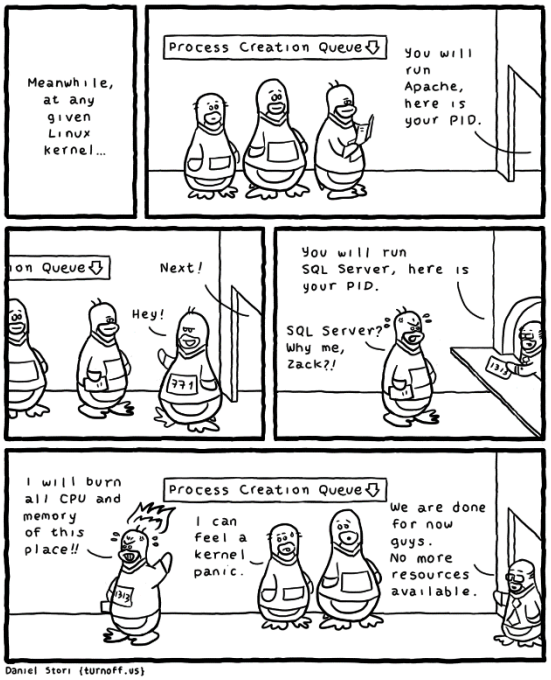
Cours – Gestion des processus et des ressources

Objectifs :

- ⇨ Découvrir la façon dont les systèmes d'exploitation modernes gèrent les processus et les ressources
- ⇨ Se sensibiliser aux problèmes de la programmation concurrente



I – Rôle d'un système d'exploitation

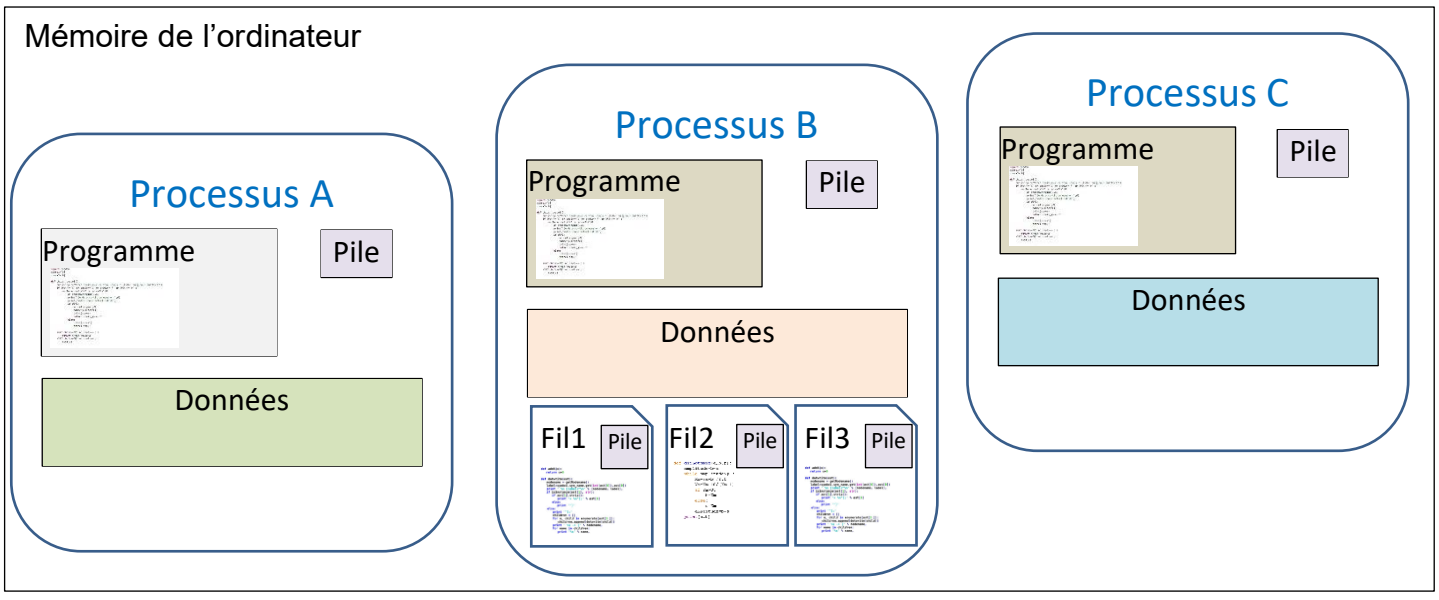
On a vu l'an dernier que le système d'exploitation ou OS (Operating System) est le logiciel qui gère l'utilisation des ressources matérielles de l'ordinateur et les interactions avec les logiciels applicatifs et l'utilisateur.
Le ou les processeurs est(sont) une ressource au même titre que la mémoire ou un fichier sur le disque.

II – Processus et fils d'exécution

Un **processus** (*process* en anglais) est une *instance d'exécution*[1] d'un programme. A ce titre, un processus dispose de :

- un ensemble d'instructions à exécuter (son programme) et une pile d'appel ;
- un ensemble de données à manipuler (son espace mémoire) ;
- éventuellement, d'autres ressources, comme des descripteurs de fichiers, des ports réseau, etc.

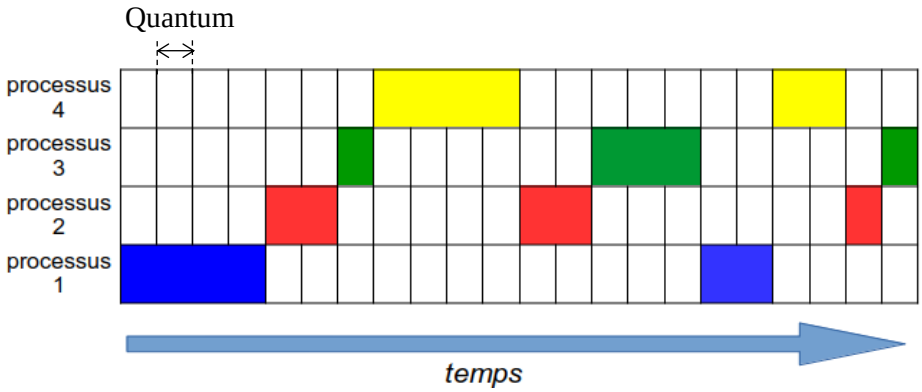
Un **fil d'exécution** (ou ***thread*** en anglais) est un processus léger qui partage la mémoire et les ressources du processus qui l'a créée. Ainsi au sein de son espace mémoire, un processus peut créer un ou plusieurs fils d'exécution qui ont chacun leur propre programme et pile d'appel mais accèdent au même espace mémoire.



[1] Ainsi un même programme peut être exécuté simultanément plusieurs fois dans des processus distincts.

III – Le multitâche

Les systèmes d'exploitation modernes sont capables d'exécuter plusieurs programmes en même temps. On dit qu'ils sont **multitâches**.
En réalité, la plupart du temps les programmes ne s'exécutent pas en même temps, mais plutôt l'un après l'autre par tout petits bouts, donnant l'illusion qu'ils s'exécutent de façon simultanée.
Dans une architecture monoprocesseur et monocore, le processeur ne peut effectuer les opérations que séquentiellement. Pour simuler l'exécution parallèle de traitements, on va utiliser la méthode du temps partagé (time-sharing » en anglais) où le processeur va exécuter tel traitement pendant un certain temps, puis passer au traitement suivant, et ainsi de suite avant de traiter à nouveau le premier. L'utilisateur a ainsi l'illusion que les traitements s'effectuent en parallèle.



Il y a un **changement de contexte** à chaque fois que le processeur doit passer d'un processus à un autre et cette opération a un coût en temps d'exécution. Le temps de changement de contexte est plus court entre des fils d'exécution qu'entre des processus séparés, car il n'y a alors pas besoin de redéfinir le contexte mémoire.

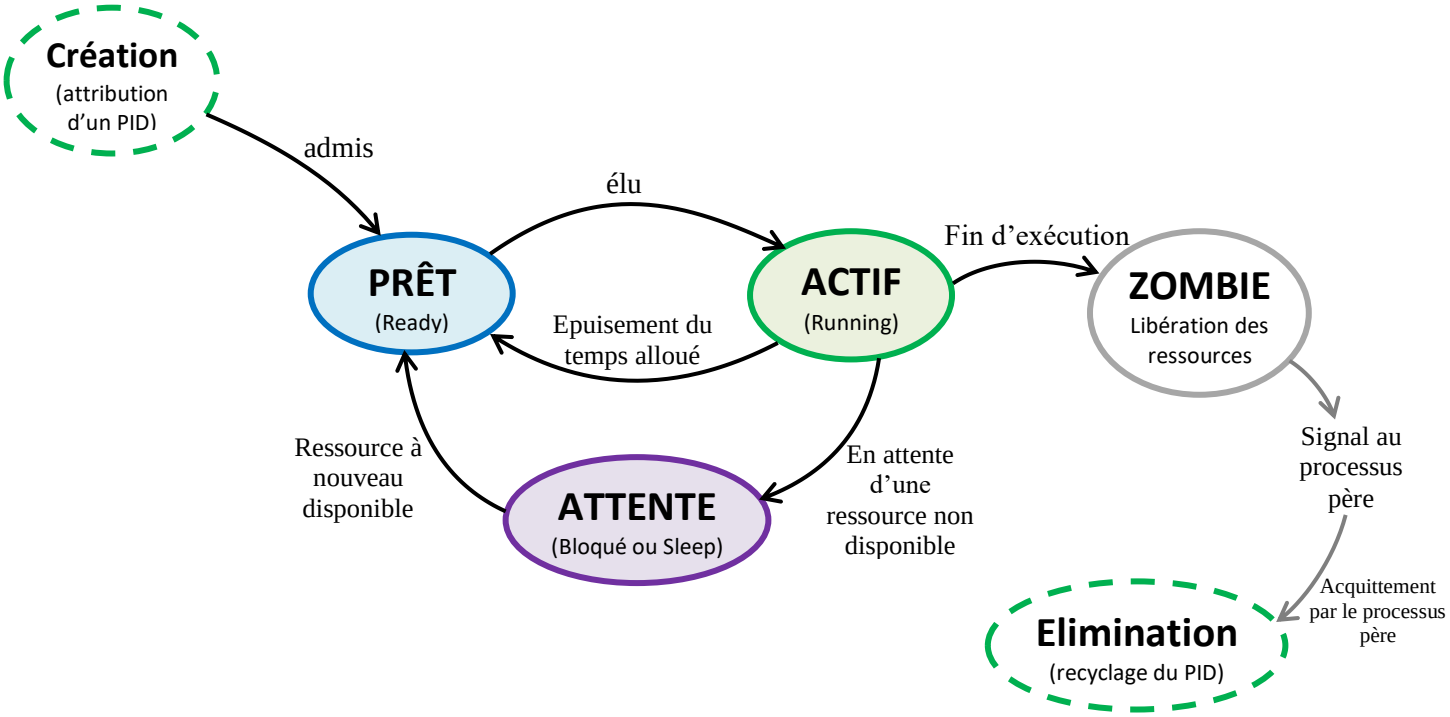
Il existe deux façons de gérer la commutation d'un processus à un autre :

- Le **multitâche coopératif** où c'est le processus qui doit de lui-même redonner la main au système d'exploitation pour qu'il donne le contrôle au processus suivant et ainsi de suite.
- Le **multitâche préemptif** où le traitement d'un processus est interrompu par le système d'exploitation pour donner la main au processus suivant.

Les premiers OS étaient en multitâche coopératif (Windows jusqu'à 3.11 et MacOS jusqu'à MacOS 9). Ce type de fonctionnement convient bien si on utilise plusieurs logiciels qui passent l'essentiel de leur temps à attendre l'utilisateur (bureautique, …), mais le système est très peu réactif lorsqu'on fait tourner des applications gourmandes en calcul (vidéos, simulation numérique, …) et une application qui part dans une boucle infinie bloque tout le système.
Le multitâche préemptif règle ces problèmes et permet même de prioriser le traitement de certaines tâches. C'est de ce type de multitâche que l'on va parler exclusivement par la suite.

1) L'ordonnanceur

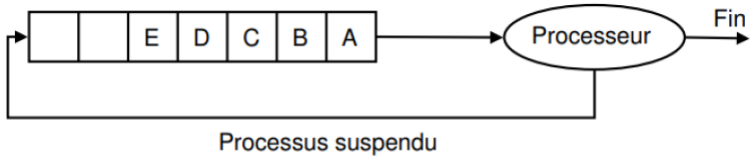
Mais comment répartir le temps de calcul de chaque processus ? Il existe plusieurs stratégies possibles et c'est le rôle de **l'ordonnanceur** (*scheduler* en anglais) de gérer le partage du temps processeur entre les différents processus.
L'ordonnanceur possède une table de tous les processus lancés sur le système. Chaque processus se voit ainsi attribué un numéro (PID : Process Identifier) lors de sa création. Ces processus peuvent être dans un des états ci-dessous :



Il existe de nombreux algorithmes pour répartir le temps machine entre les différents processus. Un des plus utilisé (et le seul que nous verrons) est celui du tourniquet circulaire.

Algorithme du tourniquet circulaire : Round Robin (RR)

L'algorithme du tourniquet, circulaire ou Round Robin (RR) représenté sur la figure ci-dessous est un algorithme ancien, simple, fiable et très utilisé. Il mémorise dans une file du type FIFO (First In First Out) la liste des processus prêts, c'est-à-dire en attente d'exécution.



Ordonnancement circulaire

- Choix du processus à exécuter
  Il alloue le processeur au processus en tête de file, pendant un quantum de temps. Si le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus (celui en tête de file). Si le processus ne se termine pas au bout de son quantum, son exécution est suspendue. Le processeur est alloué à un autre processus (celui en tête de file). Le processus suspendu est inséré en queue de file. Les processus qui arrivent ou qui passent de l'état attente à l'état prêt sont insérés en queue de file.
- Choix de la valeur du quantum
  Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop élevé augmente le temps de réponse des courtes commandes en mode interactif. Un quantum entre 20 et 50 ms est souvent un compromis raisonnable (le quantum était de 1 s dans les premières versions d'UNIX).

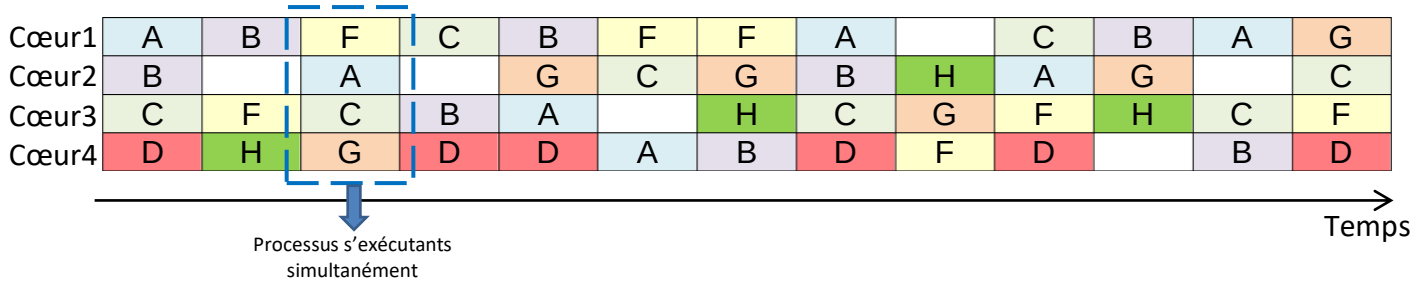
2) Exécution parallèle et exécution concurrente

Dans le cas où des processus s'exécutent tour à tour avec des commutations rapides et fréquentes de l'un à l'autre, on parle d'**exécution concurrente**.

Dans le cas ou des processus s'exécutent réellement en même temps, on parle d'**exécution parallèle**.

C'est le cas des machines multi-processeur (rare chez les particuliers) ou des processeurs multi-cores (présents dans tous les ordinateurs modernes).

Lorsque plusieurs processus ou fils d'exécution peuvent réellement s'exécuter simultanément, cela permet un gain de performance important pourvu que l'application ait été prévue pour diviser ses calculs en plusieurs threads.



3) Mise en application sur linux

Pour cette partie, nous allons utiliser des commandes dans un terminal linux. Il faut donc se mettre sous cet OS et démarrer un terminal.

a. Constatations initiales

Application 1 :

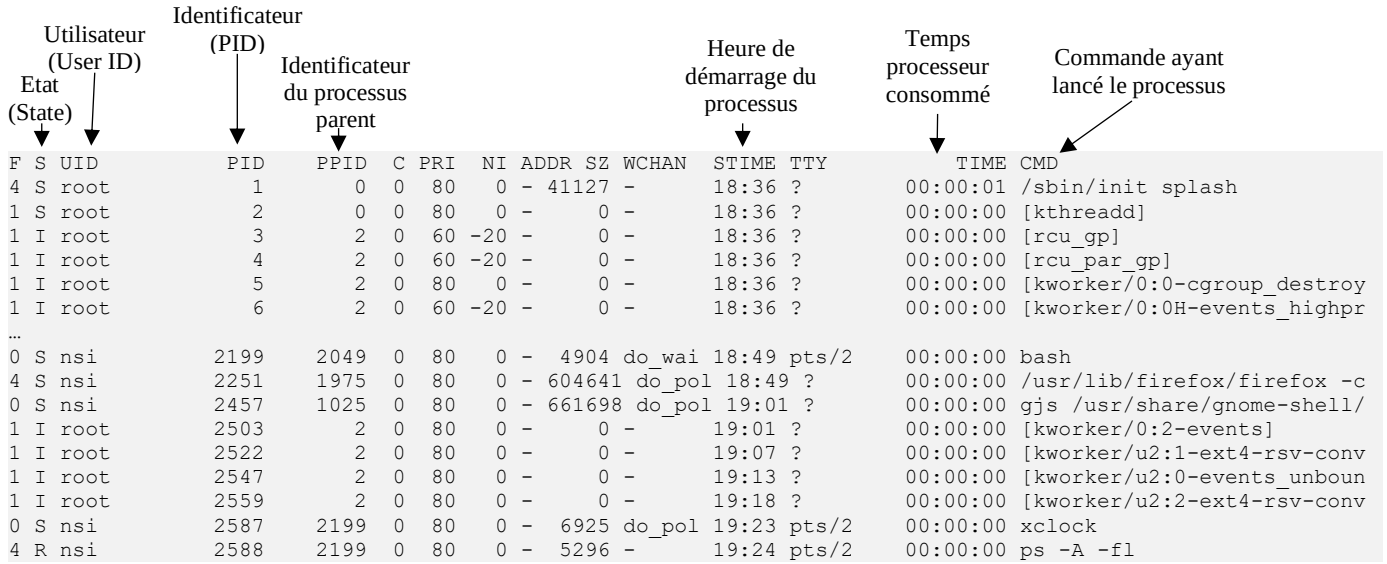
1) Dans le terminal, taper la commande `xcalc`. Qu'observez-vous ? Peut-on continuer à écrire des commandes dans la fenêtre du terminal ?
2) Que se passe-t-il si on ferme la fenêtre du terminal alors que la calculatrice xcalc est encore en fonctionnement ?

Lorsqu'on lance un programme dans la fenêtre du terminal, celui-ci lance un nouveau processus et se met en attente de la fin de celui-ci pour retourner dans la boucle REPL (Read Evaluate Print Loop). Pour éviter ce comportement et revenir directement à la boucle REPL sans attendre la fin du processus fils, il faut « détacher » la commande du terminal en ajoutant le signe « `&` » précédé d'un espace en fin de ligne.

2) Lancer les commandes `xcalc` et `xclock` en les détachant du terminal. Vérifier que vous pouvez encore utiliser le terminal en naviguant dans l'arborescence. Que se passe-t-il si on ferme la fenêtre du terminal alors que xcalc et xclock sont encore ouverts ? Même question si on ferme le terminal en utilisant la commande `exit`.

b. La commande `ps`

La commande linux `ps` (**P**rocess **S**tatus) permet de lister les processus s'exécutant sur le système. Elle possède de nombreuses options qui permettent d'obetenir des informations plus ou moins détaillées sur les processus.
Voici un exemple de sortie de la commande `ps` :




```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 41127 -      18:36 ?        00:00:01 /sbin/init splash
1 S root           2       0  0  80   0 -     0 -      18:36 ?        00:00:00 [kthreadd]
1 I root           3       2  0  60 -20 -     0 -      18:36 ?        00:00:00 [rcu_gp]
1 I root           4       2  0  60 -20 -     0 -      18:36 ?        00:00:00 [rcu_par_gp]
1 I root           5       2  0  80   0 -     0 -      18:36 ?        00:00:00 [kworker/0:0-cgroup_destroy
1 I root           6       2  0  60 -20 -     0 -      18:36 ?        00:00:00 [kworker/0:0H-events_highpr
…
0 S nsi         2199    2049  0  80   0 -  4904 do_wai 18:49 pts/2    00:00:00 bash
4 S nsi         2251    1975  0  80   0 - 604641 do_pol 18:49 ?       00:00:00 /usr/lib/firefox/firefox -c
0 S nsi         2457    1025  0  80   0 - 661698 do_pol 19:01 ?       00:00:00 gjs /usr/share/gnome-shell/
1 I root        2503       2  0  80   0 -     0 -      19:01 ?        00:00:00 [kworker/0:2-events]
1 I root        2522       2  0  80   0 -     0 -      19:07 ?        00:00:00 [kworker/u2:1-ext4-rsv-conv
1 I root        2547       2  0  80   0 -     0 -      19:13 ?        00:00:00 [kworker/u2:0-events_unboun
1 I root        2559       2  0  80   0 -     0 -      19:18 ?        00:00:00 [kworker/u2:2-ext4-rsv-conv
0 S nsi         2587    2199  0  80   0 -  6925 do_pol 19:23 pts/2    00:00:00 xclock
4 R nsi         2588    2199  0  80   0 -  5296 -      19:24 pts/2    00:00:00 ps -A -fl
```

Application 2 :

1) Dans le terminal, taper la commande `ps`. Qu'observez-vous ?

Pour obtenir davantage d'informations, il faut utiliser des options de la commande `ps`. Dans linux, la commande qui permet d'obtenir de la documentation sur une commande est `man`. Alternativement, on peut utiliser l'option `--help` (ou `--aide` dans la version francisée) pour obtenir de l'aide.

2) Utiliser la documentation pour afficher la liste de tous les processus s'exécutant sur la machine (pas seulement ceux de l'utilisateur courant et pas seulement ceux liés au terminal actif).

Une variante de la commande `ps` est `pstree` qui affiche les processus sous la forme d'une arborescence.

3) Lancer le navigateur Firefox (attention le lancement est assez long). Afficher l'arborescence des processus. Combien de processus sont lancées par Firefox ?

c. La commande `top`.

Cette commande permet d'afficher en temps réel les processus du système. Il faut utiliser la touche `q` ou `Ctrl-C` pour revenir à l'invite de commande.

Application 3 :

Dans le terminal, taper la commande `top`. Qu'observez-vous ? Dans quel ordre les processus sont-ils affichés ?

d. Signaux

La commande `kill` permet d'envoyer différents types de signaux à un processus dont on connaît l'identifiant (PID). Malgré son nom, et même si c'est son usage principal, elle ne sert pas seulement à « tuer » un processus. Les signaux les plus courants sont `SIGTERM` et `SIGKILL`, qui servent à terminer un processus. D'autres signaux fréquents sont `SIGSTOP` (suspension) et `SIGCONT` (reprise).
La syntaxe d'envoi d'un signal à un processus est : `kill -`*`signal`*` PID`, où *`signal`* est un numéro ou un nom de signal (le signal par défaut est `SIGTERM`).

Application 4 :

1) La liste des signaux que l'on peut envoyer aux processus s'obtient grâce à l'option `-l` de `kill`. Repérer les numéros des signaux mentionnés ci-dessus.
2) Tester les signaux mentionnés ci-dessus sur le processus xclock préalablement lancé (utiliser l'une des commandes précédentes pour accéder à son PID).
3) Lancer un second terminal. Repérer son identifiant de processus, puis testez les signaux `SIGTERM` et `SIGKILL` sur ce processus. Que constate-t-on ?

Ce comportement illustre le fait que dans certains cas, être « poli » ne suffit pas : `SIGTERM` demande au processus de s'arrêter (ce qu'il peut refuser), tandis que `SIGKILL` demande au système de le "tuer" (utile aussi si un programme "ne répond pas"...).

4) Repérer parmi les processus actifs un processus dont vous n'êtes pas propriétaire, et tenter de le stopper à l'aide du signal `SIGSTOP`. Que peut-on en déduire ?

IV - Problèmes liés à l'exécution concurrente

Utilisons maintenant python pour mettre en lumière certains problèmes liés à l'exécution concurrente de plusieurs programmes.
Pour cela on utilisera la bibliothèque `threading` qui permet de créer facilement des fils d'exécution dans un programme python avec la classe `threading.Thread` qui peut prendre en argument `target` une fonction à exécuter en parallèle et dans `args` un tableau des arguments à fournir à la fonction. Il suffit ensuite de lancer l'exécution en appelant la méthode `start()` de l'objet `Thread`.

Exemple :
`mon_fil = threading.Thread(target=ma_fonction, args=["Toto", 14])` créera un fil d'exécution pour la fonction `mafonction("Toto",14)`. Il suffira ensuite de faire `mon_fil.start()` pour lancer l'exécution en parallèle de la fonction.

1) Accès à la même ressource

On étudie le programme « Accès à la même ressource.py ».

Application 5 :

1) Ouvrir le programme et AVANT de l'exécuter, essayer d'expliquer ce qu'il fait.
2) Modifier le programme pour que le numéro du fil d'exécution s'affiche sans saut de ligne et sans indiquer la fin du fil d'exécution afin d'obtenir une sortie comme :
   `12212033231030232121232013032103130011 00`
3) Modifier le programme pour que le programme principal affiche un message lorsque tous les fils d'exécution ont fini. Cela suppose que chaque fil doit indiquer au programme principal qu'il a fini.
4) Supprimer le temps d'attente aléatoire et exécuter le programme plusieurs fois de suite. Obtient-on toujours la même sortie ?

Application 6 :

1) Ouvrir le programme « Compteur.py » et AVANT de l'exécuter, essayer d'expliquer ce qu'il fait.
2) Combien devrait valoir la varaible compteur à la fin de l'exécution ? Exécuter le programme plusieurs fois de suite et comparer à votre prédiction. Comment expliquer que la valeur du compteur change d'une exécution à l'autre ?

Pour résoudre ce problème, on va faire en sorte que la **portion de code critique** (celle qui accède à la ressource partagée) ne soit exécutée que par un seul fil d'exécution à la fois. Pour cela on utilise un **verrou** exclusif (*lock* en anglais) qu'un fil d'exécution devra acquérir avant sa portion critique et libérer à la fin de celle-ci. Ainsi le code critique ne sera exécuté que par un seul fil à la fois car si un deuxième arrivait à vouloir exécuter le même code, il serait bloqué au moment d'acquérir le verrou car celui-ci est déjà verrouillé par un autre fil d'exécution. Pour que cet autre fil d'exécution puisse lui aussi rentrer dans la portion de code critique il devra attendre que le fil actuel libère le verrou et qu'il puisse l'acquérir à son tour.

La gestion des verrous se fait assez simplement avec le module threading :

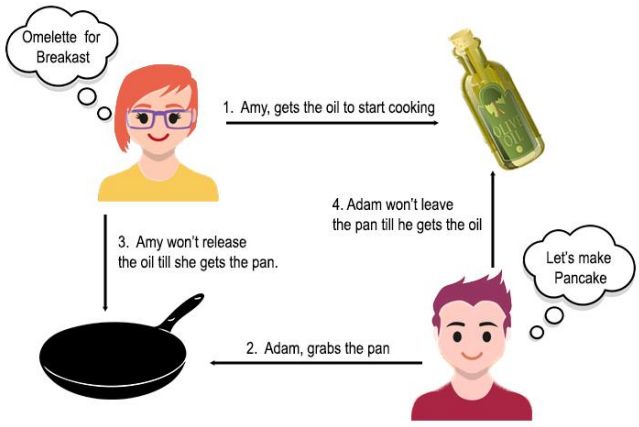
Syntaxe	Effet
`verrou = threading.Lock()`	permet de créer un verrou (initialement déverrouillé)
`verrou.acquire()`	permet de verrouiller le verrou en se l'appropriant (si le verrou est déjà acquis par un autre fil d'exécution, cette instruction met le thread en attente jusqu'à ce que le verrou se libère)
`verrou.release()`	permet de libérer le verrou qui est alors disponible pour les autres fils d'exécution

3) Modifier le code du programme « Compteur.py » pour que le comptage s'effectue correctement.

2) Interblocage

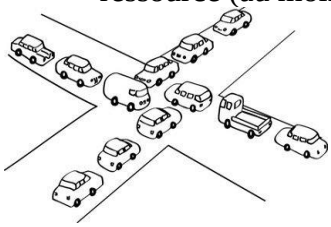
Un **interblocage** (ou étreinte fatale, *deadlock* en anglais) est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque des processus concurrents s'attendent mutuellement. Un processus peut aussi s'attendre lui-même. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique provoquant le blocage du système.

Remarque : Lorsqu'un processus attend indéfiniment une ressource on dit qu'il est dans une situation de famine (*starving* en anglais).



Il existe 4 conditions nécessaires à l'interblocage (appelées *conditions de Coffman*) :

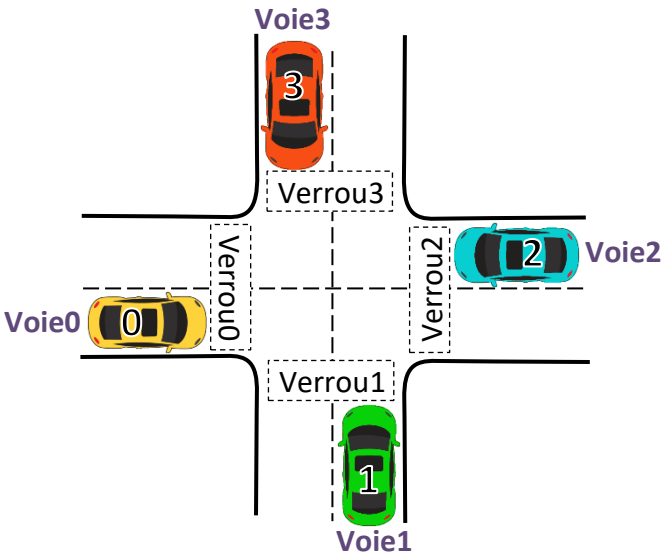
1. **Exclusion mutuelle** : Les ressources ne sont pas partageables, un seul processus à la fois peut utiliser la ressource (au moins une ressource doit être en accès exclusif).



2. **Possession et attente** : Les processus qui détiennent des ressources peuvent en demander d'autres.
3. **Sans préemption** : Les ressources ne sont pas préemptibles c'est-à-dire que les libérations sont faites volontairement par les processus. On ne peut pas forcer un processus à rendre une ressource.
4. **Attente circulaire** : Il doit exister un ensemble de processus P tel que P_i attend une ressource possédée par P_{i+1}.

Application 7 :

On va modéliser ici le fonctionnement d'un carrefour à 4 voies. Les voies seront numérotées de 0 à 3 de même que les véhicules (le véhicule *n* arrive par la voie *n* et va toujours tout droit). Lorsqu'une voiture veut s'engager sur le carrefour, il faut qu'elle laisse la priorité à droite et que sa voie de passage soit dégagée.



Pour simuler ce comportement, on va créer 4 verrous, un sur chacune des voies de circulation. Lorsqu'un véhicule veut passer, il doit d'abord acquérir le verrou de la voie sur sa gauche (afin que son voisin de gauche soit obligé de respecter sa priorité) puis le verrou sur sa voie avant de passer puis de libérer les verrous derrière son passage.

A chaque fois qu'un véhicule passe, on affiche le numéro de la voie traversée, ce qui permet de suivre le traffic des véhicules et on attend 0,1 s ce qui permet de simuler le temps que met la voiture à traverser le carrefour.

Chaque voie correspond à un fil d'exécution séparé qui reçoit en argument le numéro de voie ainsi que le nombre de passage de carrefour à réaliser.

Si besoin, on pourra s'aider du fichier « Interblocage_trame.py ».

1) Créer un programme qui simule le passage de 2 véhicules par voie et exécutez-le plusieurs fois. Passer si besoin à 3, 4, 5, … véhicules par voie jusqu'à obtenir un interblocage.
2) Montrer en utilisant les conditions de Coffman qu'un interblocage peut avoir lieu dans cette situation.

Références :

Commande `ps` : https://pubs.opengroup.org/onlinepubs/009695399/utilities/ps.html
Vidéo SoC et interblocage : https://www.youtube.com/watch?v=vKxpBDsKWO4

Exercices – Gestion des processus et des ressources

Exercice 1 : Etat d'un processus

1) On clique dans la fenêtre de la calculatrice, puis dans le terminal on exécute la commande `top`, quel est l'état du processus de la calculatrice ? R ou S ?
2) On exécute le programme Python suivant, quel est l'état du processus R ou S, avant d'entrer le nom au clavier et après avoir entré le nom ?

```
nom = input ( "Entrez votre nom " )
print ( "Bonjour " + nom )
```

3) On exécute un tri par sélection sur un tableau de 10^4 nombres. Le temps de traitement est à peu près 20 s. immédiatement après avoir lancé le tri, dans le terminal on exécute la commande `top`, quel est l'état du processus exécutant le tri ? R ou S ?

Exercice 2 : Jeu à pions

On écrit un programme pour un jeu où deux joueurs s'affrontent. Au cours du jeu, un pion peut apparaître et dans ce cas chacun des joueurs peut le prendre et le placer sur le plateau de jeu. Les pions apparaissent toujours un par un et l'apparition d'un pion est toujours suivie d'une phase où plus aucun pion ne peut apparaître.
Le programme principal lance deux fils d'exécutions J1 et J2 qui ont le même code et pilotent chacun les actions d'un joueur. On donne ci-contre un extrait du programme de J1 et J2 :

```
global pion_disponible   # True si un pion est disponible
…
    if pion_disponible:
        placer_pion()
        pion_disponible = False
    else :
…
```

4) Quel(s) problème(s) risque-il d'y avoir lors de l'exécution de J1 et J2 ?
5) Proposer une ou plusieurs solutions pour ce(s) problème(s).

Exercice 3 : Interblocage à LED

Considérons un petit système embarqué : un petit ordinateur relié à trois LED A, B et C. Une LED peut être éteinte ou allumée et on peut configurer sa couleur. On dispose de trois programmes qui affichent des signaux lumineux en faisant clignoter les LED. Chaque programme possède une LED primaire et une LED secondaire.

- Le `programme_1` affiche ses signaux sur A (primaire) et B (secondaire) en vert
- Le `programme_2` affiche ses signaux sur B (primaire) et C (secondaire) en orange
- Le `programme_3` affiche ses signaux sur C (primaire) et A (secondaire) en rouge

Comme les LED ne peuvent pas être configurées en même temps, l'ordinateur contient deux fonctions :

- `acquerirLED(nom`
- `relacherLED(nom)`

On suppose que chacun des trois programmes effectuent les 3 actions suivantes en boucle :
1.acquérir sa LED primaire
2.acquérir sa LED secondaire
3.configurer les couleurs des LED
4.emmetre des signaux
5.rendre la LED secondaire
6.rendre la LED primaire

1) Montrer qu'il existe un entrelacement d'exécutions qui mène à un interblocage.
2) Écrire un programme Python simulant cette situation.

Exercice 4 : Le diner des philosophes (source Wikipedia)

La situation est la suivante :

- cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ;
- chacun des philosophes a devant lui un plat de spaghettis ;
- à gauche de chaque plat de spaghettis se trouve une fourchette.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé ;
- être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :

- quand un philosophe a faim, il va se mettre dans l'état « affamé » et attendre que les fourchettes soient libres ;
- pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ;
- si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

1) Décrire une situation d'interblocage, en détaillant les conditions de Coffman.
2) Que faire si un philosophe meurt de faim alors qu'il a une fourchette en main (i.e. un processus se crashe alors qu'il utilise une ressource) ? La question est assez rhétorique, elle est là juste pour que vous réalisiez le problème dans ce cas.
3) On propose une solution, basée sur la règle suivante : « un philosophe ayant une seule fourchette la repose après 10 minutes, et attend 10 minutes avant de la reprendre ». Cette règle permet-elle d'éviter l'interblocage ? Justifier.
4) Une autre solution est basée sur la hiérarchisation des ressources. Les fourchettes sont numérotées de 1 à 5, pas forcément dans l'ordre de leur emplacement sur la table. Les philosophes connaissent les numéros des fourchettes dont ils ont besoin pour manger. Un philosophe prendra d'abord la fourchette de numéro le plus bas, avant de prendre celle de numéro le plus haut. Cette méthode permet-elle d'éviter l'interblocage ? Justifier
5) Proposer une ou plusieurs méthodes pour solutionner le problème.

Exercice 5 : Graphe et interblocages

Sept processus Pi sont dans la situation suivante par rapport aux ressources Ri :

• P1 a obtenu R1 et demande R2 ;
• P2 demande R3 et n'a obtenu aucune ressource tout comme P3 qui demande R2 ;
• P4 a obtenu R2 et R4 et demande R3 ;
• P5 a obtenu R3 et demande R5 ;
• P6 a obtenu R6 et demande R2 ;
• P7 a obtenu R5 et demande R2. On voudrait savoir s'il y a interblocage.

1) Construire un graphe orienté où les sommets sont les processus et les ressources, et où :

• la présence de l'arc Ri → Pj signifie que le processus Pj a obtenu la ressource Ri ;
• la présence de l'arc Pj → Ri signifie que le processus Pj demande la ressource Ri.

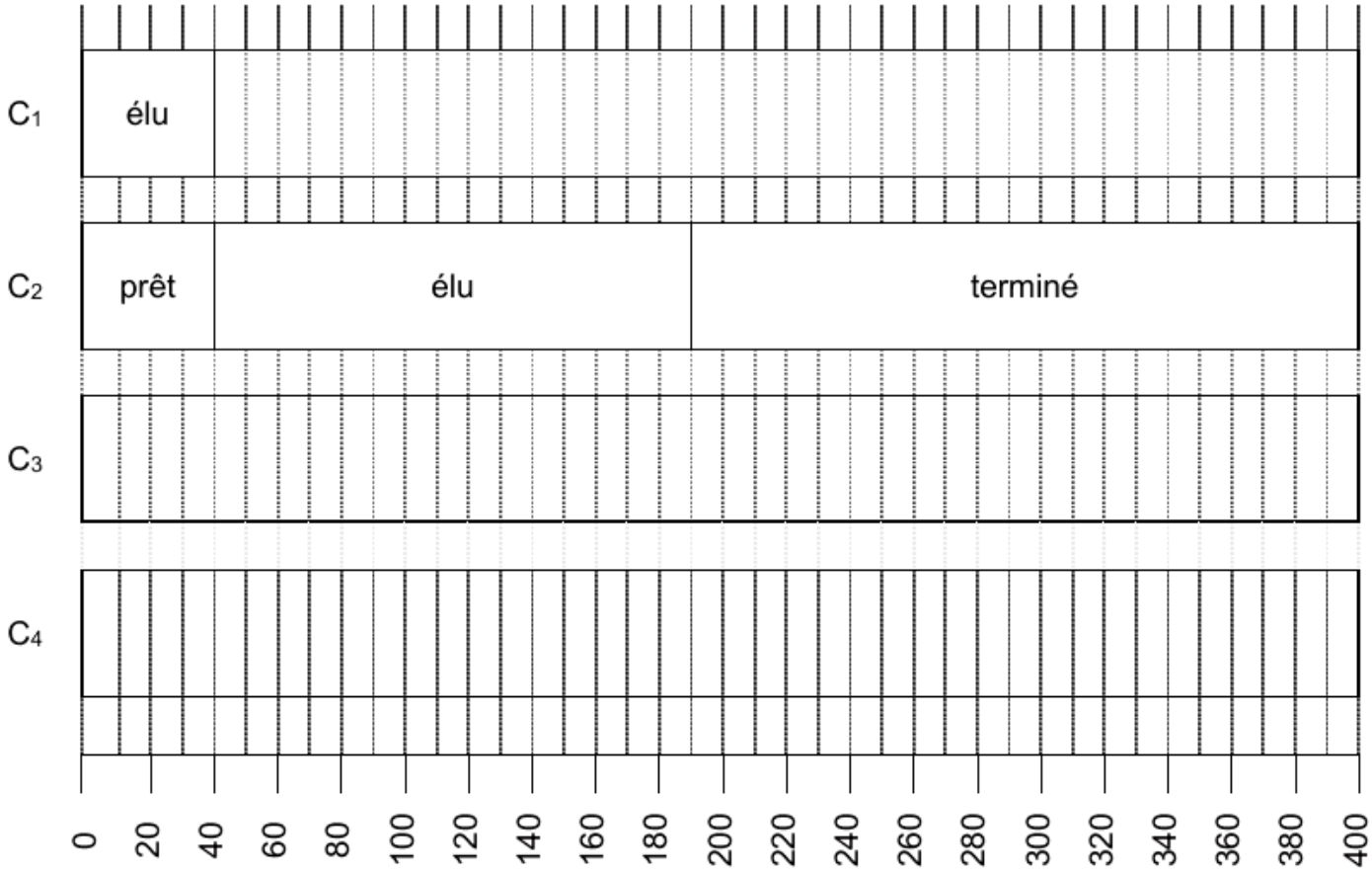
2) Il y a interblocage lorsque des cycles sont présents dans le graphe. Chercher ces cycles afin de déterminer s'il y a bien interblocage.

Exercice 6 : Sujet de bac

Cet exercice est l'exercice n°2 du sujet n°2 du BAC 2021 métropole en candidat libre.

1) Les états possibles d'un processus sont : *prêt*, *élu*, *terminé* et *bloqué*.
   a) Expliquer à quoi correspond l'état *élu*.
   b) Proposer un schéma illustrant les passages entre les différents états.
2) On suppose que quatre processus C_1, C_2, C_3 et C_4 sont créés sur un ordinateur, et qu'aucun autre processus n'est lancé sur celui-ci, ni préalablement ni pendant l'exécution des quatre processus.
   L'ordonnanceur, pour exécuter les différents processus prêts, les place dans une structure de données de type file. Un processus prêt est enfilé et un processus élu est défilé.
   a) Parmi les propositions suivantes, recopier celle qui décrit le fonctionnement des entrées/sorties dans une file :
      - Premier entré, dernier sorti
      - Premier entré, premier sorti
      - Dernier entré, premier sorti
   b) On suppose que les quatre processus arrivent dans la file et y sont placés dans l'ordre C_1, C_2, C_3 et C_4.
      - Les temps d'exécution totaux de C_1, C_2, C_3 et C_4 sont respectivement 100 ms, 150 ms, 80 ms et 60 ms.
      - Après 40 ms d'exécution, le processus C_1 demande une opération d'écriture disque, opération qui dure 200 ms. Pendant cette opération d'écriture, le processus C_1 passe à l'état bloqué.
      - Après 20 ms d'exécution, le processus C_3 demande une opération d'écriture disque, opération qui dure 10 ms. Pendant cette opération d'écriture, le processus C_3 passe à l'état bloqué.

   Sur la frise chronologique donnée en annexe (à rendre avec la copie), les états du processus C_2 sont donnés.
   Compléter la frise avec les états des processus C_1, C_3 et C_4.



3) On trouvera ci-dessous deux programmes rédigés en pseudo-code.

Verrouiller un fichier signifie que le programme demande un accès exclusif au fichier et l'obtient si le fichier est disponible.

Programme 1	Programme 2
Verrouiller fichier_1	Verrouiller fichier_2
Calculs sur fichier_1	Verrouiller fichier_1
Verrouiller fichier_2	Calculs sur fichier_1
Calculs sur fichier_1	Calculs sur fichier_2
Calculs sur fichier_2	Déverrouiller fichier_1
Calculs sur fichier_1	Déverrouiller fichier_2
Déverrouiller fichier_2	
Déverrouiller fichier_1	

a) En supposant que les processus correspondant à ces programmes s'exécutent simultanément (exécution concurrente), expliquer le problème qui peut être rencontré.
b) Proposer une modification du programme 2 permettant d'éviter ce problème.

Exercice 7 : Autre sujet de bac

Cet exercice est la première partie de l'exercice 3 du sujet 1 candidat libre Métropole 2021.
La commande UNIX `ps` présente un cliché instantané des processus en cours d'exécution.
Avec l'option `-eo pid,ppid,stat,command`, cette commande affiche dans l'ordre l'identifiant du processus `PID` (process identifier), le `PPID` (parent process identifier), l'état `STAT` et le nom de la commande à l'origine du processus.
Les valeurs du champ `STAT` indique l'état des processus :

- `R` : processus en cours d'exécution
- `S` : processus endormi

Sur un ordinateur, on exécute la commande `ps -eo pid,ppid,stat,command` et on obtient un affichage dont on donne ci-dessous un extrait.

```
$ ps -eo pid,ppid,stat,command

PID   PPID  STAT  COMMAND
1     0     Ss    /sbin/init
....  ....  ..    ...
1912  1908  Ss    Bash
2014  1912  Ss    Bash
1920  1747  Sl    Gedit
2013  1912  Ss    Bash
2091  1593  Sl    /usr/lib/firefox/firefox
5437  1912  Sl    python programme1.py
5440  2013  R     python programme2.py
5450  1912  R+    ps -eo pid,ppid,stat,command
```

À l'aide de cet affichage, répondre aux questions ci-dessous.
1) Quel est le nom de la première commande exécutée par le système d'exploitation lors du démarrage ?
2) Quels sont les identifiants des processus actifs sur cet ordinateur au moment de l'appel de la commande `ps` ? Justifier la réponse.
3) Depuis quelle application a-t-on exécuté la commande `ps` ? Donner les autres commandes qui ont été exécutées à partir de cette application.
4) Expliquer l'ordre dans lequel les deux commandes `python programme1.py` et `python programme2.py` ont été exécutées.
5) Peut-on prédire que l'une des deux commandes `python programme1.py` et `python programme2.py` finira avant l'autre ?