

Formation Python : Tâches parallèles

par Bernard CHAMBON

CC-IN2P3, Lyon - France

19, 20, 21 février 2018

- Multi-threading
 - Mise en oeuvre
 - Accès aux ressources partagées, synchronization
- Multi-processing
 - Mise en oeuvre
 - Echange entre processus
 - Récupération du code de retour, interruption
- Sous-Processus
 - Mise en oeuvre
 - Récupération du code de retour et de `stdout|err`

■ Définition

- Tâches en `//`, au sein d'un même processus => léger
- Partage la même zone mémoire, besoin de contrôler les accès aux ressources partagées
- Si utilisation d'une API, savoir si elle est 'thread safe'

■ Comment

- Par héritage de la classe `Thread` (module `threading`) et en implémentant la méthode `run()`
- Démarrage par invocation de la méthode `start()`, fin lorsque la méthode `run()` se termine
- Un thread ne s'interrompt pas!
Si besoin voir le package `concurrent` et la classe `Future` (`Future.cancel()`)
- Le 'fil' principal initiant les threads, doit se mettre en attente de la fin des threads, par la méthode `join()`

■ A savoir

- L'interpréteur python classique (CPython) n'est pas optimal en terme d'utilisation des cores simultanément
Pour information [global interpreter lock](#)
- L'implémentation python en Java (Jython) permet d'avoir un vrai multithreading et d'exploiter simultanément les cores
- Un code d'application serveur peut comporter des centaines de threads

Tâches parallèles > Multi-threading

■ Syntaxe par l'exemple :

Code

```
from threading import Thread
...
class MyBasicThread(Thread):
    """ To work with thread, you must extends
    Thread class and implements run method """
    def __init__(self):
        Thread.__init__(self)
        return

    def run(self):
        print("At {}, starting thread {}".format(
            datetime.datetime.now().strftime("%H:%M:%S"),
            threading.current_thread().getName()))

        delay_s=10
        print("Sleeping {} s ...".format(delay_s))
        time.sleep(delay_s)

        print("At {}, ending thread {}".format(
            datetime.datetime.now().strftime("%H:%M:%S"),
            threading.current_thread().getName()))
        return

def main():
    my_thread = MyBasicThread()
    my_thread.setName("number-{}".format(1))
    my_thread.start()

    print("Waiting for end of threads in {} thread"
          .format(threading.current_thread().getName()))
    my_thread.join()
    print("All running threads are over in {} thread"
          .format(threading.current_thread().getName()))

return
```

Exécution

```
At 16:57:16, starting thread number-1
Sleeping 10 s ...
Waiting for end of threads in MainThread thread
At 16:57:26, ending thread number-1
All running threads are over in MainThread thread
```

■ Accès aux ressources partagées

- Le problème

Les ressources uniques adressées dans la méthode `run()`, peuvent être accédées par plusieurs threads.

Il peut être alors nécessaire de *sérialiser* les accès

- Les solutions : Synchroniser les zones considérées comme critiques

L'utilisation de verrou (classe `threading.Lock`, méthodes `acquire()` / `release()`)

L'utilisation de la classe `Condition` et méthodes `wait()` / `notify()`

(utile pour se mettre en attente d'une condition particulière)

L'utilisation de la classe `Event` et les méthodes `set()`, `clear()`, `wait()` - NON étudié

■ Syntaxe par l'exemple : synchronisation

Code SANS synchronisation

```
class MyThread(Thread):
    def __init__(self):
        Thread.__init__(self)
        return

    def run(self):
        for c in "PYTHON":
            sys.stdout.write(c); sys.stdout.flush()
            delay_s=random.randint(1, 250)
            time.sleep(delay_s/1000)

        sys.stdout.write(" "); sys.stdout.flush()
        return

class PlayingWithThreads():
    def __init__(self):
        return

    def just_do_it(self):
        threads = [];
        for i in range(1,6):
            t = MyThread()
            threads.append(t)

        # start threads then wait
        sys.stdout.write("<"); sys.stdout.flush()
        for t in threads:
            t.start()
        for t in threads:
            t.join()
        sys.stdout.write(">"); sys.stdout.flush()
        return
```

Utilisation de la classe et exécution

```
def main():
    try:
        pwt = PlayingWithThreads()
        pwt.just_do_it()

    except Exception as ex:
        print(ex)
    # end of main
```

<PPPPPPYYHYTOTTNNHHOOTT NHHO NON >

- Syntaxe par l'exemple : synchronisation de la section critique

Code AVEC synchronisation

```
class MyThread(Thread):
    def __init__(self, lock):
        Thread.__init__(self)
        self.lock = lock
        return

    def run(self):
        self.lock.acquire() # lock usage
        for c in "PYTHON":
            sys.stdout.write(c); sys.stdout.flush()
            delay_s=random.randint(1, 250)
            time.sleep(delay_s/1000)

        sys.stdout.write(" "); sys.stdout.flush()
        self.lock.release() # lock usage
        return

class PlayingWithThreads():
    def __init__(self):
        return

    def just_do_it(self):
        # create a list of threads
        threads = [];
        lock = threading.RLock() # create lock
        for i in range(1,6):
            t = MyThread(lock)
            t.setName("number-{}".format(i))
            threads.append(t)

# following code is unchanged
...
```

Exécution

```
<PYTHON PYTHON PYTHON PYTHON PYTHON >
```

■ Exécution différée

- Classe `Timer` (du package `threading`)

Pour initier une action qui sera exécuter plus tard

Dérive de la classe `Thread`, on dispose des méthodes `start()`, `join()`

Les paramètres permettent de spécifier la fonction à exécuter et le delay en seconde

Possibilité de fournir les arguments de la fonction (voire une liste d'arguments variable via `kwargs`)

```
Timer(interval, function, args=None, kwargs=None)
```

Possibilité d'interrompre la tâche par `cancel()`

- Syntaxe par l'exemple

- Syntaxe par l'exemple : utilisation de `Timer` et synchronisation avec `Condition` + `wait()` / `notify()`

Code

```
from threading import Timer
...
class MyTimer():
    def __init__(self,):
        return

    def do_it_later(self):
        print("{} I'm running computations"
              .format(datetime.now().strftime("%H:%M:%S")))
        time.sleep(10)
        return

    def just_do_it(self):
        # will start compute task in 3 s
        timer = Timer(3, self.do_it_later)

        timer.start()
        print("{} Task has been submitted to Timer"
              .format(datetime.now().strftime("%H:%M:%S")))

        timer.join()

        print("{} Task submitted to Timer is over".
              format(datetime.now().strftime("%H:%M:%S")))

        return
```

Exécution

```
11:21:39 Task has been submitted to Timer
11:21:42 I'm running computations
11:21:52 Task submitted to Timer is over
```

■ Définition

- Tâches en `//`, chaque tâche s'exécutant dans un processus (sens unix)
- Moins de problème d'accès concurrent (chaque process a son espace mémoire) → bien plus lourd que les threads
(les process peuvent être visualisés par la commande unix `ps ...`)
- Syntaxe analogue multi-threading
- A ne pas confondre avec les sous-process (vu plus loin)

■ Comment

- Classe `Process` du package `multiprocessing`
- Soit en spécifiant une fonction, via le paramètre `target`
`Process(group=None, target=None, name=None, args=(), kwargs=*, daemon=None)`
- Soit en dérivant la classe et en implémentant la méthode `run()`
- Autres méthodes
`start()`, `join()`, `is_alive()` (déjà vu)
`terminate()` pour envoyer un `SIGTERM`
- Données membres
`pid`
`exit_code`

■ Syntaxe par l'exemple

Tâches parallèles > Multi-processing

- Syntaxe par l'exemple : spécification d'une fonction via la paramètre `target`

Code

```
class BasicMultiProcessing():
    def __init__(self,):
        return

    def do_the_job(self):
        pid = os.getpid()
        numbers=[]
        if ((pid%2) == 0 ) :
            numbers=[i for i in range(0, 10) if (i%2==0) ]
        else:
            numbers=[i for i in range(0, 10) if (i%2==1) ]
        print("sub-process pid is {}, numbers are {}".format(pid, numbers))
        return

    def run_sub_process(self):
        process = []
        # Caution with '()', specifying function name, not calling it !
        process.append(Process(name="SubProcess-1", target=self.do_the_job))
        process.append(Process(name="SubProcess-2", target=self.do_the_job))

        print("process {} initiate sub-process ".format(os.getpid()))
        for p in process:
            p.start()
        print("process {} waiting sub-process ".format(os.getpid()))
        for p in multiprocessing.active_children():
            p.join()
        rcs=[]
        for p in process:
            rcs.append(str(p.exitcode))
        print("All running sub-process are over with success, "
              "exit codes are : {}".
              .format(", ".join(rcs)))
        return
```

Exécution

```
process 96935 initiate sub-process
process 96935 waiting sub-process
sub-process pid is 96936, numbers are [0, 2, 4, 6, 8]
sub-process pid is 96937, numbers are [1, 3, 5, 7, 9]
All running sub-process are over with success,
exit codes are : 0, 0
```

Tâches parallèles > Multi-processing

- Syntaxe par l'exemple : dérivation de `Process` et implémentation de `run()`

Code

```
from multiprocessing import Process
...
class ProcessRunner(Process):
    def __init__(self, msg):
        Process.__init__(self)
        self.msg=msg
        return
    def run(self):
        print("sub-process pid {} saying '{}'"
              .format(os.getpid(), self.msg))
        return

class AdvancedMultiProcessing():
    def __init__(self, ):
        return

    def run_sub_process(self):
        process = []
        process.append(ProcessRunner("How are you ?"))
        process.append(ProcessRunner("How do you do ?"))

        print("process {} initiate sub-process ".format(os.getpid()))
        for p in process:
            p.start()
        print("process {} waiting sub-process ".format(os.getpid()))
        for p in multiprocessing.active_children():
            p.join()

        alives=[]
        for p in process:
            alives.append(str(p.is_alive()))
        print("All running sub-process are over,"
              "alive statuses are : {}"
              .format(" ".join(alives)))

        return
```

Exécution

```
process 96989 initiate sub-process
process 96989 waiting sub-process
sub-process pid 96990 saying 'How are you ?'
sub-process pid 96991 saying 'How do you do ?'
All running sub-process are over,
alive statuses are : False, False
```

■ Synchronization

- Moindre besoin qu'avec le multi-threading, car les processus partagent moins de choses
- On dispose de `Lock()`, `Condition() + wait() / notify()`, `Event()` comme pour le multi-threading

■ Echange (cité mais non étudié)

- Comme la mémoire n'est pas partagée, besoin de mécanisme pour ce besoin

- Par tube

Classe `multiprocessing.Pipe()` pour échanger entre 2 sous-processus

- Par queue

Classe `multiprocessing.Queue()` pour échanger avec plusieurs processus

"multi-producer, multi-consumer FIFO queues is modeled on the `Queue.Queue` class in the standard library"

■ Définition

Pour exécuter une commande (sens unix) dans un sous-process

■ Comment

- Classe `Popen` du module `subprocess`
- `Popen` permet de passer un exécutable, des var. d'env., un shell, un répertoire courant, etc
- Méthode `communicate()` qui se met en attente de la fin d'exécution du sous-process et qui renvoie le couple (`stdout`, `stderr`)
- Possibilité d'envoyer des signaux par `terminate()`, `kill()` ou plus généralement `send_signal(signal)`
- Possibilité de vérifier le status d'activité via `poll()`, d'attendre la fin d'exécution par `wait()`
- Possibilité de connaître le pid et code de retour via donnée membre `pid`, `returncode`

■ Syntaxe par l'exemple

Tâches parallèles > Sous-process

■ Syntaxe par l'exemple

Code

```
class LearningSubProcess():
    def __init__(self,):
        return

    def run_sub_process(self):
        command = "sleep 2; ls /tmp; echo 'BYE !' "

        sub_process = Popen(command, shell=True,
                             stdout=PIPE, stderr=PIPE)
        (sout, serr) = sub_process.communicate()
        if (sub_process.returncode != 0):
            # get error message from stderr
            print("The command failed, error code was {}, "
                  "error message was '{}'"
                  .format(sub_process.returncode,
                          serr if serr is not None else sout))
        else:
            # using poll(), just to show usage,
            # it will return immediately and giving the return code
            print("The command was run under pid {} and "
                  "succeeded since rc = {} "
                  .format(sub_process.pid, sub_process.poll())) #

            # displaying result from sout (sout is a str | bytes (Python 2 | 3)
            print("The command succeeded, result from stdout was {}"
                  .format(sout if sout is not None else None))
            print()
            # other solution, iterating over result
            # caution(sout is a str | bytes (Python 2 | 3)
            sout_as_list = sout.decode("utf-8").split("\n")
            for line in sout_as_list:
                print("{}".format(str(line)))

return
```

Exécution

```
The command was run under pid 97125 and succeeded since rc = 0
The command succeeded, result from stdout was
'com.apple.launchd.8zFNkJtGAh\n
com.apple.launchd.GGQ5NA1hD3\n
com.apple.launchd.jYmFIGUWEM\n
input.txt\n
BYE !\n'

com.apple.launchd.8zFNkJtGAh
com.apple.launchd.GGQ5NA1hD3
com.apple.launchd.jYmFIGUWEM
input.txt
BYE !
```

■ Timeout sur l'exécution d'un sous-process

- Possibilité d'interrompre un sous-process par `terminate()` (envoi d'un SIGTERM) ou `kill()` (envoi d'un SIGKILL)
(Avec Python 3.3 `.communicate(...)` permet de spécifier un timeout en paramètre (`.communicate(timeout=10)`), avec envoi de l'exception `TimeoutExpired` si timeout atteint)
- Possibilité d'envoyer un SIGTERM ou SIGKILL "au bout d'un certain temps" en utilisant la class `Timer` (vu précédemment)

Exercice 4
prévisionnel de 60 mn