SCIENCES SORBONNE UNIVERSITÉ

# Examen Réparti 1 PSCR
# Master 1 Informatique Nov 2023

UE 4I400

Année 2023-2024

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones. . . )
Clé USB en lecture seule autorisée.

### Introduction

- Le barème est sur 24,5 points et est donné à titre indicatif.
- Dans le code C++ demandé vous prendrez le temps d'assurer que la compilation fonctionne.
- On vous fournit une archive contenant un projet eclipse CDT par exercice, qu'il faudra modifier. Décompresser cette archive dans votre home, de façon à avoir un dossier `~/exam/` et les sous dossiers `~/exam/exo1/src` ...
- Pour importer ces projets dans Eclipse, le plus facile :"File->Import->General->Existing Projects into Workspace",
- Si vous préférez utiliser un autre IDE ou la ligne de commande, on vous fournit dans chaque répertoire source un Makefile trivial.
- A la fin de la séance, fermez simplement votre session. On ira par script récupérer les fichiers dans ce fameux dossier `~/exam/`. Assurez vous donc de bien suivre ces instructions.

Le sujet est composé d'exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite. Pour la majorité des questions il s'agit de fournir un code compilable correct.

## 1 Indexation de fichiers

On considère le problème suivant: étant donné un ensemble de mots clés et un ensemble de fichiers, on souhaite construire un index, c'est à dire associer à chaque mot clé la liste des fichiers auquel il appartient.

### 1.1 unordered_map, unordered_set (4 points)

**Question 1.** A l'aide des classes de la lib standard, implanter ce comportement dans "exo1". On complétera le "main.cpp" fourni qui est conçu pour être très simple (variables globales précédent les fonctions qui s'en servent). On dispose d'un cadre dans "utils.hh" pour invoquer une fonction sur chaque mot rencontré dans un fichier.

NB: Même si on recommande `std::unordered_map` et `std::unordered_set`, et plus spécifiquement un `unordered_map<string,unordered_set<string>` associant des listes de noms de fichier à chaque mot clé. Cependant on acceptera les réponses moins efficaces utilisant e.g. `vector`.

## 1.2 Multi-threading (4 points)

**Question 2.** Modifier à présent le programme pour le rendre concurrent. On souhaite engendrer un thread pour chaque fichier argument. Bien entendu, il faut protéger les structures de données manipulées contre les accès concurrents. On recommande d'écraser le contenu du fichier "mainMT.cpp" avec le contenu de votre main actuel, puis d'ajouter les éléments de synchronisation utiles et la création des thread (un par fichier argument). La lecture des mots clés restera dans le thread principal. On prendra garde à sortir proprement. A ce stade on impose une solution avec un seul mutex.

## 1.3 Synchronisation fine (4 points)

**Question 3.** (5 points) Modifier à présent le programme pour effectuer une synchronisation plus fine, dans laquelle on se donne un mutex pour chaque mot clé, de manière à pouvoir verrouiller l'ensemble des noms de fichiers associés à un mot clé donné, au lieu de devoir tout protéger ("big fat lock" comme la question précédente). De nouveau on peut commencer par écraser le contenu de "mainMTfin.cpp" avec votre version de la question précédente. Pour cette question, on recommande de revoir vos structures de donnée, on souhaite que la table qui associe l'ensemble des fichiers aux mots clés soit stable pendant que l'on itère les fichiers en concurrence. On recommande de

- définir un `unordered_map<string,int>` "keyIndex" associant à chaque mot clé un indice unique de 0 à (nombre de mots clé -1);
- définir un `vector<mutex>` "mutexes" avec (nombre de mots clé) entrées
- définir un `vector<unordered_set<string>` "files" tel que à l'indice $i$ on trouve les fichiers contenant le mot clé d'indice $i$ dans "keyIndex".
- Mettre à jour le code en conséquence (affichages, fonctions exécutées par les threads...)

. On fera attention à initialiser ces structures dans la phase séquentielle de parse des mots clés, et donc avant de lancer le premier thread.

## 1.4 Parallélisme contraint (4 points)

**Question 4.** Modifier le fichier fourni "mainContraint.cpp" pour que :

- Le programme engendre au total $N$ threads qui vont exécuter la fonction "work".
- A un instant donné, il ne doit pas y avoir plus de $K$ threads actifs
- On prendra garde à sortir proprement

. NB : cet exercice est indépendant des questions précédentes et peut être traité séparément.

## 1.5 Une classe de synchronisation (4 points)

**Question 5.** Ecrire une classe générique "`Result<T>`" munie de :

- un attribut "valeur" de type T
- `bool set(const T & val)` qui met à jour la valeur stockée si elle n'est pas encore positionnée et rend true, ou rend false sinon (et ne fait rien).
- `const T & get()` qui rend la valeur stockée, **en attendant qu'elle soit disponible**.

Cette classe sera utilisée dans un contexte concurrent, ou un thread fera le "set" et un ou plusieurs autres threads attendent d'accéder à la valeur calculée avec "get". Compléter le fichier "Result.cpp" pour définir la classe demandée et dans le main créer deux thread qui interagissent avec.

## 1.6 Question compréhension (4,5 point)

**Question 6.** Répondez dans "questions.txt" aux questions suivantes:

1. Quel est l'intérêt en pratique d'un pool de thread par rapport à l'instanciation d'un thread par tâche à traiter ?
2. Expliquez comment communiquer une donnée particulière entre deux threads. Cette stratégie est elle encore valable dans un contexte multi-processus ?
3. Expliquez quand et de quelle manière sont traités les signaux reçus en fonction du "masque" du processus.