Trabajo Práctico: *Threading*

Sistemas Operativos - Segundo cuatrimestre de 2024

Fecha límite de entrega: domingo 20 de octubre de 2024 a las 23:59

1. Introducción

En este trabajo práctico profundizaremos en un aspecto central de los sistemas operativos: **la gestión de la concurrencia**. Practicaremos cómo razonar sobre la ejecución concurrente de programas y las técnicas necesarias para gestionar la contención de recursos, evitando que se produzcan **condiciones de carrera**.

Desarrollaremos una solución que simula la gestión concurrente de **órdenes de compra** en una tienda online. Para gestionar el stock de productos en el backend de la plataforma de manera eficiente mientras se procesan múltiples compras, cada equipo deberá implementar y utilizar un `HashMapConcurrente`.

Por simplicidad, abstraeremos la comunicación del sistema a través de pedidos web y nos enfocaremos en la lectura de archivos de texto con las órdenes de compra. El resultado final será una estructura similar a un diccionario o mapa, donde las claves representarán los nombres de los productos y los valores, la cantidad de unidades solicitadas. Cabe destacar que aspectos como el costo o la logística de envíos no serán de interés para este trabajo práctico.

En cuanto a la implementación, desarrollaremos una estructura de datos denominada `HashMapConcurrente`. Esta estructura consiste en una tabla de *hash* abierta que maneja las colisiones mediante listas enlazadas. Su interfaz es similar a la de un *map* o diccionario, donde las claves son *strings* y los valores, enteros no negativos. La finalidad de esta estructura es procesar archivos de texto que contabilicen la aparición de productos (las claves serán los nombres de los productos y los valores, la cantidad de veces que aparecen en las órdenes).

Para lograrlo de manera eficiente, nos enfocaremos en el uso adecuado de *threads*, una herramienta provista por los sistemas operativos que permite ejecutar múltiples hilos concurrentes dentro de un programa. En particular, utilizaremos la interfaz para *threads* `thread` de la biblioteca estándar de C++, la cual emplea internamente `pthreads` del estándar POSIX.

2. Pautas generales

La cátedra les brinda un esqueleto de código sobre el cual trabajar, que será descripto a lo largo de los ejercicios. El mismo incluye un Makefile que podrán usar para compilar el proyecto. También se les proporciona un conjunto de *tests unitarios* para probar los

métodos que deben implementar;[1] estos tests se brindan **únicamente** a modo de guía y **no** evalúan aspectos de concurrencia, por lo que podrían no detectar condiciones de carrera u otros problemas de sus implementaciones. Bajo **ningún** punto de vista estos tests constituyen una garantía de que sus soluciones sean correctas.

El código que escriban deberá seguir buenas prácticas de programación, como usar nombres descriptivos para variables y funciones e incluir comentarios donde consideren necesario realizar aclaraciones. Es fundamental que el mismo esté **libre de condiciones de carrera y *deadlocks*.**

Se espera que el código que entreguen esté acompañado de un informe que describa el proceso de resolución del trabajo práctico. El informe deberá ser claro y prolijo, siguiendo el formato provisto en latex. A su vez debe ser conciso y mantener un hilo conductor adecuado, donde se evaluará la calidad del contenido y cómo es presentado. En cada punto del enunciado se mencionan algunas cuestiones que deben ser abordadas en el informe. Si en el texto del informe incluyen referencias al código, les pedimos, por favor, que las mismas sean claras y eviten copiar y pegar código en el informe.

3. Ejercicios

1. En el archivo `ListaAtomica.hpp` se brinda una implementación parcial de una lista enlazada que utilizaremos para almacenar los elementos de cada *bucket* de la tabla de *hash*. Se pide:

   - En el código, completen la implementación del método `insertar(T valor)`, que agrega un nuevo nodo al comienzo de la lista enlazada. **La operación de inserción debe ser atómica.**
   - En el informe, respondan brevemente y justifiquen: ¿Qué significa que la lista sea atómica? Si un programa utiliza esta lista atómica ¿queda protegido de incurrir en condiciones de carrera? ¿Cómo hace su implementación de `insertar` para cumplir la propiedad de atomicidad?

2. En el archivo `HashMapConcurrente.cpp` se brinda una implementación parcial de la clase `HashMapConcurrente`. Por simplicidad usaremos, a modo de pseudo-función de *hash*, la primera letra de cada palabra (ver el método `hashIndex`). Por lo tanto, nuestra tabla de *hash* tendrá 26 *buckets*, uno para cada letra del abecedario (sin contar la ñ), cada uno de ellos representado por una `ListaAtomica`. Los *strings* utilizados como clave solo constarán de carácteres en el rango a-z (es decir, no habrá strings con mayúsculas, números o signos de puntuación). Se pide:

   - En el código, completen las implementaciones de:
     a) `void incrementar(string clave)`. Si `clave` existe en la tabla (en la lista del bucket correspondiente), se debe incrementar su valor en uno. Si no existe, se debe crear el par `<clave, 1>` y agregarlo a la tabla. Se debe garantizar que **sólo haya contención en caso de colisión de *hash***; es decir, si dos o más *threads* intentan incrementar concurrentemente claves que no colisionan, deben poder hacerlo sin inconvenientes.
     b) `vector<string> claves()`. Devuelve todas las claves existentes en la tabla. Esta operación debe ser **no bloqueante y libre de inanición.**

[1] Pueden compilar y ejecutar los tests con el comando `make test`.

c) `unsigned int valor(string clave)`. Devuelve el valor de clave en la tabla, o `0` si la clave no existe en la tabla. Esta operación debe ser **no bloqueante y libre de inanición.**

Las tres operaciones deben estar **libres de condiciones de carrera y de *deadlock***. Con este fin, podrán agregar a la clase `HashMapConcurrente` las primitivas de sincronización que consideren necesarias, modificando si es preciso el archivo `HashMapConcurrente.hpp`.

- En el informe, expliquen cómo lograron que la implementación esté libre de condiciones de carrera, justificando sus elecciones de primitivas de sincronización. Expliquen también qué decisiones debieron tomar para evitar generar más contención o espera que las permitidas.

3. La clase `HashMapConcurrente` tiene implementado el método `promedio()`, que devuelve el promedio de los valores en la tabla. Se pide:

   - En el código:
     a) La implementación provista de `promedio` **puede ejecutarse concurrentemente con** `incrementar`, lo cual podría traer problemas. Modifiquen la implementación para que estos problemas ya no puedan suceder.
     b) Implementen el método `pair<string, unsigned int> promedioParalelo(unsigned int cantThreads)`, que realiza lo mismo que `promedio` pero repartiendo el trabajo entre la cantidad de *threads* indicada. **Cada *thread* procesará una fila de la tabla a la vez.** Se debe **maximizar la concurrencia**: en ningún momento debe haber *threads* inactivos, y los mismos deben terminar su ejecución si y solo si ya no quedan filas por procesar. No olvidar que debe poder ejecutarse concurrentemente con `incrementar`.
   - En el informe:
     a) ¿Qué problemas puede ocasionar que `promedio` e `incrementar` se ejecuten concurrentemente? Piensen un escenario de ejecución en que `promedio` devuelva un resultado que no fue, en ningún momento, el promedio de los valores de la tabla.
     b) Describan brevemente su implementación de `promedioParalelo`. ¿Tiene la misma firma *(signature)* que la función original promedio o devuelve tipos distintos? ¿Cuál fue la estrategia elegida para repartir el trabajo entre los *threads*? ¿Qué recursos son compartidos por los *threads*? ¿Cómo hicieron para proteger estos recursos de condiciones de carrera?

4. En el archivo `CargarArchivos.cpp`, se brinda una implementación parcial de la función `cargarCompras(HashMapConcurrente hashMap, string filePath)`. La misma lee el archivo indicado mediante `filePath` y carga todas sus palabras en `hashMap`. Se pide:

   - En el código:
     a) Completar la implementación de `cargarCompras`.
     b) Implementar la función `void cargarMultiplesCompras(HashMapConcurrente hashMap, unsigned int cantThreads, vector<string>filePaths)`. La misma carga todos los archivos de compras indicados por parámetro, repartiendo el trabajo entre `cantThreads` *threads*. **Cada *thread* deberá**

ocuparse de un archivo a la vez, teniendo en cuenta las mismas consideraciones mencionadas para *maximoParalelo*.

- En el informe:
  a) ¿Fue necesario tomar algún recaudo especial, desde el punto de vista de la sincronización, al implementar `cargarCompras`? ¿Por qué?
  b) Describan brevemente las decisiones tomadas para la implementación de `cargarMultiplesArchivos`.
- Por ultimo, analizar las siguientes situaciones:
  - En el informe:
    a) ¿Si su sistema tuviera que ser usado bajo durante el Black Friday con millones de compras a la vez, estaría satisfecho con la solución actual? ¿Se le ocurre algún inconveniente o oportunidad de mejora?
    b) En el caso de que la tienda tenga solo dos tipos de productos, ¿Recomendaría el uso de concurrencia? ¿Por que?.
    c) Sin escribir código, si tuvieran que agregar una nueva función llamada `calcularMedianaParalela`, para calcular la mediana de los valores del HashMap de manera eficiente en un entorno concurrente. Analizar en detalle qué dificultades podrían enfrentar al implementar este método. Considerar aspectos como la sincronización entre threads, los valores de retorno, la división de enteros y la combinación de datos. Defina los posibles cuellos de botella, y cómo optimizar el rendimiento mientras se mantiene la integridad de los datos. ¿Qué estrategias adoptarías para minimizar el tiempo de espera de los threads, la comunicación entre procesos y evitar condiciones de carrera?

Nota: las condiciones de carrera no son permitidas, es decir, **no pueden existir salidas que no correspondan a una serialización válida de threads**. Sin embargo, sí se permitirá obtener algunas inconsistencias en cuanto a la desactualización temporal de datos obtenidos, por ejemplo, para las funciones `promedio` y `valor`. Los métodos que se solicitan como no bloqueantes, no deben bloquear **toda** la tabla **todo** el tiempo que ejecutan. De ser necesario, se puede bloquear solo alguna parte de la tabla para alguna operación.

4. Condiciones de entrega

Deberán entregar un archivo comprimido que contenga:

- El informe, en formato PDF.
- El código correspondiente a su solución. El mismo debe poder compilarse sin problemas ejecutando el comando `make`. **NO incluir código compilado**.
- Todos los archivos, datos, scripts y/o instrucciones necesarias para poder replicar cualquier experimento mencionado en el informe.

La entrega se realizará por mail mandando a `so-doc@googlegroups.com` con copia a su corrector asignado. No se aceptarán entregas más allá de la fecha límite, sin excepción y las consultas son presenciales en el horario de la materia.