In [1]:

```
from common import utils
import random
u=utils("PDF") # PDF+SOL / PROF / PDF / INTER
```

```
Ce cours a été régénéré le 2018-11-06 19:59:37.432313. Mode sans corrigé. Mode statique.
```

Les flux

Les programmes informatiques (applications) communiquent avec des moyens divers, et des interlocuteurs divers.

Ils communiquent avec :

- des utilisateurs locaux (applications)
- des utilisateurs distants (client et serveur)
- des bases de données
- des écrans (et autres dispositifs d'entrée-sortie)
- d'autres ordinateurs

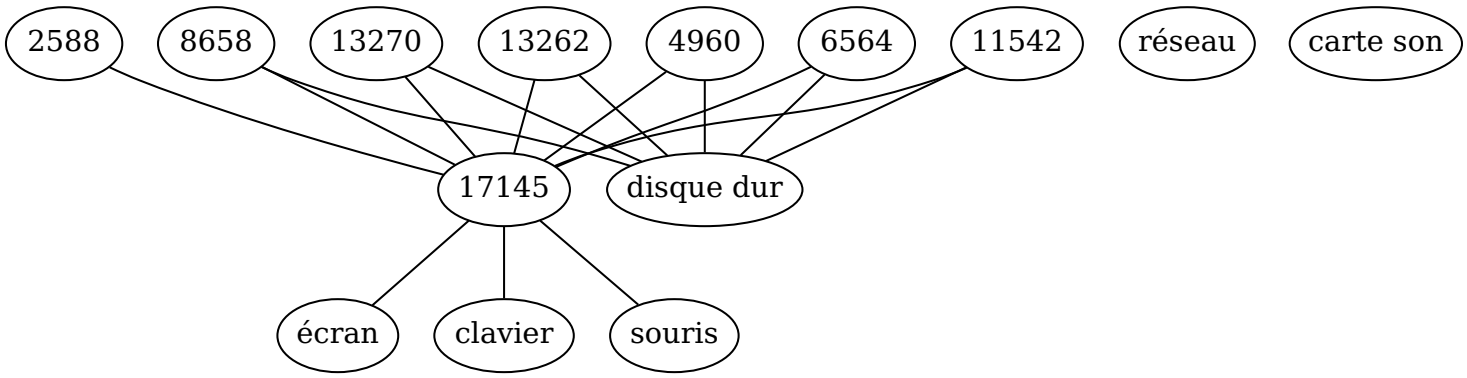
La réalité est que les seuls programmes qui ont un effet sont ceux qui permettent des entrées-sorties. Tous les autres programmes ne font que communiquer avec d'autres programmes.

On distingue plusieurs types d'entrée ou sortie du système :

- Stockage de masse
- Réseau
- Dispositifs physiques d'entrée-sortie (du clavier au bras mécanique en passant par les écrans et les moteurs de fusée).

Le shell permet de contrôler certaines communications entre les processus et autres intervenants.

In [2]:



Les communications qui nous intéressent sont les communications qui ont lieu par un certain type d'interface, les interfaces de type *fichier*. S'il paraît évident que les interfaces de type fichier sont utilisées par le disque dur, elles le sont aussi pour un grand nombre d'autres périphériques d'entrée-sortie dans la famille des systèmes d'exploitation Unix/Linux.

Elles le sont en particulier pour un certain type de communication entre processus, que nous allons apprendre à faire ici.

Un processus, des flux

Chaque processus fait appel au système d'exploitation pour communiquer avec le reste du système. Dans le cadre de la communication par un flux de type *fichier*, cette communication est codée par un certain nombre d'informations qui sont toutes regroupées dans un **descripteur de fichier** (en anglais, *file descriptor* ou *fd*). Chaque processus peut avoir au plus un certain nombre de descripteurs de fichiers ouverts en même temps qui est fixé par le système.

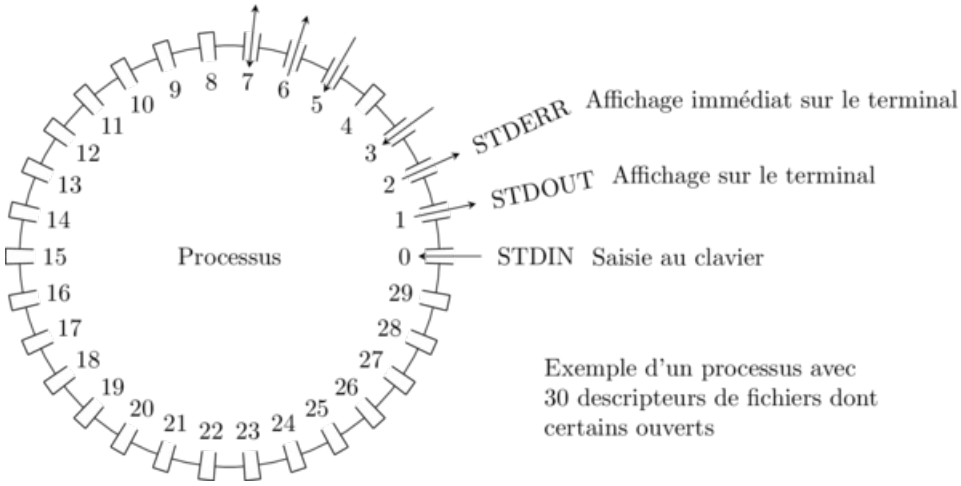
Chaque processus peut ainsi être vu comme une petite cellule, qui pour communiquer dispose de fenêtres qu'elle peut connecter à d'autres dispositifs (en général, un *fichier* sur un disque dur, mais nous verrons plus loin que ce n'est pas du tout le seul cas). Ces fenêtres, à leur ouverture, sont mises en mode *lecture*, *écriture*, ou encore bidirectionnel (*lecture et écriture*).

In [3]:

```
%%sh
#! /bin/sh
# L'explication de cette syntaxe arrive plus loin !
# ulimit -n permet de lister le nombre de descripteurs de fichiers simultanés autorisés
echo "Il est possible d'ouvrir au plus simultanément $(ulimit -n) fichiers sur ce système."
```

```
Il est possible d'ouvrir au plus simultanément 1024 fichiers sur ce système.
```

In [4]:



Exemple d'un processus avec 30 descripteurs de fichiers dont certains ouverts

Ces fenêtres peuvent être des communications avec :

- des périphériques (directement)
- un système de fichier (et donc indirectement un périphérique de stockage)
- un système de fichier *virtuel* (et donc le noyau)
- soit avec un autre processus

Comme tout appel système, cette communication se fait par du code dans le noyau du système d'exploitation. Les principaux appels systèmes qui permettent de communiquer sont

- `open` qui permet d'ouvrir une communication (ou la variante `openat`)
- `close` qui permet de fermer une communication
- `read` qui permet de lire des données
- `write` qui permet d'écrire des données.

On peut observer les appels systèmes d'un

In [5]:

```
%%sh
# On distingue
if [ -x /usr/bin/strace ]; then
    strace -e trace=open,openat,close,write,read cat /etc/debian_version
else
    echo "Désolé, strace n'est pas disponible sur ce système"
fi
```

```
buster/sid

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000,\2\0\0\0\0\0"..., 832) = 832
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/etc/debian_version", O_RDONLY) = 3
read(3, "buster/sid\n", 131072)         = 11
write(1, "buster/sid\n", 11)            = 11
read(3, "", 131072)                     = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
+++ exited with 0 +++
```

Les communications et le shell

La situation normale

À sa création, un processus a habituelle trois descripteurs de fichiers ouverts. Ils portent des noms :

- STDIN (0) est *l'entrée standard* du processus. Anciennement rattaché directement à un clavier, elle est en fait rattaché souvent à un pseudo-périphérique géré par le terminal (une autre application) qui lui-même est rattaché (parfois par le biais de plusieurs autres couches) au clavier.
- STDOUT (1) est *la sortie standard* du processus. Anciennement rattaché directement à l'écran, elle est en fait rattachée au terminal (comme pour STDIN).
- STDERR (2) est *la sortie d'erreur* du processus. Comme pour STDOUT et STDIN, elle est rattachée au terminal. La différence est qu'il y a affichage immédiat.

C'est à la création d'un processus que tout ceci est mis en place. Or, le shell est en gros une application qui ne sert qu'à une chose : créer des processus ! Il est donc tout à fait possible pour le shell de modifier cette situation pour faire des choses plus intéressantes...

Il faut comprendre qu'on ne peut pas, à partir d'un descripteur de fichier, savoir où il va (on peut, par contre, l'interroger sur ses *capacités*, par exemple la possibilité de rembobiner ou pas).

Avec le système de fichier

Le shell est capable, avec des spécifications dans la ligne de commande (ou dans un script), de transformer les entrées/sorties normales d'un processus, en particulier, de transformer une entrée-sortie vers un périphérique en une entrée-sortie vers un fichier.

La syntaxe est la suivante:

```
commande 1>/mon/fichier/de/sortie 0</mon/fichier/d/entree
```

La syntaxe `n>chemin` *redirige* un descripteur de fichier (numéro n) vers un fichier qui est désigné par le chemin (relatif ou absolu), c'est-à-dire que le descripteur de fichier est modifié pour maintenant aller vers un fichier. Le processus ne peut pas faire de différence. Le nouveau descripteur de fichier est en mode *écriture*.

La syntaxe `n>>chemin` *redirige* un descripteur de fichier (numéro n) de la même façon, sauf que le nouveau descripteur de fichier est en mode *écriture* mais est positionné à la fin du fichier déjà existent (on parle du mode *ajout* ou *append*).

La syntaxe `n<chemin` *redirige* un descripteur de fichier (numéro n) depuis un fichier qui est désigné par le chemin (relatif ou absolu), c'est-à-dire que le descripteur de fichier est modifié pour maintenant venir d'un fichier ; c'est comme si ce qui était dans le fichier était tapé au clavier (d'un seul coup). Le processus ne peut pas faire de différence. Le nouveau descripteur de fichier est en mode *lecture*.

Si on utilise juste `>` il est sous-entendu que c'est STDOUT qui est redirigée. Si on utilise juste `<` il est sous -entendu que c'est STDIN qui est capté.

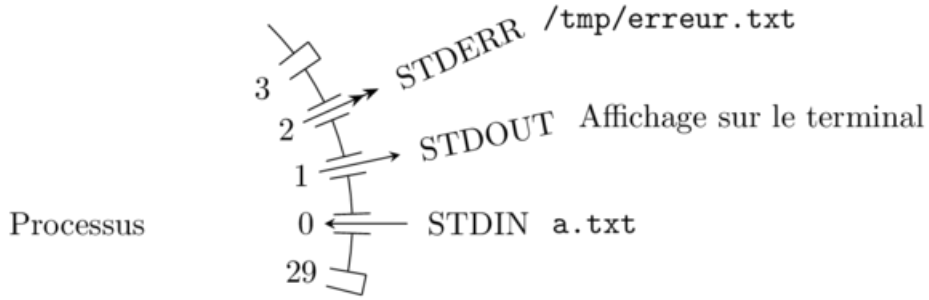
Exemple :

```
cat /etc/debian_version 1>/tmp/fichier.txt
head /proc/cpuinfo 1>/tmp/processeur.txt
n=0
while read line; do
    n=$((n+1))
done 0</usr/share/dict/words
echo $n
```

Il devient plus facile de faire la différence entre la sortie d'erreur et la sortie standard avec ce système. La plupart des commandes envoient tout leur texte sur le descripteur numéro 1, sauf les messages d'erreur.

NB: la sortie d'erreur est déjà affichée différemment dans les notebooks.

In [6]:



Exemple d'un processus avec
2>>/tmp/erreur.txt <a.txt

In [7]:

```
%%sh
cat /etc/debian_version 1>/tmp/fichier.txt
head /proc/cpuinfo 1>/tmp/processeur.txt
n=0
FILE=/usr/share/dict/words
while read line; do
    n=$((n+1))
done 0<"${FILE}"
echo "Le fichier $FILE comporte $n mots"
```

```
Le fichier /usr/share/dict/words comporte 102401 mots
```

In [8]:

```
%%sh

# Essayez aussi dans un terminal.

echo "Ceci est un texte à supprimer" 1>/tmp/aeffacer.txt
cat /tmp/aeffacer.txt /tmp/cefichiernexistepas

# Observez bien la différence entre > et >> : le premier recommence un fichier au début, le deuxième complète
cat /tmp/aeffacer.txt /tmp/cefichiernexistepas 2>/tmp/erreur.txt
rm /tmp/aeffacer.txt /tmp/cefichiernexistepas 2>>/tmp/erreur.txt
echo "La commande a renvoyé un code de retour $?"
echo "-------------------------------------"
echo "La commande a affiché sur la sortie d'erreur :"
cat /tmp/erreur.txt
```

```
Ceci est un texte à supprimer
Ceci est un texte à supprimer
La commande a renvoyé un code de retour 1
-------------------------------------
La commande a affiché sur la sortie d'erreur :
cat: /tmp/cefichiernexistepas: Aucun fichier ou dossier de ce type
rm: impossible de supprimer '/tmp/cefichiernexistepas': Aucun fichier ou dossier de ce type

cat: /tmp/cefichiernexistepas: Aucun fichier ou dossier de ce type
```

Avec un périphérique virtuel

Certains systèmes de fichiers sont dit virtuels, parce qu'il ne correspondent pas à un stockage réel sur un périphérique, mais ont leur contenu fixé par un programme (souvent, un morceau de code du noyau).

C'est le cas notamment sous Linux de `/proc` et `/dev` . Le premier permet d'exposer certaines valeurs internes du noyau (quand on peut écrire, ça permet de *donner* des valeurs au noyau, plutôt que de les lire). Le deuxième permet un accès brut aux périphériques (quand ça a du sens).

À une autre échelle, certains *périphériques* sont virtuels, parce qu'ils correspondent à des fichiers qui n'existent pas non plus ; ils se comportent comme si c'étaient de vrais fichiers (on peut les ouvrir et les fermer, les lire ou écrire dessus), mais ce ne sont pas de vrais fichiers. C'est le cas en particulier de certains comme `/dev/null` qui est toujours un fichier vide.

In [9]:

```
%%sh
echo "Activité : explorez un peu /proc et /dev"

echo "Quel est le contenu de /proc/cpuinfo ? Que reconnaissez-vous ?"
echo "Quel est le contenu de /proc/meminfo ? Que reconnaissez-vous ?"
echo "Quel est le contenu de /proc/uptime ? Cherchez sur internet la signification des deux nombres."

echo "Mettez du contenu dans /dev/null avec un cat ou un echo, puis vérifiez le contenu de /dev/null"
echo "Vérifiez en hexadécimal le contenu des 20 premiers octets de /dev/random"
hexdump -v -n 20 /dev/random
echo "Vérifiez en hexadécimal le contenu des 20 premiers octets de /dev/zero"
hexdump -v -n 20 /dev/zero
```

```
Activité : explorez un peu /proc et /dev
Quel est le contenu de /proc/cpuinfo ? Que reconnaissez-vous ?
Quel est le contenu de /proc/meminfo ? Que reconnaissez-vous ?
Quel est le contenu de /proc/uptime ? Cherchez sur internet la signification des deux nombres.
Mettez du contenu dans /dev/null avec un cat ou un echo, puis vérifiez le contenu de /dev/null
Vérifiez en hexadécimal le contenu des 20 premiers octets de /dev/random
0000000 99e4 ab88 206f 6583 6c60 a1cd 7059 6921
0000010 4c84 fddb
0000014
Vérifiez en hexadécimal le contenu des 20 premiers octets de /dev/zero
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000
0000014
```

Avec un autre descripteur de fichier

Il est possible aussi de rediriger en *copiant* un descripteur de fichier existant. C'est une copie, pas un lien ; c'est-à-dire que si on modifie ensuite l'original, on ne modifie **pas** la copie.

La syntaxe est simple:

commande n>&m

copie le descripteur m sur le descripteur n.

À partir du moment où deux descripteurs sont identiques, il n'y a, de fait, qu'une seule file d'écriture en même temps dans le fichier de sortie (ou le terminal). Les deux deviennent indistinguables.

Exemple:

```
ls /tmp/nofile / 2>&1 1>/tmp/listing.txt
```

Cette commande va afficher un message d'erreur sur l'écran (en mode normal)

In [10]:

```
%%sh

# Comparez avec la cellule suivante :

ls /tmp/nofile /usr/share/dict
```

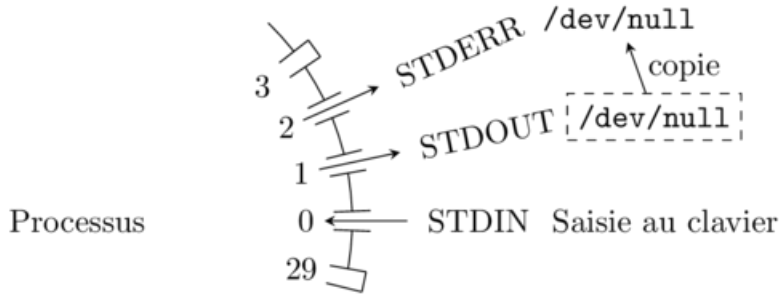
```
/usr/share/dict:
american-english
cracklib-small
french
README.select-wordlist
words
words.pre-dictionaries-common

ls: impossible d'accéder à '/tmp/nofile': Aucun fichier ou dossier de ce type
```

In [11]:

```
%%sh
ls /tmp/nofile /usr/share/dict 2>&1
```

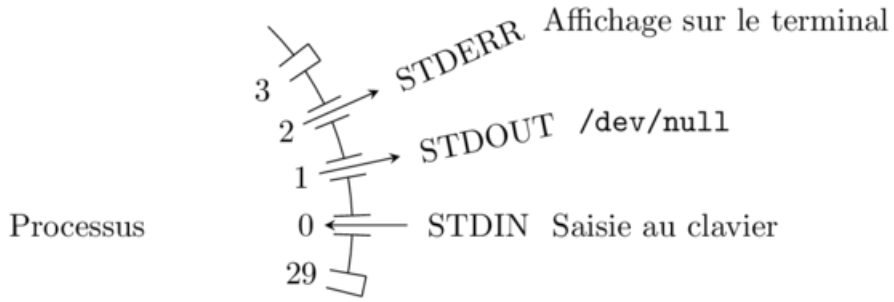
```
ls: impossible d'accéder à '/tmp/nofile': Aucun fichier ou dossier de ce type
/usr/share/dict:
american-english
cracklib-small
french
README.select-wordlist
words
words.pre-dictionaries-common
```

In [12]:



Exemple d'un processus avec
`1>/dev/null 2>&1`

In [13]:



Exemple d'un processus avec
`2>&1 1>/dev/null`

Communication inter-processus

Il est possible pour le shell de créer une communication inter-processus directement à la création des processus. Cette communication utilise pour cela la sortie standard d'un programme et l'entrée standard d'un autre. En connectant l'entrée standard du deuxième à la sortie standard du deuxième, on peut ainsi fabriquer une cascade de programmes dont le premier fait une partie du traitement, et le deuxième la suite.

Pour utiliser au mieux cette possibilité, on a créé tout un tas d'outils qui ont la particularité d'agir comme des filtres par défaut : ils consomment des données sur l'entrée standard, et sortent des données sur la sortie standard après les avoir traitées.

Le processus de création de communication est assez simple et permet ainsi d'enchaîner les petits outils pour ajuster les données.

La connaissance de ces outils de base qui permettent de traiter les données est importante pour maîtriser la puissance du shell. Elle réduit le besoin de boucles complexes puisque ces outils ont déjà les fonctions intégrées.

La syntaxe est simple :

```
commande1 | commande2 --option delacommande2 | commande3 --option delacommande3 | commande4
```

crée 4 processus et met la sortie de commande1 dans l'entrée de commande2, la sortie de commande2 dans l'entrée de commande3, la sortie de commande3 dans l'entrée de commande4. L'entrée de commande1 est normale (sauf si on a utilisé d'autres éléments comme au-dessus), la sortie de commande4 aussi.

Cette communication se crée à l'aide de l'appel système `pipe` , qui donne son nom à cette technique (*tuyau* en français).

Exemple :

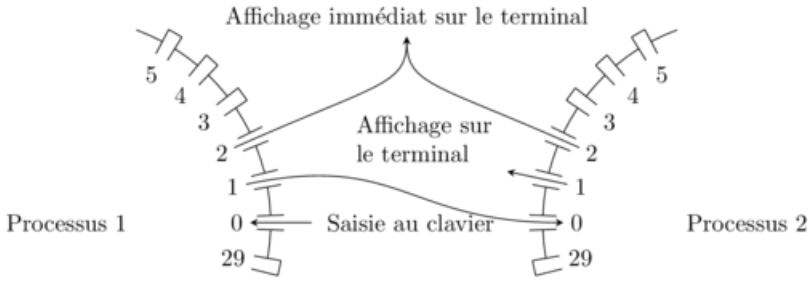
```
cat /usr/share/dict/french | head  # liste les 10 premiers mots de la liste alphabétique
```

In [14]:

```
%%sh
# Exemple : trouver les 10 premiers mots par ordre alphabétique de la langue française
cat /usr/share/dict/french | head

# Trouver le préfixe de 4 lettres le plus utilisé en français
cat /usr/share/dict/french | iconv -f utf8 -t iso-8859-1 | cut -c1-4 | sort | uniq -c | sort -nr | head -n 1
| iconv -t utf8 -f iso-8859-1
# Explication
# On prend la liste des mots
# On convertit en un codage de caractères où un caractère = un octet
# On coupe à 4 octets (caractère) chacun des mots
# On les trie
# On compte ceux qui se suivent et qui sont identiques
# On trie le résultat par ordre numérique inverse (le plus nombreux devant donc)
# On prend le premier
# On le remet dans le codage du terminal (ISO-8859-1 → UTF-8)
# Le résultat est affiché.
```

```
a
à
à-côté
à-côtés
à-coup
à-coups
à-peu-près
à-pic
à-propos
à-valoir
   2565 cont
```

In [15]:



Exemple d'un processus avec `commande1 | commande2`

La communication avec le shell

Le shell lui-même peut aussi communiquer avec les processus qu'il a créé pour récupérer les résultats au lieu de les mettre dans un fichier. Cela se fait à l'aide d'une syntaxe particulière qui permet de rediriger le résultat dans une variable.

La syntaxe est assez simple :

```
A=$(head -n 10 < /usr/share/dict/french |tail -n 1)
echo "La dixième ligne du fichier est $A"
```

La commande qui est à l'intérieur peut être aussi complexe que désirée. Dans l'exemple, on a ainsi superposé une redirection et un pipe. Toutefois, ce qui est récupéré est la sortie standard du dernier processus, si celle-ci est redirigé, la variable finira donc vide.

Les filtres unix

Les filtres sont une collection de petits utilitaires.

Ils ont tous le même principe :

- Ils traitent des fichiers en mode texte en prenant en entrée une suite de lignes et en donnant en sortie une suite de lignes
- Sans aucun argument, ils agissent sur l'entrée standard et donnent le résultat sur la sortie standard. S'il y a des arguments (pas des options), ils sont pris comme un ensemble de fichiers qui sont mis bout-à-bout comme s'ils venaient de l'entrée standard.
- Pour chaque filtre, quelques options sont présentées avec un peu d'explication pour contrôler l'action du filtre.

Réordonner et sélectionner des lignes

Les premières lignes : head

Récupère les n premières lignes de la sortie. Avec un n négatif, récupère tout sauf les n dernières. Par défaut, n=10.

```
head -n 20 /usr/share/dict/french
```

Les dernières lignes : tail

Récupère les n dernières lignes de la sortie. Avec un n négatif, récupère tout sauf les n premières. Par défaut, n=10.

```
tail -n 20 /usr/share/dict/french
```

Trouver un motif : grep

Permet de ne garder que les lignes qui comportent un certain motif (dans un premier temps, vous vous contenterez de motifs simples comme "une suite de lettres"). Avec l'option `-v` les lignes qui correspondent au motif sont enlevées et non gardées.

```
grep oom /usr/share/dict/french # trouve tous les mots qui ont oom dedans
```

Deux autres motifs sont utiles et à connaître :

- `^` marque le début de la ligne

- `$` marque la fin de la ligne

Exemples:

```
grep -v ^$ < /tmp/monfichier.txt # Affiche le fichier en enlevant toutes les lignes vides
grep ^bar < /usr/share/dict/french # Affiche tous les mots commençant par bar
```

Trier les lignes : sort

Attend toutes les lignes, et les ressort triées dans l'ordre alphabétique (en fait dans l'ordre de valeur des octets). Avec l'option -n, trie comme des nombres (c'est à dire que 110 est après 99 et non pas entre 10 et 13). Avec l'option -r, trie à l'envers. Si on veut trier suivant un champ, il faut préciser deux options : `-t: -k 3` trie à partir du troisième élément si on coupe la ligne selon le symbole ':'.

```
du -k . | sort -nr | head # Donne les fichiers les plus gros d'un répertoire
sort -n -t: -k 4 < /etc/passwd # Trie le fichier selon le champ n°4 numériquement
```

Rendre les lignes uniques : uniq

Supprime les lignes consécutives qui sont identiques à la précédente. Souvent combiné avec sort juste avant. Avec l'option -u, ne donne que les lignes qui sont uniques ; avec l'option -d, que les lignes qui sont au moins en double (mais une seule fois). Enfin, `uniq -c` comptera le nombre d'apparition de chaque ligne.

```
sort prenoms_eleves.txt | uniq -d # Trouve tous les prénoms en double
```

Modifier les lignes

Couper verticalement : cut

Pour traiter des données rapidement en shell, surtout dans des formats texte, la commande `cut` est indispensable.

Le premier usage de `cut` est de pouvoir couper verticalement les lignes, à savoir les couper suivant les positions des octets depuis le début de la ligne.

Exemple :

```
ls -l /usr/bin | cut -c1-10 | sort | uniq -c # permet de récupérer les différents modèles de permissio
ns dans /usr/bin
```

La sélection des octets se fait par un paramètre de l'option -c. On peut mettre des numéros d'octets séparés par des virgules, ou un intervalle (séparé par des tirets) : `cut -c -4,8,9,14,18-23,28-` retiendra uniquement les octets 1 à 4, 8, 9, 14, 18 à 23, et tous les octets à partir du 28^{e}. Attention : les octets sont numérotés à partir de 1.

Couper selon des séparations : cut (encore)

Le deuxième usage de `cut` est de pouvoir couper verticalement les lignes selon des *champs*, à savoir les couper suivant les positions d'un séparateur (qui est un caractère), et dire quels champs on retient.

Exemple :

```
cat /etc/passwd | cut -f1,5 -d: # identifiant d'utilisateur et nom réel des utilisateurs systèmes
```

La sélection des octets se fait par un paramètre de l'option -c. On peut mettre des numéros d'octets séparés par des virgules, ou un intervalle (séparé par des tirets) : `cut -c -4,8,9,14,18-23,28-` retiendra uniquement les octets 1 à 4, 8, 9, 14, 18 à 23, et tous les octets à partir du 28^{e}. Attention : les octets sont numérotés à partir de 1.

Modifier les lettres : tr

Permet de modifier les lettres d'une ligne.

Le premier mode permet de changer une lettre en une autre :

```
echo "Les oiseaux se cachent pour mourir" | tr 'aeiou' 'eioua'
```

Le deuxième mode permet de supprimer des lettres

```
echo "Les oiseaux se cachent-ils pour mourir ? C'est la question." | tr -d "aeiou'-"
```

On peut écrire `a-z` au lieu de `abcdefghijklmnopqrstuvwxyz` . Du coup, le `-` s'il apparaît doit toujours être le dernier caractère.

Changer le codage des caractères : iconv

Permet de transformer les lignes (vues comme des suites d'octets) d'un codage à un autre codage. Cela suppose que les caractères contenus dans le texte avec le codage de départ sont aussi contenus dans le codage d'arrivée.

Exemple:

```
iconv -f utf-8 -t iso-8859-1 monfichier.txt # Convertit un texte au format utf-8 au format iso-8859-1
```

L'utilité principale est qu'un certain nombre des utilitaires de ce cours fonctionnent non pas sur des caractères (ce qui est pratique), mais sur des octets. Lorsque c'est possible, se ramener à un codage de caractère où chaque caractère est sur un octet permet donc de mettre de côté la différence.

Montrer le contenu octet par octet : hexdump

Permet de transformer les lignes en une représentation de leur contenu octet par octet (en hexadécimal). L'option `-C` permet d'afficher également dans les colonnes voisines le contenu ASCII, ce qui permet sur un document principalement texte de voir rapidement les morceaux intéressants.

Exemple:

```
cat fichiers/villes.csv|hexdump -C|less
```

Paginer le contenu d'un fichier : less

Permet *uniquement lorsqu'il est en bout de chaîne* de paginer l'affichage, c'est-à-dire d'afficher un écran de texte, puis d'attendre pour passer à la page suivante (touche espace, `q` pour quitter).

Exemple:

```
cat fichiers/villes.csv|hexdump -C|less # oui, pareil qu'au dessus
```

Il est à remarquer que ce programme a un comportement légèrement anormal pour un filtre : il agit différement selon qu'il doit écrire sur un terminal ou dans un fichier ou pour un autre programme (notamment, il ne fait que passer l'entrée inchangée si ce n'est pas un terminal). C'est le cas d'autres programmes et c'est, de façon générale, une mauvaise pratique.

Analyser les lignes

Compter les lignes ou les mots : wc

Permet de compter les lignes, les mots et les octets d'un texte. Renvoie une seule ligne qui comprend 3 nombres. Avec l'option `-l` , un seul nombre est renvoyé (lignes), ainsi qu'avec l'option `-c` (octets).

Sauvegarder un résultat intermédiaire : tee

Permet de faire une copie dans un fichier intermédiaire de tout ce qui passe. Sinon, ne change rien au passage. Le nom de la commande vient d'une représentation sous forme de plomberie :

```
Commande1---Commande2-T-Commande3---Commande4
                  fichier
```

Le `tee` agit comme une sorte de tuyau en T au milieu de la chaîne de commandes.

```
ls -l /bin | sort | grep sh | tee /tmp/fichiers-avec-sh.txt | grep li > /tmp/fichiers-avec-li-et-sh.tx
t
```

In [16]:

```
Avec les outils à votre disposition, et à partir du fichier villes.csv qui est fourni dans le répertoire fich
iers, déterminez les éléments suivants :
Le nombre de villes du fichier (attention à la ligne d'enête)
36700
Le nombre de villes dont le nom contient la lettre Z
2494
Les 5 villes les plus peuplées de France en 2010
Paris
Marseille
Lyon
Toulouse
Nice
Le nombre de communes du département 38
533
Le nombre de communes dont le nom comporte un accent ou une cédille
7990
L'encodage UTF8 des villes avec Sébastien dans leur nom
00000000  50 72 c3 a9 61 75 78 2d  53 61 69 6e 74 2d 53 c3  |Pr..aux-Saint-S.|
00000010  a9 62 61 73 74 69 65 6e  0a 53 61 69 6e 74 2d 53  |.bastien.Saint-S|
00000020  c3 a9 62 61 73 74 69 65  6e 0a 42 6f 75 74 65 69  |..bastien.Boutei|
00000030  6c 6c 65 73 2d 53 61 69  6e 74 2d 53 c3 a9 62 61  |lles-Saint-S..ba|
00000040  73 74 69 65 6e 0a 53 61  69 6e 74 2d 53 c3 a9 62  |stien.Saint-S..b|
00000050  61 73 74 69 65 6e 2d 64  65 2d 4d 6f 72 73 65 6e  |astien-de-Morsen|
00000060  74 0a 53 61 69 6e 74 2d  53 c3 a9 62 61 73 74 69  |t.Saint-S..basti|
00000070  65 6e 2d 64 27 41 69 67  72 65 66 65 75 69 6c 6c  |en-d'Aigrefeuill|
00000080  65 0a 53 61 69 6e 74 2d  53 c3 a9 62 61 73 74 69  |e.Saint-S..basti|
00000090  65 6e 0a 53 61 69 6e 74  2d 53 c3 a9 62 61 73 74  |en.Saint-S..bast|
000000a0  69 65 6e 2d 73 75 72 2d  4c 6f 69 72 65 0a 53 61  |ien-sur-Loire.Sa|
000000b0  69 6e 74 2d 53 c3 a9 62  61 73 74 69 65 6e 2d 64  |int-S..bastien-d|
000000c0  65 2d 52 61 69 64 73 0a                           |e-Raids.|
000000c8
```

Récapitulatif des commandes vues dans ce chapitre

Commande	Utilité
head	Filtrer les premières lignes d'un fichier
tail	Filtrer les premières lignes d'un fichier
grep	Filtrer les lignes d'un fichier selon un motif
sort	Trier les lignes d'un fichier
uniq	Dédoublonner les lignes d'un fichier
cut	Découper verticalement les lignes d'un fichier
tr	Changer les caractères d'un fichier
iconv	Changer le jeu de caractère d'un fichier
wc	Compter les lignes d'un fichier
tee	Sauvegarder le résultat d'une chaîne de filtres dans un fichier

In [17]:

```
Mode interactif
```