université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

mardi 25 octobre 2022

Traitement distribué de données
M1 ISD

TP nº 1

Consignes les exercices ou questions marqués d'un ⋆ devront être d'abord rédigés sur papier (afin de se préparer aux épreuves écrites de l'examen). En particulier, il est recommandé d'être dans les même conditions qu'en examen : pas de document ni de calculatrice. Tous les TPs se font sous Linux.

1 Utilisation de la mémoire partagée

Le but de cet exercice est de coder une petite application Client/Serveur utilisant deux concepts important en informatique :

— les *verrous* sur des portions de fichiers
— des zones de mémoire associées à des fichiers (*mapped-file memory*)

xc Les spécifications des programmes sont comme suit :

1.1 Serveur

Le serveur présente à l'utilisateur un petit menu (en console) permettant d'effectuer l'une des 4 actions suivantes :

— **i**nitialisation d'un fichier de $N \times 4$ octets (tous nuls), permettant de contenir N entiers Java.
— **c**ommencement du *monitoring* d'un fichier. Toutes les 5 secondes, le serveur affiche dans la console la valeur contenu pour chaque entier du fichier.
— **t**erminaison du *monitoring* (on arrête d'afficher le contenu du fichier dans la console)
— **q**uitter : on quitte le programme, le serveur s'arrête.

Le nombre N d'entiers que l'on peut mettre dans le fichier ainsi que le chemin du fichier sont des constantes (par exemple 4 et "`/tmp/tp01.map`").

1.2 Client

Le client prend en argument sur la ligne de commande un entier i entre 0 et $N-1$ (l'entier est ramené à ces bornes s'il dépasse). Le client effectue M fois de suite les opérations suivantes :

— lire l'entier n qui se trouve à la position i du fichier
— écrire $n+1$ à la position i du fichier

On souhaite implémenter les clients de manière à ce qu'ils puissent compter indépendamment même si deux clients sont associés à la même case (dans ce cas, la case contiendra $2 \times M$ en fin de calcul).

1.3 `java.nio`

Le package `java.nio` de la bibliothèque standard de Java permet d'effectuer des opérations bas niveau efficace sur les fichiers et fournit aussi une abstraction à certaines primitives des systèmes d'exploitation. Les classes importantes pour la réalisation du serveur et du client sont les suivantes :

— `java.nio.channels.FileChannel` correspond à la notion Unix de fichier, permet d'ouvrir le fichier dans plusieurs *modes* (lecture, écriture, création, troncation, …). Voir en particulier la méthode « `.open(...)` ». On utilisera les classes utilitaires :
  — `java.nio.file.Paths` qui permet d'obtenir un objet `Path` à partir d'une chaîne de caractères
  — `java.nio.file.StandardOpenOption` qui définit les différents modes d'ouverture des fichiers
— `java.nio.MappedByteBuffer` un tableau de mémoire paginée (tel que présenté en cours). Ce tableau s'obtient via la méthode « `.map(...)` » d'un « `FileChannel` ».

— `java.nio.IntBuffer` un tableau d'entier que l'on peut obtenir à partir d'un « `MappedByteBuffer` » en utilisant « `.asIntBuffer()` ». Ce n'est pas une copie mais une simple vue de la mémoire paginée où les octets sont lus 4 par 4
— `java.nio.FileLock` un verrou sur une région d'un fichier. Un tel verrou s'obtient à partir des méthodes « `.lock()` » et « `.tryLock()` » de la classe « `FileChannel` ».

1.4 Questions

Récupérer l'archive sur la page du cours et l'importer dans Eclipse (Import → Project ...).

1. ⋆ Décrire en pseudo-code la méthode « `.initMapFile()` ». En particulier, quelle portion de votre code doit s'exécuter en section critique, c'est à dire sans qu'un client puisse écrire dans le fichier « `MAP_FILE` ».
2. Implémenter la méthode « `.initMapFile()` » en Java.
3. ⋆ la méthode « `.openMapFile()` » doit procéder de la manière suivante : ouvrir le fichier « `MAP_FILE` » en lecture seule, créer un `MappedByteBuffer` et fermer le fichier. Est-il nécessaire d'avoir un accès exclusif sur le ficher ?
4. Implémenter la méthode « `.openMapFile()` » en Java.
5. ⋆ Étudier le code de la méthode « `.monitor(. . .)` ». Se reporter à la documentation de Java pour expliquer ce que font les classe « `Timer` » et « `TimerTask` ». La méthode « `.monitor(. . .)` » contient une classe *anonyme* qui étend la classe « `TimerTask` ». Quel est l'avantage d'utiliser une classe anonyme plutôt que de définir une classe auxiliaire « `MyTimerTask` » ailleurs dans le fichier ?
6. compléter le corps de la méthode « `.run()` » de la classe anonyme. Ce dernier doit afficher sur une ligne les $i : v_i$ pour tout i entre 0 et « `MAX_NUM` ». v_i est la valeur stockée à l'indice i, ainsi que le temps écoulé depuis le premier affichage, en secondes. Exemple :

   ```
   0: 0  1: 0  2: 0  3: 0  (45s)
   ```
7. Compléter la méthode « `.main(. . .)` » de manière à ce qu'elle appelle « `.mainLoop()` ». Lancer le serveur et vérifier qu'il fonctionne correctement (en particulier vérifier qu'il crée bien un fichier `/tmp/tp01.map` ne contenant que des octets nuls).
8. ⋆ Écrire le pseudo code du client en s'intéressant en particulier à la section critique (i.e. comment disposer le verrou)
9. Implémenter dans un premier temps la méthode « `.mainLoop()` » du client en utilisant la version de « `.lock()` » sans argument
10. ⋆ tester le client en lançant une instance avec comme argument 0. Quel est le résultat affiché par le serveur une fois que le client termine ?
11. ⋆ Se déplacer dans le répertoire « `Workspace/TDD_TP1_ACOMPLETER/bin` » se trouvant à la racine du répertoire utilisateur (où `Workspace` est le répertoire contenant les projets Eclipse). Exécuter simultanément 4 instances du client avec le même entiers :

    ```
    java -cp . tdd.tp01.MMapClient 0 &
    java -cp . tdd.tp01.MMapClient 0 &
    java -cp . tdd.tp01.MMapClient 0 &
    java -cp . tdd.tp01.MMapClient 0 &
    ```

    (le `&` permet au *shell* de rendre la main et passe la commande en tâche de fond). Déterminer approximativement le temps nécessaire pour terminer les quatre clients (en regardant les affichages serveur).
12. ⋆ Recommencer maintenant en appelant 4 fois le client avec des entiers différents :

    ```
    java -cp . tdd.tp01.MMapClient 0 &
    java -cp . tdd.tp01.MMapClient 1 &
    java -cp . tdd.tp01.MMapClient 2 &
    java -cp . tdd.tp01.MMapClient 3 &
    ```

    Combien de temps cela prend il ? Est-ce que votre solution est optimale ?
13. Remplacer maintenant dans la méthode `.mainLoop()` du client l'appelle à `.lock()` par un appel choisissant judicieusement la partie du fichier à verrouiller. Recommencer la manipulation de la question précédente. Que constate t'on ?