Estructuras de Datos.
Facultad de Ciencias, UNAM 2015-2.
Práctica 07: Colas concurrentes.

Armando Ballinas Nangüelú.
armballinas@gmail.com

Fecha de entrega martes 28 de abril de 2015.

1. Introducción.

En esta práctica se implementará la estructura de datos *Cola* así como una variante llamada cola concurrente. Para lo anterior se utilizarán conceptos de concurrencia de hilos de ejecución. El problema a ser tratado es el problema del *productor-consumidor*.

2. El T.D.A. *Cola*.

El tipo de dato abstracto *cola* es una estructura de datos lineal con la característica que sus operaciones son accedidas mediante la propiedad F.I.F.O. (por sus siglas en inglés *First In, First Out*). Esto implica que cuando un elemento es introducido a la cola, este elemento debe ser el primero en ser eliminado de ella. En inglés a esta estructura de datos se le llama *queue*.

Las operaciones básicas de una cola son las de inserción y eliminación, en este caso la primera se conoce como encolar (*enqueue*) y a la segunda como decolar (*deque*). Además el t.d.a provee una operación booleana para determinar si la cola es vacía o no.

Una cola debe tener una forma de almacenar los datos y la manera de implementar las operaciones antes descritas debe manipular la estructura interna de una cola, es por lo anterior que usualmente una cola se implementa mediante otra estructura de datos subyacente, una lista, por ejemplo.

3. El problema del productor-consumidor.

El problema del productor-consumidor es un problema muy famoso en computación y consiste en lo siguiente: se tienen dos partes, la primera es una parte productora que crea datos y los almacena en un búffer, la segunda es la parte consumidora que toma esos datos del búffer.

Este problema tiene muchas analogías con problemas en otras disciplinas como por ejemplo, las líneas de ensamblaje o la administración de paquetes de correspondencia.

En computación, este problema también aparece en otras ramas, pero siempre tiene un detalle inherente a él: la sincronización de las partes.

4. Concurrencia de datos y problemas de sincronización.

Dado que puede haber muchos productores alojando objetos en el búffer y muchos consumidores tomando objetos del búffer debe haber una manera de ponerlos de acuerdo.

Imaginemos que dos productores llegan .ªl mismo tiempo"para colocar un objeto en el búffer, entonces qué criterios se siguen para decidir quién lo coloca primero dado que no pueden colocarlo al mismo tiempo.

Un problema similar surge cuando los consumidores quieren tomar un objeto del búffer.

Hay otro detalle relacionado con el planteamiento del problema. Imaginemos que el búffer está vacío y que hay consumidores esperando por datos, pero el productor también quiere acceder al búffer para colocar datos. En este caso, los consumidores deben "soltar.ªl búffer para dejar que el productor lo use, porque de lo contrario, ni el consumidor obtendrá objetos ni el productor puede crear más pues no ha puesto el que ya creó en el búffer.

A partir del párrafo anterior encontramos un problema muy famoso en computación concurrente, llamado *interbloqueo* (en inglés *deadlock*). Este problema surge cuando se tienen dos o más partes que quieren hacer uso de datos compartidos y debido a un error en el protocolo de acceso ninguno de las partes puede proseguir con su trabajo dado que depende de que otra parte realice una acción. En nuestro caso, las partes son los productores y los consumidores y los datos compartidos son el búffer de donde se ponen y toman los datos. Los consumidores no pueden proseguir porque esperan a que haya datos, pero los productores no pueden poner datos porque los consumidores "les estorban".

5. Implementación del problema del productor-consumidor.

Para poder implementar el problema en Java se debe *modelar*. El modelado significa transformar las partes del problema en programas y datos que interactúen entre sí para resolverlo. En nuestro caso, las partes del problema son tres: los productores, los consumidores y el búffer compartido. Empezaremos describiendo la implementación de este último.

El búffer compartido debe ser una estructura de datos concurrente, es decir, una estructura de datos que permita que múltiples programas accedan a ella de manera segura y sincronizada para evitar problemas de corrupción de datos o de sincronización. Para simplificar las cosas se puede pensar que los productores introducen datos en una parte del búffer y los consumidores los toman de otra. Por lo anterior la estructura de datos elegida es una cola, en este caso, una *cola concurrente*.

La cola concurrente soporta las mismas operaciones que una *cola* simple pero de manera concurrente. Esto en Java se puede lograr haciendo que todos los métodos de la cola sean sincronizados.

Una vez que se tiene la cola concurrente se debe modelar a los productores quienes insertarán (encolarán) datos a la misma. Un productor es un ente, un programa cuya única función es poner objetos en la cola, dado que puede haber muchos productores, entonces se debe modelar que todos ellos interactúen sobre la cola concurrentemente. Esto es de manera natural un hilo de ejecución.

Entonces los productores serán objetos que sean hilos de ejecución (*threads*), es decir, que implementen la interfaz `Runnable` o extiendan la clase `Thread`. Los consumidores se pueden modelar de manera similar, con otro objeto que también sea un hilo de ejecución.

Tanto los productores como los consumidores deben tener lo siguiente:

1. Un valor entero no negativo `thread_id` que es único para cada productor y consumidor.

2. Una cola concurrente de cadenas con la que trabajarán.

3. Dos valores enteros llamados `retardo_min` y `retardo_max`, que indican los extremos del intervalo de tiempo que el hilo descansará entre cada acceso a la cola.

4. Un valor entero `iteraciones` que determina el número de veces que este hilo accede a la cola.

5. Un valor booleano `imprime` que determina si el hilo debe imprimir sus mensajes de funcionamiento o no.

El algoritmo propio del productor es el siguiente:

1. Si `imprime` es verdadero, debe imprimir su identificador de hilo e indicar que ha comenzado a funcionar.

2. Realizará `iteraciones` veces lo siguiente:

   a) Espera un tiempo aleatorio entre `retardo_min` y `retardo_max`.

   b) Encola la cadena `"Token i del hilo id"` en donde `i` es la iteración actual y `id` es el identificador del hilo.

   c) Si `imprime` es verdadero, imprimirá un mensaje que notifique que dicho token fue añadido a la cola. El mensaje debe contener el número de token, el identificador del productor y el nombre de la cola.

El algoritmo para el consumidor es el siguiente:

1. Si `imprime` es verdadero, debe imprimir su identificador de hilo e indicar que ha comenzado a funcionar.

2. Realizará `iteraciones` veces lo siguiente:

   a) Espera un tiempo aleatorio entre `retardo_min` y `retardo_max`.

   b) Saca el siguiente token en la cola.

   c) Si `imprime` es verdadero, imprimirá un mensaje que notifique que sacó el token de la cola. El mensaje debe contener el número de token, el identificador del consumidor y el nombre de la cola.

Observación: Si `iteraciones` tiene un valor negativo, entonces el hilo debe ejecutarse de manera indefinida.

5.1. Detalles de implementación

1. El nombre de las colas debe ser único dado que puede haber varias colas en el programa.

2. La forma para terminar el programa si los hilos se ejecutan de manera indefinida será mediante la combinación de teclas `Ctrl + C`. Cuando esta combinación ocurre, se lanza la excepción `InterruptedException` por lo que deben atraparla y terminar a todos los hilos de manera correcta.

6. Programa de simulación.

Además de modelar el problema se debe crear una clase llamada `Simulador.java` que simulará una ejecución del problema. La clase leerá una serie de argumentos de la variable `args`. Los argumentos serán los siguientes:

1. `num_colas`. Este argumento representa el número de colas concurrentes que su programa debe crear.
2. `num_p`. Este será el número de productores a crear, obsérvese que el número de productores debe ser al menos el número de colas creadas.
3. `it_p`. Representa el número de iteraciones de cada productor.
4. `min_p`. Es el mínimo tiempo de los productores a esperar entre cada acceso a la cola.
5. `max_p`. Es el máximo tiempo a esperar de los productores entre accesos a la cola. Se debe cumplir que $\texttt{max\_p} \geq \texttt{min\_p} \geq 0$
6. `prod_mensaje`. Este argumento determina si los productores imprimirán sus mensajes o no. Si el argumento es no negativo, los productores imprimen su mensaje, en caso contrario los omiten.
7. `num_c`. Este argumento representa el número de consumidores a crear.
8. `it_c`. Es el número de iteraciones a realizar por los consumidores.
9. `min_c`. Es el mínimo tiempo de los consumidores a esperar entre cada acceso a la cola.
10. `max_c`. Es el máximo tiempo a esperar de los consumidores entre accesos a la cola. Se debe cumplir que $\texttt{max\_c} \geq \texttt{min\_c} \geq 0$
11. `cons_mensaje`. Este argumento determina si los consumidores imprimirán sus mensajes o no. Si el argumento es no negativo, los consumidores imprimen su mensaje, en caso contrario los omiten.

Su simulador debe recibir los argumentos en el orden anterior y crear las estructuras necesarias y los objetos adecuados para la simulación con dichos parámetros. Obsérvese que su simulador debe ejecutar todos los productores y consumidores a la vez, sin embargo, debe repartirlos uniformemente entre las colas, es decir, si se tienen n colas, p productores y c consumidores, entonces cada cola debe tener p/n productores, salvo tal vez la última y cada cola debe tener c/n consumidores, de nuevo, salvo tal vez, la última.

Para lanzar los hilos de los productores y consumidores es necesario que creen un ejecutor. Estos ejecutores permiten administrar el lanzamiento de hilos y se verán en el laboratorio.

7. Ejercicios.

1. (1 pt.) Hacer un archivo `ColaList.java` donde se implemente el tda Cola provisto por la interfaz `Cola.java` usando como estructura de datos subyacente. una lista creada por ustedes, es decir, ListaArreglo, ListaLigada o ListaDoblementeLigada, debido a la naturaleza de una cola se recomienda ampliamente esta última.

2. (2 pts.) Extender la clase `ColaList` por medio de una clase llamada `ColaConcurrente` para que se vuelva una cola concurrente mediante el uso de métodos sincronizados.

3. (4 pts.) Crear las clases `Productor` y `Consumidor`.

4. (2 pts.) Crear la clase `Simulador`.

5. El último punto de su práctica se evaluará el martes 28 de abril en el laboratorio mediante un pequeño examen y un ejercicio práctico.

8. Archivos a entregar.

1. `ColaList.java` que contiene la implementación de una cola basada en una lista.

2. `ColaConcurrente.java` que contiene la implementación de una cola concurrente.

3. `Productor.java` que contiene la implementación del productor.

4. `Consumidor.java` que contiene la implementación del consumidor.

5. `Simulador.java` que contiene la implementación del simulador.

Además deben entregar la interfaz `Cola.java` así como todos los archivos necesarios para la implementación de la lista utilizada en la cola.