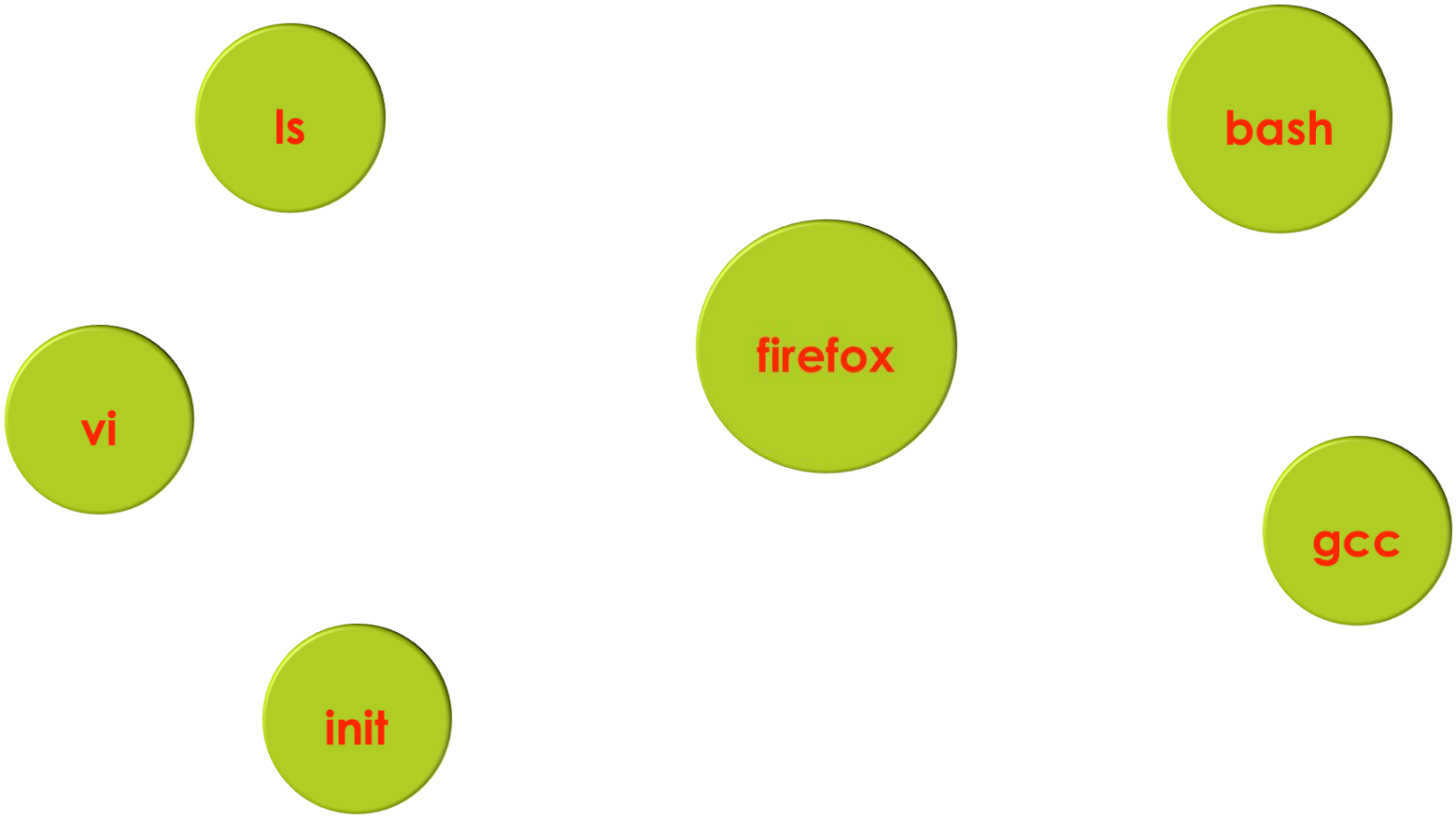
MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD

FSSM
كلية العلوم السملالية

CHAPITRE 7
Unix/Linux : Processus

Définition

- Un processus est un programme en cours d'exécution.
- Chaque programme est exécuté comme un processus



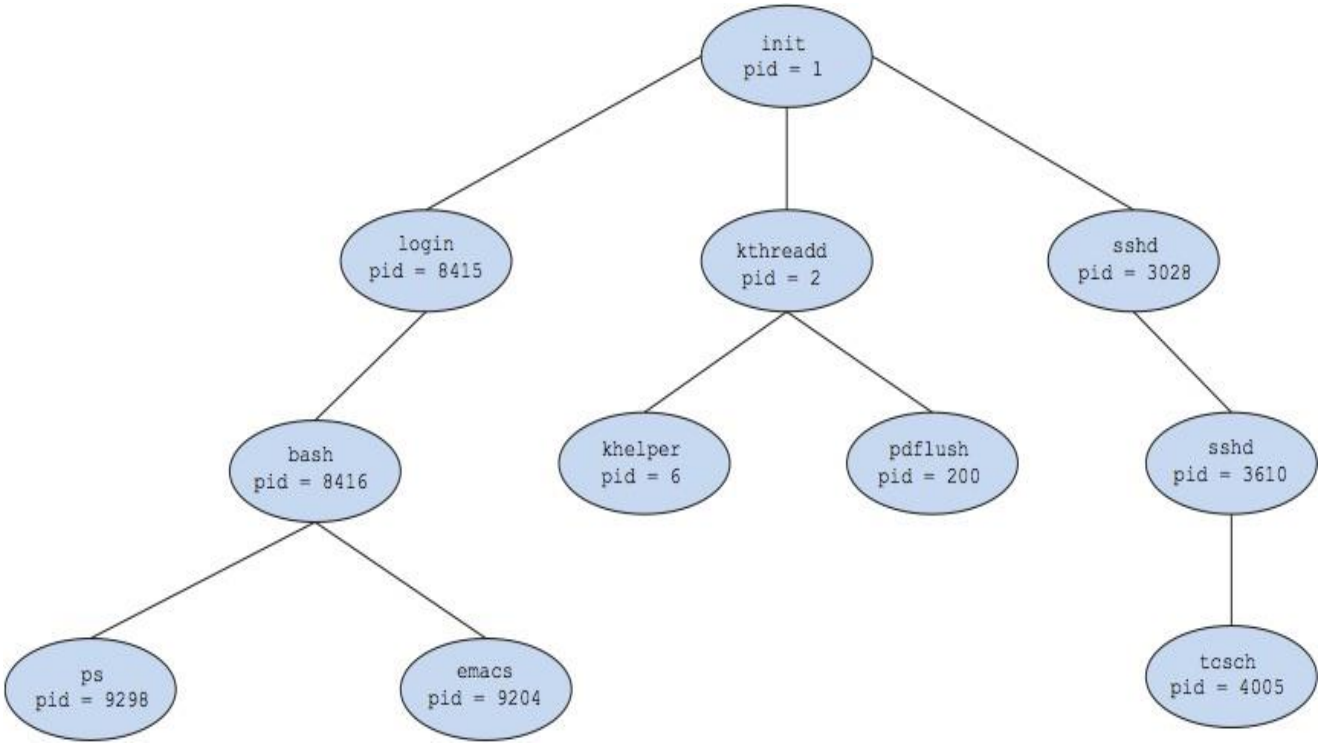
Information sur un processus

Un processus maintient d'importantes informations sur lui même, incluant :

- PID – Process Identification Number, son unique identifiant.
- UID – L'identité de l'utilisateur exécutant le programme.
- Status – Le status courant du processus, par exemple : running, waiting, stopped, etc.
- Priorité – Un facteur qui oriente l'allocation du temps CPU.
- Ressources – Mémoire, fichiers, périphériques …

Organisation des processus

- Chaque processus est initialisé par un autre processus (avec une seule exception?) ✎ tous les processus sont organisés dans un arbre



- Les processus maintiennent une relation parent-fils

Examination des processus

- Il existe essentiellement trois commandes pour visualisez les informations sur un processus :

  - ps – 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.

  - pstree – Affichage de la hiérarchie des processus

  - top – Un affichage en temps réel des processus, et en premier lieu ceux qui utilisent beaucoup de CPU.

Utilisation de ps

- Afficher les informations de l'utilisateur et tous les processus de l'utilisateur courant, y compris ceux sans un terminal associé (TTY)

```
mairaj@ubuntu:~$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
mairaj    1109  0.0  1.0  22456  5264 tty1     S    08:22   0:00 -bash
mairaj    1223  0.0  0.5  18452  2612 tty1     R+   08:40   0:00 ps ux
```

Utilisation de ps : paramètres

- Pour avoir plus d'informations, utilisez le paramètre -f.
- Le paramètre -e donne des informations sur tous les processus en cours.
- 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.
- Enfin le paramètre -l propose plus d'informations techniques.

Utilisation de pstree

```
mairaj@ubuntu:~$ pstree
init─┬─acpid
     ├─atd
     ├─cron
     ├─dbus-daemon
     ├─dhclient
     ├─5*[getty]
     ├─login───bash───pstree
     ├─rsyslogd───3*[{rsyslogd}]
     ├─systemd-logind
     ├─systemd-udevd
     ├─upstart-file-br
     ├─upstart-socket-
     └─upstart-udev-br
```

Utilisation de top

Si vous lancez *top* en ligne de commande :

```
top - 08:43:19 up 20 min,  1 user,  load average: 0.00, 0.01, 0.05
Tasks:  72 total,   2 running,  70 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem:    501496 total,   164208 used,   337288 free,    26816 buffers
KiB Swap:   520188 total,        0 used,   520188 free.    98032 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1148 root      20   0       0      0      0 S   0.3  0.0   0:01.99 kworker/0:2
 1225 root      20   0       0      0      0 S   0.3  0.0   0:00.07 kworker/u2:2
 1226 mairaj    20   0   24824   2964   2556 R   0.3  0.6   0:00.10 top
    1 root      20   0   33388   3996   2752 S   0.0  0.8   0:02.83 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.31 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.02 rcu_sched
```

Intéragir avec les processus

- Etant donné que le shell utilise une interface texte, il est commode d'interagir avec un seul processus à la fois.

- Le processus interactif est le processus foreground (premier plan).

- Les autres processus en exécution, sont des processus en background (arrière-plan).

Gestion des processus

Commande	Signification
cmd1 **;** cmd2	Exécute les commandes séquentiellement
cmd **&**	Exécute la commande en background
<*Ctrl-z*>	Arrête/Suspend le processus en foreground
<*Ctrl-c*>	Kill du processus en foreground
jobs	Liste les processus en backgroud et les processus suspendus
bg	Reprend l'exécution d'un processus arrêté en background (syntaxe : bg *job_ID*)
fg	Switch d'un processus background en foreground (syntaxe : fg *job_ID*)
kill	Envoie un signal (Termine un processus, option -9)
nice	Démarre un processus avec une priorité ajustée
renice	Change la priorité d'un processus

Envoyer un signal à un processus

- Il est possible d'envoyer des signaux à un processus auquel il pourra éventuellement réagir. Pour cela il faut employer la commande **kill**. Cette commande permet d'envoyer des signaux aux processus.
- Syntaxe : kill [-l] -Num_signal PID [PID2...]
- Le **signal** est l'un des moyens de communication entre les processus.
- 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.

Arrêt d’un processus (kill -9 PID)

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.

Communication avec les processus

- Les programmes utilisent les données. Les données peuvent venir de différentes sources :
  - Humains
  - Ligne de Commande
  - Réseau
  - Fichiers
  - Autres programmes

- Une des caractéristiques les plus utiles de Linux est son flux de données flexible.

- Les entrées et les sorties (I/O) d'un programme peuvent être sélectionnés lorsque le programme est lancé.

Communication avec les processus

- Dans Unix (et Linux généralement) tout est un flux d'octets. Les flux sont accessibles sous forme de fichiers, mais il y a trois flux qui sont rarement accessibles par un nom de fichier.

- Ce sont les flux d'entrées/sorties (I/O) attachés à chaque commande :

  - Entrée standard (Standard input, dîte *stdin*)
  - Sortie standard (Standard output, dîte *stdout*)
  - Sortie des erreurs (Standard error, dîte *stderr*)

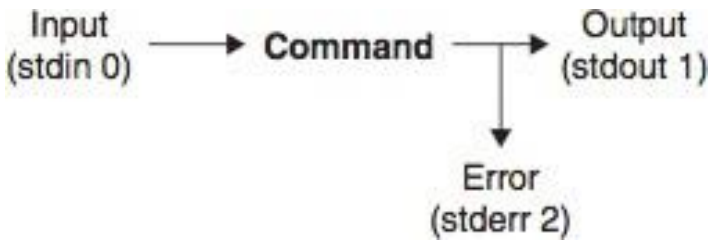
Communication avec les processus

- Par défaut, ces flux sont connectés à votre terminal.
  - Lorsqu'une commande lit un caractère ou une ligne, il lit dans le flux d'entrée standard, qui est le clavier.
  - Quand il affiche des informations, ils sont envoyées à la sortie standard, votre écran.
  - Le troisième flux, celui des erreurs, est également connecté à l'écran; comme son nom l'indique, il est utilisé pour des messages d'erreur.

- Ces flux sont désignés par des numéros, appelés descripteurs de fichiers. Ce sont 0, 1 et 2 respectivement. Les flux d'I/O peuvent être redirigés vers (ou depuis) un fichier ou redirigés par exemple dans la stdin d'une autre commande(Utiliation d'une pipeline |).

Redirection des I/O

stdin, stdout et stderr ont les descripteurs fichiers énumérés 0, 1 et 2 respectivement.



Le shell permet à l'utilisateur de sélectionner une source (ou destination) des données pour chacun de ces fichiers spéciaux. Si aucune sélection n'est faite, l'input et output par défaut sont utilisés.

Syntaxe d’une redirection basique

<	Redirige stdin à un fichier
<<	Accepte le texte sur les lignes suivantes comme standard input.
>	Redirige stdout à un fichier
>>	Redirige stdout à un fichier en concaténation (append)
|	Redirige stdout d’un programme à stdin d’un autre

Syntaxe d’une redirection basique

- Redirection de l’output d’un programme dans un fichier

```
bash-3.2$ echo "test de redirection de stdout"
test de redirection de stdout
bash-3.2$ echo "test de redirection de stdout" > fichier.txt
bash-3.2$ cat fichier.txt
test de redirection de stdout
```

- Redirection de l’input d’un programme depuis un fichier

```
bash-3.2$ cat < fichier.txt
test de redirection de stdout
```

Syntaxe d’une redirection basique

- Redirection de l’output d’un programme vers l’input d’un autre (|)

```
bash-3.2$ ls
etest.txt   fichier.txt gtest.txt
bash-3.2$ ls | sort
etest.txt
fichier.txt
gtest.txt
```

Syntaxe d’une redirection de StdErr

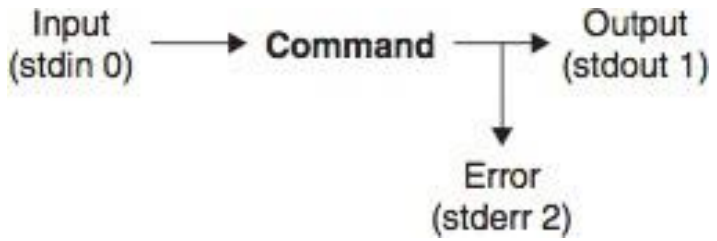
>&	Redirection de *stdout et stderr dans un fichier*

```
bash-3.2$ rm fichier.t
rm: fichier.t: No such file or directory
bash-3.2$ rm fichier.t >& output.txt
bash-3.2$ cat output.txt
rm: fichier.t: No such file or directory
```

Redirection avec les descripteurs de fichiers

n>my*file*	Redirige le descripteur fichier n à myfile
n>&*m*	Redirige le descripteur fichier n au descripteur de fichier m
n>>my*file*	Redirige le descripteur fichier n à myfile en concaténation (append)

Rappel : stdin, stdout et stderr ont les descripteurs fichiers énumérés 0, 1 et 2 respectivement.



Exemples : fichiers descripteurs

- Rediriger stderr à un fichier

```
bash-3.2$ rmdir directory
rmdir: directory: No such file or directory
bash-3.2$ rmdir directory 2> error.log
bash-3.2$ cat error.log
rmdir: directory: No such file or directory
```

- Rediriger stderr à stdout et stdout vers le fichier output.txt

```
bash-3.2$ ls -l > output.txt 2>&1
```

La sortie 2 est redirigée vers la sortie 1, donc les messages d'erreurs passeront par la sortie standard. Puis le résultat de la sortie standard de la commande **ls** est redirigé vers le fichier output.txt.
Ce fichier contiendra donc à la fois la sortie standard et la sortie d'erreur.

Exemples : fichiers descripteurs

- Rediriger stdout à la corbeille et stderr à stdout

```
bash-3.2$ ls -l > /dev/null 2>&1
```

/dev/null ressemble à un trou noir, une corbeille qui absorbe tous ce que vous lui envoyer sans laisser de trace.