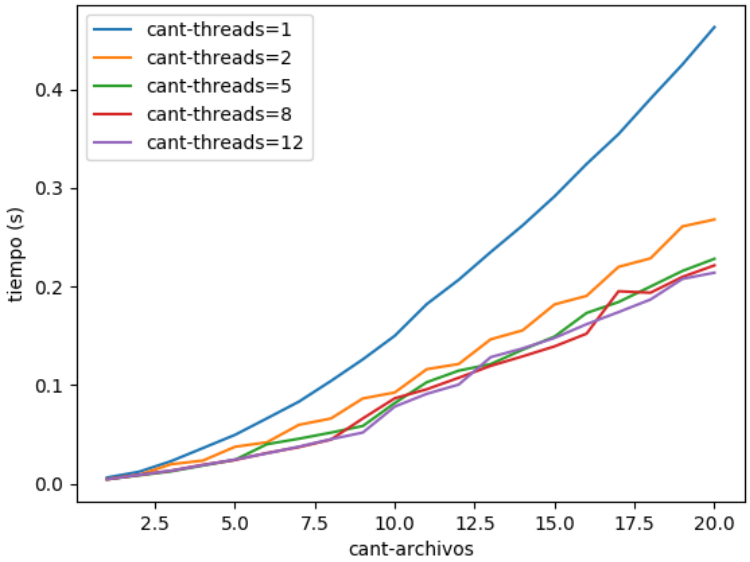

Figura 1: Tiempo de ejecución de cargarMultiplesArchivos en función de la cantidad de archivos para distinta cantidad de threads

Por motivos de claridad del gráfico y porque el resto no aportaba mayor información solo incluimos los tiempos obtenidos con ciertas cantidades de threads. Podemos observar que el tiempo de ejecución disminuye al aumentar la cantidad de threads. Sin embargo, la diferencia se hace cada vez menos significativa; el tiempo de ejecución utilizando cinco threads es muy similar al de doce. Esto contradice nuestras hipótesis acerca de la mejora hasta llegar a ocho threads. Esto puede deberse a que a partir de cuatro o cinco threads éstos logran resolver la función sin ser sacados de ejecución por lo que quedan threads que no realizan ninguna operación y por lo tanto no mejoran el tiempo total. Tampoco se puede observar una relación entre la cantidad de threads y de archivos como habíamos supuesto, atribuimos este resultado a lo explicado anteriormente.

3.2. Experimento maximoParalelo

Hipótesis

Como mencionamos anteriormente, esperamos observar un punto en el que aumentar la cantidad de threads no implica una disminución en el tiempo de ejecución. También creemos que el número de threads ”óptimo”será mayor a ocho pero no se encontrará muy alejado de esa cantidad.

Experimento

Para analizar la correctitud de nuestra hipótesis corrimos la función *maximoParalelo* con 1 a 26 threads, utilizando un archivo lo suficientemente grande (5000 palabras) como para asegurarnos de que no queden listas vacías en la tabla de hash. Al igual que al experimento anterior, lo ejecutamos 10 veces, promediando los resultados.

Análisis de resultados y conclusiones

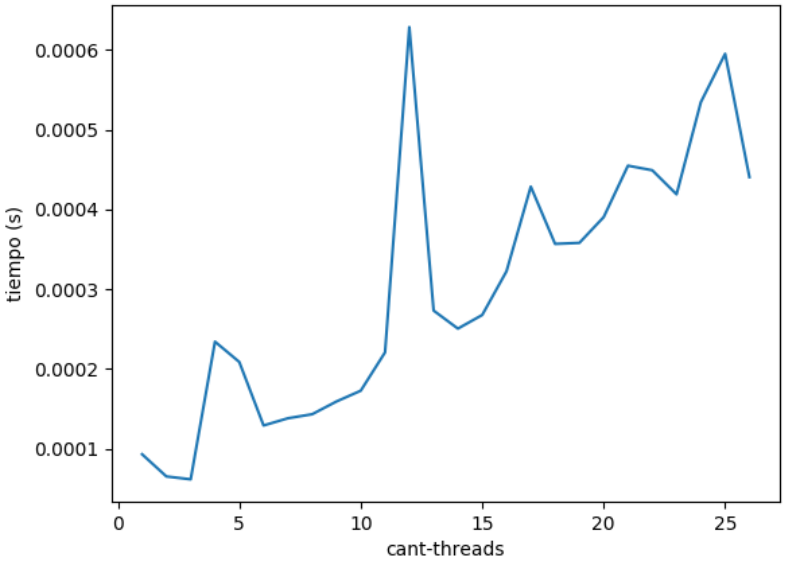

Figura 2: Tiempo de ejecución de maximoParalelo con diferentes cantidades de threads

Como podemos observar en el gráfico 2, hay efectivamente un óptimo, que parece ser 3, a partir del cual tener más threads recorriendo las listas provoca que se tarde más. Esto se corresponde con nuestra intuición inicial, aunque esperábamos que la eficiencia comenzara a decaer a partir de un número mayor. Como mencionamos, creemos que esto se debe a los costos de context switch que aumentan junto con la cantidad de threads. Igualmente notamos que la variaciones entre las diferentes cantidades son relativamente bajas.

4. Conclusión

A lo largo del trabajo describimos la implementación de la estructura *HashMapConcurrente* y los distintos métodos empleados con el objetivo de evitar condiciones de carrera al ejecutarla de forma paralela. Gracias a mutex y variables atómicas evitamos obtener resultados incorrectos al tener varios threads consultando y modificando la misma estructura simultáneamente. Analizamos qué ocurría en los casos en que se permitiera concurrencia entre diferentes funciones, decidiendo si ciertas inconsistencias que podrían resultar eran resultados erróneos o eran aceptables.

A su vez, evaluamos cómo afecta a la performance ejecutar las distintas funciones de forma concurrente, variando el nivel de paralelismo utilizado. Los resultados obtenidos nos indican que no hay una relación directa entre la cantidad de threads y el tiempo de ejecución. Es decir que no siempre que se tengan más threads disminuye el tiempo que se tarda en finalizar la función. Sin embargo, sí se pueden percibir mejoras al agregar más de un thread a la ejecución.

Como trabajo a futuro creemos que podía ser interesante analizar qué ocurre cuando se varía el tamaño de los archivos y las palabras que contienen los mismos, por ejemplo si hay muchas repetidas o que corresponden a la misma lista, ya que en esos casos entran en juego en mayor medida los mutex.