AES_Processus

I/ Notion de processus

1/ Définition

Il faut distinguer un *programme* de son *exécution* : le lancement d'un programme entraîne des lectures écritures de registre et d'une partie de la mémoire. D'ailleurs un même programme peut être exécuté plusieurs fois sur une même machine au même moment en occupant des espaces mémoires différents.
Un **processus** représente une **instance d'exécution** d'un programme dans une machine donnée.

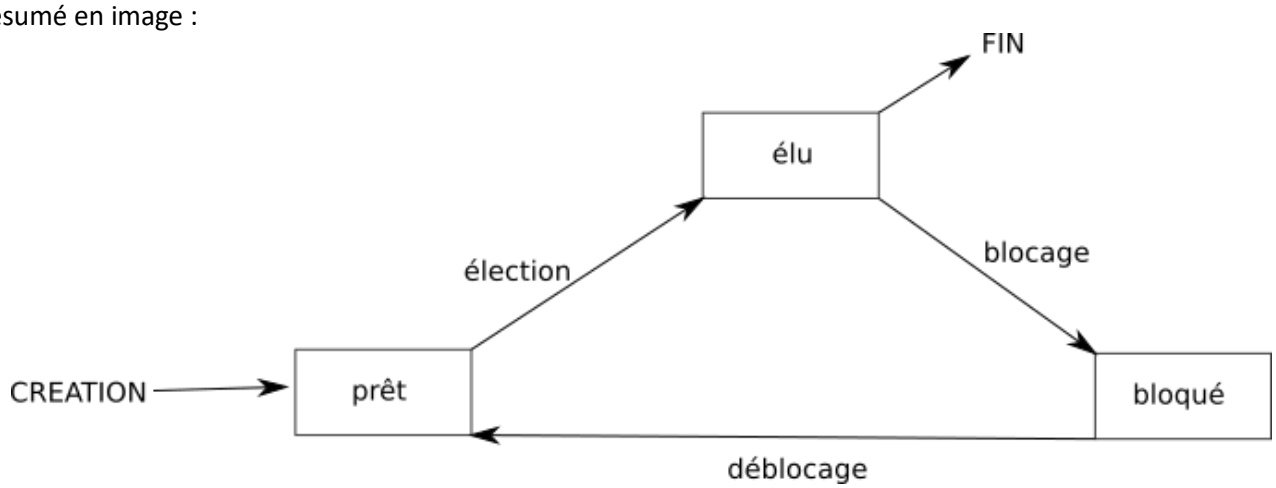
2/ Etats d'un processus

A savoir : au cours de son existence dans la machine, un **processus** peut se retrouver dans **différents états** qui sont résumées ci-dessous :

- un processus est créé et se trouve alors dans l'**état prêt** : il attend de pouvoir avoir accès au processeur.
- Le processus obtient l'accès au processeur : il passe alors dans l'**état élu**.
- Alors qu'il est élu, le processus peut avoir besoin d'attendre une ressource quelconque comme une ressource en mémoire. Il doit alors quitter momentanément le processeur pour que celui-ci puisse être utilisé pour d'autres tâches (le processeur ne doit pas attendre !). Le processus passe alors dans l'**état bloqué**.
- Le processus a obtenu la ressource attendue mais s'est fait prendre sa place dans le processeur par un autre processus : il se met donc en attente et **repasse** à **état prêt**.

Remarque : un processus ne pourra se terminer que s'il est déjà dans l'état élu sauf anomalie.

Le résumé en image :



Source : pixies.fr

Il est fondamental de bien comprendre que le "chef d'orchestre" qui attribue aux processus leur état "élu", "bloqué" ou "prêt" est le **système d'exploitation** (***O**perating **S**ystem en anglais*). On dit que le système d'exploitation gère **l'ordonnancement** des processus (un processus sera prioritaire sur un autre...).

A savoir : un processus qui utilise une ressource doit la "libérer" une fois qu'il a fini de l'utiliser afin de la rendre disponible pour les autres processus. Pour libérer une ressource, un processus doit obligatoirement être dans un état "élu".

Un processus peut être démarré par un utilisateur par l'intermédiaire d'un périphérique ou bien par un autre processus : les **applications** des utilisateurs sont des ensembles de processus plus ou moins complexes.

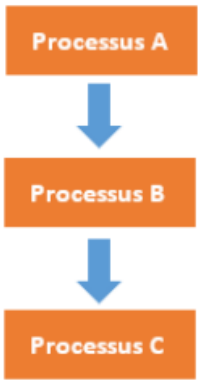
Un **processus** peut s'arrêter de plusieurs manières :

- Arrêt normal (volontaire).
- Arrêt pour erreur (volontaire).
- Arrêt pour erreur fatale (involontaire).
- Le processus est arrêté par un autre processus (involontaire).

La plupart des systèmes offrent la distinction entre processus lourd, qui sont a priori complètement isolés les uns des autres, et processus légers (*Threads en anglais*), qui ont un espace mémoire (et d'autres ressources) en commun. Dans le cas de processus comportant plusieurs processus légers (ou suivant l'expression souvent utilisée *multithread*) il existe un état du processeur (un contexte d'exécution) distinct pour chaque processus léger.

3/ Création d'un processus

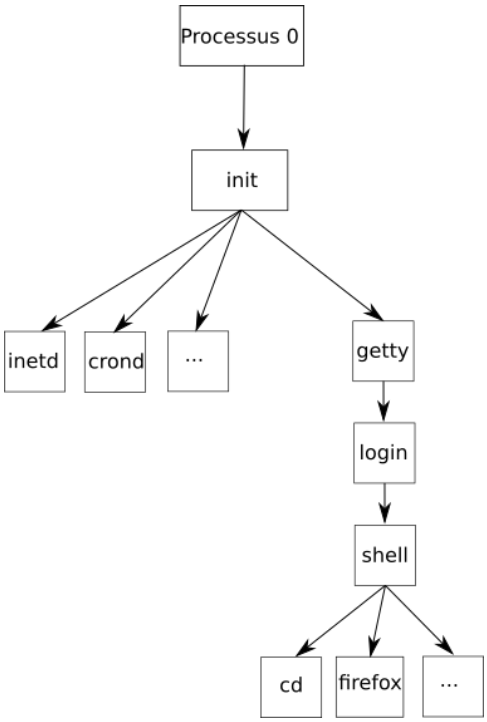
Un **processus peut créer un** ou **plusieurs processus** à l'aide d'une *commande système*.
Soit un processus A qui crée un processus B. On dira que A est le *père* de B et que B est le fils de A. B peut, à son tour créé un processus C (B sera le *père* de C et C le *fils* de B). On peut modéliser ces relations père/fils par une structure arborescente (voir le schéma à droite).



Sous un système d'exploitation comme Linux, au moment du démarrage de l'ordinateur un tout premier processus (appelé processus 0 ou encore *Swapper*) est créé à partir de "rien" (il n'est le fils d'aucun processus).

Ensuite, ce processus 0 crée un processus souvent appelé "*init*" ("*init*" est donc le fils du processus 0). À partir de "*init*", les processus nécessaires au bon fonctionnement du système sont créés (par exemple les processus "*crond*", "*inetd*", "*getty*" etc.). Puis d'autres processus sont créés à partir des fils de "*init*"...

On peut résumer tout cela avec le schéma à droite :



Tous ces noms de processus ne sont bien sûr pas à retenir, ils sont juste donnés pour l'exemple. Il est simplement nécessaire d'avoir compris les notions de *processus père*, *processus fils* et *structure arborescente*.

4/ Identification d'un processus

Chaque processus possède un identifiant appelé **PID** (***P**rocess **Id**entification* en anglais), ce PID est un nombre entier. Le premier processus créé au démarrage du système à pour PID 0, le second 1, le troisième 2...

Le système d'exploitation utilise un compteur qui est incrémenté de 1 à chaque création de processus, le système utilise ce compteur pour attribuer les PID aux processus. Chaque processus possède aussi un **PPID** (***P**arent **P**rocess **Id**entification* en anglais). Ce **PPID** permet de connaitre le **processus parent d'un processus** (par exemple le processus "*init*" vu ci-dessus à un **PID** de **1** et un **PPID** de **0**).

Remarque : le processus 0 ne possède pas de PPID (c'est le seul dans cette situation).

II/ Ordonnancement des processus

1/ Définition

Dans un système multi-utilisateur à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution. Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. C'est **l'ordonnanceur** (*scheduler* en anglais) qui s'acquitte de cette tâche.
Le **répartiteur** (*dispatcher* en anglais) alloue quant à lui un processeur à des buts dans le cas d'une architecture multiprocesseur.

2/ Objectifs d'un ordonnanceur

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- s'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur
- minimiser le temps de réponse
- utiliser le processeur à 100%
- utiliser d'une manière équilibrée les ressources
- prendre en compte les priorités
- être prédictible.

Ces objectifs sont parfois complémentaires, parfois contradictoires : augmenter la performance par rapport à l'un d'entre eux peut se faire au détriment d'un autre. Il est impossible de créer un algorithme qui optimise tous les critères de façon simultanée.

Il existe plusieurs politiques d'ordonnancement dont le choix va dépendre des objectifs du système voici quelques exemples :

- **Premier arrivé, premier servi** : simple, mais peu adapté à la plupart des situations.
- **Plus cours d'abord** : très efficace, mais il est la plupart du temps impossible de connaître à l'avance le temps d'exécution d'un processus.
- **Priorité** : le système alloue un niveau de priorité au processus, cependant des processus de faible priorité peuvent ne jamais être lus.
- **Tourniquet** : un quantum de temps est alloué à chaque processus. Si le processus n'est pas terminé au bout de ce temps il est mis en bout de fil en état *prêt*.

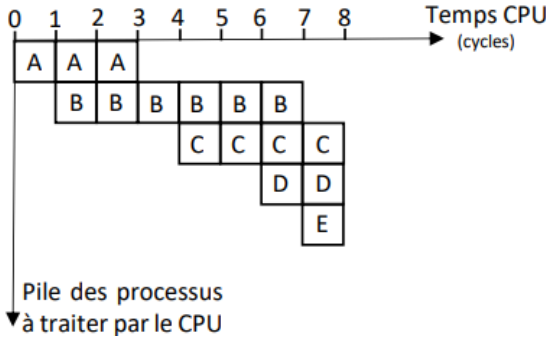
Un exemple de gestion de processus :
Considérons par exemple un lot de quatre processus notés A, B, C, D dont les temps respectifs d'exécution sont , t_A , t_B, t_C et t_D . Le premier processus se termine au bout du temps t_A ; le deuxième processus se termine au bout du temps $t_A + t_B$; le troisième processus se termine au bout du temps $t_A + t_B + t_C$; le quatrième processus se termine au bout du temps $t_A + t_B + t_C + t_D$.

Le temps moyen de séjour noté est $T_{moy} = (4\ t_A + 3\ t_B + 2\ t_B + t_D) / 4$
On obtient le meilleur temps de séjour pour $t_A \leq t_B \leq t_C \leq t_D$. Toutefois, l'ordonnancement du plus court d'abord est optimal que si les travaux sont disponibles simultanément.

Processus	Temps d'exécution	Temps d'arrivage
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

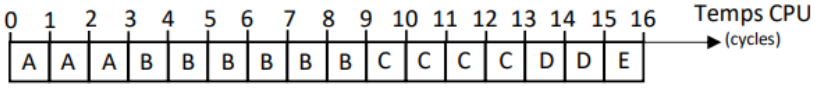
Considérons cinq processus notés A, B, C, D et E, dont les temps d'exécution et leurs arrivages respectifs sont donnés dans le tableau ci-contre.

On peut également représenter le tableau précédent de la manière suivante afin de faciliter la compréhension de l'ordonnancement des processus :



■ **Algorithme "Premier arrivé premier servi" : First-Come First-Served (FCFS)**

Schéma d'exécution :



Au temps 0, seulement le processus A est dans le système et il s'exécute. Au temps 1 le processus B arrive mais il doit attendre que A se termine car il lui reste encore 2 unités de temps à effectuer. Ensuite B s'exécute pendant 4 unités de temps. Au temps 4, 6, et 7 les processus C, D et E arrivent mais B a encore 2 unités de temps. Une fois que B a terminé, C, D et E entrent dans le système dans l'ordre (on suppose qu'il y a aucun blocage).

Le **temps de séjour** pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison. Ainsi :

Processus	Temps de sejour
A	3-0 = 3
B	9-1 = 8
C	13-4 = 9
D	15-6 = 9
E	16-7 = 9

Le **temps moyen de séjour** est : (3 + 8 + 9 + 9 + 9) / 5 = 38 / 5 = 7,6.

Le **temps d'attente** est calculé soustrayant le temps d'exécution du temps de séjour :

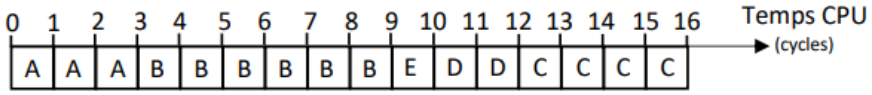
Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
C	9-4 = 5
D	9-2 = 7
E	9-1 = 8

Le **temps moyen d'attente** est : (0 + 2 + 5 + 7 + 8) / 5 = 23 /5 = 4,4

Il y a cinq tâches exécutées dans 16 unités de temps, alors 16 / 5 = 3,2 unités de temps par processus.

■ **Algorithme "Le plus court d'abord" : Short Job First (SJF)**

Schéma d'exécution :



Le processeur traite d'abord comme précédemment les processus A puis B. Le processus B s'achève à la date t = 9 qui est supérieure au temps d'arrivage des processus C, D et E : ceux-ci sont donc déjà présents dans la pile des processus en attente de traitement par le processeur. Celui-ci choisi d'abord d'exécuter le processus le plus court des 3 c'est-à-dire E conformément à l'algorithme SJF. Puis selon la même logique, il exécute le processus D et enfin le processus C.

Temps de séjour :

Processus	Temps de séjour
A	3-0 = 3
B	9-1 = 8
E	10-7 = 3
D	12-6 = 6
C	16-4 = 12

Temps moyen de séjour : (3 + 8 + 3 + 6 + 12) / 5 = 32 / 5 = 6,4

Temps d'attente :

Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
E	3-1 = 2
D	6-2 = 4
C	12-4 = 8

Temps moyen d'attente : (0 + 2 + 2 + 4 + 8) / 5 = 16 / 5 = 3,2

On observe que dans ce cas de figure l'algorithme *Short Job First* (SJF) est plus performant que l'algorithme *FirstCome First-Served* (FCFS).

Source : LGT Saint-Exupéry, Mantes-la-Jolie

III/ Commandes Linux de la gestion des processus

1/ La commande *ps*

La commande ***ps*** est utilisée pour afficher des informations sur un processus, l'option ***u*** permet de préciser le propriétaire et l'option ***l*** permet un affichage avec plus d'attributs :

```
$ ps -lu moi
F S UID  PID   PPID C PRI  NI  ADDR SZ  WCHAN      TTY TIME     CMD
0 S 1000 9412  3048 0 80   0   -   1156 wait       ? 00:00:00 sh
0 S 1000 9413  9412 0 80   0   - 178754 poll_s     ? 00:00:07 tilda
0 S 1000 9423  9413 0 80   0   -   7537 wait   pts/2 00:00:00 bash
4 R 1000 28867 9423 0 80   0   -  11140 -    pts/2 00:00:00 ps
```

- La colonne S indique l'état du processus : S pour *stopped*, R pour *running* et Z pour *zombie*.
- PID est le processus identifier : un identifiant sous forme d'entier donné par le système.
- PPID et le parent processus identifier qui donne l'identifiant du parent qui a engendré le processus.
- CMD est le nom de la commande.

2/ La commande *kill*

Il n'est parfois pas possible de fermer un processus graphique en cliquant par exemple sur la croix prévue pourtant à cet effet. On peut utiliser la commande ***kill*** si on connaît le PID du processus à « tuer » qui peut être obtenu avec l'option *C* de *ps* :

```
$ ps -lC codium
F S UID  PID  PPID C  PRI NI ADDR SZ WCHAN TTY TIME     CMD
0 S 1000 3091 1    3  80  0  - 218751 poll_s ? 00:00:01 codium
0 S 1000 3093 3091 0  80  0  -  94034 poll_s ? 00:00:00 codium
0 S 1000 3116 3091 2  80  0  - 134596 poll_s ? 00:00:01 codium
0 S 1000 3129 3091 0  80  0  - 111394 futex_ ? 00:00:00 codium
0 S 1000 3138 3091 11 80  0  - 276851 futex_ ? 00:00:05 codium
0 S 1000 3162 3138 3  80  0  - 173887 ep_pol ? 00:00:01 codium
0 S 1000 3192 3091 1  80  0  - 182402 futex_ ? 00:00:00 codium
```

On découvre une généalogie des processus. 3091 est le parent de tous les autres c'est lui qu'il faut stopper :

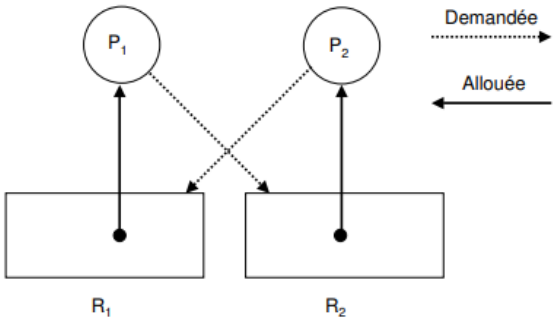
```
$ kill 3091
$ ps -lC codium
F S  UID  PID  PPID  C  PRI  NI  ADDR  SZ  WCHAN  TTY  TIME  CMD
```

<u>A savoir</u> : les commandes ps et kill sont à connaître.

IV/ Situation d'interblocage

1/ Définition et exemples

Des problèmes peuvent survenir, lorsque les processus obtiennent des accès exclusifs aux ressources. Par exemple, un processus P1 détient une ressource R1 et attend une autre ressource R2 qui est utilisée par un autre processus P2 ; le processus P2 détient la ressource R2 et attend la ressource R1. On a une situation d'**interblocage** (*deadlock en anglais*) car P1 attend R2 et P2 attend R1. Les deux processus vont s'attendre indéfiniment.



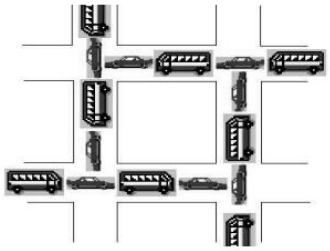
En général, un ensemble de processus est en interblocage si chaque processus attend la libération d'une ressource qui est allouée à un autre processus de l'ensemble. Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres. Ils attendront tous indéfiniment.

Rappel : un processus ne peut libérer une ressource que s'il est dans l'état « élu ».

Quelques exemples d'interblocage :

Circulation routière

On considère deux routes à double sens qui se croisent comme dans la figure (à droite), où la circulation est impossible. Un problème d'**interblocage** y est présent.



Accès aux périphériques

On suppose que deux processus A et B veulent imprimer, en utilisant la même imprimante, un fichier stocké sur une bande magnétique. La taille de ce fichier est supérieure à la capacité du disque. Chaque processus a besoin d'un accès exclusif au dérouleur et à l'imprimante simultanément. On a une situation d'interblocage si :
– Le processus A utilise l'imprimante et demande l'accès au dérouleur.
– Le processus B détient le dérouleur de bande et demande l'imprimante.

Accès à une base de données

On suppose deux processus A et B qui demandent des accès exclusifs aux enregistrements d'une base de données. On arrive à une situation d'interblocage si :
– Le processus A a verrouillé l'enregistrement R1 et demande l'accès à l'enregistrement R2 .
– Le processus B a verrouillé l'enregistrement R2 et demande l'accès à l'enregistrement R1.

2/ Conditions d'une situation d'interblocage

Pour qu'une situation d'interblocage ait lieu, les quatre conditions suivantes doivent être remplies (Conditions de Coffman) :
– **L'exclusion mutuelle** : à un instant précis, une ressource est allouée à un seul processus.
– **La détention et l'attente** : les processus qui détiennent des ressources peuvent en demander d'autres.
– **Pas de préemption (*)** : les ressources allouées à un processus sont libérées uniquement par le processus.
– **L'attente circulaire** : il existe une chaîne de deux ou plus processus de telle manière que chaque processus dans la chaîne requiert une ressource allouée au processus suivant dans la chaîne.

Par exemple, dans le problème de circulation précédent le trafic est impossible. On observe que les quatre conditions d'interblocage sont bien remplies :
– **Exclusion mutuelle** : seulement une voiture occupe un endroit particulier de la route à un instant donné.
– **Détention et attente** : aucune voiture ne peut faire marche arrière.
– **Pas de préemption** : on ne permet pas à une voiture de pousser une autre voiture en dehors de la route.
– **Attente circulaire** : chaque coin de la rue contient des voitures dont le mouvement dépend des voitures qui bloquent la prochaine intersection.

(*) Un système **préemptif** est un système qui peut désallouer d'autorité des ressources à un processus même s'il n'est pas dans un état « élu ». Ce processus sera donc stoppé.

Il existe des solutions pour éviter ces interblocages, notamment un mécanisme de « verrou » que l'on verra en activité.

Méthode : repérer la présence d'un interblocage

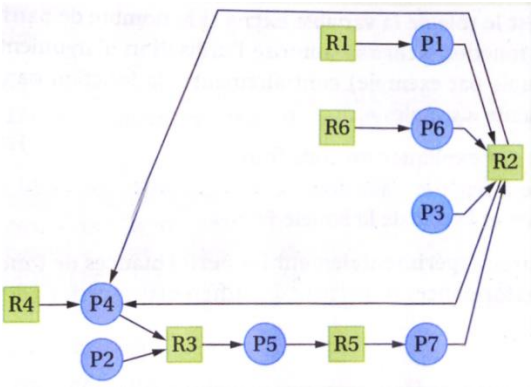
Sept processus Pi sont dans la situation suivante par rapport aux ressources Ri :
- P1 a obtenu R1 et demande R2 ;
- P2 demande R3 et n'a obtenu aucune ressource tout comme P3 qui demande R2 ;
- P4 a obtenu R2 et R4 et demande R3 ;
- P5 a obtenu R3 et demande R5 ;
- P6 a obtenu R6 et demande R2 ;
- P7 a obtenu R5 et demande R2.

On voudrait savoir s'il y a interblocage.

a. 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.

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

Graphe orienté



Il y a présence d'un cycle : R2 – P4 – R3 – P5 – R5 – P7 – R2. Il s'agit donc bien d'une situation de *deadlock* (interblocage).