DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Pthreads

Sistemas Operativos
Primer Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Kevin Frachtenberg	247/14	kevinfra94@gmail.com
Nicolas Bukovits	546/14	nicko_buk@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

Resumen

En el siguiente trabajo práctico, se realizó una implementación de un ConcurrentHashMap el cual es una tabla de hash abierta, que significa que en caso de colisión se genera una lista enlazada dentro del bucket. Esta lista tiene la particularidad de poder ser utilizada por threads concurrentes. Su interfaz de uso es como un diccionario. Las claves son strings y los valores son enteros.

Índice

1. Resolucion de Ejercicios **3**
1.1. Ejercicio 1 . . . 3
1.2. Ejercicio 2 . . . 3
1.3. Ejercicio 3 . . . 3
1.4. Ejercicio 4 . . . 4
1.5. Ejercicio 5 . . . 4
1.6. Ejercicio 6 . . . 4

2. Detalles implementativos y pruebas **5**

1. Resolucion de Ejercicios

1.1. Ejercicio 1

En este ejercicio se completó la implementación del método push_front de la lista atómica provista por la cátedra. El método fue crear un nuevo nodo. Al nuevo nodo se le cargó como nodo siguiente el nodo inicial actualmente. Se compara atomicamente el valor del actual nodo inicial con el siguiente del nuevo nodo inicial. Si son iguales reemplaza el valor del actual nodo incial con el nuevo nodo creado. Si no son iguales se vuelve a recuperar el nodo inicial y se intenta nuevamente.

Luego se procedió a implementar la clase ConcurrentHashMap, la cual contiene un constructor que crea una tabla con 26 entradas (una por cada letra del abecedario) en la cual en cada una hay una lista de pares `<string, entero>` para que la tabla sea de hashing abierto y en caso de colisión se agregue la nueva palabra al principio de la lista. La función de Hash es simplemente tomar la primera letra de la palabra. Además de crearse e inicializarse la tabla, se crea un array de 26 mutex llamando a la función `pthread_mutex_init` para que durante el uso de threads, podamos asegurarnos que solo uno escribe en una entrada de la tabla al mismo tiempo.

Si bien el constructor no era parte de la especificación de ConcurrentHashMap, para darle consistencia a la clase se optó por implementarlo, aunque luego de terminar de desarrollar la mayor parte de los métodos solicitados, ocurrio un problema en las referencias al devolver una copia del ConcurrentHashMap asi que por sugerencia de los docentes se definió quitar el destructor.

La implementación de `addAndInc(string key)` consiste en obtener, en primer lugar, la posición en la tabla usando la función de hash. Luego, se pide el mutex de la entrada correspondiente y si la clave no esta definida se agrega al principio de la lista el par (key,0). Se realiza un unlock de la entrada y se recorre toda la lista de la posición en la tabla usando el iterador. Cuando se encuentra a la clave se pide el mutex y se incrementa en uno el valor entero del par (clave,valor). Luego se vuelve a desloquear la entrada.

El método `member(string key)` simplemente recupera la posición en la tabla y con el iterador recorre la lista. Si encuentra la clave definida devuelve verdadero.

El último método perteneciente a ConcurrentHashMap, `maximum(unsigned int nt)` : devuelve el par (k, m) tal que k es la clave con máxima cantidad de apariciones y m es ese valor. Los `nt` threads procesaran una fila de la tabla. Si no tienen filas por procesar terminarán su ejecución. La implementación consistió en definir un entero atómico y un Elem atómico que se van a usar para llevar registro de la cantidad de posiciones de la tabla a analizar y el elemento máximo encontrado. Se crean los nt threads, se pide el mutex de todas las entradas de la tabla, se cargan todas las estructuras auxiliares que van a usar los threads y se lanzan a correr los threads a los cuales se le pasa como paŕametro una función max_thread. La misma incrementa en uno atómicamente a la variable siguiente y mientras siguiente sea menos que 26, se recorre la lista en cada posicion y se va actualizando atomicamente el valor del Elem atómico que representa el máximo. Se espera a que todos los threads terminen y se hace un unlock de todas las entradas de la tabla.

1.2. Ejercicio 2

En este ejercicio se implementó una función ConcurrentHashMap `count_words(string arch)` que toma un archivo de texto y devuelve un ConcurrentHashMap cargado con las palabras del archivo. Las palabras se consideran separadas por espacio. Por especificación, no es concurrente. Su funcionamiento consiste en abrir el archivo y llamar a la función getline que toma el arhivo, un string y un caracter delimitador. Mientras haya palabras para leer se cargan en el string y por cada una de ellas se agregan al ConcurrentHashMap usando la función addAndInc implementada en el ejercicio anterior. Una vez que se hayan leido todas las palabras se cierra el archivo. Se puede asegurar que esta funcion no es concurrente ya que si bien addAndInc soporta el uso de multithreading, en esta función no se crean threads.

1.3. Ejercicio 3

En este ejercicio se implementó una función `count_words(list<string>archs)` que toma como parámetros una lista de archivos de texto y devuelve un ConcurrentHashMap cargado con las palabras. Utiliza un thread por archivo. Para ello se crean tantos pthread_t como archivos se reciben por parametro. Se crean n `pair<string*,ConcurrentHashMap*>` siendo n la cantidad de archivos. Se itera por cada archivo y se carga un `pair<string*,ConcurrentHashMap*>` para cada thread. En el

mismo se carga el archivo y la referencia de un ConcurrentHashMap. Se llama a la función `pthread_create` al cual se le pasa el pthread_t creado, la función que va a ser la que va a ejecutar el thread que es count_words_threads y la referencia al par creado. La función `count_words_threads` recibe el input del `pair<string*,ConcurrentHashMap*>` y llama a la función `count_words` con el nombre del archivo. Se espera a que todos los threads terminen llamando n veces a la función `pthread_join`. Se devuelve el ConcurrentHashMap que cargaron los threads.

1.4. Ejercicio 4

En este ejercicio se implementó una función `count_words(unsigned int n, list<string>archs)` que hace lo mismo que la anterior función pero utiliza n threads, pudiendo ser n menor que la cantidad de archivos. Similar a la función anterior pero crea n threads. Se le pasa a cada thread una estructura que contiene el nombre de los archivos y mientras no se hayan cargado todos los archivos los threads llaman a la función `count_words(file,ConcurrentHashMap)`.

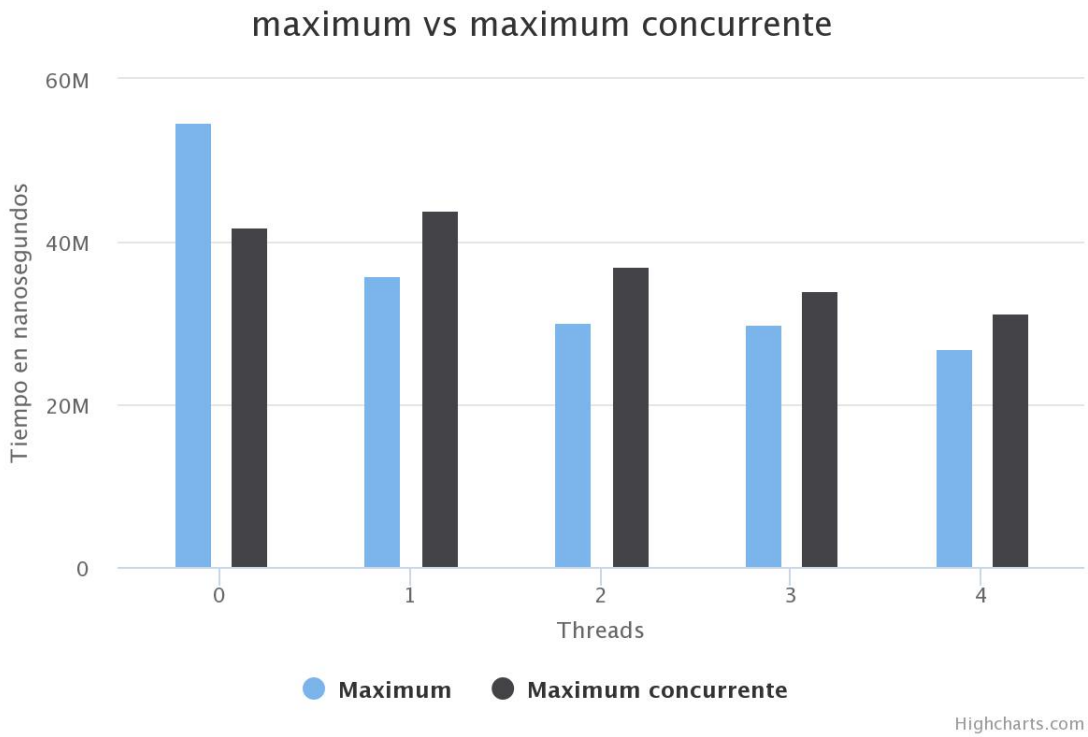
1.5. Ejercicio 5

En este ejercicio se implementó una función `maximum(unsigned int p_archivos, unsigned int p_maximos, list<string>archs)` que retorna un par con un string como primer elemento que representa la palabra que mas apariciones tiene y un entero como segundo elemento que es la cantidad de apariciones. Utiliza p_threads threads para leer los archivos y p_maximos threads para calcular los máximos. No se utilizó las versiones concurrentes de la función count_words. Similar al maximum pero crea p_archivos threads para leer los archivos que funciona como el ejercicio anterior y utiliza p_maximos threads que llenan un ConcurrentHashMap general al que despúes se le calcula el maximo usando la función maximum.

1.6. Ejercicio 6

En este ejercicio se implementó la misma funcionalidad del ejercicio anterior pero utilizando la versión concurrente de count_words del ejercicio 4. La segunda parte de la funcion, realiza el mismo procedimiento que la funcion maximum del ejercicio 5.

Para comparar resultados, creamos un archivo max-compare que sirve para ejecutar una vez cada función `maximum` con `p_archivos` y `p_maximos` pasados por parametro. Luego creamos un script en shell para ejecutar max-compare 500 veces tomando la cantidad maxima de `p_archivos` por parametro y fijando el parametro `p_maximos` en 5, ya que justamente la segunda parte de las funciones `maximum` se mantuvieron similares. Los resultados de la corrida del script con hasta 5 threads fueron los siguientes:



Se esperaba que la funcion maximum_concurrent ejecute mas rápido que la función maximum. Sin embargo luego de ver los resultados de la experimentación y analizar el código se llegó a la conclusión de que la función maximum es más rápida ya que hace un mejor uso de los threads. Se observó que a mayor cantidad de threads los tiempos de ejecución diminuían para ambas funciones.

2. Detalles implementativos y pruebas

Para seguir las consignas del enunciado en las cuales se pedia que solamente las funciones del primer ejercicio pertenezcan a la clase ConcurrentHashMap, fue necesario modificar los tests provistos por la catedra, ya que asumian que funciones como el count_words estaban en el namespace de la clase ConcurrentHashMap, cuando en realidad no.

Se agregó el archivo test-1.cpp el cual incluye tests particulares sobre ConcurrentHashMap. El archivo tiene tres partes en el cual se prueban las funciones member,addAndInc y maximum del ConcurrentHashMap, las cuales para poder ejecutarse prueban también el constructor del mismo. Para poder realizar los tests fue necesario agregar funciones auxiliares que se usan para este fin las cuales son add, Inc y count_word. Estas funciones fueron agregadas para testear member,addAndInc y maximum sin asumir que las demas funcionan excepto el maximum. Los tests se desarrollaron de manera incremental empezando con member, luego una vez que se testeo correctamente member, se testeó addAndInc y finalmente maximum.