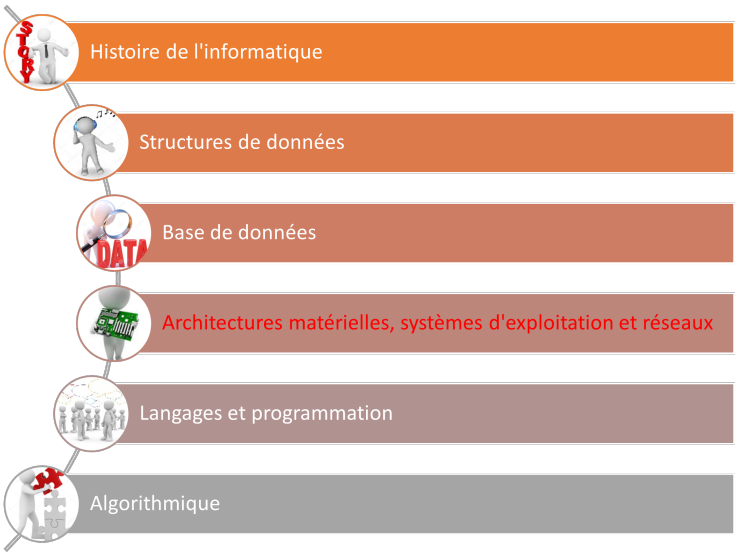


Table des matières

Sources : www.wikipédia.org, www.genumsi.inria.fr, www.monlyceenumerique.fr, Laurent Godefroy, Dominique Laporte, "Numérique et sciences informatiques Tle : 24 leçons avec exercices corrigés" Ellipses, "Numérique et sciences informatiques Tle" collection Prépas Sciences Ellipses, "Apprendre à programmer avec Python 3" Eyrolles, documentation Python : https ://docs.python.org/fr/3/library/concurrency.html

1 Situation déclenchante

Considérons la situation suivante : vous rédigez un compte rendu pour un projet informatique, vous avez donc "ouvert" un logiciel de traitement de texte pour écrire le rapport. Votre navigateur web est aussi ouvert avec divers onglets, l'un pointant sur Wikipédia, l'autre vers un moteur de recherche et un troisième vers un site de réseau social via lequel vous échangez avec vos camarades. Vous utilisez un logiciel de dessin pour ajouter des illustrations à votre compte rendu. Votre projet étant en python, vous avez ouvert votre éditeur de code et vous avez lancé l'exécution de votre programme afin d'en vérifier les résultats. Enfin, vous écoutez de la musique grâce au lecteur de musique de votre ordinateur.
Tous ces programmes s'exécutent "en même temps". Pourtant nous avons vu en première le fonctionnement des ordinateurs. Ils ne disposent que d'un nombre limité de processeurs. Or, comme on le sait, un programme n'est qu'une suite d'instructions en langage machine, ces dernières étant exécutées une à une par le processeur. Comment le processeur peut-il donc exécuter "en même temps" les instructions des différents programmes en cours d'exécution dans notre exemple ci-dessus ?

2 Systèmes d'exploitation

Le système d'exploitation (OS) constitue l'interface entre la machine et l'utilisateur. Il est constitué d'un ensemble de programmes pour gérer les ressources et les partager :

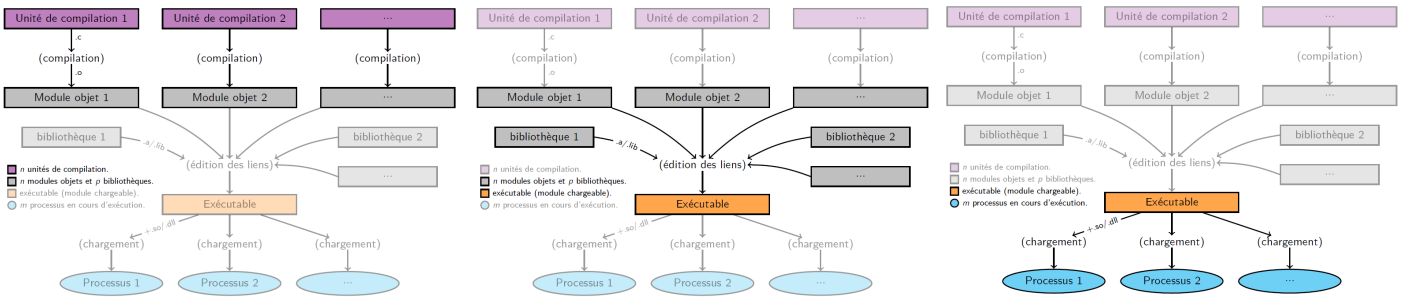
- la gestion des processus,
- la gestion des fichiers,
- la gestion de la mémoire,
- la gestion des périphériques,
- le traitement des entrées-sorties,
- la sécurité.

Une machine peut posséder plusieurs système d'exploitation. L'utilisateur peut choisir l'OS au démarrage (système LINUX et Windows par exemple). Le système d'exploitation peut se trouver sur le disque dur ou sur un autre support (clé USB par exemple).

3 Cycle de vie d'un programme

3.1 Cheminement d'un programme dans un système

1. Analyse du problème, conception, algorithmique, ..., sur papier !
2. Le code source est écrit dans un langage donné (en Python par exemple) à l'aide d'un éditeur de texte.
3. Chaque unité de compilation est compilée séparément, i.e. convertie dans un langage machine de type assembleur. On obtient ainsi des programmes appelés modules objets.
4. Les modules objets sont combinés en un unique programme exécutable (appelé module chargeable) lors de l'édition des liens.
5. Le programme est chargé en mémoire centrale afin d'être exécuté (chaque occurrence d'exécution est appelée processus).



3.2 Vocabulaire

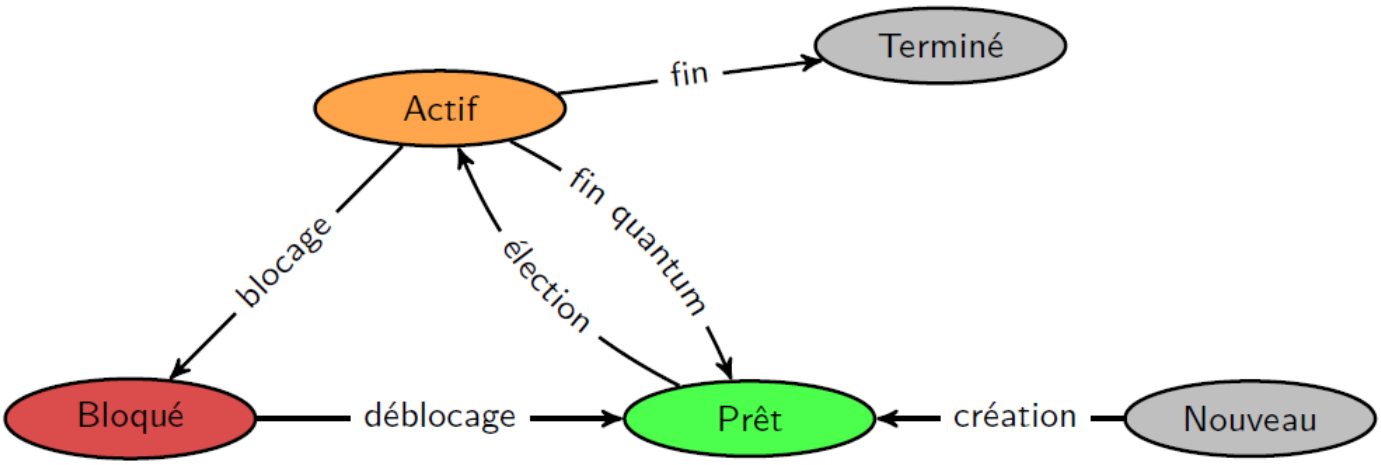
Définition :
Un programme est une entité statique, il s'agit d'un fichier exécutable stocké en mémoire. Un ou plusieurs processus particuliers peuvent correspondre à ce programme en cours d'exécution.

Définition :
On appelle processus toute occurrence d'exécution d'un programme sur la machine. Il donne l'impression d'exploiter toutes les ressources de la machine, dont le processeur, comme s'il était seul. Un processus est dynamique (il a un début et une fin).

3.3 Les différents états d'un processus

Dans un système à temps partagé, un processus peut être dans 3 états différents (on néglige les états transitoires) :

- le processus pour lequel le CPU exécute les instructions est dit **actif** ou **élu** ;
- un processus dont seule la ressource CPU fait défaut est **prêt** ;
- et un processus à qui il manque au moins une autre ressource que le CPU est dit **bloqué** (il attend, par exemple, la fin d'une opération d'E/S).



Remarque : Un processus peut créer d'autres processus.

Des identificateurs sont associés à chaque processus :

- Le PID (Process Identification) est le numéro du processus,
- Le PPID est le numéro du processus père,
- l'UID (User Identification) est l'identifiant de l'utilisateur qui a démarré le processus,
- le GID (Group Identification) est l'identifiant du groupe de l'utilisateur qui a démarré le processus.

On peut distinguer trois types de processus :

- les applications,
- les processus qui fonctionnent en permanence en arrière plan (services sous Windows, deamons sous Linux), et s'exécutent dès le démarrage de la machine, comme les antivirus,
- les processus du système d'exploitation.

Dans la plupart des cas les processus sont créés par d'autres processus (processus père/fils).

3.4 Les processus sous Windows

3.4.1 Le gestionnaire de tâches sous Windows

Le gestionnaire des tâches permet d'observer en détail les processus en cours et de les interrompre.

Sous Windows, faire un clic droit sur la barre des tâches et sélectionner "gestionnaire des tâches" ou bien CTRL + ALT + SUPP.
Dans l'onglet détail les processus sont affichés dans l'ordre alphabétique. On peut voir les PID.
Dans l'onglet Fichier on peut lancer un processus (firefox.exe par exemple). Avec un clic droit, on peut arrêter un processus.

3.4.2 L'invite de commande sous Windows

L'invite de commande permet de contourner l'interface graphique Windows (taper cmd dans la zone de recherche, sélectionner "Exécuter en tant qu'administrateur").

Entrer la commande `tasklist`. Tous les processus sont listés.

```
C:\WINDOWS\system32>tasklist
```

Entrer la commande `tasklist | findstr "firefox"`. Tous les processus `firefox` sont listés.

```
C:\WINDOWS\system32>tasklist | findstr "firefox"
firefox.exe                  6072 Console                    8   216 892 Ko
firefox.exe                  8484 Console                    8    60 684 Ko
firefox.exe                  8332 Console                    8    56 508 Ko
firefox.exe                 15316 Console                    8    73 716 Ko
firefox.exe                  8736 Console                    8    34 068 Ko
firefox.exe                 13192 Console                    8    33 980 Ko
firefox.exe                  5308 Console                    8    33 932 Ko
```

Pour arrêter `firefox`, il faut tuer tous les processus.
Avec le PID :

```
C:\WINDOWS\system32>taskkill /PID 6072 /T
Opération réussie : un signal de fin a été envoyé au processus de PID 8484,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 8332,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 15316,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 8736,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 13192,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 5308,
processus enfant de PID 6072.
Opération réussie : un signal de fin a été envoyé au processus de PID 6072,
processus enfant de PID 1228.
```

L'option `/T` sert à tuer tous les processus enfants.
Avec le nom :

```
C:\WINDOWS\system32>taskkill /f /im firefox.exe
Opération réussie : le processus "firefox.exe" de PID 15288 a été arrêté.
Opération réussie : le processus "firefox.exe" de PID 14508 a été arrêté.
Opération réussie : le processus "firefox.exe" de PID 6660 a été arrêté.
Opération réussie : le processus "firefox.exe" de PID 13276 a été arrêté.
```

L'option `/f` sert à forcer l'arrêt et l'option `/im` tuer tous les processus enfants.

3.5 Les processus sous Linux

On ouvre un terminal. Voici les principales commandes disponibles :

- La commande `top` permet de lister tous les processus en cours d'exécution.
- La commande `htop` propose d'exécuter différentes actions sur les processus affichés.
- La commande `ps` liste les processus.
- La commande `ps aux` liste les processus par ordre de PID.
- La commande `ps aux | grep fire` liste les processus qui contiennent fire dans leur nom.
- La commande `pstree -p` affiche les processus sous forme d'arbre (pères et les fils).
- La commande `ps -aef` affiche en plus le PPID
- La commande `kill 1522` permet d'arrêter le processus dont le PID est 1522.

4 Thread et multithreading

Définition :
Un processus est une suite d'instructions traitées par le processeur. Ces instructions sont regroupées en séquences appelées **thread**. Un thread est donc l'exécution d'un ensemble d'instructions du langage machine d'un processeur, on parle aussi de fil d'exécution, de tâche ou même d'activité.

- Le processeur n'exécute qu'un seul thread à la fois.
- Contrairement aux processus, les threads partagent le même espace d'adressage, le même environnement (variables d'environnement, fichiers, ...).
- Chacun possède son propre contexte d'exécution qui comporte le pointeur sur la pile d'exécution, le compteur ordinal, ...
- Un thread possède donc moins de ressources propres qu'un processus. Sa gestion est moins coûteuse.

Définition :
Multithreading : est une technique qui permet à un processus de lancer plusieurs threads simultanément. Les threads partagent les ressources (mémoire, caches, CPU, ...) du processus.

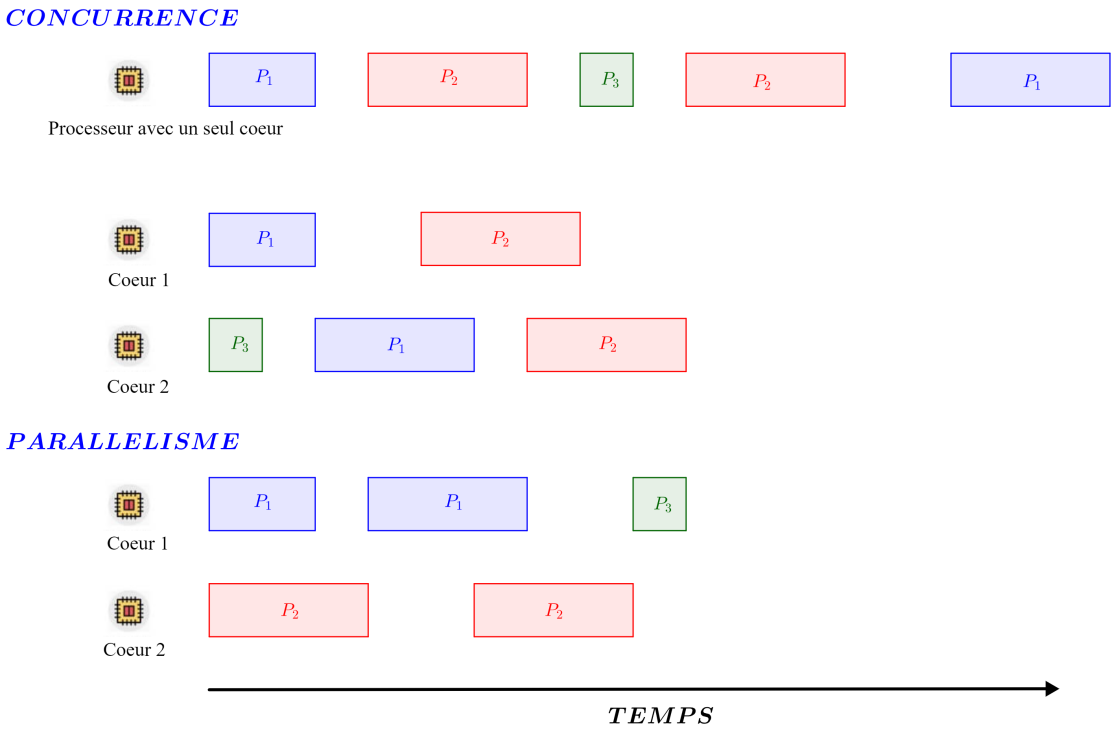
Le multithreading permet de traiter simultanément deux threads. Un processeur ne peut pas exécuter plus d'instructions simultanées qu'il n'a de coeurs. La simultanéité n'est qu'apparente. En réalité, le processeur exécute successivement et alternativement des threads de chaque processus.

4.1 Parallélisme et concurrence

Définition :
Un processeur est dit multicœurs lorsque plusieurs unités de calcul (cœurs) sont disposées côte à côte sur la même puce (processeur). Le support (la connectique qui relie le processeur à la carte électronique) lui ne change pas.

Définition :
On dit que deux processus s'exécutent en parallèle lorsqu'ils s'exécutent sur des CPU différents.

Définition :
On dit que deux processus sont concurrents lorsqu'ils concourent pour l'obtention d'une même ressource CPU. Leur exécution est alors entrelacée. Chacun dispose à son tour d'un quantum de temps calcul.



La programmation concurrente peut être réalisée avec un mono processeur ou un processeur multicœurs alors que le parallélisme implique un nombre de cœurs identique au nombre de tâches.

4.2 Ordonnanceur du système d'exploitation

Définition :
Dans les systèmes d'exploitation, l'ordonnanceur désigne le composant du noyau du système d'exploitation choisissant l'ordre d'exécution des processus sur les processeurs d'un ordinateur. En anglais, l'ordonnanceur est appelé scheduler.

Le fait que l'ordonnanceur interrompe un processus et sauve son état s'appelle une commutation de contexte. Il existe plusieurs types d'interruptions. Certaines sont générées par le matériel (par exemple le disque dur indique qu'il a fini d'écrire des octets, une carte réseau signale que des paquets de données arrivent ...). Lorsque le processeur reçoit une interruption, il interrompt son exécution à la fin de l'instruction courante et exécute un programme se trouvant à une adresse prédéfinie. Ce programme reçoit en argument une copie des valeurs courantes des registres, ainsi qu'un code numérique lui permettant de savoir à quel type d'interruption il fait face. Ce programme spécial s'appelle le gestionnaire d'interruption.
Afin de pouvoir choisir parmi tous les processus lequel exécuter lors de la prochaine interruption, le système d'exploitation conserve pour chaque processus une structure de données nommée PCB (pour l'anglais Process Control Bloc ou bloc de contrôle du processus). Le PCB est simplement une zone mémoire dans laquelle sont stockées diverses informations sur le processus.

Nom	Description
PID	Process ID, l'identifiant numérique du processus
Etat	l'état dans lequel se trouve le processus
Registres	La valeur des registres lors de sa dernière interruption
Mémoire	zone mémoire allouée par le processus lors de son exécution.
Ressources	Liste des fichiers ouverts, connexions réseaux en cours d'utilisation,...

L'ordonnanceur (ou scheduler) est le module du système d'exploitation qui se charge de l'organisation de l'exécution des processus. On appelle quantum de temps le plus petit temps d'exécution que l'OS peut attribuer à un thread. L'ordre de grandeur d'un quantum de temps est 10 ms. Il existe de nombreux algorithmes d'ordonnancement :

- first-come, first-served (FCFS)
- shortest job first (SJF)
- shortest remaining time (SRT)
- round robin (RR, algorithme du tourniquet)
- ordonnancement avec priorité

5 Gestion des ressources

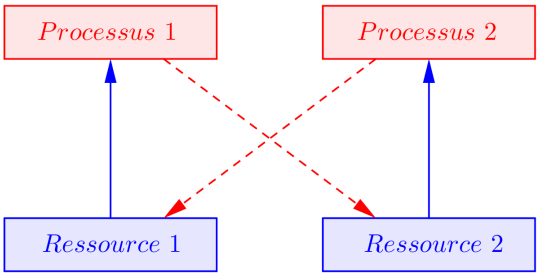
Les notions de programmation concurrente ou de parallélisme sont primordiales dans les systèmes d'exploitation, cela offre à un ordinateur la possibilité d'exécuter plusieurs tâches en parallèle, en fonction de ses performances physiques. Cette co-existence entre plusieurs processus doit être gérée de manière intelligente par le système afin d'éviter des dysfonctionnements Plusieurs types de ressources sont à la disposition d'un processus : RAM, disques, clés USB, fichiers, périphériques ..., on peut avoir deux ou plusieurs processus qui veulent accéder à la même ressource simultanément, créant ainsi des conflits. La synchronisation des ressources est donc nécessaire et consiste au fait d'utiliser des mutex ou verrous, qui sont tout simplement des jetons. On résume cette opération en 3 étapes :

- **Demande de la ressource :** la demande se fait en acquérant le mutex qui bloque la ressource. Si le mutex est indisponible, cela signifie que la ressource est en cours d'utilisation par un autre processus. Il faut donc attendre.
- **Utilisation de la ressource :** le processus peut utiliser la ressource tant qu'il a le mutex.
- **Libération de la ressource :** le processus débloque la ressource une fois qu'il a fini de l'utiliser en libérant le mutex.

5.1 Notion d'interblocage

Imaginons le scénario suivant :

- Un processus 1 détient le mutex pour utiliser une ressource 1 ;
- Un processus 2 détient le mutex pour utiliser une ressource 2 ;
- Le processus 1 attend la ressource 2 avant de libérer le mutex de la ressource 1 ;
- Le processus 2 attend la ressource 1 avant de libérer le mutex de la ressource 2.



Les processus sont bloqués dans cet état, il s'agit donc d'une situation d'impasse à éviter. La solution consiste à arrêter au moins un des processus concernés.
Cette situation s'appelle l'interblocage (deadlock en anglais), elle se produit lorsque des processus concurrents s'attendent mutuellement (Un processus peut aussi s'attendre lui-même).

5.2 Conditions pour un interblocage

Définition :
Une situation d'interblocage sur une ressource peut survenir si et seulement si toutes les conditions suivantes sont réunies simultanément dans un système :

- **Condition 1** : les ressources utilisées sont en **exclusion mutuelle** (un seul processus peut utiliser la ressource à un instant donné).
- **Condition 2** : Une ressource ne peut être libérée que volontairement par le processus qui la détient (**Non préemption**).
- **Condition 3** : Un processus détient actuellement au moins une ressource et demande des ressources supplémentaires qui sont détenues par d'autres processus (**Hold and wait** ou ressource holding).
- **Condition 4** : Chaque processus attend une ressource qui est détenue par un autre processus, qui à son tour attend que le premier processus libère la ressource (**Attente circulaire**).

Ces quatre conditions sont connues sous le nom de "conditions de Coffman" d'après leur première description dans un article de 1971 par Edward G. Coffman, Jr.

6 Processus et threads en Python

6.1 Générer des processus en python : le module `multiprocessing`

`multiprocessing` est un paquet qui permet l'instanciation de processus. Il autorise la programmation concurrente sur une même machine.

6.1.1 La classe `Process`

Dans le module `multiprocessing`, les processus sont instanciés en créant un objet de la classe `Process` et en appelant sa méthode `start()` :

```
p = Process(target = f, args = ('bob',))
```

- `target` : associe une fonction au processus
- `args` : la liste des arguments dans un tuple
- `p.start()` : lance le processus
- `p.join([timeout])` : attend l'arrêt du processus
- `p.name` : renvoie le nom du processus
- `pid` : renvoie l'ID du processus
- `is_alive` : renvoie vrai si le processus est en vie, faux sinon.
- `terminate()` : termine le processus.

```
"""Process : simple utilisation"""

from multiprocessing import Process
from os import getpid
from time import sleep

def ecriture():
    pid = str(getpid())
    with open("test.txt", "w") as fichier:
        for i in range(10**3):
            fichier.write("{} : {}\n".format(pid, i))
            fichier.flush()
            sleep(0.1)

if __name__ == '__main__':
    p1 = Process(target = ecriture, args = ())
    p2 = Process(target = ecriture, args = ())
    p3 = Process(target = ecriture, args = ())
    p1.start()
    p2.start()
    p3.start()
    p1.join()
    p2.join()
    p3.join()
```

 Process_simple.py

En observant l'exécution de ce script, on a l'impression que certaines lignes n'ont été écrites que par certains des processus et le choix du processus qui a écrit la ligne en question semble être aléatoire. Ils semblent s'être réparti les travaux d'écriture alors qu'ils fonctionnent indépendamment les uns des autres !

Chaque processus, lors de l'ouverture du fichier, enregistre la position du curseur d'écriture. À chaque écriture de caractères, ce curseur est avancé du nombre de caractères (octets en fait) correspondant.
Chaque fois qu'un processus est interrompu par l'ordonnanceur, il conserve en mémoire son état, en particulier la position du curseur au moment de l'interruption. Lorsqu'il est à nouveau appelé par la suite il reprend donc depuis cette position ... et peut donc écraser les entrées des autres processus si ces derniers étaient plus avancés.
Le fonctionnement multitâches d'un système d'exploitation est un avantage la plupart du temps mais soulève des problèmes lorsque des processus partagent des ressources car un processus ne sait pas qu'il est interrompu et reprend son travail dans l'état même dans son état au moment de l'interruption sans prendre en compte ce l'action des autres processus sur la ressource partagée.

6.1.2 La classe `Pool`

On peut créer un pool de processus qui exécuteront les tâches qui lui seront soumises avec la classe `Pool`. C'est un moyen pratique de paralléliser l'exécution d'une fonction sur plusieurs valeurs d'entrée, en répartissant les données d'entrée entre les processus.

- `p = Pool(processes = 4)` : est un ensemble de 4 processus sur lequel une fonction peut être appliquée.
- `map(func, itérable [, chunksize])` est un équivalent parallèle de la fonction intégrée `map()` (elle ne prend cependant en charge qu'un seul argument itérable). Il bloque jusqu'à ce que le résultat soit prêt. Cette méthode découpe l'itérable en un certain nombre de blocs qu'elle soumet au pool de processus en tant que tâches distinctes. La taille (approximative) de ces morceaux peut être spécifiée en définissant `chunksize` à un entier positif.

```python
"""pool_simple_map_application"""

from multiprocessing import Pool

def cube(x):
    return x**3

if __name__ == '__main__':
    p = Pool(processes=4)
    results = p.map(cube, range(1,7))
    print(results) #[1,8,27,64,125,216]
```

 Pool_simple_map_application.py

6.2 Générer des threads en python : le module `Threading`

Python nous propose dans sa bibliothèque standard plusieurs modules pour faire de la " programmation parallèle", c'est-à-dire que plusieurs instructions de code s'exécuteront en même temps, ou presque en même temps.

6.2.1 Un code linéaire

```python
from random import randint
from sys import stdout
from time import sleep

# Repete 20 fois
i = 0
while i < 20:
    stdout.write("1") #afficher le chiffre sur la sortie standard (l'écran, par défaut)
    stdout.flush()  #pour demander à Python d'afficher le chiffre tout de suite, sinon c'est à la fin d'exécution
    attente = 0.2
    attente += randint(1, 60) / 100 # on crée une variable attente et on la fait varier, grâce à random, entre 0.2 et 0.8
    # attente est à présent entre 0.2 et 0.8
    sleep(attente)
    i += 1
```

 code_lineaire.py

6.2.2 Premier exemple : définition de 2 threads

Nous allons regarder de plus près le module threading qui propose une interface simple pour créer des threads, c'est-à-dire des portions de notre code qui seront exécutées en même temps.
Pour créer un thread, il faut créer une classe qui hérite de `threading.Thread`. On peut redéfinir son constructeur et la méthode `run`.
Cette seconde méthode est appelée au lancement du thread et contient le code qui doit s'exécuter en parallèle du reste du programme.

```
from random import randint
from sys import stdout
from time import sleep
from threading import Thread

class Afficheur(Thread):

    """Thread chargé simplement d'afficher une lettre dans la console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 20:
            stdout.write(self.lettre)
            stdout.flush()
            attente = 0.2
            attente += randint(1, 60) / 100
            sleep(attente)
            i += 1

# Création des threads
thread_1 = Afficheur("1")
thread_2 = Afficheur("2")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

 exemple_1_threading.py

Comme vous le voyez, les deux threads s'exécutent en même temps. Puisque le temps de pause est variable, parfois on a un seul chiffre 1 qui s'affiche avant un chiffre 2, parfois on en a plusieurs. Au final, il y en a bien 20 de chaque.

6.2.3 Deuxième exemple : accès simultané à une même ressource

```
from random import randint
from sys import stdout
from time import sleep
from threading import Thread

class Afficheur(Thread):

    """Thread chargé simplement d'afficher une lettre dans la console."""

    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 5:
            for lettre in self.mot:
                stdout.write(lettre)
                stdout.flush()
                attente = 0.2
                attente += randint(1, 60) / 100
                sleep(attente)
            i += 1

# Création des threads
thread_1 = Afficheur("canard")
thread_2 = Afficheur("TORTUE")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

 exemple_2_threading.py

Comme vous le voyez, nos mots sont complètement mélangés, ce qui n'est pas bien surprenant. Vous pouvez toujours suivre la partie en majuscule ou minuscule et vérifier que les mots s'affichent bien, mais puisque nous écrivons sur la même ressource partagée (la console, ici), le résultat s'affiche mélangé.

6.3 Conclusion

Le **GIL** ou **Global Interpreter Lock** est un verrou unique auquel l'interpréteur Python fait appel constamment pour protéger tous les objets qu'il manipule contre des accès concurrentiels : un thread à la fois accède à l'interpréteur.
Le module threading passe par le GIL (Global Interpreter Lock) ce qui le rend inefficace pour du calcul scientifique.
L'API multiprocessing contourne le GIL en faisant appel à des sous-processus.

7 Exercices

Exercice n° 1.

On considère la fonction `hello` suivante.

```
def hello(n):
    for i in range(5):
        print ("Je suis le thread", n, "et ma valeur est", i)
    print ("------ Fin du Thread ", n)
```

 exercice_1.py

Écrire une boucle générant 4 threads avec le module `threading` pointant chacun vers la fonction `hello` et avec `args` un tuple contenant l'argument passé à la fonction.

Exercice n° 2.

Allons un peu plus loin dans les processus parent et fils :

```
from multiprocessing import current_process, Process
from os import getpid

def info(title):
    print(title)
    print('module name :', __name__)
    print('parent process :', getpid())
    print('process id :', getpid())

def f(num):
    my_p = current_process() #objet représentant le process
    print('Hello:', num, my_p.pid, my_p._parent_pid, my_p.name)

if __name__ == "__main__":
    info('main program')
    num = 0
    p = Process(target=f, args=(num,))
    p.start()
    print('\nprocess name :', p.name)
    print('process id :', p.pid)
    p.join()
```

 exercice_2.py

Commenter ce script.

Exercice n° 3.

On considère le programme suivant :

```
from threading import Thread
from time import sleep

def incr():
    global compteur
    for _ in range(10**5):
        compteur += 1

compteur = 0
thread_1 = Thread(target = incr, args = ())
thread_2 = Thread(target = incr, args = ())
thread_1.start()
thread_2.start()

while thread_1.is_alive() and thread_2.is_alive():
    sleep(1)
print("valeur finale", compteur)
```

 exercice_3.py

Pour chacune des affirmations suivantes, dire si elle est vraie ou fausse et justifier.

1. Le programme affiche toujours 200 000.
2. Le programme peut afficher un nombre plus petit que 200 000.
3. Le programme peut afficher un nombre plus grand que 200 000.

4. Si l'on supprime la boucle `while thread_1.is_alive() and thread_2.is_alive(): sleep(1)` et si on ajoute `thread_i.joi` après `thread_i.start()`, le programme affiche toujours 200 000.
5. Le script suivant affiche toujours 200 000, expliquer pourquoi.

```
from threading import Thread, Lock
from time import sleep

def incr():
    global compteur
    for _ in range(10**5):
        verrou.acquire()
        compteur += 1
        verrou.release()

compteur = 0
verrou = Lock()
thread_1 = Thread(target = incr, args = ())
thread_2 = Thread(target = incr, args = ())
thread_1.start()
thread_2.start()

while thread_1.is_alive() and thread_2.is_alive():
    sleep(1)
print("valeur finale", compteur)
```

 exercice_3_bis.py

Exercice n° 4.

On présente le script suivant :

```
from threading import Thread, Lock
from time import sleep

verrou1 = Lock()
verrou2 = Lock()

def f1():
        verrou1.acquire()
        print ("Section critique 1.1")
        verrou2.acquire()
        print ("Section critique 1.2")
        verrou2.release()
        verrou1.release()

def f2():
        verrou2.acquire()
        print ("Section critique 2.1")
        verrou1.acquire()
        print ("Section critique 2.2")
        verrou1.release()
        verrou2.release()

t1 = Thread(target=f1, args = ())
t2 = Thread(target=f2, args = ())
t1.start ()
t2.start ()
```

 exercice_4.py

A quelle condition peut-il présenter une situation d'interblocage ?
Pour le vérifier pratiquement, ne pas hésiter à insérer des `time.sleep()` après chaque affichage.

Exercice n° 5.

On se propose de reprendre le fichier `exemple_2_threading.py` et d'ajouter un verrou de manière à ce que les mots `canard` et `TORTUE` s'écrivent toujours en "entier" (sans se mélanger).

Exercice n° 6.

On considère la situation de la figure ci-dessous, dans laquelle quatre voitures sont bloquées à une intersection. Montrer qu'il s'agit d'un inter blocage, c'est à dire que les quatre conditions de Coffman sont réunies. On indiquera précisément quelles sont les ressources et les processus dans cette situation. On fera l'hypothèse que les conducteurs sont raisonnables, qu'ils ne veulent pas provoquer d'accident.

Exercice n° 7.

On donne le fichier `traitement_image.py`, il comporte trois fonctions

- `couleur_moyenne(img)` qui renvoie la couleur moyenne de l'ensemble des pixels de l'image passée en paramètre.
- `cadre(img)` qui renvoie l'image passée en paramètre en lui ajoutant autour un cadre de la couleur moyenne de cette image
- `traitement(nom_img)` ouvre l'image stockée dans le fichier `nom_img` et renvoie l'image encadrée générée par la fonction précédente.

Le programme ci-dessous appelle le module `traitement_image.py` et génère le cadre sur 20 images et les affiche une par une :

```
from PIL import Image
from traitement_image import cadre, couleur_moyenne
from time import time

def charger_images():
    liste_images = []
    for i in range(1, 21):
        liste_images.append(Image.open(str(i)+'.jpg'))
    return liste_images

def traiter_les_images(debut, fin):
    liste_images = charger_images()
    liste_images_retravaillees = []

    for img in liste_images[debut : fin]:
        imageRetravaille = cadre(img)
        imageRetravaille.show()
        liste_images_retravaillees.append(imageRetravaille)
    return liste_images_retravaillees

if __name__ == '__main__':
    Début = time()
    traiter_les_images(1, 21)
    print(time()- Début)
```

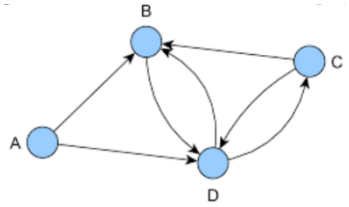
exercice_7.py

1. (a) Observer le temps de calcul pour terminer le travail.
   (b) Ouvrir le gestionnaire de tâches (CTRL + ALT + SUPPR), observer les processus en action.
   (c) Dans l'onglet "performance", cliquer sur "Ouvrir le moniteur de ressources" et observer la charge des différents processeurs.
2. Modifier le fichier de manière à générer deux processus (avec le module `mutiprocessing`, l'un gérant les 10 premières photos, le second les dix dernières).
   Reprendre les questions précédentes.
3. Faire de même en utilisant des threads avec le module `threading`.
   Reprendre la première question.

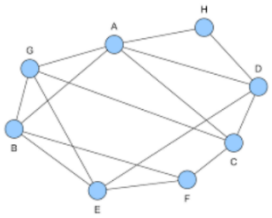
Exercice n° 8.

Dans un graphe, on distingue les objets appelés sommets et les relations entre sommets. Il existe deux types de graphes :

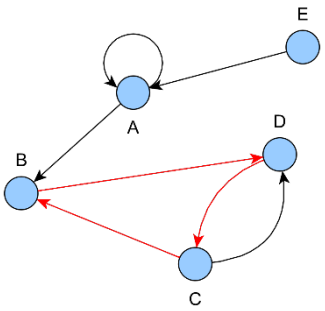
Les graphes orientés où les relations sont appelées arcs.



Les graphes non orientés où les relations sont appelées arêtes.



On note $A \to B$ l'arc (A, B) d'un graphe orienté où A est le sommet de départ et B celui d'arrivée.
On appelle chemin, toute suite de sommets consécutifs reliés par des arcs, il est dit élémentaire si elle ne comporte pas plusieurs fois le même sommet.
On appelle circuit, un chemin dont le sommet de début et le sommet de fin sont identiques.



Sept processus P_i ($i \in \mathbb{N}$ et $1 \leqslant i \leqslant 7$) sont dans la situation suivante par rapport aux ressources R_i :
- P_1 a obtenu R_1 et demande R_2 ;
- P_2 demande R_3 et n'a obtenu aucune ressource ;
- P_3 demande R_2 et n'a obtenu aucune ressource ;
- P_4 a obtenu R_2 et R_4 et demande R_3 ;
- P_5 a obtenu R_3 et demande R_5 ;
- P_6 a obtenu R_6 et demande R_2 ;
- P_7 a obtenu R_5 et demande R_2.

On voudrait savoir s'il y a interblocage.
Construire un graphe orienté où les sommets sont les processus et les ressources, tels que :
- la présence de l'arc $R_i \to P_j$ signifie que le processus P_j a obtenu la ressource R_i ;
- la présence de l'arc $P_j \to R_i$ signifie que le processus P_j demande la ressource R_i.

Il y a interblocage lorsque des circuits sont présents dans le graphe.
Chercher ces circuits afin de déterminer s'il y a bien interblocage ou non.

Exercice n° 9. Interblocage et base de données

Si deux transaction travaillent sur un même objet (par exemple sur une même table), alorsla seconde est bloquée jusqu'à ce que la première soit terminée.
Nous pouvons maintenant expliciter ce mécanisme : lors qu'une transaction accède à une table, elle tente de prendre un verrou sur cette dernière. Les verrous sont relâchés au moment du COMMIT ou ROLLBACK. On considère deux tables :

```
CREATE TABLE T(num INTEGER);
CREATE TABLE S (num INTEGER);
INSERT INTO T VALUES (1000);
INSERT INTO S VALUES (1000);
```

qui peuvent représenter de façon simplifiée des comptes en banque. Considérons les deux transactions suivantes :

```
START TRANSACTION;
  UPDATE T SET num = num+100;
  UPDATE S SET num = num-100;
COMMIT;
```

```
START TRANSACTION;
  UPDATE S SET num = num+100;
  UPDATE T SET num = num-100;
COMMIT;
```

La première simule un virement de S vers T et la seconde un virement de T vers S. Montrer qu'il s'agit d'un inter blocage, c'est à dire que les quatre conditions de Coffman sont réunies.

Exercice n° 10.

Les parties A et B peuvent être traitées indépendamment.

Partie A :

Dans un bureau d'architectes, on dispose de certaines ressources qui ne peuvent être utilisées simultanément par plus d'un processus, comme l'imprimante, la table traçante, le modem.

Chaque programme, lorsqu'il s'exécute, demande l'allocation des ressources qui lui sont nécessaires. Lorsqu'il a fini de s'exécuter, il libère ses ressources.

Programme 1	Programme 2	Programme 3
demander (table traçante)	demander (modem)	demander (imprimante)
demander (modem)	demander (imprimante)	demander (table traçante)
exécution	exécution	exécution
libérer (modem)	libérer (imprimante)	libérer (table traçante)
libérer (table traçante)	libérer (modem)	libérer (imprimante)

On appelle p_1, p_2 et p_3 les processus associés respectivement aux programmes 1, 2 et 3.

1. Les processus s'exécutent de manière concurrente.
   Justifier qu'une situation d'interblocage peut se produire.
2. Modifier l'ordre des instructions du programme 3 pour qu'une telle situation ne puisse pas se produire. Aucune justification n'est attendue.
3. Supposons que le processus p_1 demande la table traçante alors qu'elle est en cours d'utilisation par le processus p_3. Parmi les états suivants, quel sera l'état du processus p_1 tant que la table traçante n'est pas disponible : a) élu b) bloqué c) prêt d) terminé

Partie B :

Avec une ligne de commande dans un terminal sous Linux, on obtient l'affichage suivant :

```
UID        PID  PPID  C STIME TTY          TIME CMD
...
pi        6211   831  8 09:07 ?        00:01:16 /usr/lib/chromium-browser/chromium-browser-v7 --disable-quic --enable-tcp-fast-open --p
pi        6252  6211  0 09:07 ?        00:00:00 /usr/lib/chromium-browser/chromium-browser-v7 --type=zygote --ppapi-flash-path=/usr/lib
pi        6254  6252  0 09:07 ?        00:00:00 /usr/lib/chromium-browser/chromium-browser-v7 --type=zygote --ppapi-flash-path=/usr/lib
pi        6294  6211  4 09:07 ?        00:00:40 /usr/lib/chromium-browser/chromium-browser-v7 --type=gpu-process --field-trial-handle=1
pi        6300  6211  1 09:07 ?        00:00:16 /usr/lib/chromium-browser/chromium-browser-v7 --type=utility --field-trial-handle=10758
pi        6467  6254  1 09:07 ?        00:00:11 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       11267  6254  2 09:12 ?        00:00:15 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       12035   836  0 09:13 ?        00:00:00 /usr/lib/libreoffice/program/oosplash --writer file:///home/pi/Desktop/mon_fichier.odt
pi       12073 12035  2 09:13 ?        00:00:15 /usr/lib/libreoffice/program/soffice.bin --writer file:///home/pi/Desktop/mon_fichier.o
pi       12253   831  1 09:13 ?        00:00:07 /usr/bin/python3 /usr/bin/sense_emu_gui
pi       20010  6211  1 09:21 ?        00:00:00 /usr/lib/chromium-browser/chromium-browser-v7 --type=utility --field-trial-handle=10758
pi       20029  6254 56 09:21 ?        00:00:28 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       20339  6254  4 09:21 ?        00:00:01 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       20343  6254  2 09:21 ?        00:00:00 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       20464  6211 17 09:22 ?        00:00:00 /proc/self/exe --type=utility --field-trial-handle=10758631334788949 17,6306120996223181
pi       20488  6254 14 09:22 ?        00:00:00 /usr/lib/chromium-browser/chromium-browser-v7 --type=renderer --field-trial-handle=1075
pi       20519   676  0 09:22 pts/0    00:00:00 ps -ef
```

La documentation Linux donne la signification des différents champs :

- UID : identifiant utilisateur effectif ;
- PID : identifiant de processus ;
- PPID : PID du processus parent ;
- C : partie entière du pourcentage d'utilisation du processeur par rapport au temps de vie des processus ;
- STIME : l'heure de lancement du processus ;
- TTY : terminal de contrôle
- TIME : temps d'exécution
- CMD : nom de la commande du processus

1. Parmi les quatre commandes suivantes, laquelle a permis cet affichage ?
   a) ls -l b) ps –ef c) cd .. d) chmod 741 processus.txt
2. Quel est l'identifiant du processus parent à l'origine de tous les processus concernant le navigateur Web (chromium-browser) ?
3. Quel est l'identifiant du processus dont le temps d'exécution est le plus long ?

Exercice n° 11.

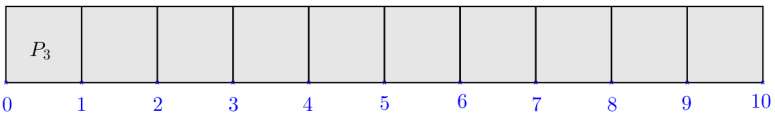
Partie A :

1. Un processeur choisit à chaque cycle d'exécution le processus qui doit être exécuté. Le tableau ci-dessous donne pour trois processus P_1, P_2, P_3 :
   - la durée d'exécution (en nombre de cycles),
   - l'instant d'arrivée sur le processeur (exprimé en nombre de cycles à partir de 0),
   - le numéro de priorité.

   Le numéro de priorité est d'autant plus petit que la priorité est grande. On suppose qu'à chaque instant, c'est le processus qui a le plus petit numéro de priorité qui est exécuté, ce qui peut provoquer la suspension d'un autre processus, lequel reprendra lorsqu'il sera le plus prioritaire.

Processus	Durée d'exécution	Instant d'arrivée	Numéro de priorité
P_1	3	3	1
P_2	3	2	2
P_3	4	0	3

   Reproduire le tableau ci-dessous sur la copie et indiquer dans chacune des cases le processus exécuté à chaque cycle.



2. On suppose maintenant que les trois processus précédents s'exécutent et utilisent une ou plusieurs ressources parmi R_1, R_2 et R_3.

   Parmi les scénarios suivants, lequel provoque un interblocage ? Justifier.

Scénario 1	Scénario 2	Scénario 3
P1 acquiert R1	P1 acquiert R1	P1 acquiert R1
P2 acquiert R2	P2 acquiert R3	P2 acquiert R2
P3 attend R1	P3 acquiert R2	P3 attend R2
P2 libère R2	P1 attend R2	P1 attend R2
P2 attend R1	P2 libère R3	P2 libère R2
P1 libère R1	P3 attend R1	P3 acquiert R2

Partie B :

Dans cette partie, pour une meilleure lisibilité, des espaces sont placées dans les écritures binaires des nombres. Il ne faut pas les prendre en compte dans les calculs.

Pour chiffrer un message, une méthode, dite du masque jetable, consiste à le combiner avec une chaîne de caractères de longueur comparable. Une implémentation possible utilise l'opérateur `XOR` (ou exclusif) dont voici la table de vérité :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Dans la suite, les nombres écrits en binaire seront précédés du préfixe `0b`.

1. Pour chiffrer un message, on convertit chacun de ses caractères en binaire (à l'aide du format Unicode), et on réalise l'opération `XOR` bit à bit avec la clé.

   Après conversion en binaire, et avant que l'opération `XOR` bit à bit avec la clé n'ait été effectuée, Alice obtient le message suivant : `m = 0b 0110 0011 0100 0110`

(a) Le message `m` correspond à deux caractères codés chacun sur 8 bits : déterminer quels sont ces caractères. On fournit pour cela la table ci-dessous qui associe à l'écriture hexadécimale d'un octet le caractère correspondant (figure 2). Exemple de lecture : le caractère correspondant à l'octet codé `4A` en hexadécimal est la lettre J.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 2

(b) Pour chiffrer le message d'Alice, on réalise l'opération `XOR` bit à bit avec la clé suivante : `k = 0b 1110 1110 1111 0000`

Donner l'écriture binaire du message obtenu.

2. (a) Dresser la table de vérité de l'expression booléenne suivante : `(a XOR b) XOR b`

(b) Bob connaît la chaîne de caractères utilisée par Alice pour chiffrer le message. Quelle opération doit-il réaliser pour déchiffrer son message ?