Fiche Ressource n°1 SAE22

Notion de thread, d'assignement et de copie superficielle

---

1. Notion de thread

Un **programme** est un fichier texte de code qui est compilé sous forme binaire (1 et 0) afin de s'exécuter sur l'ordinateur. Un autre type de programme est appelé « interprété », et au lieu d'être compilé à l'avance pour être exécuté, il est interprété en code exécutable au moment de son exécution.

Un **processus** est un programme qui a été chargé en mémoire avec toute la mémoire et les diverses ressources du système d'exploitation dont il a besoin pour fonctionner. Le système d'exploitation gère la tâche de gestion des ressources nécessaires pour transformer votre programme en un processus en cours d'exécution.

Les ressources essentielles dont chaque processus a besoin sont des **registres « Register »**, un **compteur de programme** « **counter** », une **pile** « **stack** » et un tas « **heap** ».

Les **registres** sont des lieux de stockage de données qui font partie du processeur de l'ordinateur (CPU). Un registre peut contenir une instruction, une adresse de stockage ou tout autre type de données nécessaires au processus.

Le **compteur de programme**, également appelé **pointeur d'instruction**, garde une trace de l'endroit où se trouve un ordinateur dans sa séquence du programme.

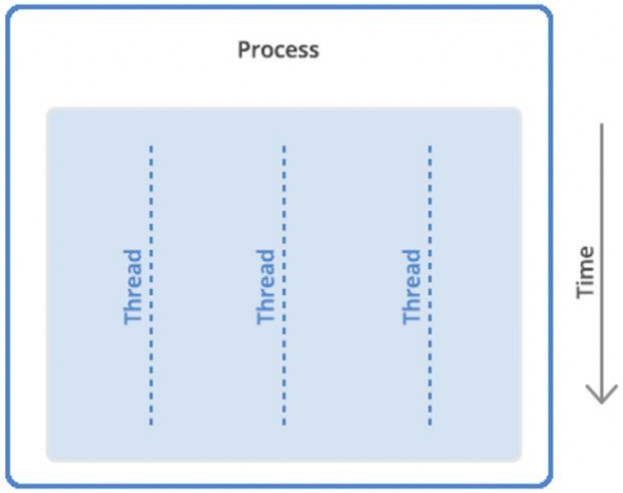
La **pile** fonctionne comme la structure de données du même nom, c'est-à-dire selon le principe du LIFO (last in, first out) ou dernier arrivé, premier sorti. Lors de l'exécution d'un programme, chaque appel de fonction implique la création d'un nouveau bloc au

sommet de la pile, correspondant à la fonction en question. L'intérêt est que cela permet de connaitre l'endroit où chaque fonction active (dont l'exécution n'est pas terminée) doit retourner à la fin de son exécution.

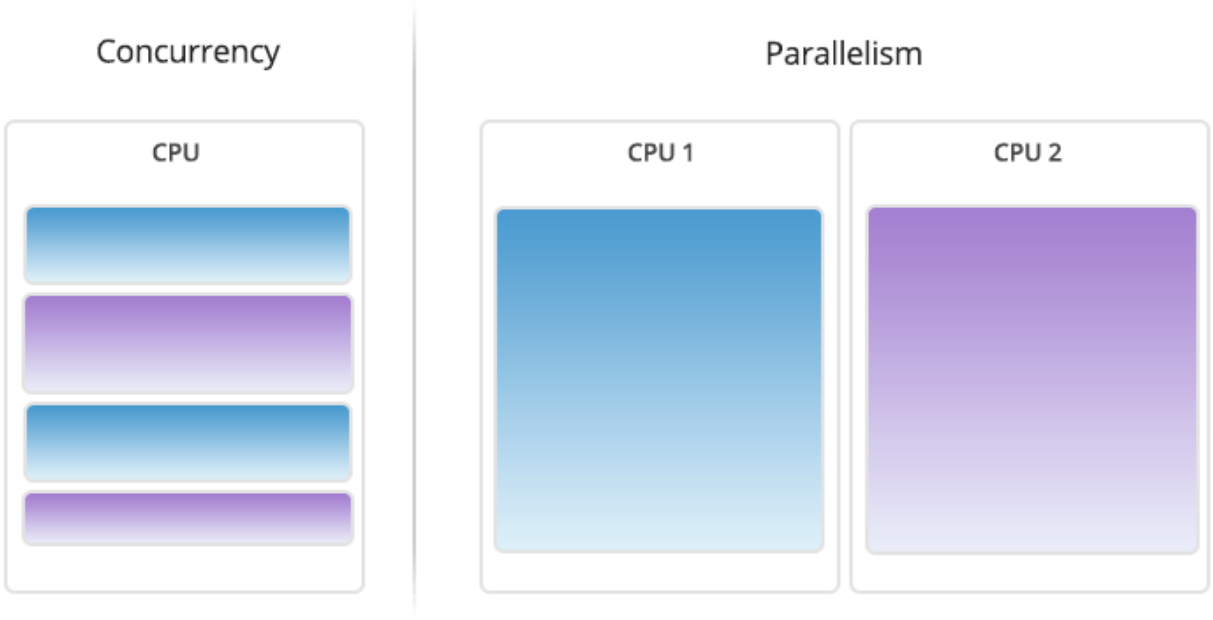
La pile présente un avantage considérable : l'accès est plus rapide (que le tas, voir section suivante) et l'allocation et désallocations de mémoire est géré par le processeur.

Le **tas** « **heap** » est un emplacement de la mémoire utilisé pour les allocations dynamiques, par exemple pour les objets en programmation objet. Contrairement à la pile, n'importe quel bloc de cet espace peut être alloué ou libéré à n'importe quel moment. L'avantage est que les variables créées sur le tas sont accessibles par n'importe quelle fonction et n'importe où dans le programme, rendant leur portée globale. Inversement la gestion du tas plus complexe et moins rapide car il est nécessaire de connaitre en permanence quel bloc est alloué.

Un **thread** est l'unité d'exécution d'un processus. Un processus peut avoir un seul ou plusieurs threads. Dans les processus multithreads, le processus accomplit plusieurs tâches en même temps ou presque.



Dans un fonctionnement **concurrent des threads**, on a un va-et-vient entre les tâches des différents threads. Ceci se produit si rapidement qu'il n'est généralement pas perceptible. Dans un fonctionnement en **parallélisme** (exécution simultanée authentique) les tâches sont exécutées en même temps sur différents processeurs.



2. Assignement vs copie superficielle

Quand on utilise l'opérateur = en Python par exemple pour copier une liste : y = x , on fait un assignement c'est à dire qu'on indique à la variable y de pointer vers la même zone mémoire que la variable x. Si on modifie y, on modifie donc la zone mémoire et donc x par la même occasion comme illustré ci-dessous :

```
Entrée [17]:
1 x=[1,2,3,4,5]
2 y=x
3 y.append(3)
4 print(x, y)
```

```
[1, 2, 3, 4, 5, 3] [1, 2, 3, 4, 5, 3]
```

On peut voir l'ID de l'adresse mémoire utilisée par chaque liste et vérifier que c'est bien la même :

```
Entrée [18]:
1 print(f"L'ID de l'adresse mémoire de la liste x est {id(x)}")
2 print(f"L'ID de l'adresse mémoire de la liste y est {id(y)}")
```

```
L'ID de l'adresse mémoire de la liste x est 4784656576
L'ID de l'adresse mémoire de la liste y est 4784656576
```

Pour faire une copie en utilisant une nouvelle adresse mémoire, on doit utiliser une copie superficielle comme illustré ci-dessous :

Entrée [19]:

```
a=[1,2,3,4,5]
b=a[:]
# ou b=a.copy()
b.append(3)
print(a, b)
```

```
[1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 3]
```

On peut voir l'ID de l'adresse mémoire utilisée par chaque liste et vérifier que c'est bien une adresse différente :

Entrée [20]:

```
print(f"L'ID de l'adresse mémoire de la liste a est {id(a)}")
print(f"L'ID de l'adresse mémoire de la liste b est {id(b)}")
```

```
L'ID de l'adresse mémoire de la liste a est 4571803136
L'ID de l'adresse mémoire de la liste b est 4784755072
```