Sistemas Operativos – TP1: pthreads

Angel Yan – 230/16
Guido Navalesi – 120/11
Leandro Yampolski – 588/08

Primer cuatrimestre 2018

1. Ejercicio 1

Definimos como *item* a un par $(string, entero)$ y como *bucket* a una lista enlazada de *items*.

1.1. Estructura de ConcurrentHashMap

La clase cuenta con un miembro público `bucket* tabla[26]` que guardará los buckets correspondientes a cada clave. En un miembro privado `mutex *bucket_mutex[26]` se almacenará por cada bucket un mutex que permitirá aplicar exclusión mutua sobre ellos (ver [1]).

Otros miembros privados son `unsigned int hash_key(string *key)`, que mapea la primera letra de `key` a $\{0, 1, \cdots, 25\}$ de forma biyectiva, `structs` de parámetros y funciones de threads, y otras funciones auxiliares.

1.2. Constructor y destructor

El constructor simplementa crea los 26 buckets y los correspondientes mutexs y el destructor los elimina liberando su memoria.

1.3. `void addAndInc(string key)`

```
1 void addAndInc(string key) {
2     Obtener el bucket correspondiente a la clave
3     Activar el lock del bucket
4     Buscar la clave en el bucket
5     Si se encuentra la clave, incrementar la cuenta
```

```
    Si no se encuentra la clave, agregarlo al bucket con la cuenta en 1
    Desactivar el lock del bucket
}
```

El locking del bucket antes de realizar la búsqueda e incrementar/iniciar el contador garantiza que sólo exista contención en caso de colisión de hash.

1.4. bool member(string key)

```
bool member(string key) {
    Obtener el bucket correspondiente a la clave
    Buscar la clave en el bucket
    Devolver si se encuentra la clave
}
```

Como no hay locking en la función, esta es wait-free.

1.5. item maximum(unsigned int nt)

La función crea threads que ejecutan la función `maximum_thread_function` y reciben sus argumentos a través de la estructura `maximum_thread_args`.

```
item maximum(unsigned int nt) {
    atomic<int> index = 0;
    item maximo_global = item("", 0);
    mutex maximo_global_mutex;
    Activar el lock de todos los buckets
    Crear nt threads {
        int i = index.getAndInc();
        Mientras i < 26 {
            Guardar en maximo_local el item con la máxima cuenta del bucket i
            i = index.getAndInc();
        }
        Si se encontró un maximo_local {
            Activar el lock de maximo_global
            Actualizar maximo_global con el máximo entre maximo_global y maximo_local
            Desactivar el lock de maximo_global
        }
    }
    Esperar a que terminen los threads
    Desactivar el lock de todos los buckets
    Devolver general_maximum
}
```

El uso de `atomic<int> index` le permite a cada thread obtener un único índice correspondiente a un bucket sin procesar en cada iteración del ciclo.

El locking de todos los buckets antes de buscar el máximo garantiza que la búsqueda no sea concurrente con la función `void addAndInc(string key)`.

Los threads procesan buckets únicos hasta que no quede ninguno. Cuando a un thread no le quedan buckets, si encontró un máximo local, se activa el locking de la variable compartida que guarda el valor máximo global y se actualiza su valor. Esto evita las condiciones de carrera sobre esta variable.

2. Ejercicio 2

2.1. `void push_front(const T& val)`

```
1 void push_front(const T& val) {
2     Nodo *newHead = new Nodo(val);
3     newHead->_next = _head;
4     while (!_head.compare_exchange(newHead->_next, newHead));
5 }
```

La función crea un nuevo nodo `newHead` con el valor de `val` y luego lo enlaza con la cabeza de la lista `_head`. Luego, para insertar este nuevo nodo en la lista, se usa el método `compare_exchange`. Este método es atómico y funciona como un compare-and-swap, pero además si `_head` es distinto de `newHead->_next`, guarda en este último el valor actual de `_head`. Coloquialmente, el método observa si el valor de `_head` es el nodo al que apunta `newHead`, en cuyo caso pone a `newHead` como nuevo `_head`. Si no lo es, quiere decir que otro proceso cambió el valor de `_head` y entonces corrige el valor el `newHead->next` por este nuevo `_head`. Después, por el ciclo, vuelve a intentar.

2.2. `ConcurrentHashMap count_words(string arch)`

```
1 ConcurrentHashMap count_words(string arch) {
2     ConcurrentHashMap map;
3     Leer el archivo arch guardando cada palabra en una lista de
      ↪ strings
4     Por cada palabra de la lista hacer map.addAndInc(palabra)
5     return map
6 }
```

2.3. ConcurrentHashMap count_words(list<string> archs)

```
ConcurrentHashMap count_words(list<string> archs) {
    ConcurrentHashMap map;
    list<string>::iterator iterador = archs.begin();
    mutex iterador_mutex;
    Crear un thread por cada elemento de archs {
        Activar lock de iterador
        Guardar iterador en it
        Si iterador != archs.end(), avanzar iterador
        Desactivar lock de iterador
        Si it != archs.end(), contar en map las palabras del archivo
        ↪ apuntado por it con addAndInc
    }
    Esperar a que terminen los threads
    return map
}
```

Hacer locking de iterador asegura que cada thread obtenga un archivo único de la lista de archivos.

2.4. ConcurrentHashMap count_words(unsigned int n, list<string> archs)

```
ConcurrentHashMap count_words(list<string> archs) {
    ConcurrentHashMap map;
    list<string>::iterator iterador = archs.begin();
    mutex iterador_mutex;
    Crear n threads {
        bool quedan_archivos = true;
        Mientras quedan_archivos {
            Activar lock de iterador
            quedan_archivos = iterador != archs.end();
            Si quedan_archivos {
                Guardar en arch el archivo apuntado por iterador
                Avanzar iterador
            }
            Desactivar lock de iterador
            Si quedan_archivos, contar en map las palabras del
            ↪ archivo apuntado por it con addAndInc
        }
    }
    Esperar a que terminen los threads
    return map
}
```

Hacer locking de iterador asegura que cada thread obtenga un archivo único de la lista de archivos. El ciclo hace que los threads sigan procesando archivos siempre que queden archivos sin procesar.

2.5. item maximum(unsigned int p_archivos, unsigned int p_maximos, list<string> archs)

```
1  item maximum(unsigned int p_archivos, unsigned int p_maximos,
   ↪  list<string> archs) {
2      list<ConcurrentHashMap*> maps;
3      Insertar un ConcurrentHashMap* en maps por archivo en archs
4      list<ConcurrentHashMap>::iterator maps_iterador = maps.begin();
5      list<string>::iterator archs_iterador = archs.begin();
6      mutex iteradores_mutex;
7      Crear p_archivos threads {
8          bool quedan_archivos = true;
9          Mientras quedan_archivos {
10             Activar lock de los iteradores
11             quedan_archivos = arch_iterador != archs.end();
12             Si quedan_archivos {
13                 Guardar en arch el archivo apuntado por arch_iterador
14                 Avanzar arch_iterador
15                 Guardar en map el ConcurrentHashMap* apuntado por
                    ↪  maps_iterador
16                 Avanzar map_iterador
17             }
18             Desactivar lock de los iteradores
19             Si quedan_archivos, contar en *map las palabras del
                ↪  archivo apuntado por it con addAndInc
20         }
21     }
22     Esperar a que terminen los threads
23     Juntar los mapas de maps en un mapa general_map
24     return general_map.maximum(p_maximos);
25 }
```

Hacer locking de los iteradores asegura que cada thread obtenga un archivo y un mapa único.

3. Tests

3.1. test-1

El test `text-1` se encarga de verificar la correctitud de las funciones `void addAndInc(string key)`, `bool member(string key)` y `item maximum(unsigned int nt)`. En él se crea un `ConcurrentHashMap` y se agregan algunas palabras con `addAndInc`, verificando luego de cada llamado el valor de `member`. Para `maximum` se comprueba su valor usando un solo thread y también usando un cantidad de threads mayor a la cantidad de buckets.

3.2. test-4

En el test `test-4` se prueba la concurrencia de `addAndInc` creando varios threads que agregan palabras a un mismo `ConcurrentHashMap`, verificando la pertenencia al map con `member` y corroborando, una vez que terminan los threads, que `maximum` devuelve el valor correcto.

3.3. test-6

El test `test-6` tiene el objetivo de poner a prueba la concurrencia de las distintas funciones implementadas. Se crean varios threads que agregan palabras a un mapa propio y a otro mapa compartido por todos, usando distintas cantidades de threads para las funciones concurrentes como `count_words` y `maximum`. La idea es tener una gran cantidad de threads corriendo concurrentemente para tratar de exponer errores. Al final se comprueba si es correcto el valor de `maximum` de cada uno de los mapas.

3.4. test-7

El fin de este test es comparar el tiempo de ejecución de las dos versiones de la función `item maximum(unsigned int p_archivos, unsigned int p_maximos, list<string> archs)`: la que usa un mapa por thread y luego los combina en uno (`maximum`) y la que llama a `count_words` (`maximum2`).

Se espera que `maximum` sea más lento que `maximum2` ya que el primero crea varios mapas que luego debe combinar, mientras que el segundo cuenta las palabras con sólo un mapa.

Para la prueba se ejecutaron las funciones para una lista de mil archivos, usando valores de `p_archivos` y `p_maximos` de 1 a 5.

En la figura 1 se puede ver efectivamente que el tiempo de ejecución de `maximum` es mayor al de `maximum2` en todos los casos, llegando a ser más de 50 % más lento.

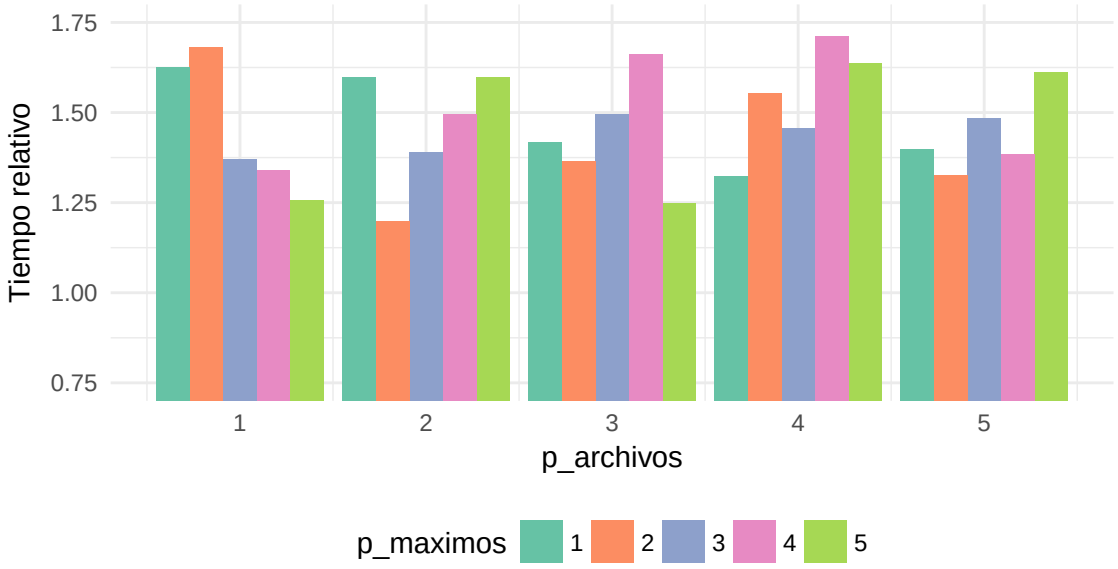

Figura 1: Tiempo de `maximum` relativo al tiempo de `maximum2`

3.5. test-8

El test `test-8` tiene el objetivo de verificar si la función `void addAndInc(string key)` implementa correctamente el mecanismo de locking que debería hacer exclusión mutua en caso de colisión únicamente.

Para hacer esto se crean dos listas de palabras de igual tamaño, cada una con una misma palabra repetida, pero que palabras de distintas las listas correspondan a buckets distintos (por ejemplo una lista <arbol, arbol, ..., arbol> y otra <bolso, bolso, ..., bolso>). La razón por la cual se repite la misma palabra es que garantiza que `addAndInc` tenga un tiempo de ejecución constante.

Primero se crean dos threads que agregan la *misma* la lista *cada uno* a un `ConcurrentHashMap` y se mide el tiempo que demoran. Luego se crean otros dos threads que agregan cada uno una lista *distinta* a un `ConcurrentHashMap` y se mide el tiempo que tarda. La idea es que cuando los dos threads agregan una misma lista el locking forzará un agregado secuencial; mientras que cuando agregan listas distintas los dos threads pueden agregar palabras concurrentemente ya que no hay colisión entre threads. Luego, los threads que agregan la misma lista deberían tardar mucho más.

Corriendo este test con listas de 100.000 palabras se obtuvo que los threads que agregan listas distintas demoran entre el 30 % y 34 % de lo que tardan los threads que agregan la misma lista. Esto confirma entonces que la implementación del locking es correcta.

3.6. test-9

Este test verifica que `void addAndInc(string key)` y `item maximum(unsigned int nt)` no sean concurrentes. Primero se carga un `ConcurrentHashMap` con palabras y se mide el tiempo que se tarda en llamar secuencialmente addAndInc y maximum una cierta cantidad de veces. Luego, se carga otro mapa con las mismas palabras y se mide el tiempo que se tarda en hacer lo mismo pero esta vez con dos threads, uno que llama a addAndInc y otro a maximum, la misma cantidad de veces. Se espera que los tiempos obtenidos sean aproximadamente el mismo.

El test dio como resultado que la ejecución con threads tarda entre 2 % y 5 % más que la ejecución secuencial. Este pequeño incremento puede atribuirse al overhead que conlleva el uso de threads. Con esto se comprueba entonces que las dos funciones no son concurrentes.

Referencias

[1] Clase `mutex` de `C++`. http://en.cppreference.com/w/cpp/thread/mutex

[2] Clase `atomic` de `C++`. http://en.cppreference.com/w/cpp/atomic/atomic