UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

TRABAJO PRÁCTICO Nº 1

Nombre de grupo: **“Los Borbotones”**

Alumnos:

LAVEZZARI, Fausto

GORRITI, Camilo Nicolás

GIORDA, Marcos Daniel

SIAMPICHETTI, Gino

MAFFINI, Agustín

Profesores:

Dr. Ing. Micolini, Orlando

Ing. Ventre, Luis

Ing. Ludemann, Mauricio

25 de abril 2022

INTRODUCCIÓN

En el siguiente trabajo práctico se detalla y explica la resolución de una consigna utilizando técnicas de programación concurrente. Se mencionan y explican las problemáticas surgidas por la concurrencia de hilos, y cómo se implementa una posible solución a estas.

Los objetivos planteados para el trabajo práctico son los siguientes:

- Implementar correctamente el uso de recursos compartidos.
- Implementar correctamente la sincronización de hilos.
- Entender y diseñar correctamente un diagrama de secuencia
- Entender la relación entre clases para diseñar su diagrama correspondiente.

DESARROLLO

Consigna:

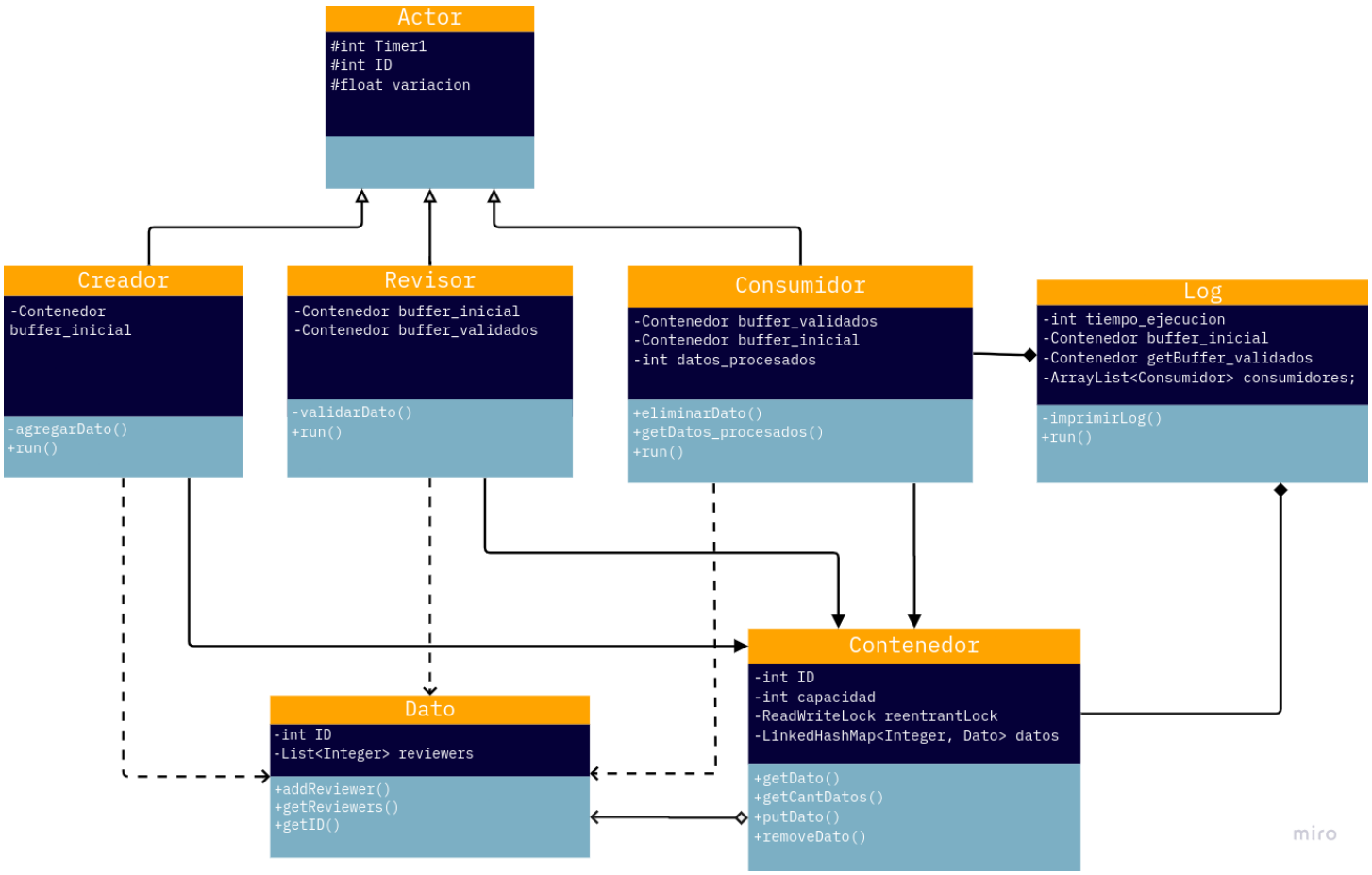
Se plantea un sistema de adquisición de datos con la existencia de dos buffers y tres categorías de actores. Los buffer se denominan inicial y validados, y cuentan con una capacidad máxima de 100 datos (no pueden recibir datos si están llenos, si les llega alguno cuando ya están llenos se descarta y se elimina). En cuanto a los actores serán Creadores de datos, Revisores de Datos o Consumidores de Datos.
El ciclo de funcionamiento normal del sistema comienza con la creación de un dato por parte de un "Creador de Datos". Este proceso lleva un tiempo aleatorio en ms (no nulo, a elección del grupo); una vez creado es almacenado en el Buffer Inicial. En este buffer debe
permanecer hasta que el total de "Revisores" hayan revisado el mismo, la revisión del dato
lleva un tiempo de milisegundos (no nulo a elección del grupo). Una vez que todos los "Revisores" hayan revisado el dato, el último "Revisor" guardará una copia del mismo en el Buffer de Validados. Los "Revisores" no pueden revisar más de una vez cada dato.
Los consumidores de datos son los encargados de eliminar los datos de ambos buffers, siempre y cuando ya hayan sido validados; la eliminación de un dato lleva un tiempo en ms
(no nulo a elección del grupo). Se debe tener en cuenta lo siguiente:

- Los datos tienen un atributo "reviews" el cual registra cuántos revisores han revisado
  el dato hasta el momento.
- Los datos deben ser revisados por todos los revisores del sistema para pasar al buffer de validados. Un revisor no puede revisar más de una vez cada dato.
- Una vez que el dato alcanzó la cantidad de revisiones, debe copiarse en el buffer de
  validados, sin borrarlo del buffer inicial. Este proceso es llevado a cabo por el último
  revisor que revisa el dato.
- Los Consumidores de datos, leen los datos del "buffer de validados" y los eliminan
  de ambos buffers.
- Al finalizar la ejecución es necesario verificar cuantos datos revisó cada revisor.
- Es necesario validar que cada revisor haya revisado solo una vez cada dato.

Además, el sistema debe contar con un LOG con fines estadísticos, el cual registre cada 2 segundos en un archivo:

- Cantidad de datos procesados.
- Ocupación del “Buffer Inicial”.
- Ocupación del “Buffer de Validados”.

Para la realización de esta consigna desarrollaremos un software basado en el siguiente diagrama de clases.



Tres tipos de actores entran en juego (Creadores, Revisores, Consumidores), como comparten una serie de atributos decidimos utilizar una superclase “Actor”, la cual definirá los atributos básicos a todos(id, Timer1, variación). Todos los actores demoran un tiempo para realizar sus acciones, para hacerlo más realista decidimos aplicar una pequeña variación aleatoria a los tiempos base establecidos. Por ejemplo: si Creador tiene asignado demorar 10[ms] por dato, de forma aleatoria este tiempo podrá aumentar entre un 0 y 30 por ciento cada iteración, llegando hasta 13[ms].

Los actores serán los que implementen la interfaz Runnable, siendo los threads a utilizar.

Los datos se componen de solamente dos atributos. Un id para identificarlos y una lista que guarda los revisores que ya lo validaron, los métodos asociados a modificar esta lista están sincronizados para evitar concurrencia de lectura/escritura.
El creador es el encargado de crear los datos y asignarles un ID. Al haber 4 creadores, es importante que estos no generen IDs repetidos. Para ello aplicaremos el siguiente algoritmo.

Cada actor tiene asignado un ID propio, así que tomando este como base (0,1,2 o 3) y sumando la cantidad de creadores (4) por iteración, nunca se crearán IDs repetidas.
Una vez creado el dato este es agregado al buffer inicial a través del método privado agregarDato() que llama a putDato() dentro del buffer.

Utilizaremos una sola clase Contenedor para representar los buffers. Estos guardarán los datos en un LinkedHashMap para poder acceder a ellos fácilmente cuando sean requeridos.
El contenedor recibirá solicitudes para escribir y leer datos constantemente, decidimos protegerlo de la concurrencia utilizando la técnica de ReadWriteLock. Gracias a esto varios hilos en simultáneo pueden acceder a leer los datos teniendo por seguro que mientras lo hagan no habrán otros hilos escribiendo y corrompiendo el almacenamiento. De igual forma, mientras se esté escribiendo no se podrá acceder a métodos de lectura. Activamos el parámetro de "fairness" para que los hilos que están esperando para leer/escribir hace más tiempo tengan prioridad a la hora de acceder.

Los buffers serán los encargados de darle el dato adecuado ya sea al Revisor o al Consumidor dejando a estos solamente la tarea de procesarlos. Esto se hace a través del método getDato(). Como decidimos usar una sola clase para ambos buffers es necesario identificar qué actor está pidiendo un dato. Si es un revisor el buffer buscará entre los datos almacenados alguno que no haya sido validado por este, cuando lo encuentre se lo devuelve y deja de buscar. Si en cambio el que pide es un consumidor, simplemente devolverá el primer elemento del mapa ya que se asume que lo está pidiendo al buffer de validados, como estamos trabajando con un LinkedHashMap el orden de las claves se mantiene, así que al tomar siempre el primer elemento del mapa lo estaríamos utilizando como una cola. En caso de no encontrar un dato por cualquier motivo el método crea y devuelve un nuevo dato con ID = -1 como flag para avisar lo ocurrido.

A la hora de remover un dato primero se preguntará si el ID solicitado está contenido en el buffer, en caso de que no, retorna un false. En cambio sí está y se elimina correctamente devuelve un true.

En cuanto a los revisores, una vez obtienen el dato del getDato(id)) sobre el buffer inicial proceden a esperar el tiempo de validación. Una vez terminan se agregan como validador a la lista de validadores del dato y pregunta cuántas validaciones se hicieron. Si la cantidad de validaciones es igual a la de validadores entonces agrega ese dato al buffer de validados. En caso de que haya más validaciones que validadores quiere decir que ocurrió un error e imprime por consola lo ocurrido.

La tarea principal de los consumidores es eliminar los datos ya validados, debe borrarlos de ambos contenedores. Para esto recibe como parámetro ambos buffers, y obtiene el primer dato validado del buffer de validados con la función getDato(3 (indica que es un consumidor)) de la clase contenedor. Luego elimina el dato de ambos buffers y realiza una demora controlada.

Desde el main se pueden establecer los tiempos base de cada actor y se pone en marcha el programa. Utilizamos una clase Log (Runnable) que obtiene los datos de los contenedores y consumidores y actualiza el log cada 2 segundos

Conclusión en base a resultados:

Para las demoras del sistema, tuvimos en cuenta la cantidad de creadores de datos (4), revisores de datos (2) y consumidores de datos (2). Estableciendo un tiempo base de 4 milisegundos para cada creador de datos, en 1 segundo vamos a tener 1000 datos creados. Luego, con un tiempo de revisor de 6 milisegundos y tiempo de consumidor de 7 milisegundos, obtenemos 920 datos procesados a los 6 segundos y 1224 datos procesados a los 8 segundos, lo cual cumple con la consigna (5<t<10). Además, para los tiempos mencionados, el buffer inicial tiene 99 datos y el buffer de validados 0 datos almacenados. Esto nos dice que los datos creados se generan muy rápido y, a su vez, los datos se validan a menor velocidad de la que se consumen, provocando que el buffer siempre esté en 0. Podemos solventar esto ralentizando el consumo de datos. Por ejemplo, estableciendo un tiempo de consumidor de 14 milisegundos obtuvimos que entre 6 y 8 segundos se procesaron los 1000 datos solicitados, teniendo el buffer inicial siempre lleno y el buffer de validados aproximadamente con 30 datos almacenados. Al seguir teniendo un tiempo de creación muy alto el buffer inicial se mantiene continuamente lleno, pero de igual manera que con el buffer de validados se puede ajustar haciendo más lento el tiempo de creación o aumentando la velocidad de validación y consumición.