



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	3
1.5. Ejercicio 5	3
1.6. Ejercicio 6	4
2. Detalles implementativos	4

1. Resolución 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 atómicamente el valor del actual nodo inicial con el siguiente del nuevo nodo inicial. Si son iguales reemplaza el valor del actual nodo inicial con el nuevo nodo creado. Si no son iguales se vuelve a recuperar el nodo inicial y se intenta nuevamente. Luego de haber completado la implementación de la lista se debe implementar la clase `ConcurrentHashMap` con las siguientes especificaciones:

`ConcurrentHashMap()`: Constructor. Crea la tabla. La misma tendrá 26 entradas (una por cada letra del abecedario 1). Cada entrada consta de una lista de pares (string, entero). La función de hash será la primer letra del string. `void addAndInc(string key)`: Si key existe, incrementa su valor, si no existe, crea el par (key, 1). Se debe garantizar que sólo haya contención en caso de colisión de hash. Esto es, deberá haber locking a nivel de cada elemento del array. `bool member(string key)`: true si y solo si el par (key, x) pertenece al hash map para algún x. Esta operación deberá ser wait-free. `pair(string, unsigned int) 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. No puede ser concurrente con `addAndInc`, si con `member`, y tiene que ser implementada con concurrencia interna. El parámetro nt indica la cantidad de threads a utilizar. Los threads procesarán una fila del array. Si no tienen filas por procesar terminarán su ejecución.

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. Debe ser no concurrente. Para realizar lo pedido la función abre el archivo y llama a la función `getline` que toma el archivo, un string y un caracter delimitador. Mientras haya palabras para leer se van cargando 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 leído todas las palabras se cierra el archivo.

1.3. Ejercicio 3

En este ejercicio se implementó una función `count_words(liststring&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 paristring*, ConcurrentHashMap*` siendo n la cantidad de archivos. Se itera por cada archivo y se carga un `paristring*, 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 `paristring*, 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, liststring&archs)` que hace lo mismo que la anterior función pero utiliza n threads, pudiendo ser n menor que la cantidad de archivos.

1.5. Ejercicio 5

En este ejercicio se implementó una función `maximum(unsigned int p archivos, unsigned int p maximos, liststring&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`.

1.6. Ejercicio 6

En este ejercicio se implementó la misma funcionalidad del ejercicio anterior pero utilizando las versiones concurrentes de `count.words`.

2. Detalles implementativos

La clase `ConcurrentHashMap` es una clase compuesta de un array de listas enlazadas que representan la tabla de Hash y un array de locks, ambos arrays de tamaño 26 ya que es la cantidad de letras posibles.

Al comienzo del trabajo práctico, mientras implementábamos el constructor, se nos ocurrió también implementar el destructor, para que de esa forma la clase sea coherente. Sin embargo, al desarrollar la implementación de `ConcurrentHashMap count_words(string archivo)` observamos que lo que devolvía la función era un `ConcurrentHashMap` con basura o vacío, a pesar de que dentro de la función construía bien el diccionario. Luego de investigar el problema, llegamos a la conclusión de que el problema se daba al momento de realizar la copia del resultado: primero se llamaba al destructor y luego se copiaban las referencias, y por eso se generaba el `ConcurrentHashMap` con basura o vacío. Las soluciones posibles en ese momento eran redefinir la copia o quitar el destructor, pero por sugerencia de un docente, decidimos eliminar el destructor.