EXERCICE 3 (4 points)

Cet exercice traite du thème architecture matérielle, et plus particulièrement des processus et leur ordonnancement.

1. Avec la commande `ps -aef` on obtient l'affichage suivant :

```
PID   PPID  C STIME TTY          TIME CMD
8600     2  0 17:38 ?        00:00:00 [kworker/u2:0-fl]
8859     2  0 17:40 ?        00:00:00 [kworker/0:1-eve]
8866     2  0 17:40 ?        00:00:00 [kworker/0:10-ev]
8867     2  0 17:40 ?        00:00:00 [kworker/0:11-ev]
8887  6217  0 17:40 pts/0    00:00:00 bash
9562     2  0 17:45 ?        00:00:00 [kworker/u2:1-ev]
9594     2  0 17:45 ?        00:00:00 [kworker/0:0-eve]
9617  8887 21 17:46 pts/0    00:00:06 /usr/lib/firefox/firefox
9657  9617 17 17:46 pts/0    00:00:04 /usr/lib/firefox/firefox -contentproc -childID
9697  9617  4 17:46 pts/0    00:00:01 /usr/lib/firefox/firefox -contentproc -childID
9750  9617  3 17:46 pts/0    00:00:00 /usr/lib/firefox/firefox -contentproc -childID
9794  9617 11 17:46 pts/0    00:00:00 /usr/lib/firefox/firefox -contentproc -childID
9795  9794  0 17:46 pts/0    00:00:00 /usr/lib/firefox/firefox
9802  7441  0 17:46 pts/2    00:00:00 ps -aef
```

On rappelle que : *PID* = Identifiant d'un processus (*Process Identification*)

PPID = Identifiant du processus parent d'un processus (*Parent Process Identification*)

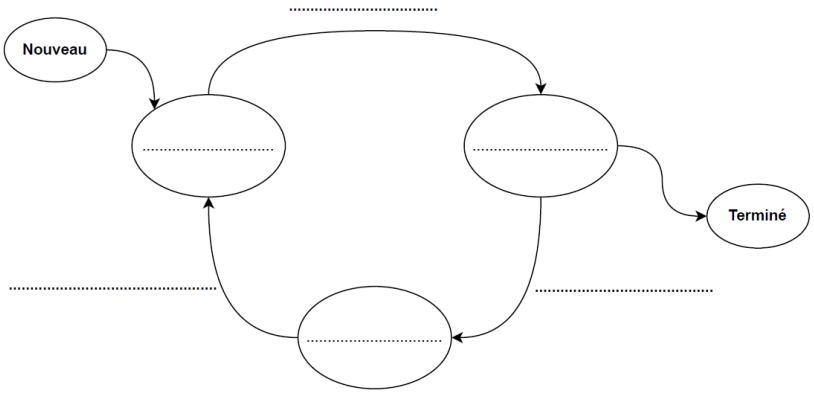
a. Donner sous forme d'un arbre de PID la hiérarchie des processus liés à *`firefox`*.

b. Indiquer la commande qui a lancé le premier processus de *`firefox`*.

c. La commande *`kill`* permet de supprimer un processus à l'aide de son *PID* (par exemple *`kill 8600`*). Indiquer la commande qui permettra de supprimer tous les processus liés à *`firefox`* et uniquement cela.

2.

a. Recopier et compléter le schéma ci-dessous avec les termes suivants concernant l'ordonnancement des processus : *Élu, En attente, Prêt, Blocage, Déblocage, Mise en exécution*



3. On se propose de programmer l'algorithme du premier ordonnanceur. Chaque processus sera représenté par une liste comportant autant d'éléments que de durées (en nombre de cycles). Pour simuler la date de création de chaque processus, on ajoutera en fin de liste de chaque processus autant de chaînes de caractères vides que la valeur de leur date de création.

```
1  p1 = ['1.8','1.7','1.6','1.5','1.4','1.3','1.2','1.1']
2  p2 = ['2.6','2.5','2.4','2.3','2.2','2.1','','']
3  p3 = ['3.2','3.1','','','']
4  p4 = ['4.2','4.1','','','','','','','']
5  liste_proc = [p1, p2, p3, p4]
```

La fonction `choix_processus` est chargée de sélectionner le processus dont le temps restant d'exécution est le plus court parmi les processus en liste d'attente.

a. Recopier sans les commentaires et compléter la fonction `choix_processus` ci-dessous. Le code peut contenir plusieurs lignes.

```
1  def choix_processus(liste_attente):
2      """Renvoie l'indice du processus le plus court parmi
3      ceux présents en liste d'attente liste_attente"""
4      if liste_attente != []:
5          mini = len(liste_attente[0])
6          indice = 0
7          ...
8          return indice
```

Une fonction `scrutation` (non étudiée) est chargée de parcourir la liste `liste_proc` de tous les processus et de renvoyer la liste d'attente des processus en fonction de leur arrivée. À chaque exécution de `scrutation`, les processus présents (sans chaînes de caractères vides en fin de liste) sont ajoutés à la liste d'attente. La fonction supprime pour les autres un élément de chaîne de caractères vides.

b. Recopier et compléter les différentes instructions de la fonction `ordonnancement` pour réaliser le fonctionnement désiré.

```
1  def ordonancement(liste_proc):
2      """Exécute l'algorithme d'ordonnancement
3      liste_proc -- liste des processus
4      Renvoie la liste d'exécution des processus"""
5      execution = []
6      attente = scrutation(liste_proc, [])
7      while attente != []:
8          indice = choix_processus(attente)
9          ... # A FAIRE (plusieurs lignes de code) ...
10         attente = scrutation(liste_proc, attente)
11     return execution
```

On donne dans le tableau ci-dessous quatre processus qui doivent être exécutés par un processeur. Chaque processus a un instant d'arrivée et une durée, donnés en nombre de cycles du processeur.

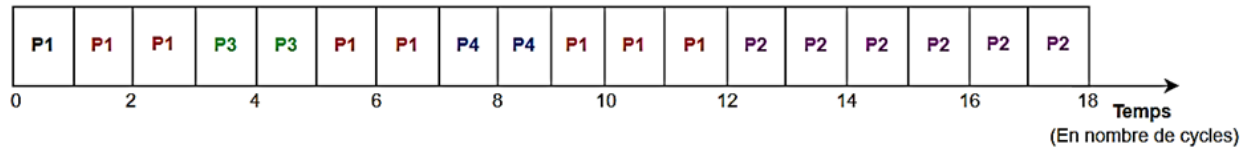
Processus	**P1**	**P2**	**P3**	**P4**
Instant d'arrivée	0	2	3	7
Durée	8	6	2	2

Les processus sont placés dans une file d'attente en fonction de leur instant d'arrivée.

On se propose d'ordonnancer ces quatre processus avec la méthode suivante :

- Parmi les processus présents en liste d'attente, l'ordonnanceur choisit celui dont la durée **restante** est la plus courte ;
- Le processeur exécute un cycle de ce processus puis l'ordonnanceur désigne de nouveau le processus dont la durée restante est la plus courte ;
- En cas d'égalité de temps restant entre plusieurs processus, celui choisi sera celui dont l'instant d'arrivée est le plus ancien ;
- Tout ceci jusqu'à épuisement des processus en liste d'attente.

On donne en exemple ci-dessous, l'ordonnancement des quatre processus de l'exemple précédent suivant l'algorithme ci-dessus.

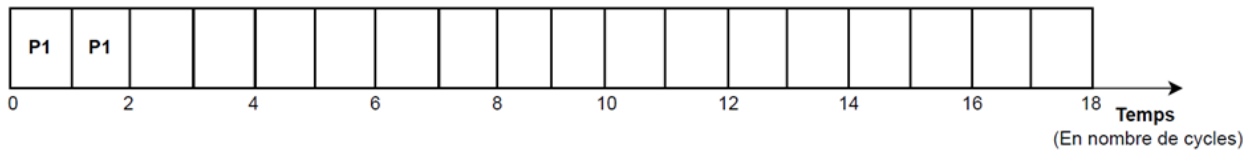


On définit le temps d'exécution d'un processus comme la différence entre son instant de terminaison et son instant d'arrivée.

b. Calculer la moyenne des temps d'exécution des quatre processus.

On se propose de modifier l'ordonnancement des processus. L'algorithme reste identique à celui présenté précédemment mais au lieu d'exécuter un seul cycle, le processeur exécutera à chaque fois deux cycles du processus choisi. En cas d'égalité de temps restant, l'ordonnanceur départagera toujours en fonction de l'instant d'arrivée.

c. Recopier et compléter le schéma ci-dessous donnant le nouvel ordonnancement des quatre processus.



d. Calculer la nouvelle moyenne des temps d'exécution des quatre processus et indiquer si cet ordonnancement est plus performant que le précédent.

EXERCICE 3 (4 points)

Cet exercice porte sur la programmation en Python, la manipulation des chaînes de caractères, les arbres binaires de recherche et le parcours de liste.

1. On rappelle ici quelques notions sur la manipulation des chaînes de caractères en Python.

Une chaîne de caractères se comporte comme un tableau de caractères que l'on ne peut pas modifier.

Par exemple, on a le comportement suivant :

```
>>> une_chaine = 'Bonjour'
>>> une_chaine[3]
'j'
>>> une_chaine[3] = 'z'
 TypeError: 'str' object does not support item assignment
```

On peut aussi utiliser l'opérateur de concaténation +.

```
>>> une_chaine = 'a' + 'b'
>>> une_chaine
'ab'
>>> une_chaine = une_chaine + 'c'
>>> une_chaine
'abc'
```

On définit la fonction `bonjour` par le code suivant :

```
1  def bonjour(nom)
2      return 'Bonjour ' + nom + ' !'
```

a. Donner le résultat de l'exécution de `bonjour('Alan')`.

On exécute le programme suivant :

```
une_chaine='Bonjour'
x = (une_chaine[2] == une_chaine[3])
y = (une_chaine[4] == une_chaine[1])
```

b. Donner le type et les valeurs des variables `x` et `y`.

c. Écrire une fonction `occurrences_lettre(une_chaine,une_lettre)` prenant en paramètres une chaîne `une_chaine` et une lettre `une_lettre` et renvoyant le nombre d'occurrences de `une_lettre` dans `une_chaine`.

2. On rappelle qu'un arbre binaire de recherche est un arbre binaire pour lequel chaque nœud possède une étiquette dont la valeur est supérieure ou égale à toutes les étiquettes des nœuds de son fils gauche et strictement inférieure à celles des nœuds de son fils droit.
Sa taille est son nombre de nœuds ; sa hauteur est le nombre de niveaux qu'il contient.
On rappelle aussi que l'on peut comparer des chaînes de caractères en utilisant l'ordre alphabétique. On a par exemple :

```
>>> 'ab' < 'aa'
False
>>> 'abc' < 'acb'
True
```

On considère la liste de mots `animaux = ['python', 'chameau', 'pingouin', 'renard', 'gnou']`

a. Dessiner un arbre binaire de recherche contenant tous les mots de la liste `animaux` et de hauteur minimale.

b. Dessiner un arbre binaire de recherche contenant tous ces mots et de hauteur maximale.

3. On considère l'implémentation objet suivante d'un arbre binaire de recherche :
On dispose d'une classe `Abr` contenant notamment les méthodes et attributs suivants :

- Si `un_abr` est une instance d'`Abr` alors `un_abr.est_vide()` renvoie `True` si l'arbre est vide et `False` sinon.
- Si `un_abr` est une instance d'`Abr` et si `un_abr.est_vide()` renvoie `False`, alors `un_abr.valeur` contient une chaîne de caractères représentant la valeur de la racine de l'arbre.
- Si `un_abr` est une instance d'`Abr` et si `un_abr.est_vide()` renvoie `False`, alors `un_abr.sous_arbre_gauche` et `un_abr.sous_arbre_droit` contiennent chacun une instance d'`Abr`.

On considère que la variable `liste_mots_francais` est une liste de 336531 mots en français et que la variable `abr_mots_francais` est une instance d'`Abr` contenant les mots de la liste.

On considère la fonction suivante :

```
1  def mystere(un_abr):
2      if un_abr.est_vide():
3          return 0
4      else:
5          return 1 + mystere(un_abr.sous_arbre_gauche) \
6              + mystere(un_abr.sous_arbre_droit)
```

<u>Remarque</u> : on rappelle que le caractère \ en fin de ligne 5 indique que l'instruction se poursuit sur la ligne suivante.

a. Donner le résultat de `mystere(abr_mots_francais)`, en justifiant le résultat.

On veut calculer la hauteur de `abr_mots_francais`.

b. Donner le code d'une fonction `hauteur(un_abr)` permettant de faire ce calcul, en vous inspirant du code précédent.

4. Dans cette question, nous nous servirons uniquement de `liste_mots_francais` et plus de `abr_mots_francais`.

Pour aider à la résolution de mots croisés, on a décidé d'écrire une fonction `chercher_mots(liste_mots, longueur, lettre, position)` où `liste_mots` est une liste de mots français, `longueur` est la taille du mot recherché, `lettre` est une lettre du mot se trouvant à l'indice `position`.

Par exemple `chercher_mots(liste_mots_francais,3,'x',2)` renverra `['aux', 'box', 'dix', 'eux', 'fax', 'fox', 'lux', 'max', 'six']`.

a. Recopier et compléter la ligne 4 de la fonction ci-dessous :

```
1  def chercher_mots(liste_mots,longueur,lettre,position):
2      res = []
3      for i in range(len(liste_mots)):
4          if ..... and ..... :
5              res.append(liste_mots[i])
6      return res
```

b. Expliquer ce que donne la commande suivante.

```
>>> chercher_mots(chercher_mots(liste_mots_francais,3,'x',2),3,'a',1)
```

On cherche un mot de 5 lettres dont on connait la fin `'ter'`.

c. Ecrire la commande permettant de chercher les mots candidats dans `liste_mots_francais`.