VI - TP 3 Ordonnanceur

L'objectif de ce TP est d'établir de manière automatique des **chronogrammes**, étant donné une liste de **processus** et un **algorithme d'ordonnancement**. Ceci nous permettra de calculer ensuite des **mesures** des différents algorithmes d'ordonnancement afin de pouvoir les comparer entre eux.

1 Représentation des processus

Les **processus** sont représentés à l'aide d'instances de la classe `Processus`. Ils sont munis de quatre attributs :

- leur `pid` (pour *process identifier*, c'est un entier unique au processus en question) ;
- leur `duree` totale d'exécution (exprimée en nombres de cycles du processeur)
- leur instant d'`arrivee` (exprimé en nombre de cycles du processeur après l'intant initial $t = 0$)
- leur `duree_restante` (exprimée en nombre de cycles du processeur) qui représente le temps d'exécution nécessaire avant la fin du processus.

Question 1. **1.** Écrire une classe `Processus` permettant d'instancier des variables munies de ces attribus.

2. Instancier des variables `p1`, `p2`, `p3`, `p4` et `p5` de représentant les processus dont on donne la description dans le tableau ci-dessous :

Processus	Durée d'exécution	Instant d'arrivée
P1	10	0
P2	4	3
P3	8	2

Code python

```
class Processus:
    """Représente un processus"""

```

1.1 Représentation d'un processus

Écrire une méthode `__repr__` de la classe `Processus` qui affiche une chaine de caractère représentant un processus à l'aide de son `pid` selon le schéma de l'énoncé.

Code python

```
def __repr__(self):
    """ Processus -> str
    Renvoie une chaine de caractère représentant le processus self """
    pass
```

Code python

```
print(Processus(123, 0, 0))
print(p1, p2, p3)
```

Résultat

```
<P123>
<P1> <P2> <P3>
```

2 Chronogramme

On représente un **chronogramme** à l'aide d'une instance de la classe `Chronogramme`. Une telle instance permet d'obtenir le temps écoulé `t` (exprimé en nombre de cycles du processeur) depuis le démarrage du premier processus et est munie d'un attribut `table` de type `list` qui stocke la suite des processuus ayant été exécutés par le processeur à chacun de ses cycles jusqu'à l'instant `t`. On remarquera que `t` est donc égal au nombre d'éléments de `table`.

Par exemple, si `p1`, `p2`, et `p3` sont trois objets de type `Processus` et `table` est la liste `[p1, p1, p2, p3]`, cela veut dire que le processus `p1` a été exécuté pendant deux cycles du processeur, puis que `p2` et `p3` ont tous les deux été exécutés pendant un cycle. Ainsi, 4 cycles du processeur se sont écoulés depuis l'instant initial.

Question 2. Écrire une classe `Chronogramme` permettant d'instancier une variable munie de cet attribut.

Code python

```python
class Chronogramme:
    """Représente un chronogramme"""

```

2.1 Méthode `temps_ecoule`

Écrire une méthode `temps_ecoule` qui renvoie le nombre de cycles écoulés depuis l'instant initial. Pourquoi est-il incorrect d'utiliser une instruction du type `c.table = [1, 1, 2, 3]` ?

Code python

```python
def temps_ecoule(self):
    """ Chronogramme -> int
    Renvoie le nombre de cycles écoulés depuis l'instant initial """
    pass
```

Code python

```python
c = Chronogramme()
c.table = [1, 1, 2, 3] # incorrect, utilisé uniquement à des fins de test
print(c.temps_ecoule())
```

Résultat

```
4
```

3 Ordonnanceur FIFO

On s'intéresse dans cette partie à un ordonnanceur de type `FIFO` préemptif : celui-ci suit la logique du premier arrivé, premier servi. Ainsi, les différents processus sont exécutés suivant leur ordre d'arrivée. L'ordonnanceur interrompt les processus à intervalles réguliers. Si le dernier processus exécuté n'est pas terminé, alors il est à nouveau exécuté. On modélise un tel ordonnanceur à l'aide d'une classe `OrdonnanceurFIFO`.

Celui-ci sera muni des attributs :

- `chrono` : un chronogramme qui sera mis à jour à chaque cycle du processeur. Cet attribut sera initialisé avec un chronogramme vide.
- `file_processus` : une file contenant les processus à exécuter. Cet attribut sera initialisé par un objet de type `File`, vide.

- `quantum` : le nombre de cycles du processeur pendant lesquels les processus s'exécutent avant d'être interrompus par le système. À chaque interruption, l'ordonnanceur sélectionne un processus pour exécution. Cet attribut sera initialisé avec une valeur par défaut de `1`, qui pourra être modifiée en passant au constructeur de la classe un paramètre optionnel.

Question 3. Écrire une classe `OrdonnaceurFIFO` permettant d'instancier une variable munie de ces attributs. On importera pour cela la classe `File` depuis le module `ds.py`. Les objets de type `File` sont munis d'une méthode `examine` renvoie l'élément présent au début de la file *sans le supprimer* de la file.

Code python
```python
class OrdonnanceurFIFO:
    """Représente un ordonnanceur"""

```

3.1 Ajout des processus

L'ordonnanceur FIFO exécute les processus dans leur ordre d'arrivée. On souhaite écrire une méthode `ajoute_processus` de la classe `OrdonnanceurFIFO` qui prend en argument une **liste** de processus `p_list` et qui les ajoute à la file `self.file_processus` de l'ordonnanceur `self` par ordre croissant d'instant d'arrivée. Pour cela, tant que `p_list` n'est pas vide, on cherche dans `p_list` le processus dont l'attribut `arrivee` est le plus petit, on le supprime de `p_list` et on l'enfile dans la file des processus de l'ordonnanceur.
On procède en deux étapes, détaillées ci-après.

3.1.1 Sélection du premier processus arrivé

Écrire une fonction `renvoie_et_supprime_premier` qui étant donné une liste (supposée non vide) de processus `p_list` renvoie le processus ayant le plus petit temps d'arrivée parmi les processus de la liste, en le supprimant de celle-ci. On écrira donc un algorithme de recherche de la valeur du minimum ainsi que de son indice d'occurrence, puis et on utilisera une instruction du type `l.pop(i)` (supprime de la liste `l` l'élément d'indice `i` en le renvoyant).

Code python
```python
def renvoie_et_supprime_premier(p_list):
    """ [Processus] -> Processus
    Renvoie et supprime de p_list le processus dont l'attribut arrivee est
    ↪ le plus petit """
    pass
```

Code python
```python
p_list = [p1, p2, p3]
for _ in range(len(p_list)):
    print(renvoie_et_supprime_premier(p_list))
```

Résultat
```
<P1>
<P3>
<P2>
```

3.1.2 Ajout des processus à la file

Écrire une méthode `ajoute_processus` de la classe `OrdonnanceurFIFO` qui prend en argument une **liste** de processus `p_list` et qui les ajoute à la file `self.file_processus` par ordre croissant d'instant d'arrivée.

Code python

```python
def ajoute_processus(self, p_list):
    """ OrdonnanceurFIFO, [Processus] -> Nonetype
    Ajoute à self.file_processus les processus de p_list par ordre
    d'arrivée """
    pass
```

Code python

```python
ordo = OrdonnanceurFIFO()
p_list = [p1, p2, p3]
ordo.ajoute_processus(p_list)
print(ordo.file_processus)
```

Résultat

```
[début] <P1> <P3> <P2> [fin]
```

3.2 Exécution d'un quantum

On souhaite écrire une méthode `exec_quantum` qui simule la sélection puis l'exécution sur le processeur d'un processus pendant `self.quantum` cycles. Dans le cas d'un ordonnanceur FIFO l'algorithme est très simple :

- récupérer le premier processus de la file des processus ;
- *enregistrer* dans le chronogramme l'information de l'exécution du processus ;
- *avancer* l'exécution du processus de `self.quantum` unités de temps ;
- si le processus *est terminé* à l'issue de cette étape, le retirer de la file des processus à exécuter.

3.2.1 Méthode `enregistrer`

Écrire une méthode `enregistrer` de la classe `Chronogramme` qui enregistre dans le chronogramme `self` l'information selon laquelle le processus `p` a été exécuté pendant `q` cycles du processeur. Ainsi, cette méthode ajoute `q` fois le processus `p` la fin du tableau `self.table`.

Code python

```python
def enregistrer(self, p, q):
    """ int, int -> Nonetype
    Enregistre l'information : le processus p est exécuté pendant q cycles
    """
    pass
```

Code python

```python
c = Chronogramme()
c.enregistrer(Processus(1, 0, 2), 2)
c.enregistrer(Processus(2, 2, 3), 3)
print(c.table)
```

Résultat

```
[<P1>, <P1>, <P2>, <P2>, <P2>]
```

3.2.2 Méthode `avancer`

Écrire une méthode `avancer` de la classe `Processus` qui diminue la durée d'exécution restante du processus `self` de `d` si l'instant `t` est supérieur à l'instant d'arrivée du processus.

Code python

```python
def avancer(self, d, t):
    """ Processus, int, int -> Nonetype
    Le processus (re)prend son exécution à la date t pour une durée d """
    pass
```

Code python

```python
p = Processus(42, 5, 24)
print(p.duree_restante)
p.avancer(4, 10)
print(p.duree_restante)
```

Résultat

```
24
20
```

3.2.3 Méthode `est_termine`

Écrire une méthode `est_termine` de la classe `Processus` qui renvoie `True` si et seulement si le processus `self` a terminé son exécution. On effectuera pour cela un test utilisant l'attribut `duree_restante` du processus `self`.

Code python

```python
def est_termine(self):
    """ Processus -> bool
    Détermine si le processus p a terminé son exécution """
    pass
```

Code python

```python
p.avancer(20, 14)
print(p.est_termine())
```

Résultat

```
True
```

3.2.4 Méthode `exec_quantum`

Écrire une méthode `exec_quantum` qui implémente l'algorithme d'ordonnanceur FIFO décrit dans l'énoncé.

Code python

```python
def exec_quantum(self):
    """ OrdonnanceurFIFO -> Nonetype
    Simule l'exécution d'un quantum """
    pass
```

Code python

```python
ordo = OrdonnanceurFIFO()
ordo.ajoute_processus([p1, p2, p3])
ordo.exec_quantum()
print(ordo.chrono.table)
```

Résultat

```
[<P1>]
```

3.3 Représentation d'un chronogramme et conclusion

3.3.1 Méthode `__repr__`

Écrire une méthode `__repr__` de la classe `Chronogramme` qui affiche la chaine de caractère de votre choix qui permet de représenter un chronogramme. On pourra utiliser l'instruction `str(p)` où `p` est un processus pour obtenir une chaine de caractère représentant le processus `p`.

Code python

```python
def __repr__(self):
    """ self -> str
    Renvoie une chaine de caractère représentant le chronogramme self """
    pass
```

3.3.2 Chronogramme complet

Écrire une méthode `exec` de la classe `OrdonnanceurFIFO` qui étant donné un ordonnanceur FIFO `self` simule son exécutition jusqu'à ce que la file des processus à exécuter soit vide. Cette méthode renverra le chronogramme de l'exécution.

Comparer avec les résultats vus en exercice.

Code python

```python
def exec(self):
    """ OrdonnanceurFIFO -> Chronogramme
    Simule l'exécution des processus de la file et en renvoie le
    ↪ chronogramme """
    pass
```

Code python

```python
ordo = OrdonnanceurFIFO()
ordo.ajoute_processus([p1, p2, p3])
print(ordo.exec())
```

Résultat

```
<P1> | <P1> | <P1> | <P1> | <P1> | <P1> | <P1> | <P1> | <P1> | <P3> |
↪ <P3> | <P3> | <P3> | <P3> | <P3> | <P3> | <P3> | <P2> | <P2> | <P2> |
↪ <P2>
```

4 Étude des processus

On souhaite étudier "l'efficacité" d'un algorithme d'ordonnancement. Pour cela, on étudie le temps de séjour, le temps d'attente, et le temps de latence moyen des processus gérés. On rappelle les définitions suivantes :

temps de termaison instant t auquel le processus termine.

temps d'exécution c'est le temps pendant lequel le processus a utilisé le processeur.

temps de séjour c'est la différence entre le temps de terminaison et le temps d'arrivée du processus. C'est la durée pendant laquelle l'algorithme d'ordonnancement doit gérer le processus.

temps d'attente c'est la différence entre le temps d'exécution et le temps de séjour. C'est la durée pendant laquelle le processus a attendu un accès au processeur.

temps de latence c'est la différence entre le début de l'exécution du processus sur le processeur et l'arrivée du processus dans la file des processus en cours d'exécution. C'est la durée pendant lequel le processus "attend" son premier accès au processeur.

4.1 Modification des attributs des `Processus`

Ajouter à la classe `Processus` deux attributs initialisés avec la valeur `-1` :

- l'attribut `debut` : représente l'instant `t` de la première exécution du processus par le processeur. Dans ce contexte, `-1` signifie que le processus n'a pas encore accédé au processeur.
- l'attribut `fin` : représente l'instant `t` de la fin de la dernière exécution du processus par le processeur. Dans ce contexte, `-1` signifie que le processus n'est pas encore terminé.

Code python

```python
def __init__(self, pid, arrivee, duree):
    """ Initialise un processus """
    pass
```

Code python

```python
p1 = Processus(1, 0, 10)
print(p1.debut, p1.fin)
```

Résultat

```
-1 -1
```

4.2 Modification de la méthode `avance`

Modifier la méthode `avancer` de la classe `Processus` : `p.avancer(d, t)` diminue la durée restante d'exécution du processus `p` et met à jour les attributs `debut` et `fin` de la manière suivante :

- s'il s'agit de la première exécution du processus, alors on sauvegarde l'instant `t` de l'exécution du processus ;
- si après diminution de la durée restante d'exécution le processus est terminé : on sauvegarde l'instant `d + t` de fin du processus.

Code python

```python
def avancer(self, d, t):
    """ Processus, int, int -> Nonetype
    Avance le processus """
    pass
```

Code python

```python
p = Processus(5, 4, 10)
p.avancer(12, 6)
print(p.debut, p.fin)
```

Résultat

```
6 18
```

4.3 Mesurer l'exécution d'un processus

Ajouter à la classe `Processus` les méthodes `sejour`, `attente`, `latence` qui renvoient respectivement les temps de séjour, d'attente et de latence du processus `self`, lorsque le processus a terminé son exécution.

Code python

```python
def sejour(self):
    """ Renvoie le temps de séjour de self """
    pass

def execution(self):
    """ Renvoie le temps d'exécution de self """
    pass

def attente(self):
    """ Renvoie le temps d'attente de self """
    pass

def latence(self):
    """ Renvoie le temps de latence de self """
    pass
```

Code python

```python
p1 = Processus(1, 0, 10)
p2 = Processus(2, 3, 4)
p3 = Processus(3, 2, 8)
ordo = OrdonnanceurFIFO()
ordo.ajoute_processus([p1, p2, p3])
ordo.exec()
for p in [p1, p2, p3]:
    print(f"{p} : séjour : {p.sejour()}, attente : {p.attente()}, latence :
    ↪ {p.latence()}")
```

Résultat

```
<P1> : séjour : 10, attente : 0, latence : 0
<P2> : séjour : 19, attente : 15, latence : 15
<P3> : séjour : 16, attente : 8, latence : 8
```

4.4 Statistiques d'un algorithme d'ordonnancement

Question 4. **1.** Déduire des questions précédentes une méthode `statistiques` de la classe `OrdonnanceurFIFO` qui affiche sur la sortie standard les temps de séjour moyen, d'attente moyen et de latence moyen de l'ordonnancement généré par l'ordonnanceur `self` pour les processus ayant été exécutés.

Cette méthode sera toujours appelée après la méthode `exec`. On pourra modifier la méthode `ajoute_processus` de telle sorte qu'en plus de remplir la `file_processus` elle sauvegarde également la liste `liste_processus` des processus exécutés (on pourra pour cela ajouter un attribut à la classe `OrdonnanceurFIFO`). On prendra garde aux effets de bords de la fonction `renvoie_et_supprime_premier`.

2. **a.** Adapter le code de la classe `OrdonnanceurFIFO` pour écrire une classe `OrdonnanceurSJF` représentant un ordonnanceur préemptif appliquant la politique du plus court travail d'abord (Shortest Job First) : à chaque intervalle de temps, l'ordonnanceur choisit dans la file d'attente le processus avec la durée d'exécution restante la plus courte.

b. Comparer les statistiques d'un ordonnanceur FIFO avec celles d'un ordonnanceur SJF.