# Parallélisation maximale automatique

Projet pratique en Systèmes d'exploitation

sergiu.ivanov@univ-evry.fr

## 1 Objectif

Développer une libraire en Python pour automatiser la parallélisation maximale de systèmes de tâches. L'utilisateur doit pouvoir spécifier des tâches quelconques, interagissant à travers un ensemble arbitraire de variables, et pouvoir :

1. obtenir le système de tâches de parallélisme maximal réunissant les tâches en entrée,
2. exécuter le système de tâches en parallèle, tout en respectant les contraintes du parallélisme maximal.

## 2 Réalisation

Cette section précise les exigences du projet et donne quelques approches possibles de la solution. Les approches suggérées ne sont pas obligatoires ; vous devez cependant **respecter les exigences** définies dans l'objectif : votre librairie doit permettre la construction du *système de tâches* de parallélisme maximal et son *exécution*.

### 2.1 Tâches

Votre librairie peut être contenue dans un seul fichier Python, par exemple `maxpar.py`. Elle doit proposer à l'utilisateur de définir des tâches. Une façon de le faire est de déclarer la classe suivante :

```
class Task:
    name = ""
    reads = []
    writes = []
    run = None
```

Les significations des champs sont les suivantes :

— `name` : le nom de la tâche, unique dans un système de tâche donné ;
— `reads` : le domaine de lecture de la tâche ;
— `writes` : le domaine d'écriture de la tâche ;
— `run` : la fonction qui déterminera le comportement de la tâche.

Une utilisation de cette classe peut ressembler au code suivant :

```
X = None
Y = None
Z = None

def runT1():
    global X
    X = 1
```

```
def runT2():
    global Y
    Y = 2

def runTsomme():
    global X, Y, Z
    Z = X + Y

t1 = Task()
t1.name = "T1"
t1.writes = ["X"]
t1.run = runT1

t2 = Task()
t2.name = "T2"
t2.writes = ["X"]
t2.run = runT2

tSomme = Task()
tSomme.name = "somme"
tSomme.reads = ["X", "Y"]
tSomme.writes = ["Z"]
tSomme.run = runTsomme
```

Ce code définit 3 objets de la classe `Task` : `t1`, `t2` et `tSomme`, et leur associe les noms, les domaines de lecture et d'écriture, ainsi que les fonctions donnant leurs comportements (`runT1`, `runT2` et `runTsomme`). Pour exécuter ces trois tâches à la main, on peut utiliser le code suivant :

```
t1.run()
t2.run()
tSomme.run()
print(X)
print(Y)
print(Z)
```

## 2.2 Systèmes de tâches

Un ensemble d'objets de la classe `Task` définie dans la section précédente *ne forme pas* un système de tâches, car aucune information sur la précédence n'est disponible directement dans un tel ensemble. Si vous suivez cette approche, votre librairie doit construire le système de tâches (qui sera de parallèlisme maximal) en deux étapes : identification des tâches interférentes et établissement de la relation de précédence.

L'identification des paires de tâches interférentes peut se faire par l'application des conditions de Bernstein. Cependant, si deux tâches sont interférentes, il n'est pas possible de savoir en utilisant ces conditions *dans quel ordre* ces tâches doivent être exécutées : en effet, l'ordre dépendra des préférences de l'utilisateur ou de l'utilisatrice.

La procédure de construction d'un système de tâches sera donc paramétrée par les deux objets suivants :

1. une liste de tâches (objets de la classe `Task`),
2. un dictionnaire [1] donnant les préférences de précédence.

Le dictionnaire des préférences de précédence contiendra, pour chaque nom de tâche, les noms des tâches par lesquelles elle doit être précédée *si* un ordonnancement s'impose.

1. Les dictionnaires, aussi connus sous le nom de table, hash table, map ou hash map dans d'autres langages de programmation, sont des tables associatives, e.g. `https://realpython.com/python-dicts/`.

Deux tâches non interférentes doivent être parallélisées quelles que soient les préférences de précédence.

Supposons par exemple que vous réalisiez la construction d'un système de tâches dans le constructeur [2] de la classe `TaskSystem`. Dans le cas des trois tâches `t1`, `t2` et `tSomme` définies dans la section précédente, cette construction peut s'écrire de la façon suivante :

```
s1 = TaskSystem([t1, t2, tSomme], {"T1": [], "T2": ["T1"], "somme": ["T1", "T2"]})
```

Le dictionnaire des préférences précise que la tâche dont le nom est `"T2"` doit s'exécuter après celle qui a le nom `"T1"`, dans le cas où ces deux tâches doivent être ordonnées. De la même façon, la tâche dont le nom est `"somme"` doit s'exécuter après les deux autres tâches, si jamais un ordonnancement est nécessaire.

L'objet `s1` issu de cette procédure de construction doit être un système de tâches de parallélisme maximal. La classe `TaskSystem` doit réaliser au moins les méthodes suivantes :

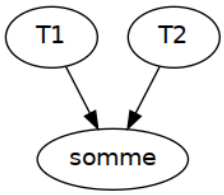
- `getDependencies(nomTache)` : pour un nom de tâche donné, renvoyer la liste des noms des tâches qui doivent s'exécuter avant la tâche `nomTache`,
- `run()` : exécuter les tâches du système en parallélisant celles qui peuvent être parallélisées selon la spécification du parallélisme maximal.

### 2.3 Bonus 1 : validation des entrées

Les entrées fournies à la procédure de construction du système de tâches peuvent avoir des défauts : les noms des tâches peuvent être dupliqués, le dictionnaire des préférences de précédence peut contenir des noms de tâches inexistantes, peut ne pas être suffisamment complet pour le problème de minimisation donné, etc. Réalisez un ensemble de vérifications de validité des entrées, en affichant des messages d'erreur détaillés.

### 2.4 Bonus 2 : affichage du système de parallélisme maximal

Rajoutez à votre librairie une fonction qui permettrait d'afficher graphiquement le graphe de précédence du système de parallélisme maximal construit. Si, par exemple, vous réalisez cette fonctionnalité sous forme de méthode `draw` de la classe `TaskSystem`, l'appel `s1.draw()` peut produire l'image suivante :




### 2.5 Parenthèse : les variables globales

Ce projet met en avant l'utilisation des variables globales : en effet, la communication entre les tâches se fait uniquement par ce biais. L'utilisation des variables globables est généralement une très mauvaise idée, parce qu'elle engendre des *interférences*. Cependant, dans certaines situations l'usage de ressources globales est imposé par des contraintes externes et ne peut être évité. Cette consigne se focalise sur ce cas de figure précisément.

2. Le constructeur est la méthode `__init__` d'une classe et décrit les actions qui doivent être exécutées à la création de ses objets. Vous pouvez voir un exemple ici : `https://www.w3schools.com/python/gloss_python_class_init.asp`.

# 3 Organisation du travail et évaluation

Le projet sera évalué de deux façons : sur le code soumis et sur l'exposé de 10 minutes. Le travail en *binôme ou trinôme* est conseillé. Le travail en monôme est déconseillé mais accepté. Le travail dans des groupes de $>$ **3 personnes est interdit**.

## 3.1 Évaluation du code rendu

Le code rendu sera évalué par votre chargé·e de TD selon la grille suivante :

— construction du système de parallélisme maximal : *7 points*,
— exécution du système de parallélisme maximal : *7 points*,
— bonus 1 : *3 points*,
— bonus 2 : *3 points*.

Votre code doit être *soigneusement commenté*. Sans aller jusqu'à expliquer chaque ligne de code individuellement, vos commentaires doivent rendre compte de votre compréhension : pourquoi faites-vous telle ou telle action, pourquoi la faire de cette façon et non d'une autre, etc.

## 3.2 Évaluation de l'exposé

À la dernière séance de TD, vous ferez un exposé de **10 minutes**, lors duquel vous expliquerez votre solution, les difficultés que vous aurez rencontrées, les façon de les résoudre, etc. Votre compréhension de l'utilité de la parallélisation automatique sera évaluée également.

La limite de temps est stricte. Chaque minute prise au-delà des 10 imparties sera pénalisée d'un **retrait d'un point** par minute à la note sur 20.

Dans le cas d'un écart flagrant d'investissement, l'enseignant·e peut donner des notes différentes aux membres d'un groupe.