Chapitre 5 - Gestion des processus et des ressources par un système d'exploitation

I - Introduction

Dans les années 1970 les ordinateurs personnels étaient incapables d'exécuter plusieurs tâches à la fois : il fallait attendre qu'un programme lancé se termine pour en exécuter un autre.

Les systèmes d'exploitations récents (GNU/Linux, macOS, iOS, Android, Windows...) permettent d'exécuter des tâches "simultanément". En effet, la plupart du temps, lorsque l'on utilise un ordinateur, plusieurs programmes sont exécutés "en même temps" : par exemple, on peut très bien ouvrir simultanément un navigateur Web, un traitement de texte, un IDE Python, un logiciel de musique (sans parler de tous les programmes exécutés en arrière-plan) ...

Ces programmes en cours d'exécution s'appellent des **processus**. Une des tâches du système d'exploitation est d'allouer à chacun des processus les ressources dont il a besoin en termes de mémoire, entrées-sorties ou temps d'accès au processeur, et de s'assurer que les processus ne se gênent pas les uns les autres.
Pourtant, on rappelle qu'un programme n'est qu'une suite d'instructions machine exécutées l'une après l'autre par le processeur (cf. cours de Première sur le Modèle d'architecture d'un ordinateur) et qu'un processeur n'est capable d'exécuter qu'une seule instruction à la fois.
Comment est-il alors possible que plusieurs programmes soient exécutés en même temps ?

II - Processus

Qu'est-ce qu'un processus ?

Il ne faut pas confondre programme et processus :

Définition 1

Un **programme** est un fichier binaire (on dit aussi un exécutable) contenant des instructions machines que seul le processeur peut comprendre.

Définition 2

Un **processus** est un programme en cours d'exécution, autrement dit le phénomène dynamique lié à l'exécution d'un programme par l'ordinateur.

Ainsi, lorsque nous cliquons sur l'icône d'un programme (ou lorsque nous exécutons une instruction dans la console pour lancer un programme), nous provoquons la naissance d'un ou plusieurs processus liés au programme que nous lançons.

Un processus est donc une **instance d'un programme** auquel est associé :

- du code
- des données/variables manipulées
- des ressources : processeur, mémoire, périphériques d'entrée/sortie (voir paragraphe suivant)

Il n'est d'ailleurs pas rare qu'un même programme soit exécuté plusieurs fois sur une machine au même moment en occupant des espaces mémoires différents : par exemple deux documents ouverts avec un traitement de texte, ou trois consoles distinctes... qui correspondent à autant d'instances du même programme et donc à des processus différents.

Observer les processus

Il est très facile de voir les différents processus s'exécutant sur une machine.

Sous GNU/Linux, on peut utiliser la commande `ps` (comme **proces**, la traduction anglaise de processus) pour afficher les informations sur les processus. En passant des options à cette commande on peut obtenir des choses intéressantes.

Par exemple, en exécutant dans un terminal la commande `ps -aef`, on peut visualiser tous les processus en cours sur notre ordinateur :

```
mikael    109886    2144  0 jul16 ?        00:00:13 /usr/libexec/gvfsd-smb --spawner :1.26 /org/gtk/gvfs/exec_spaw/23
mikael    144879    1972  0 jul16 ?        00:00:03 /usr/libexec/gnome-terminal-server
mikael    144887  144879  0 jul16 pts/0    00:00:00 bash
mikael    289890    1972  0 jul18 ?        00:00:00 /usr/libexec/evinced
mikael    292151    1972  0 jul18 ?        00:01:07 evince /home/mikael/enseignement/2025_2026/Terminale_NSI/latex/C05_gestion_des_processus_et_ressources/NSI_gestion_des_processus_TD.pdf
mikael    292600   97472  0 jul18 ?        00:00:57 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
root      407845       2  0 jul19 ?        00:00:00 [irq/125-mei_me]
root      407894       2  0 jul19 ?        00:00:00 [kworker/0:2H-kblockd]
mikael    409372    1972  0 jul19 ?        00:00:17 evince /home/mikael/enseignement/2024_2025/Terminale_NSI/séquence/S05_gestion_des_processus_et_ressources/S05_NSI_Processus_A_SAVOIR.pdf
mikael    411197   97472  0 jul19 ?        00:01:33 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    461364   97472  0 jul20 ?        00:05:01 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    471931   97472  0 jul20 ?        00:05:00 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    471971   97472  0 jul20 ?        00:01:52 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    472313   97472  0 jul20 ?        00:00:14 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    481296   97472  0 jul20 ?        00:01:41 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    485663   22001  0 jul20 ?        00:00:00 /usr/bin/python3 -i /home/mikael/enseignement/2025_2026/Terminale_NSI/sequence/C05_gestion_des_processus_et_ressources/verrou.py
root      495964       1  0 00:00 ?        00:00:00 /usr/sbin/cupsd -l
root      495965       1  0 00:00 ?        00:00:00 /usr/sbin/cups-browsed
lp        495975  495964  0 00:00 ?        00:00:00 /usr/lib/cups/notifier/dbus dbus://
root      514844       2  0 08:24 ?        00:00:00 [kworker/u9:0-rb_allocator]
mikael    516193    1972  0 08:30 ?        00:00:09 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
root      524218       2  0 10:17 ?        00:00:00 [kworker/u9:1-i915_flip]
root      529784       2  0 12:40 ?        00:00:00 [kworker/u8:0-events_unbound]
root      530342       2  0 12:56 ?        00:00:00 [kworker/u8:1-events_unbound]
root      530554       2  0 13:04 ?        00:00:00 [kworker/0:1-events]
root      530609       2  0 13:06 ?        00:00:00 [kworker/3:2-events]
root      530722       2  0 13:10 ?        00:00:00 [kworker/u8:2-i915]
root      530739       2  0 13:11 ?        00:00:00 [kworker/2:0-mm_percpu_wq]
root      530778       2  0 13:12 ?        00:00:00 [kworker/1:1-events]
root      530820    1964  0 13:13 ?        00:00:00 sleep 3600
root      530947       2  0 13:16 ?        00:00:00 [kworker/0:2-events]
root      531122       2  0 13:18 ?        00:00:01 [kworker/1:0-events]
root      531254       2  0 13:18 ?        00:00:00 [kworker/2:2-events]
root      531431       2  0 13:21 ?        00:00:00 [kworker/3:1-events]
root      531510       2  0 13:22 ?        00:00:00 [kworker/0:0-events]
root      531563       2  0 13:23 ?        00:00:00 [kworker/u9:2]
root      531630       2  0 13:24 ?        00:00:00 [kworker/2:1-events]
mikael    531689   97472  0 13:24 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
root      531731       2  0 13:25 ?        00:00:00 [kworker/1:2-events]
mikael    531737   97472  0 13:25 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    531765   97472  0 13:25 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0 -prefsLen 40344 -prefMapHandle 1 -prefMapSize 267163 -jsInitHandle 2 -jsInitLen 2474
mikael    531817  144887  0 13:26 pts/0    00:00:00 ps -aef
mikael@mikael-HP-EliteBook-840-G3:~/athle$
```

Création d'un processus

Un processus peut être créé :

- au démarrage du système
- par un autre processus
- par une action d'un utilisateur (lancement d'un programme)

Sous GNU/Linux, un tout premier processus est créé au démarrage (c'est le processus 0 ou encore `Swapper`). Ce processus crée un processus souvent appelé `init` qui est le fils du processus 0. Ensuite, à partir de `init`, les autres processus nécessaires au fonctionnement du système sont créés. Ces processus créent ensuite eux-mêmes d'autres processus, etc.

Un processus peut créer un ou plusieurs processus, ce qui aboutit à une structure arborescente comme nous allons le voir maintenant.

PID et PPID

La commande précédente permet de voir que chaque processus est identifié par un numéro : son **PID** (pour *Process Identifier*). Ce numéro est donné à chaque processus par le système d'exploitation.

On constate également que chaque processus possède un **PPID** (pour *Parent Process Identifier*), il s'agit du PID du processus parent, c'est-à-dire celui qui a déclenché la création du processus. En effet, un processus peut créer lui même un ou plusieurs autres processus, appelés processus fils.

III - Gestion des processus et des ressources

Exécution concurrente

Les systèmes d'exploitation modernes sont capable d'exécuter plusieurs processus "en même temps". En réalité ces processus ne sont pas toujours exécutés "en même temps" mais plutôt "à tour de rôle". On parle d'exécution concurrente car les processus sont en concurrence pour obtenir l'accès au processeur chargé de les exécuter.

> **Note** : Sur un système multiprocesseur, il est possible d'exécuter de manière parallèle plusieurs processus, autant qu'il y a de processeurs. Mais sur un même processeur, un seul processus ne peut être exécuté à la fois.

On peut voir assez facilement cette exécution concurrente. Considérons les deux programmes Python suivants :

`progA.py`

```
import time

for i in range(100):
    print("programme A en cours, iteration", i)
    time.sleep(0.01)  # pour simuler un traitement avec des calculs
```

`progB.py`

```
import time

for i in range(100):
    print("programme B en cours, iteration", i)
    time.sleep(0.01)  # pour simuler un traitement avec des calculs
```

En ouvrant un Terminal, on peut lancer simultanément ces deux programmes avec la commande : `python3 progA.py & python3 progB.py &`.
Le caractère & qui suit une commande permet de lancer l'exécution en arrière plan et de rendre la main au terminal.

Le shell indique alors dans la console les PID des processus correspondant à l'exécution de ces deux programmes (ici 9154 et 9155) puis on constate grâce aux affichages que le système d'exploitation alloue le processeur aux deux programmes à *tour de rôle* :

```
mikael@mikael-HP-EliteBook-840-G3:~/enseignement/2025_2026/Terminale_NSI/sequence/C05_gestion_des_processus_et_ressources$ python3 progA.py & python3 progB.py &
[1] 533343
[2] 533344
mikael@mikael-HP-EliteBook-840-G3:~/enseignement/2025_2026/Terminale_NSI/sequence/C05_gestion_des_processus_et_ressources$ programme A en cours, itération 0
programme B en cours, itération 0
programme B en cours, itération 1
programme A en cours, itération 1
programme A en cours, itération 2
programme B en cours, itération 2
programme A en cours, itération 3
programme B en cours, itération 3
programme B en cours, itération 4
programme A en cours, itération 4
programme B en cours, itération 5
programme A en cours, itération 5
programme B en cours, itération 6
programme A en cours, itération 6
programme A en cours, itération 7
programme B en cours, itération 7
programme B en cours, itération 8
programme A en cours, itération 8
programme B en cours, itération 9
programme A en cours, itération 9
programme B en cours, itération 10
programme A en cours, itération 10
programme A en cours, itération 11
programme B en cours, itération 11
programme A en cours, itération 12
programme B en cours, itération 12
programme B en cours, itération 13
programme A en cours, itération 13
programme A en cours, itération 14
programme B en cours, itération 14
```

Accès concurrents aux ressources

Une **ressource** est une entité dont a besoin un processus pour s'exécuter. Les ressources peuvent être matérielles (processeur, mémoire, périphériques d'entrée/sortie, ...) mais aussi logicielles (variables).

Les différents processus se partagent les ressources, on parle alors d'accès concurrents aux ressources. Par exemple,

- les processus se partagent tous l'accès à la ressource "processeur"
- un traitement de texte et un IDE Python se partagent la ressource "clavier" ou encore la ressource "disque dur" (si on enregistre les fichiers), ...
- un navigateur et un logiciel de musique se partagent la ressource "carte son", ...

C'est le système d'exploitation qui est chargé de gérer les processus et les ressources qui leur sont nécessaires, en partageant leur accès au processeur. Nous allons voir comment tout de suite !

États d'un processus

Au cours de son existence, un processus peut se retrouver dans trois états :

- **état élu** : lorsqu'il est en cours d'exécution, c'est-à-dire qu'il obtient l'accès au processeur
- **état prêt** : lorsqu'il attend de pouvoir accéder au processeur
- **état bloqué** : lorsque le processus est interrompu car il a besoin d'attendre une ressource quelconque (entrée/sortie, allocation mémoire, etc.)

> Il est important de comprendre que le processeur ne peut gérer qu'un seul processus à la fois : le processus élu.

En pratique, lorsqu'un processus est créé il est dans l'état prêt et attend de pouvoir accéder au processeur (d'être élu). Lorsqu'il est élu, le processus est exécuté par le processeur mais cette exécution peut être interrompue :

- soit pour laisser la main à un autre processus (qui a été élu) : dans ce cas, le processus de départ repasse dans l'état prêt et doit attendre d'être élu pour reprendre son exécution
- soit parce que le processus en cours a besoin d'attendre une ressource : dans ce cas, le processus passe dans l'état bloqué.

Lorsque le processus bloqué finit par obtenir la ressource attendue, il peut théoriquement reprendre son exécution mais probablement qu'un autre processus a pris sa place et est passé dans l'état élu. Auquel cas, le processus qui vient d'être "débloqué" repasse dans l'état prêt en attendant d'être à nouveau élu.

Ainsi, l'état d'un processus au cours de sa vie varie entre les états prêt, élu et bloqué comme le résume le schéma suivant :

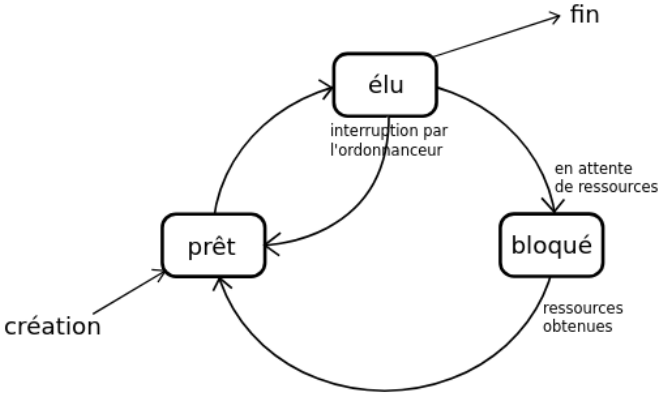


Fig. 1 - Cycle de vie d'un processus.

Lorsqu'un processus est interrompu, il doit pouvoir reprendre à l'endroit même où il a été interrompu. Pour cela, le système d'exploitation conserve pour chaque processus créé une zone mémoire (appelée PCB, pour *Process Control Bloc*, ou bloc de contrôle du processus) dans laquelle sont stockées les informations sur le processus : son PID, son état, la valeur des registres lors de sa dernière interruption, la zone mémoire allouée par le processus lors de son exécution, les ressources utilisées par le processus (fichiers ouverts, connexions réseaux en cours d'utilisation, etc.).

Ordonnancement

C'est le système d'exploitation qui attribue aux processus leurs états élu, prêt et bloqué. Plus précisément, c'est l'**ordonnanceur** (un des composants du système d'exploitation) qui réalise cette tâche appelée **ordonnancement des processus**.

L'objectif de l'ordonnanceur est de choisir le processus à exécuter à l'instant *t* (le processus *élu*) et déterminer le temps durant lequel le processeur lui sera alloué.

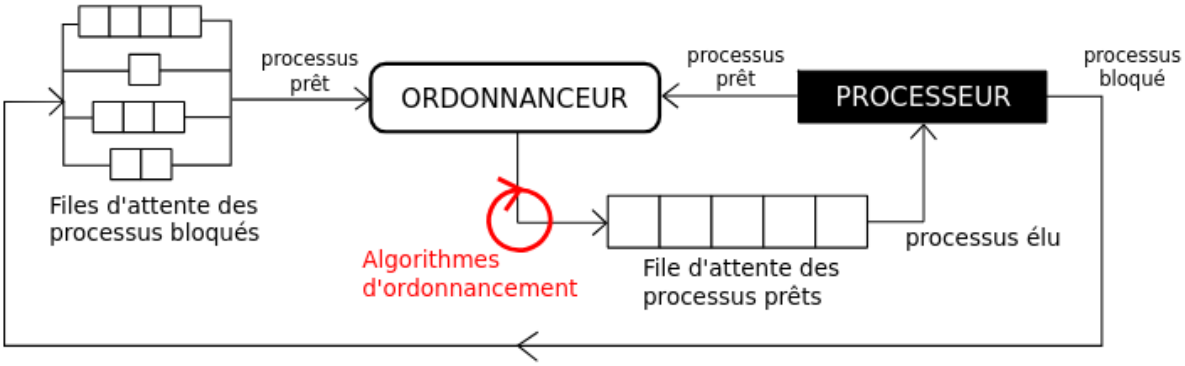


Fig. 2 - Ordonnancement des processus.

Ce choix est à faire parmi tous les processus qui sont dans l'état prêt, mais lequel sera élu ? et pour combien de temps ? Des **algorithmes d'ordonnancement** sont utilisés et il en existe plusieurs selon la stratégie utilisée. On en présente quelques-uns ci-dessous.

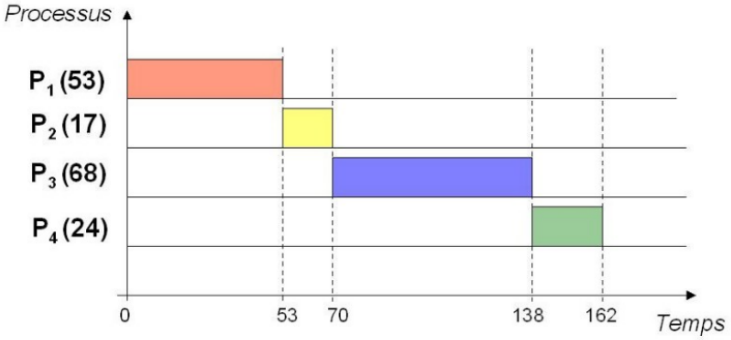
Définition 3

Ordonnancement First Come First Served (FCFS)

Principe : Les processus sont ordonnancés selon leur ordre d'arrivée ("premier arrivé, premier servi" en français).

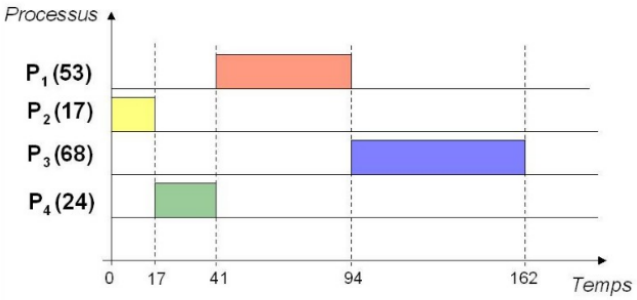
Exemple : Les processus $P_1(53)$, $P_2(17)$, $P_3(68)$ et $P_4(24)$ arrivent dans cet ordre à $t = 0$:

Cela signifie que P_1 , P_2, P_3 et P_4 ont besoin de respectivement 53, 17, 68 et 24 unités de temps pour s'exécuter.



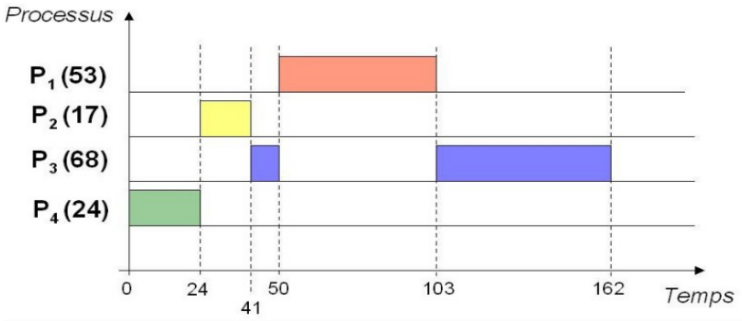
Définition 4

Ordonnancement Shortest Job First (SJF)
Principe : Le processus dont le temps d'exécution est le plus court est ordonnancé en premier.
Exemple : Les processus P_1 , P_2 , P_3 et P_4 arrivent dans cet ordre à $t = 0$.



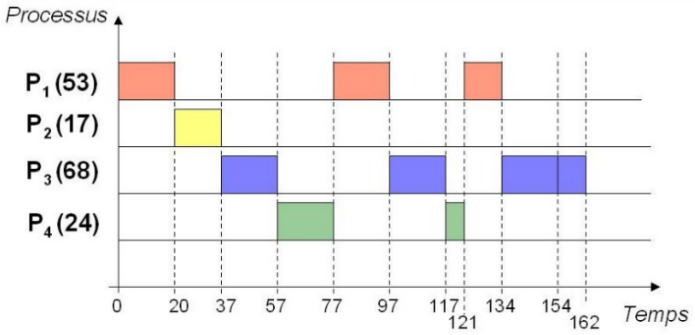
Définition 5

Ordonnancement Shortest Remaining Time (SRT)
Principe : Le processus dont le temps d'exécution restant est le plus court parmi ceux qui restent à exécuter est ordonnancé en premier.
Exemple : P_3 et P_4 arrivent à $t = 0$; P_2 à $t = 20$; P_1 à $t = 50$:



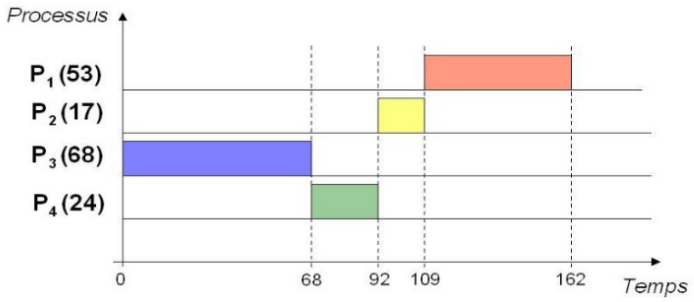
Définition 6

Ordonnancement temps-partagé (Round-Robin)
Principe : C'est la politique du **tourniquet** : allocation du processeur par tranche de temps . On appelle une tranche de temps un **quantum, noté q**.
Exemple : quantum q = 20, n = 4 processus.



Définition 7

Ordonnancement à priorités statiques
Principe : Allocation du processeur selon des priorités statiques (= numéros affectés aux processus pour toute la vie de l'application).
Exemple : priorités $(P_1,\ P_2,\ P_3,\ P_4) = (3,\ 2,\ 0,\ 1)$ où la priorité la plus forte est 0.
(attention, dans certains systèmes c'est l'inverse : 0 est alors la priorité la plus faible)



IV - Problèmes liés à l'accès concurrent aux ressources

Les processus se partagent souvent une ou plusieurs ressources, et cela peut poser des problèmes.

Problèmes de synchronisation : illustration avec Python

Exemple d'une variable partagée

Un problème de synchronisation intervient lorsque deux processus souhaitent utiliser une même variable au même instant.

Comment éviter les problèmes de synchronisation ?

On va utiliser ce qu'on appelle un **verrou** : un verrou est objet partagé entre plusieurs processus mais qui garantit qu'un seul processus accède à une ressource à un instant donné.

Concrètement, un verrou peut être acquis par les différents processus, et le premier 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 jusqu'à ce qu'il soit libéré.

Le module `multiprocessing` de Python propose un objet `Lock()` correpondant à un verrou. Deux méthodes sont utilisées :

- la méthode `.acquire()` permet de demander le verrou (le processus faisant la demande est bloqué tant qu'il ne l'a pas obtenu)
- la méthode `.release()` permet de libérer le verrou (il pourra alors être obtenu par un autre processus qui en fait la demande)

Un exemple d'utiliation de verrou sera étudié lors des TP de ce chapitre.

Nous terminons en voyant que l'utilisation de verrous n'est pas sans risque car elle peut engendrer des problèmes d'interblocage.

Risque d'interblocage

Les interblocages (*deadlock* en anglais) sont des situations de la vie quotidienne. L'exemple classique est celui du carrefour avec priorité à droite où chaque véhicule est bloqué car il doit laisser le passage au véhicule à sa droite.

En informatique l'interblocage peut également se produire lorsque plusieurs processus concurrents s'attendent mutuellement. Ce scénario peut se produire lorsque plusieurs ressources sont partagées par plusieurs processus et l'un d'entre eux possède indéfiniment une ressource nécessaire pour un autre.

Ce phénomène d'attente circulaire, où chaque processus attend une ressource détenue par un autre processus, peut être provoquée par l'utilisation de plusieurs verrous.

V - Et pour les systèmes multiprocesseurs ?

Les ordinateurs actuels possèdent généralement plusieurs processeurs, ce qui permet à plusieurs processus d'être exécutés parallèlement : un par processeur. Ce parallélisme permet bien évidemment une plus grande puissance de calcul.

Pour répartir les différents processus entre les différents processeurs, on distingue deux approches :

- l'approche partitionnée : chaque processeur possède un ordonnanceur particulier et les processus sont répartis entre les différents ordonnanceurs
- l'approche globale : un ordonnanceur global est chargé de déterminer la répartition des processus entre les différents processeurs

L'ordonnancement des processus des systèmes d'exploitation actuels est bien plus complexe que les quelques algorithmes évoqués dans ce cours, et cela dépasse largement le cadre du programme de NSI.

Source : Lycée Mounier - Angers