



# Introduction à la ligne de commande

Le 22 mars 2008, à 12:32 par Uihume...

S'il y a bien un truc qui fait peur aux nouveaux linuxiens, c'est la ligne de commande. Certains la prétendent archaïque mais il s'agit plus d'une crainte liée à une manière "nouvelle" de travailler. En effet, la ligne de commande, comme toutes les interfaces, a ses forces et ses faiblesses. Et chercher à organiser ses fichiers avec est aussi idiot que de systématiquement utiliser une interface graphique pour les décompresser. Ce billet n'est donc pas une ode à la ligne de commande, mais juste une modeste introduction permettant, je l'espère, de prendre conscience de sa puissance pour un certain nombre de tâches spécifiques.

## Les vues

Petite digression pour commencer, le système de "vues" sous Linux. En standard, Linux dispose de 6 vues textes et 6 vues graphiques. Chaque vue est accessible par la combinaison de touche `CTRL-ALT-F1` à `CTRL-ALT-F6` pour les vues textes et `CTRL-ALT-F7` à `CTRL-ALT-F12` pour les vues graphiques. Avant de jouer avec ces touches, gardez en tête que vous êtes en ce moment même sur la vue graphique n°1 et que vous pouvez donc revenir ici par `CTRL-ALT-F7`. Maintenant allez-y, testez 😊

Vous constatez tout d'abord que les vues 8 à 12 sont vides. La raison en est que le serveur graphique (X) ne les utilise pas toute, tout de suite. Chaque vue correspond à un utilisateur connecté. Pour le tester cela sous Gnome, allez dans `Système/Fermer Session.../Changer d'utilisateur`. N'ayez pas peur, vous n'allez rien "fermer" en réalité et vous arrivez juste sur une fenêtre de connexion. Si vous entrez à nouveau votre login/mot de passe, une nouvelle session est ouverte. Et cette fois, elle est sur la deuxième vue graphique. Faites le test vous-même et passez de l'une à l'autre par `CTRL-ALT-F7` et `CTRL-ALT-F8`. Sous Unix en général, et sous Linux en particulier, vous pouvez ainsi avoir autant de sessions graphiques que vous le désirez, une pour chaque utilisateur de votre choix. Pas besoin de version "pro" comme sous Windows 😊 Et surtout cela prend très peu de ressources.

Maintenant intéressons nous à notre sujet de départ et allons sur `CTRL-ALT-F1`. Là il s'agit de la première vue texte qui donne lieu à la saisie d'un login/mot de passe. Comme pour la vue graphique, vous pouvez passer d'une session texte à l'autre et vous connecter ainsi plusieurs fois.

Et bien ces sessions texte, ce sont des `consoles` permettant de saisir, une fois connecté, des `commandes`. Elles sont très pratiques car toujours accessibles, même quand la partie graphique n'est pas lancée ou est surchargée. Elles permettent dans le second cas de tuer un processus bloqueur.

## Les consoles

Ceci dit, il n'est pas très pratique de basculer en mode texte à chaque commande à saisir. On réserve cela à des situations de crise ou au démarrage de la machine. Lorsque l'on est au chaud dans sa session graphique, on utilise plutôt un émulateur de console. Si vous allez dans le menu de vos applications vous en avez forcément une dans vos `outils`. Il s'agit de `konsole` sous KDE, `gnome-terminal` sous Gnome, mais il y en a des tripotées de disponibles à installer (`mrxvt`, `xterm`, `xfce-term`, etc...).

Mais au fond toutes permettent la même chose, taper des commandes. Au lancement, elle affichent toute un écran qui simule un affichage texte et affichent une invite. L'invite est un ligne de texte indiquant généralement le dossier dans le quel on se trouve, votre nom d'utilisateur et se termine par le symbole `$` indiquant que l'on a pas les droits `root` (sinon, ce serait un symbole dièse)

Ce dernier point (root ou pas root) pose souvent des problèmes aux évadés de Redmont. Donc nouvel le petite digression.

## Les droits

La véritable raison qui fait que les virus prolifèrent sous Windows tient à la manière de gérer les droits. Un système sain va distinguer deux ensembles : les utilisateurs et l'administrateur de la machine. Les utilisateurs n'ont pas de droits hors de leur dossiers, ne peuvent installer d'application, ne peuvent installer de pilotes, ne peuvent pas ouvrir de serveur sur un port TCP/IP protégé, etc. L'administrateur, lui, a le droit de faire tout cela. Ainsi, dans un système sain, un utilisateur ne peut pas rendre le système instable en faisant une grosse bêtise (genre virer un fichier vitale). Et il en va de même pour un virus attrapé par un utilisateur, car ce dernier aura les mêmes droits réduits.

Maintenant cette vision utilisateur/administrateur est tout a fait compréhensible en entreprise, mais l'est beaucoup moins pour un ordinateur personnel où l'utilisateur est généralement son propre administrateur. Et c'est là que la logique diffère entre Windows et Unix. Car à ce stade Windows est aussi sain que Linux. Sous Windows vous avez la même séparation des droits. La seule différence est que sous Unix existe de nombreux mécanisme permettant à un utilisateur d'acquérir simplement les droits administrateur (root) pendant un temps donné. Sous Windows, c'est beaucoup plus compliqué. En conséquence, nombre d'utilisateur Windows, très logiquement agacés lorsqu'ils sont bloqués par la moindre installation de logiciel, vont dans leur profile et se déclare "administrateur" de la machine. Et là les ennuis commencent, ils peuvent déstabiliser le système et surtout les virus ont, eu aussi, les droits administrateur et peuvent faire de vrais dégâts.

Et Vista n'a rien arrangé au problème. Ils ont augmenté la séparation utilisateur/administrateur rendant le système tellement contraignant que je n'ai jamais vu un seul utilisateur de Vista qui n'a pas désactivé purement et simplement la sécurité....

Sous Linux, il existe de nombreux moyens de prendre pour un temps les droits administrateur. Sous Gnome par exemple, dès qu'une application graphique a besoin de ses droits spéciaux, elle va systématiquement demander le mot de passe de l'administrateur. Il en va de même sous KDE. Et en ligne de commande, nous allons le voir, c'est très facile aussi. En conclusion, la règle absolu est de ne **jamais se connecter en session graphique en tant que root**, et de **toujours laisser les mots de passe activés**.

Lorsque vous êtes en console, le moyen préconisé pour exécuter une commande qui demande les droits root, **n'est pas de se connecter en root via un su -**. Le meilleur moyen est d'utiliser la commande `sudo`. Cette dernière va vous donner les droits root pendant un court temps, ou pour juste une commande en ne vous demandant que votre mot de passe utilisateur. Cette commande est configurée par défaut dans la majorité des distribution, sinon, cherchez de l'aide sur `visudo`. Par exemple pour installer une application en ligne de commande :

exemple d'utilisation de sudo pour installer

```
gaston$ sudo urpmi openoffice
```

```
mot de passe: votre mot de passe utilisateur
gaston$
```

## L'interpréteur de ligne de commande

Tout à l'heure je vous disais qu'une console affichait une invite. En réalité ce n'est pas tout à fait vrai. Une console ne fait que simuler un écran texte et lance, par défaut un utilitaire appelé « interpréteur de ligne de commande ». Souvent cet interpréteur est `bash`, mais il y en a d'autres chacun avec leur syntaxe. Pour ce qui suit, nous allons parler de `bash`.

Le rôle de l'interpréteur de commande est, comme son nom l'indique, de lire ce que vous tapez et lorsque vous validez, d'interpréter le contenu et de lancer les commandes correspondant à la syntaxe que vous avez saisie.

Ainsi toute la logique que nous allons maintenant voir est en réalité gérée par `bash` qui est donc une sorte de langage. Dans une première approche cela n'a pas beaucoup d'importance, mais cela permet de comprendre qui fait quoi.

## Trouver son chemin

Revenons donc à nos moutons. Lorsque l'on ouvre une console, vous êtes dans un dossier indiqué dans l'invite. Normalement ce dossier est votre dossier utilisateur qui se trouve dans le dossier `home` suivi de votre nom d'utilisateur. Vous pouvez le vérifier grâce à la commande `pwd` qui va renvoyer le dossier en court :

```
gaston$ pwd
/home/gaston
gaston$
```

Pour changer de dossier courant, vous utilisez la commande `cd` suivi du chemin visé.

```
gaston$ cd ..
gaston$ pwd
/home
gaston$
```

Enfin, la commande `cd` lancée tout seule (sans paramètres) vous transporte dans votre dossier personnel.

## Entrées et sorties standard

Commençons par la classique commande `ls` (l'équivalent de `dir` sous DOS/Windows) :

```
gaston$ ls
Bureau/ tmp/
gaston$
```

La commande `ls`, qui donne la liste des fichiers dans le dossier où l'on se trouve, nous a renvoyé cette liste à l'écran. En réalité la commande a envoyé la liste dans la `sortie standard` qui par défaut est affichée à l'écran. Pour s'en convaincre, utilisons une nouvelle syntaxe, la redirection, que nous

expliquerons mieux plus loin :

```
gaston$ ls > liste
gaston$ cat liste
Bureau/
tmp/
gaston$
```

La syntaxe `>` a **redirigé** la sortie standard non plus vers l'écran mais vers un fichier. La commande `cat` va lire le fichier et l'envoyer sur sa sortie standard. Et vu que là, il n'y a pas de `>`, cela sort directe sur l'écran. Voyons maintenant la même chose, mais pour un message d'erreur :

```
gaston$ ls truc > liste
ls: ne peut accéder truc: Aucun fichier ou répertoire de ce type
gaston$
```

Surprise, notre message d'erreur, lui, ne finit pas dans le fichier. La raison en est que toutes les commandes ont en réalité deux canaux de sortie : la sortie standard, et la sortie des erreurs. Si je veux rediriger les erreurs dans un fichier, je vais devoir modifier légèrement ma syntaxe :

```
gaston$ ls truc 2> erreur
gaston$ cat erreur
ls: ne peut accéder truc: Aucun fichier ou répertoire de ce type
gaston$
```

Cette fois c'est la sortie des erreurs que j'ai redirigé vers un fichier, et du coup, ma commande `ls`, redeviens muette et l'erreur est bien contenue dans le fichier `erreur`.

On appelle la sortie standard `stdout`, et la sortie des erreurs `stderr`.

Maintenant que nous savons comment gérer ce qui « sort » d'une commande, voyons ce que l'on peut y faire « entrer » :

```
gaston$ cat > un_message
blablabla
CTRL-D
gaston$ cat un_message
blablabla
gaston$
```

Comme nous l'avons vu dans les exemples précédents, la commande `cat` prend en entrée un fichier. Mais si l'on omet ce fichier, c'est l'entrée standard `stdin` qui est utilisée par la commande. Et comme, par défaut, l'entrée standard d'une commande, c'est le clavier, la première ligne correspond à copier chaque caractère que vous tapez et l'envoyer sur la sortie standard. Sortie standard qui est redirigée sur notre fichier `mon_message`.

Une commande unix est donc une application, souvent placée dans le dossier `/usr/bin`, qui lorsqu'elle est exécutée, prend des données dans son entrée standard (`stdin`), ou de ses paramètres, et écrit ses résultats dans la sortie standard (`stdout`) et ses erreurs dans la sortie des erreurs (`stderr`). Par défaut, si

rien n'est précisé, ce qui est tapé au clavier est envoyé sur l'entrée standard, et tout ce que

l'application écrit, que ce soit des erreurs ou des résultats, est envoyé sur l'écran.

Maintenant nous allons voir comment jouer aux legos avec nos commandes, grâce aux redirections et aux tubes.

## Jouons aux legos?

Nous avons déjà vu la syntaxe `>`. Elle permet de rediriger une des deux sorties vers un fichier. Voyons tout ce que l'on peut faire avec cela :

```
# envoyer la sortie standard (stdin) vers un fichier. Les deux syntaxes sont équivalentes.
commande > fichier
commande 1> fichier

# envoyer la sortie des erreurs (stderr) vers un fichier
commande 2> fichier

# Plus fort, envoyer toutes les erreurs vers la sortie standard, et la sortie standard vers un fichier
commande 2>&1 > fichier

# une version simplifiée de la commande précédente, stderr et stdout vont tout deux dans le fichier
commande &> fichier
```

Maintenant, la question que les amateurs de legos se posent déjà, c'est « comment envoyer le résultat d'une commande comme entrée d'une autre commande ». En effet, la notation `>` est réservée aux fichiers, on ne peut donc pas envisager des choses comme `commande1 > commande2`. La solution est d'utiliser un tube (pipe).

Voyons comment utiliser l'indispensable commande `grep` qui permet de chercher dans ce que vous lui donnez en entrée (stdin) une chaîne de caractère. Par exemple imaginons que nous voulions chercher le mot `Bureau` dans notre liste de fichiers renvoyée par `ls` :

```
gaston$ ls | grep -i bureau
Bureau
gaston$
```

Voilà donc comment jouer aux legos. La traduction du `|` est **redirige la sortie de la commande `ls` vers l'entrée de la commande `grep`**. Le `-i` indique seulement à Grep de ne pas faire de différence majuscules/minuscules.

Maintenant nous pouvons combiner tout ce que nous avons vu en une ligne un peu barbare :

```
gaston$ ls toto 2>&1 | grep -i fichier > resultat
gaston$
```

Mon exemple est idiot en soit mais c'est juste un exemple. Il s'agit là de dire : redirige les erreurs de la commande `ls` sur la sortie standard, cherche le mot "fichier" dedans et stockes le résultat dans le fichier

"resultat".

Nous verrons des exemples un peu moins idiots plus loin 😊

## Substitutions

Alors nous avons vu comment utiliser la sortie d'une commande comme entrée d'une seconde commande. Maintenant comment utilisez cette sortie comme paramètre de la seconde commande. Simplement avec une substitution. Imaginons que nous voulions convertir tous les fichiers jpeg du dossier courant en un seul fichier pdf. Une liste des fichiers jpeg peut être obtenu par la commande `ls | grep jpg`. Maintenant nous allons utiliser la notation `$( )` pour l'injecter dans la commande `convert` :

```
Convert $(ls | grep jpeg) resultat.pdf
```

Le membre `$(ls | grep jpeg)` va être substitué lorsque vous validez la ligne par le résultat de la commande entre parenthèses. `Convert` va donc avoir en paramètre une liste de fichier jpeg en plus du paramètre final `resultat.pdf`.

Bon, ceci est un exemple, et comme beaucoup d'exemples il est idiot car j'aurais pu obtenir le même résultat sans substitutions par un simple `convert *.jpeg resultat.pdf` ou encore en utilisant plutôt `$(ls *.jpeg)`. Mais il reste néanmoins intéressant car le caractère `*` est en réalité lui aussi une substitution. En effet, contrairement à ce que l'on peut imaginer, ce n'est pas la commande qui va recevoir le `*.jpeg` et en déduire une liste de fichier, mais `bash` qui va lui-même chercher cette liste et faire une substitution. Une conséquence directe de cela est que si votre `*.jpeg` renvoie trop de fichiers, le système vous renverra une erreur pour cause de trop nombreux paramètres passé à la commande.



Une autre syntaxe pour la substitution consiste à utiliser ``commande``. C'est exactement la même chose que `$(commande)` sauf que son utilisation est découragée pour cause d'obsolescence. Plus d'informations [ici](#) <sup>[1]</sup>

## Obtenir de l'aide

Unix c'est un énorme paquet de commandes et chaque commande possédant ces paramètres propres, il est illusoire d'imaginer les connaître tous. La solution est l'incontournable commande `man` qui s'utilise suivi de la commande dont on désire connaître le fonctionnement, par exemple pour `rsync` :

```
man rsync
```

Une fois l'aide affichée, il est possible d'en sortir en tapant `q`, se déplacer avec les flèches mais aussi de rechercher des choses dedans en utilisant un mythique syntaxe `/chose_a_chercher` et en validant. La chaîne cherchée devrait s'afficher en surligné.

Enfin, pour les kdeistes, une version graphique du `man` est utilisable dans `konqueror` en tapant l'URL spéciale `man:/commande`.

C'est tout pour la théorie, maintenant voyons des exemples issus de la « vraie vie ».

# Exemples concrets

## La substitution qui tue

Pour rester dans le domaine de la substitution, voyons un exemple d'une efficacité redoutable mais à manier avec **beaucoup de précautions**. Imaginons que nous voulions désinstaller TOUT kde d'un coup. Avec une interface graphique cela prendrait un temps fou, alors qu'avec une simple ligne de commande :

```
rpm -e $(rpm -qa | grep kde)
```

Attention, cette ligne est violente. La commande entre parenthèse nous permet d'obtenir la liste exhaustive de tous les paquets installés qui contiennent la chaîne `kde`, et on substitue simplement cette liste comme paramètres dans la commande `rpm -e $(rpm -qa | grep kde)` qui vas **sans vérification de dépendance** les désinstaller.

## Télécharger et décompresser en même temps

Dans cet exemple, imaginons que nous voulions décompresser un fichier **directement à partir d'internet**

```
wget -O - http://ftp.drupal.org/files/projects/zental-5.x-1.x-dev.tar.gz |  
tar -zxv -f -
```

Magique non ? Ici nous avons indiqué à la commande `wget` de télécharger en envoyant ce qu'elle télécharge directement sur la sortie standard par `-O -`. Ensuite grâce au tube, nous avons envoyé le résultat sur la commande `tar` à qui nous indiquons de ne pas décompresser un fichier mais directement son entrée standard par le paramètre `-f -`.

Ce n'est pas une règle générale mais beaucoup de commande unix fonctionnent ainsi. C'est une sorte de convention qui dit que le caractère `-` à la place d'un nom de fichier veut dire entrée ou sortie standard selon le contexte.

De la même manière il est possible de télécharger et graver sur un CD en même temps, mais cette fois en utilisant la commande `cdrecord` au lieu de `tar`.

## Synchroniser une base de donnée distante

Il est possible de faire la même chose que plus haut, mais à distance grâce à l'indispensable commande `ssh`. Par exemple, imaginons que nous avons deux bases de données postgresql. Une en locale, l'autre sur un serveur distant.

```
# on commence par supprimer la base de donnée locale  
dropdb ma_base -U postgres  
# on en recré une nouvelle  
createdb --encoding=UNICODE -U postgres ma_base  
  
# Et là on va synchroniser le tout en une seule ligne  
ssh mon_serveur "pg_dump -U postgres ma_base -h localhost" | psql -U  
postgres ma_base
```

lance la commande qui est entre les deux `"`. Cette commande demande à postgres de créer un fichier SQL de la base distante et de l'envoyer sur la sortie standard. Lorsque la commande distante s'exécute, ssh transmet en local son résultat sur sa sortie standard. Sortie standard que je redirige par un tube vers la commande `psql` qui va recréer ma base en locale, au fur et à mesure de l'arrivée des données.

Le seul problème c'est qu'une base, c'est gros, ce qui serait sympa c'est de compresser tout cela en temps réel... Pas de problème 😊

```
ssh mon_serveur "pg_dump -U postgres ma_base -h localhost | gzip" | gunzip | psql -U postgres ma_base
```

Et voilà, j'ai rajouté un tube dans la commande distante qui va compresser le script SQL avant de l'envoyer dans la sortie standard. Et à l'arrivée, j'ai rajouté un `gunzip` pour décompresser avant exécution de ses commandes SQL. Et le tour est joué.

## Connaître les fichiers les plus gros

Là nous allons utiliser trois nouvelles commandes : `du`, `sort` et `more`.

```
du . -xb --block-size=1M | sort -rn | more
```

La commande `du` va me sortir de manière récursive la liste de tous les fichiers avec leur taille. La commande `sort` va opérer un tri de cette liste pour mettre les plus gros fichiers en tête et la commande `more`, indispensable, va juste bloquer l'affichage page par page.

La commande `more` fait partie d'un jeu de commandes ultra-utilisées avec `head` et `tail`. `head` permet de ne prendre que les premières lignes de l'entrée standard, et `tail` fait l'inverse, et ne prend que les dernières lignes. Ainsi, pour avoir les 10 premiers fichiers les plus gros :

```
du . -xb --block-size=1M | sort -rn | head
```

Et pour les dix fichiers les plus petits :

```
du . -xb --block-size=1M | sort -rn | tail
```

## Conclusion

La ligne de commande permet avec un investissement relativement faible, d'obtenir des résultats qu'il est juste impossible d'avoir avec une interface graphique. Interface graphique et ligne de commande sont donc complémentaires, la première cherchant à rendre simple les opérations les plus courantes, la seconde à permettre tout le reste. J'espère en tout cas que cette petite introduction vous donnera envie d'aller plus loin.

### Liens:

[1] <http://bash-hackers.org/wiki/doku.php/scripting/obsolete>