Exercices de TP autour des processus en Python

Nicolas Louvet

1 Programmation avec des processus

Le but est d'expérimenter avec les possibilités proposées dans les modules standards de Python 3 pour programmer avec des processus. Les exercices proposées sont très scolaires, et permettent de rappeler quelques problèmes classiques : exécution concurrente, échange de données entre processus, accès concurrents à la mémoire... On commence par voir rapidement comment les appels POSIX peuvent être utilisés sous Linux via le module `os` [1], puis on passe au module multiprocessing [2], qui est très bien fourni en ce qui concerne la communication et la synchronisation des processus, et qui permet également de travailler sous Windows.

1.1 Module `os`

Lorsque l'on travaille sous Linux, il est possible d'utiliser les appels POSIX de création de processus, en utilisant le module `os`. Vous pouvez expérimenter par exemple avec le programme suivant (si vous êtes sous Windows 10, il *devrait* fonctionner avec le Windows Subsystem for Linux).

```
1  import os, time
2
3  pid = os.fork()
4  if pid > 0: # processus père
5      print('Père de PID ', os.getpid())
6      print('Père : j\'attends mon fils')
7      os.wait()
8      print('Père : mon fils est terminé')
9      exit(0)
10 else: # pid == 0, processus fils
11     print('Fils de PID ', os.getpid())
12     time.sleep(1)
13     print('Fils : je me termine')
14     exit(0)
```

Notez que le module `os` ne met pas `fork()` à votre disposition sous Windows, ni même une interface pour l'appel `CreateProcess()` (l'appel de l'API C/C++ de Windows pour créer un processus). Il n'est donc pas possible d'écrire du code utilisant les processus qui soit portable avec ce module.

EXERCICE 1 ▶ **documentation et premières exprériences autour de** `os.fork()`

1) Rappelez rapidement le rôle et le fonctionnement (ne notez pas les spécifications précises, elles sont dans la documentation!) des méthodes `os.fork()`, `os.getpid()`, `os.wait()`.

2) Modifiez le programme de façon à ce que le fils affiche l'identifiant de son père.

3) Les processus père et fils partagent-ils la même mémoire? Modifiez le programme pour vérifier votre réponse.

4) Que se passe-t-il quand le processus père se termine avant le processus fils? Modifiez le programme de façon à observer ce qui se passe dans l'arborescence des processus dans ce cas (utilisez les commandes `ps` et `pstree`).

1.2 Module `multiprocessing`

Il est possible de faire les choses d'une façon plus portable avec le module multiprocessing, dont vous êtes invités à consulter la documentation. Voici une nouvel exemple dans lequel le processus principal crée un fils.

```
1  import multiprocessing as mp, time
2
3  def fct_p(): # fonction qui sera exécutée par le fils
4      pid = mp.current_process().pid
5      print("Fils : pid =", pid)
6      time.sleep(10)
7      print("Fils : je me termine")
8
9  if __name__ == '__main__':
10     pid = mp.current_process().pid
11     p = mp.Process(target=fct_p) # création du fils
```

```
12      p.start()                  # lancement du fils
13      print("Père : pid = ", pid)
14      print("Père : pid du fils =", p.pid)
15      print("Père : j'attends mon fils...")
16      p.join()                   # attente du fils
17      print("Père : mon fils est terminé")
```

EXERCICE 2 ▶ **documentation et premières expériences autour de** `multiprocessing.Process()`

1) Résumez le fonctionnement du constructeur `Process()`, et des méthodes `start()` et `join()` de la classe `Process`.

Tout cela est expliqué dans la documentation du module : je ne résumé pas ici...

2) Que retourne la fonction `current_process()` du module `multiprocessing`?

`mp.current_process()` retourne une référence au processus courant.

3) L'attribut `pid` permet d'obtenir l'identifiant d'un processus sur le système. A partir de quelle version de Python pouvez vous afficher, avec une méthode de `multiprocessing`, l'identifiant du processus parent d'un processus?

En regardant la documentation pour `mp.current_process()`, on peut voir celle pour `mp.current_process()` en dessous : cela retourne une référence au processus parent du courant. Mais la documentation indique bien que c'est une nouveauté dans la version 3.8 de Python!

4) Est-il possible, avec la méthode `join()`, qu'un processus attende sa propre terminaison? Que se passe-t-il alors?

C'est possible en utilisant l'objet donné par `mp.current_process()`, qui est une référence au processus courant. On peut essayer dans le programme par exemple avec `mp.current_process().join()`. Le problème est qu'il y a un interblocage : le processus se retrouve à l'exécution à attendre un événement (sa terminaison), qui ne se produira jamais. Heureusement, une exception `AssertionError: can only join a child process` est levée, qui indique que cela n'a pas de sens qu'un processus s'attende lui-même.

5) Différentes méthodes de lancement des processus peuvent être configurées avec `mp.set_start_method()` : `'fork'`, `'spawn'` et `'forkserver'`. Pourquoi? Quelle est la méthode utilisée par défaut sur votre système?

La méthode `'fork'` est très spécifique aux systèmes POSIX, donc essentiellement à Linux, mais peut marcher sur MacOSX. La méthode `'spawn'` va fonctionner aussi sous Windows. Par défaut, c'est la méthode `'fork'` qui est utilisée sous Linux, et `'spawn'` sous Windows ou MacOSX. On peut le vérifier avec `mp.get_start_method()` : tout cela est expliqué dans la documentation du module!

6) Les processus père et fils partagent-ils la même mémoire? Modifiez le programme pour vérifier votre réponse; expérimentez si possible avec les méthodes `'fork'` et `'spawn'`.

La réponse est : non, les processus créés reçoivent bien une copie de l'état de la mémoire du processus père. Par contre, il existe des différences entre les méthodes `'fork'` et `'spawn'`, qui ne sont pas super bien documentées :

— `'fork'` : le fils reçoit une copie de la mémoire du père lors de la création avec processus avec l'appel à `start()`. C'est vraiment un appel à la primitive POSIX `fork()`.

— `'spawn'` : un nouvel interpréteur Python est lancé, et il reçoit une copie des données nécessaires à l'exécution du code du fils grâce au module de sérialisation d'objets `Pickle` (voir la documentation du module ici, ainsi qu'une page qui donne des explications sur la méthode `'spawn'` là. En expérimentant, on peut constater que la copie est vraisemblablement effectuée, lors de l'appel au constructeur `Process()`.

Dans tous les cas, retenez qu'il n'est pas recommandable de modifier des variables globale entre la construction du processus et son lancement!

Pour expérimenter, on peut essayer (comme avec `os.fork()`) en prenant un objet mutable et un autre non-mutable, tous les deux définis de façon globale. Notez que le processus père modifie sournoisement les objets entre l'appel au constructeur `Process()` et la méthode `start()`, de façon à mettre en évidence les changements de comportements entre `'fork'` et `'spawn'`.

```
1  import multiprocessing as mp, time, array
2
3  n = 4 # un objet non-mutable
4  t = array.array('i', [1]) # un objet mutable
5
6  def fct_p():
7      global n, t
```

```
    pid = mp.current_process().pid
    print("Fils : pid =", pid)
    print("Fils : je trouve n = {} et t[0] = {}".format(n, t[0]))
    n = 64 # modifications de n et
    t[0] += 1 # de t[0] chez le fils
    print("Fils : je mets n = {} et t[0] = {}".format(n, t[0]))
    print("Fils : je me termine")

if __name__ == '__main__':
    mp.set_start_method('spawn')
    print("Père : initialement n = {} et t[0] = {}".format(n, t[0]))
    pid = mp.current_process().pid
    p = mp.Process(target=fct_p)
    n = 42 # le père change sournoisement les
    t[0] += 16 # valeurs de n et de t[0] !
    p.start()
    print("Père : pid = ", pid)
    print("Père : pid du fils =", p.pid)
    print("Père : j'attends mon fils...")
    p.join()
    print("Père : mon fils est terminé")
    print("Père : au final n = {} et t[0] = {}".format(n, t[0]))
```

On pourrait aussi essayer de passer un objet mutable en argument du processus lors de sa création... Nous n'essayons pas ici, mais les résultats seraient les mêmes !

7) Modifiez le programme de façon à observer ce qui se passe quand le processus père se termine avant le processus fils ?

EXERCICE 3 ▶ **Lancement d'un ensemble de processus**

1) Écrivez un programme dans lequel le processus principal crée 10 processus, chacun recevant un indice différent entre 0 et 9, puis lance chacun de ces processus, et enfin les attend. Une fois lancé, chacun de ces processus se met en attente passive pendant une seconde, puis affiche son propre indice avant de se terminer.

2) Assurez-vous que les 10 processus créés s'exécutent bien de façon concurrente.

3) Sous Linux, essayez les méthodes `fork` et `spawn` pour le lancement des processus : observez-vous une différence à l'exécution avec 10 processus ? Modifiez votre programme pour qu'il lance 100 processus, et mesurer son temps d'exécution total avec la commande `time` du shell, selon que vous utilisez la méthode `fork` ou la méthode `spawn` : observez-vous maintenant une différence ?

```
import multiprocessing as mp, time

def fct_proc(i):
    pid = mp.current_process().pid
    print("Fils {} (pid = {}): je commence".format(i, pid))
    time.sleep(10)
    print("Fils {} (pid = {}): je me termine".format(i, pid))

if __name__ == '__main__':
    mp.set_start_method('fork')
    print("start_method =", mp.get_start_method())
    procs = []
    # création des 10 processus
    for i in range(100):
        p = mp.Process(target=fct_proc, args=(i,))
        procs.append(p)
    # lancement des processus créés
    for p in procs:
        p.start()
    # attente des processus lancés
    for p in procs:
      print("Père : j'attends un fils...")
      p.join()
```

```
    print('Père : je me termine')
```

Le constructeur `Process()` permet de passer différents arguments aux processus lancés : comme on vient de le voir, le processus principal peut ainsi envoyer des données différenciées à chacun de ces fils. Mais il peut être utile également de permettre aux fils d'envoyer des données au père, et aux processus créés d'échanger des données entre-eux. Le module `multiprocessing` fournit différentes techniques, et on va s'intéresser à deux d'entre elles :

— la classe `multiprocessing.Queue`,
— le partage de `ctypes` avec `multiprocessing.Value()`.

EXERCICE 4 ▶ **File partagée entre processus**

Le constructeur `multiprocessing.Queue()` crée une file qui peut être partagée entre différents processus (les différentes problèmes d'accès concurrents à la file sont déjà réglés pour vous).

Avec les files de cette classe, il est très simple de réaliser des échanges de données d'un processus à un autre selon le modèle producteur-consommateur. Il suffit pour cela d'utiliser deux méthodes de la classe :

— `put(obj)` permet de placer `obj` en queue de file (l'appel est bloquant tant que la file est pleine),
— `get()` retire de la file l'objet en tête, et le retourne (l'appel est bloquant tant qu'aucune donnée n'est disponible).

On occulte ici pas mal de détails : consulter la documentation pour des spécifications plus précises !

Écrivez un programme dans lequel le processus principal crée N processus (définissez une constante `N` à 10 par exemple dans votre programme), que l'on nomme $P_0, \ldots, P_{N-1}$. Pour tout i tel que $0 \leq i \leq N-2$, le processus P_i dispose d'une file Q_{i+1} pour pouvoir envoyer un entier à P_{i+1}. De plus, le processus principal envoie l'entier 1 à P_0 via une file Q_0, et reçoit un entier de la part de P_{N-1} via Q_N. Lorsqu'un processus P_i reçoit un entier, il le multiplie par deux, reste une seconde en attente passive, puis envoie le résultat à P_{i+1}.

Vous ajouterez tous les affichages nécessaires (dans chacun des processus) pour qu'il soit possible de bien suivre l'exécution du programme.

```
import multiprocessing as mp, time

N = 10 # nombre de processus qui seront lancés

def fct_proc(i, qin, qout):
    v = qin.get()
    print("Fils {} : j'ai reçu {}".format(i, v))
    v *= 2
    time.sleep(1)
    print("Fils {} : je passe {} au suivant".format(i, v))
    qout.put(v)

if __name__ == '__main__':
    mp.set_start_method('spawn')
    print("start_method =", mp.get_start_method())
    q = []
    for i in range(N+1):
        q.append(mp.Queue())
    procs = []
    for i in range(N):
        p = mp.Process(target=fct_proc, args=(i, q[i], q[i+1]))
        procs.append(p)
    for p in procs:
        p.start()

    v = 1
    print("Père : j'envoie {} au premier fils".format(v))
    q[0].put(v);
    v = q[N].get()
    print("Père : le dernier fils m'envoye {}".format(v))

    for p in procs:
        p.join()
    print('Père : tous mes fils sont terminés, je me termine')
```

EXERCICE 5 ▶ **Zone de mémoire partagées entre processus** Le module `multiprocessing` fournit deux classes permettant de partager des données natives entres des processus : ce sont les classes `Value` et `Array`. Dans cet exercice, vous allez créer

une variable partagée `count` de type `int` avec `Value('i')` ; vous pourrez alors accéder à variable (en lecture ou en écriture), via l'attribut `value` de `count`.

La documentation précise bien qu'une opération de la forme `count.value += 1` ne se fait pas de façon atomique. En effet, une telle opération se décompose en (au moins) trois instructions : une lecture en mémoire, une incrémentation et une écriture en mémoire. Rien ne garantit a priori que, lorsque processus est en train d'exécuter ces trois instructions il est le seul à accéder à l'emplacement mémoire concerné. C'est en cela que l'accès n'est pas atomique.

Vous devez précisément mettre en évidence ce problème ! Vous allez écrire un programme qui lance N processus (il faudra essayer avec différentes valeur de N). Vous créez une variable `count`, initialisée à 0, partagée entre tous les processus, et que chaque processus doit essayer d'incrémenter. Une fois tous les processus terminés, le programme principal affiche `count`. Si chaque processus manipulait `count` de façon atomique, vous devriez avoir au final `count` $= N$: vérifiez que ça n'est pas toujours le cas (il faudra prendre N un peu grand, genre 50 ou 100) !

```
1  import multiprocessing as mp, time
2
3  N = 3 # nombre de processus qui seront lancés
4
5  def fct_proc(i, v):
6      pid = mp.current_process().pid
7      time.sleep(10000)
8      v.value = v.value + 1
9      print("Fils {} (pid = {}): je me termine".format(i, pid))
10
11 if __name__ == '__main__':
12     # mp.set_start_method('spawn')
13     # print("start_method =", mp.get_start_method())
14     v = mp.Value('i', 0)
15     procs = []
16     # création des N processus
17     for i in range(N):
18         p = mp.Process(target=fct_proc, args=(i,v))
19         procs.append(p)
20     # lancement des processus créés
21     for p in procs:
22         p.start()
23     # attente des processus lancés
24     for p in procs:
25       print("Père : j'attends un fils...")
26       p.join()
27     print("Père : je me termine")
28     print("Père : counter = {} ({} attendu)".format(v.value, N))
```

1.3 Module subprocess

EXERCICE 6 ▶ **Zone de mémoire partagées entre processus** Vous allez écrire un programme qui ouvre n'importe quel fichier en choisissant le meilleur logiciel pour cela.

Plus exactement, ce programme prend en parametre un nom de fichier. Il utilise une première commande linux `file` qui permet d'obtenir le type mime du fichier (une description de ce qu'il contient), récupère le résultat de la commande, et à partir de ce type choisi l'application permettant de l'ouvrir.

1) écrivez un programme qui prend un parametre obligatoire (le nom de fichier) et un parametre optionnel `--debug` [1].

2) appelez la commande find avec `subprocess.run` de manière à récupérer le retour de cette commande et extrayez le type mime. La commande `find` doit etre appelée de la manière suivante :

```
find -i nomdufichier
```

En mode debug affichez le résultat de la commande

3) récupérez le type mime du résultat et selon la valeur lancer une 2^e commande permettant de l'ouvrir. Par exemple (en fonction des logiciels présents sur votre ordinateur)
— un fichier text doit etre ouvert avec la commande `code` ou `geany`
— un fichier pdf avec la commande `evice` ou `okular`

1. profitez en pour utiliser le module `argparse`

— un fichier video avec la commande `vlc`
— ...

```python
#! /usr/bin/env python3
import argparse
import os
import sys
import subprocess as sub
import re









if __name__ == '__main__':

    parser = argparse.ArgumentParser(description="Ouvre un document en cherchant le type du document puis e
    parser.add_argument("-d", "--debug", dest="debug", required=False, default=False, help="debug level DEF

    parser.add_argument(dest="file", help="Fichier àouvrir")

    args = parser.parse_args()


    com = "file"
    if args.debug:
        print(f"Ouverture du fichier {args.file}")

    pf = sub.run([com, "-i", args.file], capture_output=True)
    if pf.returncode != 0:
        print(f"La commande {com} lancée avec les arguments {pf.args} a eu l'erreur {pf.returncode} la sort
        sys.exit(1)

    m = re.match(f"^{args.file}: *([^;]+);.*$", pf.stdout.decode())
    if not(m):
        print(f"Impossible de retrouver le type mime de {args.file} dans le résultat {pf.stdout.decode()}")
        sys.exit(1)
    mimetype = m.group(1)

    if args.debug:
        print(f"Le type trouvé est '{mimetype}'")


    if re.match(r"^application/pdf.*$", mimetype):
        sub.run(["evince", args.file])
    elif re.match(r"^text/.*$", mimetype):
        sub.run(["code", args.file])
    elif re.match(r"^video/.*$", mimetype):
        sub.run(["vlc", args.file])
    else:
        print(f"Impossible de trouver un logiciel pour {mimetype}")
        sys.exit(1)
```

2 Un exemple d'application

Le petit Nicolas veut savoir comment calculer le volume d'une sphère en dimension $\texttt{dim} \geq 1$ et de rayon 1. Il en parle à Agnan, le premier de la classe, qui évidemment se lance dans des calculs longs et compliqués... Joachim suggère à Nicolas de faire un petit programme pour estimer le volume de la sphère : en générant uniformément `N` points dans le pavé $[-1,1]^{\texttt{dim}}$, si

Nin de ces points sont à une distance au plus 1 de l'origine, Joachim se dit que le volume de la sphère peut être approché par

$$2^{dim} \times \frac{\texttt{Nin}}{\texttt{N}}$$

quand le nombre d'échantillons N est assez grand. Nicolas fait d'ailleurs remarquer que la pertinence de la technique pourra au moins être vérifié dans les cas bien connus des dimensions 1, 2, et 3 ; il pourra ensuite expérimenter dans des dimensions supérieurs, et ainsi vérifier les calculs d'Agan si aboutissent un jour ! Nicolas écrit donc le programme ci-dessous.

```
import argparse
import sys, getopt, time, random, math

def vol(dim, N):
    Nin = 0.0
    for i in range(N):
        disto = 0.0
        for d in range(dim):
            x = random.uniform(-1, 1)
            disto += x*x
        if disto <= 1.0:
            Nin += 1.0
    return math.pow(2.0, dim)*Nin/N

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="LAnce le calcul du volume de la sphere")
    parser.add_argument("-d", "--dim", dest="dim", required=False, default=3, type=int, help="dimension")
    parser.add_argument("-N", "--number", dest="N", default=100, type=int, help="Nombre de test")

    args = parser.parse_args()
    print(f"dim={args.dim}")
    print(f"Nb tests={args.N}")
    print("Estimation du volume d'une sphére de rayon 1 en")
    print("dimension {} et avec {} échantillons...".format(args.dim, args.N))
    t_bgn = time.time();
    V = vol(args.dim, args.N)
    t_end = time.time()
    print("volume = {}".format(V))
    print("temps écoulé = {:5.3}s".format(t_end-t_bgn))
```

En dimension 2 notamment, Nicolas constate qu'il obtient des résultats intéressants, mais il se rend compte aussi que le temps d'exécution devient vite important pour obtenir une approximation raisonnable du résultat... Voici un exemple d'exécution :

```
$ python3 src/volsphere.py -d 2 -N 10000000
dim = 2
N = 10000000
Estimation du volume d'une sphére de rayon 1 en
dimension 2 et avec 10000000 échantillons...
volume = 3.1420716
temps écoulé = 6.93s
nlouvet@nlbook:~/docsvn/cours-capes/tp-sys-py1$
```

Vous devez aider le petit Nicolas à accélérer ses calculs en les parallélisant à l'aide de processus !

1) Exprimez le temps d'exécution théorique du programme de Nicolas en fonction de dim et de N.

2) Supposons que l'on arrive à répartir le travail réalisé par le programme de Nicolas équitablement sur n processeurs pouvant travailler de façon indépendante : quel sera le temps d'exécution du programme ?

3) Sur papier, réfléchissez à la manière de paralléliser le programme de Nicolas : restez pour l'instant au niveau algorithmique (différentes solutions sont possibles).

4) Programmer votre algorithme, en utilisant le module multiprocessing.

5) Tester votre programme : le temps d'exécution se comporte-t-il bien comme vous aviez prévu ?

Références

[1] https://docs.python.org/fr/3.8/library/os.html

[2] https://docs.python.org/fr/3.8/library/multiprocessing.html