Chapitre 2

Processus

2.1 Notion de processus

Problème. *Comment exécuter plusieurs programmes simultanément sur une machine possédant un seul processeur ?*

Dans un système d'exploitation multitâche comme Windows ou Linux, il semble que plusieurs programmes puissent s'éxécuter en même temps : on navigue sur les internet depuis un navigateur tout en écoutant de la musique, pendant que l'ordinateur se met à jour en installant des fichiers volumineux. Mais cette simultanéité n'est qu'une illusion.

En réalité, un processeur ne peut pas exécuter plus d'instructions simultanément qu'il n'a de coeurs.
Alors comment fait-il en pratique ?

2.1.1 Programmes et Processus

Il convient dans un premier temps de définir les notions de *programmes* et de *processus*.

Un **programme** est un ensemble d'instructions susceptibles d'être exécutées. On parle par exemple de *programme Python* pour désigner une suite d'instructions dans le langage du même nom. Tant qu'on ne l'exécute pas, un tel jeu d'instructions reste un programme.

Un **processus** est une instance d'un programme, chargée en mémoire et exécutée de manière continue ou discontinue par le système d'exploitation. Si on exécute plusieurs fois le même programme, alors il existera plusieurs processus correspondant à ce même programme.

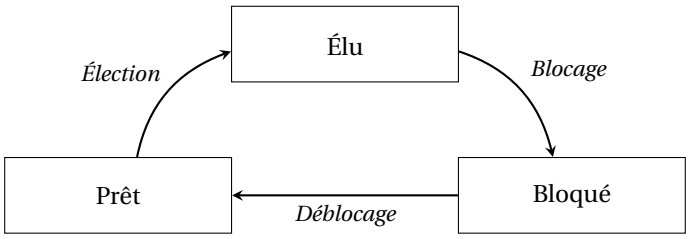
Exemple. On ouvre un fichier à l'aide d'un éditeur de texte. On modifie ce fichier, mais sans fermer l'éditeur de texte, on ouvre un second fichier avec ce même éditeur de texte. Bien que ce soit le même programme, il existera alors deux processus distincts, un pour chaque exécution du programme éditeur de texte.

2.1.2 Différents états d'un processus

Tout processus peut être dans un des 3 états suivants :

- **élu** : on dit qu'un processus est élu s'il est en train de s'exécuter (et utilise donc le processeur)
- **bloqué** : si un processus élu demande l'accès à une ressource non disponible (par exemple une imprimante qui est déjà utilisée), il ne peut alors pas poursuivre son exécution. En attendant d'accéder à cette ressource, il passe de l'état *élu* à l'état *bloqué*
- **prêt** : lorsque la ressource est enfin disponible, le processus à l'état bloqué est prêt à reprendre son exécution. Mais entre temps, le processeur a été sollicité par un autre processus, qui a pris la place du premier. Il faut alors attendre « que la place se libère »

Chaque changement d'état possède un nom : **élection** pour le passage de *prêt* à *élu*, **blocage** pour le passage de *élu* à *bloqué*, et **déblocage** pour le passage de *bloqué* à *prêt*.



Il est vraiment important de bien comprendre que le « chef d'orchestre» qui attribue aux processus leur état est le système d'exploitation : on dit qu'il gère l'**ordonnancement des processus**, et définit ainsi quel processus sera le plus prioritaire.
Autre point important : un processus qui utilise une ressource doit la libérer une fois qu'il a fini de l'utiliser, afin de la rendre disponible pour les autres processus. Pour libérer une ressource, un processus doit obligatoirement être dans un état *élu*.

2.1.3 Processus en cours d'exécution

Sous Linux, on peut accéder à la liste des processus en cours d'exécution grâce à la commande `ps -ef` :

```
thomas@thomas-Latitude-E7450: ~
thomas@thomas-Latitude-E7450:~$ ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 mars26 ?        00:00:32 /sbin/init splash
root           2     0  0 mars26 ?        00:00:00 [kthreadd]
root           4     2  0 mars26 ?        00:00:00 [kworker/0:0H]
root           6     2  0 mars26 ?        00:00:00 [mm_percpu_wq]
root           7     2  0 mars26 ?        00:00:01 [ksoftirqd/0]
root           8     2  0 mars26 ?        00:03:19 [rcu_sched]
root           9     2  0 mars26 ?        00:00:00 [rcu_bh]
root          10     2  0 mars26 ?        00:00:00 [migration/0]
root          11     2  0 mars26 ?        00:00:00 [watchdog/0]
root          12     2  0 mars26 ?        00:00:00 [cpuhp/0]
root          13     2  0 mars26 ?        00:00:00 [cpuhp/1]
root          14     2  0 mars26 ?        00:00:00 [watchdog/1]
root          15     2  0 mars26 ?        00:00:00 [migration/1]
root          16     2  0 mars26 ?        00:00:19 [ksoftirqd/1]
root          18     2  0 mars26 ?        00:00:00 [kworker/1:0H]
root          19     2  0 mars26 ?        00:00:00 [cpuhp/2]
root          20     2  0 mars26 ?        00:00:00 [watchdog/2]
root          21     2  0 mars26 ?        00:00:00 [migration/2]
root          22     2  0 mars26 ?        00:00:05 [ksoftirqd/2]
root          24     2  0 mars26 ?        00:00:00 [kworker/2:0H]
root          25     2  0 mars26 ?        00:00:00 [cpuhp/3]
root          26     2  0 mars26 ?        00:00:00 [watchdog/3]
```

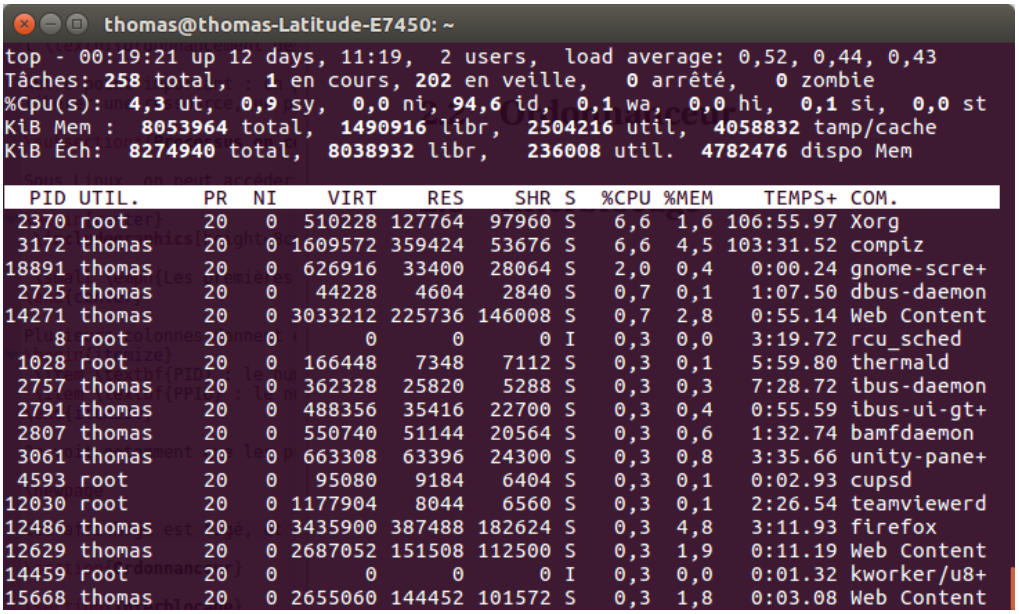
Les premières lignes retournées par la commande `ps -ef`

Plusieurs colonnes donnent des informations importantes :

- **PID** : le numéro d'identification du processus
- **PPID** : le numéro du processus parent, qui a créé le processus courant

On voit notamment que les processus 1 et 2 ont pour processus parent le processus 0 : c'est le tout premier processus lancé à partir de rien par le système d'exploitation.

Cet affichage est figé, et donne l'ordre des processus dans leur ordre d'exécution. Pour connaître les processus les plus utilisés en temps réel, on utilise la commande `top` :



Résultat de la commande `top`

Il est possible de « tuer » un processus, c'est à dire d'y mettre un terme, avec la commande `kill` suivie du PID du processus.

2.2 Ordonnanceur

L'ordonnanceur est le module du système d'exploitation qui se charge d'organiser l'exécution des processus à tour de rôle. Il a la délicate mission d'octroyer un peu de temps d'exécution processeur (on parle de *quantum de temps*) à chacun des processus, en gérant les priorités, les blocages, les élections...

Le partage du processeur se fait sur tous les processus en cours d'exécution, y compris l'ordonnanceur lui-même ! Comme toujours, il existe de nombreux algorithmes permettant de répartir « équitablement » le temps processeur :

- *First-come, first-served (FCFS) : premier arrivé, premier servi*

  Les processus sont stockés dans une file. Le premier arrivé est admis immédiatement et s'exécute tant qu'il n'est pas bloqué ou terminé. Lorsqu'il se bloque, le processus suivant s'exécute, et le processus bloqué va se placer au bout de la file. C'est un algorithme simple, mais favorisant les processus longs.

- *Shortest Job First (SJF) : priorité au plus court*

  Le processus dont le temps de traitement sera supposé le plus court est prioritaire. Inconvénient : si de multiples processus courts arrivent sans cesse, les plus longs ne sont jamais exécutés.

  ☞ *C'est le fonctionnement rencontré à une caisse de supermarché : un client avec 1 article arrive après un client avec 100 articles. En général, le client aux 100 articles laisse passer celui n'achetant qu'un seul article.*

- *Round Robin (RR) : algorithme du tourniquet*

  Chaque processus est exécuté pendant un certain quantum de temps, à tour de rôle, dans l'ordre d'arrivée.

  Pas de jaloux !

☞ *Il existe de nombreux autres algorithmes, mais nous ne verrons que ces 3 principaux*

Exemple. 3 processus non concurrents (ne pouvant pas se bloquer entre eux) P1, P2, P3 sont dans une file d'attente dans cet ordre (P1 est le premier, P3 est le dernier). Leur exécution demande un temps total de service exprimé en unités arbitraires :

Processus	P1	P2	P3
Temps de service estimé	6	2	4

Voyons comment chaque algorithme va répartir le temps processeur entre ces différents processus :

- *FCFS (premier arrivé, premier servi)*

  Comme P1 arrive en premier, il est exécuté en premier, pendant 6 unités de temps. Puis P2 est exécuté, pendant 2 unités de temps, et enfin P3 est exécuté.

  Le temps d'attente pour l'exécution de chaque programme est de 6 unités de temps pour P1, de 8 unités de temps pour P2 (depuis le début) et de 12 unités de temps pour P3, soit un temps d'attente moyen de $\frac{6+8+12}{3} \simeq 8,67$ unités de temps.

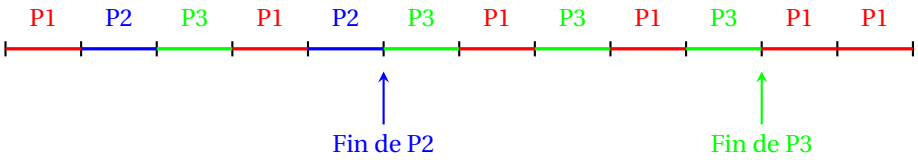
- *SJF (prorité au plus court)*

  Comme P2 est le programme le plus court, il est exécuté en premier, suivi (dans cet ordre) de P3 et P1 :

  Le temps d'attente pour l'exécution de chaque programme est de 2 unités de temps pour P2, de 6 unités de temps pour P3 et de 12 unités de temps pour P1, soit un temps d'attente moyen de $\frac{2+6+12}{3} \simeq 6,67$ unités de temps.

- *RR (le tourniquet)*

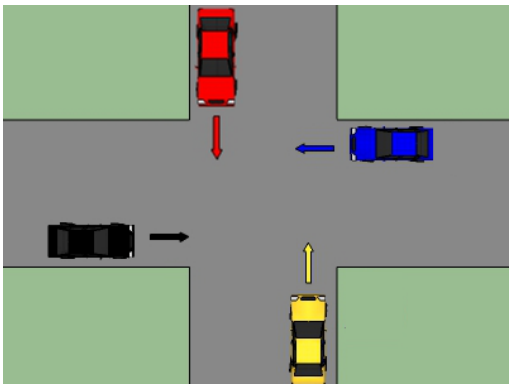
  Chaque processus est exécuté avec un quantum de 1, tour à tour, jusqu'à ce que chacun se termine :



  Le temps d'attente pour l'exécution de chaque programme est de 5 unités de temps pour P2, de 10 unités de temps pour P3 et de 12 unités de temps pour P1, soit un temps d'attente moyen de $\frac{5+10+12}{3} = 9$ unités de temps.

2.3 Interblocage

Imaginons la situation suivante : 4 voitures arrivent à une intersection. La règle est de laisser la priorité à droite. Quelle voiture passe en premier ?



Nous sommes face à une situation d'**interblocage** : tout le monde a la priorité, mais personne ne l'a !
Cette situation peut survenir entre plusieurs processus. Par exemple, imaginons 3 processus P1,P2,P3 qui tentent d'accéder à 3 ressources différentes :

- P1 souhaite accède à la carte audio, puis à l'écran. Il « relâche » alors l'écran puis la carte audio, et se termine.
- P2 souhaite accède à l'écran, puis à l'imprimante. Il relâche l'imprimante et l'écran puis se termine.
- P3 souhaite accède à l'imprimante, puis à la carte audio. Il relâche la carte audio et l'imprimante puis se termine.

Dans un fonctionnement en tourniquet, on aurait la situation suivante :

- P1 accède à la carte audio
- P2 accède à l'écran
- P3 accède à l'imprimante
- P1 tente d'accéder à l'écran, mais P2 y accède déjà : P1 devient bloqué
- P2 tente d'accéder à l'imprimante, mais P3 y accède déjà : P2 devient bloqué
- P3 tente d'accéder à la carte audio, mais P1 y accède déjà : P3 devient bloqué

Et c'est l'interblocage, ou *deadlock* en anglais.

L'interblocage est un vrai problème pour les systèmes d'exploitation, son traitement peut être préventif ou curatif, avec évidemment des avantages et des inconvénients.

Exercices

Exercice 01 Décrire l'exécution des processus et calculer le temps moyen d'attente dans le cadre des politiques d'ordonnancement FCFS (premier arrivé, premier servi), SJF (plus court d'abord) et RR (tourniquet) :

Processus	P1	P2	P3
Temps de service estimé	5	7	3

Exercice 02 Même exercice :

Processus	P1	P2	P3	P4	P5
Temps de service estimé	10	1	2	1	5

Exercice 03 Trois processus P1, P2, P3 ont été chargés sur un système informatique aux dates indiquées ci-dessous. Leur demande en durée de service est également indiquée (unités de temps arbitraires).

Processus	Date arrivée	Temps de service
P1	0	4
P2	0	2
P3	3	1

Comparer les algorithmes FCFS, SJF et RR en calculant le temps d'attente moyen d'exécution des différents processus.

Exercice 04 Même exercice :

Processus	Date arrivée	Temps de service
P1	0	3
P2	1	5
P3	3	2
P4	9	5
P5	12	5

Exercice 05 Écrire une fonction Python `ordonnancement(processus, algorithme)` qui retourne l'ordre d'exécution de plusieurs processus. Le paramètre `processus` est une liste de tuples `(nom_processus, temps_service)`, et `algorithme` est une chaîne de caractères.

```
>>> ordonnancement([("P1",6), ("P2",2), ("P3",4)], "FCFS")
'P1-P1-P1-P1-P1-P1-P2-P2-P3-P3-P3-P3'
>>> ordonnancement([("P1",6), ("P2",2), ("P3",4)], "SJF")
'P2-P2-P3-P3-P3-P3-P1-P1-P1-P1-P1-P1'
>>> ordonnancement([("P1",6), ("P2",2), ("P3",4)], "RR")
'P1-P2-P3-P1-P2-P3-P1-P3-P1-P3-P1-P1'
```

Exercice 06 Écrire une fonction Python `temps_attente(processus, algorithme)` qui retourne le temps d'attente moyen d'exécution de plusieurs processus. Le paramètre `processus` est une liste de tuples `(nom_processus, temps_service)`, et `algorithme` est une chaîne de caractères.

```
>>> temps_attente([("P1",6), ("P2",2), ("P3",4)], "FCFS")
8.666666666666666
>>> temps_attente([("P1",6), ("P2",2), ("P3",4)], "SJF")
6.666666666666667
>>> temps_attente([("P1",6), ("P2",2), ("P3",4)], "RR")
9
```