Exercices - Gestion des processus et ressources

Exercice 1 Cet exercice reprend l'énoncé du DM extrait BAC 2023.
L'objectif est d'implémenter et de tester le code de l'algorithme Round Robin, puis de le modifier en implémentant les algorithmes FCFS et SRT.

PARTIE 1 : Les processus arrivent au même instant t

On considère trois processus soumis à l'ordonnanceur au même instant pour lesquels on donne les informations ci-dessous :

PID	**Durée** (en cycles CPU)	**Ordre d'arrivée**
11	4	1
20	2	2
32	3	3

1. Classe `Processus` : Nous allons utiliser une liste pour simuler la file d'attente des processus et la classe `Processus`.
   Complétez les 3 méthodes de la classe.

```
class Processus:
  def __init__(self, pid, duree):
    self.pid = pid
    self.duree = duree
    # Le nombre de cycle qui restent a faire :
    self.reste_a_faire = duree
    self.etat = "Pret"

  def execute_un_cycle(self):
    """Met a jour le reste a faire apres l'execution d'un cycle.
    """
    ..........................................

  def change_etat(self, nouvel_etat):
    """Change l'etat du processus avec la valeur passee en parametre.
    """
    ..........................................

  def est_termine(self):
    """Renvoie True si le processus est termine, False sinon, en se basant sur
       le    reste a faire.
    """
    ..........................................
```

2. La fonction `tourniquet` ci-dessous implémente l'algorithme décrit dans l'exercice.
   Elle prend en paramètre une liste d'objets `Processus` donnés par ordre d'arrivée et un nombre entier positif correspondant au quantum. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur.

```
#PROGRAMME PRINCIPAL

def tourniquet(liste_attente, quantum):
  ordre_execution = []
  while liste_attente != []:
    # On extrait le premier processus
    processus = liste_attente.pop(0)
    processus.change_etat("En cours d'execution")
    compteur_tourniquet = 0
    while ..................... and .....................:
      ordre_execution.append(...........)
      processus.execute_un_cycle()
      compteur_tourniquet = compteur_tourniquet + 1
    if .....................:
      processus.change_etat("Suspendu")
      liste_attente.append(processus)
    else:
      processus.change_etat(...........)
  return ordre_execution

#TEST
liste_attente = = [Processus(...,...), ..........., .........]
liste_pid = tourniquet(.....,.......)
print(liste_pid)
```

Complétez le code manquant et testez votre fonction avec les 3 processus dont les informations vous sont données en début d'exercice avec un quantum de 1, puis un quantum de 2.
Vérifiez que vous obtenez les résultats suivants :

- Quantum = 1 : `[11, 20, 32, 11, 20, 32, 11, 32, 11]`
- Quantum = 2 : `[11, 11, 20, 20, 32, 32, 11, 11, 32]`

3. Implémentez une fonction `fcfs`. Elle prend en paramètre une liste d'objets `Processus` donnés par ordre d'arrivée. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur selon l'algorithme d'ordonnacement FCFS.
   Testez votre fonction avec les 3 processus dont les informations vous sont données en début d'exercice.
   Vérifiez que vous obtenez le résultat suivant : `[11, 11, 11, 11, 20, 20, 32, 32, 32]`.
4. La fonction `srt` ci-dessous implémente l'algorithme d'ordonnacement SRT (Shortest Remaining Time).
   Elle prend en paramètre une liste d'objets `Processus` donnés par ordre d'arrivée. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur.

```
#PROGRAMME PRINCIPAL

def shortest_processus(liste_attente):
    shortest_p = liste_attente[0]..........
    shortest_i = ..............
    for i in range(..............):
        if liste_attente[i].reste_a_faire < shortest_p:
            shortest_p = ..................
            ........................
    return ..................

def srt(liste_attente):
    ordre_execution = []
    while liste_attente != []:
        processus = liste_attente.pop(.........................)
        processus.change_etat("En cours d'execution")
        ordre_execution.append(..................)
        ....................................
        if not(processus.est_termine()):
            processus.change_etat("Suspendu")
            ..............................
        else:
            processus.change_etat("termine")
    return ...................

#TEST
liste_attente = [Processus(11,4),Processus(20,2) , Processus(32,3)]
liste_pid = ............................
print(liste_pid)
```

Complétez le code manquant et testez votre fonction avec les 3 processus dont les informations vous sont données en début d'exercice.
Vérifiez que vous obtenez les résultats suivants : `[20, 20, 32, 32, 32, 11, 11, 11, 11]`

PARTIE 2 : Les processus arrivent à des instants t différents

Nous allons maintenant tenir compte de la date d'arrivée des processus dans l'ordonnanceur. Nous allons considérer les instants d'arrivées suivants :

- PID 11 : t=0
- PID 20 : t=1
- PID 32 : t=2

1. Modifiez la classe `Processus` pour que l'on puisse tenir des instants d'arrivées des processus.
2. La fonction `srt_tick` ci-dessous implémente l'algorithme d'ordonnacement SRT (Shortest Remaining Time).
   Elle prend en paramètre une liste d'objets `Processus`. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur selon l'algorithme SRT en tenant compte des instants d'arrivées des processus.

```
#PROGRAMME PRINCIPAL

def shortest_processus(liste_attente):
    shortest_p = liste_attente[0]..........
    shortest_i = ..............
    for i in range(..............):
        if liste_attente[i].reste_a_faire < shortest_p:
            shortest_p = ..................
            ........................
    return ..................

def srt_tick(liste_processus):
    ordre_execution = []
    liste_attente=[]
    tick = ............
    while liste_attente != [] or tick == 0:
        for p in liste_processus:
            if ............ == tick:
               ........................
        processus = liste_attente.pop(.....................)
        processus.change_etat("En cours d'execution")
        ..............................................
        ..............................................
        if not(processus.est_termine()):
            processus.change_etat("Suspendu")
            liste_attente.append(processus)
        else:
            processus.change_etat("termine")
        tick = ......................
    return ordre_execution

#TEST
liste_processus = ...................................
l = srt_tick(liste_processus)
print(l)
```

Complétez le code manquant et testez votre fonction avec les 3 processus dont les informations vous sont données en début d'exercice.
Vérifiez que vous obtenez les résultats suivants : `[11, 20, 20, 11, 11, 11, 32, 32, 32]`

3. On ajoute deux nouveaux processus dont les caractéristique sont les suivantes :
   - PID = 46, durée = 1, date arrivée = 1
   - PID = 8, durée = 1, date arrivée = 5

   (a) Déterminez manuellement, sur papier, la liste d'exécution des PID.

   (b) Modifiez votre programme principal et vérifiez que votre programme retourne le résultat attendu.

Exercice 2 Un petit problème ?

Dans un programme de jeu multi-joueur, une variable `nb_pions` représente le nombre de pions disponibles pour tous les joueurs.
Une fonction `prendre_un_pion()` permet de prendre un pion dans le tas commun de pions disponibles, s'il reste au moins un pion évidemment. Nous sommes dans la situation où il ne reste plus qu'un pion dans le tas commun et on suppose que deux joueurs souhaitent prendre le dernier pion.
Ils utilisent la fonction `prendre_un_pion()` , ce qui conduit à la création de deux processus `p1` et `p2` , chacun correspondant à un joueur.
Avec Python, on peut utiliser le module `multiprocessing` pour créer des processus. Le programme Python `pions.py` suivant permet de réaliser la situation de jeu décrite :

```
from multiprocessing import Process, Value
import time
def prendre_un_pion(nombre):
  if nombre.value >= 1:
    time.sleep(0.001) # pour simuler un traitement avec des calculs
    temp = nombre.value
    nombre.value = temp - 1 # on decremente le nombre de pions

if __name__ == '__main__':
  # creation de la variable partagee initialisee a 1
  nb_pions = Value('i', 1)
  # on cree deux processus
  p1 = Process(target=prendre_un_pion, args=[nb_pions])
  p2 = Process(target=prendre_un_pion, args=[nb_pions])

  # on demarre les deux processus
  p1.start()
  p2.start()
  # on attend la fin des deux processus
  p1.join()
  p2.join()
  print("nombre final de pions :", nb_pions.value)
```

1. Copiez ce programme dans votre éditeur Python
2. Exécutez plusieurs fois ce programme python et observez le résultat. Très perturbant non ?
3. Expliquez le comportement étrange de ce programme. Pour cela, vous pouvez ajouter quelques instructions d'affichage pour suivre ce qu'il se passe.
4. Pour éviter ce problème de synchronisation, on va utiliser ce qu'on appelle un **verrou** : un verrou est objet partagé entre plusieurs processus mais qui garantit qu'un seul processus accède à une ressource à un instant donné.

   Concrètement, un verrou peut être acquis par les différents processus, et le premier à faire la demande acquiert le verrou. Si le verrou est détenu par un autre processus, alors tout autre processus souhaitant l'obtenir est bloqué jusqu'à ce qu'il soit libéré.

   Le code suivant permet de corriger le problème vu précédemment.
   Analysez ce code, listez les nouvelles lignes de code et expliquez leur rôle.

```
from multiprocessing import Process, Value, Lock
import time
def prendre_un_pion(v,nombre):
  v.acquire()
  if nombre.value >= 1:
    time.sleep(0.001) # pour simuler un traitement avec des calculs
    temp = nombre.value
    nombre.value = temp - 1 # on decremente le nombre de pions
  v.release()

if __name__ == '__main__':
    # creation de la variable partagee initialisee a 1
    nb_pions = Value('i', 1)
    verrou = Lock()
    # on cree deux processus
    p1 = Process(target=prendre_un_pion, args=[verrou, nb_pions])
    p2 = Process(target=prendre_un_pion, args=[verrou, nb_pions])

    # on demarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)
```

Exercice 3 Un autre problème de synchronisation

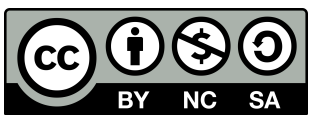
Voici un programme Python dans lequel on crée une variable globale `nombre` qui vaut qui vaut 0 au départ et à laquelle on ajoute 100 quatre fois successivement grâce à la fonction **ajoute_100**.

```
import time
def ajoute_100():
  global nombre
  for i in range(100):
    time.sleep(0.001) # pour simuler un traitement avec des calculs
    nombre = nombre + 1

if __name__ == '__main__':
  nombre = 0
  for i in range(4):
    ajoute_100()
  print("valeur finale :", nombre)
```

1. Copiez et exécutez le programme dans un pour vérifier que la valeur finale de la variable `nombre` est bien égale à 400.
2. On va maintenant supposer qu'une telle variable est partagée par 4 processus, chacun étant chargé d'ajouter 100 à cette variable.
   On a vu dans l'exercice précédent un exemple de problème de synchronisation dans le cas de ressources partagées (une variable partagée `nb_pions`) entre plusieurs processus.

   Inspirez-vous de ce programme pour créer un script permettant :
   - de créer une variable partagée entière appelée `nombre_partage` et initialisée à 0.
   - de créer 4 processus ayant pour rôle d'exécuter une fonction `ajouter_100` à laquelle on passe `nombre_partage` en argument :
     - ⋆ la fonction `ajouter_100(nombre)` doit ajouter 100 à la variable partagée `nombre` en utilisant une boucle `for` qui incrémente 100 fois d'une unité la variable
     - ⋆ on laissera une temporisation pour simuler d'autres calculs
   - de démarrer les 4 processus et d'attendre la n de leur exécution d'afficher la valeur nale de la variable `nombre_partage`
3. Exécutez ce programme et observez que la valeur finale de la variable `nombre_partage` n'est pas (toujours) égale à 400 comme on pourrait s'y attendre. Comment peut-on expliquer cela ?
4. Ajoutez des affichages dans la fonction `ajouter_100` pour observer ce qu'il se passe. Vous afficherez le numéro du processus en cours d'exécution et la valeur de la variable partagée à la fin de chaque tour de boucle.
5. Utilisez ensuite un verrou pour régler ce problème de synchronisation en protégeant la section critique de la fonction `ajouter_100`. Vérifiez que la valeur finale est toujours égale à 400.



Source : Lycée Mounier - Angers