

1. 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 ?

2. Processus

2.1. Qu'est-ce qu'un *processus* ?

Il ne faut pas confondre **programme** et **processus** :

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

2.2. 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 **process**, 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 :

```
(base) terminale@mounier-01:~$ ps -aef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0  17:45 ?        00:00:02 /sbin/init splash
root           2         0  0  17:45 ?        00:00:00 [kthreadd]
root           3         2  0  17:45 ?        00:00:00 [rcu_gp]
root           4         2  0  17:45 ?        00:00:00 [rcu_par_gp]
root           6         2  0  17:45 ?        00:00:00 [kworker/0:0H-events_highpri]
root           9         2  0  17:45 ?        00:00:00 [mm_percpu_wq]
root          10         2  0  17:45 ?        00:00:00 [rcu_tasks_rude_]
root          11         2  0  17:45 ?        00:00:00 [rcu_tasks_trace]
root          12         2  0  17:45 ?        00:00:00 [ksoftirqd/0]
root          13         2  0  17:45 ?        00:00:01 [rcu_sched]
root          14         2  0  17:45 ?        00:00:00 [migration/0]
root          15         2  0  17:45 ?        00:00:00 [idle_inject/0]
root          16         2  0  17:45 ?        00:00:00 [cpuhp/0]
root          17         2  0  17:45 ?        00:00:00 [cpuhp/1]
root          18         2  0  17:45 ?        00:00:00 [idle_inject/1]
root          19         2  0  17:45 ?        00:00:00 [migration/1]
root          20         2  0  17:45 ?        00:00:00 [ksoftirqd/1]
root          22         2  0  17:45 ?        00:00:00 [kworker/1:0H-events_highpri]
...
termina+     3152     1557  0  18:42 ?        00:00:07 /usr/libexec/gnome-terminal-server
termina+     3160     3152  0  18:42 pts/0    00:00:00 bash
root         3194         2  0  18:42 ?        00:00:00 [kworker/0:1-events]
termina+     3202     1557  0  18:42 ?        00:00:09 /usr/lib/firefox/firefox -new-window
termina+     3261     3202  0  18:42 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -parentBuildID
termina+     3283     3202  0  18:42 ?        00:00:02 /usr/lib/firefox/firefox -contentproc -childID 1 -is
termina+     3330     3202  0  18:42 ?        00:00:01 /usr/lib/firefox/firefox -contentproc -childID 2 -is
termina+     3374     3202  0  18:43 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 3 -is
termina+     3401     1802  0  18:44 ?        00:00:00 /usr/lib/libreoffice/program/oosplash --writer
termina+     3437     3401  0  18:44 ?        00:00:01 /usr/lib/libreoffice/program/soffice.bin --writer
root         3500         2  0  18:58 ?        00:00:00 [kworker/u8:1-phy0]
termina+     3521     3202  0  18:59 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 4 -is
root         3548         2  0  19:03 ?        00:00:00 [kworker/u8:2-events_unbound]
root         3553         1  1  19:03 ?        00:00:00 /usr/libexec/fwupd/fwupd
root         3563         2  0  19:03 ?        00:00:00 [kworker/1:0-events]
termina+     3566     3160  0  19:04 pts/0    00:00:00 ps -aef
```

2.3. 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.

2.4. 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*.

À vous de jouer !

Si ce n'est pas déjà fait, lancez un terminal et exécutez la commande `ps -aef` (élargissez la fenêtre du terminal au maximum pour bien voir toutes les colonnes).

1. À quoi correspond le dernier processus lancé ? Quel est son processus parent et pourquoi ?
2. Ouvrez Writer de LibreOffice puis exécutez à nouveau la commande.
 - Cherchez dans la liste des processus celui ou ceux correspondant à l'exécution du programme Writer ?
 - Cherchez le processus parent du processus correspondant à l'exécution du programme Writer ?
3. Fermez LibreOffice puis exécutez à nouveau la commande et vérifiez qu'il n'y a plus de processus correspondant à l'exécution de Writer.
4. Vous devriez avoir le navigateur ouvert pour visualiser ce cours (lancez-le si ce n'est pas le cas). Exécutez à nouveau la commande puis :
 - Cherchez dans la liste des processus le premier correspondant à l'exécution du navigateur
 - Cherchez ensuite les processus fils de ce processus.
 - Ouvrez un nouvel onglet et rendez-vous sur une page Web de votre choix. Exécutez à nouveau la commande, vous devriez constater qu'au moins un nouveau processus lié

à l'exécution du navigateur a été créé.

Sous GNU/Linux il est possible de voir l'arborescence des processus avec la commande `ps tree`.

```
(base) terminale@mounier-01:~$ ps tree
systemd├─ModemManager─2*[{ModemManager}]
      │NetworkManager─2*[{NetworkManager}]
      │accounts-daemon─2*[{accounts-daemon}]
      │acpid
      │apache2─5*[{apache2}]
      │avahi-daemon─avahi-daemon
      │bluetoothd
      │colord─2*[{colord}]
      │cron
      │cups-browsed─2*[{cups-browsed}]
      │cupsd
      │dbus-daemon
      │fwupd─4*[{fwupd}]
      │gdm3├─gdm-session-wor├─gdm-x-session├─Xorg─9*[{Xorg}]
      │      │              │              │gnome-session-b├─ssh-agent
      │      │              │              │              │2*[{gnome-session-b}]
      │      │              │              └─2*[{gdm-x-session}]
      │      └─2*[{gdm3}]
      │gnome-keyring-d─3*[{gnome-keyring-d}]
      │irqbalance─{irqbalance}
      │2*[{kerneloops}]
      │mysqld─36*[{mysqld}]
      │networkd-dispat
      └─polkitd─2*[{polkitd}]
```

À vous de jouer !

Testez la commande et cherchez dans l'arborescence les processus correspond à l'exécution du navigateur (qui doit être lancé, cela va de soi)

3. Gestion des processus et des ressources

3.1. 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.

Remarque : 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, itération", 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, itération", 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

```
$ python progA.py & python 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* :

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python progA.py & python progB.py &
[1] 9154
[2] 9155
(base) terminale@mounier-01:~/Documents/TNSI/processus$ programme B en cours, itération 0
programme A en cours, itération 0
programme B en cours, itération 1
programme A en cours, itération 1
programme B en cours, itération 2
programme A 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 A en cours, itération 8
...
programme B en cours, itération 92
programme A en cours, itération 92
programme B en cours, itération 93
programme A en cours, itération 93
programme B en cours, itération 94
programme A en cours, itération 94
programme A en cours, itération 95
programme B en cours, itération 95
programme A en cours, itération 96
programme B en cours, itération 96
programme B en cours, itération 97
programme A en cours, itération 97
programme B en cours, itération 98
programme A en cours, itération 98
programme A en cours, itération 99
programme B en cours, itération 99

[1]- Fini          python progA.py
[2]+ Fini          python progB.py
```

3.2. 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 !

3.3. 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 **en bloqué** : lorsque le processus est interrompu car il a besoin d'attendre une ressource quelconque (entrée/sortie, allocation mémoire, etc.)

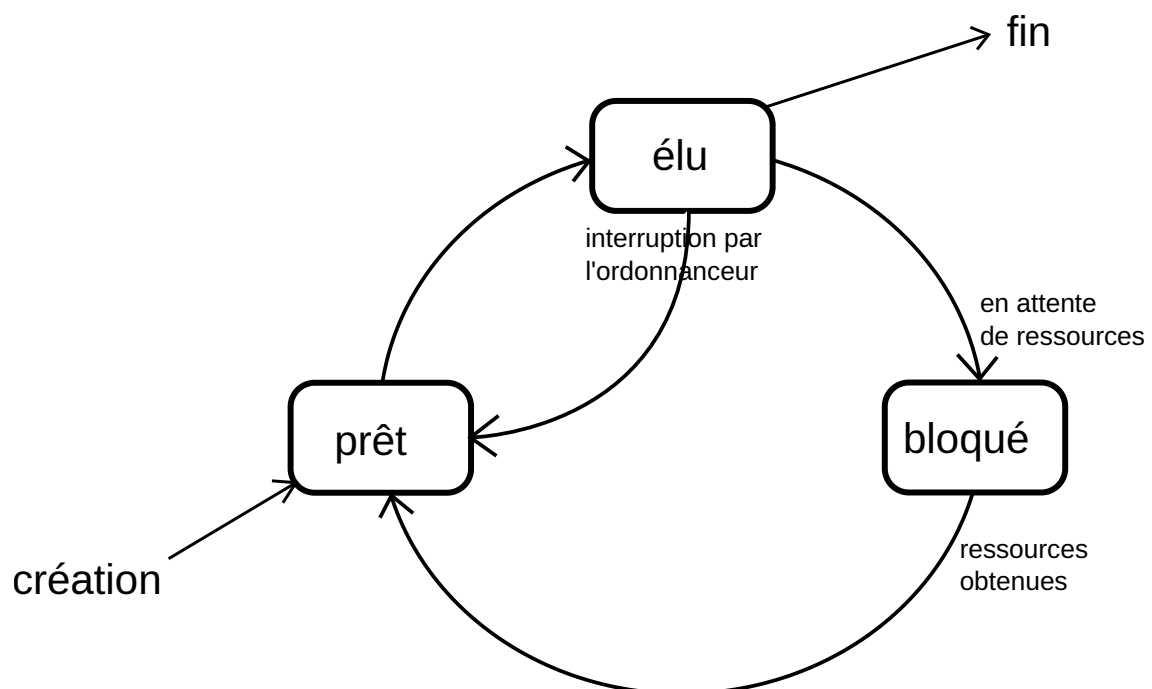
Il est important de comprendre que le processeur ne peut gérer qu'un seul processus à la fois : le processus *élu*.

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 :



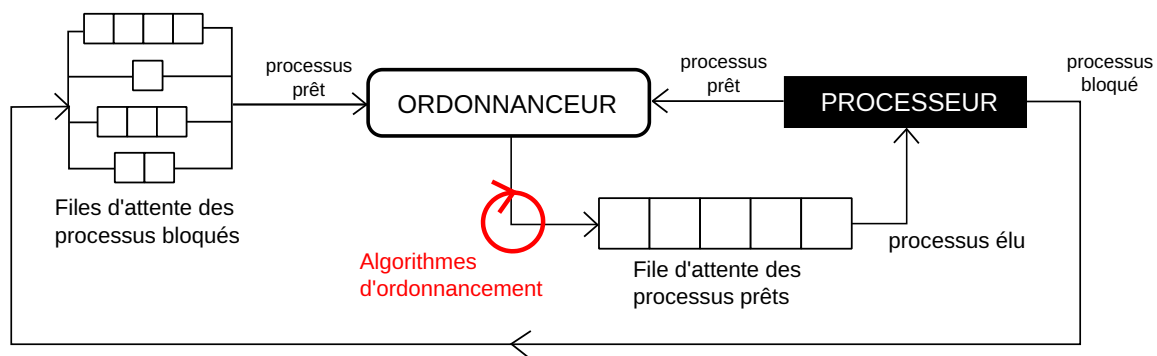
Lorsqu'un processus est interrompu, il doit pouvoir reprendre à l'endroit même où il a été interrompu. Pour cela, le système d'exploitation conserve pour chaque processus créé une zone mémoire (appelée PCB, pour *Process Control Bloc*, ou bloc de contrôle du processus) dans laquelle sont stockées les informations sur le processus : son PIB, 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.).

 Faites les exercices 1 et 2.

3.4. 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.



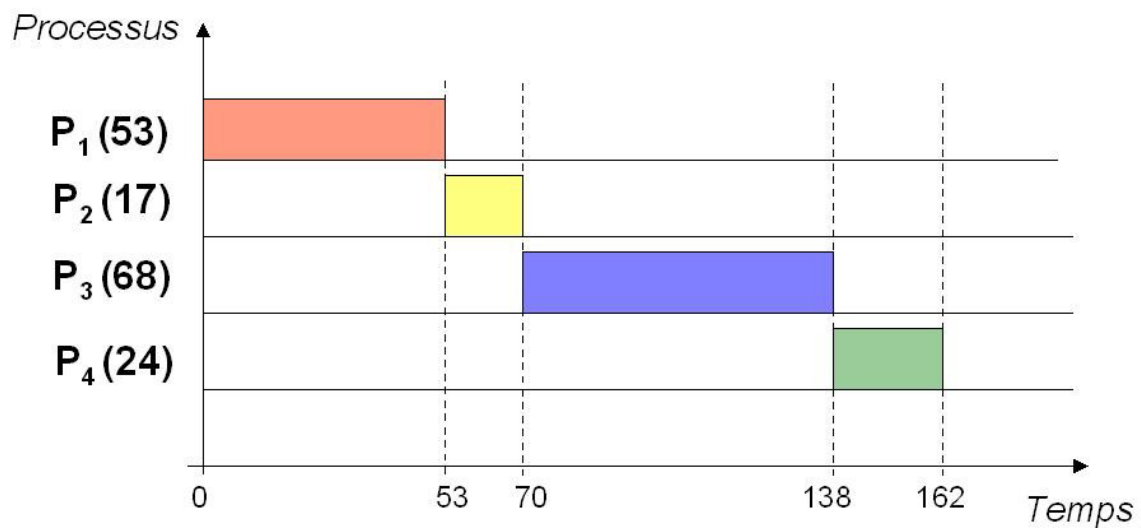
Ce choix est à faire parmi tous les processus qui sont dans l'état *prêt*, mais lequel sera élu ? et pour combien de temps ? Des algorithmes d'ordonnancement sont utilisés et il en existe plusieurs selon la stratégie utilisée. On en présente quelques-uns ci-dessous.

3.4.1. 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 $t = 0$:

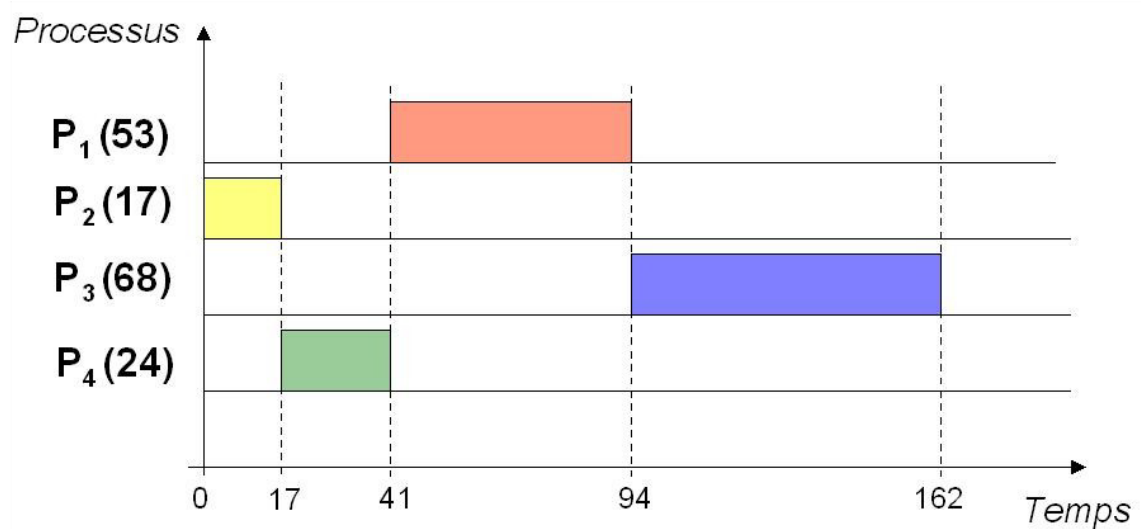
Cela signifie que P_1 , P_2 , P_3 et P_4 ont besoin de respectivement 53, 17, 68 et 24 unités de temps pour s'exécuter.



3.4.2. Ordonnancement Shortest Job First (SJF)

Principe : Le processus dont le temps d'exécution est le plus court est ordonné en premier.

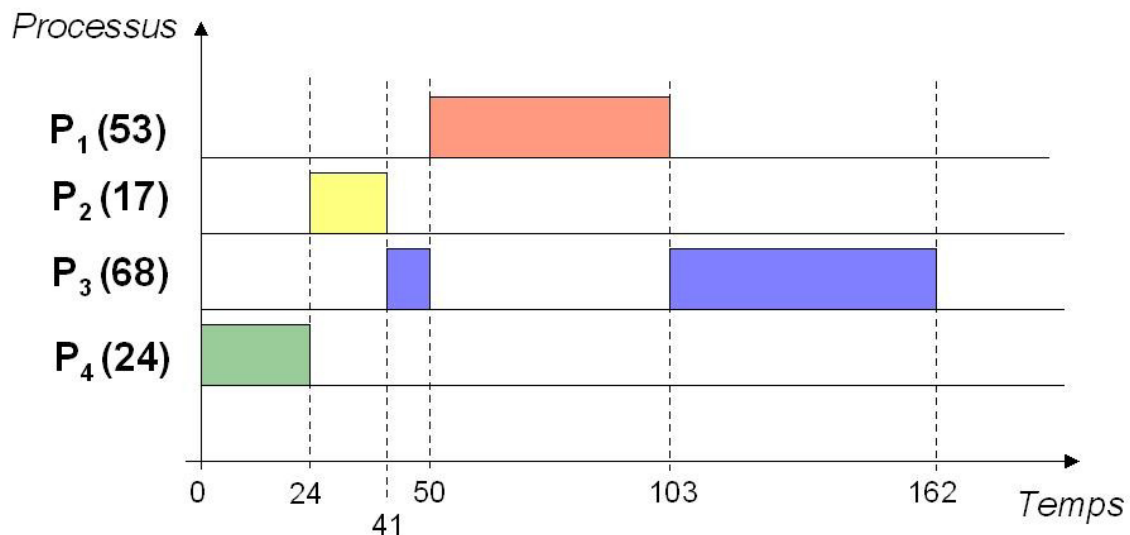
Exemple : P_1 , P_2 , P_3 et P_4 arrivent à $t = 0$:



3.4.3. 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 ordonné en premier.

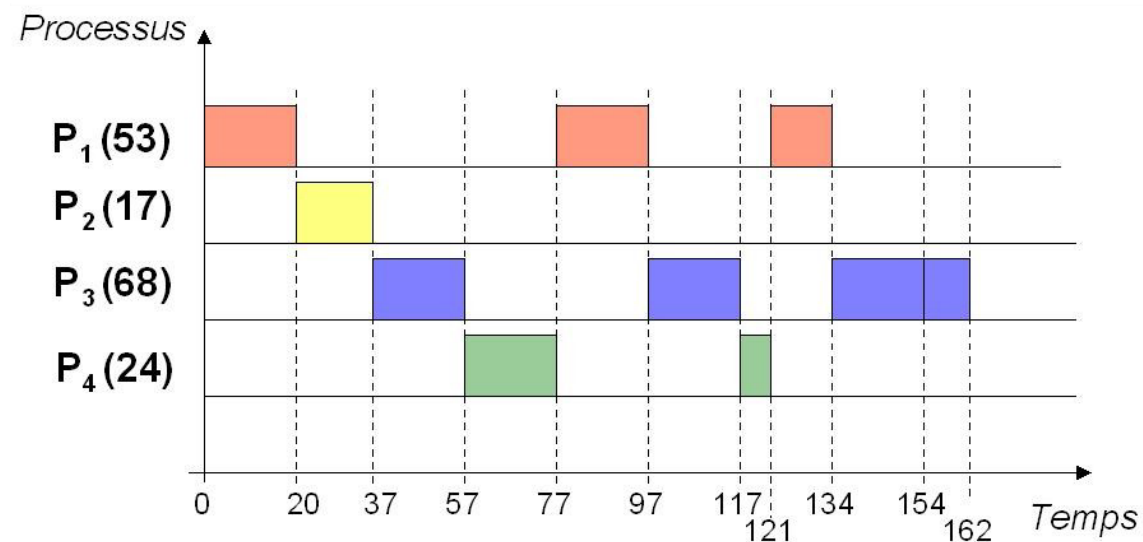
Exemple : P_3 et P_4 arrivent à $t = 0$; P_2 à $t = 20$; P_1 à $t = 50$:



3.4.4. Ordonnancement temps-partagé (Round-Robin)

Principe : C'est la politique du tourniquet : allocation du processeur par tranche (= quantum q) de temps.

Exemple : quantum $q = 20$ et $n = 4$ processus

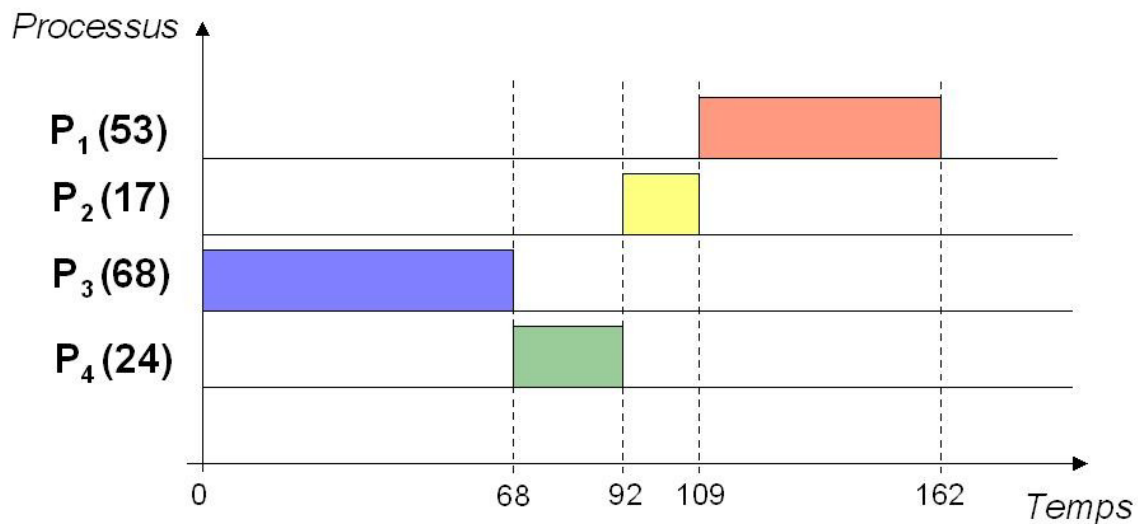


Dans ce cas, s'il y a n processus, chacun d'eux obtient le processeur au bout de $(n - 1) \times q$ unités de temps au plus

3.4.5. 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)



 Faites l'exercice 3.

4. Problèmes liés à l'accès concurrent aux ressources

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

4.1. Problèmes de synchronisation : illustration avec Python

4.1.1. Exemple d'une variable partagée

Prenons l'exemple d'une variable (= ressource logicielle) partagée entre plusieurs processus. Plus précisément, considérons un programme de jeu multi-joueur dans lequel une variable `nb_pions` représente le nombre de pions disponibles pour tous les joueurs.

Une fonction `prendre_un_pion()` permet de prendre un pion dans le tas commun de pions disponibles, s'il reste au moins un pion évidemment.

On va se mettre dans la situation où il ne reste plus qu'un pion dans le tas commun et on suppose que deux joueurs utilisent la fonction `prendre_un_pion()`, ce qui conduit à la création de deux processus `p1` et `p2`, chacun correspondant à un joueur.

Avec Python, on peut utiliser le module `multiprocessing` pour créer des processus. Le programme Python `pions.py` suivant permet de réaliser la situation de jeu décrite :

```
from multiprocessing import Process, Value
import time

def prendre_un_pion(nombre):
    if nombre.value >= 1:
```

```

        time.sleep(0.001) # pour simuler un traitement avec des calculs
        temp = nombre.value
        nombre.value = temp - 1 # on décrémente le nombre de pions

if __name__ == '__main__':
    # création de la variable partagée initialisée à 1
    nb_pions = Value('i', 1)
    # on crée deux processus
    p1 = Process(target=prendre_un_pion, args=[nb_pions])
    p2 = Process(target=prendre_un_pion, args=[nb_pions])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)

```

Explications :

- Le `if __name__ == '__main__':` permet de ne créer qu'une seule fois les processus `p1` et `p2` qui suivent (c'est nécessaire sous Windows, pas sous GNU/Linux car la création des processus ne se fait pas de la même manière, mais cela reste conseillé ne serait-ce que pour des raisons de compatibilité), on n'en dira pas davantage ici car cela dépasse le niveau de ce cours.
- On a utilisé la classe `Process` du module `multiprocessing` pour instancier deux processus `p1` et `p2`.
 - L'argument `target` est le nom de la fonction qui sera exécutée par le processus : ici les deux processus doivent exécuter la même fonction `prendre_un_pion()`
 - L'argument `args` est une liste des arguments passés à la fonction cible : ici il s'agit de la variable `nb_pions` qui est partagée par les deux processus.
- Par défaut, deux processus ne partagent pas de données en mémoire : on ne peut pas donc pas utiliser `nb_pions` comme une variable globale. Il faut utiliser la classe `Value` du module `multiprocessing` pour créer `nb_pions` dans une mémoire partagée entre les processus. L'argument `'i'` indique que `nb_pions` est un entier (signé) et le deuxième argument est la valeur initiale de la variable, ici 1.
- La fonction `prendre_un_pion()` prend un nombre en paramètre et décrémente sa valeur d'une unité si le nombre est au moins égal à 1.
 - Lors de l'exécution de la fonction par les deux processus, l'argument en question sera l'objet `nb_pions` de la classe `Value` et on accède à sa valeur avec l'attribut `value`.
 - On a ajouté une temporisation permettant de simuler d'autres calculs qui pourraient avoir lieu (par exemple, des instructions de mise à jour du nombre de pions des joueurs)
- Les dernières lignes permettent de démarrer les deux processus et attendre qu'ils soient terminés pour afficher la valeur finale de `nb_pions`.

Si on exécute ce programme, les deux processus `p1` et `p2` sont exécutés et on s'attend au comportement suivant (en supposant qu'il ne reste qu'un seul pion dans le tas commun) :

- l'un des deux est élu en premier, par exemple `p1`, et exécute la fonction `prendre_un_pion()`, le nombre de pions est égal à 1 donc `nb_pions` est décrémenté d'une unité et prend donc la valeur 0, le processus `p1` est terminé ;
- le processus `p2`, qui était en attente, est ensuite élu, et comme le nombre de pions est désormais égal à 0 rien ne se passe et `p2` termine.

Ainsi, le premier joueur a pu prendre le pion restant et le second s'est retrouvé coincé, et la valeur finale de `nb_pions` vaut 0.

Et pourtant, il est tout à fait possible que les choses ne se passent pas ainsi ! En effet, en exécutant plusieurs fois le programme `pions.py` dans un terminal, on obtient parfois une valeur finale égale à 0 et parfois égale à -1 :

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
```

C'est un résultat très perturbant non ? Expliquons pourquoi !

Pour cela, on peut ajouter quelques instructions d'affichage pour suivre ce qu'il se passe. On obtient le script `pions_v2.py` suivant :

```
from multiprocessing import Process, Value
import time

def prendre_un_pion(nombre, numero_processus):
    print(f"début du processus {numero_processus}")
    if nombre.value >= 1:
        print(f"processus {numero_processus} : étape A")
        time.sleep(0.001) # pour simuler un traitement avec des calculs
        print(f"processus {numero_processus} : étape B")
        temp = nombre.value
        nombre.value = temp - 1 # on décrémente le nombre de pions
        print(f"nombre de pions restants à la fin du processus {numero_processus} : {nombre.value}")

if __name__ == '__main__':
```

```
# création de la variable partagée initialisée à 1
nb_pions = Value('i', 1)
# on crée deux processus
p1 = Process(target=prendre_un_pion, args=[nb_pions, 1])
p2 = Process(target=prendre_un_pion, args=[nb_pions, 2])
# on démarre les deux processus
p1.start()
p2.start()
# on attend la fin des deux processus
p1.join()
p2.join()
print("nombre final de pions :", nb_pions.value)
```

Explications :

- Lors de la création des processus, on passe un deuxième argument à la fonction `prendre_un_pion()`, le numéro du processus : 1 pour `p1` et 2 pour `p2`.
- Cela permet d'afficher dans cette fonction le numéro du processus à des endroits stratégiques : au début, à l'entrée dans le `if`, juste avant de décrémenter le nombre de pions et à la fin du processus.

En exécutant `pions_v2.py` dans un terminal, on obtient ce genre de choses :

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
début du processus 2
processus 1 : étape A
processus 2 : étape A
processus 1 : étape B
processus 2 : étape B
nombre de pions restants à la fin du processus 2 : -1
nombre de pions restants à la fin du processus 1 : 0
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
processus 1 : étape A
processus 1 : étape B
début du processus 2
nombre de pions restants à la fin du processus 1 : 0
nombre de pions restants à la fin du processus 2 : 0
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
processus 1 : étape A
début du processus 2
processus 1 : étape B
processus 2 : étape A
nombre de pions restants à la fin du processus 1 : 0
processus 2 : étape B
nombre de pions restants à la fin du processus 2 : -1
nombre final de pions : -1
```

Analysons la première exécution du programme.

- le processus `p1` est élu en premier (affichage de "début du processus 1") mais est de suite interrompu par l'ordonnanceur qui élit `p2` (affichage de "début du processus 2")

- puis c'est de nouveau le processus `p1` qui a la main et il rentre dans le `if` (affichage de "processus 1 : étape A") mais est interrompu à nouveau et l'ordonnanceur donne la main à `p2` qui rentre aussi dans le `if` (affichage de "processus 2 : étape A") : en effet, à ce stade le nombre de pions n'a pas encore été décrémenté par `p1` car il a été interrompu avant l'étape B, et donc `p2` a pu entrer dans le `if` puisque la condition `nombre.value >= 1` est toujours vraie à ce moment là !!
- ensuite le processeur est alloué alternativement à `p1` et `p2` (voir les affichages restants) mais le mal est fait puisque les deux processus sont désormais chacun entrés dans le `if`, ils vont chacun décrémenter le nombre de pions d'une unité et chacun des deux joueurs aura pioché un pion alors qu'il n'y en avait qu'un seul au départ !

Vous remarquerez que la troisième exécution du programme met en évidence le même problème car les deux processus ont chacun pu entrer dans le `if`, même si l'ordre des instructions exécutées après n'est pas tout à fait le même.

Si on analyse la seconde exécution du programme qui donne le comportement souhaité, on constate que `p1` a eu suffisamment de temps pour décrémenter le nombre de pions (qui vaut désormais 0) *avant* que `p2` ne fasse le test `nombre.value >= 1` et se rende compte que cette condition est fausse. Dans ce cas, seul le premier joueur a pu piocher un pion.

Heureusement, on peut éviter le problème mis en évidence dans l'exemple précédent.

4.1.2. 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()` correspondant 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)

On peut alors régler le problème de l'exemple précédent avec le script `pions_v3.py` suivant dans lequel on a laissé les affichages pour bien suivre :

```
from multiprocessing import Process, Value, Lock
import time

def prendre_un_pion(v, nombre, numero_processus):
```

```

print(f"début du processus {numero_processus}")
v.acquire() # acquisition du verrou
if nombre.value >= 1:
    print(f"processus {numero_processus} : étape A")
    time.sleep(0.001)
    print(f"processus {numero_processus} : étape B")
    temp = nombre.value
    nombre.value = temp - 1
v.release() # verrou libéré
print(f"nombre de pions restants à la fin du processus {numero_processus}
: {nombre.value}")

if __name__ == '__main__':
    # création de la variable partagée initialisée à 1
    nb_pions = Value('i', 1)
    # verrou partagé par les deux processus
    verrou = Lock()
    # on crée deux processus
    p1 = Process(target=prendre_un_pion, args=[verrou, nb_pions, 1])
    p2 = Process(target=prendre_un_pion, args=[verrou, nb_pions, 2])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)

```

En exécutant (plusieurs fois) ce script dans un terminal on constate que le nombre final de pions est toujours égal à 0.

```

(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v3.py
début du processus 1
processus 1 : étape A
début du processus 2
processus 1 : étape B
nombre de pions restants à la fin du processus 1 : 0
nombre de pions restants à la fin du processus 2 : 0
nombre final de pions : 0

```

Avant de faire le test du `if`, le processus essaye d'acquérir le verrou avec `v.acquire()`. Dès qu'il est acquis, le processus a la garantie qu'il est le seul à pouvoir exécuter le code jusqu'à l'instruction `v.release()`. Cette portion de code protégée s'appelle une *section critique*. Cela ne veut pas dire que le processus détenant le verrou ne peut pas être interrompu, mais il ne le sera pas par un processus qui essaie d'acquérir le même verrou.

```

def prendre_un_pion(v, nombre, numero_processus):
    v.acquire()
    # début section critique
    if nombre.value >= 1:
        time.sleep(0.001)
        temp = nombre.value
        nombre.value = temp - 1
    # fin de la section critique
    v.release()

```


Si vous analysez l'affichage précédent dans le terminal, on voit d'ailleurs que `p1` est entré en section critique (affichage "processus 1 : étape A") mais est interrompu, puis c'est `p2` qui a la main (affichage "début processus 2") mais il va se retrouver bloquer à l'instruction `v.acquire()` puisque c'est `p1` qui détient le verrou. Lorsque `p1` reprendra la main, il pourra exécuter ses instructions jusqu'à `v.release()` sans être interrompu par `p2` (alors `nb_pions` sera décrémenté d'une unité). Lorsque `p1` libère le verrou, `p2` pourra alors l'obtenir, exécuter sa section critique et constater que la condition `nombre.value >= 1` est fausse : le deuxième joueur ne pourra alors pas prendre de pion.

 Faites l'exercice 4.

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

4.2. 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.

Considérons le script `interblocage.py` suivant dans lequel on a créé deux verrous `v1` et `v2` utilisés par deux fonctions `f1` et `f2` exécutées respectivement par deux processus `p1` et `p2`. Le processus `p1` essaie d'acquérir d'abord `v1` puis `v2` tandis que le processus `p2` essaie de les acquérir dans l'ordre inverse.

```
from multiprocessing import Process, Lock
import time
import os
```

```

def f1(v1, v2):
    print("PID du processus 1:", os.getpid())
    for i in range(100):
        time.sleep(0.001)
        v1.acquire()
        v2.acquire()
        print("processus 1 en cours, itération ", i)
        v2.release()
        v1.release()

def f2(v1, v2):
    print("PID du processus 2:", os.getpid())
    for i in range(100):
        time.sleep(0.001)
        v2.acquire()
        v1.acquire()
        print("processus 2 en cours, itération ", i)
        v1.release()
        v2.release()

if __name__ == '__main__':
    # création de deux verrous
    v1 = Lock()
    v2 = Lock()
    # création de deux processus
    p1 = Process(target=f1, args=[v1, v2])
    p2 = Process(target=f2, args=[v1, v2])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()

```

Si on exécute ce programme, il y a de grandes chances de se retrouver bloqué. Par exemple, dans le cas de l'exécution suivante :

- le processus `p1` est élu : il s'exécute jusqu'à l'acquisition de `v1` mais avant la tentative d'acquisition de `v2`, puis est interrompu
- le processus `p2` est à son tour élu : il s'exécute et acquiert `v2` qui est toujours libre, puis bloque sur l'acquisition de `v1` (qui est détenu par `p1`).
- le processus `p1` reprend la main et bloque sur l'acquisition de `v2` (détenu par `p2`).

Chaque processus détient un verrou et attend l'autre : ils sont en interblocage et l'attente est infinie.

On peut lancer (plusieurs fois si nécessaire) le script à partir du terminal et constater que l'interblocage a lieu très souvent.

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python interblocage.py
PID du processus 1: 8099
PID du processus 2: 8100
processus 1 en cours, itération 0
processus 2 en cours, itération 0
processus 1 en cours, itération 1
processus 2 en cours, itération 1
processus 1 en cours, itération 2
processus 2 en cours, itération 2
processus 1 en cours, itération 3
processus 2 en cours, itération 3
processus 1 en cours, itération 4
processus 2 en cours, itération 4
processus 1 en cours, itération 5
processus 2 en cours, itération 5
processus 1 en cours, itération 6
processus 2 en cours, itération 6
processus 1 en cours, itération 7
processus 2 en cours, itération 7
processus 1 en cours, itération 8
processus 2 en cours, itération 8
processus 1 en cours, itération 9
processus 2 en cours, itération 9
processus 1 en cours, itération 10
processus 2 en cours, itération 10
processus 1 en cours, itération 11
```

Il n'y a alors pas d'autres choix que d'interrompre les processus en interblocage, par exemple avec la commande `kill`.

Cependant, ce problème a lieu ici car les deux processus essaie d'acquérir les verrous dans l'ordre contraire. Si l'ordre d'acquisition est le même pour les processus, le problème n'a plus lieu (n'hésitez pas à tester !).

De manière générale, dans des problèmes complexes les situations d'interblocage sont difficiles à détecter et il se peut très bien que le programme se comporte bien pendant toute une phase de tests mais bloque lors d'une exécution ultérieure puisque l'on ne peut pas prévoir l'ordonnancement des processus.

 Faites les exercices 5 et 6.

5. 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. Si vous souhaitez en savoir plus, voici néanmoins une vidéo intéressante (en français) sur l'ordonnancement du noyau Linux : https://www.youtube.com/watch?v=uCGe5WWd1OI&t=195s&ab_channel=Vitonimal.

6. Bilan

- Un programme en cours d'exécution s'appelle un *processus*. Les systèmes d'exploitation récents permettent d'exécuter plusieurs processus simultanément.
- En réalité, ces processus sont exécutés à *tour de rôle* par le système d'exploitation qui est chargé d'allouer à chacun d'eux 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.
- Au cours de leur vie, les processus varient entre trois états : *élu* si le processus est exécuté par le processeur, *prêt* si le processus est prêt à être exécuté, et *bloqué* si le processus est en attente d'une ressource.
- C'est l'*ordonnanceur* qui est chargé de définir l'ordre dans lequel les processus doivent être exécutés par le processeur. Ce choix se fait grâce à des algorithmes d'ordonnancement.
- Les processus se partagent les différentes ressources, on parle d'*accès concurrent* aux ressources. Ce partage des ressources n'est pas sans risque et peut conduire à des problèmes de synchronisation. Ces problèmes peuvent être évités en utilisant un *verrou*, qui permet à un processus de ne pas être interrompu dans sa section critique par un autre processus demandant le même verrou.
- L'utilisation de plusieurs verrous peut entraîner des *interblocages*, c'est-à-dire des situations où chaque processus attend une ressource détenue par un autre, conduisant à une attente cyclique infinie. L'ordre d'acquisition des verrous est important mais pas toujours évident à écrire dans le cas de problèmes complexes.

Références : - Equipe éducative DIU EIL, cours sur le *Partage des ressources et virtualisation*, Audrey Queudet, Université de Nantes. - Cours d'Olivier Lecluse sur la [Gestion des ressources](#) - Documentation officielle de la bibliothèque [multiprocessing](#) de Python. - Cours de David Roche sur les [processus](#). - Livre *Numérique et Sciences Informatiques, 24 leçons, Terminale*, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, éditions ELLIPSES.

Ressource éducative libre distribuée sous [Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International](#)

