GNU/Linux - Processus

- Objet : Les processus sous GNU/Linux
- Niveau requis :
  débutant, avisé
- Commentaires : *Initiation à la gestion des processus.*
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !
- Suivi :
  à-tester
  - Création par smolski le 03/05/2013
  - Testé par .... le ....
- Commentaires sur le forum : C'est ici[1)]

Généralités

Linux sait gérer plusieurs programmes simultanément. C'est ce que l'on nomme le multitâche.
Dans un environnement graphique, ce concept est plus évident de part la présence de plusieurs fenêtres à l'écran.
Toutefois cet article s'interessera plutôt à la gestion du multitâche en ligne de commande.

Ce qui est désigné comme processus est une instance de programme s'exécutant à un instant donné ainsi que son contexte (ou environnement). Ce dernier terme désigne l'ensemble des ressources utilisées par le programme pour pouvoir se dérouler comme par exemple la mémoire ou les fichiers ouverts.

Les processus sont identifiés par un numéro unique dans le système à un moment donné : le *PID*.
C'est à l'aide de ce nombre que l'on peut désigner une instance de programme et interagir avec.
Ils sont également caractérisés par un propriétaire. Il s'agit de l'utilisateur qui a demandé l'exécution.

En général, seul ce propriétaire pourra entreprendre des actions sur le processus.

Ils sont de plus organisés en hiérarchie.
Chaque processus doit être lancé par un autre.

1. Le premier porte le nom de processus **père** ou **parent** et
2. ceux initiés le nom d'enfants ou processus **fils**.

Cette hierarchie a une racine.
Il s'agit le plus souvent sur un système GNU/Linux du programme **init** qui est lancé par le noyau à la fin de son initialisation et a le *PID 1*.
Il va alors entre autres se charger de lancer les différents services du système qui sont appelés démons.

Un démon (francisation du terme anglais daemon) est un processus qui est constamment en activité et fournit des services au système ou à l'extérieur.

Un exemple serait un programme de serveur *web* ou *FTP* ou bien celui gérant la restitution de sons.

init lance aussi un programme permettant à un utilisateur (*ou plusieurs*) de se connecter au système. Ce pourra être une fenêtre en *mode graphique* ou une invite en *mode texte*[2)] qui permettra de saisir le nom d'utilisateur et le mot de passe avant de lancer une interface utilisateur (également graphique ou non).

Etats des processus

Lors de sa vie (entre le moment où il est lancé et celui où il se termine), un processus peut passer par différents états. Les principaux sont les suivants :

```
Actif
Exécutable
Endormi
Zombie
```

Actif

Le premier état correspond au processus qui en train de réaliser des actions à un instant donné. Il possède le processeur et réalise les opérations nécessaires à son déroulement.

Exécutable

Le deuxième état est pour un processus qui pourrait tout à fait être en train de s'exécuter, mais il doit attendre que le processus actif laisse sa place.

En effet, le multitâche n'est en quelque sorte qu'une illusion. Sur une machine possédant un seul processeur, à un instant donné un seul programme peut opérer.

Ces deux premiers états ne sont généralement pas distingués par l'utilisateur du point de vue duquel, ils correspondent, l'un comme l'autre, à un programme en cours d'exécution. Cette nuance concerne davantage le noyau.

Endormi

Un **processus endormi** ne fait rien.

- Il attend une condition pour redevenir exécutable ou actif.
- Il peut se mettre lui-même en sommeil.

Un programme par exemple peut attendre quelques secondes avant de poursuivre son exécution pour laisser le temps à l'utilisateur de lire un message affiché. Mais il peut aussi être mis en sommeil par le noyau en attendant que ce qu'il demande soit disponible.

Pour illustrer ceci, on peut observer un programme de chat (IRC) permettant de dialoguer sur Internet avec d'autres personnes.
Tant que personne n'écrit rien, ce programme n'a rien à faire.

- Il va tout d'abord dire au système qu'il souhaite lire les informations qui arrivent d'Internet.
- Si rien n'est présent, il est mis dans un état endormi.
- Il sera alors réveillé dès que quelqu'un écrit un message afin que le processus puisse le traiter.

Zombie

Le dernier des états évoqués ici est un peu particulier. Il s'agit de l'état zombie.
Un tel processus est en réalité terminé.
Il a fini son exécution et n'a donc plus de raisons d'exister.

Seulement pour diverses raisons possibles, son *père* n'a pas été informé de ceci. Et tout processus doit prendre connaissance de la fin de ceux qu'il a lancés (ses *fils*).
Le système conserve donc les informations correspondant au processus *enfant* afin que son *parent* puisse voir qu'il a fini son exécution.

Lister les processus

Comme indiqué précédemment, un processus peut être identifié avec son PID.

Pour connaître celui-ci, il existe un outil qui s'appelle la commande ps.
Il affiche la liste des processus en cours d'exécution (pas forcément tous selon les options) avec d'autres informations comme la commande utilisée pour le lancer ou son état.

Pour connaître les options disponibles avec ps, le plus simple est de consulter sa page de manuel, d'autant que selon les systèmes ces options peuvent changer.

Une option courante est **-e** qui permet de lister tous les processus du système (et non ceux de l'utilisateur courant uniquement).

Ci-dessous un exemple d'affichage à l'utilisation de ps.
L'option **-o** a aussi été utilisée. Elle permet de spécifier quelles sont les informations devant être affichées.
Sans s'étendre sur les possibilités offertes, il faut juste préciser qu'a été choisi ici d'afficher :

1. le PID,
2. le PPID (le PID du parent),
3. l'utilisateur propriétaire et
4. le programme concerné.

```
ps -e -o"%p  %P  %U  %c"
```

```
 PID   PPID  USER      COMMAND
   1      0  root      init
 506      1  root      syslogd
1411      1  root      httpd
```

```
1456      1  root      login
3713   1456  Tian      bash
7083   3713  Tian      ps
```

On retrouve donc bien ici :

1. le processus *init* qui n'a pas de père (*il n'y a pas de processus ayant 0 comme PID*).
2. Ensuite sont présents *syslogd* (*responsable de l'enregistrement des messages d'information ou d'erreur*),
3. httpd (*le programme de serveur web*) et
4. login (*ce dernier permet à un utilisateur de se connecter sur le système*).

On peut voir ici que

1. **login** a pour fils *bash* (*le PPID de bash est 1456 qui est le PID de init*). Il s'agit du shell qui est lancé pour que l'utilisateur puisse interagir avec le sustème.
2. **bash** a comme processus enfant *ps* qui est créé par la commande tapée pour obtenir ce résultat.

Le programme ps ne permet d'obtenir qu'une vision instantanée des processus courant.
Il existe un exécutable qui affiche ceux-ci à intervalles réguliers. Il s'agit de la commande top qui montre par défaut les processus en ordre décroissant d'utilisation du processeur.

Il indique aussi *l'utilisation mémoire* ce qui fait de top un outil d'administration utile pour connaître les programmes consommateurs de ressources.

Intéragir avec les processus

A un processus donné peut être envoyé un signal.
Ceci peut être vu comme une indication donnée au programme de manière asynchrone. Ce dernier terme signifie que le signal doit être pris en compte immédiatement quelles que soient les opérations en cours de traitement.

La personne ayant conçu le programme peut décider du comportement à adopter à la réception du signal pour remplacer celui par défaut.
La partie du programme responsable de ceci est appelée *gestionnaire de signal* (en anglais *signal handler*).

Un signal sera désigné soit par sa valeur numérique soit par son nom.

> Etant donné que les valeurs numériques peuvent varier d'un système à un autre, il vaut mieux utiliser le nom.

Le tableau suivant liste les signaux les plus utilisés avec leur valeur sur un système Linux.
Principaux signaux

Signal	Valeur numérique	Comportement par défaut	Description
SIGINT	2	Terminer le processus	Il s'agit d'une demande venant du clavier, le plus souvent à l'aide la combinaison de touches Ctrl+C
SIGKILL	9	Terminer le processus	Ce signal permet d'arrêter tout programme car il ne peut être géré différemment que le comportement par défaut. L'arrêt du programme est brutal.
SIGUSR1	10	Terminer le processus	Ce signal n'a pas de signification particulière. Il peut être utilisé de manière différente par chaque programme.
SIGSEGV	11	Terminer le processus	Ce signal est envoyé à un programme lorsque qu'il tente d'accéder à un endroit invalide en mémoire.
SIGUSR2	12	Terminer le processus	Identique à SIGUSR1.
SIGTERM	15	Terminer le processus	Si le programme n'a pas prévu de gérer ce signal, l'arrêt sera aussi brutal que pour SIGKILL. Mais comme le comportement par défaut peut être changé, le programme a la possibilité de réaliser des opérations avant de se terminer.
SIGCHLD	17	Ignorer ce signal	Envoyé à un processus dont un fils est arrêté ou terminé.
SIGCONT	18	Reprendre le processus	Permet de faire se continuer un processus qui avait été arrêté par exemple à l'aide de SIGSTOP (voir ci-après).
SIGSTOP	19	Arrêter le processus	Ce signal demande au processus de suspendre son exécution. Comme SIGKILL, ce signal ne peut être géré différemment.

Ces signaux sont donc envoyés par le sytème à un processus pour le prévenir de certains événements.

Mais un utilisateur peut aussi envoyer des signaux grâce à la commande intégrée à bash, la commande kill.
Elle s'utilise en indiquant le numéro de signal à l'aide de l'option **-s** puis du **PID** du processus.

Par exemple :

```
kill -s SIGSTOP 256
```

```
kill -SIGSTOP 256
```

Ces deux commandes sont identiques.

La deuxième est plus courte à utiliser mais ne fonctionne pas sur tous les systèmes.

L'une comme l'autre enverra le signal **SIGSTOP** au processus de **PID 256**.

Si l'on utilise kill sans spécifier de numéro de signal, c'est un **SIGTERM** qui sera envoyé.

Généralement lorsque l'on souhaite arrêter un programme qui refuse de le faire ou présente un comportement étrange, on commence par essayer de lui envoyer un **SIGTERM** (*avec kill sans option de signal par exemple*).
Cela permet de lui laisser une chance de se quitter proprement.

Si toutefois cela ne fonctionne pas, il faut alors utiliser **SIGKILL** qui va obligatoirement terminer le processus.

Gestion des jobs

Le shell utilisé le plus fréquemment sur les systèmes GNU/Linux, bash, propose un système de contrôle des jobs.

- Un job correspond à une tâche demandée par l'utilisateur.
- Cette tâche peut être composée de plusieurs processus.
- Ce qui les regroupe est un tube.

Un **job** est donc un regroupement de plusieurs commandes devant s'exécuter les unes à la suite des autres en communiquant leurs résultats.

Un exemple tout simple de job pourrait être le suivant :

```
ls | less
```

La commande ls permet de lister les fichiers. Mais quand ils sont trop nombreux, la consultation est dificile.
La commande less permet alors de *paginer* ce qui lui est passé sur son entrée standard et de naviguer facilement à l'aide des touches fléchées du clavier.

Il faut bien voir que lorsque ceci aura été tapé, 2 processus seront créés.

1. Un pour exécuter **ls** et
2. l'autre pour **less**.

Mais *bash* fournit une abstraction permettant de manipuler ceci comme un tout, appelée donc **job**.

> Mais un job pourrait tout aussi bien être constitué d'une seule commande sans utilisation de tube.
> L'utilisation d'uniquement **ls** constituerait aussi le lancement d'un *job*.

La commande intégrée à bash permettant de les lister s'appelle **jobs**.
La lancer dans un shell ne produira aucun résultat normalement.

Il est possible de lancer un job en arrière-plan. Cela permet alors de pouvoir continuer à utiliser le shell pour d'autres tâches pendant qu'il s'exécute.
Emacs est un éditeur de texte très utilisé (notamment pour faire ce site).
Si dans une console au sein d'un environnement graphique on le lance en tapant son nom, une *fenêtre Emacs* va s'ouvrir par défaut.

Mais la console ne sera plus utilisable car le shell attendra la fin de l'exécution de l'éditeur avant de laisser à nouveau la possibilité de lancer des commandes.

Pour pouvoir donc lancer ce logiciel par exemple, il faut rajouter le symbole & à la fin de la définition de la tâche. Pour cet exemple :

```
emacs &
```

```
[1] 6373
```

1. La première valeur est le numéro du job. Celui-ci est unique dans le shell courant. Si plusieurs sont utilisés simultanément (comme par exemple avec plusieurs fenêtres de console), chacun aura ses propres jobs et ne pourra agir sur ceux des autres.
2. La seconde valeur affichée est le PID du dernier processus lancé pour ce job. Il correspondra à la dernière commande dans le tube. Etant donné qu'il n'y en a qu'une ici, ce sera donc le PID pour le processus exécutant Emacs. Avec l'exemple précédent, ce serait celui correspondant à less qui serait affiché.

On peut alors utiliser la commande jobs vue précédemment :

```
jobs
```

```
[1]+  Running       emacs &
```

On retrouve ici le numéro du job et la commande utilisée pour le lancer (dans la dernière colonne).

1. Le texte *Running* signifie qu'il est en train de s'exécuter.
2. Le signe + indique qu'il s'agit du job le plus récemment manipulé (ce qui sera utile pour les commandes qui seront vues ensuite).

Si le programme avait été lancé en oubliant d'ajouter le **&** il est tout de même possible de pouvoir continuer d'utiliser le shell.
Pour cela, il faut utiliser la combinaison de touches Ctrl+Z On aura alors ceci :

```
jobs
```

```
[1]+  Stopped       emacs
```

Contrairement à l'exemple précédent, l'état du job est indiqué comme *Stopped*.

1. Le job est alors en sommeil et ne réalise aucune action.
2. La fenêtre d'Emacs est inactive et on ne peut utiliser ce programme.

Il existe toutefois une commande qui permet de demander à ce qu'un job arrêté reprenne son exécution en arrière-plan.
Il s'agit de **bg** qui prend en paramètre le numéro de job.
Par défaut, bg agit sur le job portant le + dans la liste affichée.
Voici ce que l'on obtient alors :

```
>
```

```
bg
```

```
[1]+ emacs &
> jobs
[1]+  Running       emacs &
```

Tout se passe alors comme si le & avait été indiqué lors du lancement du programme.
On peut remarquer qu'il est d'ailleurs ajouté à l'affichage.

A l'opposé de **bg**, il existe **fg** qui permet de mettre en avant-plan un job.
Celui-ci s'exécutera alors dans le shell empêchant de saisir d'autres commandes.
Le paramètre passé à fg est le même que pour bg (*le numéro de job et par défaut celui avec un* +).
fg peut être utilisé pour un job tournant en arrière-plan ou arrêté.

L'avant-dernier job manipulé est lui aussi indiqué d'une manière particulière. Voici un exemple fictif d'affichage :

```
jobs
```

```
[1]   Running       mozilla &
[2]-  Stopped       su
[3]+  Stopped       emacs -nw &
[4]   Running       dtfile &
```

En tapant seulement **fg** (sans paramètre donc), c'est *Emacs* qui sera mis en avant-plan (*l'option -nw permet de l'utiliser en mode texte directement dans la console*).
Pour rendre le programme de la commande su actif et utilisable, on peut taper indifféremment une des deux commandes suivantes :

```
fg 2
```

```
> fg -
```

L'utilisation du - permet donc d'accéder à l'*avant-dernier job* et d'ainsi facilement passer d'un job à un autre sans connaître leurs numéros.

Etant donné que kill est une commande intégrée à bash, il permet aussi d'intéragir avec les jobs.
Pour cela, il faudra à la place du PID indiquer le numéro du job précédé du caractère %.
Comme par exemple :

```
kill -SIGSTOP %1
```

Qui stoppera le job numéro 3.

En fait l'utilisation de Ctrl+Z vue précédemment est équivalente à l'envoi d'un signal SIGSTOP.

1)
N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !

[2]

terminal ou console