

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 $\{0, 1, \dots, 25\}$ de forma biyectiva, `structs` de parámetros y funciones de threads, y otras funciones auxiliares.

1.2. Constructor y destructor

El constructor simplemente 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
```

```

6     Si no se encuentra la clave, agregarlo al bucket con la cuenta en
    ↪ 1
7     Desactivar el lock del bucket
8 }

```

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)`

```

1 bool member(string key) {
2     Obtener el bucket correspondiente a la clave
3     Buscar la clave en el bucket
4     Devolver si se encuentra la clave
5 }

```

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`.

```

1 item maximum(unsigned int nt) {
2     atomic<int> index = 0;
3     item maximo_global = item("", 0);
4     mutex maximo_global_mutex;
5     Activar el lock de todos los buckets
6     Crear nt threads {
7         int i = index.getAndInc();
8         Mientras i < 26 {
9             Guardar en maximo_local el item con la máxima cuenta del
            ↪ bucket i
10            i = index.getAndInc();
11        }
12        Si se encontró un maximo_local {
13            Activar el lock de maximo_global
14            Actualizar maximo_global con el máximo entre
            ↪ maximo_global y maximo_local
15            Desactivar el lock de maximo_global
16        }
17    }
18    Esperar a que terminen los threads
19    Desactivar el lock de todos los buckets
20    Devolver general_maximum
21 }

```

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.exchange(newHead);
4 }
```

Como `_head.exchange(newHead)` cambia el valor de `_head` por el de `newHead` y devuelve su valor anterior de forma atómica, no hay condiciones de carrera sobre esta variable y `newHead` pasa a ser el primer elemento de la lista. Dado que la asignación de `newHead->_next` no es atómica, no se puede garantizar que inmediatamente después del `exchange` `newHead` esté enlazada al valor que tenía `_head` antes del `exchange`; es posible que en este período se agregue otro nodo delante de `newHead`. Pero dado que `_head.exchange(newHead)` devuelve el valor anterior de `_head` atómicamente, la asignación lo guardará en `newHead->_next` eventualmente y los nodos quedarán correctamente enlazados.

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
4     ↪ strings
5     Por cada palabra de la lista hacer map.addAndInc(palabra)
6     return map
7 }
```

2.3. `ConcurrentHashMap count_words(list<string> archs)`

```
1 ConcurrentHashMap count_words(list<string> archs) {
2     ConcurrentHashMap map;
```

```

3     list<string>::iterator iterador = archs.begin();
4     mutex iterador_mutex;
5     Crear un thread por cada elemento de archs {
6         Activar lock de iterador
7         Guardar iterador en it
8         Si iterador != archs.end(), avanzar iterador
9         Desactivar lock de iterador
10        Si it != archs.end(), contar en map las palabras del archivo
        ↪ apuntado por it con addAndInc
11    }
12    Esperar a que terminen los threads
13    return map
14 }

```

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)

```

1  ConcurrentHashMap count_words(list<string> archs) {
2      ConcurrentHashMap map;
3      list<string>::iterator iterador = archs.begin();
4      mutex iterador_mutex;
5      Crear n threads {
6          bool quedan_archivos = true;
7          Mientras quedan_archivos {
8              Activar lock de iterador
9              quedan_archivos = iterador != archs.end();
10             Si quedan_archivos {
11                 Guardar en arch el archivo apuntado por iterador
12                 Avanzar iterador
13             }
14             Desactivar lock de iterador
15             Si quedan_archivos, contar en map las palabras del
            ↪ archivo apuntado por it con addAndInc
16         }
17     }
18     Esperar a que terminen los threads
19     return map
20 }

```

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.

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>

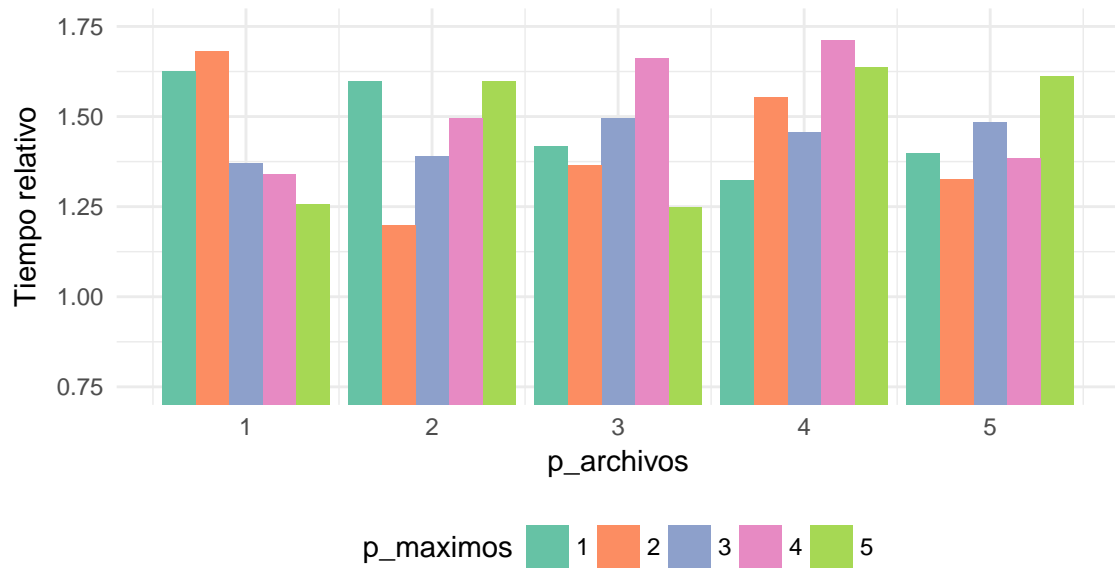


Figura 1: Tiempo de `maximum` relativo al tiempo de `maximum2`