DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Programación concurrente mediante pthreads

Sistemas operativos

Primer Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Berríos Verboven, Nicolás	046/12	nbverboven@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

1. Implementación del ConcurrentHashMap

Se implementó una tabla de hash abierta consistente en un arreglo de 26 posiciones (una por letra del abecedario), cada una de las cuales tiene una lista enlazada para resolver las colisiones de cada palabra que empiece con la misma letra.

Se definieron un constructor por copia y un operador de asignación para los casos provistos por la cátedra. Después, nos pareció conveniente agregar ciertas funciones en la parte protegida de la clase las cuales nos ayudan con la implementación de los ejercicios y no quedan expuestas a la utilización de otras clases. Por ejemplo, un mutex para la función addAndInc.

1.1. addAndInc

De acuerdo a la especificación, implementamos la función para que busque la palabra en el diccionario, bloqueando mediante un mutex el acceso a la fila dada por el resultado de aplicar la función de hash al elemento que se desea agregar. De esta forma, no puede ocurrir que dos threads accedan simultáneamente al mismo ítem. Si la palabra estaba definida, se aumenta su cantidad; si no, se agrega. El bloqueo de filas se logra con un arreglo de 26 mutex, donde el mutex en la i-ésima posición corresponde a la i-ésima fila de la tabla de hash.

1.2. member

Esta función genera un iterador a la fila del HashMap determinada por la primera letra de la clave que desea buscar. Luego, itera sobre dicha lista hasta encontrarla (en cuyo caso devuelve `true`) o quedarse sin elementos por recorrer. En este último caso, devuelve `false`.

Observando el código, puede verse que en ningún momento se realiza un bloqueo o se llama a alguna función que pudiera bloquearse. En consecuencia, dado el caso de un proceso/thread que quisiera acceder a la sección crítica (que en este caso sería la totalidad de la función) podría hacerlo, asumiendo que el scheduler no lo relegara indefinidamente (es decir, asumiendo *fairness*). Entonces, se cumple que en cualquier momento de alguna ejecución del programa, si un proceso intenta acceder a la sección crítica, existirá un instante posterior en el cuál lo conseguirá. Esto es, la función `member` es *wait-free*.

1.3. maximum

Para evitar la concurrencia con addAndInc implementamos dos semáforos que corresponden a la cantidad de threads que se encuentran ejecutando cada una de las dos funciones. Ambos se inicializan en 0 apenas entra a la función y dependiendo cuál es ejecutada primero (si maximum o addAndInc) la otra esperará hasta que la misma termine.

Dado que se utilizaron *nt* threads para procesar las filas del HashMap, se recurrió a un entero atómico para indicar la siguiente fila a recorrer. De esta forma, se garantiza que solo un thread acceda a cada fila.

maximum(**in** nt : `int`)

```
1: threads ← CrearVector < thread > (nt)
2: fila_actual ← Atomic < int > 0
3: resultados_filas ← CrearVector < pair < string, int >> ()
4: datos_para_thread ← CrearEstructuraDatosThread()
5: datos_para_thread.resultados ← &resultados_filas
6: datos_para_thread.hash_map ← this
7: datos_para_thread.actual ← &fila_actual
8: datos_para_thread.cant_threads = nt
9:
10: for tid ← 0 to nt do
11:     CrearThread(&threads[tid], masAparicionesPorFila, &datos_para_thread)
12: end for
13:
14: for tid ← 0 to nt do
15:     JuntarThread(threads[tid])
16: end for
17: solucion ← ('', 0)
18: for i ← 0 to resultados_filas.size() do
19:     if resultados_filas[k].second > solucion.second then
20:         solucion ← resultados_filas[k]
21:     end if
22: end for
23:
24: return solucion
```

2. Desarrollo de Ejercicios

2.1. Ejercicio 1

Para completar la implementación del push_front utilizamos la función *compare_exchange_strong* para evitar que dos procesos actualicen simultáneamente el valor del primer elemento de la lista. Lo que hace la función es (atomicamente, dado que *_head* es atómica) devolverte la comparación entre *_head* y el primero de la lista, en caso de que sean iguales pone al nuevo item como primero y en el caso que no sean pone en la variable *primero* al nuevo primero. Ahora mientras sea distinto, como se encuentra en un while, vuelve a setear en la variable $nuevo- > \_next$ al nuevo primero (seteado por *compare_exchange_strong*).

push_front(**in** val : `T`)

```
1: primero ← _head.load()
2: nuevo ← newNodo(val)
3: (nuevo → _next) ← primero
4: while ¬ _head.compare_exchange_strong(primero, nuevo) do
5:     (nuevo → _next) ← primero
6: end while
```

2.2. Ejercicio 2

Para esta función, dado que no es concurrente es bastante fácil el procedimiento. Inicializo un HashMap nuevo, abro el archivo y después recorro las lineas y las palabras (por si hay mas palabras por linea) y las agrego al HashMap.

```
count_words( in arch : string )
 1: dicc ← ConcurrentHashMap()
 2: file ← open(arch)
 3:
 4: while line ← getLine(file) do
 5:     while word ← getWord(line) do
 6:         dicc.addAndInc(word)
 7:     end while
 8: end while
 9:
10: return dicc
```

2.3. Ejercicio 3

Para la versión de count_words concurrente primero creo el HashMap que voy a mandar a todos los threads y como el addAndInc tiene un locking a nivel elemento (mejor dicho, a nivel de fila) voy a saber que no van a poder incrementar una palabra a la vez por 2 threads distintos. Después inicializo una lista de threads que voy a usar por cada archivo y una de datos que voy a mandar a cada thread.

Ahora solo me falta recorrer la cantidad de threads que voy a crear, asignarle un archivo a cada thread y crearlo.

Cada thread va a ejecutar la misma función (CountWordsByFile) que lo que hace es lo mismo que en el ejercicio 2, salvo que sin crear el HashMap y utilizar el pasado con la estructura de datos. La función abre el archivo y guarda las palabras en el HashMap.

Por último solo tengo que recorrer de vuelta la cantidad de threads para juntar todos cuando terminen y devolver el HashMap.

```
count_words( in archs : lista<string> )
 1: dicc ← ConcurrentHashMap()
 2: cantThreads ← archs.size()
 3: threads ← CrearLista < thread > (cantThreads)
 4: thread_data ← CrearLista < Thread_data > (cantThreads)
 5:
 6: for tid ← 0 to cantThreads do
 7:     therads_data[tid].thread_id ← tid
 8:     therads_data[tid].file ← archs.getNext()
 9:     therads_data[tid].hash_map ← &dicc
10:     CrearThread(&threads[tid], CountWordsByFile, &threads_data[tid])
11: end for
12:
13: for tid ← 0 to cantThreads do
14:     JuntarThread(threads[tid])
15: end for
16:
17: return dicc
```

2.4. Ejercicio 4

En esta versión de count_words hago algo parecido salvo que en vez de mandarle a cada thread una estructura de datos distinta con el mismo HashMap, le mando la misma a todos y la estructura cambia. La misma va a tener la lista de archivos a leer, el HashMap (mismo para todos) y un entero atómico que indica el elemento de la lista a leer. De este modo, voy a poder controlar que no se lea 2 veces el mismo archivo.

count_words(**in** n : `int`, **in** *archs* : `lista<string>`)

```
1: if archs.size() < n then
2:     cantThreads ← archs.size()
3: else
4:     cantThreads ← n
5: end if
6: dicc ← ConcurrentHashMap()
7: threads ← CrearLista < thread > (cantThreads)
8: thread_data ← CrearTheadsData_countWords_list(&dicc, archs, 0)
9:
10: for tid ← 0 to cantThreads do
11:     CrearThread(&threads[tid], CountWordsByFileList, &thread_data)
12: end for
13:
14: for tid ← 0 to cantThreads do
15:     JuntarThread(threads[tid])
16: end for
17:
18: return dicc
```

CountWordsByFileList(**in/out** *thread_data* : `ThreadData_mutex`)

```
1: hasFile ← true
2: while hasFile do
3:     i ← getAndInc(thread_data → actual)
4:     if thread_data.files.vacio() then
5:         hasFile ← false
6:     else
7:         filename ← (thread_data → files).getNext()
8:     end if
9:     if hasFile then
10:        Leo el i-ésimo archivo de la lista y meto todas las palabras al HashMap
11:    end if
12: end while
```

2.5. Ejercicio 5

Procedo a explicar el desarrollo de la función directamente el pseudocodigo, indicando lo que hago en cada paso.

maximum(**in** *p_archivos* : `int`, **in** *p_maximos* : `int`, **in** *archs* : `list<string>`)

1: Copio los elementos de *archs* en un vector para reducir el costo de acceder a cada posición.
2: Inicializo una estructura *thread_data_read* la cual va a guardar el vector de archivos, un puntero a un HashMap global (uno para todos los threads), un int atómico que indica por que archivo del vector estoy y un mutex para actualizar el HashMap resultante de a un thread a la vez.
3: **for** *tid* ← 0 **to** *p_archivos* **do**
4: Creo los threads con la función *leoArchivos* y le paso la estructura que creé anteriormente.
5: **end for**
6: **for** *tid* ← 0 **to** *p_archivos* **do**
7: Junto todos los threads.
8: **end for**
9: Llamo a la función *maximum* del HashMap global y le paso como parámetro la cantidad de threads *p_maximos*.
10: Devuelvo el máximo que retornó la función anterior.

leoArchivos(**in/out** *thread_data* : `thread_data_read`)

```
1: HashMap ← CrearHashMap()                                   ▷ Creo un HashMap
2: while (i ← thread_data.actual.fetch_add(1)) < cant do      ▷ actual es el int atómico
3:     archivo ← thread_data.archivos_a_leer[i]
4:     tempHashMap ← count_words(archivo)                     ▷ Versión no concurrente
5:     for tuplaPalabra in temHashMap do ▷ Copio las palabras de temHashMap en HashMap
6:         for cantRepeticiones ← 0 to tuplaPalabra.second do
7:             HashMap.addAndInc(tuplaPalabra.first)
8:         end for
9:     end for
10: end while
11:
12: thread_data.mtx.lock()                                    ▷ Mutex, entro zona crítica
13: thread_data.resultados.push_back(HashMap)                 ▷ Actualizo el HashMap global
14: thread_data.mtx.unlock()                                  ▷ Mutex, salgo zona crítica
```

2.6. Ejercicio 6

maximum2(**in** *p_archivos* : `int`, **in** *p_maximos* : `int`, **in** *archs* : `list<string>`)

```
1: hashMap ← count_words(archs)          ▷ Versión concurrente en archs.size() threads
2: res ← hashMap.maximum(pmaximos)
3: return res
```

maximum3(**in** *p_archivos* : `int`, **in** *p_maximos* : `int`, **in** *archs* : `list<string>`)

```
1: hashMap ← count_words(p_archivos, archs)   ▷ Versión concurrente en p_archivos threads
2: res ← hashMap.maximum(pmaximos)
3: return res
```

3. Desarrollo experimental

Se realizaron dos experimentos para estudiar la variación del tiempo de ejecución de la función `maximum(uint p_archivos, uint p_maximos, list<string>archs)` en relación con la cantidad de threads utilizados para el procesamiento de archivos y para la búsqueda del elemento con más repeticiones. Para esto, se utilizaron los archivos `corpus-0`, `corpus-1`, `corpus-2`, `corpus-3` y `corpus-4` generados por el test 5 provisto por la cátedra. En ambos experimentos se fijó uno de los dos parámetros que determinaba el número de threads en 1 y se varió el otro entre 1 y 10. Se tomaron 100 mediciones en cada caso y se calculó el promedio.

El tiempo de ejecución se determinó mediante la función `clok_gettime` perteneciente a la biblioteca `time.h`. Se tomó el tiempo inmediatamente antes y después de la instrucción que llamaba a la función `maximum`, se calculó la diferencia y se expresó la misma en milisegundos.

Las pruebas se realizaron en un equipo con las siguientes características:

- Sistema Operativo GNU/Linux Ubuntu 16.04
- Procesador Intel Core i5-3550 3,30 GHz
- Motherboard Intel DH67BLB3
- Memoria RAM Kingston KVR1333D3N9/4G (x2)
- HD 500GB SATA 3

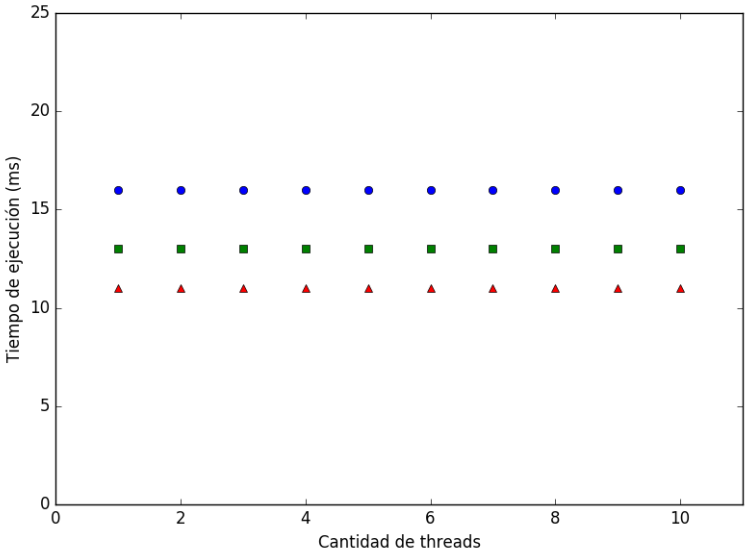

Figura 1: Tiempo de ejecución de los algoritmos maximum (●) , maximum2 (■) y maximum3 (▲) en función de la cantidad de threads utilizados para calcular el elemento con mayor cantidad de apariciones. En todos los casos, el tamaño de `archs` fue 5 y el parámetro `p_archivos` se mantuvo en 1. Se tomaron 50 muestras para cada valor de `p_maximo`.

4. Resultados

Al variar la cantidad de threads utilizados para calcular el elemento con mayor cantidad de apariciones (figura 1) se encontró que `maximum3` finalizaba en un tiempo menor a `maximum2` y que este último se comportaba de la misma manera con respecto a `maximum`.

Para el caso de un sólo thread, y considerando que la función para encontrar elemento más repetido es la misma en los tres casos, la diferencia puede deberse a las diferentes formas de procesar los archivos. `maximum` lo hace de manera análoga a la versión no concurrente de `count_words`, que toma como parámetro solamente el nombre del archivo a procesar. En consecuencia, por cada elemento de la lista que recibe `maximum` se debe realizar el equivalente una llamada a `count_words`, menos la llamada en sí y la copia del HashMap; además, se genera un HashMap por cada thread, por lo que el costo de combinar todas las soluciones parciales puede influenciar el tiempo global.

Por otra parte, los otros dos algoritmos utilizan las versiones concurrentes de `count_words` , evitando así generar un exceso de HashMaps y el costo de combinar las soluciones de cada thread. La diferencia de performance entre `maximum2` y `maximum3` puede explicarse con el hecho de que el primero utiliza siempre tantos threads como archivos posea la lista de entrada, mientras que el segundo varía de acuerdo con el parámetro `n`. En este caso, es posible que el costo de cambiar de thread favorezca al algoritmo que no debe hacerlo.

No obstante estas diferencias iniciales, era razonable suponer que al aumentar el número de threads utilizados para calcular el máximo, dado que los tres algoritmos se ven afectados de igual forma por este, su tiempo de ejecución se vería reducido. Sin embargo, puede verse en la figura 1 que esto no ocurre; aún teniendo más threads que filas para recorrer en el HashMap, los tiempos no se ven modificados. Esto es un fuerte indicador de que, tal vez, el paso que determina la velocidad de ejecución de los tres algoritmos es el procesamiento de archivos.

Con relación a esto último, puede apreciarse en la figura 2 que , aunque inicialmente los tiempos son equivalentes a los obtenidos modificando `p_maximo`, con el primer thread que se agrega tanto `maximum` como `maximum3` ven afectado su tiempo de ejecución. `maximum2` en este caso funciona como control, pues siempre utiliza una cantidad constante de threads. En el caso de `maximum`, el tiempo se reduce debido a que la creación de los HashMaps no recae exclusivamente en un thread sino que, al aumentar su número, es más probable que se reparta equitativamente entre todos (por la forma en la que se implementó, eventualmente se llega a exactamente un thread por archivo); `maximum3`, por otro lado, ve aumentado su tiempo de ejecución. Dado que en ambos casos se llega en última instancia a aproximadamente un thread por archivo, la diferencia puede residir en las contenciónes que pueden producirse al intentar modificar un único HashMap en `maximum3`. Es posible que el hecho de que `maximum` utilize HashMaps temporales para esta tarea influya en la diferencia de tiempos.

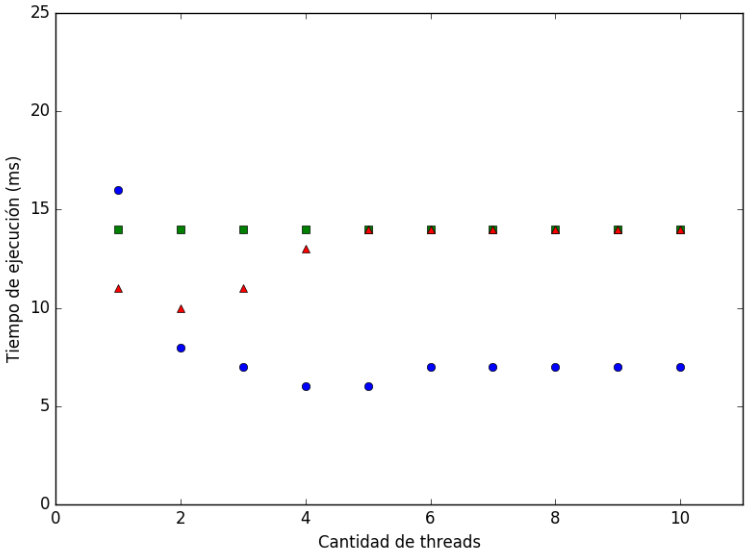

Figura 2: Tiempo de ejecución de los algoritmos maximum (●) , maximum2 (■) y maximum3 (▲) en función de la cantidad de threads utilizados para leer la lista de archivos de entrada. En todos los casos, el tamaño de `archs` fue 5 y el parámetro `p_maximo` se mantuvo en 1. Se tomaron 50 muestras para cada valor de `p_archivos`.

5. Conclusiones

En este trabajo se estudió el uso de pthreads como medio para implementar un modelo de ejecución concurrente y se estudiaron las potenciales ventajas y desventajas que conlleva su uso.

Se concluyó que, si bien existen situaciones en los que la concurrencia es útil (y hasta necesaria), no es cierto que esto sucede en cualquier circunstancia. En el caso estudiado, por ejemplo, la utilización de threads para procesar una lista de archivos resultó en una mejor performance, mientras que su uso para encontrar el elemento más repetido en un arreglo de listas enlazadas no produjo ningún efecto. Por otro lado, el manejo de la concurrencia desde el código no resultó trivial, ya que el número de posibles interacciones indeseadas implica la necesidad de un mayor cuidado a la hora de programar.

Entonces, antes de intentar abordar la resolución de un problema utilizando este medio, en preciso analizar el beneficio potencial que podría traer en cada caso particular y, en última instancia, decidir si este justifica el aumento en la complejidad del software que se requiere para coordinar eficientemente todos los procesos.