Processus

I. Commandes Unix

1. Commandes `ps` et `top`

Nous allons utiliser les deux commandes `ps` et `top` qui permettent de lister les processus dans le terminal.

1. Dans un terminal : *à taper*

```
nsi@ordi$ gnome-text-editor &
nsi@ordi$ ps
```

2. résultat : *à compléter*

```
PID TTY          TIME CMD
.
.
.
```

3. Dans le terminal : *à taper*

```
nsi@ordi$ ps -u
```

4. Que représente chacune des colonnes ? *à compléter*
   - `USER` indique
   - `PID` donne l'identifiant numérique du processus.
   - `%CPU` et `%MEM` indiquent respectivement
   - `STAT` indique l'état du processus, `S` pour *sleeping*, le processus est en attente et `R` pour *running*, le processus est dans l'état prêt ou élu.
   - `COMMAND` indique la commande utilisée pour lancer le programme.
   - `START` et `TIME` indiquent respectivement
5. Fermer l'application `gnome-text-editor` et utiliser à nouveau la commande `ps` dans le terminal.

   Que constatez-vous ?
6. Dans le terminal : *à taper*

```
nsi@ordi$ top
```

   Quelles sont les principales différences avec la commande `ps` ?
   -
   -

2. Commandes `kill` et `killall`

La commande `cat` n'est utilisée ici qu'à titre d'exemple, sa fonctionnalité n'est pas importante ici.

1. Dans un premier terminal : *à taper*

   ```
   nsi@ordi$ cat
   ```

2. Nous allons envoyer un signal de terminaison au processus `cat` par le biais de la commande `killall` qui envoie un signal aux processus dont le nom est indiqué.
   Dans un deuxième terminal : *à taper*

   ```
   nsi@ordi$ killall cat
   ```

3. Que constatez-vous ?

4. Dans le premier terminal taper à nouveau `cat`, puis, dans le deuxième terminal, à l'aide de la commande `ps -u`, déterminer le PID du processus ainsi créé : *à compléter*

   ```
   USER     PID %CPU     %MEM     VSZ RSS TTY STAT     START    TIME     COMMAND
                0.0      0.0                  S+               0:00     cat
   ```

   Remarque On peut également utiliser la commande `pgrep` en tapant dans le terminal la commande `pgrep cat`.

5. dans le deuxième terminal, *à taper*
   en utilisant le PID de `cat` que vous avez obtenu :

   ```
   nsi@ordi$ kill 17369
   ```

6. Que constatez-vous ?

3. En résumé

ps : liste les processus actifs attachés au terminal, les processus sont identifiés par leur PID.
top : fournit une vue dynamique temps réel du système en cours d'exécution.
kill : interrompt le processus dont le PID est donné en paramètre.
killall : interrompt **les** processus dont le nom est donné en paramètre.

Tester les différentes options `-u`, `-a`, `-e` et `-f` de la commande `ps`.
Déterminer leur fonctionnement à l'aide de la commande `man ps`.

- `ps -u` :
- `ps -a` :
- `ps -e` :
- `ps -f` :

II. Les processus

1. Définition d'un processus

Définition Un **processus** est une instance d'exécution d'un programme.

- Un processus est décrit par :
  - la mémoire allouée par le système pour l'exécution du programme ;
  - les ressources utilisées par le programme ;
  - les valeurs stockées dans les registres du processeur.
- Un processus possède un numéro unique, le PID (*Process ID*).

Les notions de programmes et de processus sont différentes : le même programme exécuté plusieurs fois générera plusieurs processus.

2. Un exemple

1. Dans un terminal, exécuter la commande suivante :

   ```
   nsi@ordi$ gnome-system-monitor
   ```

2. Dans un deuxième terminal, faire en sorte d'obtenir le PID du processus comme nous l'avons vu précédemment.
3. Dans le deuxième terminal toujours : *à taper*
   En utilisant le PID de `gnome-system-monitor` que vous avez obtenu :

   ```
   nsi@ordi$ top -p 5963
   ```

4. Que constatez-vous comme changements dans l'affichage de la fonction `top` lorsque vous utilisez le programme `gnome-system-monitor` ?

3. Les différents états d'un processus

Définition Les principaux états d'un processus sont les suivants :

Prêt : le processus peut être le prochain a s'exécuter. Il est dans la file des processus qui attendent leur tour.

Élu (actif ou exécution) : les processus est entrain de s'exécuter.

Bloqué : le processus est interrompu et en attente d'un événement externe (entrée/sortie, allocation mémoire, etc.)

On peut rajouter à ces trois états deux états éphémères **nouveaux** et **terminé** qui correspondent respectivement à un processus en cours de création et au système d'exploitation qui désalloue les ressources attribués a un processus qui vient de se finir.

4. Le cycle de vie d'un processus

Après sa création un processus est mis dans l'état prêt. En temps normal un processus variera de entre *prêt*, *élu* et *bloqué*.

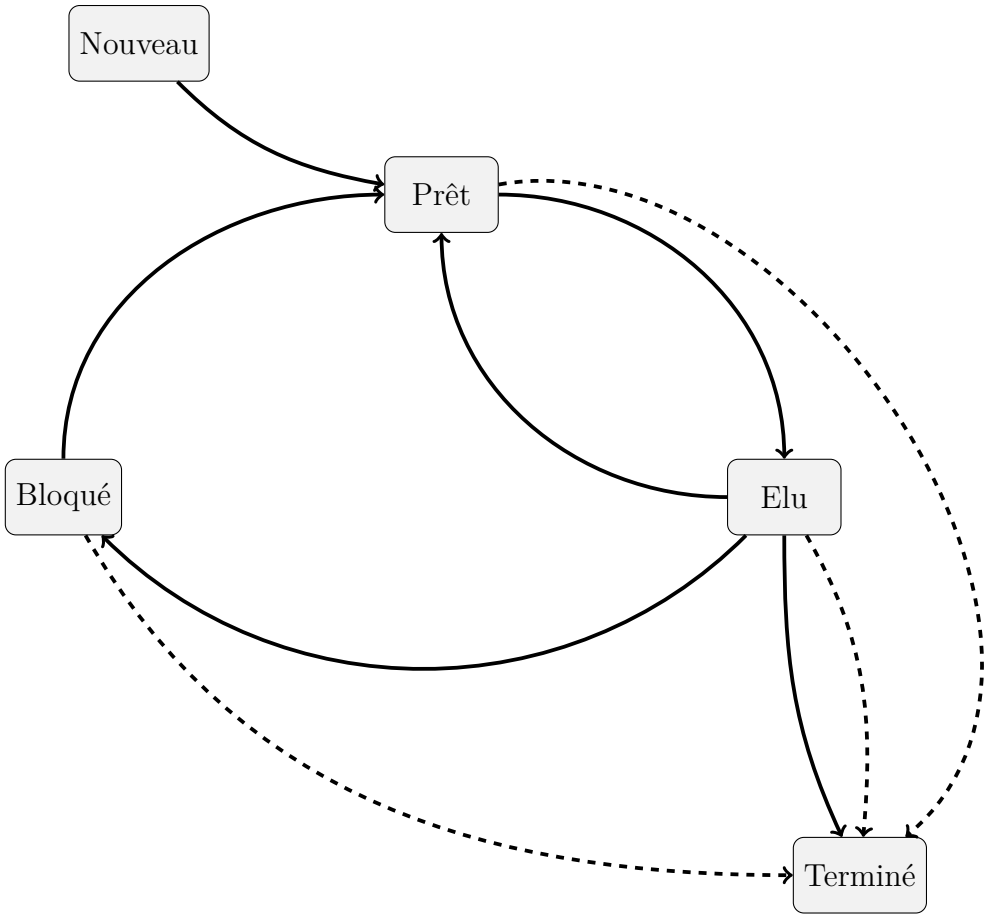
Plusieurs processus peuvent être dans l'état *prêt* mais un seul sera placé dans l'état *élu* ; c'est **l'ordonnanceur** (*scheduler*) du système d'exploitation qui est chargé de classer les processus dans une file. C'est lui qui décide quel processus est actif et a pour charge, de manière permanente et à une fréquence élevée, de désactiver le processus actif pour en activer un autre, faisant ainsi fonctionner alternativement les différents processus, donnant l'impression qu'ils fonctionnent en même temps.

Alors qu'il est élu, le processus peut avoir besoin d'attendre une ressource quelconque comme, par exemple, une ressource en mémoire. Il doit alors quitter momentanément le processeur, pour que ce dernier puisse être utilisé à d'autres tâches. Le processus passe donc dans l'état bloqué.

Une fois le processus terminé, il est placé dans l'état *terminé*, le système d'exploitation libère alors les ressources qui lui sont allouées. Notons que quel que soit l'état du processus, il peut se terminer de façon anormale (erreur dans un programme, problème matériel, interruption de l'utilisateur, etc.).

Le schéma ci-dessous résume le cycle de vie d'un processus. À chaque flèche du schéma (sauf celle partant de « Nouveau »), ajouter le numéro correspondant parmi ceux donnés ci-dessous (certains numéros peuvent apparaître plusieurs fois) :

1. *mise en exécution par l'ordonnanceur ;*
2. *interruption par l'ordonnanceur ;*
3. *terminaison normale ;*
4. *terminaison anormale ;*
5. *attente d'une ressource ;*
6. *obtention de la ressource.*



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*)

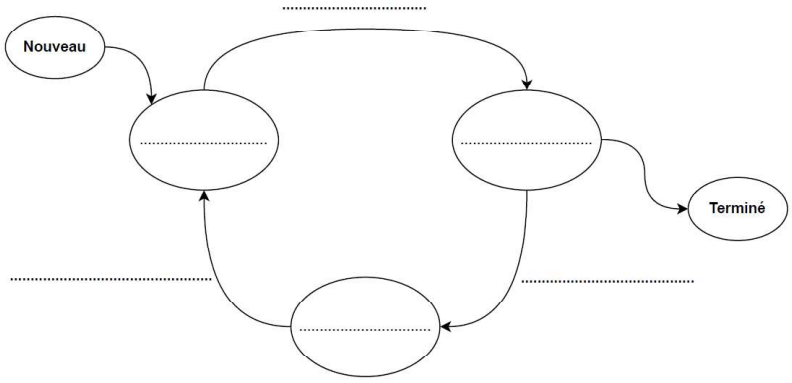
a. Donner sous forme d'un arbre de PID la hiérarchie des processus liés à *`firefox`*.

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*



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.

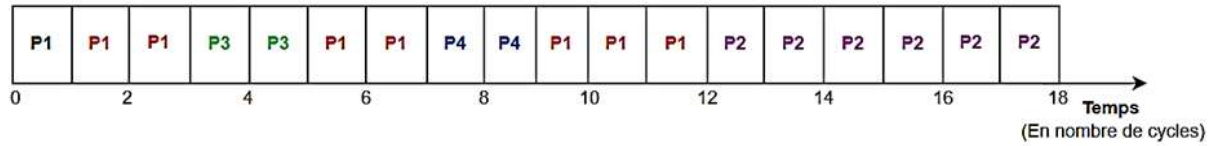
Processus	P1	P2	P3	P4
Instant d'arrivée	0	2	3	7
Durée	8	6	2	2

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.

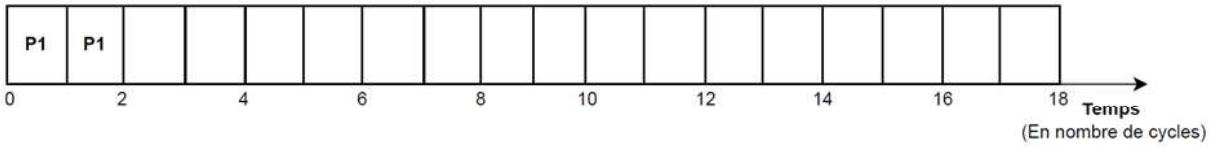


On définit le temps d'exécution d'un processus comme la différence entre son instant de terminaison et son instant d'arrivée.

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.

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
```

Exercice 2 — **4 points**

Cet exercice porte sur la gestion des processus et la programmation orientée objet

On rappelle qu'un processus est l'instance d'un programme en cours d'exécution. Il est identifié par un numéro unique appelé PID. L'ordonnanceur est la composante du système d'exploitation qui gère l'allocation du processeur entre les différents processus. Nous allons nous intéresser à l'algorithme d'ordonnancement du tourniquet dont le fonctionnement est résumé ci-dessous :

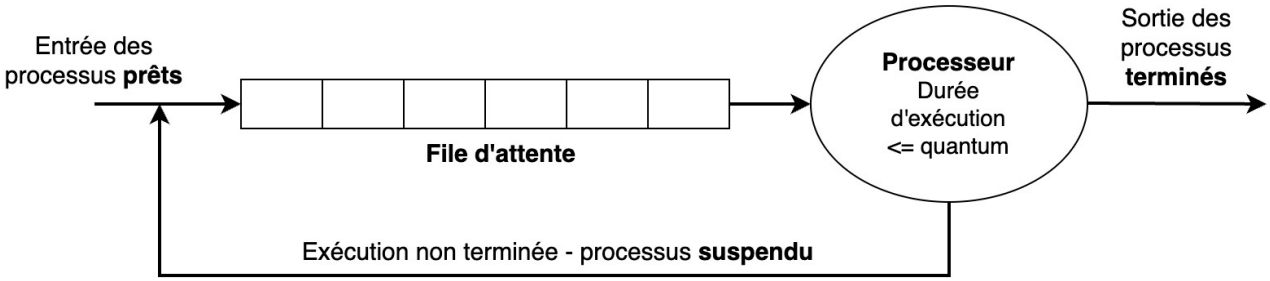


Schéma d'ordonnancement du tourniquet

- Les processus prêts à être exécutés sont placés dans une file d'attente selon leur ordre d'arrivée ;
- L'ordonnanceur alloue le processeur à chaque processus de la file d'attente un même nombre de cycles CPU, appelé **quantum** ;
- Si le processus n'est pas terminé au bout de ce temps, son exécution est suspendue et il est mis à la fin de la file d'attente ;
- Si le processus est terminé, il sort définitivement de la file d'attente.

1. 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

a) Si le quantum du tourniquet est d'un cycle CPU, recopier et compléter la suite des PID des processus dans l'ordre de leur exécution :
11, 20, 32, 11, ...

b) Donner la composition de la suite des PID lorsque le quantum du tourniquet est de deux cycles CPU.

2. L'objectif de la suite de l'exercice est d'implémenter en langage Python l'algorithme du tourniquet.
Nous allons utiliser une liste pour simuler la file d'attente des processus et la classe `Processus` dont le constructeur est donné ci-dessous :

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

Les états possibles d'un processus sont : « *Prêt* », « *En cours d'exécution* », « *Suspendu* »et « *Terminé* ».

a) Recopier et compléter l'instruction Python suivante permettant de créer la liste d'attente initiale des processus donnés dans le tableau précédent (le processus PID 11 est à l'indice 0 de la liste d'attente) :

```
liste_attente = [Processus(...,...), ..........., .........]
```

b) Recopier (sans les commentaires) et compléter les trois méthodes suivantes de la classe `Processus` :

```
    def execute_un_cycle(self):
        """Met à jour le reste à faire après l'exécution d'un
        cycle."""
        ..........................................

    def change_etat(self, nouvel_etat):
        """Change l'état du processus avec la valeur passée en
        paramètre."""
        ..........................................

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

c) 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.
Recopier et compléter sur la copie le code manquant.

```
1  def tourniquet(liste_attente, quantum):
2      ordre_execution = []
3      while liste_attente != []:
4          # On extrait le premier processus
5          processus = liste_attente.pop(0)
6          processus.change_etat("En cours d'exécution")
7          compteur_tourniquet = 0
8          while ..................... and .....................:
9              ordre_execution.append(...........)
10             processus.execute_un_cycle()
11             compteur_tourniquet = compteur_tourniquet + 1
12         if  .....................:
13             processus.change_etat("Suspendu")
14             liste_attente.append(processus)
15         else:
16             processus.change_etat(...........)
17     return ordre_execution
```

EXERCICE 1 (6 points)

Cet exercice porte sur la programmation Python, la programmation orientée objet, les structures de données (file), l'ordonnancement et l'interblocage.

On s'intéresse aux processus et à leur ordonnancement au sein d'un système d'exploitation. On considère ici qu'on utilise un monoprocesseur.

1. Citer les trois états dans lesquels un processus peut se trouver.

On veut simuler cet ordonnancement avec des objets. Pour ce faire, on dispose déjà de la classe `Processus` dont voici la documentation :

Classe Processus:

```
p = Processus(nom: str, duree: int)
    Crée un processus de nom <nom> et de durée <duree> (exprimée en
cycles d'ordonnancement)

p.execute_un_cycle()
    Exécute le processus donné pendant un cycle.

p.est_fini()
    Renvoie True si le processus est terminé, False sinon.
```

Pour simplifier, on ne s'intéresse pas aux ressources qu'un processus pourrait acquérir ou libérer.

2. Citer les deux seuls états possibles pour un processus dans ce contexte.

Pour mettre en place l'ordonnancement, on décide d'utiliser une file, instance de la classe `File` ci-dessous.

Classe File

```
1  class File:
2      def __init__(self):
3          """ Crée une file vide """
4          self.contenu = []
5
6      def enfile(self, element):
7          """ Enfile element dans la file """
8          self.contenu.append(element)
9
10     def defile(self):
11         """ Renvoie le premier élément de la file et l'enlève de
la file """
12         return self.contenu.pop(0)
13
14     def est_vide(self):
```

```
15          """ Renvoie True si la file est vide, False sinon """
16          return self.contenu == []
```

Lors de la phase de tests, on se rend compte que le code suivant produit une erreur :

```
1 f = File()
2 print(f.defile())
```

3. Rectifier sur votre copie le code de la classe `File` pour que la function `defile` renvoie `None` lorsque la file est vide.

On se propose d'ordonnancer les processus avec une méthode du type *tourniquet* telle qu'à chaque cycle :

- si un nouveau processus est créé, il est mis dans la file d'attente ;
- ensuite, on défile un processus de la file d'attente et on l'exécute pendant un cycle ;
- si le processus exécuté n'est pas terminé, on le replace dans la file.

Par exemple, avec les processus suivants

Liste des processus		
processus	cycle de création	durée en cycles
A	2	3
B	1	4
C	4	3
D	0	5

On obtient le chronogramme ci-dessous :

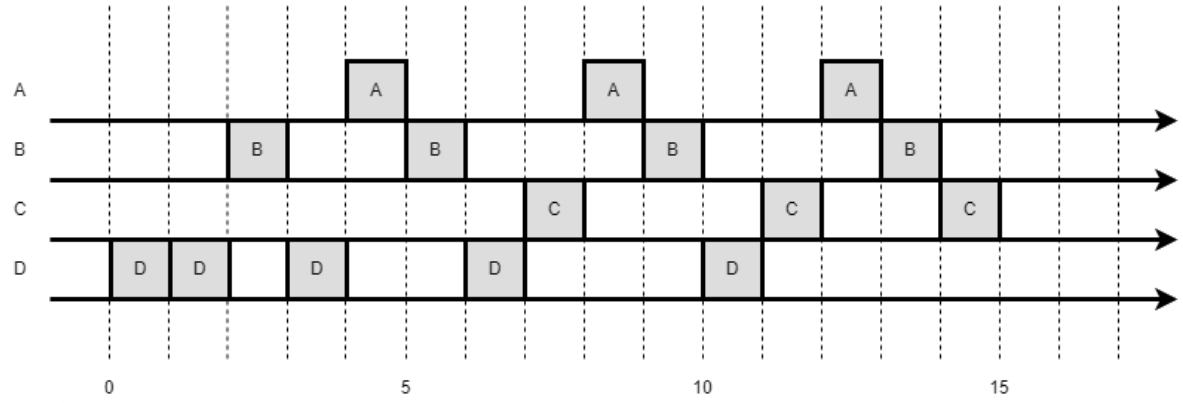


Figure 1. Chronogramme pour les processus A, B, C et D

Pour décrire les processus et le moment de leur création, on utilise le code suivant, dans lequel `depart_proc` associe à un cycle donné le processus qui sera créé à ce moment :

```
1 p1 = Processus("p1", 4)
2 p2 = Processus("p2", 3)
3 p3 = Processus("p3", 5)
4 p4 = Processus("p4", 3)
5 depart_proc = {0: p1, 1: p3, 2: p2, 3: p4}
```

Il s'agit d'une modélisation de la situation précédente où un seul processus peut être créé lors d'un cycle donné.

4. Recopier et compléter sur votre copie le chronogramme ci-dessous pour les processus `p1`, `p2`, `p3` et `p4`.

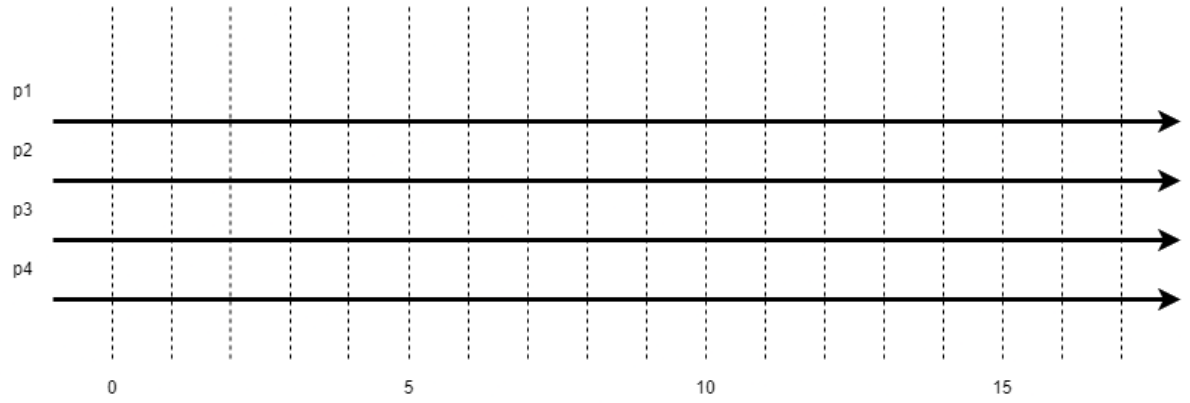


Figure 2. Chronogramme pour les processus p1, p2, p3 et p4

Pour mettre en place l'ordonnancement suivant cette méthode, on écrit la classe `Ordonnanceur` dont voici un code incomplet (l'attribut `temps` correspond au cycle en cours) :

```
1  class Ordonnanceur:
2
3      def __init__(self):
4          self.temps = 0
5          self.file = File()
6
7      def ajoute_nouveau_processus(self, proc):
8          '''Ajoute un nouveau processus dans la file de
9            l'ordonnanceur. '''
10           ...
11
12       def tourniquet(self):
13           '''Effectue une étape d'ordonnancement et renvoie le nom
14             du processus élu.'''
15           self.temps += 1
16           if not self.file.est_vide():
17               proc = ...
18               ...
19               if not proc.est_fini():
20                   ...
21               return proc.nom
22           else:
23               return None
```

5. Compléter le code ci-dessus.

À chaque appel de la méthode `tourniquet`, celle-ci renvoie soit le nom du processus qui a été élu, soit `None` si elle n'a pas trouvé de processus en cours.

6. Écrire un programme qui :

- utilise les variables `p1`, `p2`, `p3`, `p4` et `depart_proc` définies précédemment ;
- crée un ordonnanceur ;
- ajoute un nouveau processus à l'ordonnanceur lorsque c'est le moment ;
- affiche le processus choisi par l'ordonnanceur ;
- s'arrête lorsqu'il n'y a plus de processus à exécuter.

Dans la situation donnée en exemple (voir Figure 1), il s'avère qu'en fait les processus utilisent des ressources comme :

- un fichier commun aux processus ;
- le clavier de l'ordinateur ;
- le processeur graphique (GPU) ;
- le port 25000 de la connexion Internet.

Voici le détail de ce que fait chaque processus :

Liste des processus			
A	B	C	D
acquérir le GPU	acquérir le clavier	acquérir le port	acquérir le fichier
faire des calculs	acquérir le fichier	faire des calculs	faire des calculs
libérer le GPU	libérer le clavier	libérer le port	acquérir le clavier
	libérer le fichier		libérer le clavier
			libérer le fichier

7. Montrer que l'ordre d'exécution donné en exemple aboutit à une situation d'interblocage.