Initiation à l'environnement Unix
CM8 : Processus (3) tubes et statut de sortie

Pierre Rousselin

Université Paris 13
L1 informatique
décembre 2021

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

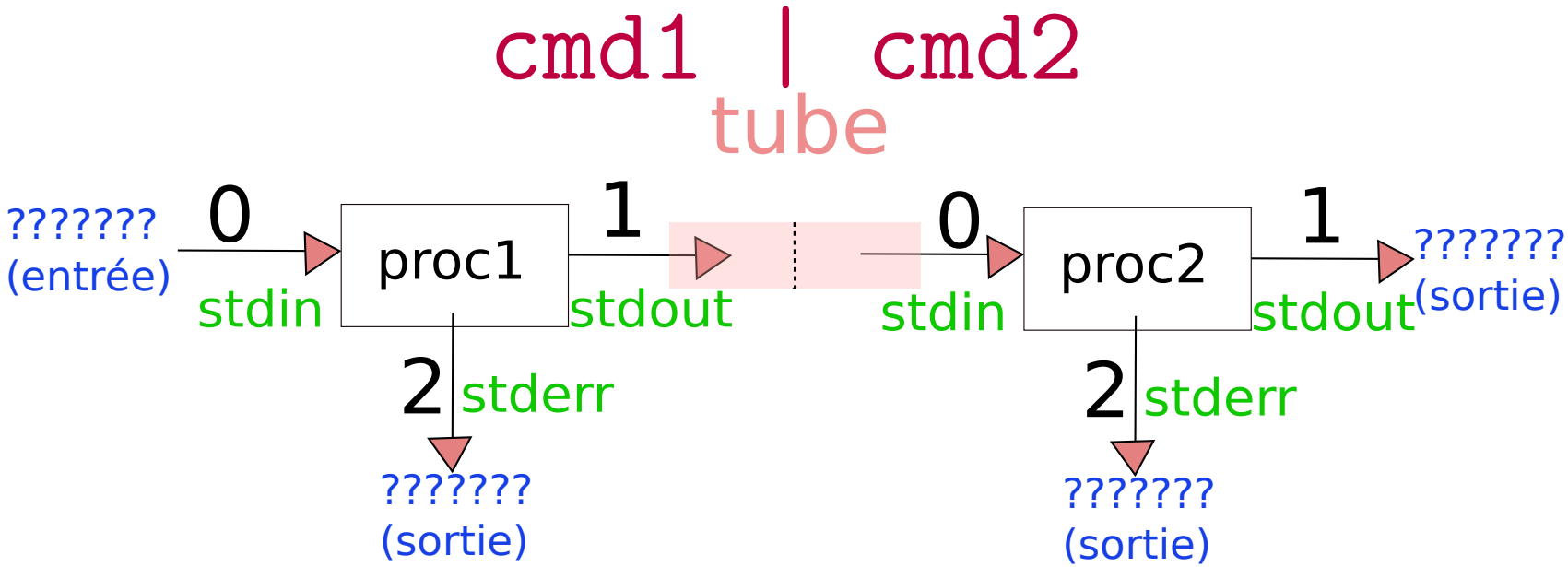
Illumination

```
cmd1 | cmd2
```

Douglas McIlroy (1932–)

Un jour, Doug McIlroy, à la tête du département de recherche en techniques informatiques de Bell Labs (berceau d'Unix) entre 1965 et 1986, a inventé le tube (*pipe*).

Schéma d'un tube



Le tube est un fichier particulier.
La sortie standard de `cmd1` est reliée à l'entrée du tube.
L'entrée standard de `cmd2` est reliée à la sortie du tube.

Exemples de tubes

Question : combien y a-t-il de fichiers `.h` dans le répertoire `/usr/include` ?

Exemples de tubes

Question : combien y a-t-il de fichiers `.h` dans le répertoire `/usr/include` ? Sans tube :

```
$ printf '%s\n' /usr/include/*.h
/usr/include/aio.h
/usr/include/aliases.h
... trop de fichiers !
$ printf '%s\n' /usr/include/*.h >includes.txt
$ wc -l includes.txt # compte le nombre de lignes
206
$ rm includes.txt
```

Exemples de tubes

Question : combien y a-t-il de fichiers `.h` dans le répertoire `/usr/include` ? Sans tube :

```
$ printf '%s\n' /usr/include/*.h
/usr/include/aio.h
/usr/include/aliases.h
... trop de fichiers !
$ printf '%s\n' /usr/include/*.h >includes.txt
$ wc -l includes.txt # compte le nombre de lignes
206
$ rm includes.txt
```

Avec un tube :

```
$ printf '%s\n' /usr/include/*.h | wc -l
206
```

Exemples de tubes

Question : Combien y a-t-il de scripts shell dans `/bin/` ?
Premier ingrédient : la commande `file` va nous aider, elle « devine » des informations sur le fichier.
Par exemple,

```
$ file /bin/rm
bin/rm: ELF 64-bit LSB pie executable, x86-64, ...
```

montre que `/bin/rm` est un fichier exécutable compilé (en langage machine), et donc pas un script.

Exemples de tubes

Question : Combien y a-t-il de scripts shell dans `/bin/` ?
Premier ingrédient : la commande `file` va nous aider, elle « devine » des informations sur le fichier.
Par exemple,

```
$ file /bin/rm
bin/rm: ELF 64-bit LSB pie executable, x86-64, ...
```

montre que `/bin/rm` est un fichier exécutable compilé (en langage machine), et donc pas un script.
Autre exemple :

```
$ file /bin/ipython3
bin/ipython3: Python script, ASCII text executable
```

montre que mon fichier `/bin/ipython3` est un script python.

Exemples de tubes

Question : Combien y a-t-il de scripts shell dans `/bin/` ?
Deuxième ingrédient : la commande **grep** cherche une chaîne dans un fichier ou plusieurs fichiers, ou si aucun n'est fourni en argument dans son entrée standard, et affiche les lignes qui la contiennent.

```
$ cat fj
Frère Jacques, frère Jacques,
Dormez-vous, dormez-vous,
Sonnez les matines, sonnez les matines !
Ding ! Ding ! Dong !
$ grep 'ez' fj # argument fichier
Dormez-vous, dormez-vous,
Sonnez les matines, sonnez les matines !
$ grep '!' <fj # lecture entrée standard
Sonnez les matines, sonnez les matines !
Ding ! Ding ! Dong !
```

Exemples de tubes

Question : Combien y a-t-il de scripts shell dans `/bin/` ?

```
$ file /bin/* | grep 'shell script'
/bin/rpmdev-packager: Bourne-Again shell script, UTF-8...
/bin/rpminfo:         Bourne-Again shell script, ASCII...
/bin/maxima:          POSIX shell script, ASCII...
(...)
$ file /bin/* | grep 'shell script' | wc -l
55
```

Exemples de tubes

Question : Combien y a-t-il de scripts shell dans `/bin/` ?

```
$ file /bin/* | grep 'shell script'
/bin/rpmdev-packager: Bourne-Again shell script, UTF-8...
/bin/rpminfo:         Bourne-Again shell script, ASCII...
/bin/maxima:          POSIX shell script, ASCII...
(...)
$ file /bin/* | grep 'shell script' | wc -l
55
```

Remarques :

- Bien sûr, le résultat dépend de votre installation.
- Utiliser `file` n'est pas parfait, la commande fait de son mieux pour deviner le type du fichier, mais peut se tromper.
- Ça donne tout de même une idée (sans doute correcte dans ce cas).

Tubes et flux de données

Deux visions différentes des fichiers s'affrontent maintenant

- le « fichier comme un livre » :
  - début et fin du fichier bien définis et connus à l'avance ;
  - possibilité d'aller directement à telle ou telle page (en Unix, appel système `lseek` pour *long seek*).

Tubes et flux de données

Deux visions différentes des fichiers s'affrontent maintenant

- le « fichier comme un livre » :
  - début et fin du fichier bien définis et connus à l'avance ;
  - possibilité d'aller directement à telle ou telle page (en Unix, appel système `lseek` pour *long seek*).
- le fichier comme un *flux* (en anglais *stream*) de données :
  - écrire dans le fichier revient à alimenter le flux, comme si on versait de l'eau dans un tuyau ;
  - lire dans le fichier revient à consommer du flux, comme si on utilisait l'eau qui sort du tuyau ;
  - fin du fichier imprévisible (théoriquement, le fichier pourrait même être infini).

Filtres

Un filtre est un processus qui :

- lit des données sur son entrée standard ;
- les transforme d'une certaine manière ;
- écrit le résultat de ces transformations sur sa sortie standard.

Filtres

Un filtre est un processus qui :

- lit des données sur son entrée standard ;
- les transforme d'une certaine manière ;
- écrit le résultat de ces transformations sur sa sortie standard.

Exemples :

- `tr`, déjà mentionné, en est l'exemple parfait ;
- `cut` : sélectionner, pour chaque ligne du flux, certaines colonnes ;
- `grep` : sélectionner les lignes du flux contenant une chaîne donnée ;
- `head` : sélectionner seulement les premières lignes du flux ;
- `tail` : sélectionner seulement les dernières lignes du flux ;

Exemples (suite) :

- `uniq` : effacer les lignes consécutives qui sont identiques ;
- `sed` : *Stream EDitor*, éditeur de flux, permet de faire à peu près ce qu'on veut sur un flux (voir `sedtris`) ;
- `awk` : langage de programmation simple créé par Aho, Kernighan et Weinberger pour écrire des filtres ;
- `sort` : lire toutes les lignes du flux puis les trier.

Compositions des filtres

Afficher le nombre de processus pour chaque utilisateur :

Compositions des filtres

Afficher le nombre de processus pour chaque utilisateur :

```
ps -eo user= | sort | uniq -c | sort -nr
```

Compositions des filtres

Afficher le nombre de processus pour chaque utilisateur :

```
ps -eo user= | sort | uniq -c | sort -nr
```

- filtre $\leftrightarrow$ fonction
- tube `|` $\leftrightarrow$ composition des fonctions $\circ$

Compositions des filtres

Afficher le nombre de processus pour chaque utilisateur :

```
ps -eo user= | sort | uniq -c | sort -nr
```

- filtre $\leftrightarrow$ fonction
- tube `|` $\leftrightarrow$ composition des fonctions $\circ$

Philosophie Unix :

- *Make each program do one thing well.* (Doug McIlroy, 1978)
- *(...) the power of a system comes more from the relationships among programs than from the programs themselves.* (Brian Kernighan et Rob Pike, 1984)

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

Mort d'un processus

Lorsqu'un processus se termine (on dit qu'il *meurt*) :

- toutes les ressources qui lui étaient allouées sont libérées ;

Mort d'un processus

Lorsqu'un processus se termine (on dit qu'il *meurt*) :

- toutes les ressources qui lui étaient allouées sont libérées ;
- il transmet, dans un ultime effort, un message *à son parent* : un petit entier entre 0 et 255.
- Ce petit entier vaut :

Mort d'un processus

Lorsqu'un processus se termine (on dit qu'il *meurt*) :

- toutes les ressources qui lui étaient allouées sont libérées ;
- il transmet, dans un ultime effort, un message *à son parent* : un petit entier entre 0 et 255.
- Ce petit entier vaut :
  - 0 si, d'après ce processus, tout s'est passé sans problème particulier ;
  - > 0 si, encore d'après ce processus, une erreur ou quelque chose d'anormal s'est produit.
- Raisonnement : il n'y a rien à dire si tout s'est déroulé comme prévu, alors que les sources d'erreur peuvent être multiples (permissions insuffisantes sur un fichier, fichier inexistant, pas assez d'arguments, ...).

Succès et erreur

Le paramètre spécial $? contient le code de retour de la dernière commande exécutée.

```
$ ls /usr/
bin games include lib lib64 libexec local sbin share src tmp
$ echo $?
0
$ ls /ust/
ls: impossible d'accéder à '/ust/':
Aucun fichier ou dossier de ce type
$ echo $?
2
$ sleep 30 # Ctrl-C interrompt la commande en cours (SIGINT)
^C
$ echo $?
130
```

Succès et erreur

```
$ azerty
bash: azerty: commande inconnue...
$ echo $?
127
```

- C'est le programme lui-même qui choisit ses codes de retours d'erreur, entre 1 et 125 et les documente (plus ou moins bien...).
- Le code de retour 127 correspond à une commande non trouvée.
- Le code de retour 126 correspond à une commande trouvée mais non exécutable.
- Les codes de retour > 128 correspondent à une commande qui s'est terminée à cause d'un signal qu'elle a reçu (dans l'exemple `sleep` précédent, le signal `SIGINT` envoyé par le shell car l'utilisateur a tapé `Ctrl-c`).

La commande `exit`

- Dans un script shell, la commande `exit` met fin au script.
- Après la commande `exit 0`, le script se termine avec ce que l'on considère être un succès, c'est-à-dire avec le code de retour 0.
- Après la commande `exit 1` ou `exit 2` ou ... ou `exit 125`, le script se termine en indiquant au shell qu'une erreur s'est produite.
- Après la commande `exit` (sans argument), le script se termine et son code de retour est celui de la dernière commande exécutée.

En langage C

```c
int main()
{
    return 0;
}
```

En langage C

```c
int main()
{
    return 0;
}
```

- En langage `C`, c'est la valeur de retour de la fonction `main` qui sert de statut de sortie.
- La constante symbolique `EXIT_SUCCESS` vaut `0` sur les systèmes Unix.
- ```
  $ grep EXIT_SUCCESS /usr/include/*.h
  /usr/include/stdlib.h:#define        EXIT_SUCCESS        0
  $ grep EXIT_FAILURE /usr/include/*.h
  /usr/include/stdlib.h:#define        EXIT_FAILURE        1
  ```
  ```

En langage C

```c
int main()
{
    return 0;
}
```

- En langage `C`, c'est la valeur de retour de la fonction `main` qui sert de statut de sortie.
- La constante symbolique `EXIT_SUCCESS` vaut `0` sur les systèmes Unix.
- ```
  $ grep EXIT_SUCCESS /usr/include/*.h
  /usr/include/stdlib.h:#define        EXIT_SUCCESS        0
  $ grep EXIT_FAILURE /usr/include/*.h
  /usr/include/stdlib.h:#define        EXIT_FAILURE        1
  ```
- On peut aussi mettre fin au processus depuis n'importe quelle fonction avec l'appel système `void exit(int status)` qui prend en argument le statut de sortie que l'on souhaite transmettre.
  ```

Récapitulatif

- Code de retour 0 : succès, vrai.

Récapitulatif

- Code de retour 0 : succès, vrai.
- Code de retour > 0 : échec, faux.

Récapitulatif

- Code de retour 0 : succès, vrai.
- Code de retour > 0 : échec, faux.
- C'est le contraire de ce dont on a l'habitude (par exemple en C)

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

Premier exemple : `mkmv`

Un script qui crée un répertoire et y déplace des fichiers.

```
#!/bin/sh
# mkmv rep fichier...
# crée un répertoire, et y déplace les fichiers
rep=$1
if mkdir "$rep"; then
    printf "Le répertoire $rep a été créé\n"
    shift
    mv "$@" "$rep"
else
    status=$?
    printf "Erreur dans la création du répertoire $rep!\n"
    exit $status
fi
```

Premier exemple : mkmv

```
$ chmod u+x mkmv
$ touch ab ac ad
$ ./mkmv truc/ ab ac ad
Le répertoire truc a été créé
$ ls truc/
ab ac ad
$ touch bc bd
$ mkdir machin
$ ./mkmv machin/ bc bd
mkdir: impossible de créer le répertoire « machin/ »: Le fichier existe
Erreur dans la création du répertoire machin/!
$ echo $?
1
```

Syntaxe

```
if commande_if
then
    commandes_bloc_if
elif commande_elif1
then
    commandes_bloc_elif1
elif commande_elif2
then
    commandes_bloc_elif2
else
    commandes_bloc_else
fi
```

- Le mot-clé **then** *doit* suivre un saut de ligne ou bien un point-virgule.
- On peut mettre autant de **elif** que l'on veut.
- Les parties **elif** et **else** sont optionnelles.

Sémantique

- La commande `commande_if` est exécutée et en cas de succès (code de retour 0), les commandes `commandes_bloc_if` sont exécutées et le script continue son exécution après le mot-clé `fi` ;
- En cas d'échec de `commande_if` (code de retour > 0), s'il y a lieu la commande `commande_elif1` est exécutée et si elle réussit (code de retour 0), les commandes `commandes_bloc_elif1` sont exécutées et le script continue son exécution après le mot-clé `fi` ;
- En cas d'échec de `commande_elif1`, la commande `commande_elif2`, s'il y a lieu, est exécutée, etc.
- En d'échec de `commande_if`, `commande_elif1`, ..., si la partie `else` est présente, les commandes `commandes_bloc_else` sont exécutées.

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

true et false

- La commande `true` ne fait rien d'autre que se terminer avec un code de retour 0 ;
- La commande `false` ne fait rien d'autre que se terminer avec un code de retour 1.

Remarque : la commande vide `:` (deux-points) peut remplacer la commande `true`.

La commande `test`, ou `[`

La commande `test`, aussi appelée `[` (crochet ouvrant) permet de vérifier le type d'un fichier ou de comparer des valeurs (chaînes de caractères ou numériques).

```
$ ls
bc bd mkmv shell_cm5.tex truc
$ test -e bc # est-ce que le fichier existe ?
$ echo $?
0
$ [ -e bzz ] # même chose (syntaxe [ ... ])
$ echo $?
1
$ [ -x mkmv ] # fichier exécutable ?
$ echo $?
0
$ [ -d truc ]; echo $? # répertoire ?
0
```

La commande `test`, ou `[`

La commande `test`, aussi appelée `[` (crochet ouvrant) permet de vérifier le type d'un fichier ou de comparer des valeurs (chaînes de caractères ou numériques).

```
$ ls
bc bd mkmv shell_cm5.tex truc
$ test -e bc # est-ce que le fichier existe ?
$ echo $?
0
$ [ -e bzz ] # même chose (syntaxe [ ... ])
$ echo $?
1
$ [ -x mkmv ] # fichier exécutable ?
$ echo $?
0
$ [ -d truc ]; echo $? # répertoire ?
0
```

Syntaxe : `[` est une commande et `]` est son dernier argument : il faut mettre des espaces.

`test` et les fichiers

On ne se souvient jamais de tout ce que peut faire `test`, voir `man test` ou `help test` (`test` est intégrée à la plupart des shells pour des raisons de performance). Quelques commandes `test` très utilisées pour les fichiers :

`test` et les fichiers

On ne se souvient jamais de tout ce que peut faire `test`, voir `man test` ou `help test` (`test` est intégrée à la plupart des shells pour des raisons de performance). Quelques commandes `test` très utilisées pour les fichiers :

- `[ -e nom_chemin ]` vrai (code de retour 0) si le fichier de chemin `nom_chemin` existe, faux sinon ;
- `[ -d nom_chemin ]` vrai si existe et est un répertoire ;
- `[ -f nom_chemin ]` vrai si existe et est un fichier normal ;
- `[ -r nom_chemin ]`, `[ -w nom_chemin ]`, `[ -x nom_chemin ]`, vrai si existe et est un fichier pour lequel le shell courant a la permission `r` (respectivement `w` et `x`).

`test` et les chaînes

- `[ -z chaine ]` vrai si la chaîne est vide, c'est-à-dire a pour longueur 0 (*zero*).
- `[ chaine ]` ou `[ -n chaine ]` vrai si la chaîne n'est pas vide, c'est-à-dire n'a pas pour longueur 0 (*non zero*).
- `[ ch1 = ch2 ]` vrai si les chaînes `ch1` et `ch2` sont égales.
- `[ ch1 != ch2 ]` vrai si les chaînes `ch1` et `ch2` sont différentes.

`test` et les nombres entiers

Dans les formes qui suivents, les arguments numéro 1 et 3 de `test` sont interprétées comme des nombres entiers.

- `[ n1 -eq n2 ]` vrai si les nombres `n1` et `n2` sont égaux (*equal*).
- `[ n1 -ne n2 ]` vrai si les nombres `n1` et `n2` sont différents (*not equal*).
- `[ n1 -gt n2 ]` vrai si le nombre `n1` est supérieur à (*greater than*) `n2`.
- `[ n1 -lt n2 ]` vrai si le nombre `n1` est inférieur à (*less than*) `n2`.
- `[ n1 -ge n2 ]` vrai si le nombre `n1` est supérieur ou égal à (*greater than or equal to*) `n2`.
- `[ n1 -le n2 ]` vrai si le nombre `n1` est inférieur ou égal à (*less than or equal to*) `n2`.

Exemple

```
#!/bin/sh
# mkmv rep fichier...
# version 2
# crée un répertoire s'il n'existe pas,
# et y déplace les fichiers
if [ $# -lt 2 ]; then
    printf "Nombre d'arguments insuffisants\n"
    exit 1
fi
rep=$1
if [ -d "$rep" ]; then
    : # il faudrait vérifier les permissions w et x
else
    mkdir "$rep"
fi
shift
mv "$@" "$rep"
```

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

Premier exemple

```
#!/bin/sh
# mdp : script nul de demande de mot de passe
mdp=secret
reponse=
printf "Mot de passe : "
IFS= read -r reponse
while [ "$reponse" != "$mdp" ]; do
    printf "Mot de passe incorrect\n"
    printf "Mot de passe : "
    IFS= read -r reponse
done
printf "Bienvenue sur $(hostname), $USER\n"
```

Syntaxe

```
while commande
do
    cmds_bloc_while
done
```

- Le mot-clé `do` doit être précédé d'un saut de ligne ou d'un point-virgule.
- Au passage : pour lire une ligne de l'entrée standard et la mettre dans la variable `reponse` :

  ```
  IFS= read -r reponse
  ```

  - On vide l'IFS juste pour cette commande (sinon `read` va séparer la ligne en mots...)
  - Option `-r` pour lui dire de ne pas interpréter spécialement les contre-obliques...

Sémantique

1. La commande `commande` est exécutée.
2. En cas d'échec de `commande`, (code de retour > 0), la prochaine commande exécutée est celle qui suit le mot-clé **`done`**.
3. En cas de succès de `commande`, (code de retour 0), les commandes entre les mots-clés **`do`** et **`done`** sont exécutée, puis l'on retourne au point 1.

Boucler un certain nombre de fois

La boucle `for` du shell est très différente de celles de langages plus classiques qui servent à boucler un nombre de fois connu à l'avance, avec un compteur. On s'en sort sans problème avec `while` et un développement arithmétique :

Boucler un certain nombre de fois

La boucle **for** du shell est très différente de celles de langages plus classiques qui servent à boucler un nombre de fois connu à l'avance, avec un compteur. On s'en sort sans problème avec **while** et un développement arithmétique :

```
N=10
i=0 # initialisation
while [ $i -lt $N ] # condition
do
    printf "carré de $i : $(( $i * $i ))\n"
    i=$(( $i + 1)) # incrémentation
done
```

Boucle infinie, `break` et `continue`

Boucle infinie :

```
while true; do
    commandes_while1
    if commande_if; then
        break
    fi
    commandes_while2
done
```

- La commande `break` permet de sortir de la boucle.
- La commande `continue` permet de passer directement à l'itération suivante.
- Avec un argument numérique (par exemple `break 2` ou `continue 3`), on peut faire référence à une boucle extérieure.

Les tubes

Statut de sortie

Construction `if`

Les commandes qui n'existent que pour leur code de retour

Construction `while`

Mots-clés `&&`, `||` et `!`

Négation

Avant une commande, le mot-clé `!` (point d'exclamation) nie simplement le code de retour de la commande :

- Si le code de retour de `cmd` est 0 (succès, vrai), alors celui de `! cmd` est 1.
- Si le code de retour de `cmd` est > 0 (échec, faux), alors celui de `! cmd` est 0.

```
$ ls /usr/
bin games include lib lib64 libexec local sbin share src tmp
$ echo $?
0
$ ! ls /usr/
bin games include lib lib64 libexec local sbin share src tmp
$ echo $?
1
$ ! ls /ust/
ls: impossible d'accéder à '/ust/'...
$ echo $?
0
```

Conjonction court-circuit

Rappel de logique : table de vérité de « et » (P et Q sont des variables propositionnelles).

P	Q	P et Q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

Conjonction court-circuit

Rappel de logique : table de vérité de « et » (P et Q sont des variables propositionnelles).

P	Q	P et Q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

On remarque que si P est faux, ce n'est pas la peine de considérer Q, le résultat sera de toute façon faux !
C'est le principe de l'évaluation *court-circuit* des opérations logiques (*short-circuit evaluation*).

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

```
cmd1 && cmd2 && ... && cmdn
```

et est exécutée de la façon suivante :

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.
- En cas de succès, la commande suivante (`cmd2`) est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd2`.
- En cas de succès, etc
- Si toutes les commandes réussissent, le code de retour de la liste-et est 0.

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé &&.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.
- En cas de succès, la commande suivante (`cmd2`) est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd2`.
- En cas de succès, etc
- Si toutes les commandes réussissent, le code de retour de la liste-et est 0.

```
if [ -e "$rep" ] && ! [ -d "$rep" ]; then
    printf "$rep existe et n'est pas un répertoire !\n"
    exit 1
fi
```

Disjonction court-circuit

Rappel de logique : table de vérité de « ou » (P et Q sont des variables propositionnelles).

P	Q	P ou Q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

Disjonction court-circuit

Rappel de logique : table de vérité de « ou » (P et Q sont des variables propositionnelles).

P	Q	P ou Q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

On remarque que si P est vrai, ce n'est pas la peine de considérer Q, le résultat sera de toute façon vrai !

Le mot-clé `||`

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

```
cmd1 || cmd2 || ... || cmdn
```

et est exécutée de la façon suivante :

Le mot-clé `||`

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

`cmd1 || cmd2 || ... || cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.

Le mot-clé `||`

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

`cmd1 || cmd2 || ... || cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, la commande suivante (`cmd2`) est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, etc
- Si toutes les commandes échouent, le code de retour de la liste-ou est celui de la dernière commande.

Le mot-clé ||

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

```
cmd1 || cmd2 || ... || cmdn
```

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, la commande suivante (`cmd2`) est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, etc
- Si toutes les commandes échouent, le code de retour de la liste-ou est celui de la dernière commande.

```
mkdir rep_temp || exit 1
```