Introduction à la Concurrence : Programmation Asynchrone

1 Introduction

L'objectif de ce TP est de vous familiariser avec un nouveau paradigme de programmation : la programmation concurrente. Ce paradigme de programmation regroupe plusieurs approches (programmation distribuée avec passage de message, parallèle avec mémoire partagée, etc.). Dans ce TP introductif, nous allons étudier la *programmation asynchrone*, qui est l'approche la plus simple à mettre en œuvre.

L'objectif de ce TP sera d'écrire un simulateur d'ordinateur, avec ses différentes unités (e.g. processeur, mémoire, carte réseau, etc.). L'objectif sera de simuler l'exécution simultanée de différentes tâches sur ces différentes unités.

1.1 Modèle d'Ordinateur

Pour écrire notre simulateur, nous allons modéliser un ordinateur comme un *ensemble d'unités* devant réaliser un *ensemble de tâche*.

Les tâches. Chaque tâche est caractérisée par une durée d'exécution, et une unité sur laquelle la tâche doit s'exécuter. Par exemple, si on doit faire un calcul sur processeur qui prend 5 secondes, on aura une tâche `T`, qui sera associée à l'unité `CPU`, et dont la durée est de 5 secondes.

De plus, après son exécution, chaque tâche peut continuer avec une liste de tâche suivantes (qu'on appelle des *continuations*). Par exemple, la tâche `T` mentionnée plus haut peut, une fois terminée, continuer avec un accès mémoire (tâche `M`) et l'utilisation de la carte réseau (tâche `R`). Dans ce cas là, les continuations de `T` sont exprimées sous la forme de la liste `[M, R]`.

On distingue les continuations qui peuvent exécuter simultanément des continuations qui doivent s'exécuter séquentiellement. Dans l'exemple qui précède `M` et `R` sont indépendantes et peuvent s'exécuter en même temps : les deux sont lancées à la fin de la tâche `T`.

Si au contraire, on voulait que l'utilisation de la carte réseau attende la fin de l'accès mémoire (e.g. si on veut envoyer les données lues en mémoire), alors la continuation de `T` serait `[M]` uniquement, et la continuation de `M` serait `[R]`.

Par soucis de simplicité, chaque tâche peut créer des tâches après son exécution, mais on ne s'autorise pas à créer des tâches qui dépendent de plusieurs tâches (e.g. on créé la tâche `T` quand `T1` *et* `T2` sont finies). Dit autrement, le graphe de dépendance des tâches forme un arbre (ou une forêt si on a plusieurs tâches initiales), et non un graphe orienté acyclique.

Ainsi, les tâches sont modélisées par une classe `Tache` ayant pour attributs les champs :

- `nom_unite` : le nom de l'unité qui doit exécuter la tâche
- `duree` : la durée nécessaire à l'accomplissement de la tâche
- `continuation` : la liste de tâche qui suivent la tâche courante

Les unités. Les tâches s'exécutent sur des unités d'exécution. Chaque unité d'exécution est caractérisée par un nom. Par exemple, le nom du processeur peut-être la chaîne de caractères `"CPU"`. De plus, chaque unité d'exécution a une liste de tâches à accomplir. Dès que possible, chaque unité prend une tâche dans la liste des tâches qu'elle doit exécuter, et l'exécute. L'exécution d'une tâche `T` est simulée par l'attente d'un délai supérieur ou égal à la durée de la tâche `T`. Une fois l'exécution finie, la tâche `T` est supprimée de la liste de tâches à exécuter, et les continuations de `T` sont rajoutées dans les listes de tâches des unités d'exécution idoines.

Ainsi, les unités sont modélisées par une classe `UniteExecution` ayant pour attributs les champs :

- `nom` : le nom de l'unité
- `taches` : la liste de tâches à exécuter
- `ordi` : une référence vers l'ordinateur auquel appartient l'unité (voir ci-dessous)

L'ordinateur. Nous introduisons finalement l'ordinateur en lui-même, en tant qu'ensemble d'unités d'exécutions. Son rôle sera uniquement de créer les différentes unités d'exécution au démarrage et de repartir auprès des unités d'exécutions adéquates les tâches créées au long de l'exécution.

Code fourni. Le code fourni contient la déclaration des classes ainsi que quelques méthodes utilitaires. Vous aurez à écrire les fonctions (asynchrones) permettant l'exécution du modèle.

2 Exécution de tâches

Pour commencer, nous allons nous pencher sur l'implémentation de l'exécution des tâches.

Exécution d'une tâche

Écrivez une méthode asynchrone `executer(self)` de la classe `Tache`. Cette méthode doit attendre `self.duree` secondes (minimum), puis renvoyer la liste de continuations de `self`.

Vous pouvez vérifier l'implémentation de votre code en utilisant python en mode interactif, en appelant `asyncio.run(ma_tache.executer())` (où `ma_tache` est une tâche à exécuter). L'appel doit attendre le temps correspondant avant de terminer.

3 Gestion naïve d'une unité d'exécution

Nous allons maintenant programmer l'exécution d'une unité. Dans un premier temps, nous allons ignorer les continuations : nous partons du principe que toutes les tâches sont présente dans la file d'attente de l'unité dès le démarrage.

Exécution naïve d'une unité d'exécution

Écrivez une méthode asynchrone `demarrer(self)` de la classe `UniteExecution`. Cette méthode doit consommer les tâches de la liste de tâche de l'unité. Pour chaque tâche de cette liste, il faut simuler l'exécution (c.f. la méthode `executer()` écrite plus haut).

⚠ La méthode `executer()` de la classe `Tache` est asynchrone. Veuillez vérifier code votre implémentation de `demarrer` *attend* effectivement la résolution de l'exécution, à l'aide de `await`.

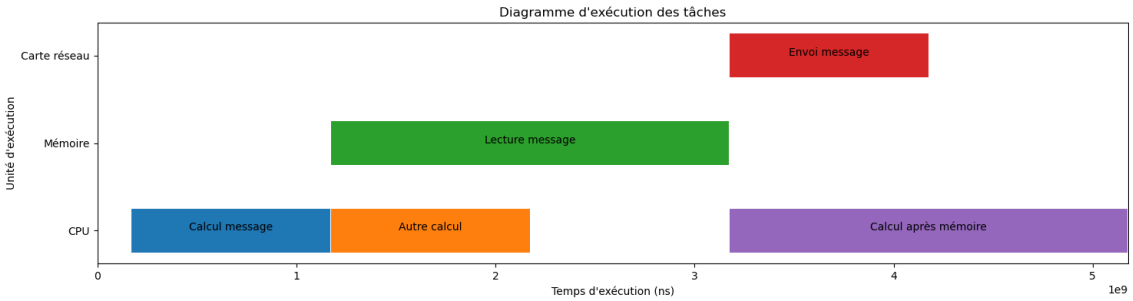

Figure 1: Exemple de diagramme d'exécution, une fois l'exercice terminé.

Pour vous aider, vous pouvez utiliser la méthode fournie `UniteExecution.log(self, msg)` qui permet d'afficher un message de log.

Vous pouvez vérifier l'implémentation de votre code en utilisant python en mode interactif. Vérifiez par exemple qu'une unité d'exécution avec deux tâches les exécute bien séquentiellement.

4 Concurrence de plusieurs unités d'exécution

Nous allons maintenant compléter la classe `Ordinateur` pour avoir plusieurs unités d'exécutions concurrentes.

Démarrage de l'ordinateur

Écrivez une méthode asynchrone `demarrer(self)` de la classe `Ordinateur`. Cette méthode doit démarrer toutes les unités d'exécutions de l'ordinateur. Puisque les unités d'exécutions doivent s'exécuter concurremment, on ne peut pas en exécuter une, puis la suivante quand la première termine.

Pour faire cela, créez une tâche `asyncio` par unité. Cette tache exécutera l'appel à `demarrer()` de la tâche.

⚠ Consultez la page de documentation de `create_task`[a].

Une fois les unités d'exécutions démarrées, l'ordinateur doit *attendre* que les tâches terminent avant de terminer la méthode.

[a] `https://docs.python.org/3/library/asyncio-task.html#creating-tasks`

La classe `Ordinateur` fourni des fonctions permettant d'enregistrer et de visualiser les différentes tâches exécutées. Appelez les méthodes `_init_log` et `_fin_log` respectivement au début et à la fin de votre méthode `Ordinateur.demarrer(self)`. En faisant cela, vous pouvez rajouter des appels à `self.ordi.log_tache(...)` dans votre méthode `UniteExecution.demarrer()` afin de vérifier le comportement correct de votre programme.

Vous pouvez vérifier le comportement de votre implémentation en testant un ordinateur à deux unités, qui doivent s'exécuter concurremment.

5 Ajout des continuations

Il nous reste plus qu'à gérer les continuations des tâches. Dans un premier temps, nous allons ajouter l'infrastructure nécessaire à l'ajout de tâches dans les listes des unités d'exécution; puis, dans un second temps, nous modifierons l'implémentation des unités d'exécutions pour prendre en compte d'éventuels ajouts.

5.1 Ajout de tâches dans les listes

Dans un premier temps, nous allons ajouter une méthode aux `UniteExecutions` permettant d'ajouter une tache.

Ajouter des tâches à une unité d'exécution

Ajoutez une méthode `ajouter_tache(self, tache: Tache)` à la classe `UniteExecution`. Cette méthode doit ajouter `tache` à la fin de la liste d'attente de l'unité.

Enfin, nous allons modifier la classe `Ordinateur` en elle-même.

Ajouter des tâches à l'ordinateur

Ajoutez une méthode `ajouter_tache(self, tache: Tache)` à la classe `Ordinateur`. Cette méthode doit récupérer le nom de l'unité d'exécution sur laquelle doit s'exécuter `tache` (utilisez `tache.unite`), puis ajoutez cette `tache` à l'unité à l'aide de la méthode `ajouter_tache` de la classe `UniteExecution` écrite plus haut.

Maintenant que les méthodes d'ajout sont écrites, nous pouvons gérer l'ajout des continuations.

Gestion des continuations

Modifiez la méthode `demarrer()` de la classe `UniteExecution` afin d'ajouter les continuations dans les files respectives des unités de l'ordinateur.

Pour rappel, la méthode `exécuter` d'une tâche renvoie la liste des continuations.

5.2 Modification de la logique des unités d'exécution.

À ce stade, vous devriez pouvoir vérifier le comportement des méthodes d'ajout de tâches écrites. Cependant, ajouter des tâches à la volée rend sans doute incorrect l'implémentation naïve des unités d'exécution : une unité ne peut pas simplement se contenter de consommer toutes les tâches quelle a, puis de terminer. En effet, il se peut qu'une unité commence en étant inactive (donc en ayant une file vide), puis que des tâches soient ajoutées lors de l'exécution. Nous devons donc modifier l'implémentation pour prendre en compte cela.

Exécution infinie des unités d'exécution

Modifiez la méthode `demarrer` de la classe `UniteExecution`. La nouvelle version doit essayer en boucle d'exécuter la première tâche de la file d'attente. Si la liste est vide, alors l'unité boucle.

⚠ Pour éviter qu'une unité monopolise le fil d'exécution asynchrone, pensez à rendre la main. Une manière simple de faire cela est d'attendre 0 secondes : `asyncio.sleep(0)`[a].

[a] `https://docs.python.org/fr/3/library/asyncio-task.html#asyncio.sleep`

Testez votre programme. Normalement, vous devriez pouvoir constater l'exécution attendue des différentes tâches. Le seul défaut est que, les unités d'exécution étant des boucles infinies, le programme ne s'arrêtent jamais.

Gestion de la fin de l'exécution

La condition d'arrêt des unités d'exécution est qu'il n'y a plus *aucune* tâche à exécuter dans aucune file. Rajoutez, dans la classe `Ordinateur`, un compteur de tâches. Ce compteur doit-être incrémenté à chaque fois qu'une continuation est créée (e.g. dans la méthode `ajouter_tache` de la classe `Ordinateur`), et décrémenté à chaque fois qu'une tâche est finie (e.g. ajoutez une méthode `terminer_tache` à la classe `Ordinateur`, et appelez cette méthode à la fin du corps de la boucle d'exécution des unités).

Utilisez ce compteur pour sortir de la boucle d'exécution des unités d'exécution.

Appelez la fonction `exemple()` pour simuler un petit exemple d'ordinateur.