

Gestion des processus et des ressources

- Programmation concurrente -

J. Boucher

Lycée Pierre-Paul RIQUET, Terminale NSI

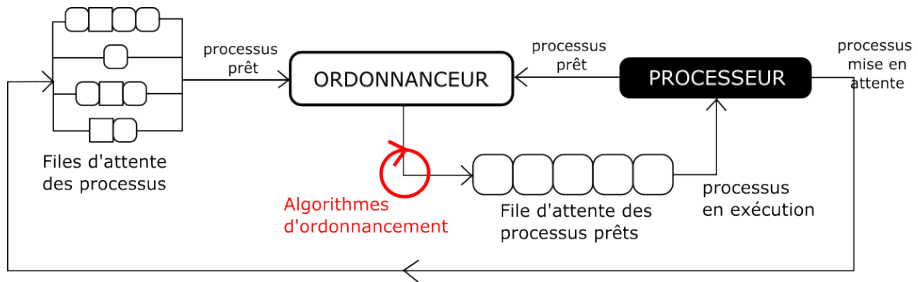
10 avril 2025

Plan

- 1 Ordonnanceur
- 2 **Programmation concurrente**
- 3 Bilan

1. Processus concurrents

L'exécution des processus s'effectue de manière concurrente, dans un ordre décidé par l'ordonnanceur selon différents **algorithmes d'ordonnancement** : « premier arrivé, premier servi », « le plus court d'abord », temps-partagé, etc.



Avantages :

- exécution d'un très grand nombre de programmes sur une machine monoprocesseur ;
- optimisation dynamique des ressources de la machine.

Par exemple, un processus en attente d'un événement peut être mis en pause, libérant le processeur pour d'autres processus. Cela maximisera le taux d'occupation du processeur. Inversement, si de nombreux processus sont en attente d'un événement, le système peut réduire la fréquence de fonctionnement du processeur.

Inconvénients : problèmes liés à la gestion des **ressources partagés** par plusieurs processus.

Problème de l'accès concurrent à une ressource

Le programme suivant requiert l'accès en écriture d'un fichier :

```
1 from os import getpid
2 pid = str( getpid() )
3 with open('test.txt', 'w') as fichier :
4     for i in range(1000):
5         |   fichier.write(pid + ' : ' + str(i) + '\n')
6         |   fichier.flush()
```

Question : Que se passe-t-il si on lance son exécution simultanément dans trois processus concurrents ?

```
$ python3 ecrit_fichier.py & ecrit_fichier.py & ecrit_fichier.py
```

82208

```
curseur : 9970
write('82208 : 840\n')
write('82208 : 841\n')
curseur : 9994
```

```
curseur : 9994
write('82208 : 842\n')
write('82208 : 843\n')
curseur : 10018
```

82209

```
curseur : 9970
write('82209 : 840\n')
write('82209 : 841\n')
write('82209 : 842\n')
curseur : 10006
```

```
curseur : 10006
write('82209 : 843\n')
curseur : 10018
```

82210

```
curseur : 9970
write('82210 : 840\n')
write('82210 : 841\n')
curseur : 9994
```

```
curseur : 9994
write('82210 : 842\n')
curseur : 10006
```

```
curseur : 10006
write('82210 : 843\n')
curseur : 10018
```

Contenu du fichier test.txt :

```
...  
82209 : 840  
82209 : 841  
82208 : 843  
...
```

Analyse :

- Chaque processus initialise dans sa mémoire la variable de position du caractère à écrire (ou curseur).
- Les différents processus peuvent donc écrire aux mêmes positions dans le fichier.
- La détermination de l'ordre d'exécution des processus par l'ordonnanceur peut être considérée comme aléatoire (en réalité : non déterministe).
→ si on réexécute de la même manière ces trois programmes, on obtiendra un fichier différent.

Comment éviter les problèmes d'accès concurrent à une ressource ?

- Un **verrou** est objet permettant de garantir l'**accès exclusif** d'une ressource à un processus.
- Le premier processus à faire la demande acquiert le verrou.
- Si le verrou est détenu par un autre processus, alors tout autre processus souhaitant l'obtenir est bloqué .
- Le verrou peut être rendu à tout moment par le processus qui l'a acquis.

En Python, on peut utiliser la bibliothèque **threading** à cet effet :

```
1 import threading
2 verrou = threading.Lock()
3 ...
4 verrou.acquire()
5 ...
6 verrou.release()
```

Selon le code de la route, quelle voiture doit passer en premier ?



Selon le code de la route, quelle voiture doit passer en premier ?



→ Cette situation s'appelle un **interblocage**.

2. Interblocage

Exemple dans un contexte informatique

On considère les deux programmes suivants :

- **enregister_micro** : acquiert la carte son en *accès exclusif* pour écrire les sons enregistrés sur la sortie standard pendant 10 secondes.
- **jouer_son** : acquiert la carte son en *accès exclusif* et joue le contenu qu'il reçoit sur son entrée standard.

→ **Les cartes sons sont des périphériques ne pouvant être utilisés que par au plus un processus.**

On peut les exécuter séquentiellement de la façon suivante :

```
1 $ enregister_micro > message.mp3
2 $ cat message.mp3 | jouer_son
```

Que se passe-t-il si on redirige la sortie de l'un sur l'entrée de l'autre ?

```
1 $ enregistrer_micro | jouer_son
```

Que se passe-t-il si on redirige la sortie de l'un sur l'entrée de l'autre ?

```
1 $ enregistrer_micro | jouer_son
```

- 1 **enregistrer_micro** ouvre la carte son et commence à envoyer les données sur la sortie standard.
- 2 **jouer_son** tente alors d'ouvrir la carte son et se retrouve **bloqué** ; il ne lit aucun octet sur son entrée standard.
- 3 **enregistrer_micro** est alors lui aussi **bloqué** lorsqu'il tente d'écrire sur sa sortie standard ; les octets non lus s'accumulent dans une mémoire tampon qui bloque le programme lorsqu'elle est pleine.

→ **Les deux processus sont en interblocage.**

L'interblocage est le « grand danger » de la programmation concurrente. Il existe quatre conditions nécessaires à la présence de l'interblocage (conditions de *Coffman*) :

- 1 **Exclusion mutuelle** : au moins une ressource du système doit être en accès exclusif.
- 2 **Rétention et attente** : un processus détient une ressource et demande une autre ressource détenue par un autre processus.
- 3 **Non préemption** : une ressource ne peut être rendue que par un processus qui la détient, sans être « préemptée » ou acquise de force par un autre processus.
- 4 **Attente circulaire** : l'ensemble des processus bloqués $P_1, \dots, P_n$ sont tels que P_1 attend une ressource tenue par $P_2, \dots, P_n$ attend une ressource détenue par P_1 .

Plan

1 Ordonnanceur

2 Programmation concurrente

3 Bilan

Bilan

À retenir :

- Les systèmes d'exploitation multitâches permettent d'exécuter de façon **concurrente** plusieurs programme.
- L'exécution d'un programme s'appelle un **processus**.
- C'est le système d'exploitation, en particulier **l'ordonnanceur**, qui détermine quel processus s'exécute à quel instant donné.
- Plusieurs processus s'exécutant de façon concurrente peuvent **s'interbloquer** s'ils attendent de pouvoir accéder à un même ensemble de **ressources en accès exclusif**.