DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Multithreading

Sistemas Operativos

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Manuel Panichelli	72/18	panicmanu@gmail.com
Vladimir Pomsztein	364/18	blastervla@gmail.com
Gaston Einan Rosinov	37/18	grosinov@gmail.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

Índice

1. Abstract

Se aborda la problemática de programación de forma concurrente con *POSIX threads*, en particular para una implementación de un *HashMap* que cuenta la cantidad de apariciones de palabras en un conjunto de archivos. Además, se realiza un análisis del impacto en performance que trae aumentar el nivel de concurrencia.

keywords: *POSIX, threads, concurrency, HashMap, List, synchronization, atomic, performance*

2. Introducción

Con la creciente exigencia técnica requerida para mejorar los procesadores modernos, de componentes cada vez más minúsculos, y el cada vez mayor uso de sistemas multiprogramables, el multiprocesamiento resulta una rama de estudio muy importante para la computación moderna. De ella se desprende el concepto de concurrencia, central en este estudio.

La concurrencia es una herramienta muy útil, ya que nos permite realizar tareas de forma más eficiente y veloz al poder distribuir la carga de procesamiento entre varios procesos o threads de un mismo programa; posibilita la paralelización de tareas, resultando en un potencial mejor rendimiento del programa o sistema. Sin embargo, estas ventajas no vienen sin sus propios problemas y dificultades, como lo son la contención entre los recursos, condiciones de carrera e inconsistencias, entre otras.

Un aspecto fundamental a tener en cuenta es que muchas veces los diferentes componentes de nuestro sistema que se ejecutan de manera concurrente deben ser *sincronizadas* de alguna manera, tal que trabajen estando conscientes de sus posibles impactos en otros procesos o threads, evitando así los problemas previamente descriptos.

Existen varias herramientas para lograr dicha sincronización, desde ahora llamadas *primitivas de sincronización*. Semáforos, locks, objetos atómicos, son tan solo algunas de ellas. Se profundizará en estos conceptos más adelante.

Con el fin de lograr ejecutar tareas de forma concurrente, este estudio hará uso de *threading*, en particular la interfaz *pthreads* que forma parte del estándar POSIX[1].

[1] https://es.wikipedia.org/wiki/POSIX

En este trabajo, se implementará un `HashMap` concurrente con *chaining*, haciendo uso de `LinkedLists` atómicas para el manejo de colisiones. Luego, se usará dicho `HashMap` para contar la cantidad de apariciones de palabras en un conjunto de archivos, que también se leerán de manera concurrente. Por lo tanto bastará con que nuestro mapa tenga como claves *strings* y valores enteros no negativos.

3. Detalles de implementación

A continuación, se presenta un análisis de cómo fueron implementadas las estructuras de datos para proveer ciertas garantías para su uso en programas concurrentes.

3.1. Lista atómica

La operación de inserción en nuestra lista es ***atómica***. Repasando la definición de atomicidad, recordaremos que

> Una operación sobre un objeto será *atómica* si un observador no puede distinguir estados intermedios durante el transcurso de su ejecución. Luego, un objeto será *atómico* si todas sus operaciones lo son.

Como podremos notar en Listing 1, la operación en cuestión efectivamente cumple con la atomicidad previamente planteada, puesto que la creación del nodo no modifica la lista. Luego, se modifica dicho nodo (que ya aclaramos no modifica aún la lista hacia afuera), y se repite este paso hasta que se puede reemplazar la cabeza de la lista por el nuevo nodo, de manera atómica. De esta manera, podemos concluir que la operación propuesta es atómica.

```
insertar(valor):
    Nodo n(valor);

    do:
        n.siguiente = cabeza.load();
    while(cabeza.compareAndSwap(n.siguiente, n));
```

Listing 1: Pseudocódigo de insertar

Donde `compareAndSwap` del objeto atomico `cabeza` es el de la definicion usual,

```
atomic T compareAndSwap(T cmp, T swp):
    # reg es el registro atomico
    T res = reg;
    if (cmp == res) reg = swp;
    return res;
```

Listing 2: Pseudocoidgo de Compare And Swap

Gracias a esto, como `insertar` es la única operación que realiza modificaciones sobre la lista, toda la lista es atómica. Pero un punto a tener en cuenta es que a pesar de que lo sea, no quiere decir que cualquier programa que la use quede protegido de ***race conditions***. Citando al *Silberschatz*[2],

> A **race condition** is when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

Luego en un programa como el siguiente,

```
l = ListaAtomica()
thread l.insert("a")
thread l.insert("b")
```

[2]Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts*

```
4 thread l.insert("c")
```

Listing 3: Programa con race conditions

Dependiendo del *orden* en el que se ejecuten los threads, el *resultado*, que en este caso es el orden de la lista, variaría. Produciendo una *race condition*.

3.2. HashMap concurrente

A continuación describiremos la implementación de las operaciones del HashMap, de forma tal que estén libres de *race conditions* y *deadlocks*.

3.2.1. Incrementar

Al incrementar una clave, se le aplica la funcion de *hash* llevandola a un *bucket* del mapa, donde se encuentra una lista. Luego, será necesario buscar el elemento dentro de esa lista e incrementar su valor. Notamos entonces que si no hubiera colisión de hash, no podría haber condiciones de carrera pues las listas serían distintas. Cuando no hay colisión de hash, las claves pueden ser iguales o distintas.

- Cuando son distintas, en ningún caso habría race conditions,
  - Si ambas claves existen, se modificarán valores de nodos distintos.
  - Si ambas claves son inexistentes, se insertarán de manera atómica.
  - Si una clave es inexistente y la otra existente, se insertará de manera atómica y modificará el valor de un nodo ya existente. El órden en el que se ejecuten estas operaciones es irrelevante.
- Cuando son iguales,
  - Si la clave no existe, podría ocurrir que se la inserte dos veces.
  - Si la clave existe, podría ocurrir que se pierda uno de los dos incrementos a la misma.

Para evitar estas *race conditions*, bastaría con garantizar la *exclusión mutua* de las listas, es decir, que no haya más de un thread modificando la misma lista a la vez. Esto se puede lograr a través de una primitiva de sincronización conocida como ***mutex***. De esta manera, aún permitimos el acceso concurrente a todos buckets, por lo que no agregamos demasiadas restricciones y mantenemos la natrualeza concurrente del `HashMap`.

3.2.2. Claves y valor

Para estas funciones no resulta necesario el uso de ninguna *primitiva de sincronización*, pues no se pueden generar race conditions en las lecturas (gracias a que `insertar` es atómico). Por lo tanto, las implementaciones resultan triviales.

3.2.3. Búsqueda del máximo

Si máximo no se sincroniza de ninguna manera con la inserción, podrían darse condiciones de carrera. Para ilustrarlo, analizaremos algunos problemas que podrían surgir considerando la ejecución concurrente de la función `insertar` y una implementación trivial de la función `maximo`, que se encarga de calcular cuál es la palabra con mayor cantidad de apariciones en todo el `HashMap`.

Imaginemos el siguiente código de un potencial usuario de nuestro `HashMap` atómico (en pseudocódigo), donde la llamada `in_thread` corre lo especificado en un nuevo thread.

```
# Thread principal
map = HashMapConcurrente()

# Thread A
in_thread(func (){
  for 1 to 50:
    map.insertar("ardilla")

  for 1 to 21:
    map.insertar("zorro")
})

# Thread B
in_thread(map.maximo)
```

Listing 4: Programa naive

Todos los **"ardilla"** irían a parar a la misma lista, la correspondiente al slot 0 del `HashMap`, y los **"zorro"** al último. Entonces para algún scheduling posible, podría suceder que el thread A, que hace las inserciones, ejecute hasta insertar 20 veces "ardilla", y luego sea desalojado por el scheduler en favor de B. Luego, este podría llegar a recorrer toda la lista de la letra A, y tomar a ardilla como el máximo actual con valor 20. El mismo thread recorrería las listas siguientes sin contar la última. Después le tocaría nuevamente a A que hace el resto de las inserciones de "ardilla" y las inserciones de "zorro", y finaliza su ejecición.

Cuando se ejecute B hasta finalizar, verá que "zorro" tiene 21, tomándolo como máximo.

Pero, para cuando máximo terminó su ejecución, notamos que devolvió un valor que nunca fue el máximo, "zorro", mientras que "ardilla" debería haber sido el resultado correcto.

```
# Thread A     |      # Thread B (maximo)
1  ardilla     |
2  ardilla     |
...            |
20 ardilla     |
               |      maximo
               |      # Toma a ardilla como max con valor 20
               |      ...
...            |
50 ardilla     |
1  zorro       |
2  zorro       |
...            |
21 zorro       |
# FIN          |
               |      maximo
               |      # Toma a zorro como max con valor 21
               |      ...
               |      # FIN
               |      # Resultado final: zorro con valor 21
```

Listing 5: Secuencia de inserciones

3.2.3.1 Paralelización de máximo

Para la implementación de `maximoParalelo`, una función que dada una cantidad de threads procesa en paralelo el máximo del `HashMap` con la cantidad de threads indicada, pensamos en hacer que un hilo

principal lance los threads, y que cada uno sea autónomo. Es decir, que cada uno buscará el máximo en una lista (bucket) que no haya sido revisada todavía, y que cambiará el máximo actual en caso de que el que encontró sea mayor.

Para ello, hacemos uso de los siguientes recursos compartidos:

- **HashMapConcurrente *hashMap**: Un detalle de la implementación, es que como **pthreads** no soporta correr en un thread un método de un objeto, tuvimos que implementar un *workaround* con una función wrapper en el medio que tome este objeto como parámetro.
- **std::atomic<int> *nextList**: La siguiente lista a recorrer para buscar el máximo. Basta con un `atomic` para protegerlo, ya que solamente es necesario hacer **getAndInc** (**fetch_add** en **std::atomic**).
- **std::mutex *maxMux y hashMapPair *currentMax**: El maximo actual global. Como era necesario que fuera de acceso concurrente, y la operación a realizar no podía resolverse con un `atomic` (comparar por mayor y en función del resultado decidir si reemplazar o no), recurrimos al uso de un simple `mutex` para garantizar la exclusión mutua al modificarlo.

Como optimización, en realidad cada thread no chequea si su máximo es mayor al actual después de ver cada lista, sino que lo hace al final, cuando ya no tiene más listas por recorrer. De esta forma, se evitan locks innecesarios sobre el `mutex`.

3.3. Carga de archivos

Para la implementación de **cargarArchivo** no resultó necesario tomar ningún tipo de recaudo nuevo, pues se utiliza sólamente **incrementar** que como vimos en Listing 1 fue implementado de forma *thread safe*.

La implementación de `cargarMultiplesArchivos` sigue los mismos principios que `maximoParalelo`, aunque con menos recursos compartidos entre los threads. La lista de archivos no necesita de ningún mecanismo de protección ya que es solo de lectura.

4. Análisis de ventajas y puntos débiles de la ejecución concurrente

Con la finalidad de obtener un mayor entendimiento del impacto de la concurrencia sobre el rendimiento de programa, propondremos una sencilla pero interesante línea de experimentación.

4.1. Propuesta de análisis

Ejecutaremos nuestro programa, variando la cantidad de threads que se utilizarán para la resolución del problema de carga de datos y de búsqueda del máximo utilizando el `HashMap`.

Para ello, y con el fin de amortiguar el impacto del ruido producido por el Sistema Operativo sobre los resultados, ejecutaremos el programa para cada cantidad de threads 25 veces, y luego tomaremos el promedio de los resultados obtenidos. El programa se ejecutará para entre 1 y 30 threads.

Más aún, con el objetivo de obtener datos de calidad, propondremos la utilización de una lista de 10000 palabras, compuesta por 2280 palabras distintas, y haremos que el programa cargue dicho archivo 30 veces. El motivo para la utilización de dicha cantidad de repeticiones en la carga, de un archivo de tales características se remonta a poder garantizar que cada uno de los 30 threads podrá tomar la carga de un archivo, sumado a que nos gustaría que el tiempo que cada proceso pasa cargando el archivo sea significante como para poder analizarlo.

En particular, con este experimento esperamos ver que el rendimiento mejore inicialmente, para cantidades de threads mayores a 1, pero que al acercarse a la cantidad de núcleos lógicos de la computadora sobre la que corre el programa comience a empeorar, pudiendo ser incluso peor que con un único thread. El motivo de este razonamiento es que teniendo más threads que unidades de procesamiento (aunque lógicas), los threads no podrían ejecutarse realmente en simultáneo, ya que como solo se ejecuta un thread por cada unidad, el resto queda en estado *"ready"* (sin ejecutar) hasta que se libere alguna.

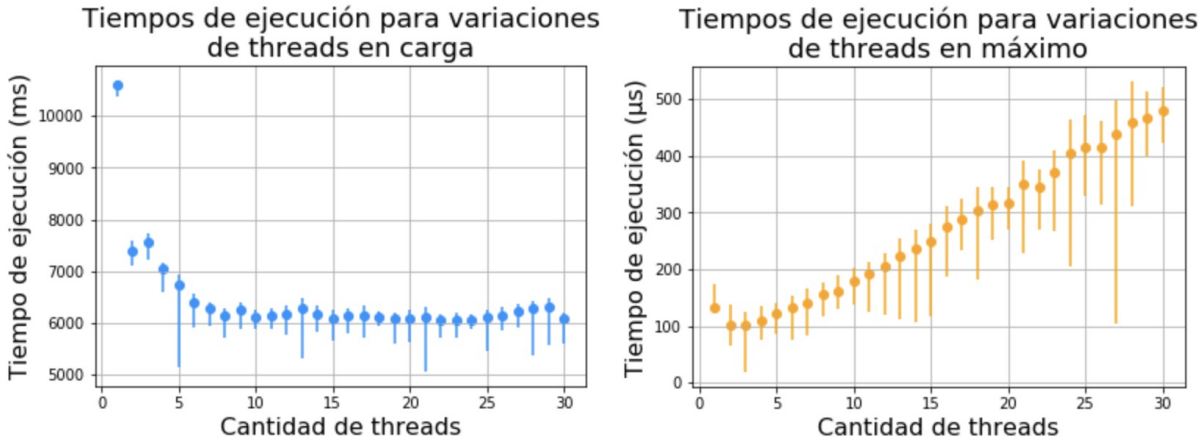
Notamos que no esperamos que el punto de corte sea, necesariamente, para la cantidad de threads igual a la de los núcleos lógicos de la computadora, si no un poco antes. Esto es porque suponemos que

al estar corriendo sobre un Sistema Operativo multiproceso, sería muy ambicioso pensar que nuestro programa podrá tener efectivamente todos los núcleos de la computadora asignados para correrlo. De esta manera, en una computadora de 4 núcleos lógicos como la que se utilizará para la experimentación, esperamos que el punto de inflexión en los resultados se dé para 3 threads.

4.2. Resultados y discusión

A continuación, se presentan los resultados de la experimentación para la variación de threads de ejecución para la carga de archivos y la búsqueda del máximo bajo las condiciones presentadas previamente. El programa fue ejecutado en una *MacBook Pro (13-inch, 2017, Procesador 2.5 GHz Intel Core i7 de 2 núcleos, con 4 núcleos lógicos en total y Memoria de 16 GB 2133 MHz LPDDR3)*.

Notar la diferencia de escala en ambos gráficos, puesto que los resultados se presentan en órdenes de magnitud completamente distintos.



(a) Rendimiento según cantidad de threads en la carga de archivos (en ms)

(b) Rendimiento según cantidad de threads en la búsqueda del máximo (en μs)

Por un lado, podemos observar que los resultados planteados para la búsqueda del máximo se condicen perfectamente con las hipótesis planteadas en 4.1.

Sin embargo, para el análisis de rendimiento empleado para la carga de archivos, notamos que la hipótesis planteada difiere ampliamente de los resultados obtenidos. Esto podría explicarse puesto que las operaciones predominantes en la carga de archivos resultan las de *Entrada/Salida*. Por lo tanto, a diferencia de los utilizados para la búsqueda del máximo, estos threads se bloquearán esperando los resultados de sus peticiones a memoria, y esto hace posible que los threads dejen de "sabotearse" los unos a los otros, puesto que se bloquearían y así cederían su tiempo de procesamiento a otros threads que ya cuenten con los resultados de sus llamados a memoria.

De esta forma, podemos entender que el rendimiento de la carga de archivos mejore para cantidades de threads superiores a la cantidad de núcleos lógicos disponibles. Más aún, notamos que en los resultados no es posible apreciar una desmejora a partir de cierto punto, si no que luego de los 10 threads la performance parecería ser más bien constante. Esto es nuevamente razonable, pues podemos suponer que el factor de mejora dadas las cuestiones de *Entrada/Salida* producidas por el aumento de threads se encuentra en un orden equivalente al factor negativo generado por el "sabotaje" de los threads entre sí, produciendo una constante.

4.2.1. Trabajos futuros

De la investigación y los resultados propuestos se desprenden algunas dudas y cuestiones que sería interesante profundizar a futuro. Por ejemplo, analizar el comportamiento de los experimentos planteados bajo distintas condiciones de ejecución, en sistemas con mayor y menor cantidad de núcleos lógicos que los utilizados.

Por otro lado, también sería interesante observar los resultados más allá de los 30 threads, y además analizar la conducta del programa en la carga para distinta cantidad de archivos, y en la carga y la búsqueda del máximo para archivos de distintas características y tamaños.

Finalmente, para ayudar a entender los resultados de aumentar la concurrencia para la carga de archivos, podría ser útil realizar un *profiling* más minucisoso, para así poder saber de forma más precisa cuánto tiempo pasaron los threads en *I/O wait*.

5. Conclusiones

En base a la experimentación propuesta y los resultados obtenidos, se puede concluir que el multithreading resulta una notable y poderosa herramienta, a tener en cuenta a futuro con el fin de mejorar el rendimiento de programas. Resulta destacable la gran mejoría vista en los resultados, producto del incremento del grado de threads utilizados. Sin embargo, es importante utilizarla a discreción, pues no siempre aumentar la cantidad mejorará la performance y hasta podría pasar que esta empeore como se puede apreciar en el caso de la búsqueda de máximos, en la figura 1b.

Además, notamos que las cuestiones de *Entrada/Salida* suponen un factor importante a tener en cuenta al determinar la cantidad de threads a utilizar, junto con la cantidad de núcleos lógicos disponibles en el procesador sobre el que el programa correrá.

Finalmente, destacaremos que la concurrencia no deja de traer sus propios problemas, a pesar de sus grandes mejoras en rendimiento. Sin embargo, estos pueden ser evitados mediante el uso de las *primitivas de sincronización* adecuadas para evitar las condiciones de carrera sin afectar demasiado la contención de recursos, con el fin de preservar el rendimiento del programa.