FAQ – v.4
Ejercicio 6 – Preguntas frecuentes

josé a. mañas

25.4.2016

1 ¿Qué hay que hacer?

Una caché (un diccionario) que recibe peticiones de lectura (get()) y escritura (put()) de varias *threads* concurrentes. Hay que hacer la caché *thread-safe*.

1.1 ¿Cómo se programa? Estructura de datos

Usaremos un diccionario implementado por medio de una tabla hash con desbordamiento en listas. El número de ranuras se especifica en el constructor.

Puede partir de su código del ejercicio 3, adaptándolo a los requisitos de acceso concurrente de este ejercicio.

1.2 ¿Cuándo se crean las listas de desbordamiento?

Puede crear una lista para cada ranura (slot) en el constructor y dejarlas ahí para siempre.

O puede dejar las ranuras con null y crear listas cuando las necesite.

Si crea las listas en el constructor …

> En este caso el ejercicio es bastante simple: no hay que proteger accesos concurrentes en TsCache y sólo hay que proteger con su monitor cada objeto TsList.

Si crea las listas cuando las necesita …

> En este caso el ejercicio se complica un poco porque es posible que un *thread* esté escribiendo en una ranura mientras otro está leyendo o que 2 *threads* intenten escribir. Como esto puede ocurrir, necesita otro monitor de tipo readers-writers para proteger las operaciones en TsCache.
>
> No es difícil, pero la otra opción es más sencilla.

1.3 Pasos

Puede ser interesante hacer primero la versión 0

> http://www.dit.upm.es/~pepe/doc/adsw/ejercicio6/ej6_v0.pdf

que simplemente admite una (1) operación de lectura o una (1) de escritura simultáneamente. Sin RW_Monitor. Y luego, cuando ya está la plataforma funcionando, se optimiza el comportamiento con el monitor.

Este paso intermedio es opcional.

2 Estado

El estado de la caché son las entradas <clave, valor> que tiene almacenadas.

Los *threads* usuarios alteran el estado de la caché al escribir (put).

Es necesario proteger el estado como una zona de exclusión mutua. Hay que proteger tanto las escrituras para no corromper el diccionario, como las lecturas para que se lea exactamente lo último que se ha escrito asociado a una clave.

Las operaciones de determinación de la ranura pueden realizarse concurrentemente sin necesidad de establecer una zona crítica.

Las operaciones en las listas de desbordamiento deben protegerse como zonas críticas.

2.1 ¿Qué hacemos con el método size() de Diccionario?

Lo hemos dejado fuera del ejercicio para evitarnos una definición complicada.

En un entorno concurrente, el tamaño exacto del diccionario es un poco arriesgado de medir. En términos más precisos ¿queremos que size() sea una operación atómica excluyendo otros accesos a la cache?

Si el tamaño se mide dinámicamente agregando el tamaño de cada lista, tendríamos que parar todos los accesos de modificación a TsCache mientras se está sumando. Esto habría que hacerlo con otro monitor del tipo readers-writers para que las demás operaciones sólo se frenaran mientras hay un size (comportamiento tipo 'writer') en curso.

La definición es especialmente insidiosa cuando pensamos en la operación clear(), que también funcionaría tipo 'writer'.

¿Vale la pena ser tan estricto? Hemos pensado que no vale la pena el esfuerzo y se ha obviado el método size().

2.2 ¿Qué hacemos con el método clear() de Diccionario?

La solución fácil es llamar en un bucle al método clear() de cada TsList. Como el método clear() está protegido por el monitor, las estructuras de datos no se corrompen.

Otro tema es saber qué significa o implica exactamente hacer un clear() no atómico sobre una estructura de datos que está siendo modificada concurrentemente. Significa que eliminamos los datos cargados antes; pero no significa necesariamente que al final el diccionario esté vacío. Y el concepto de "antes" depende de la política de ejecución de tareas de la plataforma de ejecución, salvo control explícito por parte del programador.

3 ¿Cómo se establece una zona de acceso exclusivo?

Usaremos un monitor auxiliar al que pedimos permiso para ponernos a leer o permiso para ponernos a escribir. El permiso correspondiente se devuelve al terminar.

El monitor se hace con una clase java RW_Monitor, que ofrece 4 operaciones

```
/**
 * Solicitud de permiso para hacer una lectura.
 * La thread que llama se queda esperando hasta que pueda entrar.
 */
public synchronized void openReading() { … }
```

```
/**
 * Devolución del permiso de lectura.
 */
public synchronized void closeReading() { … }
```

```
/**
 * Solicitud de permiso para hacer una escritura.
 * La thread que llama se queda esperando hasta que pueda entrar.
 */
public synchronized void openWriting() { … }
```

```
/**
 * Devolución del permiso de escritura.
 */
public synchronized void closeWriting() { … }
```

```
/**
 * Getter.
 *
 * @return numero de lectores autorizados en este momento.
 */
public synchronized int getNReadersIn() { … }
```

```
/**
 * Getter.
 *
 * @return numero de escritores autorizados en este momento.
 */
public synchronized int getNWritersIn() { … }
```

Así, por ejemplo, para hacer una operación de búsqueda de una clave en la lista:

1. se llama al método openReading() esperando hasta tener el permiso

2. se busca en la lista hasta encontrar la clave (devolviendo el valor asociado) o hasta decidir que la clave no está (devolviendo null)
3. se llama al método closeReading()

Hay que proteger adecuadamente todas las operaciones

- lectura: get()
- escritura: put(), remove() y clear()

3.1 ¿Qué pasa si no emparejamos bien?

O sea, si un openReading() pretende cerrarse con un closeWriting(). O si un openWriting() pretende cerrarse con un closeReading().

No está definido; pero puede optar por lanzar una excepción del tipo

java.lang. IllegalMonitorStateException

4 Política de acceso (fairness policy)

Hay que llevar cuenta de cuántos *threads* hay leyendo y cuántos escribiendo en cada momento.

Hay varias opciones. El alumno puede elegir alguna de las siguientes. Indique claramente en la cabecera de la clase RW_Monitor qué política implementa.

Laxa (*laissez faire*)

Nos limitamos a impedir que haya operaciones de lectura y escritura al tiempo, y a impedir que haya más de una operación de escritura al tiempo. Es decir, en un momento dado puede haber varias lecturas simultaneas o una sola escritura o nadie.

Corremos el riesgo de que muchas lecturas detengan indefinidamente a alguna escritura. Se dice que se produce un problema de hambruna o inanición (*starvation*)

Equitativo

Añadimos la regla de que cuando sale un lector, si hay un escritor este tiene prioridad sobre otros lectores. Y viceversa. O sea, que intentamos dar tantas opciones a los lectores como a los escritores.

Prioridad para los escritores

Añadimos el requisito de que, si hay un escritor esperando para entrar, los lectores deben esperar.

Corremos el riesgo de que los escritores monopolicen el monitor y los lectores esperan indefinidamente.

Prioridad para los escritores con un límite en el número de peticiones de lectura pendientes.

Añadimos el requisito de que haya como mucho N lectores esperando. Si se da esa circunstancia, forzamos que todos los lectores sean atendidos antes de atender a más escritores.

Si el alumno prefiere implementar alguna otra política,
debe describirla claramente en la cabecera de la clase RW_Monitor.

5 Pruebas

Todo sistema debe ser probado antes de entregarlo.

Un fallo en las pruebas indica que hay un error que debe ser reparado antes de entregarlo.

Pasar todas las pruebas no implica que el sistema carezca de errores: simplemente es que no los hemos encontrado.

Un buen conjunto de pruebas reduce la probabilidad de que el sistema aún contenga errores.

5.1 Unitarias de corrección

Use JUnit para validar los métodos de la caché.

JUnit ejecuta los tests en orden aleatorio; pero no concurrente. Se puede usar para probar la corrección de los métodos sin concurrencia.

Use las pruebas del ejercicio 3.

5.2 De concurrencia

Haga pruebas de humo (*smoke test*) con varios agentes que la usan concurrentemente.

Por ejemplo

```
public class TsCacheSmokeTest {
   public static final int N_SLOTS = 10;
   public static final int N_AGENTS = 50;

   public static void main(String[] args)
         throws InterruptedException {
      TsCache cache = new TsCache(N_SLOTS);
      for (int id = 0; id < N_AGENTS; id++) {
         TestAgent agent = new TestAgent(id, cache);
         Thread thread = new Thread(agent);
         thread.start();
      }
   }
}
```

Puede usar un agente de este tipo

```
public class TestAgent
      implements Runnable {
   private final Random random = new Random();
   private final int id;
   private final TsCache cache;
```

```
   /**
    * Constructor.
    *
    * @param id    identidad del agente.
    * @param cache cache compartida.
    */
   public TestAgent(int id, TsCache cache) {
      this.id = id;
      this.cache = cache;
   }
```

```
   /**
    * Ejecucion concurrente.
    */
   @Override
   public void run() {
      while (true) {
         try {
            String key = String.valueOf(random.nextInt(1000));
            if (cache.get(key) == null) {
               String val = "{" + key + "}";
               Nap.random(10, 20);
               cache.put(key, val);
            }
            Nap.sleep(10);
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

propiedades que deben satisfacerse
corrección (correctness) el diccionario cumple sus funciones de asociar claves y valores **seguridad (safety)** el estado no se corrompe; un estado corrupto genera excepciones se verifican las premisas de exclusión mutua de lectores y escritores **vivacidad (liveness)** el sistema no se queda congelado **equidad (fairness)** todas las listas atienden lecturas y escrituras

Las pruebas de corrección se hacen con JUnit.

Las pruebas de seguridad se pueden hacer con aserciones:

- cada vez que conseguimos un permiso de lectura verifique cuántos somos

  monitor.getNWritersIn() debe ser 0

- cada vez que conseguimos un permiso de escritura verifique cuántos somos

  monitor.getNWritersIn() debe ser 1

  monitor.getNReaders() debe ser 0

Puede usar la clase My.

Las pruebas de equidad se hacen con LogViewer

- cree un campo para referirse al visor común a todas las listas

  private LogViewer viewer = LogViewer.getInstance();

- cada vez que conseguimos un permiso de lectura o escritura, llame al visor

  viewer.dump(this, monitor.getNReadersIn(), monitor.getNWritersIn());

LogViewer abre una ventana para ir viendo el número de lectores (en verde) y de escritores (en rojo) en cada momento.

6 Entrega

- RW_Monitor.java
- TsCache.java
- TsList.java
- TsCacheTest.java
- … otras clases que haya usado