Labo 6 - Producteur-Consommateur pour calcul différé (Hoare)

Auteur·ices: Vitória Oliveira et Samuel Roland

Attributs partagés

```
constexpr static std::size_t NB_COMPUTATION_TYPES
    = static_cast<std ::size_t>(ComputationType::Count);
// Où count est le dernier élément de l'enum ComputationType permettant ainsi
// de récupérer le nombre de types différents.
```

Cette constante nous permet de facilement dimensionner et parcourir nos conteneurs.

```
std::array<std::map<int, Request>, NB_COMPUTATION_TYPES> requests;
std::map<int, std::optional<Result>> results;
```

Ces deux structures représentent notre buffer.
La première sert à stocker les requêtes de calculs qui n'ont pas encore été attribuées à des calculateurs. Chaque type de computation (ComputationType) possède sa propre map indexée par id.
Le deuxième conteneur permet de stocker les résultats des calculs en attente d'être renvoyés au client. Le type **`optional`** nous permet d'allouer notre résultat dans la map avant d'avoir le vrai résultat du calculateur afin d'assurer l'ordre, l'attente et l'annulation des retours des calculs.

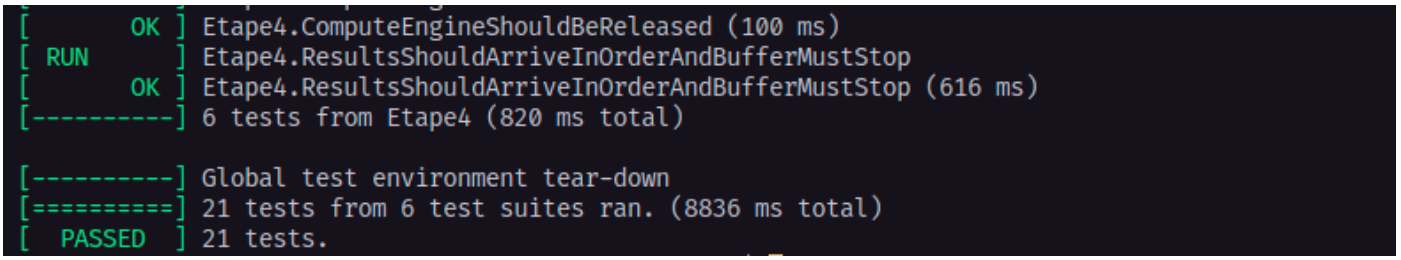
L'utilisation des maps comme file d'attente (au lieu d'une **`std::queue`**) permet de facilement accéder et retirer un élément par son id, mais aussi en utilisant l'id comme clé, cela nous assure que les requêtes et les résultats sont toujours ordonnés et nous pouvons facilement accéder à l'élément avec le plus petit id via l'itérateur **`begin()`**.

```
// Conditions
std::array<std::unique_ptr<Condition>, NB_COMPUTATION_TYPES> notFull;
std::array<std::unique_ptr<Condition>, NB_COMPUTATION_TYPES> notEmpty;
Condition nextResultReady;
```

1. **`notFull`**: Permet de faire attendre les clients lorsque la file des requêtes de calculs est pleine (c'est-à-dire quand elle a atteint **`MAX_TOLERATED_QUEUE_SIZE`**).
2. **`notEmpty`**: Permet de faire attendre les calculateurs lorsque la file de requêtes de calculs est vide.
3. **`nextResultReady`**: Permet de relâcher le thread attendant sur le prochain résultat.

Tests fournis

Les tests fournis nous ont été très utiles et passent tous. Nous avons remarqué certains cas non testés que nous avons vérifié manuellement. (D'ailleurs merci beaucoup pour ces tests, c'est très pratique!)



Étape 1

1. **`int requestComputation(Computation c)`**
    - Elle bloque si le conteneur du type de computation demandé est plein.
    - Cette méthode crée une requête de calcul et l'insère dans **`requests`** du type de computation passé en paramètre. En plus, elle place à la fin de la map **`results`** un nouvel résultat nul.
    - Elle permet de réveiller un thread qui attend sur la condition **`notEmpty`** du type de computation donné.
    - Finalement, elle retourne l'id assigné à la demande de calcul.

2. **`Request getWork(ComputationType computationType)`**
    - Cette méthode bloque lorsque **`requests`** est vide. Sinon, elle accède au premier élément de celui-ci et récupère la requête qu'y est stockée. Une fois ceci effectué, la requête est retirée du conteneur.
    - Elle permet de réveiller un thread qui attend sur la condition **`notFull`** du type de computation de la requête demandée.

Tests
En plus des tests fournis, nous avons vérifié le bon fonctionnement de ces méthodes en effectuant des tests manuels via l'interface et en consultant les logs. Nous avons initié plusieurs demandes de calcul sur les différents calculateurs et nous avons vérifié que les identifiants étaient attribués dans le bon ordre. Nous nous sommes assurés que les demandes de calcul étaient également traitées dans le bon ordre, que les requêtes étaient récupérées par les calculateurs appropriés et dans le bon ordre. De plus, nous avons confirmé que chaque calculateur obtenait une requête de calcul lorsqu'il était effectivement disponible

Étape 2

1. **`Result getNextResult()`**
    - Cette méthode bloque si **`results`** est vide ou si le premier **`Result`** dans ce conteneur n'a pas encore de valeur (c'est-à-dire le calcul n'est pas encore fini et retourné). Autrement, elle copie le

premier résultat qu'y est stocké, l'efface du conteneur et le retourne.
   - Elle permet de réveiller un thread qui attend sur le prochain résultat à retourner.

2. `` `void provideResult(Result result)` ``
   - Cette méthode insère le résultat passé en paramètre dans `` `results` `` à la position de l'id de `` `result` ``.
   - Elle permet de réveiller un thread qui attend sur le prochain résultat à retourner.

Tests
Autre que les tests déjà fournis, nous avons effectués plusieurs tests via l'interface. Nous avons vérifié que les résultats étaient retournés dans le bon ordre et le cas où `` `getNextResult()` `` est censé bloquer.

Étape 3

1. `` `void abortComputation(int id)` ``: pour annuler un calcul et gérer les 4 cas possibles de la donnée, il suffit de
   - supprimer le request dans `` `requests` `` (on ne sait pas de quel type de calcul il vient donc il faut regarder dans chaque map). Au passage on signal `` `notFull` `` comme la map n'est plus pleine.
   - supprimer l'objet result (peu importe s'il est rempli ou non) de `` `results` ``
   - Si le prochain résultat est disponible juste après sa suppression, il faut signaler `` `nextResultReady` `` (ex: calcul 1 en cours, calcul 2 terminé, calcul 1 annulé, le résultat 2 peut être retourné par `` `getNextResult()` ``).
   - Si un élément ne se trouve pas dans une map car le job a déjà été annulé ou n'existe pas sous cette id, il n'y aura aucun impact (pas de suppression, ni de signal).

2. `` `bool continueWork(int id)` ``
   - Un engine peut continuer de travailler si le calcul n'a pas été annulé, dans notre cas cela se traduit par la présence d'un élément dans notre map `` `results` ``.

Tests

1. On demande 12 (2 tout de suite pris puis 10 autres) calculs de type A très vite pour remplir la map, on en relance 2 directement qu'ils sont mis en attente (pas directement affichés), on en annule 3 pour voir si les 2 en attentes sont libérés et si on arrive encore à y remettre 1 dernier sans attente.
2. Annuler un calcul déjà annulé il n'y a pas d'impact
3. Si on lance un calcul de type A puis de type B, qu'on attend que le calcul B soit disponible, qu'on annule le A, le résultat B est directement retourné.

Étape 4

Attributs concernés:

```
bool stopped = false;
void stopExecutionIfEndOfService(Condition &cond);
```

Nous avons l'attribut `**stopped**` pour définir que le système doit s'arrêter. Nous avons adapté `**continueWork()**` pour retourner `**false**` si `**stopped == true**`.

Nous faisons `**stopped = true;**` puis nous signalons toutes les variables de conditions afin de réveiller tous les threads endormis. Un seul thread est réveillé à chaque signal, mais comme il y a toujours un signal un peu après les `**wait()**`, les réveils se font en cascade.

Dans chacun de nos `**while**` autour des `**wait()**` sur nos variables de conditions, nous avons ajouté `**!stopped && ...**` afin d'empêcher que de nouveaux threads se mettent en attente durant l'arrêt, que les threads réveillés ne se rendorment pas à cause des autres conditions du while.

Ensuite, nous avons dû répéter le modèle suivant juste après le `**while**`, que nous avons finalement extrait dans `**stopExecutionIfEndOfService()**`

```
if (stopped) {
    signal(cond);
    monitorOut();
    throwStopException();
}
```

A nouveau, nous utilisons le réveil en cascade pour que tous les threads soient bien réveillés.

Tests

1. Les calculateurs s'arrêtent après le prochain bloc de temps (à l'appel de `**continueWork()**`)
2. Rien ne se passe si on lance des calculs une fois arrêté
3. Nous avons testé de logger un peu certaines parties pour savoir si les threads se réveillent bien mais sinon nous ne voyons pas de manière de tester qu'ils sont bien réveillés comme l'UI ne le montre pas.