Laboratoire 1

Monitoring lecteurs-rédacteurs

Jeanne Michel | ISC3il-a

Cours : Paradigmes de programmation avancés 2
Présenté à : Aïcha Rizzotti et Julien Senn

21 avr. 2023

Sommaire

1 Introduction

Dans le cadre du cours de paradigmes de programmation avancés 2, un laboratoire sur le thème du paradigme lecteurs-rédacteurs est à réaliser. Six périodes en classe sont prévues pour ce travail. Une interface minimale est fournie, et nous devons ajouter ce qu'il est nécessaire pour faire fonctionner notre solution de monitoring. L'affichage peut se faire en mode console ou sur UI.

L'application à réaliser doit permettre de visualiser les files d'attentes des différents objets de synchronisation. Les lecteurs et rédacteurs partagent des documents. L'accès au document doit se faire de manière concurrente.

Le but est donc de réaliser un framework pour le monitoring de la concurrence, et plus précisément des files d'attente sur les éléments de synchronisation en Java, en utilisant le pattern Lecteurs-Rédacteurs.

2 Implémentation

2.1 Classe Document

La classe document représente la ressource concurrente que les threads se partagent. Elle contient uniquement une chaîne de caractères et un nom. Dans le TODO de la classe, il est demandé d'ajouter de locks au document afin d'assurer un accès concurrent à celui-ci.

2.1.1 Mécanisme de lock

Afin de garantir un accès concurrent au document, une variable lock de type *ReentrantReadWriteLock* a été ajoutée. La classe *ReentrantReadWriteLock* de Java est une implémentation de *ReadWriteLock*, qui prend en charge la fonctionnalité *ReentrantLock*.

Le *ReadWriteLock* est une paire de verrous associés, l'un pour les opérations de lecture seule et l'autre pour l'écriture. Le verrou de lecture peut être détenu simultanément par plusieurs threads de lecture, tant qu'il n'y a pas d'écriture. Le verrou d'écriture est exclusif.

Afin de pouvoir accéder au verrou, un getter a été ajouté.

```
// Getter du lock
public ReadWriteLock getLock() {
	return lock;
}
```

```
// Exemples d'utilisation des verrous
document.getLock().readLock().lock();
document.getLock().writeLock().lock();
```

2.2 Classe Main

La classe Main représente le thread principal gérant le bon déroulement et le démarrage de l'application. Il contient une FutureTask contenant le déroulement du programme pouvant être arrêté sur demande.

Les TODOS à remplir dans cette classe sont : Demander à l'utilisateur d'entrer un nombre de documents et un nombre de personnes, lire en continu les inputs de l'utilisateur pour permettre de visualiser la prochaine opération des logs ou de quitter le programme.

2.2.1 Entrées de l'utilisateur

Afin d'accéder aux entrées de l'utilisateur, un objet Scanner est utilisé. Cette classe permet d'obtenir les données de l'utilisateur. La méthode nextInt est utilisée afin de s'assurer que l'utilisateur entre en nombre entier entre un et neuf. Tant que l'utilisateur n'entre pas un nombre valide, la question se répète.

```
// ask user for number of documents (between 1 and 9)
do {
   System.out.print("Insert number of concurrent documents (max 9) : ");
   nbDocuments = scanner.nextInt();
} while (nbDocuments < 1 || nbDocuments > 9);
```

2.2.2 Opérations

La lecture en continu des inputs de l'utilisateur est faite afin qu'il puisse choisir la prochaine action à effectuer. S'il entre la phrase "EXIT", la tâche s'arrête et tous les threads sont interrompus. S'il écrit autre chose, le log suivant est affiché. À la fin du programme, tous les threads sont interrompus.

```
while (!consoleTask.isCancelled()) {
     System.out.println(" > Press ENTER to continue or type EXIT to stop
the program < ");
     // read user input
     String input = scanner.nextLine();
     // if user wants to exit
     if (input.equals("EXIT")) {
          // cancel console task
          consoleTask.cancel(true);

          // interrupt all threads
          System.out.println("interrupting all threads...");

          for (Thread thread : threads) {
               thread.interrupt();
          }
```

```
     } else {
          // print next log
          waitingLogger.popNextLog();
     }
}
scanner.close();
```

2.3 Classe Person

La classe Person représente à la fois les lecteurs et les rédacteurs. Elle implémente l'interface runnable et possède un nom, un document à traiter, un temps et une durée de traitement, ainsi qu'un timer interne afin de garder une trace du temps passé en activité.

Dans cette classe, il est demandé de faire patienter la personne tant que le temps écoulé ne dépasse pas son temps de départ. Une fois lancé, il faut ajouter la personne dans la file d'attente d'accès à son document. Une fois que la personne a accès au document, il faut gérer la lecture ou l'écriture.

2.3.1 Gestion de la lecture/écriture

Afin de faire patienter la personne tant que le temps écoulé ne dépasse pas son temps de départ, la méthode sleep de Thread est lancée. Afin de s'assurer que la personne n'attende pas trop longtemps, le temps passé depuis sa création est déduit du temps d'attente.

```
Thread.sleep(startingTime - timer.timePassed());
```

À la fin du sleep, la personne est ajoutée dans la liste d'attente d'accès du document.

Si le rôle de cette personne est la lecture, la première étape est d'acquérir le verrou en lecture du document. Une fois le verrou acquis, l'utilisateur est au logger pour signaler la fin de l'attente, et la lecture est effectuée. Suite à cette opération, la méthode sleep est appelée afin d'attendre le temps d'exécution demandé. Finalement, l'utilisateur est envoyé au logger pour signaler la fin de l'accès, et le verrou est rendu.

```
doc.getLock().readLock().lockInterruptibly();
waitingLogger.removeWaiting(this, timer.timePassed());

long startReading = timer.timePassed();
doc.readContent();
Thread.sleep(durationTime - (timer.timePassed() - startReading));

waitingLogger.finished(this, timer.timePassed());
doc.getLock().readLock().unlock();
```

Si le rôle de la personne est l'écriture, le verrou requis est le *writeLock*, et la méthode *setContent* est appelée.

Le tout est contenu dans un bloc try-catch, afin de pouvoir gérer l'exception *InterruptedException* lorsque le thread est interrompu. La méthode *lockInterruptibly* est utilisée car contrairement à la méthode *lock*, elle peut être interrompue.

2.4 Classe WaitingLogger

La classe WaitingLogger sert à stocker l'état actuel des files d'attentes, c'est un singleton.

Dans cette classe, il est demandé de prévoir un stockage concurrent pour les logs et pour les listes d'attente des documents. Plusieurs fonctions doivent être implémentées : ajout d'une personne dans la liste d'attente d'accès à son document, enlever une personne de la liste d'attente de son document, indiquer la fin d'accès à un document. Une fonction de récupération et traitement du prochain log est à créer. La gestion des logs permet d'indiquer l'attente d'un thread sur sa ressource, l'accès d'un thread à sa ressource partagée, ainsi que la fin d'un thread et la libération de l'accès à sa ressource. Ces logs doivent être affichés sur l'UI ou la console.

2.4.1 Stockage concurrent

Afin de stocker les différentes listes d'attente des documents, ainsi que les logs, la classe *ConcurrentLinkedQueue* a été utilisée. Cette classe est utilisée pour mettre en œuvre une file d'attente à l'aide d'une *LinkedList* de manière concurrente. C'est une implémentation non bornée de la file d'attente selon la méthode FIFO. Elle peut être utilisée lorsqu'une file d'attente est partagée entre plusieurs threads.

Quatre files d'attente sont prévues :

- *waitingList* : Liste des personnes en attente.
- *processingList* : Liste des personnes en traitement sur leur dossier.
- *finishedList* : Liste des personnes ayant fini le traitement.
- *logs* : Liste des logs des différentes étapes.

2.4.2 Ajout des logs

L'ajout des logs se fait dans trois méthodes : *addWaiting*, *removeWaiting* et *finished*. Ces méthodes sont appelées depuis la classe *Person*, lors de l'attente et de l'accès à la ressource document.

Ces méthodes ajoutent un nouveau log à la liste des logs, avec le type correspondant.

```
public void addWaiting(Person p, long timer) {
      logs.add(new Log(Log.Type.WAITING, p, timer));
}
```

2.4.3 Traitement des logs

Lorsque l'utilisateur veut afficher le prochain log, la méthode *popNextLog* est appelée. Cette méthode prend le premier élément de la liste des logs. Ensuite, selon le type du log, les différentes files d'attente vont être mises à jour. Une chaîne de caractère est aussi définie après l'ajout aux listes, afin d'afficher plus facilement le diagramme. Si aucun log n'est disponible, la méthode retourne et un message d'avertissement est affiché.

```
if (logs.size() == 0) {
      System.out.println("No logs available yet, wait a moment...");
      return;
}

Log nextLog = logs.poll();
Person p = nextLog.getPerson();

// Treat log type
switch (nextLog.getType()) {
      case WAITING:
            waitingList.add(p);

            // add to log : multiple spaces (number of time of log) + "W"
            String log = p.getLog();
            log += " ".repeat((int) nextLog.getTime() / 100) + "W";
            p.setLog(log);

            // ...
}
//...
```

2.4.4 Affichage des logs

L'affichage du log est en trois parties :

- La liste des personnes (threads) avec leur nom, leur rôle, leur temps de départ, leur durée d'exécution ainsi que leur document
- La liste des personnes en attentes pour chaque document
- Un diagramme

Threads list

L'affichage de la liste des threads se fait en itérant sur chaque objet de la liste persons. À chaque itération on récupère le nom, le rôle, le temps de départ, la durée et le document de la personne pour l'affichage.

Queues state

L'affichage des listes d'attente pour chaque document se fait en plusieurs étapes. D'abord, on itère sur chaque document. Lors de l'itération, on fait une liste de toutes les personnes en attente pour ce document, ainsi qu'une liste des personnes en train de traiter le document.

Ensuite, on crée deux chaînes de caractères qui concatènent les noms des personnes de chacune des listes précédentes. La dernière étape est d'afficher les chaînes de caractères.

```
db.getNames().forEach(name -> {
      // find all the persons that are waiting for this document
      List<Person> waitingForThisDocument = waitingList.stream()
            .filter(person -> person.getDocument().getName().equals(name))
            .collect(Collectors.toList());

      // concatenate the waitingForThisDocument persons names
      String waitingForThisDocumentNames = waitingForThisDocument
            .stream()
            .map(person -> person.getName() + " ("
                  + person.getRole().toString() + ") ")
            .collect(Collectors.joining());

      // display the document name and the persons that are processing
      // and waiting for it
      System.out.println(name + " (WAITING) : "
            + waitingForThisDocumentNames);
// ...
```

Diagram

Le diagramme affiche l'état des différents threads sur une échelle graduée sur le temps écoulé. Lorsqu'un thread est mis en attente, un 'W' s'affiche à côté du thread au temps correspondant. Les autres états sont 'R' (enlevé de l'attente), 'T' (a directement démarré sa tâche sans passer longtemps à l'attente), et 'F' (a fini le travail).

Tout d'abord, on affiche la ligne de temps avec des nombres espacés de 10 caractères entre chaque nombre. Pour ce faire, une boucle for est utilisée pour afficher chaque nombre sur la ligne de temps, avec des espaces ajoutés pour que chaque nombre soit séparé par 10 caractères. Deuxièmement, on utilise une boucle for-each pour parcourir chaque Person dans la liste. Pour chaque Person, on vérifie si cette personne est actuellement en train d'être traitée en recherchant la personne dans une autre liste (processingList). Si la personne est trouvée, on calcule le nombre de traits supplémentaires nécessaires pour afficher l'état en traitement de la personne sur la ligne de temps, en fonction de la différence entre le temps actuel et le temps d'enregistrement de la personne. La méthode setLog est ensuite appelée sur l'objet Person pour mettre à jour la représentation graphique de l'état d'attente de cette personne. Enfin, on affiche le nom, le rôle et la représentation graphique de l'état d'attente de chaque *Person*.

3 Résultats

```
------------------------------
| Java Concurrency Monitoring |
------------------------------
Insert number of concurrent documents (max 9) : 2
Insert number of readers / writers (max 9) : 7
 > Press ENTER to continue or type EXIT to stop the program <
No logs available yet, wait a moment...
```

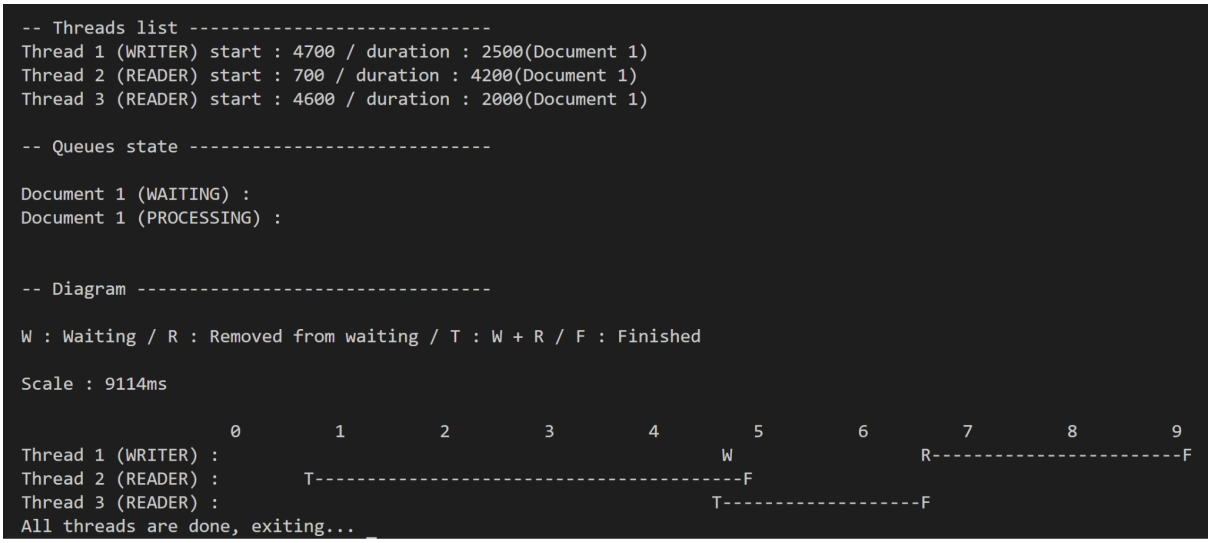
Début du programme

Comme on peut le voir sur l'image ci-dessus, au début du programme, l'utilisateur doit entrer le nombre de documents souhaités, ainsi que le nombre de threads. Si la liste des logs est vite, un message d'avertissement est affiché. L'utilisateur a le choix entre consulter le log suivant ou arrêter le programme.

```
-- Threads list ----------------------------
Thread 1 (READER) start : 800 / duration : 2200(Document 1)
Thread 2 (WRITER) start : 3200 / duration : 4300(Document 2)
Thread 3 (WRITER) start : 2800 / duration : 2500(Document 1)
Thread 4 (READER) start : 1400 / duration : 2900(Document 2)
Thread 5 (WRITER) start : 1200 / duration : 1600(Document 2)
Thread 6 (READER) start : 4700 / duration : 2300(Document 2)
Thread 7 (WRITER) start : 1200 / duration : 2900(Document 2)

-- Queues state ----------------------------

Document 1 (WAITING) :
Document 1 (PROCESSING) : Thread 1 (READER)

Document 2 (WAITING) : Thread 7 (WRITER) Thread 4 (READER)
Document 2 (PROCESSING) : Thread 5 (WRITER)
```

Affichage de : threads list et queues state

Comme on le voit sur l'image ci-dessus, la liste des threads est affichée, ainsi que la liste d'attente et la liste de traitement de chaque document.



Dernier log

L'image ci-dessus affiche le dernier log d'un déroulement du programme. Le diagramme est complet et contient les états de tous les threads durant le programme. Lorsque le programme est terminé, un message est affiché pour avertir l'utilisateur.

4 Conclusion

Lors de ce travail, nous nous sommes basés sur la vidéo de démonstration pour la réalisation de l'interface graphique. Comme vu dans les résultats, l'affichage des différentes étapes est fonctionnelle et se rapproche au maximum de la vidéo fournie. De manière générale, nous avons répondu au cahier des charges et nous avons pu appliquer nos connaissances sur le paradigme lecteurs-rédacteurs ainsi que sur le principe du *ReentrantReadWriteLock*.