# Cours R202 — Administration système et fondamentaux de la virtualisation

**IUT de Villetaneuse — R&T 1ère année**

Laure Petrucci

21 janvier 2024

# Table des matières

# Liste des tableaux

# Chapitre 1

# Environnement Unix

Sous Unix, l'utilisateur travaille avec un *interpréteur de commandes* appelé *shell*. Cet interpréteur permet à l'utilisateur de travailler dans un environnement soit standard (proposé par défaut), soit personnalisé par l'utilisateur lui-même.

Il existe divers types d'interpréteurs de commandes, dont les principaux sont `sh`, `bash`, `ksh`, `csh` et `tsch`. Ceux-ci comportent plus ou moins de différences. Dans le cadre de ce cours, nous utiliserons `bash`.

## 1.1 Commandes

Les interpréteurs utilisent deux types de commandes, dont le comportement et l'exécution diffèrent : les commandes « externes » et les commandes « internes ».

### 1.1.1 Commandes externes

Une *commande externe* est stockée dans un fichier exécutable. Pour pouvoir être exécutée, le fichier correspondant doit être accessible (voir la description de la variable d'environnement `PATH` dans la section 1.3.1). Puisqu'une telle commande correspond à un fichier, celui-ci peut être détruit et la commande n'est alors plus utilisable.

**Exemple 1** *La commande `ls` est une commande externe qui se trouve dans le fichier `/bin/ls`.*

### 1.1.2 Commandes internes

Les *commandes internes* (encore appelées « *builtins* »), sont comprises dans l'interpréteur de commandes. Elles diffèrent donc selon le *shell* utilisé. De plus, comme elles ne sont pas, contrairement aux commandes internes, contenues dans un fichier, on ne peut pas les supprimer et leur utilisation ne dépend pas de la valeur de la variable d'environnement `PATH`.

Les principales commandes internes utilisées par le *shell* `bash` sont récapitulées dans le tableau 1.1.

### 1.1.3 Différencier les types de commandes

Plusieurs manières permettent de différencier les types de commandes. On peut bien sûr « vider » la variable `PATH` pour éviter de chercher les fichiers exécutables, puis lancer la commande. Si celle-ci est trouvée, c'est une commande interne, sinon c'est une commande externe.

TABLE 1.1 – Principales commandes internes de `bash`

| Commande | Description |
|---|---|
| `alias` | crée un *alias*, c'est -à-dire un raccourci pour une commande complexe. |
| `bg` | met un processus en arrière-plan (*background*). |
| `cd` | change de répertoire de travail (*change directory*). |
| `dirs` | affiche une pile de répertoires mémorisés (*directories*). |
| `echo` | affiche une chaîne de caractères sur la sortie standard. |
| `enable` | active/désactive une commande interne ou donne la liste des commandes internes activées/désactivées. |
| `exec` | remplace le processus courant par un nouvel exécutable (*execute*). |
| `exit` | termine un processus *shell*. |
| `export` | exporte des variables pour les transmettre aux processus fils. |
| `fg` | met un processus en avant-plan (*foreground*). |
| `help` | fournit de l'aide sur les commandes internes du *shell*. Celles-ci ne sont pas explicitées dans le manuel en ligne `man`. |
| `history` | affiche l'historique des commandes utilisées. |
| `jobs` | affiche la liste des travaux en cours. |
| `kill` | envoie un signal à des processus. |
| `popd` | change de répertoire pour atteindre celui que l'on supprime de la pile des répertoires mémorisés (*pop directory*). |
| `pushd` | change de répertoire et l'ajoute à la pile des répertoires mémorisés (*push directory*). |
| `pwd` | affiche la référence absolue du répertoire de travail (*print working directory*). |
| `read` | lit des éléments sur l'entrée standard. |
| `set` | change des paramètres de l'environnement. |
| `shift` | décale un argument dans une liste. |
| `source` | exécute un script *shell*. |
| `times` | affiche des temps d'exécution. |
| `type` | affiche le type de la commande. |
| `umask` | définit le masque des droits d'exécution par défaut. |
| `unalias` | supprime un alias défini précédemment. |
| `wait` | attend la fin d'un processus. |

**Exemple 2**

```
$ export PATH=
$ ls
bash: ls: No such file or directory
$ alias bonjour="echo bonjour"
$
```

*Cet exemple montre que* `ls` *est une commande externe alors que* `alias` *est une commande interne.*

Une autre manière de procéder consiste à utiliser la commande interne `type` qui indique le type de la commande passée en paramètre.

**Exemple 3**

```
$ type ls
ls is /bin/ls
$ type alias
alias is a shell builtin
$
```

*Les résultats sont identiques à ceux obtenus dans l'exemple 2.*

L'utilisation de cette approche présente l'avantage de ne pas modifier l'environnement utilisateur (en l'occurrence la variable `PATH`).

La commande `type` permet également de distinguer d'autres sortes de commandes, et donc d'affiner les résultats.

**Exemple 4**

```
$ alias bonjour="echo bonjour"
$ bonjour
bonjour
$ type bonjour
bonjour is aliased to `echo bonjour'
$
```

*Un alias de nom `bonjour` est d'abord défini. Il doit afficher la chaîne de caractères `"bonjour"`. Ensuite, il est exécuté. Tout se passe comme si `bonjour` était une commande interne. Toutefois, `type` apporte plus de précisions en indiquant que c'est un alias, et à quoi il correspond.*

### 1.1.4 Utilisation de l'historique des commandes

Les commandes utilisées sont enregistrées dans un fichier d'historique, contenant les `HISTSIZE` (*history size*) dernières commandes tapées. Cet historique est utilisé par les flèches pour rappeler des commandes. De plus, il est affiché par `history` avec des numéros de commandes correspondants.

L'historique peut être utilisé de façon plus évoluée :

— `!!` affiche et exécute à nouveau la dernière commande ;
— `!`$n$ affiche et exécute la commande numéro $n$ de l'historique ;
— `!-`$n$ affiche et exécute la $n$-ième dernière commande ;
— `!`*chaîne* affiche et exécute la dernière commande commençant par *chaîne*.

Ces commandes peuvent être suivies du séparateur `:` et d'une commande de substitution de l'éditeur `ed`. Par exemple si la commande 10 est `cd /usr/bin`, alors la commande `!10:s/bin/include` affiche et exécute `cd /usr/include`.

## 1.2 Arborescence de fichiers

L'arborescence de fichiers d'Unix comprend des répertoires contenant des fichiers destinés à assurer des fonctions spécifiques. On dispose ainsi d'une structuration propre de l'ensemble des fichiers. Le tableau 1.2 récapitule les utilisations des principaux répertoires de l'arborescence [1].

## 1.3 Environnement utilisateur

Le comportement de l'interpréteur de commandes dépend de la valeur des *variables d'environnement*. Ces valeurs peuvent être soit des valeurs par défaut définies par l'administrateur, soit des valeurs personnalisées par l'utilisateur lui-même.

1. Les noms des répertoires sont généralement ceux indiqués dans le tableau, mais peuvent varier suivant la version du système ou l'installation.

TABLE 1.2 – Principaux répertoires d'UNIX

| Répertoire | Description du contenu |
|---|---|
| `/bin`<br>`/usr/bin` | commandes de base |
| `/usr/local`<br>`/usr/local/bin`<br>`/opt` | commandes et logiciels ajoutés par l'administrateur |
| `/sbin`<br>`/usr/sbin` | commandes d'administration système |
| `/boot` | fichiers du noyau UNIX |
| `/etc`<br>`/etc/init.d` | fichiers et répertoires de configuration du système<br>fichiers de démarrage des services |
| `/usr/include` | fichiers d'en-tête à inclure lors du développement de programmes |
| `/lib`<br>`/usr/lib` | bibliothèques utilisées pour le développement de programmes |
| `/var/log` | traces d'exécution de processus |
| `/var/mail` | contient les boîtes aux lettres des utilisateurs |
| `/var/spool`<br>`/var/spool/mail`<br>`/var/spool/lpd` | contient des files d'attente<br>files de courrier électronique<br>file d'impression |
| `/home`<br>`/users`<br>`/root` | répertoires privés des utilisateurs<br><br>répertoire privé de l'administrateur |
| `/tmp` | fichiers temporaires |
| `/dev` | fichiers correspondant aux périphériques |
| `/mnt`<br>`/mnt/cdrom`<br>`/mnt/...`<br>`/media`<br>`/media/usb`<br>`/media/...` | points de montage des périphériques amovibles |

### 1.3.1 Variables d'environnement

Les *variables d'environnement* sont des variables du *shell*. Elles sont définies lorsque l'utilisateur se connecte, lorsqu'il crée un nouveau terminal, ou dès lors qu'il y a exécution d'un processus *shell* (pour plus de détails sur les processus, voir le chapitre 4). Les variables peuvent être modifiées, et éventuellement transmises aux *processus fils*. Lorsque le processus ayant créé les variables termine, celles-ci sont perdues.

Les *noms des variables* d'environnement sont des chaînes de caractères pouvant contenir des lettres, des chiffres, le caractère `_`, et commençant toujours par une lettre. L'interpréteur de commandes est sensible à la casse, c'est-à-dire qu'il différencie les lettres majuscules des minuscules.

Certaines variables ont un rôle spécifique et ont souvent une valeur définie par défaut. Le nom de ces variables est généralement en majuscules. Le tableau 1.3 présente les principales variables d'environnement. La liste des variables d'environnement définies, avec leur nom et valeur, est obtenue par la commande `printenv`.

TABLE 1.3 – Principales variables d'environnement

| Variable | Description |
|---|---|
| DISPLAY | adresse du terminal sur lequel se fait l'affichage (utilisé lors des affichages à distance) |
| HISTSIZE | taille de l'historique (nombre de commandes stockées) |
| HOSTNAME | nom de la machine |
| HOME | référence absolue du répertoire privé de l'utilisateur |
| LANG | langue d'affichage des messages |
| LOGNAME | identité de l'utilisateur (*login name*) |
| MANPATH | liste de références des répertoires contenant les pages du manuel en ligne `man` |
| PATH | liste de références des répertoires dans lesquels rechercher les fichiers exécutables |
| PS1 | première invite de commande (*prompt*) |
| PS2 | seconde invite de commande |
| PWD | répertoire de travail |
| SHELL | référence absolue du *shell* |
| TERM | type de terminal utilisé pour l'affichage |
| UID | numéro d'identificateur de l'utilisateur (*user identifier*) |
| USER | identité de l'utilisateur (identique à LOGNAME) |

**Exemple 5**

```
$printenv
MANPATH=/usr/share/man:/usr/local/share/man:/usr/local/man:/usr/X11/man
TERM=xterm-color
SHELL=/bin/bash
SVN_EDITOR=vi
USER=petrucci
PATH=/opt/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:
/usr/X11/bin
PWD=/Users/petrucci
LANG=fr_FR
HOME=/Users/petrucci
LOGNAME=petrucci
$
```

*Cette exécution de la commande* ***printenv*** *montre les valeurs des variables d'environnement définies. Par exemple,* ***SVN_EDITOR****, variable contenant le nom de l'éditeur de texte utilisé par l'outil de développement collaboratif* ***svn*** *a pour valeur* ***vi****. Par conséquent, lorsque l'on travaille avec* ***svn****, l'éditeur utilisé est* ***vi****. On voit dans cet exemple que les variables d'environnement peuvent être liées à l'utilisation de logiciels particuliers.*

### 1.3.2 Création/modification de variables

La modification de la valeur d'une variable se fait en utilisant l'opérateur d'affectation `=`[2]. Si la variable n'existait pas encore, elle est créée.

2. Des rappels sur les délimiteurs de chaînes de caractères se trouvent en section 2.1.2.

**Exemple 6**

```
$ PS1="Bonjour>"
Bonjour>
```

*La variable d'environnement contenant le message d'invite* `PS1` *est modifiée. Sa nouvelle valeur est* `Bonjour>`*. Par conséquent, le message d'invite affiché après l'affectation de la variable correspond à cette nouvelle valeur.*

### 1.3.3 Exportation de variables

Les variables sont *a priori* locales, donc ne sont connues que du processus qui les a définies. Une variable peut être transmise aux processus fils. Pour ce faire, il faut l'« exporter » en utilisant la commande `export`.

**Exemple 7**

```
1 $ PS1="Bonjour>"
2 Bonjour>bash
3 $ exit
4 exit
5 Bonjour>export PS1
6 Bonjour>bash
7 Bonjour>
```

*Dans un premier* shell*, ligne 1, la variable* `PS1` *est modifiée. On crée ensuite un second* shell *(fils) ligne 2. Le message d'invite est alors celui par défaut (ligne 3). La valeur de la variable* `PS1` *n'a donc pas été transmise au processus fils. On quitte ce processus pour revenir au premier. On remarque, ligne 5, que la valeur de la variable* `PS1` *a bien été conservée. Cette variable est exportée. Puis on crée de nouveau un processus* shell *fils (ligne 6). Le message d'invite ligne 7 a bien été transmis au processus fils.*

### 1.3.4 Quelques variables d'environnement particulières

Dans cette section, nous décrivons les variables d'environnement les plus utilisées et ayant des caractéristiques particulières.

**Les variables `PATH` et `MANPATH`** contiennent une liste de références absolues de répertoires. Lorsque l'utilisateur veut exécuter un processus, s'il ne fournit que le nom de fichier (c'est-à-dire si l'utilisateur ne précise pas le chemin permettant d'accéder à une commande externe), le *shell* recherche la commande dans la liste des répertoires de `PATH`. La première commande trouvée est exécutée. Par conséquent, si plusieurs commandes se trouvant dans des répertoires différents possèdent le même nom, seule celle trouvée en premier sera exécutée, à moins de préciser sa référence complète.

La variable `MANPATH` a un comportement similaire pour rechercher les fichiers de documentation en ligne.

**Exemple 8** *Soit la variable* `PATH` *de l'exemple 5. Supposons que l'utilisateur veuille exécuter une commande* `la_commande`*. Le* shell *recherche un fichier de nom* `la_commande` *d'abord dans le répertoire* `/opt/local/bin`*. Si le fichier est trouvé, un processus correspondant est exécuté. Si le fichier n'est pas trouvé, la recherche est poursuivie dans* `/usr/bin`*, et ainsi de suite jusqu'à chercher le fichier dans le répertoire* `/usr/X11/bin`*. Si le fichier n'a toujours pas été trouvé, le* shell *déclare que la commande n'existe pas. Il se peut toutefois qu'elle existe mais se trouve dans un répertoire n'appartenant pas à la liste de la variable* `PATH`*.*

**Les variables `PS1` et `PS2`** décrivent le message d'invite, c'est-à-dire la chaîne de caractères affichée par le *shell* pour indiquer à l'utilisateur qu'il peut taper une commande. Dans le cas le plus général, c'est `PS1` qui est affichée. Quand la commande écrite par l'utilisateur n'est pas terminée, `PS2` est affiché.

La description du message d'invite peut comporter des chaînes de caractères fixes et des valeurs particulières. Celles-ci sont décrites par le caractère d'échappement \ suivi d'un caractère indiquant la signification de cette chaîne, comme expliqué dans le tableau 1.4.

TABLE 1.4 – Caractères spéciaux utilisés par `PS`

| Caractère | Signification |
|---|---|
| `\d` | date |
| `\t` | heure au format `hh:mm:ss` (*time*) |
| `\s` | *shell* utilisé |
| `\w` | référence du répertoire de travail (*working directory*) |
| `\W` | nom du répertoire de travail |
| `\u` | nom de l'utilisateur (*user name*) |
| `\h` | nom de la machine (*hostname*) |
| `\H` | nom complet de la machine |
| `\$` | le caractère `$` |
| `\\` | le caractère \ |

**Exemple 9**

```
1 $ PS1="\w␣>␣"
2 ~ > PS2="suite␣-␣"
3 ~ > ls /usr/bin |
4 suite - more
```

*Tout d'abord, ligne 1, la variable `PS1` est redéfinie pour indiquer le nom du répertoire de travail suivi d'un `>` (avec des espaces pour faire joli). Le second message d'invite `PS2` est lui aussi redéfini, ligne 2 pour afficher `suite -` . On constate que seul `PS1` est affiché aux lignes 2 et 3. En ligne 3, une commande est commencée, mais non terminée puisque la sortie du tube (`|`) n'est pas précisée. Par conséquent, ligne 4, l'interpréteur de commandes utilise le second message d'invite, `PS2`, pour signaler à l'utilisateur que la commande tapée n'était pas complète.*

### 1.3.5 Fichiers particuliers

Certains fichiers sont exécutés automatiquement à des moments spécifiques de l'utilisation du système.

Lors de l'*ouverture d'une session* par un utilisateur, le fichier `/etc/profile` est exécuté. Il contient des commandes communes à tous les utilisateurs et définissant donc des valeurs par défaut, mises en place par l'administrateur. Ensuite, un fichier similaire, propre à l'utilisateur est exécuté : `~/.bash_profile`, `~/.bash_login` ou `~/.login` (le premier trouvé parmi ces trois fichiers).

Lors de l'exécution d'un nouveau *shell*, le fichier `~/.bashrc` est exécuté.

Enfin, le fichier `~/.bash_logout` est exécuté *lorsque l'utilisateur se déconnecte.*

Ces fichiers permettent à l'utilisateur de personnaliser son environnement de travail.

# Chapitre 2

# Scripts *shell*

L'utilisation de scripts *shell* permet de créer des commandes personnalisées en vue d'automatiser l'exécution de suites de commandes. Cela présente le double avantage de pouvoir les réutiliser et de les paramétrer.

## 2.1 Écriture de scripts *shell*

Un *script shell* est un ensemble de commandes réunies dans un fichier exécutable.

### 2.1.1 Structure d'un script *shell*

Généralement, la première ligne du script a une syntaxe fixe qui définit l'interpréteur *shell* utilisé pour l'exécution :

```
#!référence_absolue_shell
```

**Exemple 10** *Un script dont la première ligne est :*

```
#!/bin/bash
```

*est exécutée avec l'interpréteur de commandes* `bash`*.*

Pour que le script puisse être exécuté, il faut s'assurer que le droit d'exécution est bien présent, et éventuellement changer les protections du fichier :

```
chmod +x nom_script
```

Il peut ensuite être exécuté comme une commande habituelle :

```
./nom_script
```

### 2.1.2 Délimiteurs de chaînes de caractères

Trois *délimiteurs* permettent (ou non) d'effectuer des opérations à l'intérieur même des chaînes de caractères :

- **'*chaîne*'** (apostrophe, ou *quote*) indique que la *chaîne* est utilisée telle quelle, sans aucune modification ;
- **"*chaîne*"** (guillemet, ou *double quote*) est utilisée pour que modifier la *chaîne* de caractères avant utilisation. En particulier, les noms de variables précédés de `$` sont remplacés par la valeur de ladite variable ;
- **`*chaîne*`** (apostrophe inversée, ou *backquote*) permet de considérer la *chaîne* de caractères comme une commande à effectuer, pour ensuite utiliser son résultat.

**Exemple 11** *L'exécution du script* `exemple11.sh` *:*

```
1 #!/bin/bash
2 var=variable
3 echo 'var=$var'
4 echo "var=$var"
5 echo `ls`
```

*affiche :*

```
1 var=$var
2 var=variable
3 exemple11.sh
```

*La ligne 1 du script définit l'interpréteur* shell *utilisé pour l'exécution. Ensuite, la ligne 2 affecte la chaîne de caractères* `variable` *à une variable* `var`*. Ces deux lignes ne donnent lieu à aucun affichage, contrairement aux trois lignes suivantes. L'utilisation des* apostrophes *dans la ligne 3 du script conduit à l'affichage de la ligne 1 (dans l'affichage) : la chaîne de caractères n'est pas modifiée et donc* `$var` *n'est pas remplacé par la valeur de la variable. Au contraire, lorsque l'on utilise des* guillemets *(ligne 4 du script et ligne 2 de l'affichage), la variable est interprétée avant que la chaîne de caractères ne soit utilisée. Enfin, la commande en ligne 5 utilise des* apostrophes inversées*. La chaîne de caractères est donc considérée comme une commande qui est exécutée. Le résultat de cette exécution est ensuite utilisé par la commande* `echo` *et conduit à l'affichage de la ligne 3.*

### 2.1.3 Les paramètres des commandes

Les commandes acceptent des *paramètres*, permettant ainsi des exécutions différentes. L'appel du script se fait alors par :

```
./nom_script paramètre_1 paramètre_2 ... paramètre_n
```

À l'intérieur même du script, les paramètres sont traités comme des variables, telles que le premier paramètre est référencé par `$1`, le second par `$2` et ainsi de suite jusqu'au neuvième qui est référencé par `$9`. Le nombre de paramètres n'est toutefois pas limité à 9, la commande `shift` permettant de décaler leur position et ainsi d'accéder aux paramètres suivants.

Un script manipule des variables qui sont locales au script. Elles ne sont pas « héritées » par les éventuels processus fils du script.

Différentes variables spécifiques, décrites dans le tableau 2.1 permettent d'obtenir des informations manipulables dans le script.

TABLE 2.1 – Variables spécifiques des scripts

| Variable | Description |
|---|---|
| `$1`, `$2`, ..., `$9` | paramètres de la commande |
| `$0` | référence de la commande |
| `$*` | liste de tous les paramètres de la commande |
| `$#` | nombre de paramètres passés lors de l'appel |
| `$$` | numéro du processus *shell* correspondant à la commande |
| `$?` | code de retour de la dernière commande exécutée |

**Exemple 12** *Soit le script shell* `exemple12` *suivant :*

```
#!/bin/bash
echo "paramètre 1=$1"
echo "paramètre 2=$2"
echo "paramètre 3=$3"
echo "commande=$0"
echo "tous les paramètres=$*"
echo "nombre de paramètres=$#"
echo "numéro de processus=$$"
echo "valeur de retour de la commande précédente=$?"
```

*L'exécution de* `exemple12 toto titi tata tutu` *produit l'affichage suivant, qui illustre les différentes variables du tableau 2.1 :*

```
paramètre 1=toto
paramètre 2=titi
paramètre 3=tata
commande=./exemple12
tous les paramètres=toto titi tata tutu
nombre de paramètres=4
numéro de processus=811
valeur de retour de la commande précédente=0
```

## 2.2 Structures de contrôle

Le *shell* dispose de commandes pour effectuer des tests et des répétitions.

### 2.2.1 Tests

Les tests permettent de comparer la valeur de variables ou de vérifier le type et les protections associées à un fichier. Ils ont la forme :

**[** ***chaîne***$_1$ ***op chaîne***$_2$ **]**

où *chaîne*$_1$ et *chaîne*$_2$ sont des *chaînes de caractères à comparer* et *op* est un *opérateur de comparaison* opérant sur différents types. La *chaîne*$_1$ est omise lorsque l'on examine les propriétés d'un fichier référencé par *chaîne*$_2$ ou lorsque l'on utilise un opérateur unaire [1].

- Comparaison de chaînes de caractères : = (égalité), **!=** (différence), **-n** (non vide), **-z** (vide) ;
- Comparaison de valeurs numériques : **-eq** (égalité, *equal*), **-ne** (différence, *not equal*), **-gt** (strictement supérieur), *greater than*), **-ge** (supérieur ou égal, *greater than or equal*), **-lt** (strictement inférieur, *lower than*), **-le** (inférieur ou égal, *lower than or equal*) ;
- Tests sur les fichiers : **-d** (répertoire, *directory*), **-f** (fichier, *file*), **-r**, **-w**, **-x** (permissions sur les fichiers, respectivement en lecture, écriture et exécution).

Ces tests sont principalement utilisés pour vérifier que certaines conditions sont satisfaites. Un test simple de la forme « si... alors... sinon... finsi » s'écrit :

```
if liste_commandes1
then liste_commandes2
else liste_commandes3
fi
```

1. Attention : les `[`, `]` et opérateur de comparaison doivent obligatoirement être entourés d'espaces pour pouvoir être interprétés correctement par le *shell*.

**Exemple 13** *Le script suivant teste si le premier paramètre passé lors de l'appel est la chaîne de caractères « fichier » et si c'est le cas, teste si le second paramètre est le nom d'un fichier du répertoire de travail.*

```
#!/bin/bash
if [ "$1" = 'fichier' ]
then if [ -f "$2" ]
     then echo "$2 est un fichier"
     else echo "$2 non trouvé"
     fi
fi
```

*On note, dans cet exemple, que des guillemets ont été mis dans les tests autour de* `$1` *et* `$2`*. Lorsque ces variables sont interprétées, elles sont d'abord remplacées par leur valeur avant d'être utilisées. Si elles n'ont pas de valeur, elles sont remplacées par la chaîne vide. Si l'on n'avait pas mis les guillemets, le test* `[ -f $2 ]` *serait remplacé — dans le cas où* `$2` *est vide — par* `[ -f  ]`*. L'opérateur de test unaire* `-f` *n'aurait alors pas de paramètre, ce qui conduirait à une erreur.*

Lorsque de nombreux cas sont à traiter, on préférera utiliser la commande `case` plutôt qu'une série de `if` imbriqués. Sa syntaxe est :

```
case chaîne in
    motif_1 ) liste_commandes_1 ;;
    ...
    motif_n ) liste_commandes_n ;;
    * ) liste_commandes_autres_cas ;;
esac
```

La *chaîne* de caractères est comparée d'abord à l'expression rationnelle décrite par le $motif_1$. Si elle correspond, la $liste\_commandes_1$ est exécutée, puis on sort de la structure `case` (c'est-à-dire que l'on continue l'exécution à la ligne suivant le mot-clef `esac`). Sinon, la comparaison est effectuée avec $motif_2$ et ainsi de suite jusqu'au dernier motif, $motif_n$. Si la *chaîne* de caractères ne correspond à aucun motif, on peut lui appliquer des instructions par défaut ($liste\_commandes\_autres\_cas$), en la comparant au motif `*` qui correspond à toutes les chaînes de caractères. On remarque que les listes de commandes sont terminées par `;;`. Cela permet de ne pas les confondre avec une mise en séquence de plusieurs instructions qui ne contient qu'un seul `;`.

L'exemple 16 illustre l'utilisation de la structure de cas.

### 2.2.2 Boucles

Les opérations à effectuer dans un script peuvent être répétitives. Pour cela, on utilise des *structures de boucles* qui peuvent travailler en particulier sur les paramètres d'appel de la commande.

La boucle de type « tant que… faire… fintq » repète des instructions tant que sa condition est satisfaite :

```
while liste_commandes_1
do
    liste_commandes_2
done
```

La condition s'exprime par une $liste\_commandes_1$ à effectuer. Si celle-ci s'est bien passée, elle renvoie une valeur de retour positive que la structure `while` interprète comme une condition satisfaite. La $liste\_commandes_2$ est alors exécutée, et ainsi de suite jusqu'à non-satisfaction de la condition. L'exécution se poursuit alors à la ligne suivant `done`.

**Exemple 14** *Le script suivant affiche les valeur d'une variable* `i` *qui décroît à chaque itération de la boucle, de 5 jusqu'à 1.*

```
#!/bin/bash
i=5
while [ $i -gt 0 ]
do
    echo "i=$i"
    i=$(($i-1))
done
```

La boucle de type « pour... faire... finpour » repète des instructions en parcourant une liste de valeurs :

```
for variable in liste_chaînes
do
    liste_commandes
done
```

La *variable* prend successivement les valeurs décrites dans *liste_chaînes*. À chaque itération, les instructions dans *liste_commandes* sont exécutées. Lorsque *variable* a pris toutes les valeurs à l'intérieur de la *liste_chaînes*, on sort de la boucle `for` (c'est-à-dire que l'exécution se poursuit à la ligne suivant `done`).

**Exemple 15** *Dans ce script, une variable `i` prend successivement les valeurs `1`, `5` et `toto`. À chaque itération de la boucle, la valeur de `i` est affichée.*

```
#!/bin/bash
for i in 1 5 toto
do
    echo "$i"
done
```

Lorsque la *liste_chaînes* n'est pas précisée, c'est par défaut la liste des paramètres d'appel du script qui est parcourue. Par conséquent :

```
for variable
do
    liste_commandes
done
```

est équivalent à :

```
for variable in $*
do
    liste_commandes
done
```

## 2.3 Autres commandes des scripts *shell*

Toutes les commandes habituellement utilisées dans la ligne de commande peuvent l'être dans un script *shell*. Parmi les commandes utilisées principalement dans les script, se trouvent `set`, `read` et `exit` :

- **set *chaîne*** remplace la liste des paramètres par la *chaîne* ;
- **read *liste_variables*** lit des données fournies par l'entrée standard et attribue ces valeurs aux variables de la *liste_variables* ;

— **exit *valeur*** termine l'exécution courante (donc le processus en train de s'exécuter) et renvoie la *valeur* entière (argument de `exit`) comme code de retour.

Les exemples suivants illustrent les différentes commandes.

**Exemple 16** *Le script suivant examine les fichiers du répertoire de travail. Pour chacun d'eux, il indique si c'est un répertoire. Si ce n'est pas le cas, il propose à l'utilisateur d'afficher le contenu du fichier, ce qu'il fait en cas de réponse est positive, sinon il affiche un message suivant la réponse de l'utilisateur.*

```
1  #!/bin/bash
2  set `ls`
3  for fich
4  do
5      if [ -d $fich ]
6      then echo "$fich est un répertoire"
7      else echo "Voulez-vous voir le contenu de $fich ?"
8           read rep
9           case $rep in
10              o|O ) cat $fich ;;
11              n|N ) echo "Passage au fichier suivant" ;;
12              * ) echo "Réponse incorrecte (traitée comme non)";;
13          esac
14     fi
15 done
```

*Tout d'abord, en ligne 2, la liste des paramètres est remplacée par le résultat de l'exécution de `ls`. Ensuite, ligne 3, une boucle commence, avec une variable `fich` qui prend sa valeur dans la liste des paramètres. Pour chaque itération de la boucle, un test est effectué ligne 5 pour vérifier si le fichier est un répertoire. Si c'est le cas, un message en ce sens est affiché ligne 6. Sinon, le script demande à l'utilisateur (ligne 8) s'il veut voir le contenu du fichier. Suivant le cas (lignes 9–13), le contenu du fichier ou un message est affiché.*

*On remarque qu'aux lignes 10 et 11, deux motifs sont proposés, séparés par `|` (*ou*). L'utilisateur peut donc rentrer `o` ou `O` pour une réponse positive, et `n` ou `N` pour une réponse négative.*

**Exemple 17** *Soit le script `exemple17` suivant. L'appel se fait avec un paramètre, sous la forme `exemple17 nom_prog`. Ce script arrête l'exécution de tous les processus correspondant au programme `nom_prog`.*

```
1  #!/bin/bash
2  chaine=$1
3  ps | grep $chaine | grep -v grep |
4    while read pid reste
5    do
6        kill -9 $pid
7    done
```

*Tout d'abord, la valeur du paramètre est stockée dans une variable `chaine` (ligne 2). Ceci permet de passer correctement le paramètre aux sous-*shells *créés lors de l'enchaînement de tubes de la ligne 3. Un processus exécute la commande `ps` et transmet la liste des processus à la commande `grep chaine`. Par conséquent, les lignes du résultat de `ps` correspondant au paramètre d'appel sont conservées. Parmi ces processus peut se trouver le `grep` lui-même. La ligne correspondante est supprimée de l'ensemble des lignes par le `grep -v`. Puis toutes ces lignes sont envoyées à la boucle `while` de la ligne 4. La condition lit la ligne avec `read`, et la découpe en stockant le premier élément dans la variable `pid` et les autres dans une variable `reste`. Le premier élément des lignes retournées par `ps` étant l'identificateur de processus (pid,* process identifier*), celui-ci peut être utilisé pour terminer le processus par la commande `kill` ligne 6.*

# Chapitre 3

# Gestion des utilisateurs

La *gestion des utilisateurs* consiste à créer de nouveaux utilisateurs dans le système, les affecter à des *groupes* d'utilisateurs selon les droits que l'on souhaite leur attribuer, et supprimer les utilisateurs n'ayant plus de raison d'accéder au système informatique (par exemple suite à un changement de travail).

## 3.1 Création et suppression d'un utilisateur

Les utilisateurs sont identifiés par un « login » et un mot de passe associé. Ces informations sont conservées dans le fichier `/etc/passwd` dans lequel chaque ligne correspond à un compte utilisateur.

### 3.1.1 Fichiers associés aux mots de passe

Chaque ligne de `/etc/passwd` comporte 7 champs séparés par des `:`, représentant, dans l'ordre, les informations suivantes :

1. **uname** : nom de connexion ;
2. **password** : mot de passe crypté ;
3. **UID** (*user identifier*) : numéro unique associé à l'utilisateur ;
4. **GID** (*group identifier*) : numéro d'identification du groupe principal quel l'utilisateur appartient [1] ;
5. **GECOS** : texte de description de l'utilisateur, habituellement nom et prénom ;
6. **homedir** (*home directory*) : répertoire privé de l'utilisateur ;
7. **login shell** : commande exécutée lors de la connexion (en général `/bin/bash`).

Le fichier `/etc/passwd` est lisible par tous les utilisateurs. Malgré l'utilisation du cryptage du mot de passe, cela laisse la porte ouverte à des utilisateurs malveillants. Par conséquent, pour des raisons de sécurité, LINUX utilise un second fichier, nommé `/etc/shadow`, lisible uniquement par l'administrateur du système (utilisateur `root`), dans lequel est stocké le mot de passe crypté. Le champ « password » de `/etc/passwd` contient alors la lettre `x`.

Chaque ligne de `/etc/shadow` comporte 9 champs séparés par des `:`, représentant, dans l'ordre, les informations suivantes :

1. **uname** : nom de connexion ;
2. **password** : mot de passe crypté. Si ce champ est vide, aucun mot de passe n'est précisé pour l'utilisateur. S'il contient `*`, le compte est désactivé et aucune connexion n'est possible ;

1. La gestion des groupes est détaillée dans la section 3.2.

3. nombre de jours entre le 1er janvier 1970 et le *dernier changement du mot de passe* ;
4. nombre de *jours minimum entre deux changements de mot de passe*. Un `0` indique que l'utilisateur peut changer son mot de passe à n'importe quel moment ;
5. nombre de *jours pendant lesquels le mot de passe est valide*. La valeur 99999 est utilisée pour indiquer que le mot de passe ne doit pas nécessairement être changé ;
6. nombre de *jours précédent l'expiration du mot de passe*, utilisé pour prévenir l'utilisateur ;
7. nombre de *jours de validité du compte après expiration du mot de passe*. À l'issue de ce délai, le compte est désactivé ;
8. *date d'expiration du compte*, en nombre de jours depuis le 1er janvier 1970 ;
9. champ *réservé* pour un usage futur.

### 3.1.2 Étapes de création et suppression d'un utilisateur

Lors de la création d'un nouvel utilisateur, plusieurs opérations ont lieu :
— ajout de la ligne correspondant à l'utilisateur dans le fichier `/etc/passwd` ;
— ajout de la ligne correspondant à l'utilisateur dans le fichier `/etc/shadow` ;
— ajout de l'utilisateur dans la liste des membres du groupe auquel il appartient ;
— création du répertoire privé de l'utilisateur ;
— recopie éventuelle des fichiers d'environnement par défaut ;
— création de la boîte aux lettres de l'utilisateur dans le répertoire `/var/spool/mail`.

Pour *créer un nouvel utilisateur*, on peut soit éditer tous les fichiers et créer les répertoires nécessaires, soit utiliser la commande `useradd` qui effectue toutes les opérations. Cette commande possède de nombreuses options, permettant de configurer les différents champs des fichiers `/etc/passwd` et `/etc/shadow`.

La création de l'utilisateur est complétée par l'affectation d'un mot de passe en utilisant la commande `passwd`.

**Exemple 18** *Les commandes suivantes créent un nouvel utilisateur `toto`, avec `/home/toto` pour répertoire privé, et le* shell `bash` *:*

```
$ useradd -d /home/toto -s /bin/bash toto
$ passwd toto
```

La *suppression d'un utilisateur* est l'opération inverse, détruisant le répertoire privé, la boîte aux lettres ainsi que les lignes des fichiers `/etc/shadow` et `/etc/passwd` correspondant à l'utilisateur à supprimer du système. La commande `userdel` effectue toutes les opérations nécessaires.

**Exemple 19** *Supprimons l'utilisateur `toto` :*

```
$ userdel toto
```

## 3.2 Groupes d'utilisateurs

La définition de *groupes d'utilisateurs* permet de leur attribuer des droits similaires sur certains fichiers [2]. Un même utilisateur peut appartenir à un ou plusieurs groupes. La liste des groupes existants est définie dans le fichier `/etc/group` dont chaque ligne comporte 4 champs séparés par des `:`, selon la structure suivante :

1. **group name** : nom du groupe ;

2. Pour les détails sur la gestion des droits d'accès aux fichiers, voir la section 3.3.

2. **password** : mot de passe associé au groupe. Un `x` indique que le mot de passe crypté se trouve dans un fichier *« shadow »*, `/etc/gshadow`. L'absence de valeur correspond à une absence de mot de passe ;
3. **GID** (*group identifier*) : numéro d'identification du groupe ;
4. **users** : liste des noms d'utilisateurs appartenant au groupe, séparés par des `,`.

Le fichier `/etc/gshadow` comporte également 4 champs séparés par des `:` tels que :

1. **group name** : nom du groupe ;
2. **password** : mot de passe crypté associé au groupe ;
3. **administrators** : liste des administrateurs du groupe, séparés par des `,` ;
4. **users** : liste des noms d'utilisateurs appartenant au groupe, séparés par des `,`.

### 3.2.1 Ajout d'un groupe

L'ajout d'un groupe se fait en utilisant la commande `groupadd` qui modifie à la fois le fichier `/etc/group` et `/etc/gshadow`. La suppression d'un groupe est effectuée à l'aide de `groupdel`. Cette commande ne peut avoir lieu que si aucun utilisateur n'a ce groupe comme groupe principal.

**Exemple 20** *Les commandes suivantes ajoutent un groupe* `rt1` *de numéro* 1001*, puis le supprime :*

```
$ groupadd -g 1001 rt1
$ groupdel rt1
```

### 3.2.2 Changement de groupe

Lorsqu'un utilisateur se connecte, il est rattaché à son groupe principal, déclaré lors de la création de son compte. Toutefois, il est parfois souhaitable de travailler comme si l'utilisateur appartenait à un autre groupe, par exemple avec des droits plus restreints, évitant ainsi des modifications indues.

La commande `newgrp` permet de modifier le groupe utilisé. Cette modification est temporaire et donne lieu à la création d'un processus *shell* fils [3]. Lorsque ce processus fils est quitté, l'utilisateur est de nouveau rattaché au groupe précédent.

## 3.3 Droits des utilisateurs

### 3.3.1 Droits associés aux fichiers

Les fichiers peuvent être accédés en *lecture* (***r**ead*), en *écriture* (***w**rite*) ou être *exécutés* (*e**x**ecute*). Il y a par conséquent 3 types d'opérations pour l'accès aux fichiers.

De plus, l'utilisateur souhaitant accéder à un fichier peut être son *propriétaire* (***u**ser*), membre du *groupe* (***g**roup*) auquel le fichier est associé, ou un *autre* utilisateur (***o**ther*). Ceci conduit à 3 types d'utilisateurs pour chaque fichier.

Il y a alors 9 combinaisons définissant les modes d'accès à un fichier, selon le type d'opération et le type d'utilisateur. Lors de l'utilisation de la commande `ls -l`, les droits d'accès au fichier sont indiqués par les lettres , `w` et `x`, groupées par type d'utilisateur, avec dans l'ordre `u`, `g` et `o`. Lorsqu'un droit n'est pas affecté, un `-` est affiché à la place de la lettre.

3. voir la section 4.4 sur la création de processus.

**Exemple 21** *Les protections suivantes, pour un fichier* `fich` *:*

$\underbrace{rwx}_{u}\underbrace{r-x}_{g}\underbrace{---}_{o}$

*indiquent que le propriétaire peut lire, écrire et exécuter* `fich`*, qu'un membre du groupe ne peut que le lire et l'exécuter, et enfin qu'un autre utilisateur n'a pas accès à ce fichier.*

Lorsque le fichier est un *répertoire*, l'exécution a une signification particulière : il est possible de *« traverser »* le répertoire, c'est-à-dire se déplacer à l'intérieur.

### 3.3.2 Modification des droits

La commande `chmod` (*change mode*) modifie les droits associés aux fichiers de *liste_fichiers*.
`chmod droits liste_fichiers`
Le changement de droits peut être décrit de deux manières :

1. en utilisant une représentation *octale*. Chaque élément de la chaîne décrivant les droits associés au fichier vaut en binaire 1 s'il est autorisé, 0 s'il est interdit. Le nombre binaire obtenu est alors traduit en octal.
2. sous la forme *utilisateurs*`~`*opérations* où :
   — *utilisateur* décrit le type d'utilisateur pour lequel la modification est effectuée. Il prend une ou plusieurs valeurs parmi `u`, `g` et `o` ;
   — `~` prend une seule valeur parmi `+`, `-` et `=` et indique que des droits vont être respectivement ajoutés, supprimés ou indiqués exactement pour ces types d'utilisateurs ;
   — *opérations* décrit les droits d'accès aux fichiers, prenant une ou plusieurs valeurs parmi `r`, `w` et `x`.

**Exemple 22** *Soit le fichier* `fich` *de l'exemple 21. Utilisons les deux possibilités de modification de droits offertes par* `chmod` *pour obtenir les droits de lecture et écriture pour l'utilisateur et le groupe, et de lecture seule pour les autres.*

1. *la nouvelle chaîne décrivant les droits devra être :* `rw-rw-r--`*. Elle s'écrit* `110110100` *en binaire, soit* `664` *en octal. On peut donc changer les droits avec :*

```
$ chmod 664 fich
```

2. *en comparant les droits du fichier avant et après modification, on constate que l'on souhaite retirer les droits d'exécution au propriétaire et au groupe, ajouter le droit d'écriture au groupe et ajouter le droit de lecture aux autres. Ceci peut être fait avec la suite de commandes :*

```
$ chmod ug-x fich
$ chmod g+w fich
$ chmod o+r fich
```

   *Une autre possibilité est de fixer les droits de lecture et d'écriture pour le propriétaire et le groupe, puis d'ajouter le droit de lecture aux autres :*

```
$ chmod ug=rw fich
$ chmod o+r fich
```

Les *droits attribués par défaut à un fichier* peuvent être modifiés en utilisant `umask`. Cette commande ne modifie pas les droits des fichiers existants, mais seulement de ceux qui seront créés par la suite.

Les paramètres de cette commande peuvent être le *code octal des droits non attribués* ou la *description symbolique des droits attribués*, c'est-à-dire avec les lettres `u`, `g` et `o` pour les utilisateurs, `r`, `w` et `x` pour les droits d'accès. Lorsqu'aucun paramètre n'est fourni, le masque courant est affiché. L'option `-S` permet d'afficher les droits attribués par défaut de manière symbolique.

**Exemple 23** *La séquence d'instructions suivante présente les différentes utilisations de* `umask`*.*

```
$ umask
0022
$ umask -S
u=rwx,g=rx,o=rx
$ umask 027
$ umask -S
u=rwx,g=rx,o=
$ umask u=rwx,g=r,o=
$ umask
0037
$ umask -S
u=rwx,g=r,o=
```

### 3.3.3 Changement de propriétaire ou de groupe

La commande `chgrp` (*change group*) modifie le groupe associé à des fichiers :
`chgrp` *`groupe liste_fichiers`*
Il est également possible de changer le propriétaire de fichiers à l'aide de `chown` (*change owner*) :
`chown` *`utilisateur`*`[:`*`groupe`*`]` *`liste_fichiers`*

## 3.4 Quotas

### 3.4.1 Quotas des utilisateurs

Des *quotas* peuvent être mis en place pour limiter l'espace disque alloué aux utilisateurs, et le nombre de fichiers. Ils comprennent deux limites :

— une *limite douce* (*soft limit*) qui peut être dépassée temporairement ;
— une *limite dure* (*hard limit*) qui ne peut pas être dépassée.

Pour mettre en place des quotas utilisateurs, il faut s'assurer de monter la partition avec le options idoines (`usrquota` et `grpquota`), et que deux fichiers `quota.user` et `quota.group`, avec les droits 600, soient présents à la *racine de la partition* concernée.

Il est ensuite possible de les activer, les désactiver, les éditer et vérifier leur utilisation avec les commandes `quotaon`, `quotaoff`, `edquota` et `repquota`, respectivement.

### 3.4.2 Limites du système

Les limites générales du système permettent d'éviter qu'un utilisateur ne monopolise la machine. Elles sont visualisées et modifiées avec la commande `ulimit`.

**Exemple 24** *La séquence d'instructions suivante présente l'affichage des différentes limites présentes sur un système et la modification de l'une d'entre elles.*

```
$ ulimit -a
-t: cpu time (seconds)              unlimited
-f: file size (blocks)              unlimited
-d: data seg size (kbytes)          unlimited
-s: stack size (kbytes)             8192
-c: core file size (blocks)         0
-v: address space (kbytes)          unlimited
-l: locked-in-memory size (kbytes)  unlimited
-u: processes                       2784
-n: file descriptors                256
```

```
$ ulimit -n 512
$ ulimit -n
512
```

# Chapitre 4

# Gestion des processus

## 4.1 Notion de processus

Un *processus* est une *exécution* d'une commande. Il ne faut pas confondre les notions de « processus » et de « programme » ou de « commande ». Un *programme* est un fichier exécutable stocké sur disque. Un programme peut être exécuté plusieurs fois simultanément. Dans ce cas, il y a autant de processus que d'exécutions du programme. C'est par exemple le cas lorsque plusieurs terminaux sont ouverts : chacun exécute une occurrence de l'interpréteur de commandes (*shell*), chaque occurrence étant un processus distinct des autres.

Les processus existant à un instant donné sont *indépendants* les uns des autres (en particulier, ils ne partagent *a priori* pas la mémoire), et le processeur leur est alloué par le système de manière imprévisible pour l'utilisateur.

## 4.2 Types de processus

Tout processus est associé à un utilisateur et au terminal depuis lequel il a été créé, à l'exception des processus« système » ayant des objectifs spécifiques et le plus souvent créés au démarrage de la machine. Les deux types de processus sont tels que :

— un *processus système* n'est pas attaché à un terminal. Il est créé au démarrage de la machine ou à des dates fixées par le super-utilisateur. Parmi eux, les processus « démons » (*daemon*), tels que le gestionnaire de file d'impression, ne sont interrompus que lors de l'arrêt du système ou sur demande du super-utilisateur ;

— un *processus utilisateur* est lancé par un utilisateur depuis un terminal. Lors de l'ouverture d'une session, un processus est lancé, comme indiqué pour l'utilisateur dans le fichier `/etc/passwd` (généralement un interpréteur de commandes, pas exemple `/bin/bash`).

## 4.3 Caractéristiques des processus

À chaque processus sont associées des informations et une zone mémoire permettant l'exécution. Le système gère une *table des processus* contenant une entrée par processus. Cette entrée regroupe toutes les informations nécessaires au système pour la gestion du processus. Lors du lancement d'un processus l'entrée associée est créée dans la table, et elles est détruite lorsque le processus se termine.

### 4.3.1 Principales caractéristiques

Les principales caractéristiques associées à un processus sont :

— **PID** (*process identifier*) : un numéro unique identifiant le processus ;

— **PPID** (*parent process identifier*) : identifiant du processus père ;
— **RUID** (*real user identifier*) : le numéro de l'utilisateur propriétaire réel du processus ;
— **EUID** (*effective user identifier*) : le numéro de l'utilisateur effectif du processus ;
— **RGID** (*real group identifier*) : le numéro du groupe propriétaire réel du processus ;
— **EGID** (*effective group identifier*) : le numéro du groupe effectif du processus ;
— **TERM** (*terminal*) : le terminal auquel le processus est rattaché.

Une distinction est faite entre l'utilisateur qui a lancé le processus, appelé *propriétaire réel* et le l'utilisateur avec les droits duquel le processus est exécuté, appelé *propriétaire effectif*. En général, ces utilisateurs sont identiques. Toutefois, un bit spécial, le bit « SET-UID » (*set user identifier*), peut être positionné pour un fichier exécutable. Ce mécanisme permet d'exécuter un programme avec les permissions d'accès de son propriétaire. Une distinction similaire est faite entre le *groupe réel* et le *groupe effectif*, avec positionnement éventuel du bit « SET-GID » (*set group identifier*).

**Exemple 25** *La commande* `/usr/bin/passwd` *permettant à un utilisateur de changer son mot de passe a le bit SET-UID positionné (*`s`* à la place de* `x` *dans les droits du propriétaire) :*

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  wheel  70352 28 oct 04:28 /usr/bin/passwd
```

*En effet, cette commande doit pouvoir modifier les fichiers associés au mots de passe. Un utilisateur quelconque n'a pas ce droit car il pourrait modifier les informations concernant les autres utilisateurs. Par contre, le super-utilisateur,* `root`*, peut effectuer ces modifications. L'exécution de* `passwd` *s'effectue donc avec les droits de* `root`*.*

### 4.3.2 Mémoire allouée à un processus

Lors du *lancement* d'un processus, le programme correspondant est chargé en mémoire en vue de son exécution. De plus, des zones sont allouées en mémoire pour la gestion des données propres au processus.

La *mémoire allouée* à un processus se divise en 4 zones :
— le *segment de texte* contient les instructions à exécuter ;
— le *segment de données* est une zone mémoire gérée par le système contenant les informations lui permettant de contrôler le processus. La machine doit en effet assurer l'exécution de plusieurs processus de manière transparente pour les utilisateurs. Pour ce faire, elle alloue le processeur alternativement aux différents processus — sans que l'ordre d'exécution soit *a priori* prévisible. Pendant qu'un processus s'exécute, les autres sont suspendus et attendent que le processeur leur soit alloué. Le système doit donc avoir une trace de l'état du système lors de la suspension du processus. C'est le rôle du segment de données qui contient entre autres l'état des *registres*.
— la *zone de données statiques* contient les éléments connus au démarrage du processus et qui seront présents durant toute l'exécution. On y trouve les *variables globales* du programme.
— la *zone de données dynamique* contient tous les objets non-permanents manipulés par le processus. Cette zone est elle-même divisée en 2 parties :
  — une *pile* utilisée pour l'allocation des *variables locales*, des *paramètres de fonctions* ;
  — un *tas* dans lequel se fait l'*allocation dynamique* de variables, par exemples celles accédées par des *pointeurs*.

Sous Unix, les programmes sont en général *réentrants*, c'est-à-dire que lorsque plusieurs processus exécutent le même programme, une seule copie du *segment de texte* est placée en mémoire et est utilisée par tous ces processus. Par contre, chacun d'eux a ses propres segment et zones de données. Ce mécanisme permet d'une part d'« économiser » de la mémoire en ne dupliquant pas les exécutables, et d'autre part de garantir l'indépendance des différentes exécutions simultanées d'un même programme.

## 4.4 Mécanisme d'exécution

L'exécution d'un processus fait intervenir plusieurs mécanismes : le clonage, la substitution et la suspension.

La *création* d'un processus est effectuée par *clonage*, c'est-à-dire duplication, d'un processus existant, appelé *processus père*. L'opération de clonage est réalisée par une fonction système `fork`. Le processus créé, appelé *processus fils*, est la copie conforme du processus père : il dispose des mêmes instructions, d'une copie de toutes les données, y compris des valeurs des registres — en particulier du même compteur ordinal indiquant la prochaine instruction à exécuter.

Une fois le processus fils créé, un mécanisme de *substitution* (fonction système de la famille `exec`) permet de remplacer l'image mémoire d'un processus par une nouvelle image (nouveau programme et de nouvelles données) construite à partir d'un fichier exécutable. L'exécution du processus fils se poursuit au début du nouveau programme.

Le mécanisme de *suspension* permet à un processus père d'attendre (fonction système `wait`) la terminaison d'un (ou plusieurs) processus fils.

## 4.5 Enchaînement de processus

On peut enchaîner des processus soit de manière séquentielle, soit en les exécutant en parallèle et en les faisant communiquer. De plus, un processus peut s'exécuter en tâche de fond.

### 4.5.1 Enchaînement séquentiel

Des processus indépendants les uns des autres peuvent être *exécutés séquentiellement* en les séparant par un `;` :

$comm_1$ `;` $comm_2$ `; ...;` $comm_n$

Le premier processus, correspondant à la commande $comm_1$, est lancé à l'appel de cette commande. Lorsque son exécution termine, un nouveau processus, exécutant la commande $comm_2$ est lancé. Ainsi de suite jusqu'à la fin de l'exécution de $comm_n$.

Le mécanisme d'exécution est illustré dans la figure 4.1. Le processus *shell* dans lequel la commande est appelée commence par exécuter un `fork`. Le processus père est suspendu tandis que le processus fils fait un `exec` pour remplacer son exécutable par celui de la première commande à exécuter. Lorsque celle-ci se termine, elle signale la fin d'exécution au processus père qui est réveillé et reprend son exécution. Ainsi de suite pour les autres commandes.

### 4.5.2 Enchaînement parallèle

Les tubes permettent d'enchaîner des processus communicant entre eux. Les commandes sont alors séparées par un `|` (tube ou *pipe*) :

$comm_1$ `|` $comm_2$ `| ...|` $comm_n$

Les processus sont lancés simultanément. L'entrée standard est l'entrée du premier processus, correspondant à la commande $comm_1$, et sa sortie l'entrée de $comm_2$. Ainsi de suite, la sortie de $comm_{n-1}$ est l'entrée de $comm_n$. Enfin la sortie de $comm_n$ est la sortie standard.

L'ordre de terminaison des processus est alors quelconque.

Le mécanisme d'exécution est illustré dans la figure 4.2.

### 4.5.3 Tâches de fond

Une commande *comm* peut être lancée en tâche de fond (encore appelée *en arrière-plan*) avec :

*comm* `&`

Dans ce cas, l'interpréteur de commandes n'est pas suspendu pendant l'exécution de la commande *comm*, comme indiqué sur la figure 4.3.

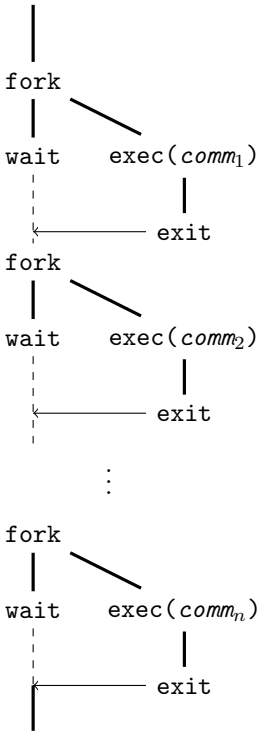


FIGURE 4.1 – Enchaînement séquentiel de processus

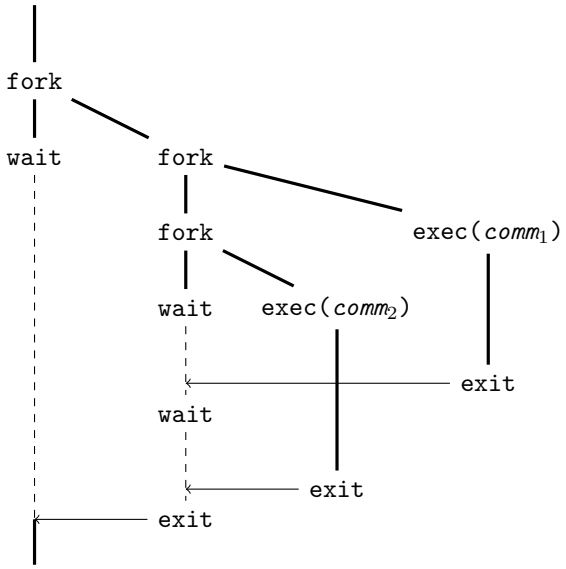


FIGURE 4.2 – Enchaînement parallèle de processus : *comm*$_1$ | *comm*$_2$

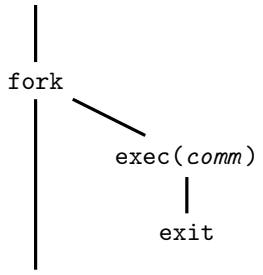


FIGURE 4.3 – Exécution en tâche de fond

## 4.6 Signaux

Les *signaux* permettent d'avertir un processus qu'un événement important s'est produit dans son environnement d'exécution. C'est un mécanisme utilisé en particulier par le système dans le cas d'erreurs.

Les signaux sont identifiés par un nombre entier, et disposent également d'un nom. La *liste des signaux disponibles* peut être obtenue par la commande `kill -l`.

TABLE 4.1 – Principaux signaux

| Numéro | Nom | Description du signal |
|---|---|---|
| 1 | SIGHUP | (*hang up*) émis à tous les processus associés à un terminal qui se déconnecte. |
| 2 | SIGINT | (*interrupt*) signal d'interruption émis lors de la frappe du caractère <CTRL-c>. |
| 3 | SIGQUIT | (*quit*) signal d'interruption émis lors de la frappe du caractère <CTRL-\>. |
| 4 | SIGILL | (*illegal*) émis en cas d'instruction illégale. |
| 9 | SIGKILL | (*kill*) tue un processus quel que soit son état. |
| 13 | SIGPIPE | (*pipe*) émis en cas d'écriture dans un tube sans processus lecteur. |
| 14 | SIGALRM | (*alarm*) signal associé à une horloge. |
| 15 | SIGTERM | (*termination*) émis lors de la terminaison normale d'un processus. |

## 4.7 Commandes de gestion des processus

### 4.7.1 Commandes de base

Les principales commandes de base [1] assurant la gestion des processus sont récapitulées dans le tableau 4.2.

La commande `pstree` permet de *visualiser l'arborescence des processus* et donc d'identifier facilement les relations entre processus pères et fils.

**Exemple 26** *Dans l'exécution suivante, on voit apparaître des processus* `more` *(ligne 24) et* `pstree` *(ligne 25) qui ont le même père. Ceci est cohérent avec le schéma d'exécution des tubes décrit dans la figure 4.2.*

1. Ces commandes ont été vues en détail dans le module I1, et sont seulement rappelées ici.

TABLE 4.2 – Commandes de base de gestion de processus

| Commande | Description |
|---|---|
| `bg` | (*background*) mise en *arrière-plan* d'un processus. |
| `fg` | (*foreground*) mise en *avant-plan* d'un processus. |
| `jobs` | liste des travaux actifs. |
| `ps` | (*process status*) donne les informations sur les processus. |
| `top` | donne les informations sur les processus, de manière interactive. |

```
$ ls | more | pstree
init-+-apache2-+-5*[apache2---12*[{apache2}]]
     |          `-10*[apache2]
     |-atd
     |-2*[automount]
     |-cron---cron---sh---run-parts---minieddie---supervision-min---sleep
     |-dbus-daemon
     |-dhclient3
     |-6*[getty]
     |-inetd
     |-master-+-pickup
     |        `-qmgr
     |-munin-node
     |-mysqld_safe-+-logger
     |             `-mysqld---11*[{mysqld}]
     |-10*[nc]
     |-ntpd
     |-portmap
     |-rpc.statd
     |-sshd-+-3*[sshd---sshd---nc]
     |      |-sshd---sshd---bash
     |      |-sshd---sshd---sftp-server
     |      `-sshd---sshd---bash-+-bash
     |                           |-more
     |                           `-pstree
     |-syslog-ng
     |-udevd
     `-ypbind---2*[{ypbind}]
```

### 4.7.2 Commandes associées aux signaux

À chaque signal est associée une *action par défaut*. Celle-ci peut être changée grâce à la commande `trap` :

`trap [comm] signal`

Associe la commande *comm* au *signal*. Par conséquent, lorsque le signal est reçu, la commande *comm* est exécutée. Lorsqu'elle n'est pas précisée, la commande par défaut est de nouveau associée au signal.

L'envoi d'un signal à un processus s'effectue par `kill` :

`kill -signum pid`

envoie le signal numéro *signum* au processus *pid*, qui réagit en conséquence.

**Exemple 27** *Dans l'exemple suivant, la commande associée au signal* 2 *devient* `echo coucou` *(ligne 1). Le numéro de processus associé à l'interpréteur de commande est obtenu ligne 4 :* 312*. L'envoi du signal* 2 *au processus par la commande kill, effectué en ligne 6, conduit à l'affichage de* `coucou`*, de même que l'interruption par* `<CTRL-c>` *comme en ligne 8. On remarque que le caractère tapé,* `<CTRL-c>` *ne s'affiche pas. Enfin, en ligne 9, la commande par défaut est réinstallée pour le signal* 2*, comme le montre l'exécution de* `kill` *en ligne 10.*

```
1  $ trap 'echo coucou' 2
2  $ ps | grep bash
3    296 ttys000    0:00.02 -bash
4    312 ttys000    0:00.02 bash
5    324 ttys000    0:00.00 grep bash
6  $ kill -2 312
7  coucou
8  $ coucou
9  $ trap 2
10 $ kill -2 312
11 $
```

Le signal `SIGHUP` est émis lorsque le terminal se déconnecte. Il se peut toutefois que l'on souhaite que le processus continue son exécution même en cas de déconnexion. Pour ce faire, `nohup` permet de lancer une commande *comm* en inhibant le signal 1 :

`nohup` *comm*

Le terminal s'étant *a priori* déconnecté, la sortie ne peut pas être affichée à l'écran. Elle est produite dans un fichier `nohup.out` du répertoire de travail (ou du répertoire privé de l'utilisateur si ce dernier ne peut pas écrire dans le répertoire de travail).

### 4.7.3 Gestion des priorités et du temps

#### Gestion des priorités

Les processus sont démarrés avec une *priorité* entre 0 et 20. Plus ce nombre est faible, plus le processus est prioritaire. Le système d'exploitation gère les priorités en allouant le processeur d'abord aux processus les plus prioritaires.

`nice -n` *priorité comm*

lance la commande *comm* avec une *priorité* spécifiée. Seul le super-utilisateur peut augmenter la priorité d'un processus. Ce mécanisme a donc surtout pour but qu'un utilisateur déclare qu'un de ses processus est peu prioritaire par rapport aux autres.

La priorité d'un processus en cours d'exécution peut également être modifiée avec `renice`.

#### Gestion du temps

Deux commandes permettent de planifier des exécutions de processus dans le temps :

- `at` *date* attend une commande sur l'entrée standard et planifie son exécution à la *date* indiquée ;
- `crontab -e` édite un fichier de description de tâches répétitives qui seront exécutées automatiquement, comme précisé dans le fichier. Un démon, nommé `cron`, gère l'exécution de ces tâches.

Les utilisateurs ne sont pas forcement autorisés à utiliser ces deux commandes. Les fichiers `/etc/at.allow` (respectivement `/etc/at.deny`) et `/etc/cron.allow` (respectivement `/etc/cron.deny`) contiennent la liste des utilisateurs autorisés (respectivement non autorisés) à utiliser les commandes `at` et `cron`.

Le fichier décrivant les commandes à exécuter répétitivement, édité par `crontab` contient 6 champs par ligne, séparés par des espaces ou des tabulations :

1. les *minutes* auxquelles l'exécution a lieu, de 0 à 59. S'il y en a plusieurs par heure, elles sont séparées par une `,` ;
2. les *heures* auxquelles l'exécution a lieu, de 0 à 23. S'il y en a plusieurs par jour, elles sont séparées par une `,` ou un `-` pour indiquer une plage d'heures. Le caractère `*` indique l'ensemble des valeurs possibles ;
3. les *jours du mois* auxquels l'exécution a lieu, de 1 à 31. S'il y en a plusieurs par mois, ils sont séparés par une `,` ou un `-` pour indiquer une plage de jours. Le caractère `*` indique l'ensemble des valeurs possibles ;
4. les *mois* auxquels l'exécution a lieu, de 1 à 12. S'il y en a plusieurs par an, ils sont séparés par une `,` ou un `-` pour indiquer une plage de mois. Le caractère `*` indique l'ensemble des valeurs possibles ;
5. les *jours de la semaine* auxquels l'exécution a lieu, de 0 pour dimanche à 6 pour samedi. S'il y en a plusieurs par semaine, ils sont séparés par une `,` ou un `-` pour indiquer une plage de jours. Le caractère `*` indique l'ensemble des valeurs possibles ;
6. la *commande* à exécuter.

**Exemple 28** *La ligne :*

$\underbrace{\texttt{0,30}}_{(1)}\ \underbrace{\texttt{8-20}}_{(2)}\ \underbrace{\texttt{*}}_{(3)}\ \underbrace{\texttt{*}}_{(4)}\ \underbrace{\texttt{1,3,5}}_{(5)}\ \underbrace{\texttt{/usr/local/bin/sauvegarde}}_{(6)}$

*spécifie une exécution :*

1. *à l'heure exacte et à la demie ;*
2. *toutes les heures de* 8 *heures à* 20 *heures ;*
3. *tous les quantièmes*[2] *;*
4. *tous les mois ;*
5. *seulement les lundi, mercredi et vendredi ;*
6. *de la commande* `/usr/local/bin/sauvegarde` *.*

2. numéro du jour dans le mois.

# Chapitre 5

# Initiation à la virtualisation

## 5.1 Système d'exploitation

### 5.1.1 Rôle du système d'exploitation

L'architecture matérielle d'un ordinateur est complexe : mémoire, disques, cartes réseau, périphériques. Les applications ne peuvent pas être programmées pour accéder directement à ces éléments matériels, d'une part parce que la programmation serait inutilement complexe, et d'autre part pour garantir un minimum de sécurité d'accès.

Le système d'exploitation est en charge de la gestion des accès au matériel. De plus, sur un système multi-utilisateurs, où de nombreux processus s'exécutent, le système ordonnance les processus de manière à ce que leur exécution semble simultanée : un processus est exécuté pendant un certain temps, puis mis en attente, un autre est alors exécuté, et ainsi de suite.

Le système d'exploitation fait donc l'interface entre le matériel et les applications.

### 5.1.2 Noyau

Le programme central du système d'exploitation s'appelle *le noyau*. Il gère l'accès des processus à la mémoire et les ordonnance sur les c$\frac{1}{2}$urs. Les programmes informatiques s'appuient sur des bibliothèques de fonctions leur permettant d'utiliser les services du système.

**Exemple 29** *Dans le langage de programmation* `C`*, la fonction* `malloc(t)` *réalise l'allocation d'une zone mémoire de taille* `t`*. A l'exécution, le noyau réserve pour le processus exécutant le programme, une zone mémoire de taille* `t` *et renvoie l'adresse de la zone. Ainsi le système gère la répartition entre les différents processus, évitant que ceux-ci utilisent les mêmes zones (et par exemple écrasent les données d'un autre processus). Lorsque le programme fait appel à la fonction* `free(adr)`*, la zone à l'adresse* `adr` *est libérée par le noyau. Elle peut alors être allouée à un autre processus (ou au même).*

Le noyau s'exécute dans un « espace noyau » où il dispose de tous les privilèges d'accès , alors que les processus utilisateurs ont des privilèges restreints.

Le noyau est le premier programme exécuté par la machine. Un seul noyau s'exécute dans l'espace noyau.

## 5.2 Virtualisation

### 5.2.1 Objectifs

De nos jours, beaucoup de serveurs sont virtualisés. Un serveur n'exploite pas la totalité des nombreuses ressources de la machine physique sur laquelle il s'exécute. La virtualisation permet de

rationaliser l'utilisation de ces ressources en exécutant plusieurs serveurs sur une même machine physique.

Toutefois, les différents services fournis pourraient s'éxécuter sur un seul système d'exploitation. Un avantage de la virtualisation est d'en améliorer la sécurité, la stabilité, et la facilité d'administration. Ainsi, chaque service s'exécute sur sa propre machine virtuelle et n'interfère pas avec les autres. Ces serveurs sont donc cloisonnés et isolés des autres.

La virtualisation permet donc d'exécuter plusieurs systèmes d'exploitation sur une même machine physique.

On distingue deux catégories de systèmes d'exploitation s'exécutant sur une machine physique :

— le *système hôte* qui orchestre l'utilisation des ressources matérielles fournies par la machine. Une seule instance de ce système est présente ;

— les *systèmes invités* qui s'exécutent en espace utilisateur. Il peut y en avoir plusieurs instances. Les systèmes invités demandent au système hôte d'accéder aux ressources matérielles.

### 5.2.2 Hyperviseurs

Un *hyperviseur* est un noyau hôte allégé dont le seul rôle est d'exécuter les systèmes invités. Il existe deux types d'hyperviseurs :

— les *hyperviseurs de type I* se situent entre le matériel de la machine et les systèmes invités ;

— les *hyperviseurs de type II* sont hébergés entre le système d'exploitation hôte et les systèmes invités.

Par exemple, `kvm` est un hyperviseur de type I, tandis que `Virtualbox` est un hyperviseur de type II.