Trabajo Práctico I - Sistemas Operativos 1c 2020: *Threading*

Dylan Socolobsky[1] y Martín Javier del Río Llobera[2,3]

***Introducción*— En este trabajo práctico buscaremos profundizar en una de las facetas principales que aparece al estudiar los sistemas operativos: la gestión de la concurrencia. Practicaremos cómo razonar sobre la ejecución concurrente de programas y las técnicas para gestionar la contención sobre los recursos y evitar que se produzcan condiciones de carrera.**

Nos centraremos especialmente en el uso de *threads*, una herramienta provista por los sistemas operativos que nos permite disponer de varios hilos de ejecución concurrentes dentro de un mismo programa. En particular, emplearemos la interfaz *threads* `pthreads` que forma parte del estándar POSIX.

Realizaremos la implementación de una estructura de datos que será denominada `HashMapConcurrente`. Se trata de una tabla de *hash* abierta, que gestiona las colisiones usando listas enlazadas. Su interfaz de uso es la de un *map* o diccionario, cuyas claves serán *strings* y sus valores, enteros no negativos. La idea es poder aplicar esta estructura para procesar archivos de texto contabilizando la cantidad de apariciones de palabras (las claves serán las palabras y los valores, su cantidad de apariciones).

I. SOBRE TIPOS ATÓMICOS Y CONDICIONES DE CARRERA

Un tipo *atómico* de datos es un tipo de datos que encapsula un valor a modo de garantizar que su acceso no cause condiciones de carrera entre múltiples threads. Es decir, las operaciones realizadas sobre estos tipos quedan en un orden tal que las operaciones que se hagan sobre el tipo sean visibles antes de que se apliquen nuevas opraciones sobre el tipo; no hay condición de carrera entre las operaciones internas de distintos llamados a sus métodos.

Los tipos atómicos son especialmente importantes para los procesos que ejecuten múltiples threads, ya que si múltiples threads operan simultaneamente sobre el mismo objeto atómico, las operaciones se aplicarán una tras otra de modo que cuando una segunda operación quiera aplicarse, los resultados de la primera ya se habrán hecho visibles en el objeto.

En nuestro caso, que nuestra lista sea *atómica* significa que sus operaciones de clase son atómicas, en particular `insertar`, `cabeza`, `iesimo` y `longitud`. Estos métodos fueron diseñados de modo que si diferentes threads quisieran ejecutarlos al mismo tiempo, se ejecutarían en un orden arbitrario, uno tras otro, sin incurrir en condiciones de carrera.

Esto no significa bajo ningún concepto que trabajar con tipos atómicos de datos elimine cualquier posiblidad de incurrir en condiciones de carrera. No solo podrían existir otras secciones dentro del programa que puedan acabar en una condición de carrera (como por ejemplo que se de un de contexto entre la guarda de un `if` y su bloque), sino que entre operaciones atómicas podrían darse situaciones que acaben en condiciones de carrera. Por dar un ejemplo, el incremento de dos valores atómicos y su posterior suma en una tercer variable atómica podría incurrir en una condición de carrera a pesar de que se cuente con atomicidad en todos los tipos sobre los que estemos operando, como explicita el pseudocódigo de **Algorithm 1**.

Algorithm 1 Condición de carrera sobre suma atómica post-incremento a través de múltiples *threads*.

Sean α_1, α_2 y β tres enteros atómicos.

Thread 1:
incrementarAtomicamente(α_1)

Thread 2:
incrementarAtomicamente(α_2)

Thread 3:
$\beta = \alpha_1 + \alpha_2$
// Se asume que esta última operacion se realiza
// atomicamente.

Puede observarse a simple vista que si el orden de las ejecuciones no se realizara de arriba hacia abajo tal como figuran en el algoritmo, el valor resultante en β sería distinto a lo esperado. Si los tipos atómicos nos previenen de condiciones de carrera sobre operaciones singulares sobre el mismo tipo atómico, a la hora de tratar sobre múltiples elementos es necesario controlar el flujo del algoritmo mediante herramientas como *semáforos* o *mutex*.

Para cumplir la propiedad de atomicidad, nuestra implementación del método `insertar` de la lista atómica se vale del método `compare_exchange_weak`. El procedimiento que sigue consta de crear un nuevo nodo con el valor que queremos insertar y luego asignar la cabeza actual de la lista como nodo siguiente al nodo que queremos insertar. Luego usamos `compare_exchange_weak` para comparar el nodo siguiente al que queremos insertar (que, en teoría, sería la cabeza actual de la lista) con la cabeza actual de la lista. Si efectivamente la cabeza actual de la lista es el nodo siguiente al nodo que queremos insertar, `compare_exchange_weak` asigna el nodo a insertar como cabeza de la lista. De lo contrario,

[1]LU: 501/18 - Email: dsocolobsky@gmail.com
[2]LU: 401/15 - Email: lartu@lartu.net
[3]El tercer integrante de nuestro grupo decidió abandonar la materia prematuramente y no nos acompaña en la entrega de este informe.

asigna la cabeza actual de la lista como siguiente al nodo que queremos insertar y reintenta el procedimiento hasta poder insertarlo satisfactoriamente. Como el método `compare_exchange_weak` es atómico, la comparación y el intercambio que ofrece se realizan de forma atómica, eliminando cualquier posibilidad de que suceda una condición de carrera entre la inserción de dos nodos (y, por lo tanto, se pierda alguno).

II. SOBRE CÓMO EVITAR CONDICIONES DE CARRERA SIN CONTENCIÓN EXCESIVA

A la hora de completar las implementaciones de los métodos `incrementar(string clave)`, `vector<string>claves()` y `unsigned int valor(string clave)`, nos fue pedido que estuvieran libres de condiciones de carrera y *deadlock*. El último método requería también ser no bloqueante y libre de espera (`vector<string>claves()` también lo requería inicialmente pero nos fue aclarado que no tuviéramos en cuenta este requisito).

La implementación del método `incrementar(string clave)` tenía como condición extra que que solo hubiera contención en caso de colisión de *hash*; es decir, si dos o más hilos intentaban incrementar concurrentemente claves que no colisionaban, debían poder hacerlo sin inconvenientes. Para esto definimos un mutex por cada posible *bucket* (uno por letra del alfabeto), como se describe en **Algorithm 2**. Al intentar incrementar una clave, se entra en un ciclo infinito. Luego se hashea su valor y se intenta adquirir el *lock* de su *bucket*. Si se logra esto y la clave existe, se incrementa el valor de la tupla. En caso contrario, el proceso se queda esperando para incrementar. Si la clave no existe, se captura el *mutex* y se crea la clave. Si dos procesos intentan crear una clave inexistente, la única condición de carrera será quién capture el *mutex* primero. Quien lo haga creará la clave y saldrá de la función. El thread que no, vuelve al ciclo y, una vez que el *mutex* se libere, lo tomará e incrementará la clave. Como cada *mutex* depende del *hash* a usar, solo hay contención en caso de colisión de *hash*.

La implementación del método `vector<string>claves()` tiene como dificultad que si tomo un acercamiento inocente al problema e intento iterar desde el primer *bucket* al último (de la A a la Z) y se inserta una clave en un bucket por el que ya pasé (supongamos que voy por H y se inserta la clave $barco$ en el *bucket* B) y luego una clave en un *bucket* por el que todavía no pasé (siguiendo el ejemplo anterior se inserta la clave $polilla$ en el *bucket* P) la clave $barco$ no será devuelta pero $polilla$ sí, lo que se resultariaá será una representación de un estado del *hashMap* que nunca existió realmente. Decidimos entonces que lo más razonable para asegurarnos de siempre devolver un estado consistente y real del *hashMap* es bloquear todos los *mutex* de cada *bucket* del objeto y luego proceder a devolver todas sus claves en orden, ahora sí tomando el *approach* más *naïve*.

Algorithm 2 Pseudocódigo de la implementación de $incrementar(stringclave)$.

```
hash = hash(clave)
while true do
  tupla = buscar(tupla)
  if tupla then
    while true do
      if trylock(mutex_bucket(hash)) then
        tupla_2 = tupla_2 + 1
        unlock(mutex_bucket(hash))
        return
      end if
    end while
  else
    if trylock(mutex_bucket(hash)) then
      insertar(clave, lista)
      unlock(mutex_bucket(hash))
      return
    end if
  end if
end while
```

Si bien trivial, el algoritmo de este método puede verse en **Algorithm 3**.

Algorithm 3 Pseudocódigo de la implementación de $vector < string > claves()$.

```
listaClaves = []
bloquearTodosLosMutex()
for all bucket in hashMap do
  for all clave in bucket do
    listaClaves = listaClaves join clave
  end for
end for
desbloquearTodosLosMutex()
return listaClaves
```

Finalmente, el método `unsigned int valor(string clave)` requiere además de estar libre de condiciones de carrera, ser no bloqueante y *wait-free*. Para eso implementamos el método auxiliar `buscar(string clave)`, que se encarga de traer el par con la clave buscada, en caso de que exista. La implementación de `valor` entonces únicamente consistirá en llamar a `buscar` y devolver el valor retornado o un puntero nulo en caso de no haber encontrado la clave.

La implementación de `buscar(string clave)` puede verse en **Algorithm 4**. Si bien la función no asegura que no se pueda insertar una clave o incrementarla mientras se está buscando, al consultar nos fue dicho que con que devolviéramos un resultado válido para alguna configuración que haya existido del *hashMap* (aunque relativamente reciente), el resultado sería valido.

Algorithm 4 Pseudocódigo de la implementación de $buscar(stringclave)$.

```
lista = bucket(hash(clave))
for i from 0 to longitud(lista) do
  tupla = iesimo(lista, i) // Método atómico
  if first(tupla) == clave then
    return tupla
  end if
end for
return null
```

III. SOBRE PROBLEMAS DE CONCURRENCIA Y REPARTO DE TRABAJO ENTRE *THREADS*

En el código provisto por la cátedra se brinda una implementación de la función `maximo()`, que devuelve la clave que mayor valor tenga en el *HashMap*.La implementación de `incrementar` se encuentra descrita en el apartado anterior.

Si ambos métodos fueran a ejecutarse en paralelo, se podría incurrir en una condición de carrera en el que se devuelva un elemento que nunca fue el máximo del *HashMap*. Dado que la implementación provista de `maximo()` itera por todos los *buckets* en orden alfabético, podría ocurrir el siguiente escenario: el *HashMap* comienza vacío. `maximo()` empieza a iterar y llega a la B, cuando ocurre un cambio de contexto. Mientras `maximo()` no está ejecutando, se agrega la clave $auto$ al *HashMap*. Luego ocurre nuevamente un cambio de contexto y `maximo()` sigue ejecutando, llegando a la C. El contexto vuelve a cambiar y se agrega la clave $auto$ otra vez. `maximo()` corre de nuevo y llega al *bucket* D. Cambio de contexto y se agrega $polilla$ al *HashMap*. Luego el no se agregan más claves y `maximo()` sigue ejecutando hasta llegar a la P, donde encuentra $polilla$ con $valor = 1$ y lo identifica, erroneamente, como el máximo de la lista.

Para solucionar este problema y liberar al método de condiciones de carrera decidimos bloquear todos los *mutex* del *HashMap* antes de realizar la búsqueda. De esta forma, mientras se está buscando el máximo no se permiten inserciones.

Paralelamente, hemos completado la implementación de `maximoParalelo`. Esta nueva iteración del método bloquea todos los *mutex* y declara un entero atómico que comienza en $cantLetras$ de *HashMapConcurrente*. Un ciclo crea tantos *threads* como le fue pedido a la función, y cada *thread* recibe un puntero a este atómico, a un *mutex* para poder acceder al máximo encontrado hasta el momento (*global* y un puntero a dicho máximo. Estos tres elementos son compartidos por todos los *threads*.

Cada *thread* luego intenta obtener una letra decrementando el atómico. Si el atómico es mayor a 0, todavía quedan *buckets* por revisar, sino retorna. Una vez que el thread ya tiene su *bucket*, procede a buscar el máximo de forma secuencial. Una vez que tiene el máximo *local*, intenta bloquear el *mutex* que permite acceder al máximo global. Cuando lo consigue, compara su máximo local con el global y, de ser el máximo, lo reemplaza. Luego desbloquea el *mutex*. Nos pareció que esta opción era elegante y sencilla, pero existe la posibilidad de usar un `compare_and_swap` alternativamente para no tener que bloquear un *mutex*.

IV. SOBRE LAS IMPLEMENTACIONES DE CARGA DE ARCHIVOS

En el trabajo se nos pide paralelizar el proceso de cargar archivos y parsearlos a un formato con el cual podamos trabajar.

Para esto no decidimos agregar ningun metodo de sincronización a la función `cargarArchivo`, la cual permanece en su estado original y totalmente abstraida del concepto de multithreading, la sincronización la realizamos por fuera gracias a las funciones `cargarMultiplesArchivos` y `cargarArchivoDeUnThread`. A un thread ejecutando `cargarArchivo` no deberia importarle la ejecucion del resto de threads para nada, ya que no tiene que interactuar con estos, es por esto que no es necesaria la sincronizacion dentro de la funcion.

En `cargarMultiplesArchivos` lo que hacemos es crear la cantidad de threads especificada por el parametro `cantThreads`; estos hilos a su vez van a ejecutar cada uno la función auxiliar `cargarArchivoDeUnThread`.

Algorithm 5 Pseudocódigo de la implementación de $cargarMultiplesArchivos$

```
for i from 0 to cantThreads do
  threads[i] =
   crearThread(cargarArchivosDeUnThread())
end for
for all thread in threads do
  thread.join()
end for
```

En esta función, se mantiene un pool de archivos que cada thread va a ir a consultar, para conseguir un archivo para cargar. Este pool es simplemente un numero entero atómico, que se va a ir decrementando a medida que cada thread carga ese archivo, con este entero indexamos en la lista de archivos para cargar y cargamos dicho archivo con la ya mencionada `cargarArchivo`.

Algorithm 6 Pseudocódigo de la implementación de $cargarArchivoUnThread$

```
while pool.load() >= 0 do
  idx = pool.fetch_sub(1)
  cargarArchivo(files[idx]))
end while
```

Finalmente de vuelta en `cargarMultiplesArchivos` esperamos a que todos los threads terminen haciendo join.

V. Sobre las experimentaciones y los resultados obtenidos

Con el fin de comprobar que nuestra implementación de los ejercicios sea correcta, y de evaluar las ventajas en terminos de performance obtenidos gracias a la utilización de threads, llevamos a cabo dos experimentaciones:

- **Busqueda de maximo:** Analizamos que ocurre a medida que aumentamos la cantidad de threads utilizados para encontrar el maximo valor en la tabla.
- **Carga de archivos:** Dado que el programa permite cargar mas de un archivo de test, analizamos que ocurre en terminos de tiempo cuando cargamos estos archivos de a diversos threads.

Los experimentos fueron realizados por separado, y en ambos casos medimos solamente la función que nos interesaba, maximoParalelo y cargarMultiplesArchivos respectivamente.

Las mediciones fueron hechas con la ayuda de std::chrono, con el binario compilado con -O3 y con un procesador Intel i7-6600U con 6 nucleos a 2.60GHz y 16GB de RAM, bajo el sistema operativo Linux. Con el fin de minimizar el ruido en las mediciones causado por baja prioridad en el proceso, desalojo en el procesador del proceso, etc. por cada test corrimos 20 samples, y nos quedamos con el promedio.

En ambos casos la hipotesis fue la misma, que el agregar mas threads, intuitivamente, el trabajo iba a repartirse entre ellos y por lo tanto el tiempo total de procesamiento iba a ser menor. Tambien teorizamos que iba a llegar un momento donde el agregar mas threads no iba a brindar mas beneficio, e incluso podría conllevar mas tiempo (debido al overhead de crear y sincronizar threads).

Por ende nuestra hipotesis era que la curva de rendimiento del tiempo transcurrido en funcion de los threads iba a resultar simil a una cuadratica negativa, llegando a un punto de inflexion donde empezariamos a ver un incremento del tiempo.

A continuacion detallamos los dos experimentos y analizamos los resultados obtenidos.

*1) **Busqueda de maximo**:* En este experimento modificamos el parametro cantThreads de la funcion HashmapConcurrente::maximoParalelo. Esta funcion busca el maximo en la tabla de hash, creando y utilizando la cantidad de threads especificada por cantThreads. Decidimos que para obtener un paneo mas general de la utilidad de los threads, ibamos a medir la busqueda del maximo con 100 entradas en la tabla, 500, 1000, 5000, 10000 y finalmente 25000; y promediar cada tamaño. Utilizamos hasta un máximo de 30 threads (iterando asi de 1 a 30 threads, en 5 escenarios diferentes con 4 pasadas cada uno, logrando 30*5*4=600 tests en total). Los resultados fueron los siguientes:

Sorpresivamente, no nos encontramos ante una curva convexa como teorizamos, si no a algo que mas bien se asemeja a una funcion lineal, indicando que el tiempo transcurrido es linealmente dependiente de la cantidad de threads utilizados, y no al inversamente como se esperaría.

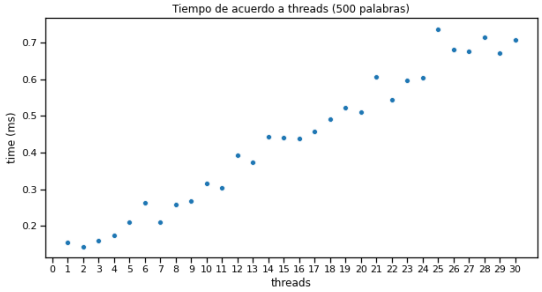

Fig. 1

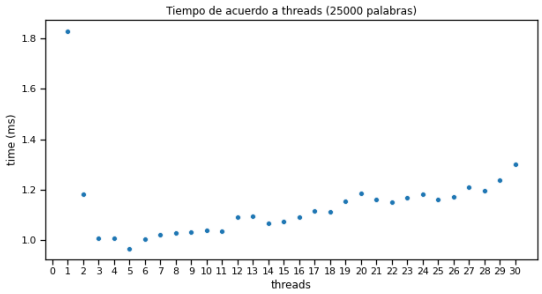

Fig. 2

Esto indica que utilizar mas threads no es beneficioso en este caso. Se ve una mejora de rendimiento solo en los primeros 3 o 4 casos, para despues empezar a empeorar.

De todas formas, a medida que el tamaño de los threads aumenta, la linea parece aplanarse, lo que indica que si el tamaño de la tabla fuese lo suficientemente grande, quiza si se empezaría a ver alguna mejora en el rendimiento.

Nuestra interpretación de los datos obtenidos es que la mejora obtenida por paralelizar el trabajo no amerita el overhead proveniente de crear los threads y el proceso de sincronización (mutexes principalmente) que aumenta con cada thread. Es necesario un problema mas grande para que esto ocurra.

*2) **Carga de archivos**:* A continuacion buscamos analizar la performance de la funcion cargarMultiplesArchivos modificando el parametro cantThreadsLectura que similarmente al caso anterior, varia la cantidad de threads creados para cargar archivos, leerlos y pasarlos al formato con el cual trabajamos en el trabajo práctico.

Para la prueba, realizamos la carga de 100 archivos, de 10000 palabras cada uno, variando de 1 a 100 threads para observar como mejoraba la performance. También realizamos un checkeo preliminar de 20 archivos con hasta 20 threads, que tambien adjuntamos por completitud. Los resultados que obtuvimos son los que siguen:

Como se puede observar, si bien esta vez no vemos la correlacion linear anteriormente vista, seguimos sin ver una clara mejora de la performance a medida que aumentamos los threads, esto solo ocurre para los primeros valores hasta los 8 threads aproximadamente. Con una significativa mejora de 1 a 2 threads.

Por lo tanto podríamos afirmar tras los dos experimentos,

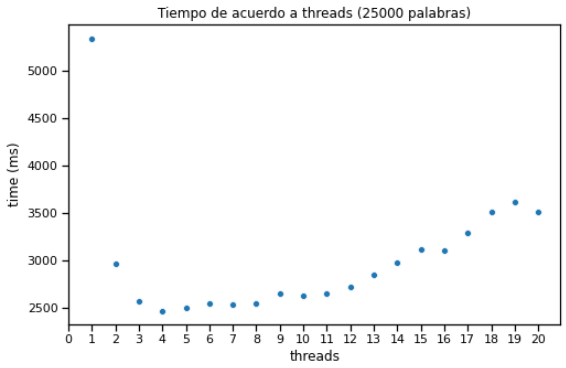

Fig. 3

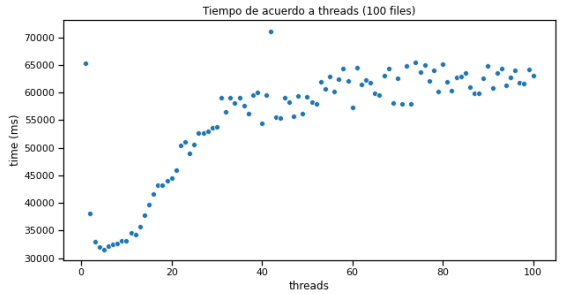

Fig. 4

que las mejoras vistas en performance al agregar mas threads al procesamiento pueden hacer una gran diferencia, y deben tenerse en cuenta, pero debemos estudiar la cantidad de threads a agregar porque tras un cierto punto, esto es inutil o incluso perjudicial, y el umbral suele ser relativamente bajo.

VI. Sobre cómo el lector puede replicar los experimentos realizados

Para poder replicar las mediciones realizadas en este informe se adjunta el archivo `README_mediciones.md`, con una breve serie de pasos a seguir.

Se adjunta también junto a este informe los ficheros `correr_tests.sh`, `correr_test_files.sh` y `correr_test_files_25k.sh`. El primero diseñado para ejecutar mediciones relativas a la la búsqueda de un máximo mediante múltiples *threads*, y los otros dos para cargar múltiples archivos en simultaneo.

Estos ficheros no son más que un llamado a los *scripts* `meditions.py` y `medition_files.py`. Sabemos que la palabra *meditions* no existe, pero decidimos dejarlo así para no vernos forzados a modificar los `sh` presentados en el párrafo anterior.

El compilado de C++ base generado por el *Makefile* provisto por la cátedra fue modificado para que su formato de salida sea siempre $resultado\_maximo, tiempo\_maximo, tiempo\_files$, separado siempre por comas, donde el primer valor corresponde al máximo encontrado, el segundo al tiempo dedicado a la búsqueda dicho valor y el tercero el invertido en la carga de los archivos a procesar.

Estos *scripts* basan sus pruebas en un nmero de archivos de *test*, que pueden encontrarse en las carpetas `test_cases*`. En cada carpeta puede encontrarse también un *scripts* `.php` dedicado a generar dichos ficheros. Las carpetas `mediciones_*` contienen los resultados de ejecución de los programas de testeo sobre estos casos de análisis. En algunos de estos casos es necesario ejecutar el archivo `promedios.py` (si está disponible) para generar archivos promedio de múltiples mediciones para eliminar ruido.

Para graficar los resultados obtenidos es necesario copiar los archivos `.csv` generados por cada prueba a la carpeta `jupyter` y ejecutar el entorno de *Jupyter Notebook* disponible en dicho directorio.

Para compilar nuestra solución implementativa a la consigna no hace falta más que correr `make`. El archivo generado se encontrará en el directorio `build` bajo el nombre de `ContarPalabras`, tal como fuera provisto inicialmente por la cátedra.