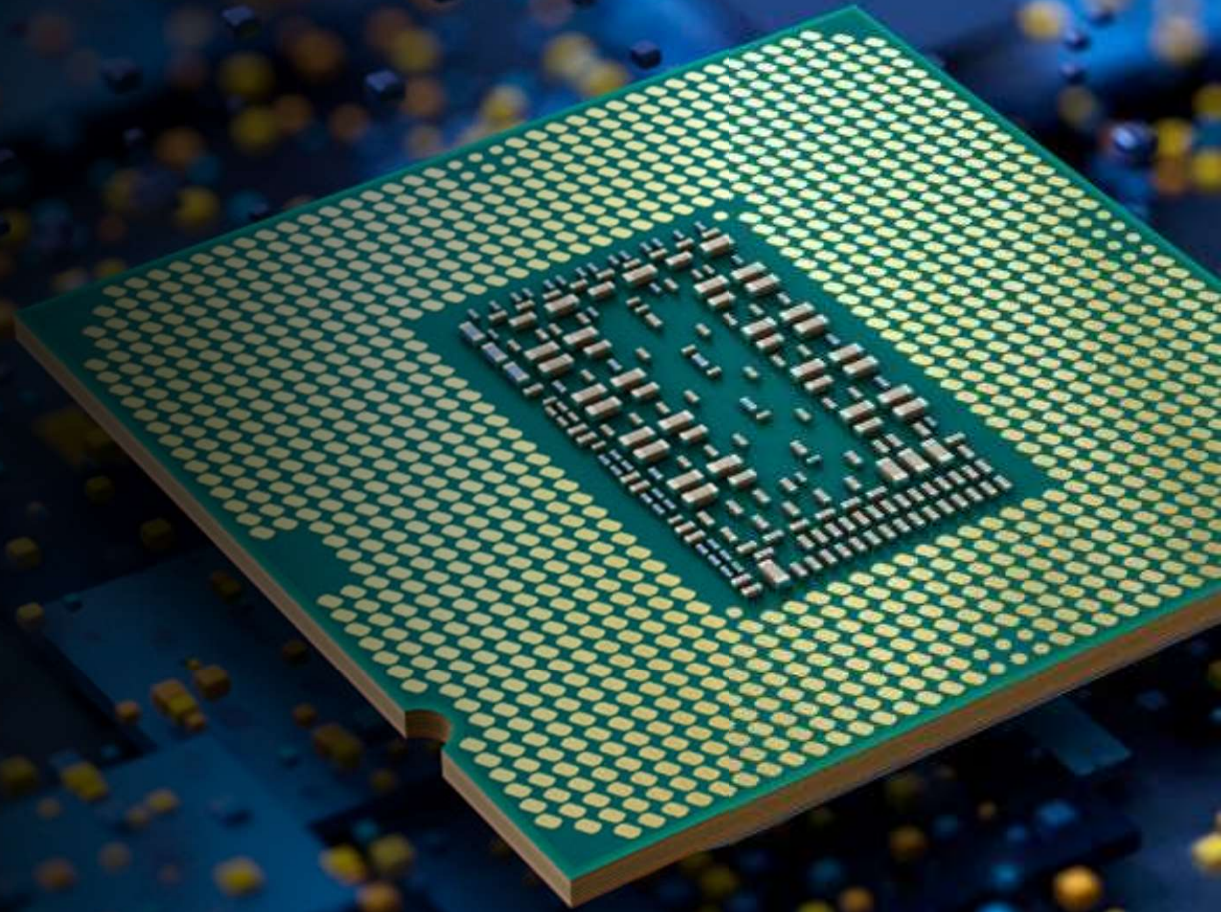


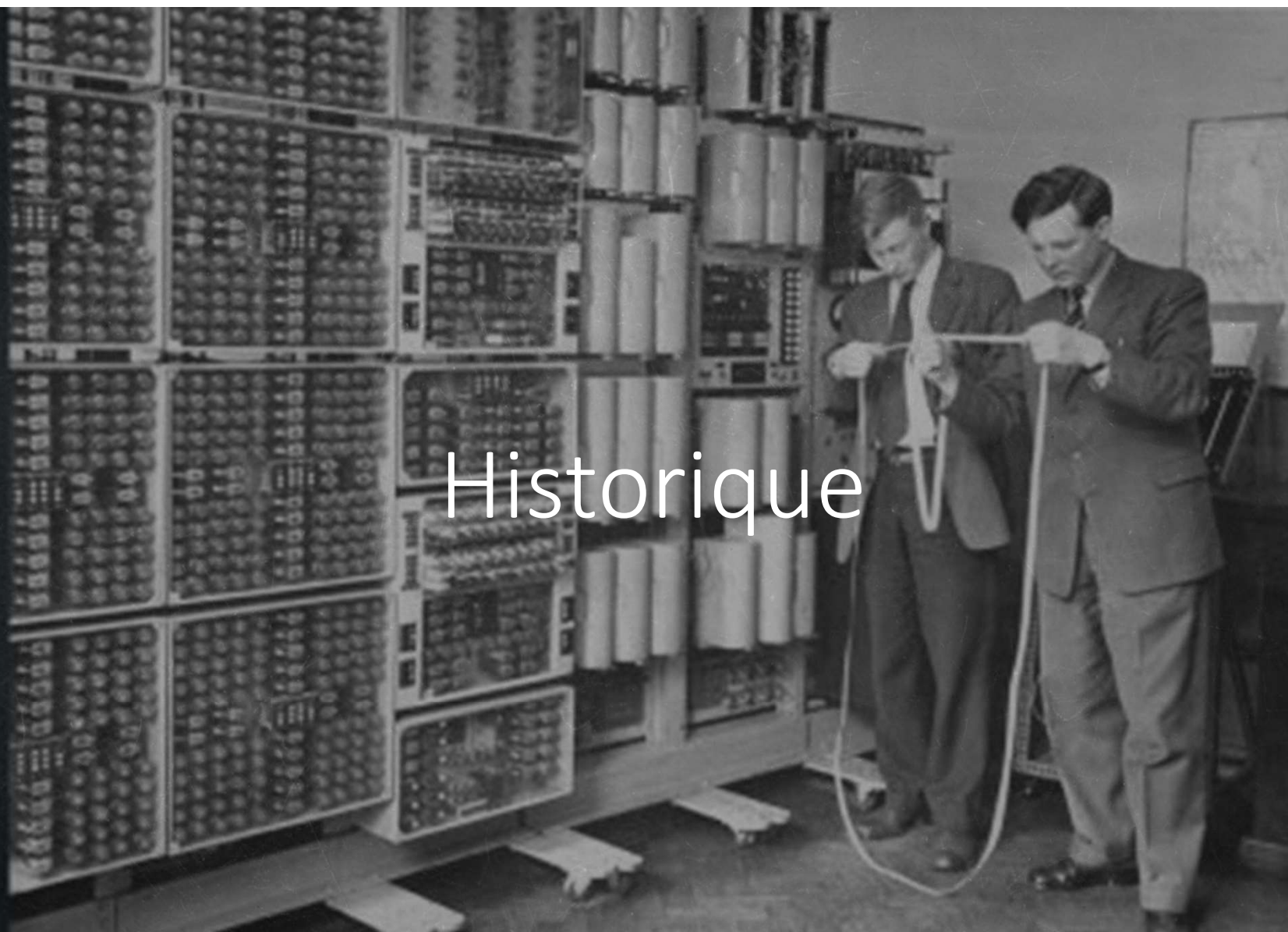
# Programmation parallèle: multithreading et multiprocessing

---

Par Jérémie Gince  
et Gabriel Genest



# Historique



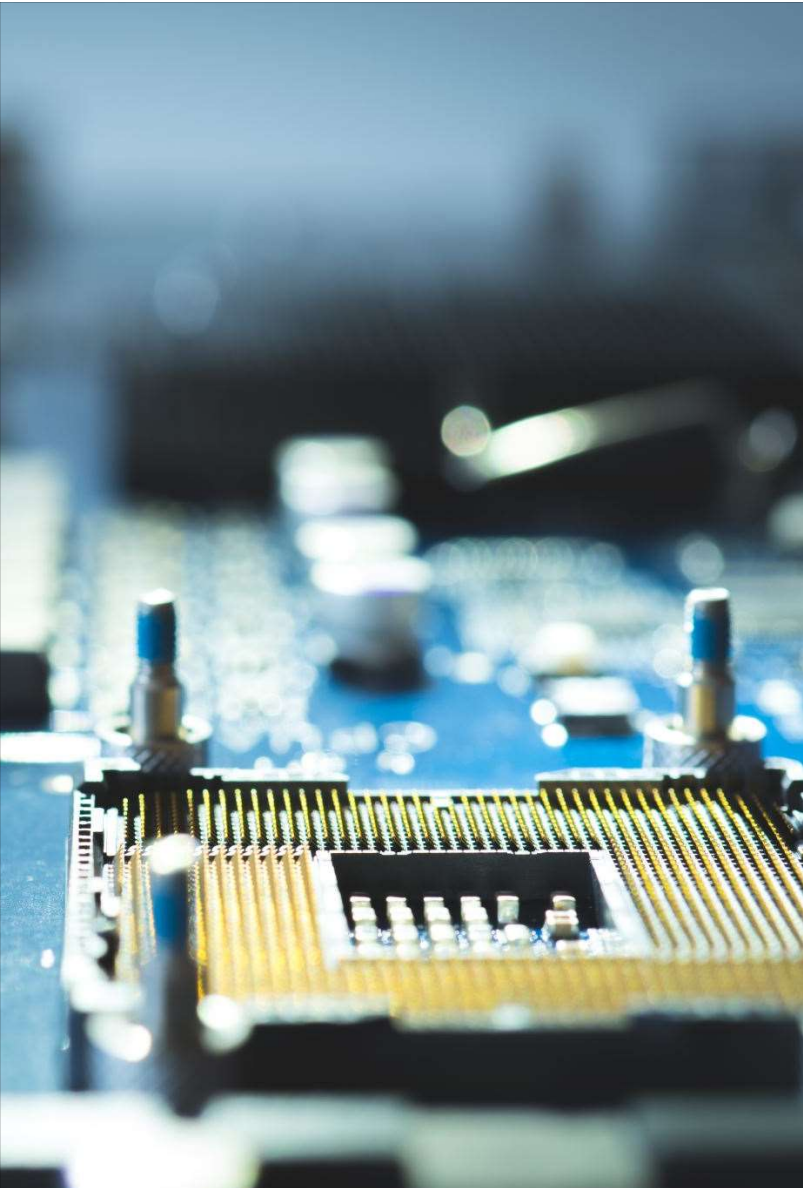


## À l'origine

---

- Enigma
- Ordinateurs étaient des pièces entières
- Très limités dans leur usage
  - Calculs de bases, pas de graphiques
  - Tube à vide, gros câblage électrique
    - Circuits électriques énormes
  - Pas beaucoup de mémoire
  - Peu de calculs par minute

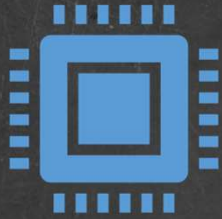




# Le terme CPU débarque

- Beaucoup de recherche se fait sur l'amélioration des ordinateurs
  - Développement de langages de programmation
  - Développement de hardware
    - Dont développement de circuits à transistors
- Arrivée des premiers CPUs vers la fin des années 50/60
  - Jusqu'à l'arrivée des circuits imprimés
- Premiers ordinateurs IBM en 1958

# Années 1950 - 1960



## Arrivées des “premiers” langages de programmation:

Avec notamment:

- FORTRAN (1957)
- ALGOL (1958)
- BASIC (1964)

Avant c'était sur des cartes trouées!



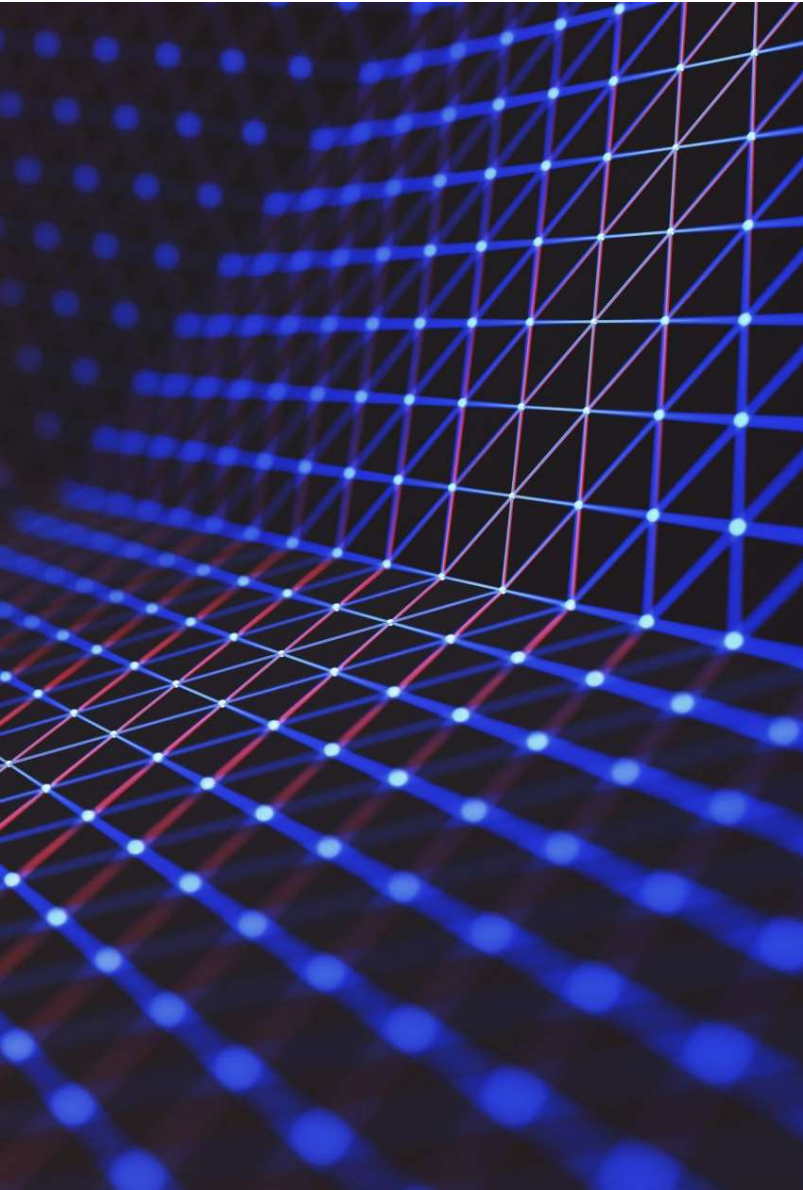
## Démocratisation des ordinateurs

Plus petits

Plus “abordables”

Mais: utilisation reste limitée





# Mais où est le multithreading? Le multiprocessing?

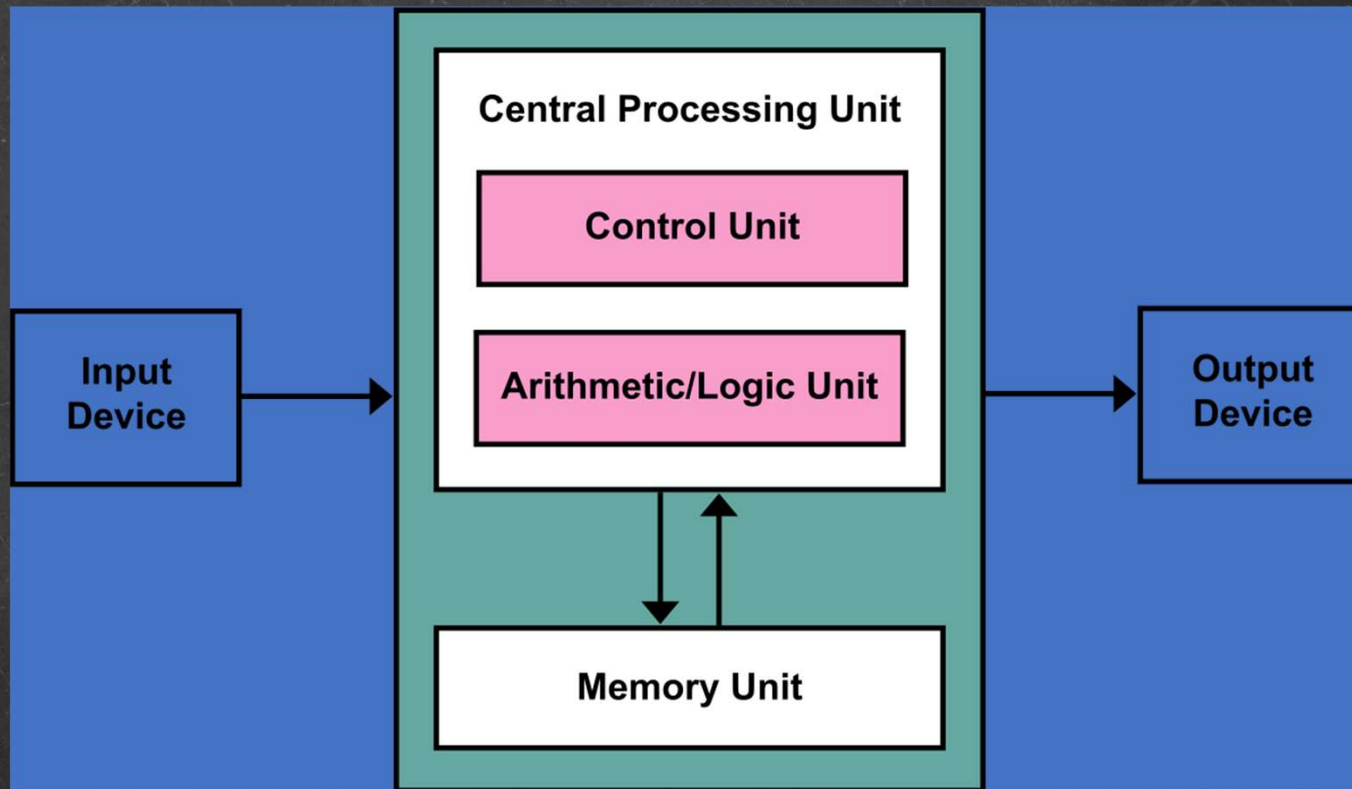
## Les threads:

- Depuis les années 50 et 60!

## Les multiprocesses:

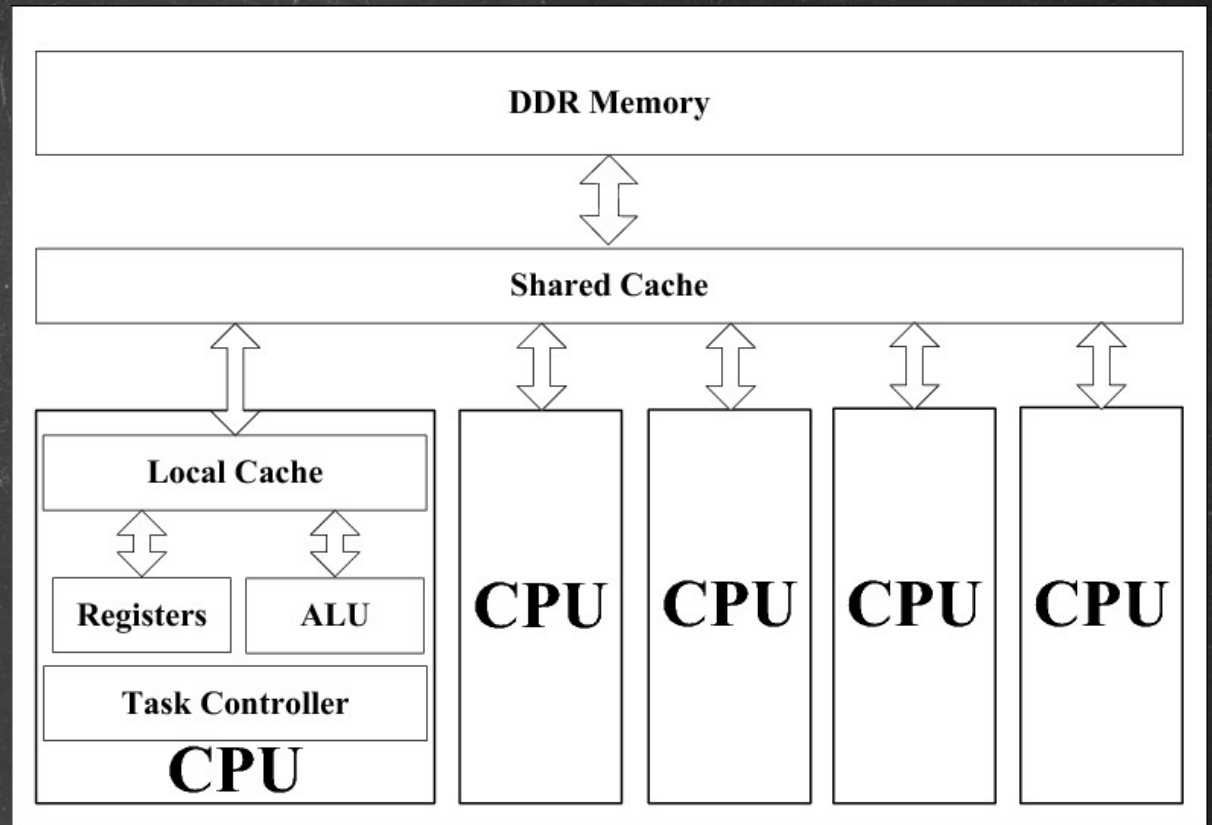
- Plus tard:
  - On doit avoir plus d'un CPU!

# Architecture Von Neumann



# Architecture Von Neumann

- Multiples CPUs *indépendants*
- On peut travailler en parallèle





# Qu'est-ce que le multithreading?

- Réfère à l'utilisation de plusieurs threads
- Mais qu'est-ce qu'un thread?
  - Objet abstrait virtuel qui vit dans le CPU
  - C'est une chaîne d'instructions dont le CPU a besoin
  - Un seul thread à la fois!
    - Contrairement à ce qu'on pense, le multithreading n'est pas parallèle
  - "C'est comme un bloc de code qu'on donne à l'ordinateur. On lui alloue du temps. S'il fini dans le temps, tant mieux. Sinon, on passe à un autre et on lui revient après."
    - A ghost physicist

# Qu'est-ce que le multiprocessing?

- Réfère à l'utilisation de plusieurs processes (processus)
- Qu'est-ce qu'un process?
  - Ensemble d'instructions exécutée par le CPU.
    - Ça ressemble à un thread non?
      - En fait: un thread vit dans un process
  - Uniquement un process à la fois peut être utilisé par le CPU\*
    - Si on a 4 coeurs (4 CPU indépendants), 4 processes peuvent rouler en même temps.

\* Attention: coeur logique

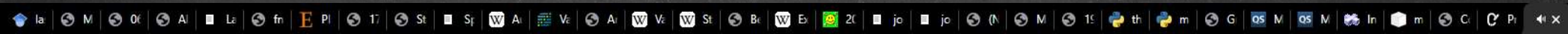


# Exemple concret d'un thread

- Vous êtes en train de faire votre devoir
  - C'est une tâche effectuée par un thread en quelque sorte.
- Vous avez soudainement soif
  - Un autre thread vient de s'ajouter
- Vous laissez votre devoir en suspens
  - Fin du temps d'allocation au thread de devoir
- Vous buvez de la bière
  - Allocation de temps au thread de boire
- Vous retournez à votre devoir
  - Vous retournez au thread de devoir, allouer du nouveau temps jusqu'à ce qu'il y ait un nouvel ajout de thread.

# Exemples de threads dans un ordinateur

- Mettre des memes sur Facebook en écoutant de la musique sur Youtube
- Les 50 tabs de chrome d'ouverts



- “Ça se fait tellement vite qu’on dirait que tout se fait en même temps!”\*\*
  - A ghost physicist
  - \*\*Attention: les ordinateurs courants ont plus de un CPU, avec multiples threads donc il se peut que ça soit vraiment en même temps.



# Exemple concret de plusieurs processus

- Vous avez (normalement) quatre membres (2 bras et 2 jambes)
- Chaque membre peut être bougé indépendamment
- C'est comme si vous aviez 4 processus (4 cœurs)
- Lien avec les threads:
  - Vous ne pouvez faire qu'une chose à la fois avec chaque membre
  - Vous pouvez garder en mémoire quoi faire
  - Le « processus » s'occupera de faire les mouvements un à la fois, en accordant du temps

# Thread dans le monde scientifique

- Téléchargement et téléversement de données
  - Bases de données
  - Web
  - Ex: Se connecter à une base de données et y mettre de l'information
- Écriture / lecture de fichiers
  - Ex: ouvrir un fichier et y inscrire de l'information

Multithreading très utile pour ce qui est I/O (peu de CPU demandé)



# Lock

- Objet virtuel -> Serrure de toilettes
- Utilisation en parallèle d'objets I/O
- Évite la corruption d'objets



# Multiprocessing en science

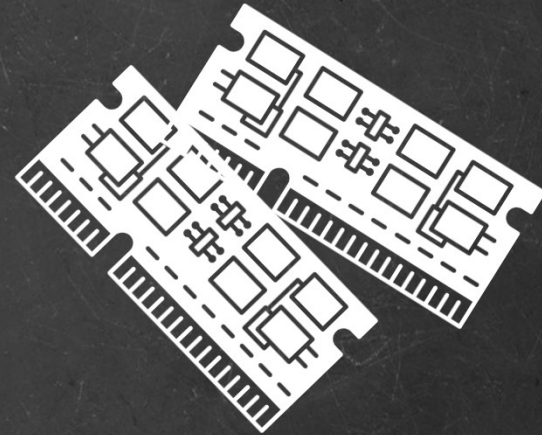
- Accélérer le traitement de données
  - ML, Big Data
  - Ex: Preprocessing, Recherche de données
- Évite surcharge d'un CPU
  - Développement logiciel
  - Ex: Minimiser instructions/CPU
- Optimisation
  - Simulations, Monte Carlo
  - Ex: Diffusion de photons

Multiprocessing très utile pour ce qui demande du CPU



# Mémoire partagée

- Mémoire partagé de facto en multithreading
- Permet de partager certains objets à travers plusieurs processus
- Évite la duplication d'objets
- Évite la nécessité de joindre des objets



# Avantages/Inconvénients Thread

## Avantages

- Mémoire partagée de facto
- Semble parallèle
- Déroulement concurrentiel sur un seul CPU possible
- Moins lourd en mémoire

## Inconvénients

- Pas vraiment parallèle
- Un seul thread à la fois / cœur

# Avantages/inconvénients Multiprocessing

## Avantages

- Vrai\* parallélisme
- Un processus peut avoir plusieurs threads
- Performances

## Inconvénients

- Pas de mémoire partagée de facto
- Limité au nombre de cœurs *logiques*
- Plus lourd en mémoire qu'un thread


\* Il peut y avoir ambiguïté, surtout en *Python*





## Attention!

- AMD ryzen 9 --> 12 cores et 24 threads ?
- 24 threads maximum? NON!
  - Pas même type de thread (ici, logical cores)



# Multiprocessing et multithreading en Python

# Comment créer des threads?

- On utilise le module `threading`
- On crée un thread comme n'importe quel autre objet:
  - `thread = threading.Thread(...)`
  - Arguments importants:
    1. `target` : fonction / code à effectuer par le thread
    2. `args` : arguments de la fonction à effectuer (optionnel)
- On démarre un thread avec la méthode `start`
  - `thread.start()`
- On conclue l'exécution d'un thread avec `join`
  - `thread.join()`
  - On peut mettre un timeout à `join` de sorte qu'il termine après un certain temps



# Exemple

```
import threading as th

def fonction(thread_id):
    print(f"Hello from thread {thread_id}")

thread1 = th.Thread(target=fonction, args=(0,))
thread2 = th.Thread(target=fonction, args=(1,))
thread1.start()
thread2.start()
thread1.join() # join permet de bloquer ce qui suit
thread2.join() # Sinon, le programme arrêterait*
```

\*: Le programme arrêterait s'il ne reste que des threads daemon

# Utiliser les même trucs: mettre un lock

```
import threading as th, time

counter = 0

def increm(quantite, lock):
    lock.acquire() # On prend le lock et on l'active
    global counter
    local_counter = counter
    local_counter += quantite
    time.sleep(0.1)
    counter = local_counter
    print(f"counter = {counter}")
    lock.release() # Ne pas oublier de le désactiver et le relâcher
```

(suite)

<...>

```
lock = th.Lock()
thread1 = th.Thread(target=increment, args=(10, lock))
thread2 = th.Thread(target=increment, args=(20, lock))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```



(suite)

- Avec le lock, la sortie sera toujours:

```
counter = 10  
counter = 30
```

- Sans le lock, on aurait possiblement:

```
counter = 20  
counter = 10  
ou  
counter = 10  
counter = 20  
ou (improbable)  
counter = 10  
counter = 30
```

# Utiliser les même trucs: mettre un lock

```
import threading as th, time

counter = 0

def increm(quantite, lock):
    lock.acquire() # On prend le lock et on l'active
    global counter
    local_counter = counter
    local_counter += quantite
    time.sleep(0.1)
    counter = local_counter
    print(f"counter = {counter}")
    lock.release() # Ne pas oublier de le désactiver et le relâcher
```

# Comment créer des processus?

- On importe `multiprocessing`
- Le reste est copié de `threading`
  - On crée un objet `Process` avec `target` et `args`
  - On le démarre avec `start`
  - On le joint avec `join`
- On peut terminer un process avec `terminate`
- À la fin (après `join`), on peut le fermer avec `close`



# Exemple

```
import multiprocessing as mp

def fonction(process_id):
    print(f"Hello from process {process_id}")

if __name__ == "__main__":
    # Très important pour des raisons techniques
    process1 = mp.Process(target=fonction, args=(0,))
    process2 = mp.Process(target=fonction, args=(1,))
    process1.start()
    process2.start()
    process1.join()
    process2.join()
    process1.close()
    process2.close()
```

# Lock de multiprocessing

- Il existe une variante de lock qui vient de multiprocessing
- Permet de verrouiller l'accès aux ressources partagées des divers processus
- Si threading seulement:
  - Utiliser `threading.Lock()`
- Si multiprocessing seulement:
  - Utiliser `multiprocessing.Lock()`
- Si mélange des deux:
  - Dépend du contexte

# Faire plusieurs processus, c'est long

- multiprocessing offre Pool
  - C'est un ensemble de processus
  - Utile lorsqu'on veut faire la même tâche sur plusieurs processus
    - En fait, ne fonctionne que pour un même target
  - start, join, close, etc. se fait automatiquement
  - Utiliser avec le *with statement*, super facile
  - Offre plusieurs méthodes pour distribuer le travail (versions synchrone ou asynchrone)
    - map
    - starmap
    - apply



# Exemple

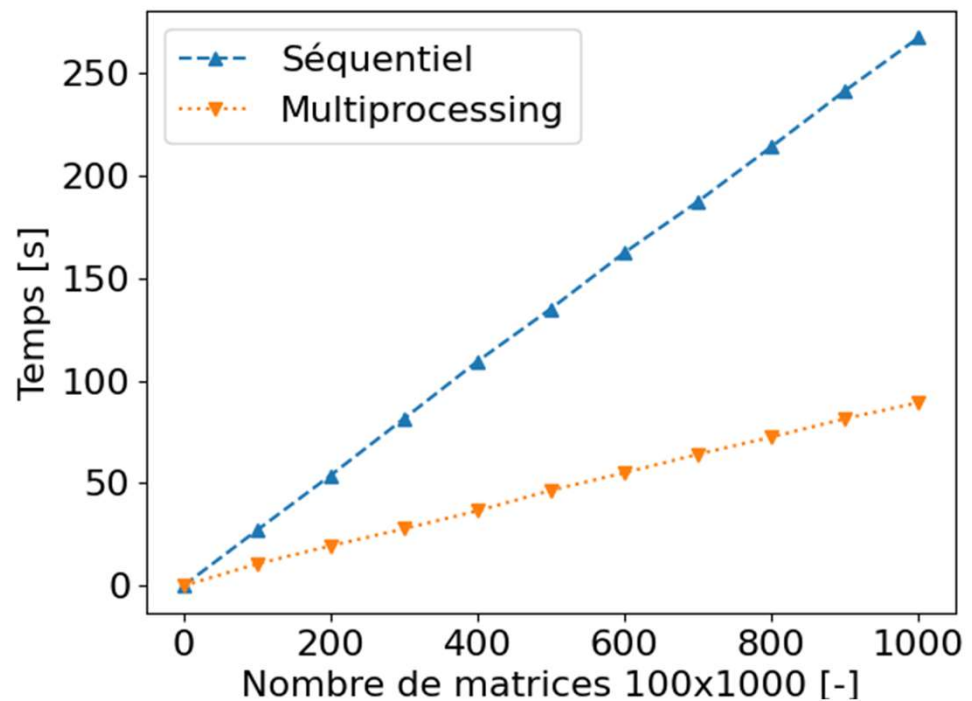
```
import multiprocessing as mp
import numpy as np
import scipy.signal as sg

def convolve(matrice):
    kernel = np.ones((7, 7))
    return sg.convolve2d(matrice, kernel)

if __name__ == "__main__":
    matrices = [np.random.randint(0, 10, (1000, 1000)) for _ in range(100)]
    nbProcesses = 6
    with mp.Pool(nbProcesses) as pool:
        convolutions = pool.map(convolve, matrices)
```

# Benchmark multiprocessing vs séquentiel

- Méthode: corrélation de Pearson de matrices  $100 \times 1000$  (corrélation colonne à colonne)
- On génère des matrices aléatoirement.
- On utilise 4 processus



## Version *héritage*

- On peut hériter de `threading.Thread` ou `multiprocessing.Process`
- Nos instances de classe deviennent donc un thread ou un processus
- Permet de personnaliser les threads et les processus.
  - Les attributs et méthodes du thread ou processus sont selon nos besoins
  - On peut personnaliser comment on démarre le thread ou le processus



# Exemple pour le thread

```
from threading import Thread
import time

class MyThread(Thread):
    def __init__(self, thread_id: int):
        # On doit appeler l'init du parent
        super(MyThread, self).__init__()
        self.thread_id = thread_id
        self.kill = False

    def run(self):
        # On redéfinit run du parent. Tant qu'on ne tue pas le thread, on fait quelque chose
        while not self.kill:
            print(f"{self.thread_id} est en vie!")
            time.sleep(2)

    def stop(self):
        self.kill = True
```

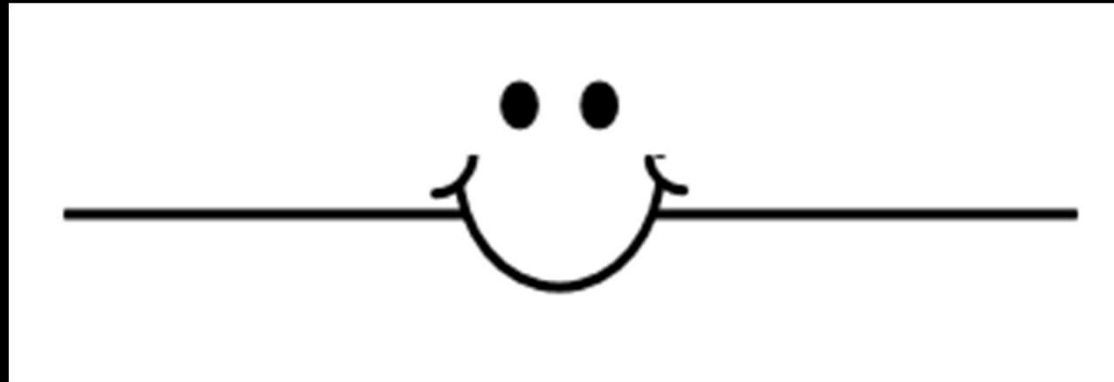
(suite)

```
<...>
if __name__ == "__main__":
    t = MyThread(0)
    t2 = MyThread(1)
    t.start()
    t2.start()
    # NE PAS JOIN LES THREADS! SINON, ON NE PEUT LES ARRÊTER
    <...>
    t.stop()
    t2.stop()
```

# Conclusion

- Utiliser le multithreading pour surtout ce qui est écrire/lire des fichiers (ou le web, bases de données)
- Utiliser le multiprocessing pour ce qui requiert pas mal de CPU
- Python offre les deux alternatives
- Maintenant, le ciel est la limite!
- Des questions?





Merci  
Bon ProgFest



# Meilleures performances « à tous les deux ans »

- Loi de Moore:
  - Le nombre de transistors double environ à tous les deux ans
  - Augmentation des performances générales
  - Des processeurs de plus en plus petits

