Exécution de tâches en parallèles

Bernard CHAMBON

CC-IN2P3, Lyon - France

12, 13, 14 novembre 2018

Tâches parallèles

- Multi-threading
  - Mise en oeuvre
  - Accès aux ressources partagées, synchronization
- Multi-processing
  - Mise en oeuvre
  - Echange entre processus
- *Sous-Processus*
  - Mise en oeuvre
  - Récupération du code de retour et des sorties standards

- 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()` des threads
- 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 simultanement les cores
  - Un code d'application serveur peut comporter des centaines de threads

- Syntaxe par l'exemple : un seul thread

Code

```
import threading
from threading import Thread
import logging
import time

# using logging to display time
# (and outside a class for sake of simplicity)
MY_FORMAT = "%(asctime)s.%(msecs)03d %(message)s"
logging.basicConfig(format=MY_FORMAT, datefmt='%H:%M:%S', level=logging.INFO)
my_logger = logging.getLogger()

class MyBasicThread(Thread):
  def __init__(self):
    Thread.__init__(self)
  def run(self):
    my_logger.info("Starting thread {}".format(threading.current_thread().getName()))

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

    my_logger.info("Ending thread {}".format(threading.current_thread().getName()))

def main():
  my_basic_thread = MyBasicThread()
  my_basic_thread.setName("#1")
  my_basic_thread.start()

  my_logger.info("Waiting for end of threads from {}'"
        .format(threading.current_thread().getName()))
  my_basic_thread.join()
  my_logger.info("All running threads are over from '{}'"
        .format(threading.current_thread().getName()))
```

Exécution

```
16:09:03.500 Starting thread #1
16:09:03.500 Sleeping 10 s ...
16:09:03.500 Waiting for end of threads from 'MainThread'
16:09:13.504 Ending thread #1
16:09:13.504 All running threads are over from 'MainThread'
```

- Syntaxe par l'exemple : plusieurs threads

Code

```
import random
...

class MyBasicThread(Thread):
  def __init__(self):
    Thread.__init__(self)
  def run(self):
    my_logger.info("Starting thread {}"
    .format(threading.current_thread().getName()))
    delay_s = random.randint(2,8)
    my_logger.info("Sleeping {} s ...".format(delay_s))
    time.sleep(delay_s)

    my_logger.info("Ending thread {}"
    .format(threading.current_thread().getName()))

def main():
  my_basic_threads = []
  for i in range(1, 4):
    my_basic_thread = MyBasicThread()
    my_basic_thread.setName(i)
    my_basic_threads.append(my_basic_thread)

  for my_basic_thread in my_basic_threads:
    my_basic_thread.start()

  my_logger.info("Waiting for end of threads from '{}'"
        .format(threading.current_thread().getName()))

  for my_basic_thread in my_basic_threads:
    my_basic_thread.join()

  my_logger.info("All running threads are over from '{}'"
        .format(threading.current_thread().getName()))
```

Exécution

```
16:23:28.593 Starting thread 1
16:23:28.593 Sleeping 3 s ...
16:23:28.594 Starting thread 2
16:23:28.594 Sleeping 5 s ...
16:23:28.594 Starting thread 3
16:23:28.594 Sleeping 8 s ...
16:23:28.594 Waiting for end of threads from 'MainThread'
16:23:31.595 Ending thread 1
16:23:33.596 Ending thread 2
16:23:36.597 Ending thread 3
16:23:36.598 All running threads are over from 'MainThread'
```

- 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 sérializer les accès
  - Les solutions : synchroniser les zones considérées comme critiques
    - ★ Soit l'utilisation de verrou (classe `threading.Lock`, méthodes `acquire()` / `release()`)
    - ★ Soit l'utilisation de la classe `Condition` et méthodes `wait()` / `notify()`
      (utile pour se mettre en attente d'une condition particulière)
    - ★ Soit 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)

  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()


def main():
  my_threads = [];
  for i in range(1, 6):
    my_threads.append(MyThread())

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

Exécution du code

```
<PPPPPYYTYYHYTTTOHTNOHHOHN  ONONN   >
```

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

Code AVEC synchronisation

```
lock = threading.RLock()

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

  def run(self):
    # processing data

    # Entering in critical section ...
    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()
    lock.release()  # lock usage
    # Leaving critical section ...

    # processing data

def main():
  # using the class in the same way as previous
  ...
```

Exécution du code

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

- Exécution différée
  - Classe `Timer` (du package `threading`)
    Pour initier une action qui sera exécutée 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

- Exemple d'utilisation de la classe `Timer`

Code

```
from threading import Timer
...
def run_the_job():
  my_logger.info("Start delayed task in thread {}"
                 .format(threading.current_thread().getName()))
  time.sleep(5)

def do_submit():
  # will start compute task in 3 s
  timer = Timer(3,run_the_job)

  timer.start()
  my_logger.info("Task has been submitted from thread {}"
                 .format(threading.current_thread().getName()))

  timer.join()

  my_logger.info("Task submitted  from thread {} is over"
                 .format(threading.current_thread().getName()))


def main():
   do_submit()
```

Exécution

```
10:26:47.344 Task has been submitted from thread MainThread
10:26:50.349 Start delayed task in thread Thread-1
10:26:55.354 Task submitted from thread MainThread is over
```

Exercise 4
prévisionnel de 1 h

- 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 conforme avec les sous-process (vu plus loin)
- Comment
  - Classe `Process` du package `multiprocessing`
  - Deux utilisations possibles
    - ★ Soit en dérivant la classe et en implémentant la méthode `run()`
    - ★ Soit en spécifiant une fonction, via le paramètre `target`
      `Process(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)`
  - Autres méthodes
    `start()`, `join()`, `is_alive()`, `terminate()` pour envoyer un SIGTERM
  - Données membres
    `pid`
    `exit_code`
- Syntaxe par l'exemple

- 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

  def run(self):
    my_logger.info("Sub-process pid {} saying '{}'"
          .format(os.getpid(), self.msg))
    time.sleep(3)


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

  my_logger.info("Launching sub-processes")
  for p in process:
    p.start()

  for p in process:
    p.join()

  for p in process:
    my_logger.info("process {} is over with exit code {}"
                   .format(p.pid, p.exitcode))

if __name__ == '__main__':
  main()
```

Exécution du code

```
11:01:10.494 Launching sub-processes
11:01:10.498 Sub-process pid 21139 saying 'How are you ?'
11:01:10.499 Sub-process pid 21140 saying 'How do you do ?'
11:01:13.502 process 21139 is over with exit code 0
11:01:13.502 process 21140 is over with exit code 0
```

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

Code

```
def run_the_job():
  my_logger.info("Running the job in process id {}"
                 .format(os.getpid() ))
  time.sleep(3)


def main():
  process = []
  process.append(Process(name="SubProcess-1", target=run_the_job))
  process.append(Process(name="SubProcess-2", target=run_the_job))

  my_logger.info("Launching sub-processes")
  for p in process:
    p.start()

  for p in process :
    p.join()

  for p in process:
    my_logger.info("process {} is over with exit code {}"
                   .format(p.pid, p.exitcode))

if __name__ == '__main__':
  main()
```

Exécution

```
11:05:05.941 Launching sub-processes
11:05:05.945 Running the job in process id 21185
11:05:05.946 Running the job in process id 21186
11:05:08.949 process 21185 is over with exit code 0
11:05:08.949 process 21186 is over with exit code 0
```

- 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
    Attention :
    A ne pas confondre avec la classe `Queue` (module `queue`), vue dans le chapitre 2, qui permet d'échanger des données dans un environnement multithreads (au sein d'un même processus)

Exercise 4 (suite et fin)
prévisionnel de 1 h

- 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 connaitre le pid et code de retour via donnée membre `pid`, `returncode`
- Syntaxe par l'exemple

- Syntaxe par l'exemple et montrant comment récupérer le stdout

Code

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

  sub_process = Popen(command, shell=True,
                      stdout=PIPE, stderr=PIPE)
  (stdout, stderr) = sub_process.communicate()
  if (sub_process.returncode == 0):
    # get result from stdout (a stream of bytes)
    # bytes stream -> str via decode, split to get a list
    lines = stdout.decode("utf-8").split("\n")
    for line in lines:
      print(line)
  else:
    # get error message from stderr (TOBEDONE)
    None



def main():
  run_sub_process()
```

Exécution

```
BBB
aaa
com.apple.launchd.HHOA2sxBzk
com.apple.launchd.tBepTDr4z4
com.apple.launchd.xlqHOn9zWM
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(...)` permets 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)
    (cas d'utilisation avec Python 2)