Les processus

1. Définition et environnement

Un **processus** représente à la fois un programme en cours d'exécution et tout son environnement d'exécution (mémoire, état, identification, propriétaire, père...).

Voici une liste des données d'identification d'un processus :

- **Un numéro de processus unique PID** (*Process ID*) : chaque processus Unix est numéroté afin de pouvoir être différencié des autres. Le premier processus lancé par le système est 1 et il s'agit d'un processus appelé généralement init. On utilise le PID quand on travaille avec un processus. Lancer 10 fois le même programme (même nom) produit 10 PID différents.

- **Un numéro de processus parent PPID** (*Parent Process ID*) : chaque processus peut lui-même lancer d'autres processus, des processus enfants (*child process*). Chaque enfant reçoit parmi les informations le PID du processus père qui l'a lancé. Tous les processus ont un PPID sauf le processus 0 qui est un pseudo-processus représentant le démarrage du système (crée le 1 init).

- **Un numéro d'utilisateur et un numéro de groupe** : correspond à l'UID et au GID de l'utilisateur qui a lancé le processus. C'est nécessaire pour que le système sache si le processus a le droit d'accéder à certaines ressources ou non. Les processus enfants héritent de ces informations. Dans certains cas (que nous verrons plus tard) on peut modifier cet état.

- **Durée de traitement et priorité** : la durée de traitement correspond au temps d'exécution écoulé depuis le dernier réveil du processus. Dans un environnement multitâche, le temps d'exécution est partagé entre les divers processus, et tous ne possèdent pas la même priorité. Les processus de plus haute priorité sont traités en premier. Lorsqu'un processus est inactif, sa priorité augmente afin d'avoir une chance d'être exécuté. Lorsqu'il est actif, sa priorité baisse afin de laisser sa place à un autre. C'est l'ordonnanceur de tâches du système qui gère les priorités et les temps d'exécution.

- **Répertoire de travail actif** : à son lancement, le répertoire courant (celui depuis lequel le processus a été lancé) est transmis au processus. C'est ce répertoire qui servira de base pour les chemins relatifs.

- **Fichiers ouverts** : table des descripteurs des fichiers ouverts. Par défaut au début seuls trois sont présents : 0, 1 et 2 (les canaux standard). À chaque ouverture de fichier ou de nouveau canal, la table se remplit. À la fermeture du processus, les descripteurs sont fermés (en principe).

- On trouve d'autres informations comme la taille de la mémoire allouée, la date de lancement du processus, le terminal d'attachement, l'état du processus, les UID effectif et réel ainsi que les GID effectif et réel.

2. États d'un processus

Durant sa vie (temps entre le lancement et la sortie) un processus peut passer par divers états ou **process state** :

- exécution en mode utilisateur (**user mode**)

- exécution en mode noyau (**kernel mode**)

- en attente E/S (**waiting**)

- endormi (**sleeping**)

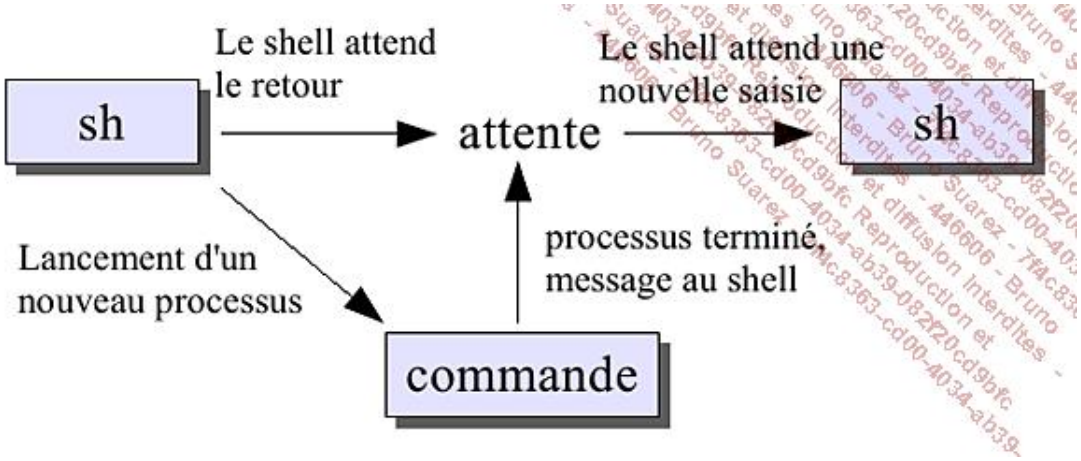
- prêt à l'exécution (**runnable**)

- endormi dans le swap (**mémoire virtuelle**)

- nouveau processus

- fin de processus (**zombie**)

3. Lancement en tâche de fond



En suivant ce que vous avez vu avant, le système étant multitâches un certain nombre de processus tournent déjà sur la machine sans que vous les voyiez. De même le shell que vous utilisez est lui-même un processus. Quand une commande est saisie, le shell crée un nouveau processus pour l'exécuter, ce processus devient un processus enfant du shell. Jusqu'à présent il fallait attendre la fin de l'exécution d'une commande pour en saisir une autre (sauf en utilisant « ; » pour chaîner les commandes).

Rien n'empêche le shell d'attendre le message du processus terminé pour rendre la main : de ce fait la commande une fois lancée, le shell peut autoriser la saisie d'une nouvelle commande sans attendre la fin de l'exécution de la commande précédente. Pour cela il suffit de saisir, après avoir tapé la commande, le **ET Commercial « & »**. Dans ce cas, le shell et la commande lancée fonctionneront en parallèle.

```
$ ls -R / > ls.txt 2/dev/null &
[1] 21976
$
[1]    Done                  ls -l -R / > ls.txt 2/dev/null
$ ls
fic1         fic3         liste        ls.txt       rep1         users
fic2         fic4         liste2       mypass       toto.tar.gz
```

Juste après la saisie un chiffre apparaît, il est à retenir car il s'agit du PID du nouveau processus lancé. Après une autre saisie une ligne Done indique que le traitement est terminé. La valeur [1] est propre à un shell particulier (ksh).

Quelques remarques sur l'utilisation du lancement en tâche de fond :

- Le processus lancé ne devrait pas attendre de saisie au risque de confusion entre cette commande et le shell lui-même.
- Le processus lancé ne devrait pas afficher de résultats sur l'écran au risque d'avoir des affichages en conflit avec celui du shell (par exemple apparition d'une ligne en milieu de saisie).
- Enfin, quand on quitte le shell, on quitte aussi tous ses fils : dans ce cas ne pas quitter le shell pendant un traitement important.

4. Background, foreground, jobs

Vous pouvez récupérer la main sous le shell si vous avez lancé un processus au premier plan. Vous pouvez le stopper temporairement en tapant [Ctrl] **Z** :

```
$ sleep 100
<CTRL+Z>
 [1]+  Stopped                 sleep 100
```

Le processus est stoppé : son exécution est suspendue jusqu'à ce que vous le replaciez au premier plan avec la commande **fg** (*foreground*) :

```
$ fg
Sleep 100
```

Quand vous lancez une commande, vous avez remarqué le premier nombre entre crochets, c'est le numéro de job. Vous pouvez en obtenir la liste avec la commande **jobs**.

```
$ jobs
[1]-  Stopped                 sleep 100
[2]+  Stopped                 sleep 100
```

Les commandes **bg** et **fg** permettent d'agir sur ces jobs en prenant comme paramètre leur numéro. La commande **bg** est exécutée sur un job stoppé pour le relancer en arrière-plan. Le job 2 est relancé en arrière-plan :

```
$ bg 2
[2]+ sleep 100 &
$
[2]+  Done                    sleep 100
```

5. Liste des processus

La commande **ps** (*process status*) permet d'avoir des informations sur les processus en cours. Lancée seule, elle n'affiche que les processus en cours lancés par l'utilisateur et depuis la console actuelle.

```
$ ps
  PID TTY          TIME CMD
 4334 pts/1    00:00:00 bash
 5017 pts/1    00:00:00 ps
```

Pour avoir plus d'informations, utilisez le paramètre `-f`.

```
ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
seb       4334 24449  0 09:46 pts/1    00:00:00 /bin/bash
seb       5024  4334  0 10:51 pts/1    00:00:00 ps -f
```

Le paramètre `-e` donne des informations sur tous les processus en cours.

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
...
seb      26431     1  0 Mar04 ?        00:00:30 kded [kdeinit]
seb      26436 26322  0 Mar04 ?        00:00:03 kwrapper ksmserver
seb      26438     1  0 Mar04 ?        00:00:00 ksmserver [kdeinit]
seb      26439 26424  0 Mar04 ?        00:00:50 kwin [kdeinit
seb      26441     1  0 Mar04 ?        00:00:28 kdesktop [kdeinit]
seb      26443     1  0 Mar04 ?        00:03:40 kicker [kdeinit]
seb      26453     1  0 Mar04 ?        00:00:00 kerry [kdeinit
seb      26454 26424  0 Mar04 ?        00:00:01 evolution
seb      26465 26424  0 Mar04 ?        00:00:11 kde-window-decorator
seb      26467     1  0 Mar04 ?        00:00:02 gconfd-2 12
seb      26474     1  0 Mar04 ?        00:00:01 knotify [kdeinit]
seb      26485     1  0 Mar04 ?        00:03:06 beagled
...
```

Le paramètre `-u` permet de préciser une liste d'un ou plusieurs utilisateurs séparés par une virgule. Le paramètre `-g` effectue la même chose mais pour les groupes, `-t` pour les terminaux et `-p` pour des PID précis.

```
$ ps -u root
  PID TTY          TIME CMD
    1 ?        00:00:05 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:09 ksoftirqd/0
```

```
    5 ?        00:00:23 events/0
    6 ?        00:00:00 khelper
   25 ?        00:00:00 kblockd/0
   26 ?        00:00:00 kacpid
   27 ?        00:00:00 kacpi_notify
  130 ?        00:00:00 cqueue/0
  131 ?        00:00:00 kseriod
  156 ?        00:00:22 kswapd0
  157 ?        00:00:00 aio/0...
```

Enfin le paramètre `-l` propose plus d'informations techniques.

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4704 24449  0  75   0 -  1213 wait   pts/3    00:00:00 bash
```

Voici le détail de quelques colonnes.

Colonne	Définition
UID	User ID, nom de l'utilisateur.
PID	Process ID, numéro du processus.
PPID	Parent Process ID, numéro du processus père.
C	Facteur de priorité, plus la valeur est grande plus la priorité est élevée.
STIME	Heure de lancement du processus.
TTY	Nom du terminal depuis lequel le processus a été lancé.
TIME	Durée de traitement du processus.
CMD	Commande exécutée.
F	Drapeaux du processus (sort du cadre de l'ouvrage).
S	État du processus S (sleeping) R (running) Z (zombie).
PRI	Priorité du processus.
NI	Nice, incrément pour le scheduler.

6. Arrêt d'un processus / signaux

Lorsqu'un processus tourne en tâche de fond, il ne peux pas être arrêté par une quelconque combinaison de touches, sauf en utilisant le gestionnaire de jobs avec fg et bg. Il peut être nécessaire de lui envoyer des signaux auquel il pourra éventuellement réagir. Pour cela il faut employer la commande **kill**. Contrairement à ce que son nom semble indiquer, le rôle de cette commande n'est pas forcément de détruire ou de terminer un processus (récalcitrant ou non), mais d'envoyer des signaux aux processus.

```
kill [-l] -Num_signal PID [PID2...]
```

Le **signal** est l'un des moyens de communication entre les processus. Lorsqu'on envoie un signal à un processus, celui-doit doit l'intercepter et réagir en fonction de celui-ci. Certains signaux peuvent être ignorés, d'autres non. Suivant les Unix on dispose d'un nombre plus ou moins important de signaux. Les signaux sont numérotés et nommés, mais attention, si les noms sont généralement communs d'un Unix à l'autre, les numéros ne le sont pas forcément. L'option **`-l`** permet d'obtenir la liste des signaux.

Signal	Rôle

1 (SIGHUP)	Hang Up, est envoyé par le père à tous ses enfants lorsqu'il se termine.
2 (SIGINT)	Interruption du processus demandé (touche [Suppr], [Ctrl] **C**).
3 (SIGQUIT)	Idem SIGINT mais génération d'un Core Dump (fichier de débuggage).
9 (SIGKILL)	Signal ne pouvant être ignoré, force le processus à finir 'brutalement'.
15 (SIGTERM)	Signal envoyé par défaut par la commande **kill**. Demande au processus de se terminer normalement.

```
$ sleep 100&
[1] 5187
$ kill 5187
$
[1]+  Complété              sleep 100
$ sleep 100&
[1] 5194
$ kill -9 5194
[1]+  Processus arrêté      sleep 100
```

7. nohup

Quand le shell est quitté (exit, [Ctrl] D...) le signal 1 SIGHUP est envoyé aux enfants pour qu'ils se terminent aussi. Lorsqu'un traitement long est lancé en tâche de fond et que l'utilisateur veut quitter le shell, ce traitement sera alors arrêté et il faudra tout recommencer. Le moyen d'éviter cela est de lancer le traitement (processus) avec la commande **nohup**. Dans ce cas le processus lancé ne réagira plus au signal SIGHUP, et donc le shell pourra être quitté, la commande continuera son exécution.

Par défaut les canaux de sortie et d'erreur standard sont redirigés vers un fichier **nohup.out**, sauf si la redirection est explicitement précisée.

```
$ nohup ls -lR / &
10206
$ Sending output to nohup.out
```

Quand un fils se termine, il envoie le signal SIGCHLD à son père. Sauf cas prévu (le père se détache du fils) le père doit obtenir autant de SIGCHLD qu'il a eu de fils ou émis de SIGHUP. Si le père se termine avant que les fils se terminent ceux-ci deviennent des zombis : le signal SIGCHLD n'est pas reçu... Le processus fils est bien terminé, il est mort, il ne consomme aucune ressource. Il ne peut donc être tué (puisqu'il est mort) mais continue à occuper une entrée dans la table des processus. C'est init qui le récupère, et init étant toujours en attente, le zombie peut finir par disparaître.

8. nice et renice

La commande **nice** permet de lancer une commande avec une priorité plus faible, afin de permettre éventuellement à d'autres processus de tourner plus rapidement.

```
nice [-valeur] commande [arguments]
```

Une valeur positive causera une baisse de priorité, une négative l'augmentation de la priorité (si autorisé). La valeur doit être comprise entre -20 et 20. Plus la valeur est élevée et plus le traitement est ralenti.

```
$ nice -10 ls -lR / >liste 2>/dev/null&
10884
$ ps -l
       F S     UID    PID   PPID    C PRI  NI  ADDR   SZ WCHAN    TTY
           TIME CMD
80808001 U N+   75  10884  10822 28.5  54  10     0 848K aa3b3a9c ttyp4
       0:03.32 ls
```

La commande **renice** fonctionne un peu comme nice mais elle permet de modifier la priorité en fonction d'un utilisateur, d'un groupe ou d'un PID. La commande visée doit donc déjà tourner.

```
renice [-n prio] [-p] [-g] [-u] ID
```

La priorité doit être comprise entre -20 et 20. L'utilisateur standard ne peut utiliser que les valeurs entre 0 et 20 permettant de baisser la priorité. L'option **-p** précise un PID, **-g** un GID et **-u** un UID.

9. time

La commande **time** mesure les durées d'exécution d'une commande, idéal pour connaître les temps de traitement, et retourne trois valeurs :

- **real** : durée totale d'exécution de la commande.
- **user** : durée du temps CPU nécessaire pour exécuter le programme.
- **system** : durée du temps CPU nécessaire pour exécuter les commandes liées à l'OS (appels système au sein d'un programme).

Le résultat est sorti par le canal d'erreur standard 2. On peut avoir une indication de la charge de la machine par le calcul real / (user+system). Si le résultat est inférieur à 10, la machine dispose de bonnes performances, au-delà de 20 la charge de la machine est trop lourde (performances basses).

```
$ time ls -lR /home
...
real   4.8
user   0.2
sys    0.5
```