Python Concurrence

Programmation concurrente (3.5.4)

Accélérons tout ça !

Python Concurrence

Histoire de la concurrence

- La concurrence est en fait dérivée des premiers travaux sur les chemins de fer et la télégraphie, d'où l'utilisation de noms tels que le sémaphore
- Essentiellement, il était nécessaire de gérer plusieurs trains sur le même réseau ferroviaire de manière à ce que chaque train puisse se rendre à destination en toute sécurité sans encourir d'accident
- Ce n'est que dans les années 1960 que les chercheurs se sont intéressés à l'informatique concurrente, et c'est Edsger W. Dijkstra qui a publié le premier article dans ce domaine, où il a identifié et résolu le problème de l'exclusion mutuelle
- Ensuite, Dijkstra a défini des concepts fondamentaux de concurrence, tels que les sémaphores, les exclusions mutuelles et les blocages, ainsi que l'algorithme du plus court chemin de Dijkstra

Histoire de la concurrence

- La concurrence, comme dans la plupart des domaines de l'informatique, est encore un domaine incroyablement jeune comparé à d'autres domaines d'étude tels que les mathématiques, et cela vaut la peine de garder cela à l'esprit
- Il y a encore un énorme potentiel de changement dans le domaine, et il reste un domaine passionnant pour tous (universitaires, concepteurs de langage et développeurs)
- L'introduction de primitives de concurrence de haut niveau et une meilleure prise en charge du langage natif ont vraiment amélioré la façon dont nous, en tant qu'architectes logiciels, mettons en œuvre des solutions concurrentes
- Pendant des années, c'était incroyablement difficile à faire, mais avec l'avènement de nouvelles API concurrentes, et la maturation des framework et des langages, cela commence à devenir beaucoup plus facile pour nous en tant que développeurs

Histoire de la concurrence

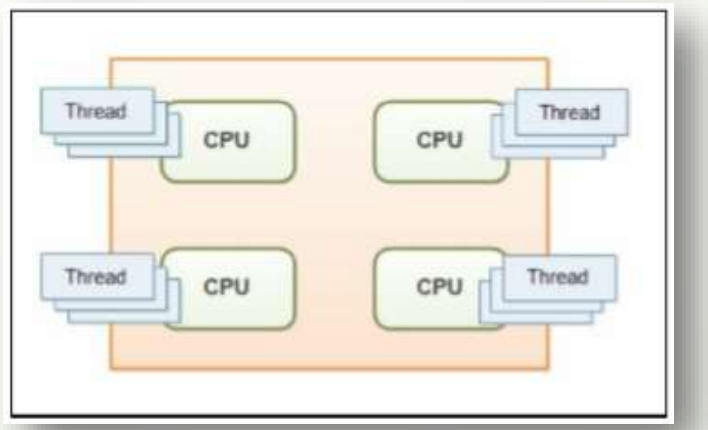
- Les concepteurs de langages sont confrontés à un défi important lorsqu'ils tentent d'implémenter une concurrence non seulement sûre, mais efficace et facile à écrire pour les utilisateurs de cette langue
- Les langages de programmation tels que Golang, Rust et même Python ont fait de grands progrès dans ce domaine, et il est beaucoup plus facile d'exploiter tout le potentiel des machines sur lesquelles vos programmes fonctionnent

Qu'est-ce qu'un thread ?

- Un thread peut être défini comme un flux ordonné d'instructions pouvant être programmé pour s'exécuter comme tel par les systèmes d'exploitation
- Ces threads, typiquement, vivent dans des processus, et consistent en un compteur de programme, une pile, et un ensemble de registres ainsi qu'un identifiant
- Ces threads sont la plus petite unité d'exécution à laquelle un processeur peut allouer du temps
- Les threads peuvent interagir avec des ressources partagées et la communication est possible entre plusieurs threads
- Ils sont également capables de partager de la mémoire, de lire et d'écrire différentes adresses de mémoire, mais c'est là un problème
- Lorsque deux threads commencent à partager de la mémoire et que vous n'avez aucun moyen de garantir l'ordre d'exécution d'un thread, vous pouvez commencer à voir des problèmes ou des bogues mineurs qui vous donnent des valeurs erronées ou font planter votre système

Qu'est-ce qu'un thread ?

- La figure suivante montre comment plusieurs threads peuvent exister sur plusieurs processeurs différents



Types de threads

- Dans un système d'exploitation typique, nous avons généralement deux types de threads distincts:
  - Threads au niveau utilisateur : Threads que nous pouvons activement créer, exécuter et tuer pour toutes nos tâches
  - Threads au niveau du noyau : Threads de très bas niveau au nom du système d'exploitation
- Python fonctionne au niveau de l'utilisateur, et donc, tout ce que nous couvrons dans ce document sera, principalement, axé sur ces threads au niveau de l'utilisateur

Qu'est-ce que le multithreading ?

- Lorsque les gens parlent de processeurs multithread, ils font généralement référence à un processeur qui peut exécuter plusieurs threads simultanément, ce qu'ils sont capables de faire en utilisant un seul cœur capable de changer très rapidement le contexte entre plusieurs threads
- Ce contexte de commutation se déroule en si peu de temps que l'on pourrait penser que plusieurs threads fonctionnent en parallèle alors qu'en réalité ce n'est pas le cas
- Lorsque vous essayez de comprendre le multithreading, il est préférable de considérer un programme multithread comme un bureau
- Dans un programme monothread, il n'y aurait qu'une seule personne travaillant dans ce bureau à tout moment, manipulant tout le travail de manière séquentielle
- Cela deviendrait un problème si l'on considère ce qui se passe lorsque ce travailleur solitaire s'embourbe dans la paperasserie administrative et est incapable de passer à un travail différent

Qu'est-ce que le multithreading ?

- Voyons quelques avantages du threading :
  - Les threads multiples sont excellents pour réduire le temps blocage des E/S liées aux programmes
  - Les threads sont légers en termes d'empreinte mémoire par rapport aux processus
  - Les threads partagent des ressources, et donc la communication entre eux est plus facile

Qu'est-ce que le multithreading ?

- Il y a aussi des inconvénients, qui sont les suivants :
  - Les threads CPython sont entravés par les limitations du verrou global de l'interpréteur (GIL), dont nous parlerons plus en détail
  - Bien que la communication entre les threads puisse être plus facile, vous devez veiller à ne pas implémenter de code soumis à des conditions de concurrence
  - Il est coûteux en temps de changer de contexte entre plusieurs threads
  - En ajoutant plusieurs threads, vous pourriez voir une dégradation des performances globales de votre programme

Qu'est-ce qu'un processus ?

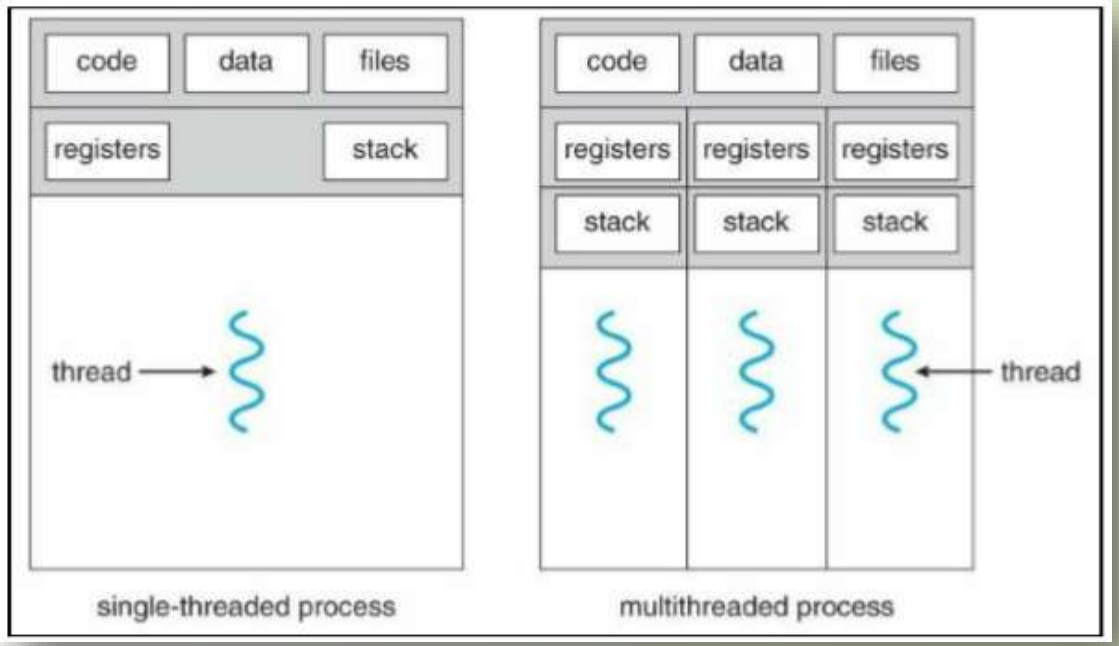
- Les processus sont très similaires aux threads
- Ils nous permettent de faire à peu près tout ce qu'un thread peut faire mais l'avantage principal est qu'ils ne sont pas liés à un noyau CPU singulier
- Si nous étendons notre analogie de bureau, cela signifie essentiellement que si nous avions un processeur à quatre cœurs, nous pourrions embaucher deux membres de l'équipe de vente dédiés et deux employés, et tous les quatre seraient en mesure d'exécuter le travail en parallèle
- Les processus sont également capables de travailler sur plusieurs choses à la fois, tout comme notre employé de bureau unique multithread

Qu'est-ce qu'un processus ?

- Ces processus contiennent un thread principal primaire, mais peuvent engendrer plusieurs sous-threads qui contiennent chacun leur propre jeu de registres et une pile
- Ils peuvent devenir multithread si vous le souhaitez
- Tous les processus fournissent toutes les ressources dont l'ordinateur a besoin pour exécuter un programme

Qu'est-ce qu'un processus ?

- Dans l'image suivante, vous voyez deux diagrammes côte-à-côte; les deux sont des exemples d'un processus
- Vous remarquez que le processus sur la gauche contient un seul thread, autrement connu comme le thread primaire
- Le processus sur la droite contient plusieurs threads, chacun avec son propre ensemble de registres et de piles



Qu'est-ce qu'un processus ?

- Avec les processus, nous pouvons améliorer la vitesse de nos programmes dans des scénarios spécifiques où nos programmes sont liés au processeur et nécessitent plus de puissance CPU
- Cependant, en engendrant de multiples processus, nous sommes confrontés à de nouveaux défis en termes de communication interprocessus, et en tentant de ne pas entraver les performances en passant trop de temps sur cette communication interprocessus (IPC)

Propriétés des processus ?

- Les processus UNIX sont créés par le système d'exploitation et contiennent généralement les éléments suivants
  - ID de processus
  - ID de groupe de processus
  - ID utilisateur et ID de groupe
  - Environnement
  - Répertoire de travail
  - Instructions de programme
  - Registres
  - Pile
  - Heap
  - Descripteurs de fichier
  - Actions de signal
  - Bibliothèques partagées
  - Outils de communication interprocessus (tels que files d'attente de messages, pipes, sémaphores ou mémoire partagée)

Propriétés des processus ?

- Les avantages des processus sont les suivants :
  - Les processus peuvent mieux utiliser les processeurs multi cœurs
  - Ils sont meilleurs que les threads multiples pour gérer les tâches gourmandes en ressources
  - Nous pouvons contourner les limitations du GIL en engendrant plusieurs processus. Programme
  - Un processus qui se termine en erreur ne provoquera pas l'arrêt du programme entier

Propriétés des processus ?

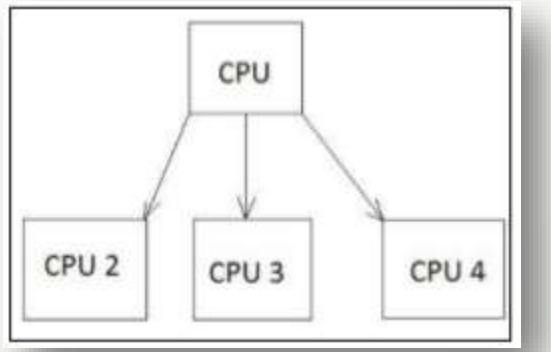
- Voici les principaux inconvénients des processus :
  - Pas de ressources partagées entre les processus
  - Nous devons mettre en œuvre une forme de IPC
  - Tout ceci nécessite plus de mémoire

Multi processing ?

- En Python, nous pouvons choisir d'exécuter notre code à l'aide de plusieurs threads ou de plusieurs processus si nous souhaitons essayer d'améliorer les performances par rapport à une approche monothread standard
- Nous pouvons aller vers une approche multithread et être limité à la puissance de traitement d'un noyau de CPU, ou inversement,
- Nous pouvons aller avec une approche multi processing et utiliser le nombre total de cœurs de processeurs disponibles sur notre machine
- Dans les ordinateurs modernes d'aujourd'hui, nous avons tendance à avoir de nombreux processeurs et cœurs, de sorte que nous limiter à un seul, rend le reste de notre machine inactive
- Notre objectif est d'essayer d'extraire tout le potentiel de notre matériel et de nous assurer d'obtenir le meilleur rapport qualité-prix et de résoudre nos problèmes plus rapidement que quiconque

Multi processing ?

- Avec le module multi processing de Python, nous pouvons utiliser efficacement le nombre total de cœurs et de processeurs, ce qui peut nous aider à obtenir de meilleures performances en ce qui concerne les problèmes liés au processeur



- La figure précédente montre un exemple de la façon dont un cœur CPU commence à déléguer des tâches à d'autres cœurs

Multi processing ?

- Dans toutes les versions de Python supérieures ou égales à 2.6, nous pouvons obtenir le nombre de cœurs de processeur disponibles en utilisant le code suivant

```
>>> import multiprocessing
>>> multiprocessing.cpu_count()
8
>>>
```

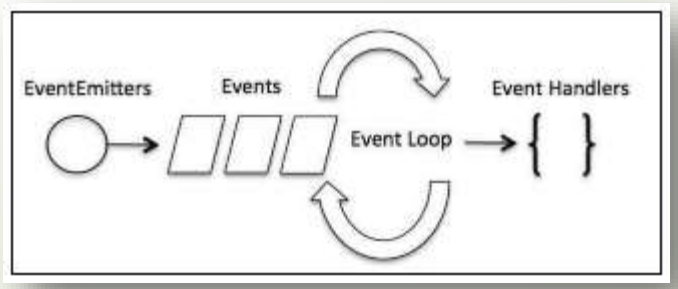
- Non seulement le multi processing nous permet d'utiliser plus de notre machine, mais nous évitons également les limitations que le verrou global d'interpréteur nous impose dans Python
- Un inconvénient potentiel de plusieurs processus est que nous n'avons intrinsèquement pas d'état partagé et que nous manquons de communication
- Nous devons donc passer à travers une forme de IPC, et la performance peut en prendre un coup
- Cependant, ce manque d'état partagé peut faciliter le travail, car vous n'avez pas à vous battre contre des conditions de concurrence potentielles dans votre code

Programmation événementielle

- La programmation événementielle est une partie importante de nos vies
- Nous en voyons des exemples tous les jours lorsque nous ouvrons notre téléphone ou travaillons sur notre ordinateur
- Ces appareils fonctionnent uniquement de manière événementielle
- Par exemple, lorsque vous cliquez sur une icône sur votre bureau, le système d'exploitation enregistre cela comme un événement, puis effectue l'action nécessaire liée à ce style d'événement spécifique.

Programmation événementielle

- Chaque interaction que nous faisons peut être caractérisée comme un événement ou une série d'événements, et ceux-ci déclenchent généralement des rappels (callbacks)
- Si vous avez une expérience antérieure avec JavaScript, vous devriez vous familiariser avec ce concept de rappel et le modèle de conception de rappel
- En JavaScript, le cas d'utilisation prédominant pour les rappels est lorsque vous effectuez des requêtes HTTP RESTful et que vous voulez pouvoir effectuer une action lorsque vous savez que cette action s'est terminée avec succès et que nous avons reçu notre réponse HTTP:



Programmation événementielle

- Si nous regardons l'image précédente, elle nous montre un exemple de la façon dont les programmes pilotés par les événements traitent les événements
- Nous avons nos EventEmitters sur le côté gauche
- Ces derniers déclenchent plusieurs événements, qui sont captés par la boucle d'événements de notre programme, et, s'ils correspondent à un gestionnaire d'événements prédéfini, ce gestionnaire est alors déclenché pour gérer ledit événement

Programmation événementielle

- Les rappels (callbacks) sont souvent utilisés dans les scénarios où une action est asynchrone
- Supposons, par exemple, que vous postuliez pour un emploi chez Google, que vous leur donniez une adresse e-mail, et qu'ils vous contacteront quand ils prendront leur décision
- Cela revient essentiellement à enregistrer un rappel sauf qu'au lieu de vous envoyer un e-mail, vous exécutez un code arbitraire chaque fois que le rappel est appelé

Turtle

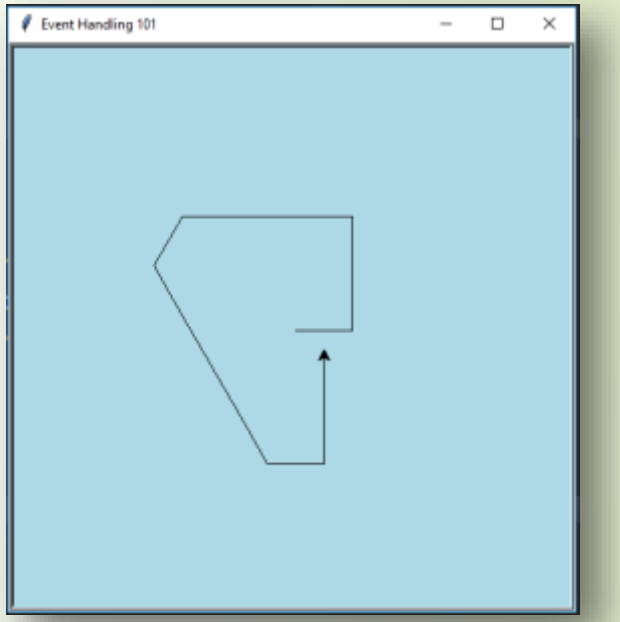
- Turtle est un module graphique qui a été écrit en Python, et c'est un excellent point de départ pour intéresser les enfants à la programmation
- Il gère toutes les complexités liées à la programmation graphique et leur permet de se concentrer uniquement sur l'apprentissage des bases tout en les gardant intéressés
- C'est aussi un très bon outil à utiliser pour démontrer des programmes axés sur les événements
- Il comporte des gestionnaires d'événements et des auditeurs, ce qui est tout ce dont nous avons besoin

turtle1.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import turtle
5
6
7  def main():
8      import turtle
9      turtle.setup(500, 500)
10     window = turtle.Screen()
11     window.title("Event Handling 101")
12     window.bgcolor("lightblue")
13     nathan = turtle.Turtle()
14
15     def moveForward():
16         nathan.forward(50)
17
18     def moveLeft():
19         nathan.left(30)
20
21     def moveRight():
22         nathan.right(30)
23
24     def start():
25         window.onkey(moveForward, "Up")
26
27     window.onkey(moveForward, "Up")
28     window.onkey(moveLeft, "Left")
29     window.onkey(moveRight, "Right")
30     window.listen()
31     window.mainloop()
32
33
34 if __name__ == '__main__':
35     main()
36
```

Turtle

- Dans la première ligne de cet exemple de code précédent, nous importons le module graphique Turtle
- Nous poursuivons ensuite pour mettre en place une fenêtre de tortue de base avec le titre Event Handling 101 et une couleur de fond de bleu clair
- Une fois la configuration initiale terminée, nous définissons trois gestionnaires d'événements distincts :
- moveForward : Quand nous voulons déplacer notre personnage de 50 unités
- moveLeft/moveRight : Quand nous voulons faire pivoter notre personnage dans les deux sens de 30 degrés
- Une fois que nous avons défini nos trois gestionnaires distincts, nous passons ensuite à la mise en correspondance de ces gestionnaires d'événements avec les touches vers le haut, vers la gauche et vers la droite en utilisant la méthode onkey



Programmation réactive

- La programmation réactive est très similaire à celle de l'événement, mais au lieu de tourner autour des événements, elle se concentre sur les données
- Plus spécifiquement, elle traite des flux de données et réagit aux changements de données spécifiques

ReactiveX - RxPy

- RxPy est l'équivalent Python du très populaire framework ReactiveX
- Si vous avez déjà effectué une programmation dans Angular 2 et versions suivantes, vous l'utiliserez lors de l'interaction avec les services HTTP
- Ce framework est une conglomération du modèle d'observateur, du modèle d'itérateur et de la programmation fonctionnelle
- Nous nous abonnons essentiellement à différents flux de données entrantes, puis créons des observateurs qui écoutent les événements spécifiques déclenchés
- Lorsque ces observateurs sont déclenchés, ils exécutent le code correspondant à ce qui vient de se passer

ReactiveX - RxPy

- Nous considérerons un centre de données comme un bon exemple de la façon dont la programmation réactive peut être utilisée
- Imaginez que ce centre de données possède des milliers de racks de serveurs, qui calculent en permanence des millions et des millions de calculs
- L'un des plus grands défis de ces datacenters est de garder suffisamment refroidis tous ces racks de serveur bien serrés pour qu'ils ne s'endommagent pas
- Nous pourrions installer plusieurs thermomètres dans notre centre de données pour nous assurer de ne pas avoir trop chaud et envoyer les lectures de ces thermomètres à un ordinateur central en continu

ReactiveX - RxPy

- Au sein de notre centrale de contrôle, nous pourrions mettre en place un programme RxPy qui observe ce flux continu d'informations sur la température
- Au sein de ces observateurs, nous pourrions alors définir une série d'événements conditionnels à écouter, puis réagir chaque fois que l'un de ces conditionnels est atteint
- Un tel exemple serait un événement qui se déclenche uniquement si la température d'une partie spécifique du centre de données devient trop chaude
- Lorsque cet événement est déclenché, nous pouvons alors automatiquement réagir et augmenter le débit de tout système de refroidissement vers cette zone particulière, et ainsi ramener la température à un bon niveau

Programmation GPU

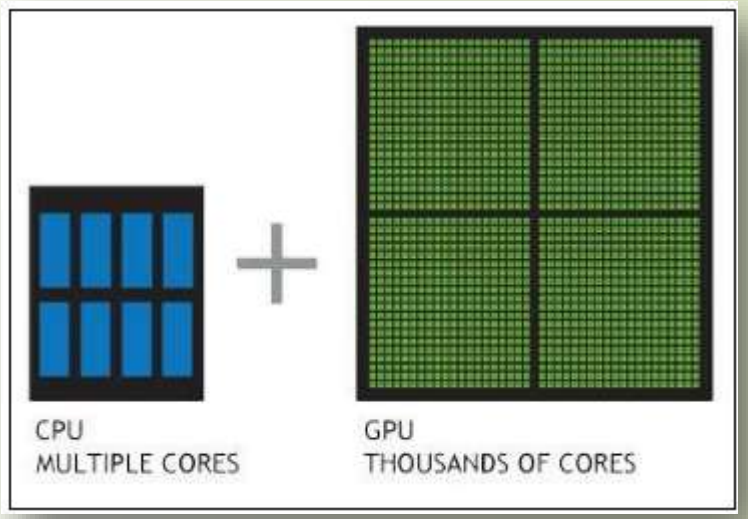
- Les GPU sont réputés pour leur capacité à rendre des jeux vidéo d'action rapides et de haute résolution
- Ils sont capables d'absorber ensemble les millions de calculs nécessaires par seconde afin de s'assurer que chaque vertex des modèles 3D de votre jeu est au bon endroit, et qu'ils sont mis à jour toutes les quelques millisecondes afin d'assurer une douceur de 60 FPS
- De manière générale, les GPU sont incroyablement performants pour effectuer la même tâche en parallèle, des millions et des millions de fois par minute
- Mais si les GPU sont si performants, alors pourquoi ne pas les utiliser à la place de nos processeurs ?
- Alors que les GPU peuvent être incroyablement performants au traitement graphique, ils ne sont cependant pas conçus pour gérer les subtilités de l'exécution d'un système d'exploitation et de l'informatique à usage général
- Les CPU ont moins de noyaux, qui sont spécifiquement conçus pour la vitesse quand il s'agit de changer de contexte entre les tâches d'exploitation
- Si les GPU avaient les mêmes tâches, vous constateriez une dégradation considérable des performances globales de votre ordinateur

Programmation GPU

- Mais comment pouvons-nous utiliser ces cartes graphiques de haute puissance pour autre chose que la programmation graphique ?
- C'est ici qu'interviennent les bibliothèques telles que PyCUDA, OpenCL et Theano
- Ces bibliothèques tentent de faire abstraction du code complexe de bas niveau auquel les API graphiques doivent interagir pour utiliser le GPU
- Ils simplifient grandement la réutilisation des milliers de cœurs de traitement plus petits disponibles sur le GPU et leur utilisation pour nos programmes coûteux en calcul

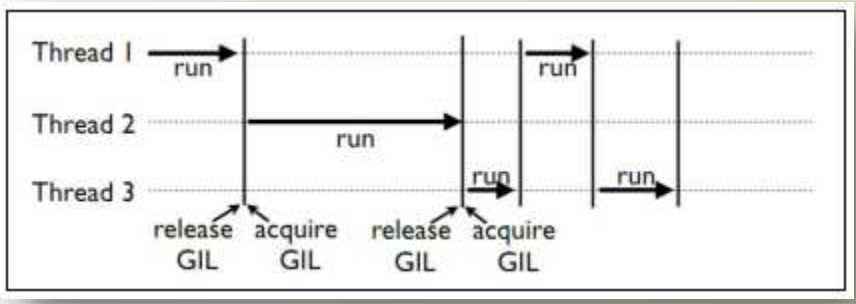
Programmation GPU

- Ces unités de traitement graphique (GPU) encapsulent tout ce que les langages de script ne sont pas
- Ils sont hautement parallélisables et construits pour un débit maximum
- En les utilisant en Python, nous sommes en mesure d'obtenir le meilleur des deux mondes
- Nous pouvons utiliser un langage favorisé par des millions de personnes en raison de sa facilité d'utilisation, et rendre nos programmes incroyablement performants



Les limitations de Python

- Plus tôt dans le chapitre, j'ai parlé des limites du GIL ou du Global Interpreter Lock présent dans Python, mais qu'est-ce que cela signifie réellement ?
- Premièrement, je pense qu'il est important de savoir exactement ce que le GIL fait pour nous
- Le GIL est essentiellement un verrou d'exclusion mutuelle qui empêche plusieurs threads d'exécuter du code Python en parallèle
- C'est un verrou qui ne peut être tenu que par un thread à la fois, et si vous voulez qu'un thread exécute son propre code, il doit d'abord acquérir le verrou avant de pouvoir exécuter son propre code
- L'avantage que cela nous procure est que pendant qu'il est verrouillé, rien d'autre ne peut fonctionner en même temps



Les limitations de Python

- Dans le diagramme précédent, nous voyons un exemple de la façon dont plusieurs threads sont entravés par ce GIL
- Chaque thread doit attendre et acquérir le GIL avant de pouvoir continuer à progresser, puis relâcher le GIL, généralement avant qu'il n'ait pu terminer son travail
- Il suit une approche aléatoire à la ronde, et vous n'avez aucune garantie quant au thread qui va acquérir le verrou en premier
- Pourquoi est-ce nécessaire, vous pourriez demander ?
- Eh bien, le GIL a longtemps été une partie de Python, et au fil des années a déclenché de nombreux débats sur son utilité
- Mais il a été implémenté avec de bonnes intentions et pour lutter contre la gestion de la mémoire Python non-thread-safe
- Il nous empêche de tirer parti des systèmes multiprocesseurs dans certains scénarios

Les limitations de Python

- Guido Van Rossum, le créateur de Python, a publié une mise à jour sur la suppression du GIL et de ses avantages dans un post ici :
- http://www.artima.com/weblogs/viewpost.jsp?thread=214235
- Il déclare qu'il ne serait pas contre quelqu'un créant une branche de Python qui soit sans GIL, et il accepterait une fusion de ce code si, et seulement si, cela n'avait pas d'impact négatif sur la performance d'une application mono threaded
- Il y a eu des tentatives antérieures pour se débarrasser du GIL, mais il a été constaté que l'ajout de tous les verrous supplémentaires pour assurer la sécurité du fil ralentissait réellement une application d'un facteur de plus de deux
- En d'autres termes, vous auriez pu faire plus de travail avec un seul processeur qu'avec un peu plus de deux processeurs
- Il y a cependant des bibliothèques comme NumPy qui peuvent faire tout ce dont elles ont besoin sans avoir à interagir avec le GIL, et travailler en dehors du GIL est quelque chose que nous approfondirons dans les prochains chapitres de ce livre

Les limitations de Python

- Il faut également noter qu'il existe d'autres implémentations de Python, telles que Jython et IronPython, qui n'offrent aucune forme de verrouillage d'interpréteur global et, en tant que telles, peuvent exploiter pleinement les systèmes multiprocesseurs
- Jython et IronPython s'exécutent sur des machines virtuelles différentes, ce qui leur permet de tirer parti de leurs environnements d'exécution respectifs

Jython

- Jython est une implémentation de Python qui fonctionne directement avec la plateforme Java
- Il peut être utilisé de manière complémentaire avec Java en tant que langage de script, et il a été démontré qu'il surpassait CPython, qui est l'implémentation standard de Python, lorsqu'il travaille avec de grands ensembles de données
- Cependant, pour la majorité des choses, l'exécution par un seul noyau de CPython surpasse généralement Jython et son approche multi cœur
- L'avantage de l'utilisation de Jython réside dans le fait que vous pouvez faire des choses très intéressantes lorsque vous travaillez en Java, comme importer des bibliothèques et des framework Java existants, et les utiliser comme s'ils faisaient partie de votre code Python.

IronPython

- IronPython est l'équivalent .NET de Jython et fonctionne au-dessus du framework .NET de Microsoft
- Encore une fois, vous serez en mesure de l'utiliser de manière complémentaire avec les applications .NET
- Ceci est quelque peu bénéfique pour les développeurs .NET, car ils sont capables d'utiliser Python comme langage de script rapide et expressif dans leurs applications .NET

Pourquoi utiliser Python ?

- Si Python a des limitations évidentes et connues quand il s'agit d'écrire des applications performantes, alors pourquoi continuons-nous à l'utiliser ?
- La réponse courte est que c'est un langage fantastique pour faire le travail, et par le travail, je ne parle pas nécessairement d'une tâche coûteuse en termes de calcul
- C'est un langage intuitif, facile à comprendre et à comprendre pour ceux qui n'ont pas forcément beaucoup d'expérience en programmation
- Le langage a vu un taux d'adoption énorme parmi les scientifiques des données et les mathématiciens travaillant dans des domaines incroyablement intéressants tels que l'apprentissage automatique et l'analyse quantitative, qui trouvent que c'est un outil incroyablement utile dans leur arsenal
- Dans les deux écosystèmes Python 2 et 3, vous trouverez un grand nombre de bibliothèques conçues spécifiquement pour ces cas d'utilisation, et en connaissant les limites de Python, nous pouvons les atténuer efficacement, et produire un logiciel efficace et capable de faire exactement ce qui est requis

Téléchargement d'images

- Un excellent exemple des avantages du multithreading est, sans aucun doute, l'utilisation de plusieurs threads pour télécharger plusieurs images ou fichiers
- C'est, en fait, l'un des meilleurs cas d'utilisation pour le multithreading en raison de la nature bloquante des E/S
- Pour mettre en évidence les gains de performance, nous allons récupérer 10 images différentes de http://lorempixel.com/400/200/sports, qui est une API gratuite qui fournit une image différente chaque fois que vous cliquez sur ce lien
- Nous allons ensuite stocker ces 10 images différentes dans un dossier temporaire afin que nous puissions les voir ou les utiliser plus tard

Téléchargement d'images : *Séquentiel*

- Premièrement, nous devrions disposer d'une base de référence permettant de mesurer les gains de performance
- Pour ce faire, nous écrirons un programme rapide qui téléchargera séquentiellement ces 10 images, comme suit :

concurrency1.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time
import urllib.request


def downloadImage(imagePath, fileName):
    print("Downloading Image from ", imagePath)
    urllib.request.urlretrieve(imagePath, fileName)


def main():
    t0 = time.time()
    for i in range(10):
        imageName = "temp/image-" + str(i) + ".jpg"
        downloadImage("http://lorempixel.com/400/200/sports", imageName)
    t1 = time.time()
    total_time = t1 - t0
    print("Execution time: {}".format(total_time))


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

```
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Execution time: 2.46917724609375
```

Téléchargement d'images : *Concurrent*

- Maintenant que nous avons notre base de référence, il est temps d'écrire un programme rapide qui téléchargera simultanément toutes les images dont nous avons besoin
- Nous allons continuer à créer et à démarrer des threads dans les prochains chapitres, donc ne vous inquiétez pas si vous avez du mal à comprendre le code
- Le point clé de ceci est de réaliser les gains de performance potentiels à avoir en écrivant des programmes simultanément

concurrency2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import threading, urllib.request, time


def downloadImage(imagePath, fileName):
    print("Downloading Image from ", imagePath)
    urllib.request.urlretrieve(imagePath, fileName)


def executeThread(i):
    imageName = "temp/image-" + str(i) + ".jpg"
    downloadImage("http://lorempixel.com/400/200/sports", imageName)


def main():
    t0 = time.time()
    threads = []
    for i in range(10):
        thread = threading.Thread(target=executeThread, args=(i,))
        threads.append(thread)
        thread.start()
    for i in threads:
        i.join()
    t1 = time.time()
    totalTime = t1 - t0
    print("Total Execution Time {}".format(totalTime))


if __name__ == '__main__':
    main()

```

```
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Downloading Image from  http://lorempixel.com/400/200/sports
Total Execution Time 0.5017025470733643
```

Améliorer le calcul

- Ainsi, nous avons vu exactement comment nous pouvons améliorer des choses telles que le téléchargement d'images, mais comment pouvons-nous améliorer la performance de notre calcul de nombres ?
- Eh bien, c'est là que le multitraitement brille s'il est utilisé correctement
- Dans cet exemple, nous allons essayer de trouver les facteurs premiers de 10 000 nombres aléatoires compris entre 20 000 et 100 000 000
- Nous ne sommes pas nécessairement préoccupés par l'ordre d'exécution tant que le travail est effectué et que nous ne partageons pas la mémoire entre nos processus

Factorisation séquentielle

- Encore une fois, nous allons écrire un script qui le fait d'une manière séquentielle, que nous pouvons facilement vérifier fonctionne correctement :

```
9991 37016767 -> [73, 507079]
9992 15042428 -> [2, 2, 811, 4637]
9993 66177718 -> [2, 3623, 9133]
9994 59639200 -> [2, 2, 2, 2, 2, 5, 5, 127, 587]
9995 10271140 -> [2, 2, 5, 11, 46687]
9996 30917369 -> [7, 643, 6869]
9997 87089 -> [73, 1193]
9998 77469870 -> [2, 3, 5, 2582329]
9999 41286402 -> [2, 3, 3, 3, 764563]
Execution Time: 1.750622272491455
```

concurrency3.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import time
import random


def calculate_prime_factors(n):
    primfac = []
    d = 2
    while d * d <= n:
        while (n % d) == 0:
            primfac.append(d)
            n //= d
        d += 1
    if n > 1:
        primfac.append(n)
    return primfac


def main():
    print("Starting number crunching")
    t0 = time.time()
    for i in range(10000):
        rand = random.randint(20000, 100000000)
        print(i, rand, '->', calculate_prime_factors(rand))
    t1 = time.time()
    total_time = t1 - t0
    print("Execution Time: {}".format(total_time))


if __name__ == '__main__':
    main()

```

Factorisation concurrente

- Voyons maintenant comment nous pouvons améliorer les performances de ce programme en utilisant plusieurs processus
- Afin de diviser cette charge de travail, nous allons définir une fonction ***executeProc*** qui, au lieu de générer 10 000 nombres aléatoires à factoriser, générera 1 000 nombres aléatoires
- Nous allons créer 10 processus, et exécuter la fonction 10 fois, de sorte que le nombre total de calculs devrait être exactement le même que lorsque nous avons effectué le test séquentiel

```
29020301 -> [29020301]
95560538 -> [2, 19, 23, 31, 3527]
8703773 -> [13, 607, 1103]
65542457 -> [19, 3449603]
22495699 -> [401, 56099]
Execution Time: 1.3285346031188965
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time
import random
from multiprocessing import Process


def calculatePrimeFactors(n):
    primfac = []
    d = 2
    while d * d <= n:
        while (n % d) == 0:
            primfac.append(d)
            n //= d
        d += 1
    if n > 1:
        primfac.append(n)
    return primfac


def executeProc():
    for i in range(1000):
        rand = random.randint(20000, 100000000)
        print(rand, '->', calculatePrimeFactors(rand))


def main():
    print("Starting number crunching")
    t0 = time.time()
    procs = []
    for i in range(10):
        proc = Process(target=executeProc, args=())
        procs.append(proc)
        proc.start()
    for proc in procs
        proc.join()
    t1 = time.time()
    totalTime = t1 - t0
    print("Execution Time: {}".format(totalTime))


if __name__ == '__main__':
    main()

```

Concurrence vs Parallélisme

- La concurrence et le parallélisme sont deux concepts généralement confondus
- La réalité, cependant, est qu'ils sont très différents, et si vous avez conçu un logiciel concurrent au lieu d'avoir besoin d'une exécution parallèle, alors vous pourriez sérieusement affecter le véritable potentiel de performance de votre logiciel
- Pour cette raison, il est essentiel de savoir exactement ce que les deux concepts signifient afin que vous puissiez comprendre les différences
- En connaissant ces différences, vous aurez les clés pour concevoir vos propres logiciels haute performance en Python

Parallélisons !

Python Concurrence

Comprendre la concurrence

- La concurrence est, essentiellement, de faire plusieurs choses en même temps, mais pas, spécifiquement, en parallèle
- Cela peut nous aider à améliorer les performances perçues de nos applications, et cela peut également améliorer la vitesse à laquelle nos applications fonctionnent
- La meilleure façon de voir comment la concurrence fonctionne est d'imaginer une personne travaillant sur plusieurs tâches et passant rapidement d'une tâche à l'autre
- Imaginez que cette personne travaille simultanément sur un programme et, en même temps, traite des demandes de support
- Cette personne se concentrerait principalement sur l'écriture de son programme, et rapidement le contexte passerait à la résolution d'un bug ou à la gestion d'un problème de support s'il y en avait un
- Une fois qu'ils ont terminé la tâche de support, ils peuvent à nouveau changer de contexte, pour écrire leur programme très rapidement

Comprendre la concurrence

- Cependant, en informatique, il existe généralement deux goulets d'étranglement des performances que nous devons surveiller et éviter en écrivant nos programmes
- Il est important de connaître les différences entre les deux goulets d'étranglement
- Si vous essayez d'appliquer la concurrence à un goulot d'étranglement basé sur le processeur, alors vous constaterez que le programme commence à voir une diminution des performances
- Et si vous avez essayez d'appliquer le parallélisme à une tâche qui nécessite réellement une solution concurrente, vous pouvez à nouveau voir les mêmes performances

Propriétés d'un système concurrent

- Tous les systèmes concurrents partagent un ensemble de propriétés similaire
- Ceux-ci peuvent être définis comme suit :
  - **Acteurs multiples :** Ceci représente les différents processus et threads qui essaient tous de progresser activement sur leurs propres tâches. Nous pourrions avoir plusieurs processus qui contiennent plusieurs threads essayant tous de s'exécuter en même temps
  - **Ressources partagées :** Cette fonctionnalité représente la mémoire, le disque et les autres ressources que les acteurs du groupe précédent doivent utiliser pour effectuer ce qu'ils doivent faire
  - **Règles :** Il s'agit d'un ensemble strict de règles que tous les systèmes concurrents doivent suivre et qui définissent quand les acteurs peuvent et ne peuvent pas acquérir de verrous, accéder à la mémoire, modifier l'état, etc. Ces règles sont essentielles pour que ces systèmes concurrents fonctionnent, sinon, nos programmes se déchireraient les uns les autres

Goulots d'étranglement d'E/S

- Les goulots d'étranglement d'E/S sont des goulots d'étranglement où votre ordinateur passe plus de temps à attendre diverses entrées et sorties qu'à traiter les informations
- Vous trouverez généralement ce type de goulot d'étranglement lorsque vous travaillez avec une application lourde en E/S
- Nous pourrions considérer votre navigateur Web standard comme un exemple d'application aux E/S lourdes
- Dans un navigateur, nous passons généralement beaucoup plus de temps à attendre que les requêtes réseau se terminent pour charger des feuilles de style, des scripts ou des pages HTML plutôt que de les afficher à l'écran
- Si la vitesse à laquelle les données sont demandées est plus lente que la vitesse à laquelle elles sont consommées, alors vous avez un goulot d'étranglement d'E/S

Goulots d'étranglement d'E/S

- L'un des principaux moyens d'améliorer la vitesse de ces applications est soit d'améliorer la vitesse des E/S sous-jacentes en achetant du matériel plus coûteux et plus rapide, soit d'améliorer la façon dont nous traitons ces demandes d'E/S
- Un excellent exemple de programme lié par les goulots d'étranglement d'E/S serait un robot d'indexation Web
- L'objectif principal d'un robot d'exploration Web est désormais de parcourir le Web et d'indexer les pages Web pour qu'elles puissent être prises en compte lorsque Google exécute son algorithme de classement des résultats pour déterminer les 10 premiers résultats d'un mot clé donné

Goulots d'étranglement d'E/S

- Nous allons commencer par créer un script très simple qui ne demande qu'une page et combien de temps il faut pour demander cette page web

concurrency5.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import urllib.request
5  import time
6
7
8  def main():
9      t0 = time.time()
10     request = urllib.request.urlopen("http://www.iana.org/domains/reserved")
11     page_html = request.read()
12     t1 = time.time()
13     print("\n\nTotal Time To Fetch Page: {} Seconds\n\n".format(t1 - t0))
14
15
16 if __name__ == '__main__':
17     #
18     # Call the main() function
19     #
20     main()
21
```

```
Total Time To Fetch Page: 0.562730073928833 Seconds
```

Goulots d'étranglement d'E/S

- Maintenant, disons que nous voulions ajouter un peu de complexité et suivre les liens vers d'autres pages afin que nous puissions les indexer dans le futur
- Nous pourrions utiliser une bibliothèque comme ***BeautifulSoup*** afin de rendre nos vies un peu plus faciles
- Vous remarquerez à partir de cette sortie que le temps d'aller chercher la page est supérieur à une demie seconde
- Maintenant, imaginons que si nous voulions exécuter notre robot d'exploration pour un million de pages Web différentes, notre temps d'exécution total serait environ un million de fois plus long…

concurrency6.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import urllib.request
import time
from bs4 import BeautifulSoup


def main():
    t0 = time.time()
    req = urllib.request.urlopen("http://www.iana.org/domains/reserved")
    t1 = time.time()
    print("\n\n\nTotal Time To Fetch Page: {} Seconds".format(t1 - t0))
    soup = BeautifulSoup(req.read(), "html.parser")
    for link in soup.find_all('a'):
        link.get('href')
    t2 = time.time()
    print("Total Execeution Time: {} Seconds\n\n".format(t2 - t0))


if __name__ == '__main__':

    main()

```

```
Total Time To Fetch Page: 0.5942203998565674 Seconds
Total Execeution Time: 0.6097383499145508 Seconds
```

Goulots d'étranglement d'E/S

- La vraie cause principale de ce temps d'exécution énorme résulterait purement et simplement du goulot d'étranglement des E/S auquel nous sommes confrontés dans notre programme
- Nous passons énormément de temps à attendre sur nos demandes réseau, et une fraction de ce temps à analyser notre page récupérée pour d'autres liens à explorer

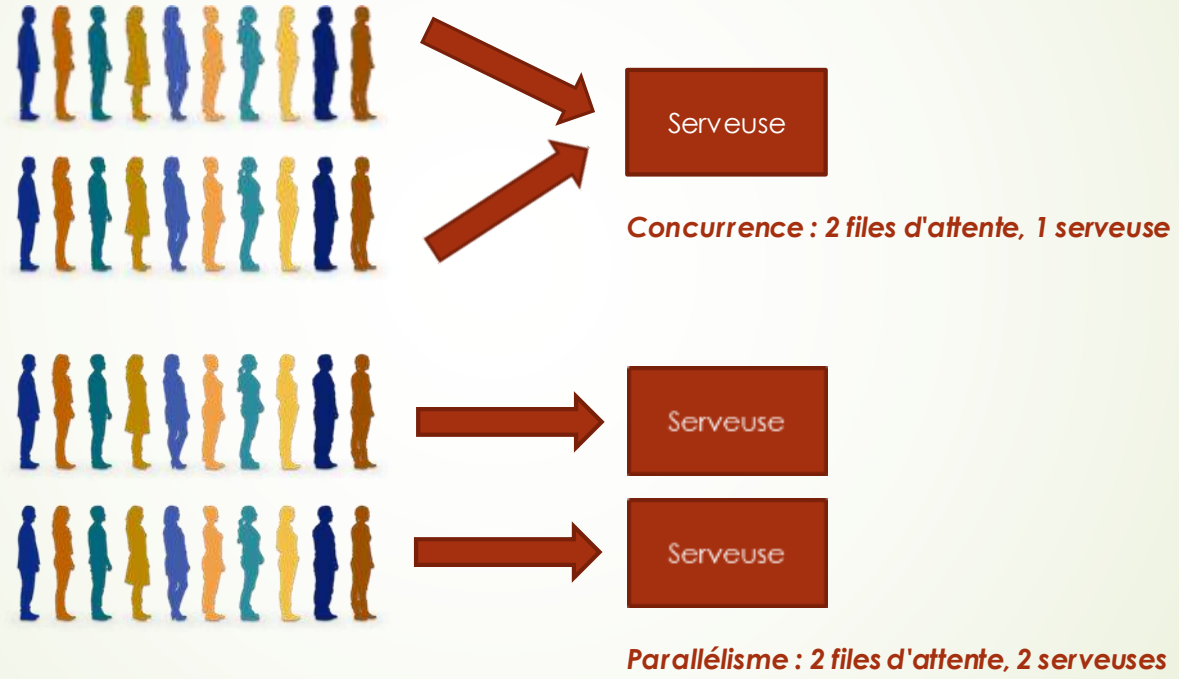
Comprendre le parallélisme

- Au début, nous avons abordé un peu les capacités de multitraitement de Python, et comment nous pourrions l'utiliser pour tirer parti d'un plus grand nombre de cœurs de traitement dans notre matériel
- Mais que voulons-nous dire lorsque nous disons que nos programmes fonctionnent en parallèle ?
- Le parallélisme est l'art d'exécuter deux ou plusieurs actions simultanément, par opposition à la simultanéité dans laquelle vous faites des progrès sur deux ou plusieurs choses en même temps
- C'est une distinction importante, et afin d'obtenir un véritable parallélisme, nous aurons besoin de plusieurs processeurs pour exécuter nos codes en même temps

Comprendre le parallélisme

- Une bonne analogie pour le traitement parallèle est de penser à une file d'attente pour le pain
- Si vous avez, par exemple, deux files d'attente de 20 personnes, toutes en attente d'avoir une baguette servie par la boulangère, ce serait un exemple de concurrence
- Maintenant, ajoutez une deuxième serveuse dans la boulangerie, ce serait alors un exemple de quelque chose qui se passe en parallèle
- C'est exactement comme cela que fonctionne le traitement en parallèle - chacune des serveuses dans cette boulangerie représente un noyau de traitement, et est capable de faire des progrès sur les tâches simultanément

Comprendre le parallélisme



Comprendre le parallélisme

- Un exemple concret qui met en évidence la véritable puissance du traitement parallèle est la carte graphique de votre ordinateur
- Ces cartes graphiques ont tendance à avoir des centaines, voire des milliers, de cœurs de traitement individuels qui vivent indépendamment, et peuvent calculer des choses en même temps
- La raison pour laquelle nous sommes en mesure d'exécuter des jeux PC haut de gamme à de telles cadences est due au fait que nous avons pu mettre autant de cœurs parallèles sur ces cartes

Goulots d'étranglement CPU

- Un goulot d'étranglement lié à l'UC est généralement l'inverse d'un goulot d'étranglement lié aux E/S
- Ce goulot d'étranglement se trouve dans les applications qui effectuent beaucoup de calculs fastidieux ou toute autre tâche coûteuse en termes de calcul
- Ce sont des programmes pour lesquels la vitesse à laquelle ils s'exécutent est liée à la vitesse du CPU
- Si vous lancez un CPU plus rapide dans votre machine, vous devriez voir une augmentation directe de la vitesse de ces programmes
- ***Si le taux de traitement des données dépasse de loin celui auquel vous demandez des données, vous avez un goulot d'étranglement lié aux CPU***

Concurrence et Parallélisme sur le CPU

- Comprendre les différences décrites dans les sections précédentes entre la concurrence et le parallélisme est essentiel, mais il est également très important de mieux comprendre les systèmes sur lesquels votre logiciel fonctionnera
- Avoir une appréciation des différents styles d'architecture ainsi que la mécanique de bas niveau vous aide à prendre les décisions les plus éclairées dans la conception de votre logiciel

Single-core CPUs

- Les processeurs à un seul cœur n'exécuteront jamais qu'un seul thread à la fois, car c'est tout ce dont ils sont capables
- Cependant, afin de s'assurer que nous ne voyons pas nos applications suspendues et qu'elles ne répondent pas, ces processeurs basculent rapidement entre plusieurs threads d'exécution plusieurs milliers de fois par seconde
- Cette commutation entre threads est ce que l'on appelle un «changement de contexte», et consiste à stocker toutes les informations nécessaires pour un thread à un moment donné, puis à le restaurer à un point différent plus bas sur la ligne
- L'utilisation de ce mécanisme d'enregistrement et de restauration en continu des threads nous permet de progresser sur un certain nombre de threads dans une seconde donnée, et il semble que l'ordinateur fasse plusieurs choses à la fois
- En fait, il ne fait qu'une chose à la fois, mais à un rythme tel qu'il est imperceptible pour les utilisateurs de cette machine

Single-core CPUs

- Lors de l'écriture d'applications multithread en Python, il est important de noter que ces commutateurs de contexte sont, par calcul, assez coûteux
- Il n'y a malheureusement aucun moyen de contourner cela, et une grande partie de la conception des systèmes d'exploitation de nos jours consiste à optimiser ces commutateurs de contexte afin de ne pas ressentir autant la douleur

Single-core CPUs

- Les avantages des processeurs mono cœur sont les suivants :
  - Ils ne nécessitent pas de protocoles de communication complexes entre plusieurs cœurs
  - Les processeurs mono cœur nécessitent moins de puissance, ce qui les rend plus adaptés aux périphériques du type IoT
- Les processeurs mono cœurs présentent toutefois les inconvénients suivants :
  - Leur vitesse est limitée et les applications plus volumineuses les empêchent de fonctionner et de geler
  - Les problèmes de dissipation thermique imposent une limite stricte à la vitesse à laquelle un processeur mono cœur peut fonctionner

Fréquence d'horloge

- L'une des principales limitations d'une application mono cœur s'exécutant sur une machine est la vitesse d'horloge du processeur
- Quand nous parlons de la fréquence d'horloge, nous parlons essentiellement du nombre de cycles d'horloge qu'un processeur peut exécuter chaque seconde
- Au cours des 10 dernières années, nous avons observé que les fabricants réussissaient à surpasser la loi de Moore, qui consistait essentiellement à observer que le nombre de transistors que l'on pouvait placer sur un morceau de silicium doublait à peu près tous les deux ans
- Ce doublement des transistors tous les deux ans a ouvert la voie à des gains exponentiels dans les fréquences d'horloge monoprocesseur, et les processeurs sont passés de la fréquence basse à la fréquence 4-5 GHz que nous voyons maintenant sur le processeur i7 6700k d'Intel

Fréquence d'horloge

- Mais avec des transistors de seulement quelques nanomètres de diamètre, cela arrive inévitablement à sa fin
- Nous avons commencé à dépasser les limites de la physique et, malheureusement, si nous allons plus loin, nous commencerons à être affectés par les effets du tunnel quantique
- En raison de ces limitations physiques, nous devons commencer à regarder d'autres méthodes afin d'améliorer les vitesses auxquelles nous sommes capables de calculer les choses
- C'est ici qu'intervient le modèle d'extensibilité de Martelli

Fréquence d'horloge

- Mais avec des transistors de seulement quelques nanomètres de diamètre, cela arrive inévitablement à sa fin
- Nous avons commencé à dépasser les limites de la physique et, malheureusement, si nous allons plus loin, nous commencerons à être affectés par les effets du tunnel quantique
- En raison de ces limitations physiques, nous devons commencer à regarder d'autres méthodes afin d'améliorer les vitesses auxquelles nous sommes capables de calculer les choses
- C'est ici qu'intervient le modèle d'extensibilité de Martelli

Modèle d'extensibilité de Martelli

- L'auteur du "Python Cookbook", Alex Martelli, a proposé un modèle sur l'évolutivité, dont Raymond Hettinger a parlé dans son brillant exposé d'une heure sur "Thinking about Concurrency" qu'il a donné à la PyCon Russia 2016
- Ce modèle représente trois types de problèmes et programmes :
  - 1 cœur : Ceci fait référence aux programmes mono-thread et single-process
  - 2-8 cœurs: Ceci fait référence aux programmes multithread et multitraitement
  - 9+ cœurs: Ceci fait référence au calcul distribué

Modèle d'extensibilité de Martelli

- La première catégorie, la catégorie mono cœur à un seul thread, est capable de gérer un nombre croissant de problèmes en raison de l'amélioration constante de la vitesse des processeurs à un seul cœur, et par conséquent, la seconde catégorie est rendue de plus en plus obsolète
- Nous atteindrons finalement une limite avec la vitesse à laquelle un système de base de 2-8 peut fonctionner, et nous devrons alors commencer à regarder d'autres méthodes, telles que plusieurs systèmes de CPU ou même l'informatique distribuée
- Si votre problème vaut la peine d'être résolu rapidement et nécessite beaucoup de puissance, alors l'approche sensée consiste à utiliser la catégorie informatique répartie et à faire tourner plusieurs machines et plusieurs instances de votre programme afin de résoudre vos problèmes de manière vraiment parallèle

Modèle d'extensibilité de Martelli

- Les grands systèmes d'entreprise qui traitent des centaines de millions de requêtes sont les principaux habitants de cette catégorie
- Vous trouverez généralement que ces systèmes d'entreprise sont déployés sur des dizaines, voire des centaines, de serveurs haute performance et incroyablement puissants dans divers endroits à travers le monde

Partage de temps - le planificateur de tâches

- L'un des éléments les plus importants du système d'exploitation est le planificateur de tâches
- Il agit comme le chef d'orchestre, et dirige tout avec une précision impeccable et un timing et une discipline incroyables
- Ce maestro n'a qu'un seul but réel, c'est de s'assurer que chaque tâche a une chance de se terminer jusqu'à l'achèvement
- Le moment et le lieu de l'exécution d'une tâche sont cependant non déterministes
- C'est-à-dire, si nous avons donné à un planificateur de tâches deux processus concurrents identiques l'un après l'autre, il n'y a aucune garantie que le premier processus se terminera en premier
- Cette nature non déterministe est ce qui rend la programmation concurrente si difficile

Partage de temps - le planificateur de tâches

- Un excellent exemple qui met en évidence ce comportement non-déterministe est le code ci-contre
- Ici, nous avons deux threads concurrents en Python qui tentent chacun d'atteindre leur propre objectif :
- décrémenter le compteur à 1 000 ou, inversement, l'augmenter à 1 000
- Dans un processeur mono cœur, il y a la possibilité que le travailleur A réussisse à terminer sa tâche avant que le travailleur B ait une chance d'exécuter, et la même chose peut être dite pour le travailleur B
- Cependant, il y a une troisième possibilité potentielle, et c'est que le planificateur de tâches continue à basculer entre le travailleur A et le travailleur B un nombre infini de fois et ne jamais terminer

concurrency7.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random

counter = 1


def worker_a():
    global counter
    while counter < 1000:
        counter += 1
        print("Worker A is incrementing counter to {}".format(counter))
        sleep_time = random.randint(0, 1)
        time.sleep(sleep_time * .5)


def worker_b():
    global counter
    while counter > -1000:
        counter -= 1
        print("Worker B is decrementing counter to {}".format(counter))
        sleep_time = random.randint(0, 1)
        time.sleep(sleep_time * .5)


def main():
    t0 = time.time()
    thread1 = threading.Thread(target=worker_a)
    thread2 = threading.Thread(target=worker_b)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    t1 = time.time()
    print("Execution time {}".format(t1 - t0))


if __name__ == '__main__':
    main()

```

Partage de temps - le planificateur de tâches

- Le code précédent, incidemment, montre également l'un des dangers de plusieurs threads accédant à des ressources partagées sans aucune forme de synchronisation
- Il n'y a aucun moyen précis de déterminer ce qui arrivera à notre comptoir, et à ce titre, notre programme pourrait être considéré comme peu fiable

Processeurs multi-cœurs

- Nous avons maintenant une idée de la façon dont les processeurs mono-cœur fonctionnent, mais maintenant il est temps de jeter un coup d'œil aux processeurs multi-cœurs
- Les processeurs multi-cœurs contiennent plusieurs unités de traitement indépendantes ou «noyaux»
- Chaque noyau contient tout ce dont il a besoin pour exécuter une séquence d'instructions stockées

Processeurs multi-cœurs

- Ces noyaux suivent chacun leur propre cycle, qui consiste en le processus suivant :
  - **Récupérer :** Cette étape consiste à récupérer des instructions dans la mémoire du programme. Ceci est dicté par un compteur de programme (PC), qui identifie l'emplacement de l'étape suivante à exécuter
  - **Décoder :** Le noyau convertit l'instruction qu'il vient d'extraire et la convertit en une série de signaux qui déclencheront d'autres parties du processeur
  - **Exécuter :** Enfin, nous exécutons l'étape d'exécution. C'est ici que nous exécutons l'instruction que nous venons d'extraire et de décoder, et les résultats de cette exécution sont ensuite stockés dans un registre CPU
- Avoir plusieurs cœurs nous offre l'avantage de pouvoir travailler indépendamment sur plusieurs cycles Fetch -> Decode -> Execute
- Ce style d'architecture nous permet de créer des programmes plus performants qui tirent parti de cette exécution parallèle

Processeurs multi-cœurs

- Les avantages des processeurs multi-cœurs sont les suivants :
  - Nous ne sommes plus limités par les mêmes limitations de performances qu'un processeur mono-cœur
  - Les applications capables de tirer parti de plusieurs cœurs auront tendance à fonctionner plus rapidement si elles sont bien conçues
- Les inconvénients des processeurs multi-cœurs :
  - Ils nécessitent plus de puissance que votre processeur mono-coeur typique
  - La communication inter-coeurs n'est pas d'une exploitation simple
  - Nous avons plusieurs façons de le faire, sur lesquelles nous reviendrons

Styles d'architecture de systèmes

- Lors de la conception de vos programmes, il est important de noter qu'il existe un certain nombre de styles d'architecture de mémoire différents qui répondent aux besoins d'une gamme de cas d'utilisation différents
- Un style d'architecture de la mémoire peut être excellent pour les tâches de calcul parallèle et l'informatique scientifique, mais il est quelque peu compliqué en ce qui concerne vos tâches informatiques domestiques standard

Styles d'architecture de systèmes

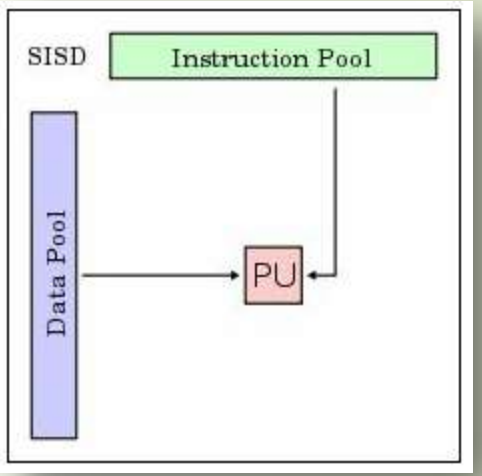
- Lorsque nous catégorisons ces différents styles, nous avons tendance à suivre une taxonomie proposée par un homme nommé Michael Flynn en 1972
- Cette taxonomie définit quatre styles différents d'architecture informatique
- Ceux-ci sont :
  - SISD : single instruction stream, single data stream
  - SIMD : single instruction stream, multiple data stream
  - MISD : multiple instruction stream, single data stream
  - MIMD: multiple instruction stream, multiple data stream

SISD

- Flux d'instructions uniques, flux de données uniques ont tendance à être vos systèmes monoprocesseur
- Ces systèmes ont un flux séquentiel de données qui les reçoit, et une seule unité de traitement qui est utilisée pour exécuter ce flux
- Ce style d'architecture représente généralement les machines "Von Neumann" classiques, et pendant un grand nombre d'années, avant que les processeurs multi-coeurs deviennent populaires, cela représentait un ordinateur domestique typique
- Vous avez un seul processeur qui gère tout ce dont vous avez besoin
- Ceux-ci, cependant, seraient incapables de choses telles que le parallélisme d'instruction et le parallélisme de données, et des choses telles que le traitement de graphiques étaient incroyablement imposantes sur ces systèmes

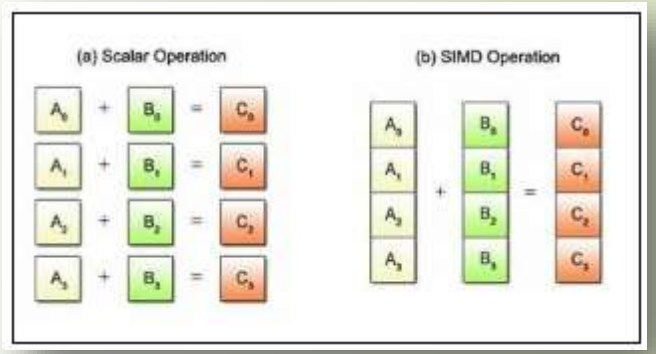
SISD

- La figure suivante montre un aperçu de l'aspect d'un système monoprocesseur
- Il comporte une source de données traitée par une seule unité de traitement
- Ce style d'architecture présente tous les avantages et les inconvénients que nous avons soulignés plus haut dans le chapitre lorsque nous couvrions les processeurs mono-cœur
- Un exemple de monoprocesseur pourrait être l'Intel Pentium 4.



SIMD

- SIMD (flux d'instructions unique, flux de données multiples)
- L'architecture de flux de données multiples est la mieux adaptée pour travailler avec des systèmes qui traitent beaucoup de multimédia
- Ceux-ci sont idéaux pour faire des choses telles que les graphiques 3D en raison de la façon dont ils peuvent manipuler des vecteurs
- Par exemple, disons que vous avez deux tableaux distincts, [10,15,20,25] et [20, 15,10, 5]
- Dans une architecture SIMD, vous pouvez les ajouter en une opération pour obtenir [30,30,30,30]
- Si nous devions le faire sur l'architecture scalaire, nous devions effectuer quatre opérations d'ajout distinctes, comme illustré dans la figure ci-contre

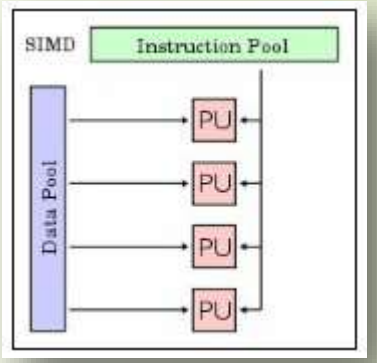


SIMD

- Le meilleur exemple de ce style d'architecture peut être trouvé dans votre unité de traitement graphique
- Dans la programmation graphique OpenGL, vous avez des objets appelés Vertex Array Objects ou VAO, et ces VAO contiennent généralement plusieurs Vertex Buffer Objects qui décrivent un objet 3D donné dans un jeu
- Si quelqu'un devait, par exemple, déplacer son personnage, chaque élément de chaque objet Vertex Buffer devrait être recalculé incroyablement rapidement afin de nous permettre de voir le personnage se déplacer doucement sur nos écrans
- C'est là que brille la puissance de l'architecture SIMD
- Nous passons tous nos éléments dans des VAO distincts
- Une fois que ces VAO ont été peuplés, nous pouvons alors dire que nous voulons tout multiplier dans ce VAO avec une matrice de rotation
- Cela conduit alors très rapidement à effectuer la même action sur chaque élément beaucoup plus efficacement qu'une architecture non-vectorielle ne pourrait le faire

SIMD

- Le diagramme suivant montre une vue d'ensemble de haut niveau d'une architecture SIMD
- Nous avons plusieurs flux de données, qui peuvent représenter plusieurs vecteurs, et un certain nombre d'unités de traitement, toutes capables d'agir sur une seule instruction à un moment donné
- Les cartes graphiques ont généralement des centaines d'unités de traitement individuelles
- Les principaux avantages de SIMD sont les suivants :
  - Nous sommes en mesure d'effectuer la même opération sur plusieurs éléments en utilisant une instruction
  - Comme le nombre de cœurs sur les cartes graphiques modernes augmente, ainsi le débit de ces cartes, grâce à cette architecture utiliser tous les avantages de ce style d'architecture

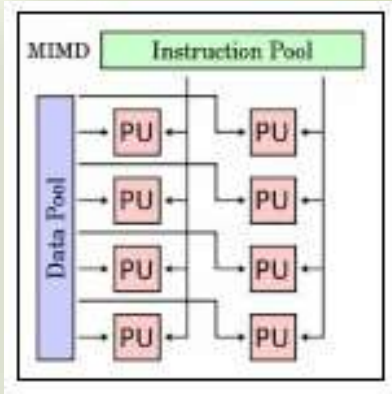


MISD

- Les flux d'instructions multiples, les flux de données uniques ou le MISD est un style d'architecture quelque peu mal aimé, sans aucun exemple réel actuellement disponible commercialement
- Il est généralement assez difficile de trouver un cas d'utilisation dans lequel un style d'architecture MISD est approprié, et se prêterait bien à un problème
- Aucun véritable exemple d'architecture MISD n'est disponible commercialement aujourd'hui

MIMD

- Les flux d'instructions multiples, les flux de données multiples constituent la taxonomie la plus diversifiée et encapsulent tous les processeurs multi-cœurs modernes
- Chacun des cœurs qui composent ces processeurs est capable de fonctionner indépendamment et en parallèle
- Contrairement à nos machines SIMD, les machines basées sur MIMD sont capables d'exécuter un certain nombre d'opérations distinctes sur plusieurs jeux de données en parallèle, par opposition à une seule opération sur plusieurs jeux de données
- Le diagramme ci-contre montre un exemple d'un certain nombre d'unités de traitement différentes, toutes avec un certain nombre de flux de données d'entrée différents agissant tous indépendamment
- Un multiprocesseur normal utilise typiquement l'architecture MIMD.



Style d'architecture de mémoire informatique

- Lorsque nous commençons à accélérer nos programmes en introduisant des concepts tels que la concurrence et le parallélisme, nous commençons à faire face à de nouveaux défis qui doivent être réfléchis et traités de manière appropriée
- L'un des plus grands défis auxquels nous sommes confrontés est la vitesse à laquelle nous pouvons accéder aux données
- Il est important de noter à ce stade que si nous ne pouvons pas accéder aux données assez rapidement, cela devient un goulot d'étranglement pour nos programmes, et peu importe comment nous concevons nos systèmes de façon experte, nous ne verrons jamais de gains de performance

Style d'architecture de mémoire informatique

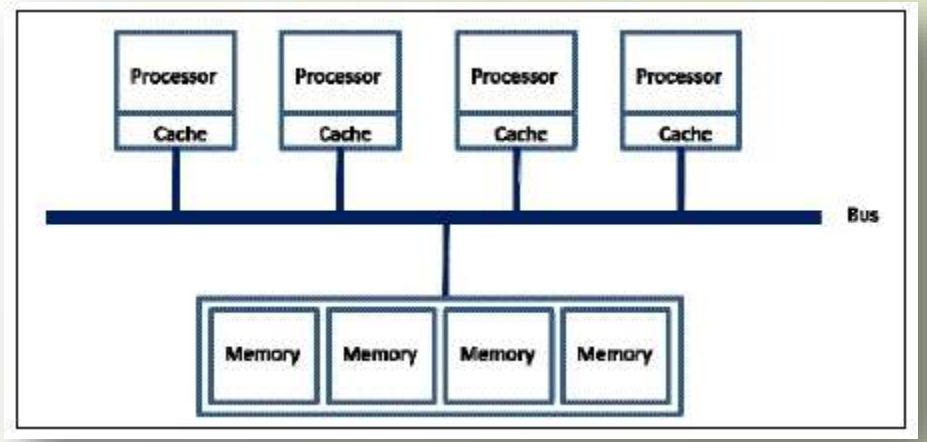
- Les concepteurs d'ordinateurs recherchent de plus en plus des moyens d'améliorer la facilité avec laquelle nous pouvons développer de nouvelles solutions parallèles aux problèmes
- L'une des façons dont ils ont réussi à améliorer les choses est de fournir un espace d'adressage physique unique auquel tous nos multiples cœurs peuvent accéder au sein d'un processeur
- Cela nous enlève une certaine quantité de complexité, en tant que programmeurs, et nous permet de nous concentrer plutôt sur la sécurité de notre code
- Il existe un certain nombre de ces différents styles d'architecture utilisés dans un large éventail de scénarios différents
- Les deux styles architecturaux principaux utilisés par les concepteurs de systèmes ont tendance à être ceux qui suivent un modèle d'accès mémoire uniforme ou un modèle d'accès mémoire non uniforme, respectivement UMA et NUMA

UMA

- UMA (Uniform Memory Access) est un style d'architecture qui dispose d'un espace mémoire partagé pouvant être utilisé de manière uniforme par n'importe quel nombre de cœurs de traitement
- En termes simples, cela signifie qu'indépendamment de l'endroit où ce noyau réside, il pourra accéder directement à un emplacement mémoire en même temps, quelle que soit la proximité de la mémoire
- Ce style d'architecture est également connu sous le nom de multiprocesseurs à mémoire partagée symétrique ou SMP en abrégé.

UMA

- L'image suivante montre comment un système de type UMA pourrait être assemblé
- Chaque processeur s'interface avec un bus, qui effectue l'ensemble de l'accès à la mémoire
- Chaque processeur ajouté à ce système augmente la pression sur la bande passante du bus, et nous ne sommes donc pas en mesure de l'adapter de la même manière que nous le ferions si nous utilisions une architecture NUMA



UMA

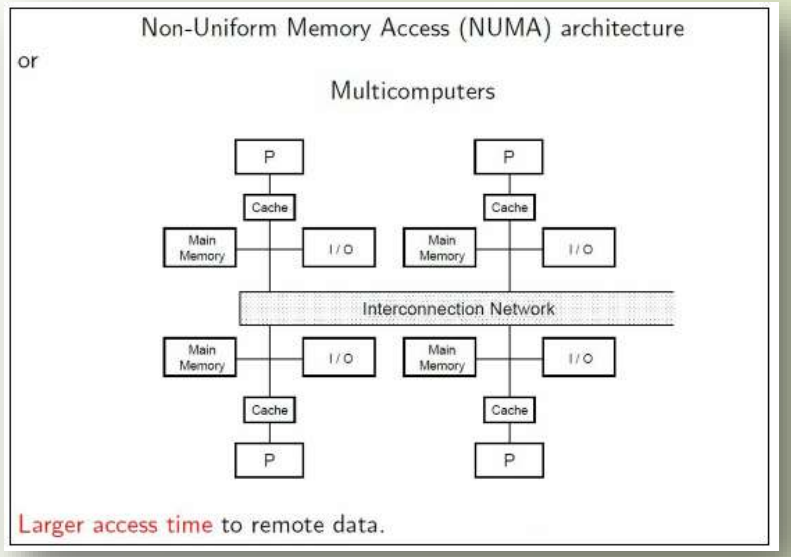
- Les avantages de l'UMA sont les suivants :
  - Tout l'accès à la RAM prend exactement le même laps de temps
  - Le cache est cohérent et consistant
  - La conception du matériel est plus simple
- Cependant, il existe un inconvénient de UMA :
  - les systèmes UMA disposent d'un bus mémoire à partir duquel tous les systèmes accèdent à la mémoire
  - Malheureusement, cela présente des problèmes de mise à l'échelle

NUMA

- NUMA (accès mémoire non uniforme) est un style d'architecture dans lequel l'accès à la mémoire peut être plus rapide que d'autres en fonction du processeur demandé
- Cela peut être dû à l'emplacement du processeur par rapport à la mémoire
- Juste après, un diagramme qui montre exactement comment un certain nombre de processeurs s'interconnectent dans le style NUMA
- Chacun a son propre cache, l'accès à la mémoire principale, et des E/S indépendantes, et chacun est connecté au réseau d'interconnexion

NUMA

- NUMA présente un avantage majeur :
  - Les machines NUMA sont plus évolutives que leurs homologues à accès uniforme
- Les inconvénients de NUMA sont les suivants :
  - Des temps d'accès mémoire non déterministes peuvent conduire à des temps d'accès très courts si la mémoire est locale ou beaucoup plus longue fois si la mémoire est dans des emplacements de mémoire distants
  - Les processeurs doivent observer les modifications apportées par d'autres processeurs
  - Le temps qu'il faut pour observer ces modifications augmente en fonction du nombre de processeurs qui en font partie



Cycle de vie d'un thread

Python Concurrence

Vie d'un thread

- Nous avons examiné en profondeur les concepts de concurrence et de parallélisme, ainsi que certains des problèmes clés auxquels nous sommes confrontés dans les applications Python multithread
- Il est maintenant temps de voir comment nous pouvons commencer à travailler avec des threads et les manipuler selon notre volonté.
- Nous plongerons maintenant dans la vie d'un thread
- Nous aborderons différents sujets tels que :
  - Les différents états d'un thread dans Différents types de threads - Windows vs POSIX
  - Les meilleures pratiques quand il s'agit de démarrer vos propres threads
  - Comment nous pouvons vous faciliter la vie quand il s'agit de travailler avec de nombreux threads
  - Enfin, nous verrons comment nous pouvons terminer les threads et les différents modèles de multithreading

Les threads en Python

- Avant d'entrer dans le détail de la vie d'un thread, il est important de savoir ce que nous allons instancier en termes réels
- Pour le savoir, cependant, nous aurons besoin de jeter un œil à la définition de la classe ***Thread*** de Python qui peut être trouvée dans ***threading.py***
- Dans ce fichier, vous devriez voir la définition de classe pour la classe Thread
- Elle a une fonction constructeur qui ressemble à ceci :

```
755     def __init__(self, group=None, target=None, name=None,
756                  args=(), kwargs=None, *, daemon=None):
```

Les threads en Python

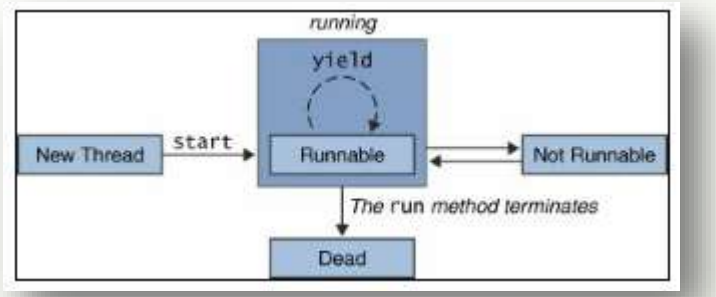
- Ce constructeur précédent prend en compte cinq arguments réels, qui sont définis comme suit dans cette définition de classe :
  - ***group*** : C'est un paramètre spécial qui est réservé pour une future extension
  - ***target*** : Ceci est l'objet appelable à appeler par la méthode ***run()***. Si il n'est pas transmis, cette valeur par défaut est None et rien ne sera démarré
  - ***name***: Ceci est le nom du thread
  - ***args*** : C'est l'argument tuple pour l'invocation de la cible. Il vaut par défaut ***()***
  - ***kwargs*** : Ceci est un dictionnaire d'arguments de mots-clés pour appeler le constructeur de la classe de base

Etats possibles des threads

- Les threads peuvent exister dans cinq états distincts : running, not-running, runnable, starting et ended
- Lorsque nous créons un thread, nous n'avons généralement pas alloué de ressources à ce thread pour le moment
- Il n'existe dans aucun état, car il n'a pas été initialisé, et il ne peut être démarré ou arrêté
- ***Nouveau thread*** : Dans l'état du nouveau thread, notre thread n'a pas démarré et aucune ressource n'a été allouée. C'est simplement une instance d'un objet
- ***Executable*** : C'est l'état où le thread est en attente d'exécution, il dispose de toutes les ressources nécessaires pour continuer, et la seule chose qui le retient est que le planificateur de tâches ne l'a pas planifié
- ***Exécution*** : dans cet état, le thread fait des progrès - il exécute la tâche pour laquelle il a été conçu et a été choisi par le planificateur de tâches pour l'exécuter. De cet état, notre thread peut entrer dans un état mort si nous choisissons de le tuer, ou il pourrait entrer dans un état ***Not-running***
- ***Not-running*** : C'est quand le thread a été suspendu d'une manière ou d'une autre. Cela peut être dû à un certain nombre de raisons, telles que l'attente de la réponse d'une demande d'E/S longue durée. Ou il pourrait être délibérément bloqué jusqu'à ce qu'un autre thread ait terminé son exécution
- ***Dead*** : Un thread peut atteindre cet état de deux manières différentes. Il peut, tout comme nous, mourir de causes naturelles ou être tué de façon anormale

Etats possibles des threads

- Le diagramme suivant représente les cinq états différents dans lesquels un thread peut se trouver ainsi que les transitions possibles d'un état à un autre :



Etats possibles des threads

- Alors maintenant que nous connaissons les différents états dans lesquels nos threads peuvent être, comment cela se traduit-il dans nos programmes Python ?
- Jetez un œil au code suivant :

threads_states.py

```
7
8   def worker():
9       print("Thread entered 'Runnig' state")
10      # Entering 'Not runnable' state
11      time.sleep(3)
12      # Thread completes its task and terminates
13      print("Thread is terminating")
14
15
16  def main():
17      # At this point in time, the thread has no state
18      # it hasn't been allocated any system resources
19      t = threading.Thread(target=worker)
20      # When we call myThread.start(), Python allocates
21      # the necessary system
22      # resources in order for our thread to run and
23      # then calls the thread's # run method.
24      # It goes from 'Starting' state to 'Runnable'
25      # but not running
26      t.start()
27      # Here we join the thread and when this method
28      # is called # our thread goes into a 'Dead' state.
29      # It has finished the job that it was intended to do.
30      t.join()
31      print("Thread has entered a 'Dead' state")
32
33
34  if __name__ == '__main__':
35      main()
36
```

Différents types de threads

- Python simplifie la plupart des complications des API de threads de niveau inférieur et nous permet de nous concentrer sur la création de systèmes encore plus complexes
- Non seulement cela, il nous permet d'écrire du code portable qui peut tirer parti des threads POSIX ou Windows en fonction du système d'exploitation sur lequel nous exécutons notre code
- Mais que voulons-nous dire quand nous mentionnons des choses comme les threads POSIX ou les threads Windows ?

Threads POSIX

- Quand nous parlons de threads POSIX, nous parlons de threads qui sont implémentés pour suivre la norme IEEE POSIX 1003.1c
- Cette norme a été enregistrée en tant que marque déposée de la fondation IEEE et a été initialement développée afin de standardiser l'implémentation des threads sur une gamme de matériel sur les systèmes UNIX
- Toutes les implémentations de threads qui suivent cette norme sont généralement appelées threads POSIX ou PThreads

Démarrer un thread

- En Python, il existe plusieurs façons de démarrer un thread
- Lorsque nous avons une tâche relativement simple que nous souhaitons multithread, nous avons la possibilité de la définir comme une seule fonction
- Dans l'exemple suivant, nous avons une fonction très simple qui dort juste pour un intervalle de temps aléatoire
- Cela représente une fonctionnalité très simple, et est idéal pour encapsuler une fonction simple, puis passer cette fonction simple comme cible pour un nouvel objet ***threading.Thread*** comme vu dans le code précédent

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random


def execute_thread(i):
    print("Thread {} started".format(i))
    sleepTime = random.randint(1, 10)
    time.sleep(sleepTime)
    print("Thread {} finished executing".format(i))


def main():
    for i in range(10):
        thread = threading.Thread(target=execute_thread, args=(i,))
        thread.start()
    for t in threading.enumerate():
        print("Active Thread:", t)


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

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 5 started
Thread 6 started
Thread 7 started
Thread 8 started
Thread 9 started
Active Thread: <_MainThread(MainThread, started 8160)>
Active Thread: <Thread(Thread-2, started 9280)>
Active Thread: <Thread(Thread-7, started 1668)>
Active Thread: <Thread(Thread-10, started 15400)>
Active Thread: <Thread(Thread-9, started 15040)>
Active Thread: <Thread(Thread-4, started 9004)>
Active Thread: <Thread(Thread-5, started 6272)>
Active Thread: <Thread(Thread-6, started 7004)>
Active Thread: <Thread(Thread-1, started 9772)>
Active Thread: <Thread(Thread-8, started 1596)>
Active Thread: <Thread(Thread-3, started 1120)>
Thread 2 finished executing
Thread 5 finished executing
Thread 1 finished executing
Thread 8 finished executing
Thread 6 finished executing
Thread 9 finished executing
Thread 3 finished executing
Thread 7 finished executing
Thread 0 finished executing
Thread 4 finished executing
```

Hériter de la classe *thread*

- Pour les scénarios qui nécessitent plus de code que ce qui peut être encapsulé dans une seule fonction, nous pouvons en fait définir une classe qui hérite directement de la classe native des threads
- Ceci est idéal pour les scénarios où la complexité du code est trop grande pour une seule fonction, et doit plutôt être divisée en plusieurs fonctions
- Bien que cela nous donne plus de flexibilité dans le traitement des threads, nous devons prendre en compte le fait que nous devons maintenant gérer notre thread dans cette classe

Hériter de la classe *thread*

- Pour que nous puissions définir un nouveau thread qui hérite de la classe de thread native de Python, nous devons faire ce qui suit au strict minimum :
- Transmettre la classe thread à notre définition de classe
- Appeler ***Thread.__init__(self)*** dans notre constructeur afin que notre thread puisse s'initialiser
- Définir une fonction ***run()*** qui sera appelée lorsque notre thread sera démarré

threads-inherit.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  from threading import Thread
6
7
8  class WorkerThread(Thread):
9      def __init__(self):
10         super().__init__()
11
12     def run(self):
13         print("Thread is now running")
14
15
16 def main():
17     worker_thread = WorkerThread()
18     print("Created the Thread object")
19     worker_thread.start()
20     print("Started thread")
21     worker_thread.join()
22     print("Finished thread")
23
24
25 if __name__ == '__main__':
26     main()
27
```

Forking

- Forker un processus est créer une deuxième réplique exacte du processus donné
- En d'autres termes, lorsque nous forkons quelque chose, nous le clonons et l'exécutons comme un processus enfant du processus que nous venons de cloner
- Ce processus nouvellement créé obtient son propre espace d'adressage ainsi qu'une copie exacte des données du parent et du code exécuté dans le processus parent
- Une fois créé, ce nouveau clone reçoit son propre ID de processus (PID) et est indépendant du processus parent à partir duquel il a été cloné
- ***Ce mécanisme n'est pas directement disponible sous Windows (implémenté dans CygWin seulement)***

fork_process.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import os


def child():
    print("We are in the child process with PID= %d" % os.getpid())


def parent():
    print("We are in the parent process with PID= %d" % os.getpid())
    newRef = os.fork()
    if newRef == 0:
        child()
    else:
        print("We are in the parent process and our child process has PID= %d" % newRef)

def main():
    parent()


if __name__ == '__main__':
    main()

```

```
==> python3 fork_process.py
We are in the parent process with PID= 6736
We are in the parent process and our child process has PID= 9132
We are in the child process with PID= 9132
deckert@LAPTOP-FEJV2L8L /cygdrive/d/pythonworkspace/learning-python/samples
==>
```

Forking

- Pourquoi voudriez-vous cloner un processus existant ?
- Si vous avez déjà fait une forme quelconque d'hébergement de site Web, vous avez probablement déjà rencontré Apache
- Apache utilise fortement le forking pour créer plusieurs processus serveur
- Chacun de ces processus indépendants est capable de gérer ses propres demandes dans son propre espace d'adressage
- Ceci est idéal dans ce scénario, car il nous donne une certaine protection dans la mesure où, en cas de panne ou de blocage d'un processus, les autres processus qui s'exécutent en même temps ne sont pas affectés et peuvent continuer à répondre aux nouvelles demandes

Démoniser un thread

- Premièrement, avant de regarder les threads démons, je pense qu'il est important de savoir de quoi il s'agit
- Les threads démons sont des threads qui, par définition, n'ont pas de point de terminaison défini
- Ils continueront à fonctionner indéfiniment jusqu'à ce que votre programme se termine
- Pourquoi est-ce utile ?
- Supposons, par exemple, que vous ayez un équilibreur de charge qui envoie des demandes de service à plusieurs instances de votre application

Démoniser un thread

- Vous pouvez avoir une forme de service de registre qui permet à votre équilibreur de charge de savoir où envoyer ces demandes, mais comment ce registre de service connaît-il l'état de votre instance ?
- Généralement, dans ce scénario, nous envoyons à intervalles réguliers un paquet appelé "pulsation" ou "keep alive" pour indiquer à notre service de registre : "Hey, je suis toujours 200!"
- Cet exemple est un cas d'utilisation principal pour les threads démon au sein de notre application
- Nous pourrions migrer le travail d'envoi d'un signal de pulsation vers notre registre de service à un thread de démon, et le lancer lorsque notre application démarre
- Ce thread démon sera ensuite placé en arrière-plan de notre programme, et enverra périodiquement cette mise à jour sans aucune intervention de notre part
- Ce qui est encore mieux, c'est que notre thread démon sera tué sans que nous ayons à nous en soucier quand notre instance s'arrêtera

Démoniser un thread

- Exemple de thread démon

```
Standard thread started
Sending heartbeat signal 1
Sending heartbeat signal 2
Sending heartbeat signal 3
Sending heartbeat signal 4
Sending heartbeat signal 5
Sending heartbeat signal 6
Sending heartbeat signal 7
Sending heartbeat signal 8
Sending heartbeat signal 9
Sending heartbeat signal 10
Standard thread stopped
```

daemon_thread.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time


def standard_thread():
    print("Standard thread started")
    time.sleep(20)
    print("Standard thread stopped")


def daemon_thread():
    i = 1
    while True:
        print("Sending heartbeat signal {}".format(i))
        i += 1
        time.sleep(2)


def main():
    standardThread = threading.Thread(target=standard_thread)
    daemonThread = threading.Thread(target=daemon_thread)
    daemonThread.setDaemon(True)
    standardThread.start()
    daemonThread.start()


if __name__ == '__main__':
    main()

```

Démarrage de nombreux threads

- Le premier exemple que nous verrons est comment nous pouvons démarrer plusieurs threads en même temps
- Nous pouvons créer plusieurs objets threads en utilisant une boucle ***for***, puis en les démarrant dans la même boucle ***for***
- Dans l'exemple suivant, nous définissons une fonction qui prend un nombre entier et qui dort pendant une durée aléatoire, en imprimant les deux quand elle commence et se termine
- Nous créons ensuite une boucle for qui boucle jusqu'à 10 et crée 10 objets thread distincts dont la cible est définie sur notre fonction execute_thread
- Il démarre ensuite l'objet thread que nous venons de créer, puis nous imprimons les threads actifs en cours.

Démarrage de nombreux threads

- Exemple

multiple_threads.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random


def execute_thread(i):
    print("Thread {} started".format(i))
    sleep_time = random.randint(1, 10)
    time.sleep(sleep_time)
    print("Thread {} finished".format(i))


def main():
    for i in range(10):
        thread = threading.Thread(target=execute_thread, args=(i,))
        thread.start()
        print("Active threads:", threading.enumerate())


if __name__ == '__main__':
    main()

```

Ralentir les programmes en utilisant des threads

- En travaillant avec des threads, il est important de savoir que démarrer des centaines de threads et les lancer tous sur un problème spécifique n'améliorera probablement pas les performances de votre application
- Il est très probable que si vous lancez des centaines ou des milliers de threads, vous pourriez, en fait, tuer complètement la performance

Obtenir le nombre total de threads actifs

- Parfois, lorsque vous souhaitez, par exemple, interroger l'état de votre application, vous pouvez interroger le nombre de threads actifs en cours d'exécution dans votre programme Python
- Heureusement, le module de threading natif de Python nous permet facilement de l'obtenir avec un simple appel comme celui démontré dans l'extrait de code suivant :

threads_count.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import random
import time


def worker(i):
    time.sleep(random.randint(2, 10))


def main():
    for i in range(5000):
        t = threading.Thread(target=worker, args=(i,))
        t.start()
    count = threading.active_count()
    while count > 1:
        count = threading.active_count()
        print("{} active threads".format(count))
        time.sleep(0.5)


if __name__ == '__main__':
    main()

```

```
5001 active threads
5001 active threads
5001 active threads
4834 active threads
4434 active threads
4277 active threads
3892 active threads
3742 active threads
3349 active threads
3176 active threads
2816 active threads
2610 active threads
2238 active threads
2060 active threads
1723 active threads
1519 active threads
1149 active threads
966 active threads
590 active threads
373 active threads
1 active threads

Process finished with exit code 0
```

Obtenir le thread courant

- Un moyen simple et rapide de déterminer le fil sur lequel nous travaillons, nous pouvons utiliser la fonction ***threading.current_thread()***, comme indiqué dans l'exemple suivant :

threads_current.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import threading
5
6
7  def worker():
8      print("Current thread is {}".format(threading.current_thread()))
9
10
11 def main():
12     threads = []
13     for i in range(10):
14         thread = threading.Thread(target=worker)
15         thread.start()
16         threads.append(thread)
17     for thread in threads:
18         thread.join()
19
20
21 if __name__ == '__main__':
22     main()
23
```

```
Current thread is <Thread(Thread-1, started 27356)>
Current thread is <Thread(Thread-2, started 25396)>
Current thread is <Thread(Thread-3, started 24708)>
Current thread is <Thread(Thread-4, started 27076)>
Current thread is <Thread(Thread-5, started 24276)>
Current thread is <Thread(Thread-6, started 24512)>
Current thread is <Thread(Thread-7, started 24944)>
Current thread is <Thread(Thread-8, started 24484)>
Current thread is <Thread(Thread-9, started 24784)>
Current thread is <Thread(Thread-10, started 23784)>

Process finished with exit code 0
```

Thread principal

- Tous les programmes Python comportent au moins un thread
- Ce thread unique est le thread principal
- En Python, nous sommes en mesure d'appeler la fonction ***main_thread()***, bien nommée, où que nous soyons pour récupérer l'objet principal

threads_main.py

```
2   # -*- coding: utf-8 -*-
3
4   import threading
5   import time
6
7
8   def worker():
9       print("Child thread starting")
10      time.sleep(1)
11      print("Current thread: {}".format(threading.current_thread()))
12      print("Main thread: {}".format(threading.main_thread()))
13
14
15  def main():
16      threads = []
17      for i in range(5):
18          thread = threading.Thread(target=worker)
19          thread.start()
20          threads.append(thread)
21      for thread in threads:
22          thread.join()
23
24
25  if __name__ == '__main__':
26      main()
27
```

```
Child thread starting
Child thread starting
Child thread starting
Child thread starting
Child thread starting
Current thread: <Thread(Thread-2, started 24916)>
Current thread: <Thread(Thread-3, started 26748)>
Main thread: <_MainThread(MainThread, started 28864)>
Current thread: <Thread(Thread-5, started 23928)>
Main thread: <_MainThread(MainThread, started 28864)>
Current thread: <Thread(Thread-4, started 25900)>
Main thread: <_MainThread(MainThread, started 28864)>
Current thread: <Thread(Thread-1, started 25924)>
Main thread: <_MainThread(MainThread, started 28864)>
Main thread: <_MainThread(MainThread, started 28864)>

Process finished with exit code 0
```

Identifier les threads

- Dans certains scénarios, il peut être très utile de pouvoir distinguer les différents threads
- Dans certains cas, votre application peut être composée de centaines de threads différents, et les identifier peut vous aider à résoudre vos problèmes de débogage et à identifier les problèmes rencontrés avec votre programme sous-jacent
- Dans les systèmes massifs, il est judicieux de séparer les threads en groupes s'ils exécutent des tâches différentes
- Supposons, par exemple, que vous ayez une application qui soit à la fois à l'écoute des changements de cours des actions entrantes et qui tente également de prédire où ira ce cours
- Vous pouvez, par exemple, avoir deux groupes de threads différents ici : un groupe à l'écoute des changements et l'autre effectuant les calculs nécessaires
- Avoir des conventions de nommage différentes pour les threads qui font l'écoute et les threads qui font les calculs pourrait rendre votre travail de débogage beaucoup plus facile

Identifier les threads

- Exemple :

```
Thead Worker_A-0 started
Thead Worker_A-1 started
Thead Worker_B-0 started
Thead Worker_B-1 started
Thead Worker_B-0 finished
Thead Worker_A-0 finished
Thead Worker_B-1 finished
Thead Worker_A-1 finished

Process finished with exit code 0
```

threads_naming.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random


def worker_a():
    print("Thead {} started".format(threading.current_thread().getName()))
    time.sleep(random.randint(1, 2))
    print("Thead {} finished".format(threading.current_thread().getName()))


def worker_b():
    print("Thead {} started".format(threading.current_thread().getName()))
    time.sleep(random.randint(1, 2))
    print("Thead {} finished".format(threading.current_thread().getName()))


def main():
    threads = []
    for i in range(2):
        thread = threading.Thread(target=worker_a, name='Worker_A-' + str(i))
        thread.start()
        threads.append(thread)
    for i in range(2):
        thread = threading.Thread(target=worker_b, name='Worker_B-' + str(i))
        thread.start()
        threads.append(thread)
    for thread in threads:
        thread.join()


if __name__ == '__main__':
    main()

```

Terminer un thread

- Tuer les threads est considéré comme une mauvaise pratique, et que je déconseille activement
- Python ne fournit pas réellement une fonction de thread natif avec laquelle tuer d'autres threads, alors que cela devrait déclencher des drapeaux
- Ces threads que vous souhaitez terminer peuvent contenir une ressource critique qui doit être ouverte et fermée correctement, ou ils peuvent également être les parents de plusieurs threads enfants
- En supprimant les threads parents sans tuer leurs threads enfants, nous créons essentiellement des threads orphelins

Meilleures pratiques pour arrêter les threads

- Si vous avez besoin d'un mécanisme d'arrêt de thread, il vous incombe de mettre en place ce mécanisme permettant un arrêt en douceur plutôt qu'un "assassinat"
- Cependant, il existe une solution de contournement
- Alors que les threads ne possèdent peut-être pas de mécanisme natif de terminaison, les processus disposent en fait d'un tel mécanisme
- Comme vous devriez le savoir maintenant, les processus sont essentiellement des versions plus robustes des threads, et bien que cela ne soit pas idéal, dans certaines situations vous devez vous assurer que vos programmes peuvent s'arrêter normalement
- Cela se présente comme une solution bien plus élégante que votre propre système terminaison de thread

Processus orphelins

- Les processus orphelins sont des processus qui n'ont pas de processus parent actif
- Ils utilisent les ressources du système et n'offrent aucun avantage, et la seule façon de les tuer est d'énumérer les processus vivants, puis de les tuer

Comment le système d'exploitation gère les threads

- Maintenant que nous avons jeté un coup d'œil sur le cycle de vie d'un thread, il est important de savoir comment ces threads fonctionnent réellement dans nos machines
- Comprendre des choses comme le modèle multithreading et comment les threads Python sont mappés aux threads du système est important pour prendre les bonnes décisions lors de la conception de votre logiciel haute performance

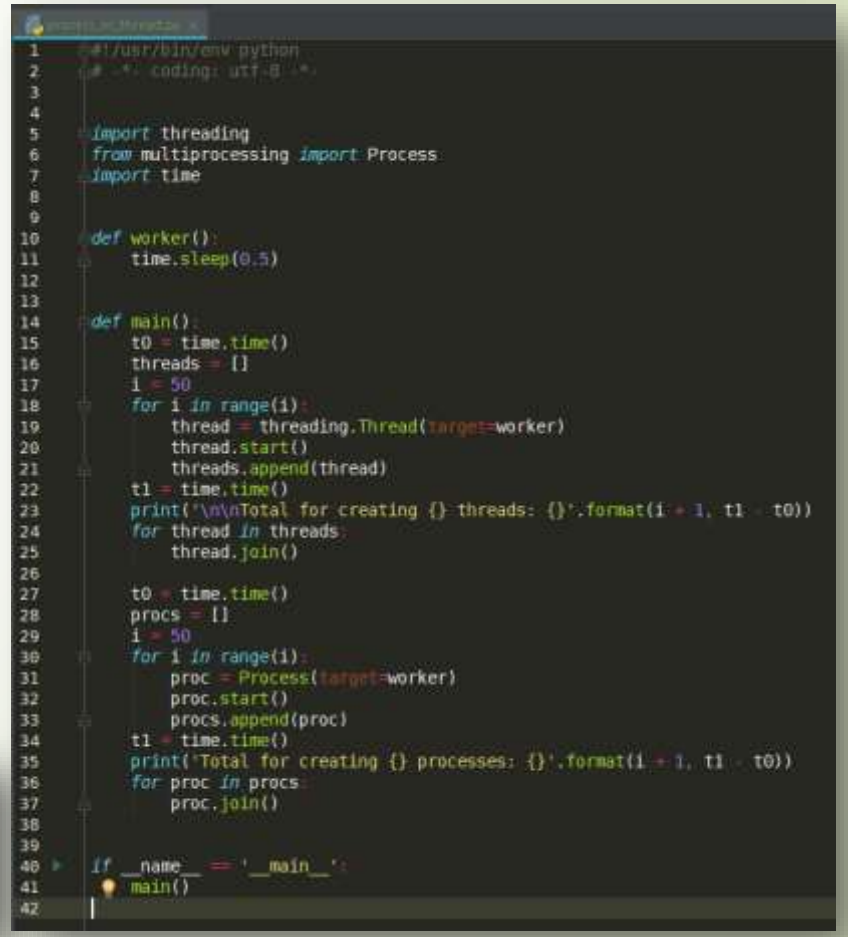
Création de processus par rapport aux threads

- Un processus, comme nous l'avons vu, est une version plus lourde d'un thread simple dans le sens où nous pouvons faire des choses comme faire tourner plusieurs threads dans un processus
- Ils peuvent effectuer plus de tâches liées au processeur mieux qu'un thread standard en raison du fait qu'ils ont chacun leur propre instance GIL séparée
- Cependant, il est important de noter que même si ceux-ci peuvent être bien meilleurs pour les CPU, ils nécessitent également beaucoup plus de ressources
- Être plus exigeant en ressources signifie qu'ils sont aussi plus chers à créer à la volée et à tuer aussi rapidement
- Dans cet exemple suivant, nous examinerons l'impact sur les performances de la rotation de plusieurs threads, et nous comparerons cela à la rotation de plusieurs processus

Création de processus par rapport aux threads

- Maintenant, bien que le temps nécessaire pour effectuer ces deux tâches soit minime pour notre exemple relativement léger, considérez l'impact sur les performances que vous obtiendriez si vous commenciez des centaines ou des milliers de processus ou de threads sur d'énormes baies de serveurs
- Une façon de lutter contre cela est de faire toute la création de processus ou de threads au début et de les stocker dans un pool afin qu'ils puissent s'endormir et attendre d'autres instructions sans que nous ayons à supporter ces coûts élevés de création

```
Total for creating 50 threads: 0.015633106231689453
Total for creating 50 processes: 0.39647436141967773

Process finished with exit code 0
```



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import threading
from multiprocessing import Process
import time


def worker():
    time.sleep(0.5)


def main():
    t0 = time.time()
    threads = []
    i = 50
    for i in range(i):
        thread = threading.Thread(target=worker)
        thread.start()
        threads.append(thread)
    t1 = time.time()
    print('\n\nTotal for creating {} threads: {}'.format(i + 1, t1 - t0))
    for thread in threads:
        thread.join()

    t0 = time.time()
    procs = []
    i = 50
    for i in range(i):
        proc = Process(target=worker)
        proc.start()
        procs.append(proc)
    t1 = time.time()
    print('Total for creating {} processes: {}'.format(i + 1, t1 - t0))
    for proc in procs:
        proc.join()


if __name__ == '__main__':
    main()

```

Modèles multithread

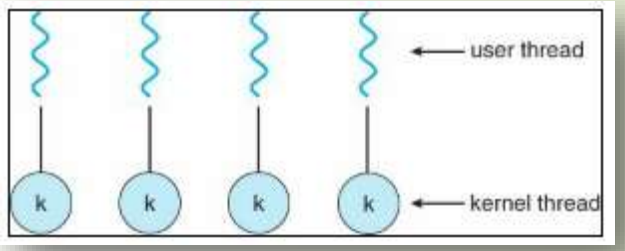
- Le début de ce document fournit une brève introduction à la concurrence, où nous avons parlé des deux types distincts de threads que nous avons sur une seule machine
- Il s'agissait de threads utilisateur et de threads du noyau, et il est utile de savoir comment ils se mêlent, ainsi que les différentes façons de les mêler
- Au total, il existe trois différents styles de mappage :
  - Un thread utilisateur vers un thread noyau
  - Plusieurs threads utilisateur vers un thread noyau
  - Plusieurs threads utilisateur vers plusieurs threads noyau
- Dans Python, nous avons généralement un mappage d'un thread utilisateur sur un thread noyau
- Ainsi, chaque thread que vous créez dans vos applications multithread occupera une quantité de ressources significative sur votre machine.

Modèles multithread

- Cependant, il existe certains modules dans l'écosystème Python qui vous permettent d'implémenter une fonctionnalité multithread à votre programme tout en restant sur un seul thread
- L'un des exemples les plus significatif est le module ***asyncio***

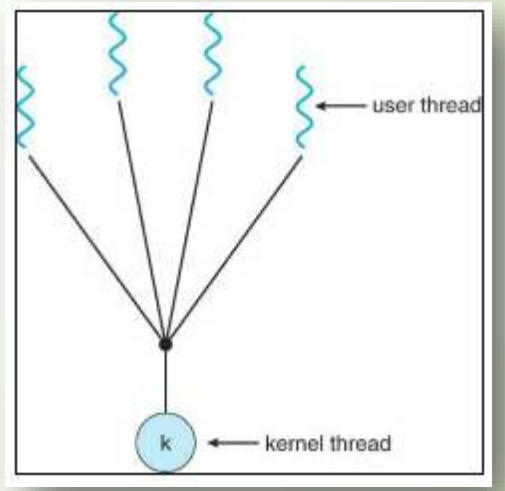
Mappage de threads un-à-un

- Dans ce mappage, nous voyons qu'un thread de niveau utilisateur est mappé directement à un thread au niveau du noyau
- Les mappages un-à-un peuvent être coûteux en raison des coûts inhérents à la création et à la gestion des threads au niveau du noyau, mais ils offrent des avantages dans la mesure où les threads au niveau utilisateur ne sont pas soumis au même niveau de blocage que le mappage de un à plusieurs



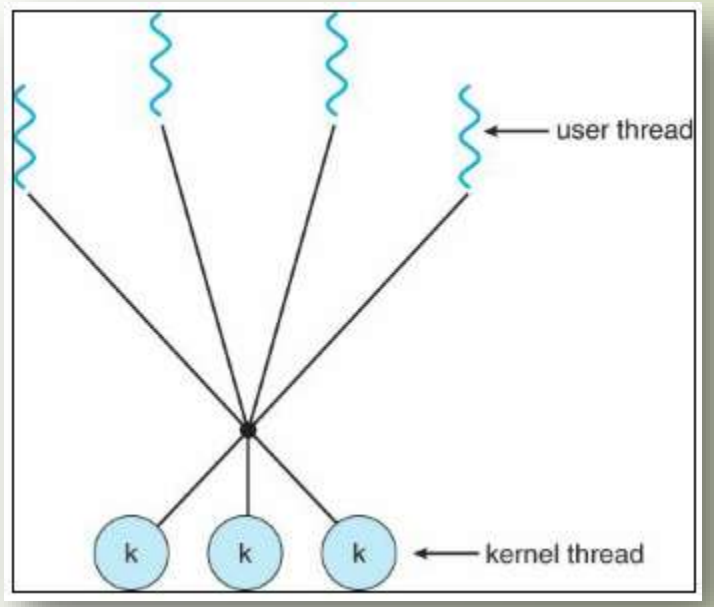
Mappage de threads un-à-plusieurs

- Dans les mappages plusieurs à un, de nombreux threads de niveau utilisateur sont mappés à un thread au niveau du noyau solitaire
- Ceci est avantageux car nous pouvons gérer efficacement les threads de niveau utilisateur
- Cependant, si le thread au niveau utilisateur est bloqué, les autres threads mappés au thread au niveau du noyau seront également bloqués



Mappage de threads plusieurs-à-plusieurs

- Dans ce modèle de thread, de nombreux threads de niveau utilisateur sont mappés à de nombreux threads de niveau noyau
- Cela se présente comme la solution aux insuffisances des deux modèles précédents
- Les threads individuels au niveau de l'utilisateur peuvent être mappés à une combinaison d'un seul thread au niveau du noyau ou de plusieurs threads du noyau
- Il nous offre, en tant que programmeurs, la possibilité de choisir les threads au niveau de l'utilisateur que nous souhaitons mapper aux threads au niveau du noyau, et, globalement, nous donne beaucoup de pouvoir pour garantir les performances les plus élevées lorsque nous travaillons un environnement multithread



Synchronisation des threads

Python Concurrence

Synchronisation entre les threads

- Maintenant que nous avons examiné les threads, et comment nous pouvons travailler avec et créer ces threads en utilisant divers mécanismes, il est temps d'examiner certaines des primitives de synchronisation de base que nous pouvons utiliser dans nos applications multithreads
- Il ne suffit pas simplement d'ajouter plusieurs threads à votre application afin d'améliorer les performances
- Vous devez également prendre en considération les complexités telles que les situations de compétition, et vous assurer que votre code est correctement protégé contre elles

Synchronisation entre les threads

- Vous savez donc ce que sont les threads et comment les démarrer et les terminer correctement en Python, et avec un peu de chance, vous commencez à réaliser au moins une partie de la complexité qu'il faut pour implémenter des programmes concurrents
- Mais comment nous assurer de mettre en œuvre le multithreading de manière sûre sans compromettre le déroulement de notre programme ?
- Nous allons voir certains des problèmes fondamentaux qui peuvent affecter les applications multithread si elles ne sont pas protégées

Synchronisation entre les threads

- Avant de couvrir certaines des primitives de synchronisation clés, nous devons d'abord examiner certains des problèmes pouvant survenir lors de l'utilisation des primitives
- Cela nous amène directement à l'un des problèmes les plus importants et les plus redoutés auxquels on peut être confronté lors de la conception de systèmes concurrents, c'est-à-dire un ***deadlock***
- L'une des meilleures façons d'illustrer ce concept de deadlock est de regarder le problème du diner des philosophes…

Le diner des philosophes

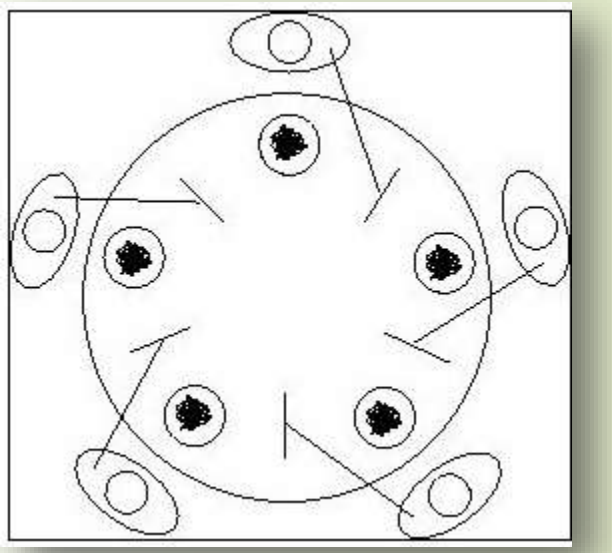
- Le problème du diner des philosophes est l'une des illustrations les plus célèbres de certains des problèmes que vous pouvez rencontrer lorsque vous travaillez dans des systèmes logiciels concurrents
- C'était, à l'origine le fameux Edsger Dijkstra qui a présenté ce problème au monde
- Ce fut Tony Hoare, cependant, qui a donné au problème sa formulation plus officielle

Le diner des philosophes

- Dans ce problème, nous rencontrons cinq philosophes célèbres assis à une table ronde mangeant des bols de spaghetti
- Entre chacun de ces bols, il y a cinq fourchettes que les philosophes peuvent utiliser pour manger leur nourriture
- Pour une raison étrange cependant, ces philosophes décident qu'ils ont chacun besoin de deux des cinq fourchettes pour manger leur nourriture
- Chacun de ces philosophes, cependant, pourrait être soit en état de manger ou de penser, et chaque fois qu'ils choisissent de plonger dans la nourriture en face d'eux, ils doivent d'abord obtenir à la fois la fourche gauche et la fourche droite
- Cependant, quand un philosophe prend une fourchette, il doit attendre d'avoir mangé avant de pouvoir abandonner cette fourchette

Le diner des philosophes

- Dans le diagramme, nous voyons juste une telle situation se produire
- Chacun des cinq philosophes a pris la fourchette gauche et est maintenant assis en train de penser jusqu'à ce que la fourchette droite soit disponible
- Puisque chaque philosophe n'abandonnera jamais sa fourchette avant d'avoir mangé, la table du dîner est dans une impasse et n'ira jamais plus loin



Le diner des philosophes

- Ce problème illustre un problème clé que nous pouvons rencontrer lorsque nous concevons nos propres systèmes concurrents et qui repos sur des primitives de synchronisation clés (verrous) afin de fonctionner correctement
- Nos fourchettes, dans cet exemple, sont nos ressources système, et chaque philosophe représente un processus concurrent

Le diner des philosophes

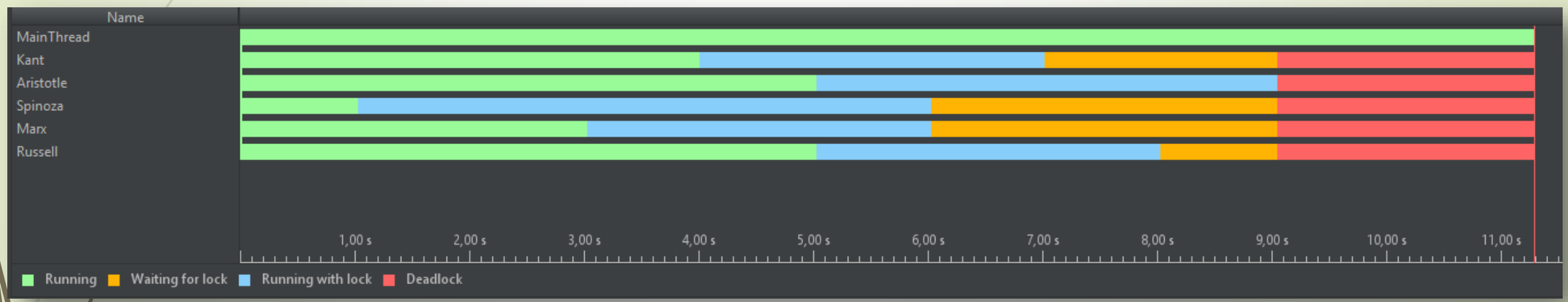
- Le programme

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import random
import time
import threading


class Philosopher(threading.Thread):


def main():

    fork1 = threading.RLock()
    fork2 = threading.RLock()
    fork3 = threading.RLock()
    fork4 = threading.RLock()
    fork5 = threading.RLock()

    philosopher1 = Philosopher("Kant", fork1, fork2)
    philosopher2 = Philosopher("Aristotle", fork2, fork3)
    philosopher3 = Philosopher("Spinoza", fork3, fork4)
    philosopher4 = Philosopher("Marx", fork4, fork5)
    philosopher5 = Philosopher("Russell", fork5, fork1)

    philosopher1.start()
    philosopher2.start()
    philosopher3.start()
    philosopher4.start()
    philosopher5.start()

    philosopher1.join()
    philosopher2.join()
    philosopher3.join()
    philosopher4.join()
    philosopher5.join()


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

```python
class Philosopher(threading.Thread):

    def __init__(self, name, leftFork, rightFork):
        print("{} Has Sat Down At the Table".format(name))
        threading.Thread.__init__(self, name=name)
        self.leftFork = leftFork
        self.rightFork = rightFork

    def run(self):
        print("{} has started thinking".format(threading.currentThread().getName()))
        while True:
            time.sleep(random.randint(1, 5))
            print("{} has finished thinking".format(threading.currentThread().getName()))
            self.leftFork.acquire()
            time.sleep(random.randint(1, 5))
            try:
                print("{} has acquired the left fork".format(threading.currentThread().getName()))

                self.rightFork.acquire()
                try:
                    print("{} has attained both forks, currently eating".format(threading.currentThread().getName()))
                finally:
                    self.rightFork.release()
                    print("{} has released the right fork".format(threading.currentThread().getName()))
            finally:
                self.leftFork.release()
                print("{} has released the left fork".format(threading.currentThread().getName()))

```

Le diner des philosophes

- Diagramme de concurrence de notre programme…



Conditions de compétition

- Maintenant que nous avons regardé les deadlocks, il est temps de parler des conditions de compétition
- Les conditions de compétition sont un aspect tout aussi gênant et souvent maudit de la programmation concurrente et qui affecte des centaines, voire des milliers de programmes à travers le monde
- La définition standard d'une condition de concurrence est la suivante :
- "Une condition de compétition est le comportement d'un système électronique, logiciel ou autre dont la sortie dépend de la séquence ou du calendrier d'autres événements incontrôlables"

Conditions de compétition

- Séparons cette définition en termes plus simples
- L'une des meilleures métaphores pour décrire une condition de concurrence est si nous imaginons l'écriture d'une application bancaire qui met à jour le solde de votre compte chaque fois que vous déposez ou retirez de l'argent de ce compte
- Imaginez, nous avons commencé avec 2.000€ sur notre compte bancaire, et disons que nous sommes sur le point de recevoir un bonus de 5.000€, parce que nous avons réussi à corriger un problème de simultanéité dans un travail qui coûtait des millions
- Maintenant, imaginez aussi que vous devez également payer un loyer de 1.000€ le même jour
- C'est là qu'une condition de compétition potentielle pourrait vous laisser de votre poche

Conditions de compétition

- Si notre application bancaire comportait deux processus, dont l'un concernerait le retrait, le processus A, et l'autre le processus de dépôt, le processus B
- Imaginons que le processus B, qui traite des dépôts sur votre compte, lise votre solde bancaire à 2.000€
- Si le processus A devait commencer son retrait pour le loyer juste après le début du processus B, le solde de départ serait de 2.000€
- Le processus B achèverait alors sa transaction et ajouterait correctement 5.000€ à nos 2.000€, ce qui nous laisserait la somme totale de 7.000€

Conditions de compétition

- Cependant, puisque le processus A a commencé sa transaction en pensant que le solde du compte de départ était de 2.000€, il nous laisserait involontairement moins de bonus lorsqu'il mettrait à jour notre solde bancaire final à 1.000€
- C'est un exemple typique d'une condition de concurrence dans notre logiciel, et c'est un danger très réel qui attend toujours de nous frapper de la manière la plus malheureuse

Séquence d'exécution du processus

- Jetons un coup d'oeil à ce qui s'est passé de manière plus détaillée
- Si nous regardons le tableau suivant, nous verrons le flux d'exécution idéal pour les processus A et B
- Cependant, étant donné que nous n'avons pas mis en œuvre les mécanismes de synchronisation appropriés pour protéger le solde de notre compte, le processus A et le processus B ont suivi le chemin d'exécution suivant et nous ont donné un résultat erroné

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

La solution

- Comment pouvons-nous résoudre le problème précédent afin que nous ne vivions plus dans la crainte de perdre nos primes à l'avenir ?
- Dans cet exemple relativement simple, la réponse serait d'encapsuler le code qui lit d'abord le solde du compte, et d'exécuter toutes les transactions nécessaires dans un verrou
- En enveloppant le code qui effectue la lecture du solde du compte et la mise à jour dans un verrou, nous nous assurons que le processus A doit d'abord acquérir le verrou pour lire et mettre à jour le solde de notre compte, et de même pour le processus B
- Ceci rendrait notre programme déterministe et libre de notre condition de race initiale
- Mais en le transformant en un programme déterministe, nous convertissons essentiellement cette section de code en une section sérielle unique de code qui pourrait avoir un impact sur les performances si nous avions plusieurs threads

Section critique de code

- Nous pouvons identifier les sections critiques comme toute partie de notre code qui modifie ou accède à une ressource partagée
- Ces sections critiques ne peuvent, en aucun cas, être exécutées par plus d'un processus à la fois
- C'est lorsque ces sections critiques sont exécutées simultanément que nous commençons à voir un comportement inattendu ou erroné
- Supposons, par exemple, que nous écrivions le code pour l'exemple d'application bancaire précédemment défini
- Nous pourrions catégoriser la partie du code qui fait la lecture initiale du compte bancaire jusqu'au moment où il met à jour la ligne de fond des comptes comme section critique
- C'est par l'exécution simultanée de cette section critique que nous avons d'abord rencontré une condition de compétition
- En comprenant où dans notre code, nous avons des sections critiques, nous, en tant que programmeurs, sommes capables de protéger plus précisément ces sections en utilisant certaines des primitives que nous allons détailler

Système de fichiers

- Il est important de noter que les conditions de course peuvent affecter notre système de fichiers ainsi que nos programmes
- Un problème potentiel pourrait être que deux processus tentent simultanément de modifier un fichier sur le système de fichiers
- Sans les contrôles de synchronisation appropriés autour de ces fichiers, il est possible que le fichier devienne potentiellement corrompu et inutilisable avec deux processus qui y écrivent

Systèmes vitaux

- L'un des pires exemples de la façon dont les conditions de compétition peuvent infecter notre logiciel est dans le logiciel qui contrôlait les appareils de radiothérapie Therac-25
- Cette condition de course était, malheureusement, suffisante pour causer la mort d'au moins trois patients qui recevaient un traitement de la machine
- La plupart du temps, le logiciel que nous écrivons ne sera pas aussi critique que le logiciel utilisé dans les dispositifs médicaux comme celui-ci
- Cependant, il s'agit d'un avertissement très morbide pour s'assurer que vous essayez de prendre toutes les mesures afin d'éviter que votre propre logiciel soit affecté

Ressources partagées et conditions de compétition

- L'une des principales choses à éviter lors de l'implémentation simultanée de vos applications est la condition de compétition
- Ces conditions de concurrence peuvent paralyser nos applications et causer des bogues difficiles à déboguer et encore plus difficiles à corriger
- Afin d'éviter ces problèmes, nous devons à la fois comprendre comment ces conditions de compétition se produisent et comment nous pouvons nous en prémunir en utilisant les primitives de synchronisation que nous allons maintenant aborder
- Comprendre la synchronisation et les primitives de base qui vous sont disponibles est essentiel si vous voulez créer des programmes performants et sécurisés pour les threads en Python
- Heureusement, nous avons de nombreuses primitives de synchronisation différentes disponibles dans le module Python de threading qui peuvent nous aider dans un certain nombre de situations concurrentes différentes

La méthode *join()*

- Quand il s'agit de développer des systèmes d'entreprise incroyablement importants, être capable de dicter l'ordre d'exécution de certaines de nos tâches est extrêmement important
- Heureusement, l'objet thread de Python nous permet de conserver une certaine forme de contrôle, car ils sont livrés avec une méthode de jointure
- La méthode ***join()***, essentiellement, empêche le thread parent de continuer à progresser jusqu'à ce que ce thread enfant ai confirmé qu'il s'est terminé
- Cela peut se faire soit par une fin naturelle, soit lorsque le thread lance une exception non gérée

Locks

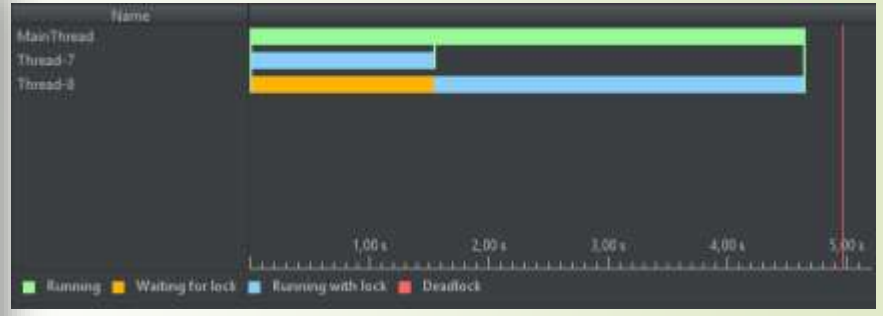
- Les verrous sont un mécanisme essentiel lorsque vous essayez d'accéder à des ressources partagées à partir de plusieurs threads d'exécution
- La meilleure façon d'imaginer cela est d'imaginer que vous avez une salle de bains et plusieurs colocataires
- Lorsque vous voulez vous rafraîchir ou prendre une douche, vous voudriez verrouiller la porte pour que personne d'autre ne puisse utiliser la salle de bain en même temps
- Un verrou en Python est une primitive de synchronisation qui nous permet de verrouiller notre porte de salle de bain
- Il peut être dans un état "verrouillé" ou "déverrouillé", et nous ne pouvons acquérir un verrou que lorsqu'il est dans un état "déverrouillé"

Locks

- Code et diagramme de concurrence

concurrency8.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random

counter = 1
lock = threading.Lock()


def worker_a():...


def worker_b():...


def main():
    t0 = time.time()
    thread1 = threading.Thread(target=worker_a)
    thread2 = threading.Thread(target=worker_b)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    t1 = time.time()
    print("Execution time {}".format(t1 - t0))


if __name__ == '__main__':
    main()

```




```python
def worker_a():
    global counter
    lock.acquire()
    try:
        while counter < 10:
            counter += 1
            print("worker A is incrementing counter to {}".format(counter))
            sleep_time = random.randint(0, 1)
            time.sleep(sleep_time * .5)
    finally:
        lock.release()


def worker_b():
    global counter
    lock.acquire()
    try:
        while counter > -10:
            counter -= 1
            print("worker B is decrementing counter to {}".format(counter))
            sleep_time = random.randint(0, 1)
            time.sleep(sleep_time * .5)
    finally:
        lock.release()
```

RLocks

- Les verrous réentrants (***RLocks***) sont des primitives de synchronisation qui fonctionnent un peu comme notre primitive de verrou standard, mais peuvent être acquis par un thread plusieurs fois si ce thread le possède déjà
- Par exemple, par exemple, thread-1 acquiert le RLock, donc, à chaque fois que thread-1 acquiert le verrou, un compteur dans la primitive RLock est incrémenté de 1
- Si thread-2 a essaye d'acquérir le RLock, alors il devrait attendre jusqu'à ce que le compteur du RLock tombe à 0 avant qu'il puisse être acquis
- Thread-2 entrerait dans un état de blocage jusqu'à ce que cette condition 0 soit remplie
- Pourquoi est-ce utile, cependant ?
- Eh bien, cela peut être utile lorsque vous voulez, par exemple, avoir un accès sécurisé pour une méthode au sein d'une classe qui accède à d'autres méthodes de classe

RLocks

- Exemple de code

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import threading
import time


class myWorker():
    def __init__(self):
        self.a = 1
        self.b = 2
        self.Rlock = threading.RLock()

    def modifyA(self):
        with self.Rlock:
            print("Modifying A : RLock Acquired:    {}".format(self.Rlock._is_owned()))
            print("{}".format(self.Rlock))
            self.a = self.a + 1
            time.sleep(5)

    def modifyB(self):
        with self.Rlock:
            print("Modifying B : RLock Acquired:    {}".format(self.Rlock._is_owned()))
            print("{}".format(self.Rlock))
            self.b = self.b - 1
            time.sleep(5)

    def modifyBoth(self):
        with self.Rlock:
            print("Rlock acquired, modifying A and B")
            print("{}".format(self.Rlock))
            self.modifyA()
            self.modifyB()
            print("{}".format(self.Rlock))


def main():
    workerA = myWorker()
    workerA.modifyBoth()


if __name__ == '__main__':
    #
    # Call the main() function
    #
    main()
```

Rlocks vs Locks

- Si nous devions essayer d'exécuter le même programme précédent en utilisant une primitive de verrou classique, alors vous devriez remarquer que le programme n'atteint jamais le point où il exécute notre fonction ***modifyA()***
- Notre programme se retrouverait essentiellement dans une impasse, car nous n'avons pas mis en place un mécanisme de libération qui permette à notre fil d'aller plus loin
- Ceci est illustré dans l'exemple de code suivant

concurrency10.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import threading
import time


class myWorker():
    def __init__(self):
        self.a = 1
        self.b = 2
        self.lock = threading.Lock()

    def modifyA(self):
        with self.lock:
            print("Modifying A : RLock Acquired: {}".format(self.lock._is_owned()))
            print("{}".format(self.lock))
            self.a = self.a + 1
            time.sleep(5)

    def modifyB(self):
        with self.lock:
            print("Modifying B : Lock Acquired: {}".format(self.lock._is_owned()))
            print("{}".format(self.lock))
            self.b = self.b - 1
            time.sleep(5)

    def modifyBoth(self):
        with self.lock:
            print("lock acquired, modifying A and B")
            print("{}".format(self.lock))
            self.modifyA()
            print("{}".format(self.lock))
            self.modifyB()
            print("{}".format(self.lock))


def main():
    workerA = myWorker()
    workerA.modifyBoth()


if __name__ == '__main__':
    main()

```

Rlocks vs Locks

- Les RLocks, essentiellement, nous permettent d'obtenir une certaine forme de sécurité de thread d'une manière récursive sans avoir à implémenter l'acquisition et la libération de verrou dans tout votre code
- Ils nous permettent d'écrire du code plus simple, plus facile à suivre et, par conséquent, plus facile à maintenir après la production de notre code

Conditions

- Une condition est une primitive de synchronisation qui attend un signal provenant d'un autre thread
- Par exemple, cela pourrait être qu'un autre thread a terminé son exécution, et que le thread actuel peut son propre code

Conditions

- Jetons un coup d'oeil à la définition de notre objet condition dans la bibliothèque native de Python

```
def Condition(*args, **kwargs):
  """Factory function that returns a new condition variable object.
  A condition variable allows one or more threads to wait until they are
  notified by another thread.
  If the lock argument is given and not None, it must be a Lock or RLock
  object, and it is used as the underlying lock. Otherwise, a new RLock objec
  is created and used as the underlying lock.
```

- Il est important de comprendre ces primitives fondamentales et comment elles fonctionnent à un niveau plus granulaire
- Le scénario le plus commun utilisé pour mettre en évidence les avantages des conditions est celui d'un producteur/consommateur
- Vous pouvez avoir un producteur qui publie des messages dans une file d'attente et avertit les autres threads, c'est-à-dire les consommateurs, qu'il y a maintenant des messages en attente d'être consommés dans cette file d'attente

Conditions

- Dans cet exemple, nous allons créer deux classes différentes qui hériteront de la classe thread
- Celles-ci seront notre éditeur et notre abonné
- L'éditeur effectuera la tâche de publication de nouveaux entiers dans un tableau d'entiers, puis notifiera aux abonnés qu'il existe un nouvel entier à utiliser dans le tableau

Conditions

- Notre classe Publisher a deux fonctions définies en son sein
- Le constructeur qui prend en référence le tableau des entiers et la primitive de condition
- La fonction ***run()*** entre dans une boucle permanente lorsqu'elle est invoquée, puis génère un entier aléatoire entre 0 et 1000
- Une fois que nous avons généré ce nombre, nous acquérons la condition, puis nous ajoutons cet entier nouvellement généré à notre tableau d'entiers
- Une fois que nous avons ajouté à notre tableau, nous informons d'abord nos abonnés qu'un nouvel élément a été ajouté à ce tableau, puis nous libérons la condition

Conditions

- Le code :

concurrency11.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random


class Publisher(threading.Thread):...


class Subscriber(threading.Thread):...


def main():
    integers = []
    condition = threading.Condition()
    pub1 = Publisher(integers, condition)
    pub1.start()
    sub1 = Subscriber(integers, condition)
    sub2 = Subscriber(integers, condition)
    sub1.start()
    sub2.start()
    sub1.join()
    sub2.join()
    pub1.join()


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

```python
class Publisher(threading.Thread):
    def __init__(self, integers, condition):
        self.condition = condition
        self.integers = integers
        threading.Thread.__init__(self)

    def run(self):
        while True:
            integer = random.randint(0, 1000)
            self.condition.acquire()
            print("Condition Acquired by Publisher: {}".format(self.name))
            self.integers.append(integer)
            print("Publisher {} appending to array: {}".format(self.name, integer))
            self.condition.notify()
            self.condition.release()
            print("Condition Released by Publisher: {}".format(self.name))
            time.sleep(1)
```

```python
class Subscriber(threading.Thread):
    def __init__(self, integers, condition):
        self.integers = integers
        self.condition = condition
        threading.Thread.__init__(self)

    def run(self):
        while True:
            self.condition.acquire()
            print("Condition Acquired by Consumer: {}".format(self.name))
            while True:
                if self.integers:
                    integer = self.integers.pop()
                    print("{} Popped from list by Consumer: {}".format(integer, self.name))
                    break
                print("Condition Wait by {}".format(self.name))
                self.condition.wait()
            print("Consumer {} releasing condition".format(self.name))
            self.condition.release()
```

Conditions

- Lorsque nous exécutons ce programme, vous devriez voir une sortie similaire à la suivante
- Vous devriez voir que lorsque l'éditeur acquiert la condition, il ajoute un nombre au tableau, puis notifie la condition et la libère
- Au moment où la condition est notifiée, la bataille commence entre les deux abonnés et ils essaient tous deux d'acquérir cette condition en premier
- Quand l'un emporte ce combat, il continue simplement à "pop" ce nombre du tableau

```
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 776
Condition Released by Publisher: Thread-1
Condition Acquired by Consumer: Thread-2
776 Popped from list by Consumer: Thread-2
Consumer Thread-2 releasing condition
Condition Acquired by Consumer: Thread-2
Condition Wait by Thread-2
Condition Acquired by Consumer: Thread-3
Condition Wait by Thread-3
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 348
Condition Released by Publisher: Thread-1
348 Popped from list by Consumer: Thread-2
Consumer Thread-2 releasing condition
Condition Acquired by Consumer: Thread-2
Condition Wait by Thread-2
```

Sémaphores

- Dans la première partie, nous avons abordé l'histoire de la simultanéité, et nous avons parlé un peu de Dijkstra
- Dijkstra était l'homme qui a effectivement pris cette idée des sémaphores des systèmes ferroviaires et les a traduits en quelque chose que nous pourrions utiliser dans nos propres systèmes concurrents complexes
- Les sémaphores sont des verrous possédant un compteur interne qui est incrémenté et décrémenté chaque fois qu'un appel d'acquisition ou de libération est effectué
- Lors de l'initialisation, ce compteur est réglé par défaut sur 1, sauf spécification contraire
- Le sémaphore ne peut pas être acquis si le compteur tombe à une valeur entière négative

Sémaphores

- Supposons que nous ayons protégé un bloc de code avec un sémaphore et que la valeur du sémaphore soit 2
- Si un thread acquiert le sémaphore, la valeur du sémaphore sera décrémentée à 1
- Si un autre thread essaye alors d'acquérir le sémaphore, la valeur du sémaphore décrémenter à 0
- À ce stade, si un autre thread venait à arriver, le sémaphore refuserait sa demande d'acquisition jusqu'à ce que l'un des deux threads d'origine appelé la méthode de libération, et le compteur incrémenté à 0 précédent

Sémaphores *Exemple*

- L'exemple suivant est basé librement sur un exemple de concurrence du service informatique de Stanford
- Dans cet exemple, nous allons créer un programme de vente de tickets simple qui comporte quatre threads distincts qui essayent chacun de vendre autant de tickets que possible avant que les tickets ne soient vendus

```python
# -*- coding: utf-8 -*-

import threading
import time
import random

tickets_disponibles = 10


class VendeurTickets(threading.Thread):
    tickets_vendus = 0

    def __init__(self, semaphore):
        super().__init__()
        self.sem = semaphore
        print('Le vendeur {} de ticket à commencé son travail'.format(self.getName()))

    def run(self):
        global tickets_disponibles
        running = True
        while running:
            self.random_delay()
            self.sem.acquire()
            if tickets_disponibles <= 0:
                running = False
            else:
                self.tickets_vendus += 1
                tickets_disponibles -= 1
                print('Le vendeur {} a vendu un ticket ({} restants)'.format(self.getName(), tickets_disponibles))
            self.sem.release()
        print('Le vendeur {} a vendu {} ticket(s) '.format(self.getName(), self.tickets_vendus))

    def random_delay(self):
        time.sleep(random.randint(0, 1))


def main():
    semaphore = threading.Semaphore()
    vendeurs = []
    for i in range(4):
        vendeur = VendeurTickets(semaphore)
        vendeur.start()
        vendeurs.append(vendeur)
    for vendeur in vendeurs:
        vendeur.join()


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

Sémaphores *Exemple*

```
Le vendeur Thread-1 de ticket à commencé son travail
Le vendeur Thread-1 a vendu un ticket (9 restants)
Le vendeur Thread-2 de ticket à commencé son travail
Le vendeur Thread-2 a vendu un ticket (8 restants)
Le vendeur Thread-3 de ticket à commencé son travail
Le vendeur Thread-3 a vendu un ticket (7 restants)
Le vendeur Thread-4 de ticket à commencé son travail
Le vendeur Thread-4 a vendu un ticket (6 restants)
Le vendeur Thread-1 a vendu un ticket (5 restants)
Le vendeur Thread-1 a vendu un ticket (4 restants)
Le vendeur Thread-3 a vendu un ticket (3 restants)
Le vendeur Thread-2 a vendu un ticket (2 restants)
Le vendeur Thread-4 a vendu un ticket (1 restants)
Le vendeur Thread-4 a vendu un ticket (0 restants)
Le vendeur Thread-4 a vendu 3 ticket(s)
Le vendeur Thread-1 a vendu 3 ticket(s)
Le vendeur Thread-2 a vendu 2 ticket(s)
Le vendeur Thread-3 a vendu 2 ticket(s)

Process finished with exit code 0
```

- Lorsque vous exécutez le programme précédent, vous devriez, espérons-le, voir une sortie similaire à celle-ci
- Dans cette course particulière, nous voyons une distribution presque égale de billets vendus entre les quatre fils distincts
- Lorsque l'un de ces threads bloque pour une durée indéterminée, un autre thread acquiert le sémaphore et tente de vendre son ticket
- Il est à noter que si nous supprimons le blocage simulé du thread en commentant ***self.random_delay()*** dans la fonction ***run()***, alors, quel que soit le thread ayant acquis le sémaphore, celui-ci vendra d'abord tous les des billets
- C'est parce que le thread qui gagne le sémaphore est dans une position de choix pour réacquérir le verrou avant que tout autre thread ne soit en mesure de le faire

Sémaphores bornés

- Les sémaphores bornés sont presque identiques aux sémaphores normaux
- Un sémaphore borné vérifie que sa valeur actuelle ne dépasse pas sa valeur initiale
- Si c'est le cas, ***ValueError*** est déclenché
- Dans la plupart des situations, les sémaphores sont utilisés pour protéger les ressources ayant une capacité limitée
- Si le sémaphore est publié trop de fois, c'est un signe d'un bug
- Si une valeur n'est pas donnée, la valeur par défaut est 1

Sémaphores bornés

- Ces sémaphores bornés peuvent généralement être trouvés dans les implémentations de serveur ou de base de données web pour éviter l'épuisement des ressources en cas de tentative de connexion simultanée ou de tentative de connexion à une action spécifique à la fois
- Il est généralement préférable d'utiliser un sémaphore borné par opposition à un sémaphore normal
- Si nous devions changer le code précédent pour utiliser ***threading.BoundedSemaphore(4)*** et l'exécuter de nouveau, nous verrions presque exactement le même comportement sauf que nous avons protégé notre code contre des erreurs programmatiques très simples qui autrement seraient restées non capturées

Evénements

- Les événements sont une forme de communication très utile mais aussi très simple entre plusieurs threads s'exécutant simultanément
- Avec les événements, un thread signale généralement qu'un événement s'est produit alors que d'autres threads écoutent activement ce signal
- Les événements sont essentiellement des objets dotés d'un indicateur interne ***true*** ou ***false***
- Au sein de nos threads, nous pouvons continuellement interroger cet objet événement pour vérifier dans quel état il se trouve, puis choisir d'agir de la manière que nous voulons quand ce drapeau change d'état

Evénements

- Dans le chapitre précédent, nous avons parlé de la façon dont il n'y avait pas de mécanismes réels pour tuer les threads nativement en Python, et c'est toujours vrai
- Cependant, nous pourrions utiliser ces objets d'événement et faire en sorte que nos threads ne s'exécutent que tant que notre objet événement n'est pas défini
- Bien que ce ne soit pas très intéressant au moment où un signal ***SIGKILL*** est envoyé, il peut cependant être utile dans certaines situations où vous devez vous arrêter normalement, mais où vous pouvez attendre qu'un thread finisse ce qu'il fait avant de se terminer
- Un événement a quatre fonctions publiques avec lesquelles nous pouvons le modifier et l'utiliser :
  - ***isSet()*** : Vérifie si l'événement a été défini
  - ***set()*** : Définit l'évènement
  - ***clear()*** : Réinitialise notre objet d'événement
  - ***wait()*** : Bloque le thread jusqu'à ce que le drapeau interne soit défini sur true

Evénements

- Exemple de code :

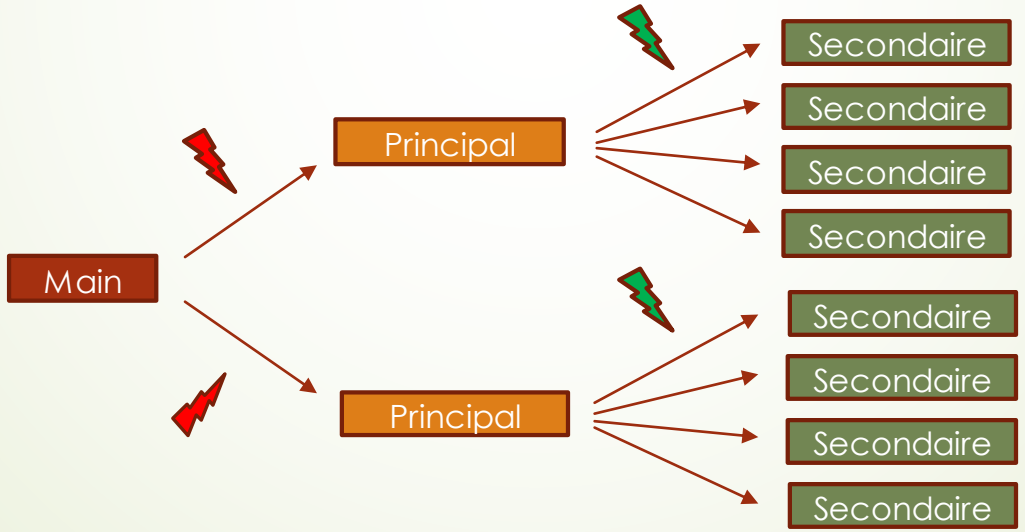
events1.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time


def my_thread(my_event):
    while not my_event.isSet():
        print('Waiting for my_event to be set...')
        time.sleep(1)
    print('Stopping: my_event has been set!')


def main():
    my_event = threading.Event()
    thread1 = threading.Thread(target=my_thread, args=(my_event,))
    thread1.start()
    time.sleep(10)
    my_event.set()


if __name__ == '__main__':
    main()

```

```
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Waiting for my_event to be set...
Stopping: my_event has been set!

Process finished with exit code 0
```

Evénements

- Un exemple plus sophistiqué :
- La fonction ***main()*** lance deux threads principaux et les stoppe par le déclenchement d'un évènement
- Chaque thread lancé par ***main()*** lance à son tour quatre threads secondaires qui seront chacun arrêtés par un évènement déclenché par les threads principaux



Evénements

- Code :

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time


def my_thread_1(my_event_1):
    my_event_2 = threading.Event()
    threads_2 = []
    print('Starting sub-threads...')
    for i in range(4):
        t = threading.Thread(target=my_thread_2, args=(my_event_2,))
        t.start()
        threads_2.append(t)
    time.sleep(2)
    print('Waiting for my_event_1 to be set...')
    while not my_event_1.isSet():
        time.sleep(2)
    print('Stopping: my_event_1 has been set!')
    my_event_2.set()
    for t in threads_2:
        t.join()


def my_thread_2(my_event_2):
    print('Waiting for my_event_2 to be set...')
    while not my_event_2.isSet():
        time.sleep(2)
    print('Stopping: my_event_2 has been set!')


def main():
    my_event_1 = threading.Event()
    threads_1 = []
    print('Starting threads...')
    for i in range(2):
        t = threading.Thread(target=my_thread_1, args=(my_event_1,))
        t.start()
        threads_1.append(t)
    time.sleep(10)
    my_event_1.set()
    for t in threads_1:
        t.join()
    print('All threads stopped.')


if __name__ == '__main__':
    main()

```

```
Starting threads...
Starting sub-threads...
Waiting for my_event_2 to be set...
Starting sub-threads...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_2 to be set...
Waiting for my_event_1 to be set...
Waiting for my_event_1 to be set...
Stopping: my_event_1 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_1 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
Stopping: my_event_2 has been set!
All threads stopped.

Process finished with exit code 0
```

Barrières

- Les barrières sont des primitives de synchronisation introduite dans la troisième version majeure du langage Python, et abordent un problème qui ne peut être résolu qu'avec un mélange quelque peu compliqué de conditions et de sémaphores
- Ces barrières sont des points de contrôle qui peuvent être utilisés pour s'assurer que la progression n'est faite que par un groupe de threads, après le point où tous les threads participants atteignent le même point
- Cela peut sembler un peu compliqué et inutile, mais il peut être incroyablement puissant dans certaines situations, et il peut certainement réduire la complexité du code

Barrières

- Dans l'exemple suivant, nous allons utiliser des barrières afin de bloquer l'exécution de nos threads jusqu'à ce que tous les threads aient atteint un point d'exécution désiré :

barriers1.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random


class MyThread(threading.Thread):
    def __init__(self, barrier):
        super().__init__()
        self.barrier = barrier

    def run(self):
        print('Thread {} working on something...'.format(self.getName()))
        time.sleep(random.randint(1, 10))
        print('Thread {} waiting for others...'.format(self.getName()))
        self.barrier.wait()
        print('Barrier has been lifted: continuing')


def main():
    my_barrier = threading.Barrier(4)
    threads = []
    for i in range(4):
        t = MyThread(my_barrier)
        t.start()
        threads.append(t)
    for t in threads:
        t.join()


if __name__ == '__main__':
    main()

```

```
Thread Thread-1 working on something...
Thread Thread-2 working on something...
Thread Thread-3 working on something...
Thread Thread-4 working on something...
Thread Thread-4 waiting for others...
Thread Thread-1 waiting for others...
Thread Thread-2 waiting for others...
Thread Thread-3 waiting for others...
Barrier has been lifted: continuing
Barrier has been lifted: continuing
Barrier has been lifted: continuing
Barrier has been lifted: continuing

Process finished with exit code 0
```

Communication entres threads

Python Concurrence

Communications entre threads

- La communication est l'une des parties les plus importantes de vos systèmes concurrents
- Sans des mécanismes de communication appropriés mis en œuvre, les gains de performance que nous parvenons à obtenir grâce à l'utilisation de la concurrence et du parallélisme pourraient être inutiles
- La communication représente l'un des plus grands défis que vous aurez à surmonter en matière de communication entre les threads et les processus, et il est essentiel de bien comprendre toutes les options disponibles avant de plonger dans une telle programmation

Structures de données standards

- Certaines fonctions de structure de données traditionnelles de Python fournissent divers degrés de sécurité de thread par défaut
- Cependant, dans la plupart des cas, nous devrons définir une forme de mécanisme de verrouillage pour contrôler l'accès à ces structures de données afin de garantir la sécurité des threads

Etendre une classe

- Pendant mon temps de travail avec la communication entre plusieurs threads en Python, j'ai découvert qu'une excellente solution pour utiliser des Sets d'une manière "thread safe" est d'étendre réellement la classe set et d'implémenter mon propre mécanisme de verrouillage autour des actions que je souhaite effectuer
- Si vous avez l'habitude de travailler en Python, étendre la classe devrait être une opération assez simple
- Nous définissons un objet de classe ***LockedSet***, qui hérite de notre classe de ***set*** Python traditionnelle
- Dans le constructeur de cette classe, nous créons un objet ***lock***, que nous utiliserons dans les fonctions suivantes afin de permettre des interactions "thread safe"

Etendre une classe

- En dessous de notre constructeur, nous définissons les fonctions ***add***, ***remove*** et ***contains***
- Celles-ci s'appuient sur la fonctionnalité ***super*** classe avec une exception clé
- Avec chacune de ces fonctions, nous utilisons le verrou que nous avons initialisé dans notre constructeur pour nous assurer que toutes les interactions ne peuvent être exécutées que par un thread à la fois, assurant ainsi la sécurité du thread
- Il convient de noter que nous pourrions utiliser cette même technique d'extension de la classe set existante avec d'autres primitives Python
- En mettant en œuvre les nôtres, nous pouvons alors, essentiellement, tirer parti des fonctionnalités sous-jacentes de ces classes avec un minimum d'effort de notre part

Etendre une classe

- Il convient de noter que cette tactique d'extension des classes existantes et d'ajout de votre propre logique thread-safe peut être effectuée pour la plupart, sinon toutes, des primitives Python
- C'est un excellent moyen de tirer parti de certaines des excellentes fonctionnalités qui accompagnent ces classes par défaut, mais vous devez vous assurer que la façon dont vous implémentez la sécurité des threads est correcte

thread_safe_set.py

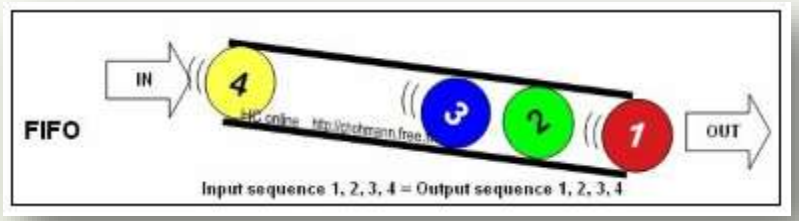
```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from threading import Lock


class LockedSet(set):
    """A thread-safe Set type implementation
    """

    def __init__(self, *args, **kwargs):
        self.__lock = Lock()
        super(LockedSet, self).__init__(*args, **kwargs)

    def add(self, element):
        with self.__lock:
            super(LockedSet, self).add(element)

    def remove(self, element):
        with self.__lock:
            super(LockedSet, self).remove(element)

```

Files d'attente

- Les files d'attente viennent dans une gamme de styles différents
- En Python, nous avons la possibilité de définir trois types différents de files d'attente à partir du module de file d'attente natif
- Ce sont des files d'attente ***normales (FIFO)***, ***LifoQueues*** et ***PriorityQueues***
- Les files d'attente, par défaut, sont "thread safe" en Python, ce qui signifie que nous n'avons pas à nous soucier de l'implémentation de mécanismes de verrouillage complexes si nous souhaitons utiliser des files d'attente dans nos applications
- Cela les rend incroyablement puissantes quand il s'agit de mettre en œuvre un moyen de communication rapide et facile à travers lequel nos nombreux threads et processus peuvent communiquer

Files d'attente *FIFO*

- Les files d'attente FIFO (premier entré, premier sorti) pour leur donner leur nom complet, sont l'implémentation de file d'attente standard que Python a à offrir
- Elles suivent exactement le même mécanisme de file d'attente que vous le feriez si vous étiez, disons, au supermarché
- La première personne à atteindre la caisse serait d'abord prise en charge, la deuxième personne attend et est servie en deuxième, et ainsi de suite
- En suivant ce mécanisme, nous nous assurons que nos clients sont traités équitablement et que vous serez en mesure d'estimer approximativement combien de temps cela prendra pour que vous soyez servis si vous étiez, disons, 7ème dans la file d'attente



Files d'attente *FIFO*

- Exemple :

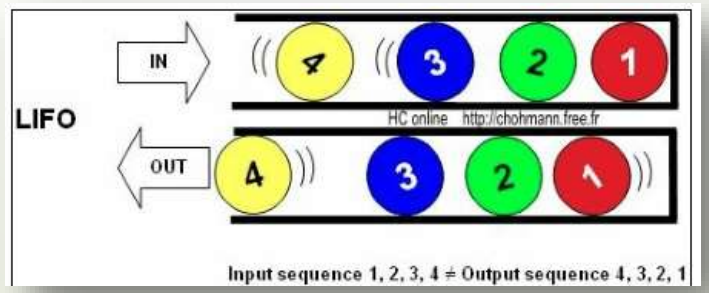
queues_fifo.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import queue


def my_subscriber(my_queue):
    while not my_queue.empty():
        item = my_queue.get()
        if item is None:
            break
        print('{} Removed {} from queue'.format(threading.current_thread().getName(), item))
        my_queue.task_done()
        time.sleep(1)


def main():
    my_queue = queue.Queue()
    for i in range(10):
        my_queue.put(i)
    print('my_queue populated')
    threads = []
    for i in range(4):
        t = threading.Thread(target=my_subscriber, args=(my_queue,))
        t.start()
        threads.append(t)
    my_queue.join()
    print('Queue is empty')
    for t in threads:
        t.join()


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

```
my_queue populated
Thread-1 Removed 0 from queue
Thread-2 Removed 1 from queue
Thread-3 Removed 2 from queue
Thread-4 Removed 3 from queue
Thread-1 Removed 4 from queue
Thread-4 Removed 5 from queue
Thread-3 Removed 6 from queue
Thread-2 Removed 7 from queue
Thread-4 Removed 8 from queue
Thread-1 Removed 9 from queue
Queue is empty

Process finished with exit code 0
```

Files d'attente *LIFO*

- Les files d'attente LIFO (dernier entré, premier sorti) agissent à l'opposé des files d'attente FIFO normales
- Pour étendre notre analogie de supermarché, en utilisant un mécanisme de mise en file d'attente LIFO, nous servons essentiellement la dernière personne à rejoindre la file d'attente avant que les membres existants de la file d'attente soient servis
- Comme vous pouvez l'imaginer, s'il s'agissait d'un supermarché de la vie réelle, il y aurait probablement un certain nombre de plaintes déposées par des gens qui passaient des heures assis dans la même file d'attente



Files d'attente *LIFO*

- Dans les files d'attente LIFO, il y a la possibilité distincte que quelques-unes des premières personnes à rejoindre la file d'attente puissent rester dans cette position indéfiniment car de plus en plus de personnes rejoignent la file d'attente avant de pouvoir être servies
- Même si cela n'a pas de sens en tant que mécanisme de mise en file d'attente dans le monde réel, le LIFO a ses avantages en matière de programmation
- Les files d'attente LIFO sont particulièrement utiles lorsqu'il s'agit de mettre en œuvre des algorithmes basés sur l'intelligence artificielle, tels que la recherche en profondeur, la recherche en profondeur limitée, etc.
- Elles est également très utiles lorsque vous voulez inverser l'ordre de quelque chose - il suffit de remplir votre file d'attente LIFO avec chaque élément, puis de les retirer à nouveau une fois que vous avez terminé

Files d'attente *LIFO*

- Exemple :

queues_lifo.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import queue


def my_subscriber(my_queue):
    while not my_queue.empty():
        item = my_queue.get()
        if item is None:
            break
        print('{} Removed {} from queue'.format(threading.current_thread().getName(), item))
        my_queue.task_done()
        time.sleep(1)


def main():
    my_queue = queue.LifoQueue()
    for i in range(10):
        print('Put {} in my_queue'.format(i))
        my_queue.put(i)
    print('my_queue populated')
    threads = []
    for i in range(4):
        t = threading.Thread(target=my_subscriber, args=(my_queue,))
        t.start()
        threads.append(t)
    my_queue.join()
    print('Queue is empty')
    for t in threads:
        t.join()


if __name__ == '__main__':
    main()

```

```
Put 0 in my_queue
Put 1 in my_queue
Put 2 in my_queue
Put 3 in my_queue
Put 4 in my_queue
Put 5 in my_queue
Put 6 in my_queue
Put 7 in my_queue
Put 8 in my_queue
Put 9 in my_queue
my_queue populated
Thread-1 Removed 9 from queue
Thread-2 Removed 8 from queue
Thread-3 Removed 7 from queue
Thread-4 Removed 6 from queue
Thread-4 Removed 5 from queue
Thread-2 Removed 4 from queue
Thread-3 Removed 3 from queue
Thread-1 Removed 2 from queue
Thread-1 Removed 1 from queue
Thread-2 Removed 0 from queue
Queue is empty

Process finished with exit code 0
```

Files d'attente *Priority*

- Si nous nous éloignons de l'analogie de notre supermarché et que nous pensons maintenant à la zone de sécurité d'un aéroport, il y a des gens qui sont plus importants que les clients réguliers
- Ce sont des gens comme les pilotes, le personnel de cabine et d'autres
- Dans ces circonstances exceptionnelles, nous les déplaçons habituellement vers l'avant de la file d'attente afin qu'ils puissent accéder aux avions dans lesquels nous sommes sur le point de voler pour les préparer au décollage
- En d'autres termes, nous leur donnons une forme de priorité dans notre mécanisme de mise en file d'attente
- Parfois, dans les systèmes que nous développons, nous devons également prendre en compte un certain type de mécanisme de hiérarchisation de sorte que des tâches incroyablement importantes ne soient pas bloquées par des millions d'opérations relativement sans importance pour des périodes indéfinies
- C'est ici que notre objet ***PriorityQueue*** entre en jeu

Files d'attente ***Priority***

- Exemple :

queues_priority.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import random
import queue


def my_subscriber(my_queue):
    while not my_queue.empty():
        item = my_queue.get()
        if item is None:
            break
        print('{} Removed {} from queue'.format(threading.current_thread().getName(), item))
        my_queue.task_done()
        time.sleep(1)


def main():
    my_queue = queue.PriorityQueue()
    for i in range(5):
        priority = random.randint(1, 10)
        print('Put {} in my_queue with priority {}'.format(i, priority))
        my_queue.put((priority, i))
    print('my_queue populated')
    threads = []
    for i in range(2):
        t = threading.Thread(target=my_subscriber, args=(my_queue,))
        t.start()
        threads.append(t)
    my_queue.join()
    print('Queue is empty')
    for t in threads:
        t.join()


if __name__ == '__main__':
    main()

```

```
Put 0 in my_queue with priority 4
Put 1 in my_queue with priority 1
Put 2 in my_queue with priority 9
Put 3 in my_queue with priority 5
Put 4 in my_queue with priority 7
my_queue populated
Thread-1 Removed (1, 1) from queue
Thread-2 Removed (4, 0) from queue
Thread-2 Removed (5, 3) from queue
Thread-1 Removed (7, 4) from queue
Thread-1 Removed (9, 2) from queue
Queue is empty

Process finished with exit code 0
```

Files d'attente pleines/vides

- Nous devons être en mesure de limiter la taille de nos files d'attente dans nos programmes
- Si nous les laissons se développer indéfiniment, alors nous pourrions théoriquement commencer à affronter MemoryErrors
- La quantité de mémoire qu'un programme Python peut prendre est limitée par la quantité de mémoire disponible sur nos systèmes
- En limitant la taille de nos files d'attente, nous sommes en mesure de nous protéger efficacement contre ces contraintes de mémoire
- Dans cet exemple, nous allons créer une file d'attente et transmettre le paramètre ***maxsize***, qui sera mis à cinq
- Nous allons ensuite créer quatre threads distincts qui tenteront chacun de remplir cette file d'attente avec un nombre arbitraire
- Nous joignons ensuite tous nos threads nouvellement créés, et essayons de mettre autant d'éléments dans notre file d'attente que possible

La fonction *join()*

- La fonction ***join()*** sur nos objets file d'attente nous permet de bloquer l'exécution de notre thread courant jusqu'à ce que tous les éléments de la file d'attente aient été consommés
- Cela nous fournit une excellente méthode de remplacement lorsque nous devons nous assurer que tout ce que nous devons faire est fait

Files d'attente pleines/vides

- Exemple :

queues_empty_full.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import threading
import time
import queue


def my_subscriber(my_queue):
    time.sleep(1)
    while not my_queue.full():
        my_queue.put(1)
        print('{} Appended 1 to queue'.format(threading.current_thread().getName()))
        time.sleep(1)


def main():
    my_queue = queue.Queue(maxsize=5)

    threads = []
    for i in range(4):
        t = threading.Thread(target=my_subscriber, args=(my_queue,))
        t.start()
        threads.append(t)
    my_queue.join()
    for t in threads:
        t.join()


if __name__ == '__main__':
    main()

```

```
Thread-2 Appended 1 to queue
Thread-3 Appended 1 to queue
Thread-1 Appended 1 to queue
Thread-4 Appended 1 to queue
Thread-1 Appended 1 to queue

Process finished with exit code 0
```

Objet *Deque*

- Une Deque ou file d'attente à double entrée est une autre primitive de communication que nous pouvons exploiter activement dans notre quête d'une communication inter-thread sûre
- Il appartient au module de collections, et il dispose de fonctionnalités similaires à celles d'une file d'attente, à l'exception du fait que nous pouvons faire prélever et pousser des éléments dans les deux extrémités de la file d'attente

Objet *Deque*

- Exemple :

collections-deque.py

```
1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3
4   import collections
5
6
7   def main():
8       double_ended_queue = collections.deque('0123456789')
9       print('Deque: {}'.format(double_ended_queue))
10      for item in double_ended_queue:
11          print('Item {}'.format(item))
12      print("Left Most Element: {}".format(double_ended_queue[0]))
13      print("Right Most Element: {}".format(double_ended_queue[-1]))
14
15
16  if __name__ == '__main__':
17      main()
18
```

```
Deque: deque(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Left Most Element: 0
Right Most Element: 9

Process finished with exit code 0
```

Objet *Deque*

- Pouvoir interroger et visualiser tous les éléments de notre objet deque peut être utile dans certaines situations, mais, typiquement, vous voudrez interagir avec ces objets
- Dans l'exemple suivant, nous allons introduire les fonctions ***append()*** et ***appendLeft()*** qui nous permettent de publier de nouveaux éléments dans notre objet ***deque*** à la première ou à la dernière position de notre objet file d'attente

collections-deque.2py

```
1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3
4   import collections
5
6
7   def main():
8       double_ended_queue = collections.deque('123456')
9       print('Deque: {}'.format(double_ended_queue))
10      double_ended_queue.append('1')
11      print('Deque: {}'.format(double_ended_queue))
12      double_ended_queue.appendleft('6')
13      print('Deque: {}'.format(double_ended_queue))
14
15
16  if __name__ == '__main__':
17      main()
18
```

```
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', '6', '1'])
Deque: deque(['6', '1', '2', '3', '4', '5', '6', '1'])

Process finished with exit code 0
```

Objet *Deque*

- Inversement, il se peut que nous devions récupérer certains des éléments que nous publions sur notre objet ***deque***
- La façon de procéder est d'utiliser les fonctions publiques ***pop()*** et ***popleft()*** fournies avec notre objet ***deque***

collections_deque3.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import collections


def main():
    double_ended_queue = collections.deque('123456')
    print('Deque: {}'.format(double_ended_queue))
    right_pop = double_ended_queue.pop()
    print('Right pop: {}'.format(right_pop))
    left_pop = double_ended_queue.popleft()
    print('Left pop: {}'.format(left_pop))
    print('Deque: {}'.format(double_ended_queue))


if __name__ == '__main__':
    main()

```

```
Deque: deque(['1', '2', '3', '4', '5', '6'])
Right pop: 6
Left pop: 1
Deque: deque(['2', '3', '4', '5'])

Process finished with exit code 0
```

Objet *Deque*

- Etre capable de peupler un objet ***deque*** est important, car sans ce mécanisme, notre objet ***deque*** ne serait pas très utile

collections_deque4.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import collections


def main():
    import collections
    double_ended_queue = collections.deque('123456')
    print("Deque: {}".format(double_ended_queue))
    double_ended_queue.insert(5, 5)
    print("Deque: {}".format(double_ended_queue))


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

```
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', 5, '6'])

Process finished with exit code 0
```

Objet *Deque*

- Deque nous donne la possibilité de faire pivoter notre objet file d'attente de n positions vers la droite ou vers la gauche selon que le nombre passé est positif ou négatif

collections_deque5.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import collections


def main():
    double_ended_queue = collections.deque('123456')
    print("Deque: {}".format(double_ended_queue))
    double_ended_queue.rotate(3)
    print("Deque: {}".format(double_ended_queue))
    double_ended_queue.rotate(-2)
    print("Deque {}".format(double_ended_queue))


if __name__ == '__main__':
    main()

```

```
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['4', '5', '6', '1', '2', '3'])
Deque deque(['6', '1', '2', '3', '4', '5'])

Process finished with exit code 0
```

Intercepter les exceptions dans les threads enfants

- Un point important à considérer lors de l'écriture d'applications multithread est comment gérer les exceptions levées dans les threads enfants ?
- Nous avons examiné la communication inter-thread dans le chapitre précédent, donc une méthode logique pour attraper et communiquer des exceptions entre les threads enfants et parents pourrait être d'utiliser une ou plusieurs des techniques que nous avons déjà discutées
- Dans l'exemple de code suivant, nous verrons exactement comment vous pouvez communiquer les exceptions générées par un thread enfant au thread parent
- Nous allons utiliser le module ***sys*** pour extraire les informations dont nous avons besoin à propos de l'exception, puis placer ceci dans les limites de notre primitive de file d'attente thread-safe

Intercepter les exceptions dans les threads enfants

- Code :

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import sys
import threading
import time
import queue


def my_worker(my_queue):
    while True:
        try:
            time.sleep(2)
            raise Exception("Exception Thrown In Child Thread {}".format(threading.current_thread()))
        except:
            my_queue.put(sys.exc_info())
            break


def main():
    my_queue = queue.Queue()
    my_thread = threading.Thread(target=my_worker, args=(my_queue,))
    my_thread.start()
    while True:
        try:
            exception = my_queue.get()
        except queue.Empty:
            pass
        else:
            print('-->', exception)
            break

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

```
--> (<class 'Exception'>, Exception('Exception Thrown In Child Thread <Thread(Thread-1, started 4004)>',), <traceback object at 0x000001415CDC8188>)

Process finished with exit code 0
```

Benchmarking

Python Concurrence

Benchmarking

- Quand nous parlons de l'étalonnage de notre code, nous parlons de mesurer la rapidité avec laquelle il peut effectuer une opération complète
- Par exemple, prenons un robot d'indexation que nous aurions construit
- Si nous évaluons ce programme, nous mesurerons typiquement le nombre de pages que nous pourrions indexer par seconde
- Lorsque nous effectuons des optimisations de performances, nous prenons un benchmark de départ qui représente l'état actuel de notre programme dans son ensemble, puis utilisons une combinaison de micro benchmarking et de profilage afin d'optimiser ces programmes et d'atteindre un débit supérieur

Benchmarking

- Le micro benchmarking consiste essentiellement à décomposer notre application en une série d'étapes, puis à analyser chacune de ces étapes individuellement afin de déterminer les goulets d'étranglement dans notre code
- Nous cassons, ce qui est essentiellement un problème difficile à optimiser dans son ensemble, dans une série de petits problèmes qui deviennent plus faciles à optimiser et à régler
- Alors, comment pouvons-nous effectuer des benchmarks sur notre code?
- Eh bien, heureusement, nous avons un certain nombre d'options que nous pouvons exploiter, qui font partie de Python

Le module *timeit*

- Le module timeit de Python est un de ces outils que nous pouvons utiliser
- Python, par défaut, est livré avec ce module timeit qui fournit un excellent moyen de mesurer les performances de petits morceaux de code Python dans votre application principale
- Le module timeit nous donne la flexibilité de faire inclure nos benchmarks dans notre base de code, ou, inversement, nous pouvons l'appeler via la ligne de commande et alimenter des sections de code que nous souhaitons chronométrer

timeit versus *time*

- Il est intéressant de noter qu'il existe un certain nombre d'avantages lors de l'utilisation du module ***timeit*** par opposition au module de temps
- timeit est spécifiquement conçu pour obtenir des mesures de temps beaucoup plus précises que le module de temps
- Avec ***timeit***, nous pouvons dire à nos programmes de faire fonctionner les choses plusieurs fois, puis nous donner une mesure précise qui est beaucoup moins susceptible d'être affectée par des facteurs externes dans notre système d'exploitation, sur lesquels nous n'avons aucun contrôle direct

Importer *timeit* dans notre code

- Cet exemple couvrira de façon très simple l'utilisation du module timeit pour mesurer le temps nécessaire à l'exécution de deux fonctions distinctes
- Ce code démarre, puis exécute chaque fonction, puis enregistre une heure de fin avant d'imprimer la différence précise entre les deux heures
- Il convient de noter que, bien que nous ayons réussi à mesurer le temps pris pour chaque fonction, nous n'avons pas réellement utilisé le module *timeit* à son plein potentiel

timeit_1.py

```
9       print("Function 1 Executing")
10      time.sleep(5)
11      print("Function 1 complete")
12
13
14  def func2():
15      print("Function 2 executing")
16      time.sleep(6)
17      print("Function 2 complete")
18
19
20  def main():
21      start_time = timeit.default_timer()
22      func1()
23      print(timeit.default_timer() - start_time)
24      start_time = timeit.default_timer()
25      func2()
26      print(timeit.default_timer() - start_time)
27
28
29  if __name__ == '__main__':
30      main()
31
```

```
Function 1 Executing
Function 1 complete
4.99061926296523
Function 2 executing
Function 2 complete
6.021829917301762

Process finished with exit code 0
```

Importer *timeit* dans notre code

- Dans ce code, nous instancions deux objets ***Timer***, chacun prenant en compte la fonction qu'ils vont chronométrer ainsi que les imports nécessaires pour les exécuter dans le temps
- Nous appelons ensuite ***.repeat()*** sur ces deux objets ***Timer***, en passant en ***repeat = 2*** pour déterminer combien de fois nous voulons chronométrer notre code, et ***number = 1*** pour déterminer combien de fois nous voulons exécuter ces tests
- L'exécution du code précédent doit fournir la sortie suivante :

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import timeit
import time


def func1():
    print("Function 1 Executing")
    time.sleep(5)
    print("Function 1 complete")


def func2():
    print("Function 2 executing")
    time.sleep(6)
    print("Function 2 complete")


def main():
    t1 = timeit.Timer("func1()", setup="from __main__ import func1")
    times = t1.repeat(repeat=2, number=1)
    for t in times:
        print("{} Seconds: ".format(t))
    t2 = timeit.Timer("func2()", setup="from __main__ import func2")
    times = t2.repeat(repeat=2, number=1)
    for t in times:
        print("{} Seconds: ".format(t))


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

```
Function 1 Executing
Function 1 complete
Function 1 Executing
Function 1 complete
4.999559901408434 Seconds:
5.009151206261252 Seconds:
Function 2 executing
Function 2 complete
Function 2 executing
Function 2 complete
6.0057848866035 Seconds:
6.003403850507119 Seconds:

Process finished with exit code 0
```

Utiliser des décorateurs

- Parfois, cependant, insérer manuellement le code précédent peut être quelque peu exagéré, et peut finir par gonfler votre base de code inutilement
- Heureusement, Python offre une solution à cela
- Nous pouvons définir notre propre fonction décorateur, qui encapsulera automatiquement l'exécution de notre fonction avec deux appels à ***timeit.default_timer()***
- Nous récupérerons ensuite les différences entre ces deux appels et l'afficherons sur la console

Utiliser des décorateurs

- Ce code précédent affichera toutes les fonctions que nous lui transmettons avec l'heure exacte de son exécution
- Lorsque vous l'exécutez, vous devriez voir la sortie suivante sur la console

timeit_2.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import timeit
import random
import time


def time_this(func):
    def function_timer(*args, **kwargs):
        start_time = timeit.default_timer()
        value = func(*args, **kwargs)
        run_time = timeit.default_timer() - start_time
        print('Function {} took {} seconds to execute'.format(func.__name__, run_time))
        return value

    return function_timer


@time_this
def long_runner():
    for x in range(5):
        sleep_time = random.choice(range(1, 3))
        time.sleep(sleep_time)


def main():
    long_runner()


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

```
Function long_runner took 7.026576977154775 seconds to execute

Process finished with exit code 0
```

timeit context manager

- Les gestionnaires de contexte sont des objets qui définissent le contexte d'exécution à établir lors de l'exécution d'une instruction ***with***
- En Python, nous pouvons définir notre propre objet de gestionnaire de contexte ***Timer***, que nous pouvons ensuite utiliser pour chronométrer des sections spécifiques de notre code sans trop d'impact négatif sur notre base de code
- Cette fois, l'objet du gestionnaire de contexte ressemblera à ceci :

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import timeit
import ssl
from urllib.request import Request, urlopen


class Timer(object):
    def __init__(self, verbose=False):
        self.verbose = verbose
        self.timer = timeit.default_timer

    def __enter__(self):
        self.start = timeit.default_timer()
        return self

    def __exit__(self, *args):
        end = timeit.default_timer()
        self.elapsed_secs = end - self.start
        self.elapsed = self.elapsed_secs * 1000
        if self.verbose:
            print('elapsed time: {} ms'.format(self.elapsed))


def my_function():
    myssl = ssl.create_default_context()
    myssl.check_hostname = False
    myssl.verify_mode = ssl.CERT_NONE
    with Timer(verbose=True) as t:
        req = Request('https://tutorialedge.net', headers={'User-Agent': 'Mozilla/5.0'})
        response = urlopen(req, context=myssl)
    print("Elapsed Time: {} seconds".format(t.elapsed_secs))


def main():
    my_function()


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

```
elapsed time: 2943.403479149243 ms
Elapsed Time: 2.943403479149243 seconds

Process finished with exit code 0
```

Profilage de code

- Lorsque nous parlons de profilage de notre code, nous avons l'intention de mesurer certains attributs clés de nos programmes, tels que la quantité de mémoire qu'ils utilisent, la complexité temporelle de nos programmes ou l'utilisation d'instructions particulières
- C'est un outil essentiel dans l'arsenal d'un programmeur quand il s'agit de tirer le meilleur parti de leurs systèmes
- Le profilage utilise généralement une technique appelée Analyse de programme dynamique pour réaliser ses mesures, ce qui implique l'exécution de nos programmes sur un processeur réel ou virtuel
- La technique remonte aux plates-formes IBM/360 et IBM/370 au début des années 1970

cProfile

- ***cProfile*** est un module basé sur C qui fait partie de Python en standard
- Nous pouvons l'utiliser pour comprendre les caractéristiques suivantes de notre code:
  - ***ncalls*** : C'est le nombre de fois qu'une ligne / fonction est appelée pendant l'exécution de notre programme.
  - ***tottime***: Temps total nécessaire à l'exécution de la ligne ou de la fonction.
  - ***percall***: Temps total divisé par le nombre d'appels
  - ***cumtime***: Temps cumulé passé à exécuter cette ligne ou fonction
  - ***percall***: Quotient de cumtime divisé par le nombre d'appels primitifs.
  - ***filename***:***lineno*** (***function***): La ligne ou la fonction que nous mesurons
- Jetons un coup d'œil à la façon dont nous pouvons utiliser le module ***cProfile*** pour atteindre ces attributs sur certains de nos précédents exemples Python

cProfile

- Reprenons un des exemples précédents :

collections-deque.2py

```
1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3
4   import collections
5
6
7   def main():
8       double_ended_queue = collections.deque('123456')
9       print('Deque: {}'.format(double_ended_queue))
10      double_ended_queue.append('1')
11      print('Deque: {}'.format(double_ended_queue))
12      double_ended_queue.appendleft('6')
13      print('Deque: {}'.format(double_ended_queue))
14
15
16  if __name__ == '__main__':
17      main()
18
```

```
D:\pythonworkspace\learning-python\samples>python -m cProfile collections_deque2.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', '6', '1'])
Deque: deque(['6', '1', '2', '3', '4', '5', '6', '1'])
         12 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 collections_deque2.py:4(<module>)
        1    0.000    0.000    0.001    0.001 collections_deque2.py:7(main)
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
        1    0.000    0.000    0.000    0.000 {method 'appendleft' of 'collections.deque' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        3    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
```

Executors et Pools

Python Concurrence

Executors et Pools

- Dans cette partie, nous examinerons en profondeur des concepts tels que les pools de threads et les pools de processus, et comment nous pouvons travailler avec la mise en œuvre de ces concepts par Python afin d'accélérer l'exécution de nos programmes
- Nous aborderons les sujets suivants en détail :
  - Concurrent Futures
  - Future Objects
  - Process Pool Executor

Concurrent futures

- Les Concurrent Futures sont une nouvelle fonctionnalité ajoutée à Python dans la version 3.2
- Si vous venez d'un environnement Java, vous connaissez peut-être ***ThreadPoolExecutor***
- Les Concurrent Futures sont l'implémentation de Python de ce concept ***ThreadPoolExecutor***
- Lorsqu'il s'agit d'exécuter des tâches multithread, l'une des tâches les plus coûteuses en termes de calcul consiste à démarrer des threads
- Les ***ThreadPoolExecutors*** contournent ce problème en créant un pool de threads qui vivront aussi longtemps que nous en avons besoin
- Nous n'avons plus à créer et exécuter un thread pour effectuer une tâche et continuer à le faire pour chaque tâche dont nous avons besoin

Concurrent futures

- Nous pouvons, au lieu de cela, créer un thread une seule fois, puis le nourrir constamment de nouveaux travaux à faire
- Une bonne façon d'y penser est d'imaginer que vous étiez dans un bureau qui compterait de nombreux travailleurs
- Nous n'embaucherions pas un employé, ne le formerions pas et nous lui attribuerions un et un seul emploi avant de le licencier
- Au lieu de cela, nous ne ferions qu'exercer ce coûteux processus de formation, puis nous leur attribuerions de nombreux emplois pendant toute la durée de leur emploi, ce qui leur permettrait d'atteindre une plus grande efficacité
- Ceci est analogue à notre pool de threads
- Nous engageons un certain nombre de threads une fois, et engageons le coût de les créer une fois avant de leur déléguer de nombreuses tâches tout au long de leur vie.

Executor Objects

- La classe ***Executor*** est incluse dans le module ***concurrent.futures*** et nous permet d'exécuter simultanément un certain nombre d'appels différents
- Elle peut être utilisé de différentes manières, comme par elle-même ou en tant que gestionnaire de contexte, et elle peut considérablement améliorer la lisibilité de notre code en manipulant des tâches fastidieuses telles que la création de threads, le démarrage et la jointure

Créer un ThreadPoolExecutor

- La chose que nous devons connaitre est comment définir nos propres ThreadPoolExecutors
- C'est un one-liner plutôt simple, qui ressemble à ceci :
- ***executor = ThreadPoolExecutor(max_workers=3)***
- Avec cette instruction, nous instancions notre ***ThreadPoolExecutor***, et transmettons le nombre maximum de travailleurs que nous voulons avoir
- Dans ce cas, nous l'avons défini comme 3, ce qui signifie essentiellement que ce pool de threads n'aura que trois threads concurrents pouvant traiter tous les jobs que nous lui soumettons
- Afin de donner quelque chose aux threads de notre ThreadPoolExecutor, nous pouvons appeler la fonction ***submit()***, qui utilise une fonction comme paramètre principal comme ceci :
- ***executor.submit(myFunction())***

Créer un ThreadPoolExecutor

- Exemple de code :

```
Executing Task #1... (<Thread(Thread-1, started daemon 4648)>)
Executing Task #2... (<Thread(Thread-2, started daemon 13468)>)
Executing Task #3... (<Thread(Thread-3, started daemon 2672)>)
Result: 0 Task #3 (<Thread(Thread-3, started daemon 2672)>)
Result: 0 Task #1 (<Thread(Thread-1, started daemon 4648)>)
Result: 0 Task #2 (<Thread(Thread-2, started daemon 13468)>)
Result: 1 Task #2 (<Thread(Thread-2, started daemon 13468)>)
Result: 1 Task #3 (<Thread(Thread-3, started daemon 2672)>)
Task executed <Thread(Thread-3, started daemon 2672)>
Result: 1 Task #1 (<Thread(Thread-1, started daemon 4648)>)
Result: 3 Task #1 (<Thread(Thread-1, started daemon 4648)>)
Task executed <Thread(Thread-1, started daemon 4648)>
Result: 3 Task #2 (<Thread(Thread-2, started daemon 13468)>)
Result: 6 Task #2 (<Thread(Thread-2, started daemon 13468)>)
Result: 10 Task #2 (<Thread(Thread-2, started daemon 13468)>)
Task executed <Thread(Thread-2, started daemon 13468)>

Process finished with exit code 0
```

executor_1.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from concurrent.futures import ThreadPoolExecutor
import threading
import time


def task(nb, name):
    print('Executing {}... ({})'.format(name, threading.current_thread()))
    result = 0
    for i in range(nb):
        result += i
        time.sleep(0.5)
        print('Result: {} {} ({})'.format(result, name, threading.current_thread()))
    print('Task executed {}'.format(threading.current_thread()))


def main():
    print('\n')
    executor = ThreadPoolExecutor(max_workers=3)
    executor.submit(task, 3, 'Task #1')
    executor.submit(task, 5, 'Task #2')
    executor.submit(task, 2, 'Task #3')


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

Context manager

- La deuxième méthode, et peut-être la plus populaire, d'instanciation de ***ThreadPoolExecutor*** est de l'utiliser comme gestionnaire de contexte :
  - ***with ThreadPoolExecutor(max_workers=3) as executor***
- Cette méthode fait en pratique le même travail que la première approche mais, syntaxiquement, elle peut être avantageuse pour nous en tant que développeurs dans certains scénarios
- Les gestionnaires de contextes, si vous ne les avez jamais rencontrés auparavant, sont un concept incroyablement puissant de Python qui nous permet d'écrire du code plus syntaxique

Context manager

- Cette fois, nous allons définir une tâche différente qui prend en entrée une variable ***n*** pour vous donner une simple démonstration de la façon dont nous pouvons le faire
- La fonction imprime simplement qu'elle traite ***n***, et rien de plus
- Dans notre fonction principale, nous utilisons notre ***ThreadPoolExecutor*** en tant que gestionnaire de contexte, puis nous appelons successivement :
  - ***future = executor.submit(task,(n))***
- pour donner à notre pool de tâches quelque chose à faire

executor_2.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from concurrent.futures import ThreadPoolExecutor
5
6
7  def task(n):
8      print('Processing {}'.format(n))
9
10
11 def main():
12     print('\n')
13     print('Starting ThreadPoolExecutor')
14     with ThreadPoolExecutor(max_workers=3) as executor:
15         executor.submit(task, 3)
16         executor.submit(task, 5)
17         executor.submit(task, 2)
18     print('All tasks complete')
19
20
21 if __name__ == '__main__':
22     main()
23
```

```
Starting ThreadPoolExecutor
Processing 3
Processing 5
Processing 2
All tasks complete

Process finished with exit code 0
```

Maps

- Les ***maps*** en Python nous permettent de faire des choses sympas comme d'appliquer une certaine fonction à tous les éléments d'un itérable :
- ***map(func, *iterables, timeout=None, chunksize=1)***
- Heureusement, dans Python, nous pouvons réellement mapper tous les éléments d'un itérateur à une fonction, et les soumettre en tant que travaux indépendants à notre ***ThreadPoolExecutor*** :
  - ***results = executor.map(multiply_by_two, values)***
- Ceci, essentiellement, nous évite de faire quelque chose de bien plus verbeux comme l'exemple suivant:
  - ***for value in values:***

    executor.submit(multiply_by_two, (value))

Maps

- Dans cet exemple, nous allons utiliser cette nouvelle fonction map pour appliquer notre fonction ***multiply_by_two*** à chaque valeur de notre tableau de valeurs :

executor_3.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from concurrent.futures import ThreadPoolExecutor


def multiply_by_two(n):
    return n * 2


def main():
    print('\n')
    print('Starting ThreadPoolExecutor')
    values = [2, 3, 4, 5, 6, 7, 8]
    with ThreadPoolExecutor(max_workers=3) as executor:
        results = executor.map(multiply_by_two, values)
    for result in results:
        print(result)
    print('All tasks complete')


if __name__ == '__main__':
    main()

```

```
Starting ThreadPoolExecutor
4
6
8
10
12
14
16
All tasks complete

Process finished with exit code 0
```

Multiprocessing

Python Concurrence

Contournement du GIL

- Le verrouillage global de l'interpréteur (GIL) peut parfois être un véritable frein aux performances pour les tâches liées à l'UC
- Tout au long de ce livre, nous avons examiné des techniques, telles que la programmation asynchrone, qui pourraient minimiser l'impact de ce verrou d'interpréteur global sur les performances de notre système python
- Cependant, avec l'utilisation du multitraitement, nous pouvons contourner efficacement cette limitation, grâce à l'utilisation de plusieurs processus
- En utilisant plusieurs processus, nous utilisons plusieurs instances de la GIL et, à ce titre, nous ne sommes pas limités à exécuter le byte code d'un thread dans nos programmes à un moment donné
- Le multitraitement en Python nous permet d'exprimer nos programmes de telle sorte que nous puissions utiliser pleinement la puissance de traitement de nos processeurs

Utilisation des sous-processus

- Au sein de Python, nous avons la possibilité de lancer plusieurs processus qui peuvent être exécutés sur des cœurs distincts au sein de notre CPU
- Dans cet exemple, nous allons créer un processus enfant qui exécutera simplement certaines instructions d'impression
- Cela représente la manière la plus simple de générer des processus enfants à l'aide du module de multitraitement, et c'est probablement la manière maladroite que vous pourriez mettre en œuvre le multitraitement dans vos applications Python :

multiprocessing_1.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import multiprocessing
5
6
7  def my_worker():
8      print("Currently Executing Child Process")
9      print("This process has it's own instance of the GIL")
10     print("Executing Main Process")
11     print("Creating Child Process")
12
13
14 def main():
15     my_process = multiprocessing.Process(target=my_worker())
16     my_process.start()
17     my_process.join()
18     print("Child Process has terminated, terminating main process")
19
20
21 if __name__ == '__main__':
22     main()
23
```

```
Currently Executing Child Process
This process has it's own instance of the GIL
Executing Main Process
Creating Child Process
Child Process has terminated, terminating main process

Process finished with exit code 0
```

Fork d'un processus

- Le "fork" est le mécanisme utilisé sur les systèmes Unix pour créer des processus enfants à partir du processus parent
- Ces processus enfants sont presque identiques à leur processus parent et similaires au monde réel
- Les enfants héritent de toutes les ressources disponibles pour le parent
- ***La commande fork est une commande système standard trouvée dans l'écosystème Unix***

spawn d'un processus

- En générant un processus séparé, nous générons un second processus d'interpréteur Python distinct
- Cela inclut son propre verrou d'interpréteur global distinct, et, en tant que tel, chaque processus est capable d'exécuter des choses en parallèle, car nous ne sommes plus limités par les limitations d'un simple verrou d'interpréteur global
- Chaque processus fraîchement engendré n'hérite que des ressources nécessaires pour exécuter ce qui est passé dans sa méthode d'exécution
- C'est le mécanisme standard que les machines ***Windows*** utilisent pour lancer de nouveaux processus, mais il peut également être utilisé sur les systèmes ***Unix***

Fork server

- Les ***fork servers*** sont un mécanisme assez étrange pour créer des processus distincts, et c'est un mécanisme qui n'est disponible que sur certaines plateformes Unix qui supportent le passage de descripteurs de fichiers sur des ***Pipes Unix***
- Si un programme sélectionne ce mécanisme pour démarrer des processus, ce qui se produit généralement est qu'un serveur est instancié
- Ce serveur gère ensuite toutes les demandes de création de processus
- Ainsi, lorsque notre programme Python tente de créer un nouveau processus, il envoie d'abord une requête à ce serveur nouvellement instancié
- Ce serveur crée ensuite le processus pour nous, et nous sommes libres de l'utiliser dans nos programmes

Daemon processus

- Les processus démons suivent à peu près le même modèle que les threads démons que nous avons rencontrés plus tôt
- Nous sommes en mesure de démoniser les processus en cours en mettant leur drapeau démon à ***True***
- Ces processus démons continueront alors à s'exécuter tant que notre thread principal est en cours d'exécution, et ne s'arrêteront que lorsqu'ils auront fini leur exécution ou quand nous tuerons notre programme principal

Daemon processus

- Dans l'exemple suivant, nous voyons à quel point il est simple de définir et de démarrer nos propres processus démon en Python

multiprocessing_2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import multiprocessing
import time
import random


def worker_process():
    print("Process started: {}".format(multiprocessing.current_process()))
    time.sleep(random.randint(3, 6))
    print("Process terminated: {}".format(multiprocessing.current_process()))


def main():
    print("Main process: {}".format(multiprocessing.current_process()))
    my_process = multiprocessing.Process(target=worker_process)
    my_process.daemon = True
    my_process.start()
    print("We can carry on as per usual and our daemon will continue to execute")
    time.sleep(10)


if __name__ == '__main__':
    main()

```

```
Main process: <_MainProcess(MainProcess, started)>
We can carry on as per usual and our daemon will continue to execute
Process started: <Process(Process-1, started daemon)>
Process terminated: <Process(Process-1, started daemon)>

Process finished with exit code 0
```

Identifier les processus utilisant des PID

- Tous les processus résidant dans un système d'exploitation comportent un identificateur de processus, généralement appelé PID
- Lorsque vous travaillez avec plusieurs processus dans vos programmes Python, vous pouvez vous attendre à ce que notre programme n'ait qu'un seul identificateur de processus, mais ce n'est pas le cas
- Au lieu de cela, chaque sous-processus que nous engendrons reçoivent leurs propres numéros de PID pour les identifier séparément dans le système d'exploitation
- Des processus séparés ayant leurs propres PID assignés peuvent être utiles quand il s'agit d'effectuer des tâches telles que la journalisation et le débogage.

Identifier les processus utilisant des PID

- Exemple :

multiprocessing_3.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import multiprocessing
import time
import random


def worker_process():
    print("Process started: {}".format(multiprocessing.current_process().pid))
    time.sleep(random.randint(3, 6))
    print("Process terminated: {}".format(multiprocessing.current_process().pid))


def main():
    print("Main process started: {}".format(multiprocessing.current_process().pid))
    my_process = multiprocessing.Process(target=worker_process)
    my_process.daemon = True
    my_process.start()
    print("We can carry on as per usual and our daemon will continue to execute")
    my_process.join()
    print("Main process terminated: {}".format(multiprocessing.current_process().pid))


if __name__ == '__main__':
    main()

```

```
Main process started: 3900
We can carry on as per usual and our daemon will continue to execute
Process started: 5804
Process terminated: 5804
Main process terminated: 3900

Process finished with exit code 0
```

Programmation évènementielle

Python Concurrence

Références

- Python.org : https://www.python.org/
- …

Outils

- IDE Pycharm Community : https://www.jetbrains.com/pycharm/
- Analyse en ligne de code Python : http://www.pythontutor.com/
- …