Gestion des processus

- Qu'est-ce qu'un processus
- Création d'un processus
    - Arborescence de processus
    - PID, PPID
- Gérer les processus sur un système Linux
    - A vous de jouer
    - Inspecter les processus en temps réel
    - Terminer un processus
- Ordonnancement des processus par l'OS
    - A vous de jouer
- Interblocage (ou deadlock)
    - A vous de jouer

Dans les années 1970 les ordinateurs personnels n'étaient pas capables d'exécuter plusieurs tâches à la fois : on lançait un programme et on y restait jusqu'à ce que celui-ci plante ou se termine. Les systèmes d'exploitation récents (Windows, Linux ou osX par exemple) permettent d'exécuter plusieurs tâches simultanément - ou en tous cas, donner l'impression que celles-ci s'exécutent em même temps. A un instant donné, il n'y a donc pas un mais plusieurs programmes qui sont en cours d'exécution sur un ordinateur : on les nomme **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 processeur, et de s'assurer que les processus ne se gênent pas les uns les autres.

Nous avons tous été confrontés à la problématique de la gestion des processus dans un système d'exploitation, en tant qu'utilisateur :

- quand nous cliquons sur l'icône d'un programme, nous provoquons la naissance d'un ou plusieurs processus liés au programme que nous lançons
- quand un programme ne répond plus, il nous arrive de lancer le gestionnaire de taches pour tuer le processus en défaut

Nous allons voir en détails dans cette séquence comment les processus sont gérés dans le système d'exploitation Linux.

Qu'est-ce qu'un processus

Un processus est un programme en cours d'exécution sur un ordinateur. Il est caractérisé par

- un ensemble d'instructions à exécuter - souvent stockées dans un fichier sur lequel on clique pour lancer un programme (par exemple *firefox.exe*)
- un espace mémoire dédié à ce processus pour lui permettre de travailler sur des données qui lui sont propres : si vous lancez deux instances de *firefox*, chacune travaillera indépendament de l'autre avec ses propres données.
- des ressources matérielles : processeur, entrées-sorties (accès à internet en utilisant la connexion Wifi).

Il ne faut donc pas confondre le fichier contenant un programme (portent souvent l'extension *.exe* sous windows) et le ou les processus qu'ils engendrent quand ils sont exécutés : Un programme est juste un fichier contenant une suite d'instructions (*firefox.exe* par exemple) alors que les processus sont des instances de ce programme ainsi que les ressources nécessaires à leur exécution (plusieurs fenêtres de firefox ouvertes en même temps).

Création d'un processus

La création d'un processus peut intervenir

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

Sur Linux, la création d'un processus se fait par clonage d'un autre processus au travers d'un appel systeme : **fork()**.

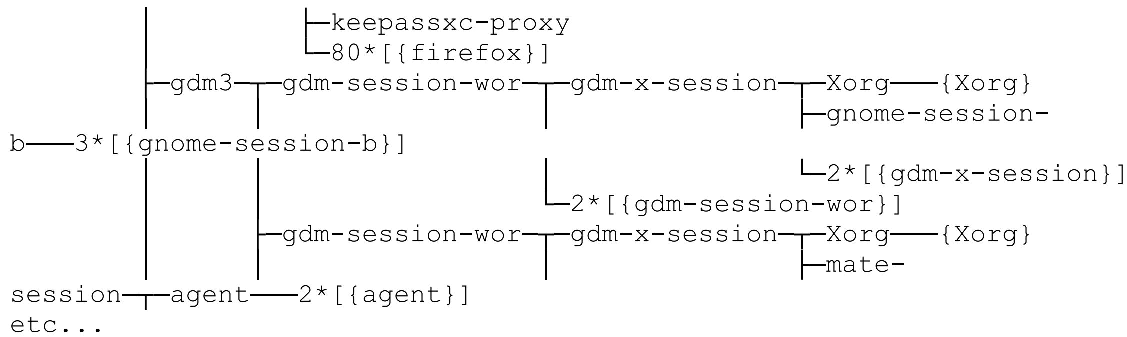
- le processus qui fait appel à **fork()** est appelé **processus père**
- le processus qui est ainsi créé par clonage est le **processus fils**
- après le clonage, un processus peut remplacer son programme par un autre programme grâce à l'appel système **exec()**.

Arborescence de processus

Ce système de création un peu particulier (désigné souvent par *fork/exec*) conduit à l'émergence d'une arborescence de processus : un processus père engendre un ou plusieurs fils qui à leur tour engendrent des fils etc... Sur Linux, le père de tous les processus se nomme **init**, il est créé au démarrage du système.

l'instruction `pstree` permet de visualiser l'arbre de processus sous linux:

```
wawa@wawa-XPS-8300:~/Bureau$ pstree
systemd─┬─NetworkManager───2*[{NetworkManager}]
        ├─PrusaSlicer-2.2───{PrusaSlicer-2.2}
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─apache2───5*[apache2]
        ├─avahi-daemon───avahi-daemon
        ├─blueman-tray───3*[{blueman-tray}]
        ├─colord───2*[{colord}]
        ├─containerd───17*[{containerd}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd───dbus
        ├─dbus-daemon
        ├─dhclient───3*[{dhclient}]
        ├─firefox─┬─RDD Process───3*[{RDD Process}]
        │         ├─Web Content───34*[{Web Content}]
        │         ├─2*[Web Content───38*[{Web Content}]]
        │         ├─2*[Web Content───36*[{Web Content}]]
        │         ├─Web Content───37*[{Web Content}]
        │         ├─Web Content───32*[{Web Content}]
        │         ├─Web Content───33*[{Web Content}]
        │         ├─WebExtensions───33*[{WebExtensions}]
```



```
                    ├─keepassxc-proxy
                    └─80*[{firefox}]
       ├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───{Xorg}
                                                 ├─gnome-session-
b───3*[{gnome-session-b}]
                                                 └─2*[{gdm-x-session}]
                                └─2*[{gdm-session-wor}]
              ├─gdm-session-wor─┬─gdm-x-session─┬─Xorg───{Xorg}
                                                 ├─mate-
session─┬─agent───2*[{agent}]
etc...
```

PID, PPID

Un processus est caractérisé par un identifiant unique : son **PID** (Process Identifier). Lorsqu'un processus engendre un fils, l'OS génère un nouveau numéro de processus pour le fils. Le fils connaît aussi le numéro de son père : le **PPID** (Parent Process Identifier).

Gérer les processus sur un système Linux

Pour illustrer cette partie nous allons utiliser l'émulateur de l'OS linux avec webminal.org/terminal/, il vous suffit de vous connecter et de créer un compte très rapidement avec votre messagerie personnelle, c'est gratuit.

Il est alors possible de visualiser les processus qui sont exécutés sur cette plateforme virtuelle grâce à la commande `ps -eF`. Pour un affichage page par page, utilisez `ps -eF | more`

A vous de jouer

1. Quel est le PID du processus **init** ?
2. Quel est le PPID de **init** ?
3. **init** possède t-il un frêre ?
4. Citer quelques descendants directs de **init**

Inspecter les processus en temps réel

Une commande indispensable à connaître sous Linux pour inspecter les processus est la commande `top`.

Lancez cette commande dans un terminal. Vous devriez avoir quelque chose du genre :

```
top - 15:07:07 up  6:47,  1 user,  load average: 0,08, 0,15, 0,15
Tâches: 355 total,   1 en cours, 354 en veille,   0 arrêté,   0 zombie
%Cpu(s):  0,5 ut,  0,2 sy,  0,0 ni, 99,4 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :   7943,8 total,   1954,2 libr,   3571,0 util,   2418,6 tamp/cache
MiB Éch:   8192,0 total,   8182,2 libr,      9,8 util.   3898,7 dispo Mem

  PID UTIL.     PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TEMPS+ COM.
13266           20   0 2401412 636112  91908 S   0,0   7,8   2:22.27 prusa-slicer
16853           20   0 3976324 521448 242448 S   0,3   6,4  14:34.10 firefox
35838           20   0 2723516 280100 203844 S   0,0   3,4   0:12.40 Web Content
22685           20   0 1003148 261068 103364 S   0,0   3,2   6:49.57 codium
16912           20   0 2805020 238128 142204 S   0,0   2,9   1:52.82 Web Content
17956           20   0 2609260 212680 107464 S   0,0   2,6   1:56.15 Web Content
27297           20   0 2558816 193960 112772 S   0,0   2,4   0:18.59 Web Content
17477           20   0 2695212 193904 117896 S   0,0   2,4   0:33.98 Web Content
23837           20   0 2594596 193492 125000 S   0,0   2,4   0:41.62 Web Content
18025           20   0 2534712 187416 105008 S   0,7   2,3   0:48.02 Web Content
16966           20   0   26,5g 183920 105228 S   0,0   2,3   1:04.27 WebExtension
28860           20   0 2585752 180304 125588 S   0,0   2,2   1:03.22 Web Content
 7832 gdm       20   0 4083444 173200  93284 S   0,0   2,1   0:10.78 gnome-shell
 9731 root      20   0  500852 139752 108040 S   1,0   1,7   6:52.23 Xorg
 9975           20   0 2099160 137236  79972 S   0,0   1,7   0:33.29 caja
22657           20   0  373364 133548  92252 S   0,0   1,6   2:40.30 codium
22626           20   0  669516 130884  92324 S   0,0   1,6   1:17.23 codium
```

L'affichage se rafraîchit en temps réel contrairement à `ps` qui fait un instantané. L'application est plus riche qu'il n'y paraît. Il faut passer un peu de temps à explorer toutes les options. Celles-ci s'activent par des raccourcis clavier. En voici quelques-uns :

- h : affiche l'aide
- M : trie la liste par ordre décroissant d'occupation mémoire. Pratique pour repérer les processus trop gourmands
- P : trie la liste par ordre décroissant d'occupation processeur
- i : filtre les processus inactifs. Cela ne montre que ceux qui travaillent réellement.
- k : permet de tuer un processus - à condition d'en être le propriétaire. Essayez de tuer init ...
- V : permet d'avoir la vue arborescente sur les processus.
- q : permet de quitter `top`

Terminer un processus

Repérez le *PID* de `top` puis tuez-le à l'aide de la commande `k`.

Pour tuer un processus, on lui envoie un *signal* de terminaison. On en utilise principalement 2 :

- SIGTERM (15) : demande la terminaison d'un processus. Cela permet au processus de se terminer proprement en libérant les ressources allouées.
- SIGKILL (9) : demande la terminaison immédiate et inconditionnelle d'un processus. C'est une terminaison violente à n'appliquer que sur les processus récalcitrants qui ne répondent pas au signal SIGTERM.

Pour terminer `top` proprement, vous lui enverrez donc un signal SIGTERM en tapant le numéro 15. Cela est équivalent à la commande shell `kill -15 PID` où *PID* désigne le numéro du processus à quitter proprement.

Si ce dernier est planté et ne réagit pas à ce signal, alors vous pouvez vous en débarrasser en tapant `kill -9 PID`.

Exercice :

1. lancez l'éditeur de textes.
2. repérez son *PID* à l'aide de la commande `ps` ou `top`
3. quittez l'application en utilisant la commande `kill`

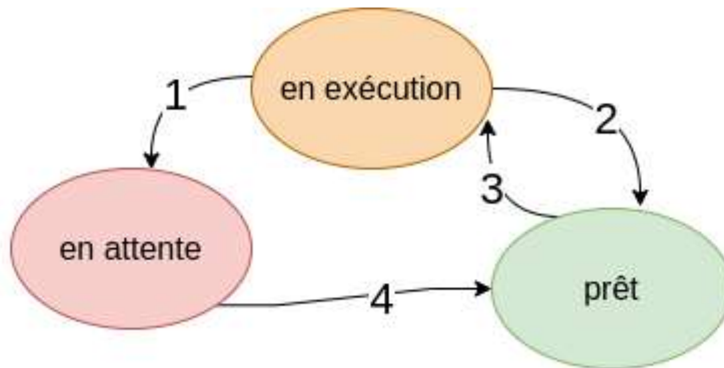
Ordonnancement des processus par l'OS

Dans un système multitâche plusieurs processus sont actifs simultanément, mais un processeur (simple coeur) ne peut exécuter qu'une instruction à la fois. Il va donc falloir partager le temps de processeur disponible entre tous les processus : c'est le travail de l'**ordonnanceur** (ou **scheduler** en anglais). Ce dernier a pour tâche de sélectionner le processus suivant à exécuter parmi ceux qui sont prêts.

Un processus peut donc se trouver dans différents états :

- prêt (*ready*): le processus attend son tour pour prendre la main
- en exécution (*running*): le processus a accès au processeur pour exécuter ses instructions
- en attente (*sleeping*) : le processus attend qu'un événement se produise (saisie clavier, réception d'une donnée par le réseau ou le disque dur ...)
- arrêté (*stopped*) : le processus a fini son travail ou a reçu un signal de terminaison (SIGTERM, SIGKILL, ...). Il libère les ressources qu'il occupe.
- zombie : Une fois arrêté, le processus informe son parent afin que ce dernier l'élimine de la table des processus. Cet état est donc temporaire mais il peut durer si le parent meure avant de pouvoir effectuer cette tâche. Dans ce cas, le processus fils reste à l'état zombie...

Les 3 premiers états sont les plus importants puisqu'ils décrivent le cycle de vie normal d'un processus :



1 : Le processus se met en attente d'un événement
2 : L'ordonnanceur passe la main à un autre processus
3 : L'ordonnanceur choisit ce processus
4 : L'événement attendu se produit

Afin d'élire quel processus va repasser en mode **exécution**, l'ordonnanceur applique un algorithme prédéfini lors de la conception de l'OS. Le choix de cet algorithme va impacter directement la réactivité du système et les usages qui pourront en être fait. C'est un élément critique du système d'exploitation.

Sous Linux, on peut passer des consignes à l'ordonnanceur en fixant des priorités aux processus dont on est propriétaire : Cette priorité est un nombre entre -20 (plus prioritaire) et +20 (moins prioritaire).

⚠ **Attention !** ⚠

les utilisateurs autres que le super-utilisateur root ne peuvent que diminuer la priorité de leurs processus. Et encore, ils sont restreints à la plage de 0 à 19. Seul root peut jouer sur l’intégralité de l’échelle, pour n’importe quel processus actif.

On peut agir à 2 niveaux :

- fixer une priorité à une nouvelle tâche **dès son démarrage** avec la commande `nice`
- modifier la priorité d'un processus **en cours d'exécution** grâce à la commande `renice`

les colonne **PR** et **NI** de la commande `top` montrent le niveau de priorité de chaque processus

PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPS+	COM.
20	0	1027180	259700	104148	S	15,2	3,2	12:25.79	codium
20	0	510080	146152	108104	S	4,3	1,8	10:37.94	Xorg
20	0	378228	131104	90176	S	3,0	1,6	4:45.04	codium
20	0	669772	131708	92376	S	2,3	1,6	2:12.91	codium
20	0	102804	6568	5308	S	1,7	0,1	2:47.83	synergys
20	0	858752	79816	46688	S	1,0	1,0	1:09.15	terminator
20	0	579292	110468	82028	S	1,0	1,4	0:03.57	codium

Le lien entre **PR** et **NI** est simple : **PR = NI + 20** ce qui fait qu'une priorité **PR** de 0 équivaut à un niveau de priorité maximal.

Exemple : Pour baisser la priorité du process `terminator` dont le *PID* est 21523, il suffit de taper

```
renice +10 21523
```

A vous de jouer

Nous allons tester l'efficacité du paramètre *nice* de l'ordonnanceur sur le temps d'exécution d'un programme python. Pour cela, nous allons charger le processeur de la machine à fond et chronométrer le temps d'exécution d'un script python pour plusieurs valeurs du paramètre *nice*.

Pour cet exercice, n'hésitez pas à ouvrir plusieurs fenêtres de terminal côte à côte.

1. en utilisant la commande `cat /proc/cpuinfo` noter le nombre de processeurs disponibles sur votre machine
2. créer un programme python nommé **infini.py** contenant une boucle infinie
3. créer un second programme test contenant

```
4. def bidon():
5.  a = 0
6.  for i in range(100000):
7.      a += a**3
```

8. lancer un interpréteur python3 et noter son numéro de processus
9. dans l'interpréteur python, tapez les commandes

```
10. >>> from timeit import timeit
11. >>> import test
12. >>> print(timeit(bidon, number = 100))
```

cette commande va lancer 100 fois la fonction bidon et renvoyer le temps d'exécution moyen.

13. taper la commande `python3 infini.py &` autant de fois qu'il y a de processeurs sur la machine le symbole `&` indique au shell de lancer le programme en arrière plan. Nous allons donc monopoliser l'ensemble des ressources processeurs de la machine avec des boucles infinies. Le travail de l'ordonnanceur sera donc bien visible car les ressources processeur vont se raréfier.
14. Relancer `timeit(test.bidon, number = 100)` dans le shell python. Vous devriez noter un ralentissement par rapport à la première exécution. En effet, le processeur a moins de temps à consacrer à l'exécution de la fonction `bidon`.
15. Changer la priorité de l'interpréteur python en mettant un *nice* à **+10**.
16. Relancer `timeit(test.bidon, number = 100)` dans le shell python. Que constatez-vous ?

Interblocage (ou deadlock)

Les interblocages sont des situations de la vie quotidienne. Un exemple 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 également, l'interblocage peut se produire lorsque des processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement. Ce scénario catastrophe peut se produire dans un environnement où des ressources sont partagées entre plusieurs processus et l'un d'entre eux détient indéfiniement une ressource nécessaire pour l'autre.

Cette situation d'interblocage a été théorisée par l'informatitien Edward Coffman (1934-) qui a énoncé quatre conditions (appelées conditions de coffman) menant à l'interblocage :

1. **Exclusion mutuelle** : au moins une des ressources du système doit être en accès exclusif.
2. **Rétention des ressources** : un processus détient au moins une ressource et requiert une autre ressource détenue par un autre processus
3. **Non préemption** : Seul le détenteur d'une ressource peut la libérer.
4. **Attente circulaire** : Chaque processus attend une ressource détenue par un autre processus. P_1 attend une ressource détenue par P_2 qui à son tour attend une ressource détenue par P_3 etc... qui attend une ressource détenue par P_1 ce qui clos la boucle.

Il existe heureusement des stratégies pour éviter ces situations. Nous ne rentrerons pas ici dans ces considérations qui dépassent le cadre du programme.

A vous de jouer

1. Identifiez et explicitez sur l'exemple du carrefour à priorité à droite les 4 conditions de *Coffman* menant à l'interblocage.
2. Imaginez des situations de la vie quotidienne - comme l'exemple du carrefour - où un interblocage peut survenir.

Sources :

[1] : Olivier Lecluse, Lycée Salvador Allende, Caen, https://www.lecluse.fr/nsi/NSI_T/archi/process/