Trabajo Práctico 1: *pthreads*

Sistemas Operativos - Segundo cuatrimestre de 2018

Fecha límite de entrega: Miércoles 3 de octubre de 2018 a las 23:59

1. Introducción

Uno de los roles más importantes que cumple un Sistema Operativo es el de permitir a los usuarios hacer un uso correcto del hardware disponible. Usualmente, el hardware es escaso en comparación con la cantidad de procesos que buscan hacer uso de él. Es decir, puede haber más de un proceso queriendo acceder a un mismo recurso, ya sea de manera *concurrente* o *paralela*.

Por esta razón, y para evitar *condiciones de carrera*, los desarrolladores de un sistema operativo deben asegurar la *contención* de estos recursos. Dicho de otra forma, deben asegurar el uso ordenado y controlado de los recursos. Además, es importante como desarrollador adquirir las destrezas necesarias para realizar programas que tengan componentes concurrentes o paralelas.

Luego, este trabajo práctico se orienta a consolidar las ideas mencionadas a través del manejo de *threads*. Los *threads* son una herramienta que proveen los Sistemas Operativos para realizar ejecuciones concurrentes de un *mismo programa*. Dado que los *threads* de un proceso pueden compartir diferentes estructuras y recursos entre si, es necesario utilizarlos de manera correcta para evitar condiciones de carrera.

Para la implementación de *threads* concurrentes, se propone el uso de una estructura de datos que se denominará `ConcurrentHashMap`. Esta estructura es una tabla de *hash* abierta que gestiona las colisiones usando listas enlazadas. Su interfaz de uso es la de un `Map`, es decir, un diccionario. Tendrá únicamente claves que sean *strings* y valores que sean *enteros*. A efectos prácticos, utilizaremos esta estructura para contabilizar la cantidad de apariciones de cada palabra en archivos. Es decir, las claves seran las palabras y los valores su cantidad de apariciones.

Se deberá implementar y utilizar una `ConcurrentHashMap` en `C++` que cumpla con la siguiente interfaz y condiciones:

- `ConcurrentHashMap()`: Constructor. Crea la tabla. La misma debe tener 26 entradas (una por cada letra del abecedario sin contar la "ñ"[1]). Para el manejo de las colisiones, cada entrada consta de una lista de pares $(string, entero)$. La función de hash corresponde a la selección de la primera letra del *string*.

  Los *strings* utilizados como clave solo constarán de los caracteres en el rango `a-z` (es decir, no habrá *strings* con mayúsculas, números o signos de puntuación).

- `void addAndInc(string key)`: Si *key* existe en la tabla, inspeccionando la lista de la entrada correspondiente, se debe incrementar su valor en uno. Si no existe, se debe crear el par $(key, 1)$ en la lista. Se debe garantizar que sólo haya contención en caso de colisión de *hash*. Esto es, si concurrentemente dos o más threads requieren acceder a la misma entrada de la tabla, se deberá utilizar un *lock* para resolver la situación.

- `list<string> keys()`: Devuelve una lista con todas las claves existentes en la `Concurrent-HashMap`. Esta operación debe ser no bloqueante (*wait-free*).

[1] a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

- `unsigned int value(string key)`: Devuelve el valor de la clave *key* en la ConcurrentHashMap. Devuelve 0 en caso de que dicha *key* no exista. Esta operación debe ser no bloqueante (*wait-free*).
- `pair<string, unsigned int> maximum(unsigned int n)`: Devuelve el par (k, m) tal que k es la clave, y m es su mayor cantidad de apariciones. No puede ser concurrente con `addAndInc`, sí lo puede ser con `keys` y `value`. El parámetro n indica la cantidad de *threads* a utilizar. Cada *thread* procesará una fila de la tabla a la vez. Si no hubiera más filas por procesar, se deberá terminar su ejecución.

2. Ejercicios

1. Completar la implementación de la clase `ConcurrentHashMap` con las especificaciones anteriormente mencionadas. Para esto debe completar también la implementación de `push_front` de la lista atómica contenida en el código provisto para este trabajo práctico.
2. Implementar las siguientes funciones fuera de la clase que define a la `ConcurrentHashMap` (no está permitido acceder a los atributos privados desde estas funciones):

   a) Implementar una función `ConcurrentHashMap countWordsInFile(string filePath)` que tome la ruta $filePath$ de un archivo de texto como parámetro y devuelva el `ConcurrentHashMap` cargado con las palabras del archivo. Se consideran como palabras las que estén separadas por uno o varios espacios. Para este único caso, se debe hacer la implementación de manera no concurrente.

   b) Implementar una función `ConcurrentHashMap countWordsOneThreadPerFile(list <string> filePaths)` que tome como parámetro una lista $filePaths$ de rutas a archivos de texto y devuelva el `ConcurrentHashMap` cargado con esas palabras. Deberá utilizar un *thread* por archivo.

   c) Implementar una función `ConcurrentHashMap countWordsArbitraryThreads( unsigned int n, list <string> filePaths)` igual a la del punto anterior, pero utilizando n *threads*, pudiendo ser n menor o mayor que la cantidad de archivos. No puede haber *threads* sin trabajo si aun hay archivos por procesar. Debe resolver la forma en que cada *thread* atiende a varios archivos.

   d) Implementar la función `pair<string, unsigned int> maximumOne(unsigned int readingThreads, unsigned int maxingThreads, list<string> filePaths)`, que devuelva el par (k, m) tal que k es la palabra con mayor apariciones en todos los archivos, y m es dicha cantidad.

   A diferencia del ejercicio $2c$, la lectura de las palabras deberá hacerse en múltiples `ConcurrentHashMap`. Esta lectura se hará utilizando la cantidad de *thread*s indicada por el parámetro $readingThreads$. Luego, de esas estructuras se deberá obtener el máximo, utilizando $maxingThreads$ cantidad de *thread*s.

   Este punto se debe resolver sin considerar la implementación de las funciones `countWordsOneThreadPerFile` y `countWordsArbitraryThreads`.

   e) Implementar la función `pair<string, unsigned int> maximumTwo(unsigned int readingThreads, unsigned int maxingThreads, list<string> filePaths)` que se comporte igual que `maximumOne`.

   A diferencia de la primera, ahora podrán realizar la lectura de las palabras en un único `ConcurrentHashMap` y re-utilizar alguna de las versiones concurrentes de `countWords`.

   Nota: **no** puede asumir que la definición de un valor de un puntero sea atómica, como por ejemplo `tabla[k].start = newnode`. Esto quiere decir, que el caracter atómico lo deben establecer en el código de forma explícita.

   La implementación que realicen deberá estar libre de condiciones de carrera y presentar la funcionalidad descrita anteriormente. Además, ningún *thread* deberá escribir un resultado ya resuelto por otro.

3. Realizar un informe que justifique la implementación desarrollada. Dicho informe no debe contener más de cuatro páginas, siendo lo más breve posible.

   Además, deberá realizar pruebas para comparar la *performance* de los ejercicios $2d$ y $2e$ utilizando la función `clock_gettime` (con la opción `CLOCK_REALTIME`) de la biblioteca `time.h` que se puede linkear utilizando `-lrt`. Agregar al informe los resultados obtenidos, utilizando los medios que considere necesarios (gráficos, figuras, etc.).

   Este trabajo práctico provee además algunos archivos de prueba. Se deberá, para los distintos puntos mencionados, describir el resultado obtenido al correr las pruebas, justificando porque los resultados obtenidos muestran que la implementación es correcta.

   Para finalizar, el informe deberá contener a modo de conclusión sus consideraciones con respecto a los siguientes puntos:

   - ¿Qué sentido le ve al uso de *thread*s para resolver tareas de este tipo?
   - ¿Cuáles son los casos que considera posibles, en este escenario, para que exista concurrencia?
   - ¿Cuáles son los casos que considera posibles, en este escenario, para que surjan condiciones de carrera?

3. Entregable

La entrega se realizará de manera electrónica a través de un correo electrónico a la dirección `so-doc@dc.uba.ar`, colocando en el asunto: `[SO;2018;C2;TP1]`. Cada correo deberá contener los datos de cada uno de los integrantes del grupo, y deberá incluir como anexo un único archivo comprimido (tar.gz) que deberá contener **únicamente**:

- El documento del informe (en **PDF**).
- El código fuente debidamente documentado **completo y con Makefiles** de los distintos algoritmos. **NO** debe incluir el código ya compilado: ejecutar `make clean` antes de enviarlo.
- Las modificaciones al *Makefile* para correr los test agregados.

Al recibir correctamente una entrega, se les enviará un mail de confirmación a todos los integrantes del grupo. Si es necesario, podrán realizar múltiples envíos, en cuyo caso solo se considerará el último envío que se haya realizado antes de la fecha límite de entrega.