Cours UNIX
Les bases du Shell

Sommaire

1 Définition du shell

Le shell est un programme ayant pour fonction d'assurer l'interface entre l'utilisateur et le système Unix. C'est un interpréteur de commandes.

Plusieurs shells sont disponibles sur les plates-formes Unix.

2 Caractéristiques d'un interpréteur de commandes

Les interpréteurs de commandes disponibles en environnement Unix ont en commun les fonctionnalités suivantes :

- Ils permettent aux utilisateurs de lancer des commandes.
- Ils proposent un jeu de caractères spéciaux permettant de déclencher des actions particulières.
- Ils possèdent des commandes internes et des mots-clés parmi lesquels certains sont utilisés pour faire de la programmation (écriture de scripts shell).
- Ils utilisent des fichiers d'initialisation permettant à un utilisateur de paramétrer son environnement de travail.

Chaque shell propose ses propres caractères spéciaux, commandes internes, mots-clés et fichiers de paramétrage. Heureusement, les interpréteurs les plus utilisés actuellement dérivent tous du shell Bourne et ont, par conséquent, un certain nombre de fonctionnalités en commun.

3 Interpréteurs de commandes (shells)

3.1 Historique

Le shell qui est considéré comme le plus ancien est le Bourne shell (**sh**). Il a été écrit dans les années 1970 par Steve Bourne aux laboratoires AT&T. Outre sa capacité à lancer des commandes, il offre des fonctionnalités de programmation. Le Bourne shell offrant moins de fonctionnalités que ses successeurs, il est de moins en moins utilisé sur les plates-formes Unix.

Durant la même période, Bill Joy invente le C-shell (**csh**), incompatible avec le Bourne, mais qui offre des fonctionnalités supplémentaires telles que l'historique des commandes, le contrôle de tâches, ainsi que la possibilité de créer des alias de commandes. Ces trois aspects seront repris plus tard dans le Korn Shell. Le C-shell est peu utilisé dans le monde Unix.

En 1983, David Korn reprend le Bourne shell et l'enrichit. Ce nouvel interpréteur prendra le nom de Korn Shell (**ksh**). Ce dernier sera de plus en plus employé et deviendra un standard de fait. Le **ksh88** (version datant de 1988) est, avec le Bourne Again Shell (voir ci-dessous), le shell le plus utilisé actuellement. Il a servi de base à la normalisation du shell (IEEE POSIX 1003.2), représentée par le **shell POSIX** (similaire au ksh88). Pour simplifier, nous pouvons dire que le shell POSIX possède les mêmes fonctionnalités que le ksh88.

En 1993, une nouvelle version du Korn Shell voit le jour (**ksh93**). Celle-ci présente une compatibilité arrière avec le ksh88, à quelques exceptions près. Le ksh93 est disponible sur certaines versions Unix récentes : Solaris 11, AIX 6 et 7, HP-UX 11 (à télécharger).

La Free Software Foundation propose le Bourne Again Shell (**bash**). Il est conforme à la norme **POSIX** à laquelle il a ajouté quelques extensions. Ce shell est l'interpréteur fourni en standard sur les systèmes Linux. Il est également disponible en standard ou en téléchargement sur les systèmes Unix. Les dernières versions du bash portent les numéros 4.4 et 5.0 (version 5.0 sortie en janvier 2019).

3.2 Avec quel shell faut-il programmer ?

3.2.1 Scripts de démarrage

L'utilisation du Bourne shell est en voie de disparition. Les plates-formes Solaris inférieures à la version 11 utilisent encore ce shell pour l'interprétation des scripts de démarrage. Si l'on souhaite modifier ces scripts ou en créer de nouveaux, il faut dans ce cas se restreindre à la syntaxe du Bourne shell. Les plates-formes HP-UX et AIX interprètent les scripts de démarrage avec un shell ksh88. Quant à Solaris 11, il utilise un shell ksh93. Les systèmes Linux utilisent le shell bash.

3.2.2 Autres scripts

Dans les cas les plus fréquents (scripts de traitement exécutés en mode de fonctionnement Unix normal), le développeur choisira soit le bash, soit le ksh (88 ou 93), selon le(s) shell(s) à disposition sur sa plate-forme. Le ksh et le bash ayant de très nombreuses fonctionnalités en commun, cela ne pose donc aucun problème d'écrire un script compatible avec ces deux shells.

Si les spécificités du ksh93 sont utilisées, le script ne sera pas compatible avec les shells Bourne, ksh (88) et bash.

4 Commandes internes et externes

Ce chapitre présente et explique de manière détaillée les fonctionnalités de base du shell couramment utilisées dans les commandes Unix.

Une commande Unix appartient à l'une des deux catégories suivantes.

4.1 Les commandes externes

Une commande externe est un fichier localisé dans l'arborescence. Par exemple, lorsqu'un utilisateur lance la commande ls, le shell demande au noyau Unix de charger en mémoire le fichier /usr/bin/ls.

Sont considérés comme commandes externes les fichiers possédant l'un des formats suivants :

- Fichiers au format binaire exécutable.
- Fichiers au format texte représentant un script de commandes (qui peut être écrit en Shell ou dans un autre langage tel que Python, Perl...).

La commande file donne une indication sur le type de données contenues dans un fichier.

Exemples

La commande **ls** *est un fichier au format binaire exécutable. Résultat de la commande* **file** *:*

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

La commande **monscript.sh** *est un script shell. Résultat de la commande* **file** *:*

```
$ file /home/christie/monscript.sh
monscript.sh: ascii text
```

4.2 Les commandes internes

Une commande interne est intégrée au processus shell (c'est un mot-clé du shell). C'est donc le shell qui exécute l'action. La commande interne ne correspond donc en aucun cas à un fichier sur le disque.

Exemples

La commande **cd** *est une commande interne.*

```
$ type cd
cd is a shell builtin
```

La commande **ls** *est une commande externe.*

```
$ type ls
ls is /usr/bin/ls
```

type *prend en argument le nom d'une commande. Si cette dernière n'est pas interne, elle est recherchée dans les répertoires cités dans PATH (cf. chapitre Paramétrage de l'environnement de travail - Variables d'environnement).*

4.3 Implémentation interne et implémentation externe

Certaines commandes ont une implémentation interne et une implémentation externe. Dans ce cas :

- la commande interne est lancée en priorité ;
- l'exécution d'une commande interne est plus rapide que l'exécution d'une commande externe ;
- la commande **type** indique que la commande est interne, mais ne précise pas qu'il existe également une implémentation externe.

5 Affichage à l'écran

5.1 La commande echo

La commande interne **echo** permet de réaliser des affichages à l'écran.

Exemple

```
$ echo Voici un livre sur la programmation shell!
Voici un livre sur la programmation shell!
$
```

Certains caractères ont, lorsqu'ils sont placés entre quotes (apostrophes ou guillemets), une signification spéciale. Ce sont des caractères d'échappement.

*La commande **echo** du bash doit être utilisée avec l'option **-e** pour que l'interprétation des caractères d'échappement ait lieu.*

5.1.1 Le caractère \n

Il sert à provoquer un saut de ligne.

Exemples avec un Bourne ou Korn shell

```
$ echo "Voici un saut de ligne\net encore un autre\net le saut de
ligne naturel de la commande echo"
Voici un saut de ligne
et encore un autre
et le saut de ligne naturel de la commande echo
$
```

Les quotes sont obligatoires :

```
$ echo a\nb
anb
$ echo "a\nb"
a
b
$
```

Exemples avec un bash

```
$ echo "a\nb"
a\nb
$ echo -e "a\nb"
a
b
$
```

5.1.2 Le caractère \c

Il sert à éliminer le saut de ligne naturel de la commande **echo**.

Le caractère ***\c*** *doit se situer impérativement en dernière position de l'argument de* ***echo*** *(juste avant le guillemet fermant).*

Exemples avec un Bourne ou Korn shell

```
$ echo "Premiere ligne" ; echo "Deuxieme ligne"
Premiere ligne
Deuxieme ligne
$ echo "Premiere ligne\c" ; echo "Deuxieme ligne\c"
Premiere ligneDeuxieme ligne$
```

Exemples avec un bash

L'option **-e** *est indispensable pour l'interprétation de* **\c***.*

```
$ echo -e "Premiere ligne\c" ; echo -e "Deuxieme ligne\c"
Premiere ligneDeuxieme ligne$
```

L'option **-n** *remplace* **\c**

```
$ echo -n "Premiere ligne" ; echo -n "Deuxieme ligne"
Premiere ligneDeuxieme ligne$
```

5.1.3 Le caractère \t

Il permet d'afficher une tabulation.

Exemple avec un Bourne ou Korn shell

```
$ echo "Voici 2 tabulations\t\tEt voila ..."
Voici 2 tabulations Et voila ...
$
```

Exemple avec un bash

```
$ echo -e "Voici 2 tabulations\t\tEt voila ..."
Voici 2 tabulations Et voila ...
$
```

5.1.4 Liste des caractères d'échappement

Caractère d'échappement	**Signification**
\\	**Antislash**
\a	**Sonnerie**
\b	**Effacement du caractère précédent**
\c	**Suppression du saut de ligne en fin de ligne**
\f	**Saut de page**
\n	**Saut de ligne**
\r	**Retour chariot**
\t	**Tabulation horizontale**
\v	**Tabulation verticale**
\0xxx	**Valeur d'un caractère exprimée en octal**

6 Le caractère ~ (tilde)

Le caractère **~** représente le répertoire d'accueil de l'utilisateur courant.

Exemples

L'utilisateur courant se nomme **christie** *:*

```
$ id
uid=505(christie) gid=505(ociensa)
```

Le répertoire courant est /tmp *:*

```
$ pwd
/tmp
```

Copie du fichier /tmp/f1 *dans le répertoire d'accueil (/home/christie) :*

```
$ cp f1 ~
```

Copie du fichier /tmp/f1 *sous le répertoire /home/christie/docs :*

```
$ cp f1 ~/docs
```

Si le caractère ~ est immédiatement suivi d'un mot, ce dernier est considéré comme un nom d'utilisateur.

Exemples

Recopier le fichier **f1** *sous le répertoire d'accueil de l'utilisateur* **sebastien** *(nous supposons que les permissions adéquates sont positionnées) :*

```
$ cp f1 ~sebastien
$ ls /home/sebastien/f1
f1
$
```

Recopier le fichier **f1** *sous le répertoire /home/sebastien/rep :*

```
$ cp f1 ~sebastien/rep
$ ls /home/sebastien/rep
f1
```

7 La commande interne cd

Nous présentons ici les syntaxes particulières de la commande **cd**.

Exemples

La commande **cd** *sans argument permet à l'utilisateur de revenir dans son répertoire d'accueil :*

```
$ cd
```

Même chose en utilisant le caractère ~ :

```
$ cd ~
```

Se déplacer dans le répertoire d'accueil de l'utilisateur **sebastien** *:*

```
$ pwd
/home/christie
$ cd ~sebastien
$ pwd
/home/sebastien
```

Revenir dans le répertoire précédent avec **cd -** *:*

```
$ cd -
/home/christie
```

8 Substitution de noms de fichiers

De nombreuses commandes prennent des noms de fichier en argument. Ces derniers peuvent être cités littéralement ou être spécifiés de manière plus générique. Le shell propose un certain nombre de caractères spéciaux qui permettent de fabriquer des expressions utilisées comme Modèles de noms de fichier.

*8.1 Le caractère **

Il représente une suite de caractères quelconques (entre 0 et n caractères).

Exemples

```
$ ls
f12 f1.i FICa fic.c fic.s monscript.pl MONSCRIPT.pl ours.c
```

Afficher tous les noms de fichier se terminant par .c *:*

```
$ ls *.c
fic.c ours.c
```

Afficher tous les noms de fichier commençant par la lettre f *:*

```
$ ls f*
f12 f1.i fic.c fic.s
```

8.2 Le caractère ?

Il représente un caractère quelconque.

Exemples

Afficher tous les noms de fichier ayant une extension composée d'un caractère :

```
$ ls *.?
f1.i fic.c fic.s ours.c
```

Afficher tous les noms de fichier composés de quatre caractères :

```
$ ls ????
f1.i FICa
```

8.3 Les caractères [ ]

Les crochets permettent de spécifier la liste des caractères que l'on attend à une position bien précise dans le nom du fichier. Il est également possible d'utiliser les notions d'intervalle et de négation.

Exemples

Fichiers dont le nom commence par f *ou* o *et se termine par le caractère* **.** *suivi d'une minuscule :*

```
$ ls [fo]*.[a-z]
f1.i fic.c fic.s ours.c
```

Fichiers dont le nom comporte en deuxième caractère une majuscule ou un chiffre ou la lettre **i***. Les deux premiers caractères seront suivis d'une chaîne quelconque :*

```
$ ls ?[A-Z0-9i]*
f12 f1.i FICa fic.c fic.s MONSCRIPT.pl
```

Il est également possible d'exprimer la négation de tous les caractères spécifiés à l'intérieur d'une paire de crochets. Ceci se fait en plaçant un **!** en première position à l'intérieur de celle-ci.

Exemple

Noms de fichier ne commençant pas par une minuscule :

```
$ ls [!a-z]*
FICa MONSCRIPT.pl
```

Il est bien sûr possible de spécifier plusieurs expressions sur la ligne de commande. Celles-ci devront être séparées les unes des autres par un espace.

Exemple

Supprimer tous les fichiers dont le nom se termine par .c *ou* .s *:*

```
$ rm -i *.c *.s
rm: remove `fic.c'? y
rm: remove `ours.c'? y
rm: remove `fic.s'? y
$
```

9 Séparateur de commandes

9.1 Le point virgule

Le caractère spécial **;** du shell permet d'écrire plusieurs commandes sur une même ligne. Les commandes sont exécutées séquentiellement, même si l'une d'elle génère un erreur.

Exemple

```
$ pwd
/home/christie
$ ls
$ mkdir rep ; cd rep ; pwd
/home/christie/rep
```

9.2 Le « && »

Il est parfois utile d'exécuter une commande que si les commandes précédentes ont réussies. Par exemple si l'on souhaite vérifier l'existence d'un dossier et, si il existe, y créer un fichier :

```
$ cd /home/~/dossier1 && touch file1
```

Ici, si la commande "cd /home/~/dossier1" ne s'exécute pas correctement (que le dossier n'existe pas), la commande "touch file1" qui créer un fichier vide ne s'exécutera pas non plus et par conséquent ne déclenchera pas de message d'erreur.

Le code de retour est utilisé par le shell pour déterminer le succès ou non de la commande. Il vaut "0" si la commande s'est exécutée correctement et à "2" dans le cas où le dossier n'existe pas.

Pour déterminer si une commande a bien réussi, Unix analyse ce code de retour et, dans le cas d'un "0" avec un enchaînement de commande "&&", il exécute la commande suivante. S'il vaut autre chose, c'est que la commande précédente ne s'est pas correctement déroulée et il met donc fin à l'exécution des prochaines commandes.

9.3 Le "||"

Dans d'autres cas, il peut être intéressant d'exécuter une commande uniquement si la commande précédente ne se déroule pas correctement au lieu de mettre fin à toute la ligne de commande. Cela est possible avec les caractères "||".

Par exemple si l'on souhaite, comme auparavant, voir si un dossier existe et si c'est le cas y créer un fichier. On peut également vouloir créer le dossier si il n'est pas encore présent :

```
cd /home/~/dossier1 || mkdir / /home/~/dossier1 && touch file1
```

C'est un exemple simple mais on peut imaginer des choses beaucoup plus complexes. Ici, si le code de retour n'est pas "0" (= la commande "cd /home/~/dossier1" à échouée), on créera le fichier avec la commande "mkdir". Dans un second temps suite à cela on créera le fichier "file1".

10 Redirections

Les redirections sont couramment utilisées dans les commandes Unix. Elles permettent de récupérer le résultat d'une ou de plusieurs commandes dans un fichier ou au contraire de faire lire un fichier à une commande. Cette partie expose de manière détaillée les différentes syntaxes possibles avec leur mécanisme interne associé.

Les redirections sont mises en place par le shell.

10.1 Entrée et sorties standards des processus

Les processus Unix ont, par défaut, leur fichier terminal ouvert trois fois, sous trois descripteurs de fichier différents.

10.1.1 Entrée standard

Le descripteur de fichier 0 est nommé également **entrée standard du processus**. Les processus qui attendent des informations de la part de l'utilisateur déclenchent en fait une requête de lecture sur le descripteur 0. Si ce dernier est associé au terminal, ce qui est le cas par défaut, cela se matérialise pour l'utilisateur par une demande de saisie au clavier.

*La majorité des commandes utilisent l'entrée standard pour déclencher une saisie. Il existe cependant des exceptions. Par exemple, la commande **passwd** ouvre le fichier terminal sous un autre descripteur.*

10.1.2 Sortie standard

Le descripteur de fichier 1 est nommé également **sortie standard du processus**. Par convention, un processus qui souhaite envoyer un message résultat à l'utilisateur doit le faire transiter via le descripteur 1. Si ce dernier est associé au terminal, ce qui est le cas par défaut, cela se matérialise pour l'utilisateur par un affichage à l'écran.

10.1.3 Sortie d'erreur standard

Le descripteur de fichier 2 est nommé également **sortie d'erreur standard du processus**. Par convention, un processus qui souhaite envoyer un message d'erreur à l'utilisateur doit le faire transiter via le descripteur 2. Si ce dernier est associé au terminal, ce qui est le cas par défaut, cela se matérialise pour l'utilisateur par un affichage à l'écran.

10.2 Redirection des sorties en écriture

La redirection en écriture permet d'envoyer les affichages liés à un descripteur particulier, non plus sur le terminal, mais dans un fichier.

10.2.1 Sortie standard

Simple redirection

Syntaxe

```
$ commande 1> fichier
```

équivalent à :

```
$ commande > fichier
```

Le nom du fichier est exprimé en relatif ou en absolu.

Si le fichier n'existe pas, il est créé. Si le fichier existe déjà, il est écrasé.

Exemple

Récupérer le résultat de la commande **ls** *dans le fichier* ***resu*** *:*

```
$ ls > resu
$ cat resu
FIC
erreur
out1
resu
$
```

Double redirection

Elle permet de **concaténer** les messages résultat d'une commande au contenu d'un fichier déjà existant.

Syntaxe

```
$ commande 1>> fichier
```

équivalent à :

```
$ commande >> fichier
```

Si le fichier n'existe pas, il est créé. Si le fichier existe déjà, il est ouvert en mode ajout.

Exemple

Ajouter le résultat de la commande **date** *à la fin du fichier* ***resu*** *créé précédemment :*

```
$ date >> resu
$ cat resu
FIC
erreur
out1
resu
lun Jan 21 18:31:56 CET 2019
$
```

10.2.2 Sortie d'erreur standard

Simple redirection

Syntaxe

```
$ commande 2> fichier
```

Exemple

Redirection de la sortie d'erreur standard. Les messages d'erreur partent dans le fichier ***erreur****, les résultats restent à l'écran :*

```
$ find / -name passwd 2> erreur
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ cat erreur
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
...
$
```

Double redirection

Elle permet de concaténer les messages d'erreur d'une commande au contenu d'un fichier existant.

Syntaxe

```
$ commande 2>> fichier
```

Exemple

Concaténation des messages d'erreur de **ls -z** *à la fin du fichier* ***erreur*** *:*

```
$ ls -z
ls: illegal option -- z
usage: ls -1RaAdCxmnlogrtucpFbqisfL [files]
$ ls -z 2>> erreur
$ cat erreur
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
ls: illegal option -- z
usage: ls -1RaAdCxmnlogrtucpFbqisfL [files]
```

```
$
```

10.2.3 Sortie standard et sortie d'erreur standard

Il est possible de rediriger plusieurs descripteurs sur une même ligne de commande.

Syntaxe

```
$ commande 1> fichier_a 2> fichier_b
```

ou :

```
$ commande 2> fichier_b 1> fichier_a
```

Les redirections sont toujours traitées de gauche à droite. Dans le cas présent, l'ordre d'écriture des deux redirections n'a pas d'importance, ce qui ne sera pas toujours le cas

Exemple

```
$ find / -name passwd 1> resu 2> erreur
$ cat resu
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ cat erreur
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/aset: Permission denied
...
$
```

10.2.4 Se protéger d'un écrasement involontaire de fichier

L'option **noclobber** du shell permet de se protéger d'un écrasement involontaire de fichier. Cette option est désactivée par défaut. Elle peut être paramétrée de manière permanente dans les fichiers **.kshrc** ou **.bashrc**.

Exemple

```
$ ls resu
resu
$ set -o noclobber
$ date > resu
-bash: resu : impossible d'écraser le fichier existant
```

Pour forcer l'écrasement il faut utiliser le symbole de redirection **>|** :

```
$ ls >| resu
```

10.2.5 Éliminer les affichages

Toutes les plates-formes Unix possèdent un fichier spécial nommé**/dev/null** qui permet de faire disparaître les affichages. Ce fichier est géré comme un périphérique et n'a pas de notion de contenu. On peut donc considérer qu'il est toujours vide.

Exemple

```
$ find / -name passwd 1> resu 2> /dev/null
$ cat resu
/var/adm/passwd
/usr/bin/passwd
/etc/default/passwd
/etc/passwd
$ ls -lL /dev/null
crw-rw-rw- 1 root sys 13, 2 Jan 21 17:22 /dev/null
$ cat /dev/null
```

10.3 Redirection de l'entrée standard

La redirection de l'entrée standard concerne les commandes qui utilisent le descripteur 0, autrement dit celles qui déclenchent une saisie au clavier.

Exemple

```
$ mail olive
RV a 13 heures au restaurant (Entrée standard)
Christine (Entrée standard)
^d (Entrée standard)
$
```

La commande **mail** lit l'entrée standard jusqu'à réception d'une fin de fichier (touches ^d). Les données saisies seront envoyées dans la boîte aux lettres de l'utilisateur **olive**.

Si l'on souhaite faire lire à la commande **mail**, non plus le clavier, mais le contenu d'un fichier, il suffit de connecter le descripteur 0 sur le fichier désiré.

Syntaxe

```
$ commande 0< fichier_message
```

équivalent à

```
$ commande < fichier_message
```

Exemple

```
$ cat message
RV a 13 heures au restaurant
Christine
```

```
$ mail olive < message
$
```

10.4 Redirections avancées

10.4.1 Rediriger les descripteurs 1 et 2 vers le même fichier

Pour envoyer la sortie standard et la sortie d'erreur standard dans le même fichier, il faut employer une syntaxe particulière. Voici ce qu'il ne faut pas écrire et la raison pour laquelle cela ne fonctionne pas.

Syntaxes incorrectes

```
$ commande 1> fichier 2> fichier
$ commande 1> fichier 2>> fichier
```

Le problème ne réside pas dans le fait que l'on ouvre deux fois le même fichier (ce qui est parfaitement légal au sein d'un même processus), mais qu'il y a un offset (position courante dans le fichier) associé à chaque ouverture. Quelles sont les conséquences ?

- La séquence des résultats dans le fichier ne sera pas forcément représentative de l'ordre dans lequel se sont déroulés les évènements.
- Les résultats émis à travers les descripteurs 1 et 2 risquent de se chevaucher.

Syntaxes correctes

Pour obtenir un résultat correct, il faut utiliser l'une des deux syntaxes suivantes :

```
$ commande 1> fichier 2>&1
```

ou :

```
$ commande 2> fichier 1>&2
```

Traitement de la redirection **1> resu** :

Même mécanisme que précédemment : création du fichier **resu**, allocation d'un enregistrement dans la table des fichiers ouverts, offset à 0.

Traitement de la redirection **2>&1** :

Pour traduire cette expression en français, on peut dire que le descripteur 2 est redirigé sur le descripteur 1 (représenté par &1). La chose importante à comprendre est que le shell, grâce à cette syntaxe, duplique simplement l'adresse du descripteur 1 dans le descripteur 2. Les deux descripteurs pointent alors sur le même enregistrement de la table des fichiers ouverts, et par conséquent, partagent le même offset. Il n'y a pas d'allocation d'un nouvel enregistrement dans la table des fichiers ouverts du noyau. Les écritures ultérieures, qu'elles soient émises par la sortie ou la sortie d'erreur standard, se serviront du même offset.

11 Tubes de communication

Un tube (pipe en anglais) permet de faire communiquer deux processus. Le tube est représenté par une barre verticale située entre deux commandes Unix. Le résultat de la

commande de gauche va partir dans le tube, tandis que la commande de droite va en extraire les données afin de les traiter.

Quelques remarques importantes

- La sortie d'erreur standard de la commande de gauche ne part pas dans le tube.
- Pour que l'utilisation d'un tube ait un sens, il faut que la commande placée à gauche du tube envoie des données sur sa sortie standard et que la commande placée à droite lise son entrée standard.

11.1 Commandes ne lisant pas leur entrée standard

Un certain nombre de commandes Unix n'ont aucun intérêt à être placées derrière un tube, car elles n'exploitent pas leur entrée standard. C'est le cas par exemple des commandes suivantes : ls, who, find, chmod, cp, mv, rm, ln, mkdir, rmdir, date, kill, file, type, echo...

11.2 Commandes lisant leur entrée standard

Les commandes qui lisent leur entrée standard sont facilement identifiables car elles demandent une saisie au clavier.

11.2.1 Exemples triviaux

```
$ mail olive
saisie clavier
saisie clavier
^d
$
$ write olive
saisie clavier
saisie clavier
^d
$
```

Ces deux commandes peuvent donc être placées derrière un tube :

```
$ who | mail olive
$ echo "RV pour dejeuner a 13 heures" | write olive
```

11.2.2 Cas des filtres

Sous Unix, un certain nombre de commandes sont regroupées sous le nom de filtres. Les plus communes sont : grep, cat, sort, cut, wc, lp, sed, awk... Ces commandes peuvent fonctionner de deux manières :

Première manière

Si la commande reçoit au moins un nom de fichier en argument, elle traite le(s) fichier(s) et ne déclenche pas de lecture de l'entrée standard.

Exemple

```
$ wc -l /etc/passwd
46 /etc/passwd
```

Deuxième manière

La commande ne reçoit aucun nom de fichier en argument. Dans ce cas, la commande traite les données qui arrivent sur son entrée standard.

Exemple

La commande **wc** *compte le nombre de lignes qui arrivent sur son entrée standard et affiche le résultat sur la sortie standard :*

```
$ wc -l
saisie clavier (Entrée standard)
saisie clavier (Entrée standard)
saisie clavier (Entrée standard)
^d (Entrée standard - Fin de saisie clavier) 3 (Sortie standard)
$
```

Il est donc possible de placer cette commande derrière un tube :

```
$ who | wc -l
4
$
```

Comment savoir si une commande lit son entrée standard ?

Voici deux méthodes qui permettent de savoir si une commande lit son entrée standard :

- L'information est contenue dans le manuel de la commande.

Exemple

Voici un extrait de la page de manuel de la commande **wc***. On constate que l'argument* **[file …]** *est facultatif, ce qui est une première indication.*

```
$ man wc
...
SYNOPSIS
wc [ -c | -m | -C ] [ -lw ] [ file ... ]
...
```

Un peu plus, loin, se trouve l'explication de l'argument **file** *; si le nom de fichier est omis, la commande lit son entrée standard.*

```
...
file A path name of an input file. If no file operands
are specified, the standard input will be used.
...
```

- Une autre possibilité consiste à tester la commande sans donner de nom de fichier en argument.

Premier exemple

Voici une commande qui traite un fichier. Elle ne déclenche pas de lecture de l'entrée standard :

```
$ cut -d':' -f1,3 /etc/passwd
root:0 (Sortie standard)
bin:1 (Sortie standard)
daemon:2 (Sortie standard)
adm:3 (Sortie standard)
...
$
```

Voici la même commande sans le nom du fichier. La commande attend une saisie au clavier :

```
$ cut -d':' -f1,3
1:2:3:4 (Entrée standard)
1:3 (Sortie standard)
10:20:30:40 (Entrée standard)
10:30 (Sortie standard)
100:200:300:400 (Entrée standard)
100:300 (Sortie standard)
^d (Entrée standard)
$
```

Cette commande peut donc être placée derrière un tube :

```
$ echo "1:2:3:4" | cut -d':' -f1,3
1:3
$
```

Deuxième exemple

Voici une autre commande qui traite un fichier.

```
$ file /etc/passwd
/etc/passwd: ASCII text
```

La même commande sans le nom du fichier génère un message d'erreur. Le nom du fichier est donc obligatoire. Cette commande ne lit pas son entrée standard et ne peut pas être placée à droite d'un tube :

```
$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
Usage: file -C [-m magic]
```

Cas particulier de certaines commandes

La majorité des commandes ne se soucient pas de savoir si elles sont placées derrière un tube ou non. Pour une commande donnée, l'action sera toujours la même.

Exemple

wc -l *lit son entrée standard dans ces deux cas :*

```
$ wc -l
$who | wc -l
```

Quelques commandes font exception à la règle. Elles testent si leur entrée standard est connectée sur la sortie d'un tube ou sur un terminal. C'est le cas de la commande **more**.

Exemples

La commande **more** *reçoit un nom de fichier en argument et pagine son contenu à l'écran. Elle ne lit pas son entrée standard :*

```
$ more /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
....
--Encore--(43%)
```

Sans le nom du fichier, la commande affiche un message d'erreur (l'argument [filename] est pourtant facultatif !). Ici, **more** *ne lit pas son entrée standard :*

```
$ more
Usage: more [-cdflrsuw] [-lines] [+linenumber] [+/pattern] [filename ...].
```

Le nom du fichier peut être omis lorsque **more** *est placée à droite d'un tube. Dans ce cas, elle lit son entrée standard et pagine les lignes qu'elle y extrait :*

```
$ cat /etc/passwd | more
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
....
--Encore--(43%)
$
```

11.3 Compléments

11.3.1 Enchaîner des tubes

Il est possible d'enchaîner plusieurs tubes sur une ligne de commande.

Exemple

Afficher le nombre de connexions de l'utilisateur christie :

```
$ who | grep christie | wc -l
3
```

11.3.2 Dupliquer les sorties

La commande **tee** permet de visualiser un résultat à l'écran et de le conserver également dans un fichier.

Exemples

La commande **tee** *affiche sur sa sortie standard les lignes extraites du tube et les écrit également dans le fichier* ***listefic****. Si* ***listefic*** *existe déjà, il est écrasé :*

```
$ ls | tee listefic
Desktop
FIC
fichier
$ cat listefic
Desktop
FIC
fichier
$
```

Le résultat de la commande **date** *est affiché à l'écran et concaténé (ajout) au fichier* ***listefic*** *existant :*

```
$ date | tee -a listefic
lun jan 28 17:54:21 CET 2019
$ cat listefic
Desktop
```

```
FIC
fichier
lun jan 28 17:54:21 CET 2019
$
```

12 Regroupement de commandes

Le regroupement de commandes peut être utilisé pour :

- rediriger la sortie écran de plusieurs commandes vers un même fichier ou vers un tube ;
- faire exécuter plusieurs commandes dans le même environnement.

Exemple

*Seule la sortie standard de la deuxième commande est redirigée dans le fichier **resultat**.*

```
$ date ; ls > resultat
lun jan 28 05:16:30 CET 2019
$ cat resultat
FIC
fichier
$
```

Les parenthèses **( )** et les accolades **{ }** permettent de regrouper les commandes. Dans le premier cas, les commandes sont exécutées à partir d'un shell enfant, dans le deuxième cas à partir du shell courant.

12.1 Les parenthèses

Dans la plupart des cas, ce sont les parenthèses qui sont utilisées pour le regroupement de commandes.

Syntaxe

```
(cmde1 ; cmde2 ; cmde3)
```

Avec les parenthèses, un shell enfant est systématiquement créé et c'est ce dernier qui traite la ligne de commande (avec duplications ultérieures si nécessaire).

Premier exemple

Ici, l'utilisateur se sert des parenthèses pour rediriger la sortie standard de deux commandes :

```
$ (date ; ls) > resultat
$ cat resultat
lun jan 28 05:21:36 CET 2019
FIC
```

```
fichier
$
```

Deuxième exemple

Les commandes **pwd** *et* **ls** *ont pour répertoire courant* ***/tmp*** *:*

```
$ pwd
/home/christie
$ (cd /tmp ; pwd ; ls) > listefic
$ cat listefic
/tmp (résultat de pwd)
dcopNYSrKn (liste des fichiers de /tmp)
listetmp
...
```

Lorsque l'exécution des trois commandes est terminée, le shell de premier niveau reprend la main. Son répertoire courant est toujours ***/home/christie****.*

```
$ pwd
/home/christie
$
```

12.2 Les accolades

Syntaxe

```
{ cmde1 ; cmde2 ; cmde3 ; }
```

- Les accolades ouvrante et fermante doivent être respectivement suivies et précédées par un espace.
- La dernière commande doit être suivie d'un ;.

La ligne de commande est traitée par le shell courant (avec duplications ultérieures si nécessaire).

Premier exemple

Les deux commandes suivantes produisent le même résultat, mais la version avec accolades est plus rapide :

```
$ ( date ; ls) > resultat
$ { date ; ls ; } > resultat
```

Deuxième exemple

Ici, l'environnement du shell de premier niveau va être modifié, ce qui n'est pas forcément très intéressant :

```
$ pwd
/home/christie
$
$ { cd /tmp ; pwd ; ls ; } > listefic
$
$ cat listefic
/tmp
dcopNYSrKn
listetmp
$ pwd
/tmp
$
```

12.3 Conclusion

Les parenthèses sont plus utilisées que les accolades pour les deux raisons suivantes :

- leur syntaxe est plus simple à utiliser ;
- quel que soit le jeu de commandes, on est toujours sûr de retrouver l'environnement de travail initial.

L'utilisation des accolades se justifiera dans le cas d'une recherche de performance ou bien dans l'objectif de modifier l'environnement courant.