Python 103

Programmation concurrente (3.5.4)

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

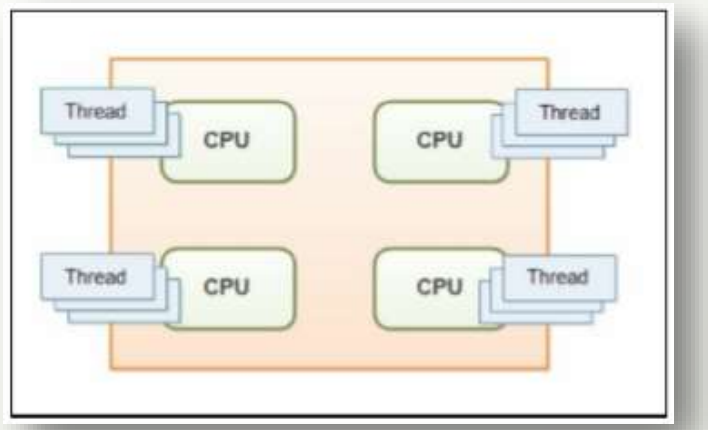
- Les concepteurs de langages sont confrontés à un défi important lorsqu'ils tentent d'implémenter une concurrence non seulement sûre, mais efficace et facile à écrire pour les utilisateurs de cette langue
- Les langages de programmation tels que Golang, Rust et même Python ont fait de grands progrès 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 ?

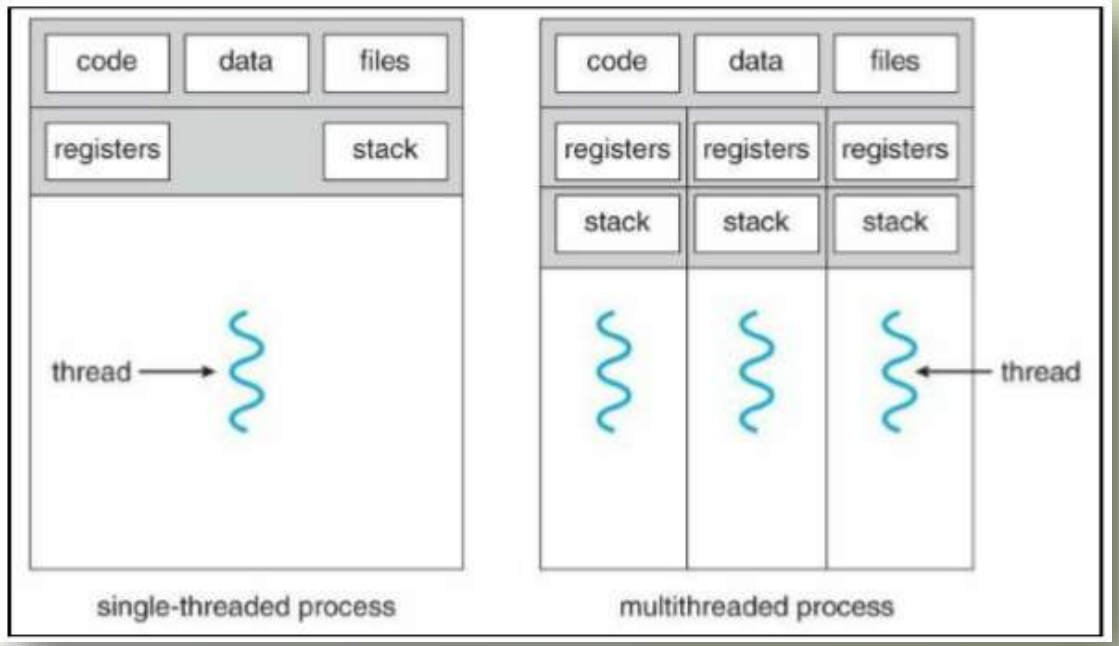
- Les processus sont très similaires aux threads
- Ils nous permettent de faire à peu près tout ce qu'un thread peut faire mais l'avantage principal est qu'ils ne sont pas liés à un noyau CPU singulier
- Si nous étendons notre analogie de bureau, cela signifie essentiellement que si nous avions un processeur 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 ?

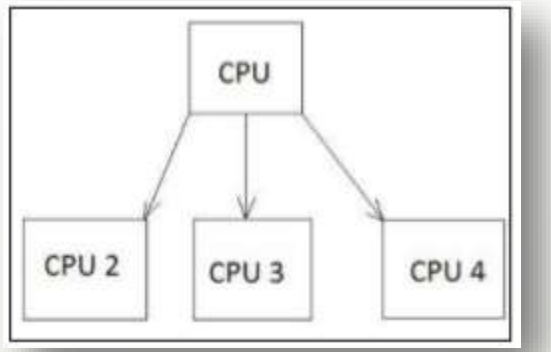
- Voici les principaux inconvénients des processus :
  - Pas de ressources partagées entre les processus
  - Nous devons mettre en œuvre une forme de IPC
  - Tout ceci nécessite plus de mémoire

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
>>>
```

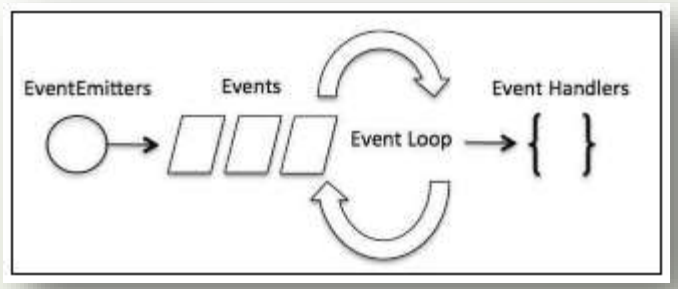
- Non seulement le multi processing nous permet d'utiliser plus de notre machine, mais nous évitons également les limitations que le verrou global d'interpréteur nous impose dans Python
- Un inconvénient potentiel de plusieurs processus est que nous n'avons intrinsèquement pas d'état partagé et que nous 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

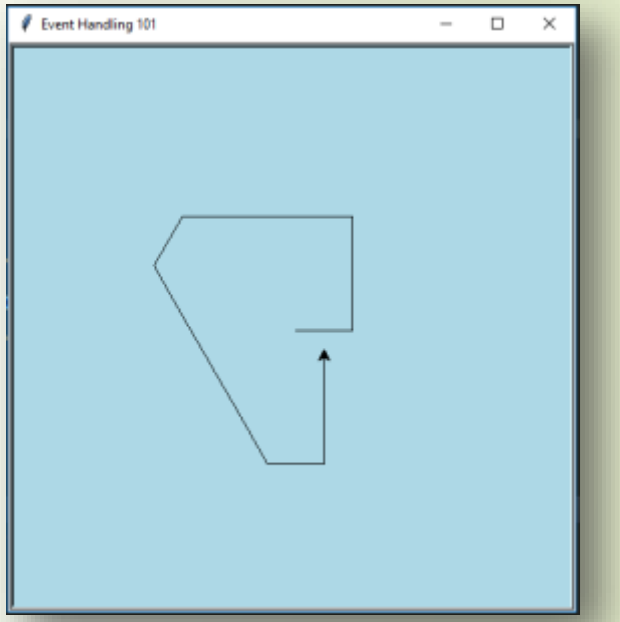
- Turtle est un module graphique qui a été écrit en Python, et c'est un excellent point de départ pour intéresser les enfants à la programmation
- Il gère toutes les complexités liées à la programmation graphique et leur permet de se concentrer uniquement sur l'apprentissage des bases tout en les gardant 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

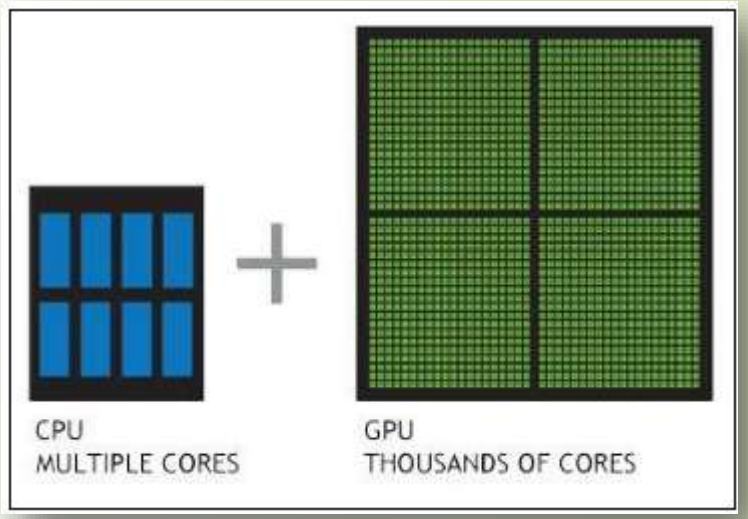
- Les GPU sont réputés pour leur capacité à rendre des jeux vidéo d'action rapides et de haute résolution
- Ils sont capables d'absorber ensemble les millions de calculs nécessaires par seconde afin de s'assurer que chaque vertex des modèles 3D de votre jeu est au bon endroit, et qu'ils sont mis à jour 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

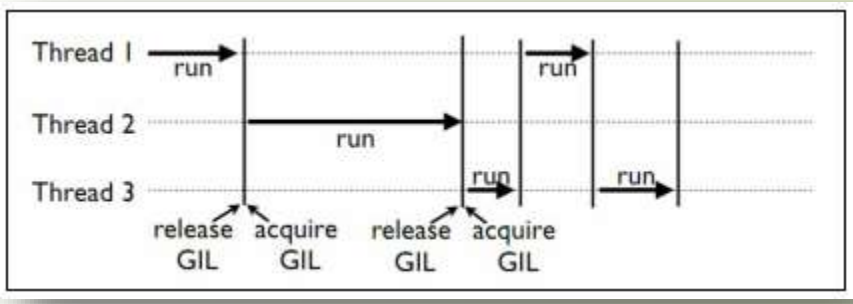
Programmation GPU

- 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

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

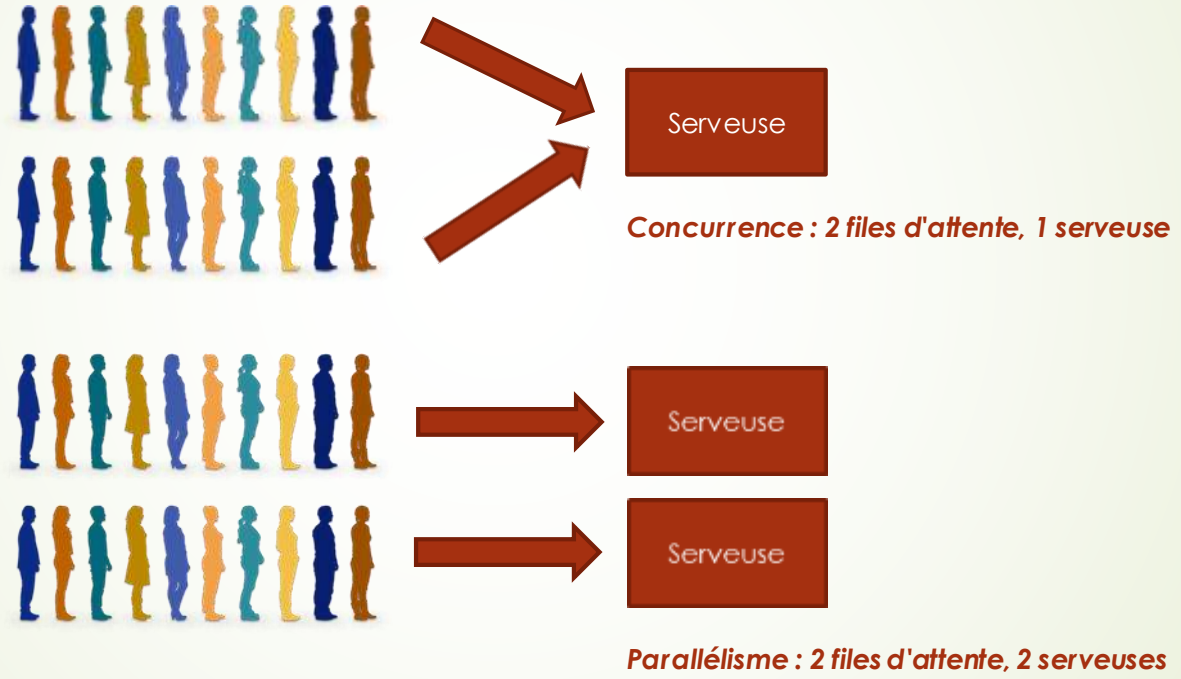
Comprendre le parallélisme

- 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

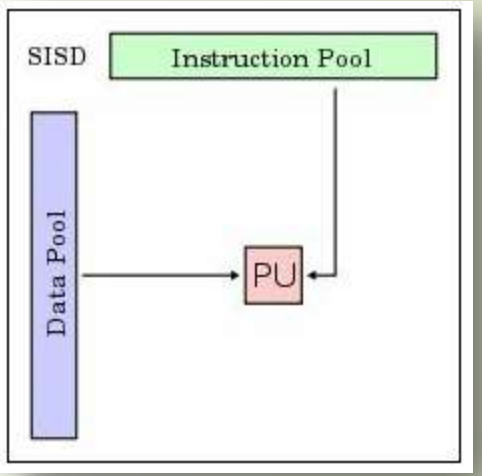
- Lorsque nous catégorisons ces différents styles, nous avons tendance à suivre une taxonomie proposée par un homme nommé Michael Flynn en 1972
- Cette taxonomie définit quatre styles différents d'architecture informatique
- Ceux-ci sont :
  - SISD : single instruction stream, single data stream
  -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

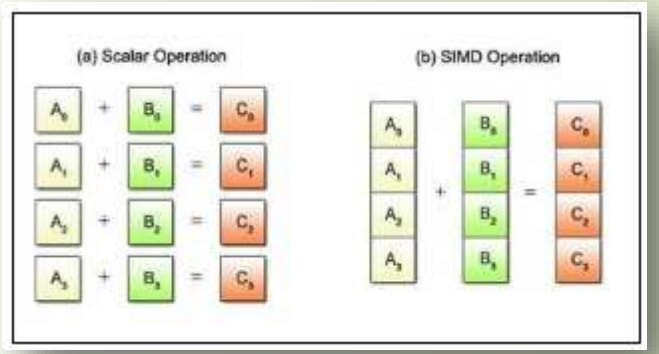
SISD

- 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

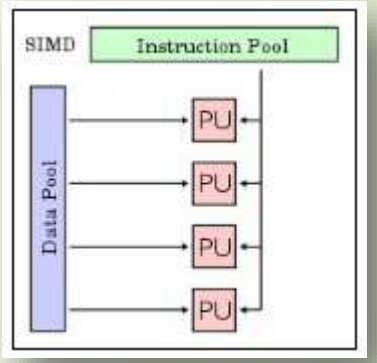


SIMD

- 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

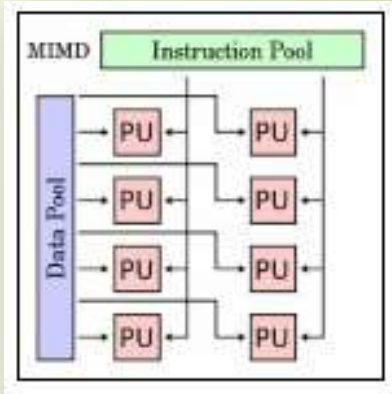


MISD

- 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

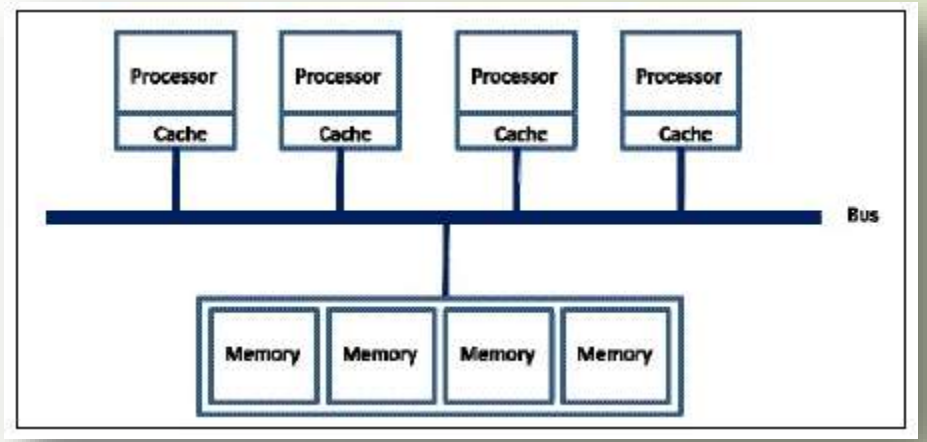
- Les concepteurs d'ordinateurs recherchent de plus en plus des moyens d'améliorer la facilité avec laquelle nous pouvons développer de nouvelles solutions parallèles aux problèmes
- L'une des façons dont ils ont réussi à améliorer les choses est de fournir un espace d'adressage physique unique auquel 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

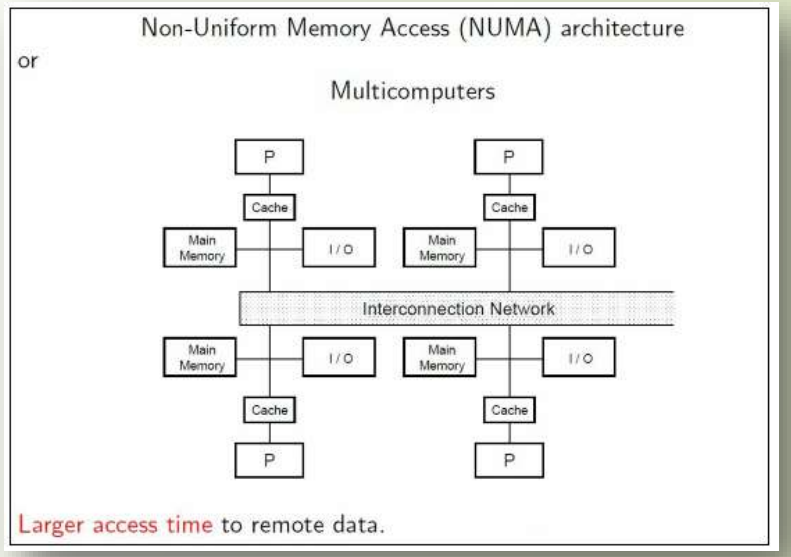
- Les avantages de l'UMA sont les suivants :
  - Tout l'accès à la RAM prend exactement le même laps de temps
  - Le cache est cohérent et consistant
  - La conception du matériel est plus simple
- Cependant, il existe un inconvénient de UMA :
  - les systèmes UMA disposent d'un bus mémoire à partir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



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

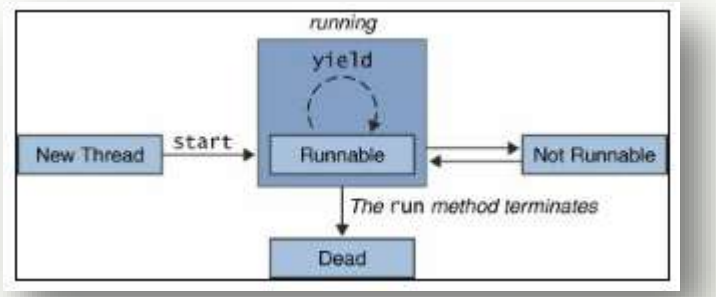
- Ce constructeur précédent prend en compte cinq arguments réels, qui sont définis comme suit dans cette définition de classe :
  - ***group*** : C'est un paramètre spécial qui est réservé pour une future extension
  - ***target*** : Ceci est l'objet appelable à appeler par la méthode ***run()***. Si 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()

```

Références

- Python.org : https://www.python.org/
- Learning Python : https://github.com/thierrydecker/learning-python
- …

Outils

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