EXERCICE 1

Cet exercice porte sur les systèmes d'exploitation, les commandes UNIX, les structures de données (de type LIFO et FIFO) et les processus.

"Linux ou GNU/Linux est une famille de systèmes d'exploitation open source de type Unix fondée sur le noyau Linux, créé en 1991 par Linus Torvalds. De nombreuses distributions Linux ont depuis vu le jour et constituent un important vecteur de popularisation du mouvement du logiciel libre."

Source : Wikipédia, extrait de l'article consacré à GNU/Linux.

"Windows est au départ une interface graphique unifiée produite par Microsoft, qui est devenue ensuite une gamme de systèmes d'exploitation à part entière, principalement destinés aux ordinateurs compatibles PC. Windows est un système d'exploitation propriétaire."

Source : Wikipédia, extrait de l'article consacré à Windows.

1. Expliquer succinctement les différences entre les logiciels libres et les logiciels propriétaires.
2. Expliquer le rôle d'un système d'exploitation.

On donne ci-dessous une arborescence de fichiers sur un système GNU/Linux (les noms encadrés représentent des répertoires, les noms non encadrés représentent des fichiers, `/` correspond à la racine du système de fichiers) :

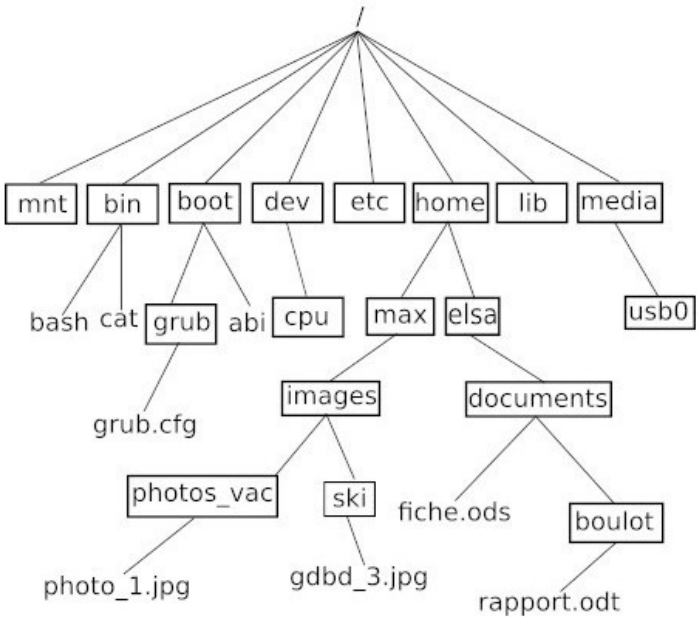

Figure 1. Arborescence de fichiers

3. Indiquer le chemin absolu du fichier `rapport.odt`.

On suppose que le répertoire courant est le répertoire `elsa`.

4. Indiquer le chemin relatif du fichier `photo_1.jpg`.

L'utilisatrice Elsa ouvre une console (aussi parfois appelée terminal), le répertoire courant étant toujours le répertoire `elsa`. On fournit ci-dessous un extrait du manuel de la commande UNIX `cp` :

```
NOM
    cp - copie un fichier

UTILISATION
    cp fichier_source fichier_destination
```

5. Déterminer le contenu du répertoire `documents` et du répertoire `boulot` après avoir exécuté la commande suivante dans la console :

```
cp documents/fiche.ods documents/boulot
```

"Un système d'exploitation est multitâche (en anglais : multitasking) s'il permet d'exécuter, de façon apparemment simultanée, plusieurs programmes informatiques. GNU/Linux, comme tous les systèmes d'exploitation modernes, gère le multitâche."

"Dans le cas de l'utilisation d'un monoprocesseur, la simultanéité apparente est le résultat de l'alternance rapide d'exécution des processus présents en mémoire."

Source : Wikipédia, extraits de l'article consacré au Multitâche.

Dans la suite de l'exercice, on s'intéresse aux processus. On considère qu'un monoprocesseur est utilisé. On rappelle qu'un processus est un programme en cours d'exécution. Un processus est soit élu, soit bloqué, soit prêt.

6. Recopier et compléter le schéma ci-dessous avec les termes suivants :

   élu, *bloqué*, *prêt*, *élection*, *blocage*, *déblocage*.

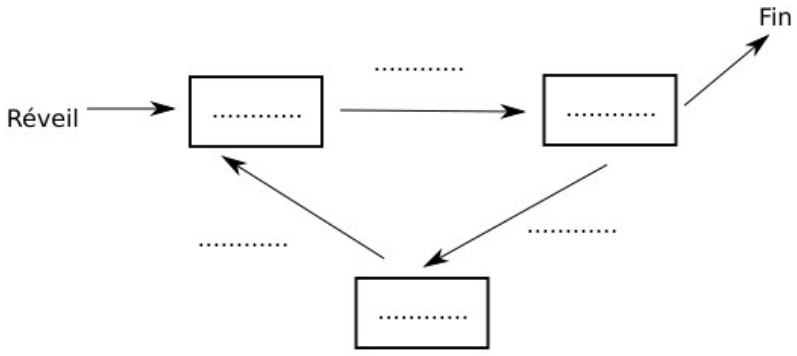

Figure 2. Schéma processus

7. Donner l'exemple d'une situation qui contraint un processus à passer de l'état élu à l'état bloqué.

“Dans les systèmes d’exploitation, l’ordonnanceur est le composant du noyau du système d’exploitation choisissant l’ordre d’exécution des processus sur le processeur d’un ordinateur.”

Source : Wikipédia, extrait de l’article consacré à l’ordonnancement.

L’ordonnanceur peut utiliser plusieurs types d’algorithmes pour gérer les processus.

L’algorithme d’ordonnancement par “ordre de soumission” est un algorithme de type FIFO (First In First Out), il utilise donc une file.

8. Nommer une structure de données linéaire de type LIFO (Last In First Out).

À chaque processus, on associe un instant d’arrivée (instant où le processus demande l’accès au processeur pour la première fois) et une durée d’exécution (durée d’accès au processeur nécessaire pour que le processus s’exécute entièrement).

Par exemple, l’exécution d’un processus P4 qui a un instant d’arrivée égal à 7 et une durée d’exécution égale à 2 peut être représentée par le schéma suivant :

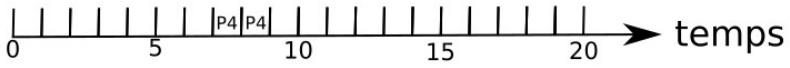

Figure 3. Utilisation du processeur

L’ordonnanceur place les processus qui ont besoin d’un accès au processeur dans une file, en respectant leur ordre d’arrivée (le premier arrivé étant placé en tête de file). Dès qu’un processus a terminé son exécution, l’ordonnanceur donne l’accès au processus suivant dans la file.

Le tableau suivant présente les instants d’arrivées et les durées d’exécution de cinq processus :

5 processus		
Processus	instant d’arrivée	durée d’exécution
P1	0	3
P2	1	6
P3	4	4
P4	6	2
P5	7	1

9. Recopier et compléter le schéma ci-dessous avec les processus P1 à P5 en utilisant les informations présentes dans le tableau ci-dessus et l’algorithme d’ordonnancement “par ordre de soumission”.

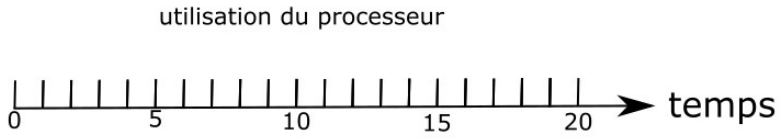

Figure 4. Utilisation du processeur

On utilise maintenant un autre algorithme d'ordonnancement : l'algorithme d'ordonnancement "par tourniquet". Dans cet algorithme, la durée d'exécution d'un processus ne peut pas dépasser une durée Q appelée quantum et fixée à l'avance. Si ce processus a besoin de plus de temps pour terminer son exécution, il doit retourner dans la file et attendre son tour pour poursuivre son exécution.

Par exemple, si un processus P1 a une durée d'exécution de 3 et que la valeur de Q a été fixée à 2, P1 s'exécutera pendant deux unités de temps avant de retourner à la fin de la file pour attendre son tour ; une fois à nouveau élu, il pourra terminer de s'exécuter pendant sa troisième et dernière unité de temps d'exécution.

10. Recopier et compléter le schéma ci-dessous, en utilisant l'algorithme d'ordonnancement "par tourniquet" et les mêmes données que pour la question 9, en supposant que le quantum Q est fixé 2.

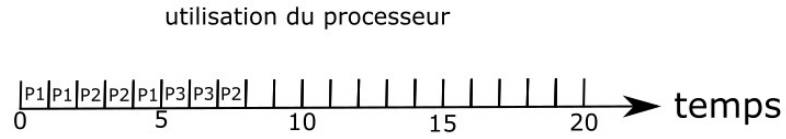

Figure 5. Utilisation du processeur

On considère deux processus P1 et P2, et deux ressources R1 et R2.

11. Décrire une situation qui conduit les deux processus P1 et P2 en situation d'interblocage.

Exercice 2

Cet exercice porte sur l'algorithmique, les structures de données, et la gestion de processus.

On cherche à créer une application de type *liste de tâches à faire* pour aider Alice à planifier sa journée. Pour cela Alice saisit les informations concernant chacune des tâches qu'elle doit effectuer : elle indique un nom pour la tâche, ainsi que la durée qu'elle estime nécessaire afin de la réaliser. On représente une tâche saisie par Alice à l'aide d'un objet de type `Tache`, muni de quatre attributs :

- le `numero` de la tâche, saisi par Alice ;
- le `nom` de la tâche, saisi par Alice ;
- la `duree` (un entier exprimé en minute) nécessaire à la réalisation de la tâche saisie par Alice ;
- la `duree_restante` (un entier exprimé en minute) avant la fin de la tâche. Cet attribut sera initialisé avec la durée totale nécessaire à la réalisation de la tâche.

Avancer de n minutes (n entier positif) dans une tâche consiste à diminuer de n la durée restante de cette tâche. Une tâche est terminée si la durée restante est négative ou nulle.

Lors de la phase de planification de ses tâches (aucune d'entre elles n'est commencée), Alice liste les tâches suivantes qui doivent être effectuées :

Numéro	Nom	Durée	Durée restante
1	Répondre aux e-mails	45	45
2	Ranger ma chambre	60	60
3	Réviser la NSI	90	90
4	S'entraîner aux échecs	30	30
5	Apprendre le vocabulaire de chinois	30	30
6	Lire Fondation	60	60
7	Écrire ma lettre au Père Noël	20	20

On dispose de la classe `Tache` ci-dessous pour représenter les tâches :

```
1 class Tache:
2     def __init__(self, numero, nom, duree):
3         self.numero = numero
4         self.nom = nom
5         self.duree_initiale = duree
6         self.duree_restante = duree
7 
8     def __repr__(self):
9         return '<t'+str(self.numero)+'>'
```

1. Donner le code Python qui permet d'instancier deux variables `tache1` et `tache2` représentant les tâches :

   – tâche numéro 1 : Répondre aux e-mails. Durée estimée : 45 minutes.

   – tâche numéro 2 : Ranger ma chambre. Durée estimée : 60 minutes.

On supposera dans la suite que les variables `tache1`, `tache2`, ..., `tache7` représentent les tâches établies par Alice lors de la phase de planification.

La méthode `__repr__` renvoie une représentation de l'instance sous forme d'une chaîne de caractères. La fonction `print` utilise cette méthode. Ainsi on a :

```
>>> print(tache1)
<t1>
```

2. Recopier et compléter le code de la méthode `avancer` de la classe `Tache` qui permet d'avancer la tâche `self` de n minutes.

```
1     def avancer(self, n):
2         ...
```

3. Recopier et compléter le code de la méthode `est_terminee` de la classe `Tache` qui renvoie `True` si la tâche est terminée, ou `False` sinon.

```
1     def est_terminee(self):
2         ...
```

Afin d'aider Alice à planifier sa journée, on lui propose d'associer à chacune des tâches une priorité. La priorité d'une tâche est représentée par un entier de la manière suivante : 1 est la priorité minimale et, plus le nombre est grand, plus la tâche associée est prioritaire.

Pour stocker toutes les tâches à effectuer, on utilise une file, dans laquelle les éléments sont des tuples `(tache, priorite)`. Les éléments stockés dans la file doivent respecter les deux conditions ci-après.

- Condition 1 : les éléments sont rangés par ordre décroissant de priorité. L'élément de priorité maximale se trouve au début de la file, l'élément le moins prioritaire se trouve à la fin de la file.

- Condition 2 : parmi les éléments de même priorité, les éléments sont rangés dans l'ordre dans lequel ils ont été insérés dans la file. Ainsi, le premier élément de priorité p inséré se trouve devant les éléments de même priorité p insérés plus tard.

Par exemple, si la file de tâches `f` est la file :

```
[début] (<t3>, 4) (<t1>, 3) (<t2>, 3) (<t4>, 1) (<t5>, 1) [fin]
```

Cela signifie que :

- la tâche de priorité maximale est la tâche numéro 3 ;
- les deux tâches à exécuter en priorité après la tâche numéro 3 sont les tâches numéro 1 et numéro 2. La tâche numéro 1 a été ajoutée à la file des tâches à traiter avant la tâche numéro 2 ;
- il n'y a pas de tâche de priorité 2 ;
- les tâches les moins prioritaires de la file sont les tâches numéro 4 et numéro 5. La tâche numéro 4 a été ajoutée avant la tâche numéro 5.

4. Représenter l'état de la file `f` lorsqu'on lui ajoute successivement la tâche numéro 6 avec la priorité 2, puis la tâche numéro 7 avec la priorité 4 en respectant les conditions 1 et 2 décrites ci-dessus.

On suppose déjà définies les méthodes suivantes pour la classe `File` :

- `File()` : crée et renvoie un objet de type `File`, vide.
- `enfiler(self, e)` : ajoute l'élément e à la fin de la file `f`.
- `defiler(self)` : renvoie, en le supprimant de la file, le premier élément de la file si cela est possible.
- `examiner(self)` : renvoie, sans le supprimer de la file, le premier élément de la file si cela est possible.
- `est_vide(self)` : renvoie `True` si la file est vide, ou `False` sinon.

5. En repartant de la file `f` suivante :

   ```
   [début](<t3>, 4)(<t1>, 3)(<t2>, 3)(<t4>, 1)(<t5>, 1)[fin]
   ```

   donner la valeur de `f.defiler()[0]`, et représenter le contenu de la file `f` après l'exécution de cette instruction.

6. En repartant de la file `f` suivante :

   ```
   [début](<t3>, 4)(<t1>, 3)(<t2>, 3)(<t4>, 1)(<t5>, 1)[fin]
   ```

   donner la valeur de `f.examiner()[1]`, et représenter le contenu de la file `f` après l'exécution de cette instruction.

On souhaite écrire une fonction `ajouter_file_prio` qui prend en paramètres :

- une file `f` dont les éléments sont des tuples `(tache, priorite)` respectant les deux conditions de l'énoncé ;
- une tâche `t` ;
- la priorité `p` de la tâche `t` ;

et qui ajoute le tuple `(t, p)` à la bonne position dans la file `f`.

On utilise une file auxiliaire `f_aux` que l'on remplit en défilant les éléments en début de file `f` tant que la priorité du premier élément de la file est supérieure ou égale à `p`. Puis on enfile l'élément `(t, p)` dans la file auxiliaire. On défile ensuite tous les éléments restants de `f` dans `f_aux` et enfin on enfile dans `f` tous les éléments de `f_aux`.

7. Recopier et compléter le code de la fonction `ajouter_file_prio`.

```
1 def ajouter_file_prio(f, t, p):
2     f_aux = File()
3     while ...:
4         ...
5     ...enfiler(...)
6     while not ...:
7         ...
8     while not ...:
9         ...
```

8. Donner le coût d'exécution temporel dans le pire des cas de la fonction `ajouter_file_prio`, en fonction du nombre `m` d'éléments de la file `f`.

Une fois qu'Alice a entré les tâches qu'elle doit effectuer, leur durée estimée, ainsi que la priorité à laquelle elle doit les effectuer, l'application lui propose un planning en utilisant la technique dite Pomodoro :

- la tâche à effectuer est la tâche qui se trouve en tête de file ;
- on défile cette tâche de la file des tâches à effectuer ;
- on avance cette tâche de 25 minutes ;
- si cette tâche n'est pas terminée, on rajoute cette tâche dans la file des tâches à effectuer, avec la même priorité qu'initialement (en utilisant la fonction `ajouter_file_prio`) ;
- si cette tâche se termine au cours des 25 minutes, alors Alice attend la fin des 25 minutes en se reposant ;
- on continue ces étapes tant que la file des tâches à effectuer n'est pas vide.

On rappelle les tâches à effectuer ci-dessous, classées par ordre de priorité. On considérera que les tâches sont ajoutées à la file de priorité dans l'ordre du tableau ci-dessous :

Numéro	Nom	Durée	Priorité
3	Réviser la NSI	90	4
7	Écrire ma lettre au Père Noël	20	4
1	Répondre aux e-mails	45	3
2	Ranger ma chambre	60	3
6	Lire Fondation	60	2
4	S'entraîner aux échecs	30	1
5	Apprendre le vocabulaire de chinois	30	1

9. Indiquer pour chaque bloc de 25 minutes la tâche qui avance, en suivant le modèle proposé, jusqu'à la fin de toutes les tâches.

   On fera particulièrement attention au cas où la tâche n'est pas terminée : celle-ci est rajoutée à la file des tâches à effectuer (dont elle avait été supprimée) avec la même priorité qu'initialement, en respectant les conditions 1 et 2 de l'énoncé.

10. Écrire le code d'une fonction `planning` qui prend en paramètre une file de priorité `f` dont les éléments sont des tuples `(tache, prio)`, et qui renvoie une liste de tâches, dans l'ordre dans lequel elles vont être effectuées par tranche de 25 minutes avec la méthode Pomodoro.

    Par exemple, si `tache1`, `tache2` et `tache3` sont les tâches numéro 1, numéro 2 et numéro 3, alors le programme suivant :

```
1 file = File()
2 for t, p in [(tache1, 3), (tache2, 3), (tache3, 4)]:
3     ajouter_file_prio(file, t, p)
4 print(planning(file))
```

    produit l'affichage :

```
[<t3>, <t3>, <t3>, <t3>, <t1>, <t2>, <t1>, <t2>, <t2>]
```