
Introduction au shell

Shell et langages de script

Isabelle Ryl & Yann Hodique

{ryl,hodique}@lifl.fr

LIFL / USTL Lille 1

Introduction à UNIX

Les programmes

Il existe deux catégories de logiciels :

- ▶ **les programmes système** s'occupent du fonctionnement des ordinateurs ;
- ▶ les programmes d'applications s'occupent des besoins des utilisateurs.

Le *système d'exploitation* d'un ordinateur est le programme fondamental de tous les programmes système. Il contrôle les ressources de l'ordinateur et fournit la base sur laquelle se construisent les programmes d'application.

Le système d'exploitation fonctionne sous deux modes différents :

- ▶ mode **noyau** (ou **superviseur**) ;
- ▶ mode **utilisateur**.

Qu'est-ce qu'un OS ?

« **Tout le code que vous n'avez pas écrit** » :-)

On donne généralement une définition double :

- ▶ une **machine virtuelle** : il masque les détails bas-niveau et construit des « couches logicielles » qui fournissent de plus en plus de fonctionnalités.

Exemple :

l'OS gère les faces/pistes/secteurs des disquettes.

- ▶ un **gestionnaire de ressources** : il gère les ressources matérielles, par exemple protège de l'accès simultané ou assure l'équité de l'accès aux ressources.

Exemple :

l'accès au lecteur de disquettes par deux utilisateurs est réalisé en exclusion mutuelle, le temps CPU est partagé entre les processus.

UNIX

Systeme multi-utilisateurs et multi-tâches composé de :

- ▶ un noyau (gestion mémoire, entrées/sorties bas niveau, gestion des tâches, ...);
- ▶ des outils de bases :
 - ▶ des shells ;
 - ▶ des commandes de manipulation de fichiers ;
 - ▶ des commandes de gestion des processus ;
 - ▶ des commandes pour la communication ;
 - ▶ des éditeurs, compilateurs,...

UNIX – Philosophie

- ▶ le code source est (souvent) disponible et facile à lire.
- ▶ l'interface utilisateur est simple.
- ▶ **il n'y a qu'un petit nombre de primitives, mais les combinaisons sont très nombreuses.**
- ▶ toutes les interfaces avec les périphériques sont unifiées (via le système de gestion des fichiers).
- ▶ le système est indépendant de l'architecture matérielle.

UNIX – caractéristiques

- ▶ système de fichiers hiérarchisé, arborescence unique (voir le FHS – Filesystem Hierarchy Standard).
- ▶ les shells.
- ▶ aspects multi-tâches, multi-utilisateurs.
- ▶ hiérarchie de processus et « génétique » de processus.
- ▶ appels système : points d'accès aux services du noyau.
- ▶ interface unique pour les entrées/sorties de tous types (compatibles avec la notion de **fichier**).
- ▶ allocation des entrées/sorties des processus (**redirection**,...).

Utilisateurs

- ▶ Chaque utilisateur se connecte en saisissant son nom d'utilisateur (ou **login**) et son mot de passe.
- ▶ Chaque utilisateur possède son propre espace : son **home directory**. Il peut être le seul à accéder à son espace ou choisir de donner des **droits** aux autres utilisateurs pour chacun de ses fichiers.
- ▶ Chaque utilisateur est identifié par un numéro (**uid**). La correspondance entre **uid** et **login** est effectuée grâce au fichier `/etc/passwd`.
- ▶ Les utilisateurs sont répartis en un certain nombre de groupes également identifiés par un numéro (**gid**).
- ▶ Les processus, comme les fichiers, ont des propriétaires, des groupes propriétaires. De plus, ils possèdent des droits en fonction de leur propriétaire.
- ▶ Le **super-utilisateur** `root` possède **TOUS** les droits.

Le Shell

- ▶ interpréteur de commandes, il est la première interface entre l'utilisateur et le système d'exploitation.
- ▶ lors de la connexion d'un utilisateur, un shell est lancé avec les caractéristiques associées à l'utilisateur (identifiant, droits, ...).
- ▶ le shell interprète les commandes, en mode interactif, ligne par ligne. Sauf demande contraire, les commandes sont lues sur l'**entrée standard** et les résultats affichés sur la **sortie standard**.
- ▶ si la commande est interne (ex : `cd`, affectation d'une variable), le shell l'exécute. Si la commande est externe (ex : `ls`) l'exécution est confiée à un nouveau processus.
- ▶ le shell est aussi un véritable langage de programmation.

En TP, le shell par défaut sera le `bash`

Syntaxe des commandes

Les différents langages de commande (shells) utilisent tous la même syntaxe générale pour la description d'une commande :

```
commande [options...] [arguments...]
```

Une commande peut (cela n'est pas obligatoire) être suivie :

- ▶ d'options qui précisent le mode de fonctionnement de la commande, une façon particulière de fonctionner.

COMMENT

- ▶ de paramètres ou arguments qui permettent de spécifier des éléments que la commande doit prendre en compte.

QUOI

Une ligne de commande peut comporter plusieurs commandes si elles sont séparées les unes des autres par le caractère *;*

La commande `man`

Le manuel UNIX est disponible en ligne par la commande

```
man [options] [section] <entrée recherchée>
```

Pour chaque commande, la page de manuel est organisée en paragraphes. Principaux paragraphes :

- ▶ NAME
- ▶ SYNOPSIS
- ▶ DESCRIPTION
- ▶ SEE ALSO
- ▶ DIAGNOSTICS
- ▶ BUGS

Organisation du manuel

Le manuel est organisé en sections. La section peut être spécifiée dans la commande :

```
bash$ man 1 kill
```

```
KILL(1)                Linux Programmer's Manual                KILL(1)
```

```
NAME
```

```
kill - terminate a process
```

```
SYNOPSIS
```

```
kill [ -s signal | -p ] [ -a ] [ -- ] pid ...
```

```
kill -l [ signal ]
```

```
...
```

Organisation du manuel(2)

```
bash$ man 2 kill
```

```
KILL(2)                Linux Programmer's Manual                KILL(2)
```

NAME

```
kill - send signal to a process
```

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

DESCRIPTION

```
The kill system call can be used to send any
signal to any process group or process.
```

```
...
```

Organisation du manuel(3)

1. Commandes utilisateur
2. Appels système
3. Sous-routines
4. Fichiers spéciaux et périphériques
5. Formats de fichiers
6. Jeux
7. Divers
8. Commandes d'administration

En cas de doute... `man man`

Les fichiers

Philosophie

- ▶ Sous UNIX, **tout est fichier**.
- ▶ Il existe différents types de fichiers :
 - ▶ fichiers réguliers (ordinaires) ;
 - ▶ répertoires ;
 - ▶ fichiers spéciaux.
- ▶ Les fichiers prennent place dans une hiérarchie.
- ▶ Au niveau interne (noyau), les fichiers ont tous la même structure.

Les noms de fichiers

Format des noms de fichiers :

- ▶ 255 caractères maximum ;
- ▶ sensibles à la casse des lettres ;
- ▶ caractères spéciaux déconseillés, sauf '.', '_', '-'.

```
bash$ ls -a
.      essai      essai.c      essai.java
..     .essai     essai.fr.java mon_essai_a_moi
```

Nommage des fichiers

L'arborescence a une origine unique : / , les différents répertoires d'un chemin sont séparés par le caractère / .

Chemin absolu (décrit à partir de la racine) :

```
bash$ pwd
/home/ens/hodique
bash$ cd examens
bash$ pwd
/home/ens/hodique/examens
bash$
```

Chemin relatif (par rapport au répertoire courant) :

```
bash$ cd ../algo
bash$ pwd
/home/ens/hodique/algo
bash$
```

.. est toujours le répertoire père, et . le répertoire courant.

Manipulation des fichiers

- ▶ `cd` change le répertoire courant :

```
cd [repertoire]
```

Si aucun répertoire n'est précisé, le **home directory** devient le répertoire courant, sinon le répertoire indiqué devient le répertoire courant.

- ▶ `ls` liste le **contenu** d'un répertoire :

```
ls [options] [repertoires]
```

Exemple d'option : `-l` (affichage long)

Manipulation des fichiers(2)

- ▶ **mkdir** **création** d'un répertoire :

```
mkdir [options] <rep1> ... <repN>
```

chacun des noms peut être un nom relatif (création dans le répertoire courant) ou un nom absolu.

exemple d'option : **-p** vérifie chacun des répertoires figurant dans les chemins donnés et le crée si besoin.

- ▶ **rmdir** **efface** un répertoire vide :

```
rm dir [options] <rep1> ... <repN>
```

Manipulation des fichiers(3)

▶ **cp copie** de fichiers

```
cp [options] <src> <dest>
cp [options] <src> <rep>
```

exemple d'option : **-r** (copie récursive de répertoire)

▶ **mv déplacement** de fichiers

```
mv [options] <src> <dest>
mv [options] <src1> ... <srcN> <rep>
```

Dans le premier cas, le fichier « source » devient le fichier « destination », dans le deuxième cas, les fichiers « source » sont déplacés vers le répertoire donné en dernier argument.

exemple d'option : **-u** (ne pas écraser les fichiers ayant une date de modification identique ou plus récente).

Manipulation des fichiers(4)

- ▶ **rm destruction** de fichiers :

```
rm [options] src1 ... srcN
```

exemple d'option : **-i** (demander confirmation avant d'effacer chaque fichier).

```
bash$ rm -iessai.txt
remove regular file `essai.txt'? y
bash$
```

Attention, il ne s'agit pas d'une mise à la « corbeille » !

- ▶ **cat** , **more** , **less** , ...

Les méta-caractères

Utilisation des méta-caractères pour générer des noms de fichiers. Le shell remplace le motif donné par tous les noms de fichiers correspondant à ce motif (la liste est triée).

- ▶ `?` représente un caractère.
- ▶ `[xyz]` représente un caractère parmi ceux entre les crochets.
- ▶ `[x-z]` représente un caractère dans l'intervalle donné.
- ▶ `*` représente un nombre quelconque de caractères (y compris 0).

Rq : les méta-caractères ne remplacent pas le «.» en premier caractère.

Exemples

```
bash$ ls -a
./  ../  .essai.txt  essai.txt  essai1.txt  essai2.txt
bash$ ls *
essai.txt  essai1.txt  essai2.txt
bash$ ls essai?. *
essai1.txt  essai2.txt
bash$ ls essai? *
essai.txt  essai1.txt  essai2.txt
bash$ ls essai[12345]. *
essai1.txt  essai2.txt
bash$ ls essai[1-5] *
essai1.txt  essai2.txt
bash$ ls *.essai. *
ls: No match.
```

Type et droits du fichier

Chaque fichier a un **propriétaire**, celui-ci peut définir les **droits** qu'il donne sur ce fichier. Type et droits sont codés sur 2 octets

				u	g	s	r	w	x	r	w	x	r	w	x
type				spécial			propriétaire			groupe			autres		

- ▶ Le type est codé sur 4 bits (répertoire, fichier bloc, fichier caractères, fichier normal, . . .)
- ▶ Les bits **u**, **g**, **s** servent à spécifier des droits spéciaux.
- ▶ Les 9 bits de droits indiquent les droits de lecture (**r**), écriture (**w**) et exécution (**x**) du fichier pour les propriétaire, membres du groupe et autres.

Utilisation des droits

Les droits sont :

- ▶ **r**, le fichier peut être **lu** ;
- ▶ **w**, le fichier peut être **écrit** (ou **effacé**) ;
- ▶ **x**, le fichier peut être **exécuté**. Dans le cas d'un répertoire, il peut être traversé.

Pour décider de l'accès à un fichier :

- ▶ si l'**uid** de l'utilisateur est égal à l'**uid** du propriétaire du fichier, alors les droits sont donnés par les trois premiers bits ;
- ▶ sinon, si l'un des **gid** de l'utilisateur est égal au **gid** du fichier, alors les droits sont donnés par les trois bits suivants ;
- ▶ sinon, les droits sont donnés par les trois derniers bits.

Exemple

```
-rw-r--r-- 1 hodique  ens  33004  Dec  15  13:24  .emacs
-rw----- 1 hodique  ens    318  Sep   2  18:27  .emacs-cust
drwxr-xr-x 1 hodique  ens     13  Sep   3  13:13  .emacs.d
-rw-r--r-- 1 hodique  ens  60369  Nov  10  15:24  .emacs.htm
-rw----- 1 hodique  ens   1027  Sep   2  18:27  .emacs-opt
-rw----- 1 hodique  ens    473  Sep   2  18:27  .emacs-vars
-rw----- 1 hodique  ens  31758  Nov   6   09:49  .gnus.el
drwx----- 2 hodique  ens   4096  Dec  10  16:03  .ssh
```

Modification des droits

La commande

```
chmod <mode> <file>
```

permet au propriétaire du fichier de modifier ses droits d'accès.

Le nouveau mode peut être exprimé sous forme octale, ou sous forme symbolique (voir [man chmod](#))

```
chmod 0754essai.txt
```

u	g	s	r	w	x	r	w	x	r	w	x
0	0	0	1	1	1	1	0	1	1	0	0
0			7			5			4		

Modification des droits(2)

La modification à effectuer sur le mode courant peut être spécifiée de manière symbolique par un code dont la syntaxe est :

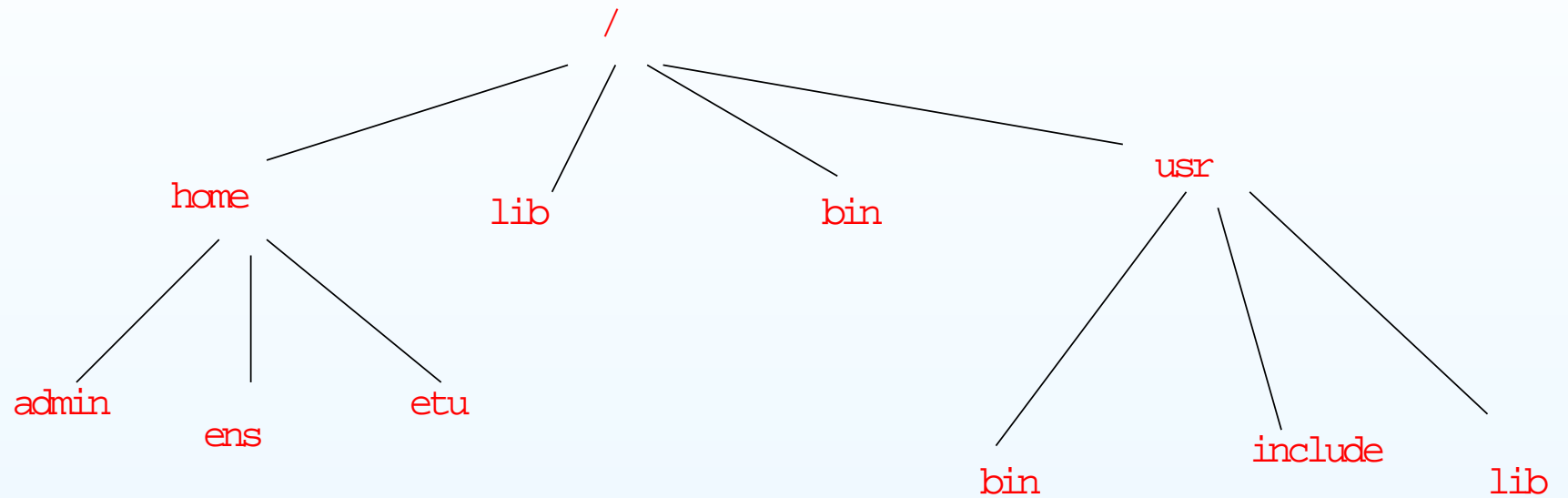
<personne><action><accès>

<personne>		<action>		<accès>	
u	propriétaire	+	ajouter	r	lecture
g	groupe	-	enlever	w	écriture
o	autres	=	initialiser	x	exécution
a	tous				

Hiérarchie Standard

<code>/bin</code>	Commandes utilisateur de base
<code>/dev</code>	Périphériques
<code>/etc</code>	Fichiers de configuration
<code>/home</code>	Répertoires des utilisateurs
<code>/lib</code>	Bibliothèques partagées
<code>/sbin</code>	Commandes d'administration
<code>/tmp</code>	Fichiers temporaires
<code>/usr</code>	Seconde hiérarchie
<code>/var</code>	Données variables

Hiérarchie standard – Exemples



Les processus

Les processus

- ▶ Un processus est un **objet dynamique** qui correspond à l'exécution d'un programme.
- ▶ Il est constitué du programme lui-même (suite d'instructions), des données que le programme manipule et du contexte d'exécution : l'ensemble des informations dont le système a besoin (état d'avancement, pile d'exécution, système d'entrées/sorties, ...).
- ▶ Toute activité (utilisateur ou système) est exécutée par un processus. L'**ordonnanceur** (scheduler) est donc une partie vitale du système.

Les processus (2)

Un processus possède un certain nombre de caractéristiques :

- ▶ son identification (**pid**) ;
- ▶ l'identification de son processus parent ;
- ▶ son propriétaire ;
- ▶ son groupe propriétaire ;
- ▶ éventuellement son groupe d'attachement ;
- ▶ des attributs (priorité, répertoire de travail, . . .)

La commande `ps`

La commande `ps` permet d'obtenir la liste des processus appartenant à un ensemble précisé par les options et certaines de leurs caractéristiques.

```
bash$ ps
  PID  TTY          TIME CMD
 24893 pts/1        00:00:00 bash
 24973 pts/1        00:00:01 emacs
 25008 pts/1        00:00:00 gkrellm
 25062 pts/1        00:00:00 ps
```

La commande `ps` (2)

```
bash$ ps aux
root          1  0.0  0.0 1488    80 ?        S    2003   0:06  init
root          2  0.0  0.0     0     0 ?        SW   2003   0:00  [kevent
root          0  0.0  0.0     0     0 ?        SWN  2003   0:00  [ksofti
root          0  0.0  0.0     0     0 ?        SW   2003   0:06  [kswapd
...
xfs           1258  0.0  0.4  6620   2368 ?        S    2003   0:00  [xfs]
daemon        1366  0.0  0.0  1516    104 ?        S    2003   0:00  [atd]
...
hodique       8665  0.0  0.4  3476   2348 pts/6    S    12:37   0:00  bash
hodique       8719  0.0  2.1  6952  10976 ?        S    12:37   0:01  qnet
hodique       8952  0.0  0.3  3404   1916 pts/4    S    12:39   0:01  ssh bal
hodique       8475  0.0  0.1  2324    768 pts/1    R    13:12   0:00  ps aux
```

La commande `kill`

La commande `kill` envoie un signal à un processus :

```
bash$ kill 355
```

envoie un **signal de terminaison** (`TERM` , signal par défaut) au processus de **pid** 355.

Le signal à envoyer peut être précisé par son numéro ou son nom :

```
bash$ kill -9 355
```

OU

```
bash$ kill -KILL 355
```

envoie un signal de terminaison immédiate (`KILL`) au processus de **pid** 355.

Les terminaux

Les terminaux permettent l'interaction entre les utilisateurs et les applications. Ils remplissent les fonctions de :

- ▶ **fichier** sur lequel il est possible de lire ou d'écrire ;
- ▶ **contrôle** des processus qui dépendent d'un terminal par certains caractères spéciaux.

Ils peuvent être :

- ▶ des terminaux physiques connectés à des ports ;
- ▶ des pseudo-terminaux (fenêtre X, ...).

Dans tous les cas, ils sont associés à un fichier spécial de `/dev` :

- ▶ la commande `tty` permet de connaître le nom du fichier associé à un terminal ;
- ▶ le fichier associé à un terminal peut être utilisé directement (ouverture, redirections, ...).

Lancer une tâche

- ▶ Une tâche peut être lancée en mode **foreground** : le shell est en attente tant que la commande n'est pas terminée.

```
bash$ tar -zxf essai.tgz
bash$ ls essai
essai.txt      essai      essai2
bash$
```

- ▶ Une tâche peut être lancée en mode **background** : le shell n'attend pas la terminaison de la commande, les processus sont exécutés en parallèle.

```
bash$ tar -zxf essai.tgz &
[1] 1179
bash$ ls essai
essai.txt      essai
[1]+  Done                  tar -zxf essai.tgz
```

Entrées/Sorties

tous les processus gèrent une table stockant le nom des différents fichiers qu'ils utilisent. Chaque index de cette est appelé **descripteur de fichiers**.

Par convention, les trois premiers descripteurs correspondent à :

- 0 l'**entrée standard** : si le programme exécuté par le processus a besoin de demander des informations à l'utilisateur, il les lira dans ce fichier (par défaut c'est le terminal en mode lecture) ;
- 1 la **sortie standard** : si le programme a besoin de donner des informations à l'utilisateur il les écrira dans ce fichier (par défaut c'est le terminal en mode écriture) ;
- 2 la **sortie d'erreur** : si le programme a besoin d'envoyer un message d'erreur à l'utilisateur, il l'écrira dans ce fichier (par défaut c'est le terminal en mode écriture).

Redirections

En shell, il est possible de modifier les fichiers identifiés par les descripteurs.

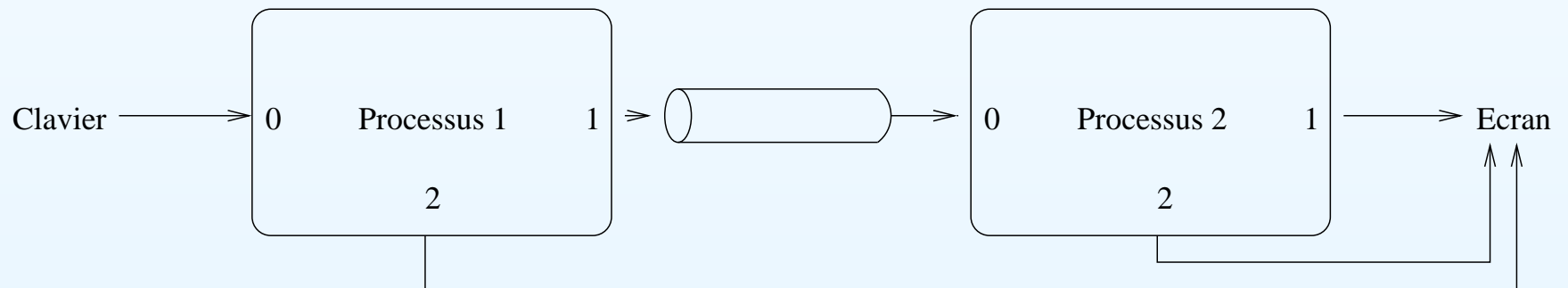
- ▶ Redirection de la sortie standard avec `>` et `>>` :
 - ▶ `commande > fichier` La sortie de la commande est placée dans le fichier. si le fichier n'existait pas, il est créé par le shell, si il existait son précédent contenu est détruit.
 - ▶ `commande >> fichier` La sortie de la commande est placée dans le fichier. si le fichier n'existait pas, il est créé par le shell, si il existait déjà la sortie de la commande est ajoutée à la fin.
- ▶ Redirection de l'entrée standard avec `<` et `<<` :
 - ▶ `commande < fichier` La commande lit ses données dans le fichier.

Communication inter-processus

Il est possible d'avoir plusieurs processus fonctionnant en parallèle qui communiquent entre eux par le biais de **tubes** (pipes). Le système assure alors la synchronisation de l'ensemble des processus ainsi lancés.

Le principe est assez simple :

La sortie standard d'un processus est redirigée vers l'entrée d'un **tube** dont la sortie est dirigée vers l'entrée standard d'un autre processus.



Communication inter-processus (2)

Le lancement concurrent de processus communiquant deux par deux par l'intermédiaire des tubes est réalisé par une commande de la forme :

```
commande1 | commande2 | ... | commandeN
```

Ce mécanisme est la force principale d'UNIX :

un ensemble de petits programme **fiables** qui communiquent entre eux via le système d'exploitation.

Processus et terminaux

- ▶ Chaque commande analysée par le shell correspond à une tâche.
- ▶ Les tâches (ou **jobs**) d'un terminal peuvent être visualisées par la commande **jobs** :

```
bash$ jobs
[1]+  Running                  xeyes &
bash$ jobs -l
[1]+  3655  Running                  xeyes &
```

- ▶ Une tâche peut être désignée par :
 - ▶ son numéro de job précédé du caractère **%** ;
 - ▶ **%%** ou **%+** pour la tâche dite courante (désignée par + dans la liste des jobs) ;
 - ▶ **%-** pour la tâche courante précédente (désignée par - dans la liste des jobs) ;
 - ▶ ...

Job Control

- ▶ Une tâche peut être composée de plusieurs processus (par exemple s'il s'agit de processus communiquant par tubes (|)) et peut être lancée en avant plan ou en arrière plan.
- ▶ Une tâche peut être suspendue par l'envoi d'un signal **STOP**, utilisation de **Ctrl-z** en mode interactif.
- ▶ Une tâche suspendue, par exemple de numéro de job 2, peut être réactivée en mode interactif par **fg %2** (en premier plan) ou **bg %2** (en arrière plan). L'utilisation de **fg** ou **bg** sans argument concerne le dernier job suspendu.
- ▶ La commande **kill** permet d'envoyer un signal à une tâche :

```
bash$ xcalc &
[2] 3688
bash$ kill %2
bash$ jobs -l
[1]- 3655 Running          xeyes &
[2]+ 3688 Terminated      xcalc
```

Exemple

```
bash$ more /etc/services | grep tcp
tcpmux          1/tcp # TCP port service multiplexer

[2]+  Stopped                  more /etc/services | grep tcp
bash$ jobs -l
[1]-  3655  Running                    xeyes &
[2]+  6796  Stopped                    more /etc/services
      6797                          | grep tcp

bash$ fg %2
more /etc/services | grep tcp
tcpmux          1/udp # TCP port service multiplexer
rje             5/tcp # Remote Job Entry
echo            7/tcp
...
```

Introduction au shell

Interpréteur de commandes

Le shell est l'interpréteur de commandes du système UNIX, interface entre l'utilisateur et le système. Le shell lit des lignes de commandes et les interprète.

- ▶ Une commande est composée d'une suite de mots séparés par des espaces, le premier mot est le nom de la commande à exécuter, les suivants sont les arguments de la commande.
- ▶ Quand une commande doit être exécutée, l'interpréteur crée un nouveau processus, lui confie cette exécution et attend que le processus se termine.
- ▶ Une commande peut être exécutée en arrière plan (*i.e.* l'interpréteur n'attend pas la terminaison du processus) en ajoutant l'opérateur & à la fin de la commande.
& est un opérateur reconnu par l'interpréteur de commandes, il n'est **PAS** passé en argument à la commande.

Entrées/Sorties

Le système associe à chaque processus une table de descripteurs de fichiers :

- ▶ le descripteur 0 désigne l'entrée standard (par défaut le clavier) ;
- ▶ le descripteur 1 désigne la sortie standard (par défaut le terminal).

Il est possible de modifier, le temps de l'exécution d'une commande, les entrées et sorties standards par des opérateurs reconnus par l'interpréteur :

- ▶ `commande > nom_fichier` place la sortie de la commande dans le fichier `nom_fichier` (si le fichier n'existe pas, il est créé, s'il existe, il est écrasé) ;
- ▶ `commande >> nom_fichier` permet d'ajouter la sortie de la commande à la fin du fichier ;
- ▶ `commande < nom_fichier` prend comme entrée standard de la commande le fichier spécifié.

Exemples

```
bash$ ls > toto
bash$ ls
essai.txt      fichier.txt    toto
bash$ more toto
essai.txt
fichier.txt
toto
bash$ ls >> toto
bash$ more toto
essai.txt
fichier.txt
toto
essai.txt
fichier.txt
toto
bash$ wc < toto
6          6          54
```

Les tubes

Un tube permet de relier la sortie standard d'une commande à l'entrée standard d'une autre (symbole `|`). Exemple :

```
ls -l | wc
```

- ▶ les deux commandes sont reliées par un tube du système d'exploitation ;
- ▶ les deux commandes forment un pipeline ;
- ▶ la synchronisation est réalisée par le système lui-même, `ls -l` est arrêté quand le tube est plein, `wc` est arrêté quand le tube est vide.

Les filtres

On appelle filtre un programme qui lit des données sur l'entrée standard, les transforme puis envoie le résultat sur la sortie standard. Par exemple `grep` affiche les lignes de l'entrée contenant un motif donné.

Exemple. Filtres et tubes.

```
bash$ ls
essai.c      essai.txt    fichier.txt   sauv_essai.txt  toto
bash$ ls -l | grep essai
-rw-r--r--   1 hodique    ens    4 Oct 31 11:51  essai.c
-rw-r--r--   1 hodique    ens    4 Oct 31 11:36  essai.txt
-rw-r--r--   1 hodique    ens    4 Oct 31 11:51  sauv_essai.txt
bash$ ls -l | grep essai | wc -l
3
```

Les méta-caractères

Utilisation des méta-caractères pour générer des noms de fichiers. Le shell remplace le motif donné par tous les noms de fichiers correspondant à ce motif (la liste est triée).

- ▶ `?` représente un caractère.
- ▶ `[xyz]` représente un caractère parmi ceux entre les crochets.
- ▶ `[x-z]` représente un caractère dans l'intervalle donné.
- ▶ `*` représente un nombre quelconque de caractères (y compris 0).

Rq : les méta-caractères ne remplacent pas le «.» en premier caractère.

Exemples

```
bash$ ls -a
./  ../  .essai.txt  essai.txt  essai1.txt  essai2.txt
bash$ ls *
essai.txt  essai1.txt  essai2.txt
bash$ ls essai?. *
essai1.txt  essai2.txt
bash$ ls essai? *
essai.txt  essai1.txt  essai2.txt
bash$ ls essai[12345]. *
essai1.txt  essai2.txt
bash$ ls essai[1-5] *
essai1.txt  essai2.txt
bash$ ls *.essai. *
ls: No match.
```

Exemples

Attention : dans les deux cas, tous les fichiers sont perdus !!!

```
bash$ ls
essai.c      essai.txt      fichier.txt      sauv_essai.txt      toto
bash$ rm essai *
bash$ ls
fichier.txt      sauv_essai.txt      toto
bash$ rm essai *
rm cannot remove 'essai' :No such file or directory
bash$ ls
bash$
```

```
bash$ ls
essai.c      essai.o      titi      toto
bash$ rm */o
o: Command not found.
bash$ ls
bash$
```

Neutralisation des méta-caractères

Les méta-caractères ont une signification particulière pour l'interpréteur (<, >, |, *, ?, &). Pour utiliser ces caractères sans qu'ils soient interprétés par l'interpréteur, il faut les neutraliser :

- ▶ **par un caractère **
- ▶ *par des quotes (apostrophes)*
- ▶ *par des guillemets (voir plus loin)*

```
bash$ echo *
essai toto
bash$ echo \*
*
```

Cas particulier : `\<newline>` désigne une continuation de ligne (permet d'écrire une commande sur plusieurs lignes).

Neutralisation des méta-caractères

Les méta-caractères ont une signification particulière pour l'interpréteur (<, >, |, *, ?, &). Pour utiliser ces caractères sans qu'ils soient interprétés par l'interpréteur, il faut les neutraliser :

- ▶ *par un caractère *
- ▶ **par des quotes (apostrophes)**
- ▶ *par des guillemets (voir plus loin)*

```
bash$ echo '? *a? *'  
? *a? *
```

En aucun cas, une ' ne peut apparaitre entre deux quotes ;

Neutralisation des méta-caractères

Les méta-caractères ont une signification particulière pour l'interpréteur (<, >, |, *, ?, &). Pour utiliser ces caractères sans qu'ils soient interprétés par l'interpréteur, il faut les neutraliser :

- ▶ *par un caractère *
- ▶ *par des quotes (apostrophes)*
- ▶ **par des guillemets (voir plus loin)**

```
bash$ echo "*a\"?"  
*a"?
```

Différents shell

- ▶ `sh` . Bourne shell : le premier shell, pas très convivial.
- ▶ `ksh` . Korn shell : amélioration de sh.
- ▶ `bash` . GNU Bourne-Again Shell. Nouvelle amélioration de sh.
- ▶ `csh` . C-shell, syntaxe plus proche de celle du C, plus convivial pour les utilisateurs.
- ▶ `tcsh` . Turbo C-shell, amélioration de csh.

Par défaut, le reste du cours concerne le `bash` .

Langage de programmation

L'interpréteur exécute des lignes de commandes qui peuvent provenir du terminal (mode interactif) ou d'un fichier. L'utilisateur peut donc définir ses propres commandes.

- ▶ **Script.** Un script est un fichier qui contient une suite d'instructions à exécuter.
- ▶ **Attention.** Pour pouvoir être exécuté, le script doit être un fichier exécutable!!!
- ▶ **Exemple.**

essai.sh

```
#!/bin/bash  
echo "aaa"
```

exécution

```
bash$ chmod 755 essai  
bash$ ./essai  
aaa
```

Les variables

Deux sortes de variables existent en shell :

- ▶ les variables locales, connues par le shell qui les a créées ;
- ▶ les variables d'environnement connues par tous les sous-shells et les commandes que le shell va lancer.

Nom de variable. Commence par une lettre et se compose de lettres, de chiffres et du caractère souligné.

Type d'une variable. Les variables sont des chaînes de caractères.

Déclaration d'une variable. Les variables ne se déclarent pas, elles sont considérées comme déclarées à leur première utilisation.

Variables prédéfinies

Voici quelques exemples de variables prédéfinies :

- ▶ **HOME**, répertoire de l'utilisateur ;
- ▶ **USER**, nom de l'utilisateur ;
- ▶ **TERM**, type de fenêtre pour le shell ;
- ▶ **PATH**, liste des répertoires où chercher les commandes à exécuter ;
- ▶ **DISPLAY**, écran sur lequel les affichages doivent être réalisés ;
- ▶ **HOSTNAME**, nom de la machine ;
- ▶ **PWD**, répertoire courant ;
- ▶ ...

Les variables définies dans les fichiers `.login` et `.profile` sont connues de toutes les applications de la session.

Les fichiers `.bashrc` et `.cshrc` contiennent respectivement les paramètres d'initialisation de `bash` et `csh` .

Manipulation des variables

Opération	bash	csH
Variables locales		
Affectation	<code>variable=valeur</code>	<code>set variable=valeur</code>
Liste des variables	<code>set</code>	<code>set</code>
Supression	<code>unset variable</code>	<code>unset variable</code>
Variables d'environnement		
Affectation	<code>variable=valeur</code> <code>export variable</code>	<code>setenv variable valeur</code>
Liste des variables	<code>printenv</code>	<code>printenv</code> ou <code>setenv</code>
Suppression	<code>unset variable</code>	<code>unsetenv variable</code>

Manipulation de variables (2)

- ▶ **Attention !** Lors des affectations, il ne doit pas y avoir d'espaces de part et d'autre du signe `=`. Exemple : `nom=toto` .
- ▶ Pour affecter la chaîne vide à une variable, on utilise un caractère espace comme valeur à affecter. Exemple : `toto=` .
- ▶ La valeur d'une variable est obtenue en utilisant le nom de la variable (entre accolades ou non) précédé du signe `$` (`$nom` ou encore `${nom}`).

Exemples.

```
bash$ nom=toto
bash$ echo $nom
toto
bash$ echo ${nom}to
tototo
bash$
```

Variables spéciales

- ▶ `$$` numéro de processus du shell en cours d'exécution.
- ▶ `$!`, numéro de processus du dernier processus exécuté en arrière plan.
- ▶ `$-`, options du shell en cours d'exécution.
- ▶ `$?`, code de retour de la dernière commande exécutée (chaîne de caractères décimaux).
- ▶ `$_` dernier argument de la commande précédente.
- ▶ `$0`, le nom de la commande en cours d'exécution.
- ▶ `$#`, nombre d'arguments du script.
- ▶ `$*`, liste des arguments du script. Entre guillemets, donne la liste des arguments en une seule chaîne.
- ▶ `$@`, liste des arguments du script. Entre guillemets, donne la liste des arguments en tant que mots séparés.
- ▶ `$1 ... $9`, 9 premiers arguments du script.

Exemples

essai	exécution
<pre>#!/bin/bash echo "nom = \$0" echo "nb args = \$#"</pre>	<pre>bash\$ essai toto 5 titi nom = ./essai nb args = 3 args = toto 5 titi</pre>
essai2	exécution
<pre>#!/bin/bash echo \$2 \$1 echo "aaa \" \a \ * !!!"</pre>	<pre>bash\$ essai2 toto titi titi toto aaa " \a \ * !!!</pre>

Arguments

Les 9 premiers arguments sont accessibles par les variables spéciales

`$1 ... $9`

- Pour accéder aux suivants, il faut effectuer un décalage.

`shift [n]`

permet de décaler de n vers la gauche les arguments.

L'argument $n+1$ devient `$1`, l'argument $n+2$ devient `$2` et ainsi de suite. Les anciens arguments de 1 à n sont perdus, les anciens arguments de `$# - n + 1` à `$#` sont libérés.

Par défaut, la valeur du décalage est 1.

```
#!/bin/bash
echo $*
shift 2
echo $*
```

```
bash$  essai  1 2 3 4 5
1 2 3 4 5
3 4 5
```

Arguments (2)

- ▶ Les arguments ne peuvent pas être affectés comme des variables (notez que les noms des arguments ne sont pas des noms de variables). Ils peuvent être affectés globalement par la commande `set` .

```
#!/bin/bash
para=$ *
shift
echo $*
set $para
echo $*
```

```
bash$  essai  1 2 3 4 5
2 3 4 5
1 2 3 4 5
```

Chaînes

- ▶ Une chaîne délimitée par des apostrophes (ou quotes) est considérée telle quelle (*i.e.* les caractères spéciaux qu'elle contient perdent leur signification (à l'exception de `\<newline >`)).
- ▶ Les guillemets peuvent également servir à délimiter une chaîne. Les caractères spéciaux perdent leur signification sauf :
 - ▶ \$ pour l'évaluation d'un paramètre ;
 - ▶ ' pour le remplacement de commande ;
 - ▶ " pour fin de chaîne ;
 - ▶ \ pour neutraliser l'un de précédent caractères ;
 - ▶ il n'y a pas d'interprétation des espaces ni de génération de noms de fichiers.

Remarque.

`echo "$ * "` est équivalent à

`echo "$1 $2 $3..."`

`echo "$@"` est équivalent à

`echo "$1" "$2" "$3"..."`

Remplacement de commande

La sortie standard d'une commande peut être utilisée comme argument d'une commande :

- ▶ une chaîne placée entre `backquotes` est considérée comme une commande à exécuter. La commande est exécutée et est remplacée par son résultat ;
- ▶ les règles de neutralisation habituelles s'appliquent.

Exemples.

`ls `echo "$1"`` est équivalent à `ls $1`.

`for i in `ls *.c` ; do ...` La variable `i` prend successivement les valeurs des fichiers d'extension `c` du répertoire courant.

Expressions

Le shell manipule des chaînes de caractères. Pour évaluer des expressions arithmétiques, il faut faire appel à une commande particulière :

`expr`

- ▶ évalue une expression (résultat sur la sortie standard) ;
- ▶ les opérandes sont considérés comme des entiers ou des chaînes en fonction de l'opérateur à appliquer.

```
#!/bin/bash
echo `expr 2 \| 3`
echo `expr 2 + 3`
```

```
bash$  essai
2
5
```

Expressions (2)

Les opérateurs disponibles :

▶ opérateurs logiques :

▶ | retourne le premier argument s'il est non nul, le deuxième sinon,

▶ & retourne le premier argument si aucun argument n'est nul, 0 sinon,

▶ < <= = == != >= > retourne 1 si la relation est vraie, 0 sinon. Essaie de réaliser une comparaison d'entiers puis une comparaison de chaînes si ce n'est pas possible ;

▶ opérations sur les entiers :

▶ + - * / % avec les sens habituel ;

▶ opérations sur les chaînes :

▶ : match substr index length

Test

La commande suivante permet de tester une expression :

```
test expression
```

Voici quelques exemples d'expressions, pour plus de détails, voir le man.

▶ Sur les fichiers :

- ▶ `-e nom` , retourne 0 si la référence `nom` existe ;
- ▶ `-d nom` , retourne 0 si `nom` existe et est un répertoire ;
- ▶ `-f nom` , retourne 0 si `nom` existe et est un fichier régulier ;
- ▶ `-s nom` , retourne 0 si `nom` existe et n'est pas vide ;
- ▶ `nom1 -nt nom2` , retourne 0 si `nom1` est plus récent que `nom2` ;
- ▶ ...

Test (2)

- ▶ Sur les chaînes :
 - ▶ `-z string` retourne 0 si la longueur de la chaîne est 0 ;
 - ▶ `-n string` retourne 0 si la longueur de la chaîne n'est pas 0 ;
 - ▶ `string1 = string2` retourne 0 si les deux chaînes sont égales ;
 - ▶ `string1 != string2` retourne 0 si les deux chaînes ne sont pas égales.
- ▶ Des tests booléens :
 - ▶ `!expr` retourne 0 si `expr` est fausse ;
 - ▶ `expr1 -a expr2` retourne 0 si les deux expressions retournent 0 ;
 - ▶ `expr1 -o expr2` retourne 0 si l'une des deux expressions retourne 0.

Test (3)

- ▶ Des expressions arithmétiques :

`arg1 OP arg2`

- ▶ `OP` vaut `-eq` , `-ne` , `-lt` , `-le` , `-gt` , ou `-ge` . Le résultat est 0 si `arg1` est respectivement égal, différent, strictement inférieur, inférieur ou égal, strictement supérieur, ou supérieur ou égal à `arg2` .
- ▶ `arg1` et `arg2` sont des entiers (positifs ou négatifs) ou une expression du type `-l string` qui évalue la longueur d'une chaîne.

Structure if

```
if liste_de_commandes
    then liste_de_commandes
    else liste_de_commandes
fi
```

- ▶ La commande teste l'état de sortie de la dernière commande de la liste de commandes qui suit le **if**. Si le retour est 0, le test est considéré comme vrai.
- ▶ La partie **else** est facultative.
- ▶ Les **if** peuvent être imbriqués sans ambiguïté.

Exemple.

```
# !/bin/bash
if test -n "$*"
    then echo "liste d'arguments non vide"
    else echo "liste d'arguments vide"
fi
```

Structure **if** (2)

!Simplification de syntaxe :

```
if liste_de_commandes  
    then liste_de_commandes  
    else if liste_de_commandes  
        then liste_de_commandes  
    fi  
fi
```

peut s'écrire :

```
if liste_de_commandes  
    then liste_de_commandes  
    elif liste_de_commandes  
        then liste_de_commandes  
fi
```

Structure case

```
case mot in  
    cas_1) liste_de_commandes ;;  
    ...  
    cas_n) liste_de_commandes ;;  
esac
```

- ▶ Lorsqu'une équivalence est trouvée entre la chaîne *mot* et l'une des chaînes *cas*, les instructions correspondantes sont exécutées (jusqu'à ;;), la structure **case** est terminée.
- ▶ Les *cas* sont essayés suivant leur ordre de définition.
- ▶ Les règles usuelles s'appliquent pour les caractères spéciaux, par exemple le cas *** représente toute chaîne de caractères et peut être utilisé pour le cas par défaut.

Structure for

```
for variable in mot1 mot2 ...  
    do liste_de_commandes  
done
```

- ▶ Les mots réservés (comme **do** et **done**) doivent être précédés par un passage à la ligne ou un point virgule.
- ▶ La *variable* de la boucle prend successivement les valeurs *mot1 mot2 ...*
- ▶ Si la partie **in** *mot1 mot2 ...* n'est pas présente, la *variable* prend successivement les valeurs des arguments (équivalent à **in** \$*).

Exemple

```
#!/bin/bash
for i in `ls`
do
    case $i in
        *.c)      echo "$i --> fichier C";;
        *.cc)     echo "$i --> fichier C++";;
        *.ada)    echo "$i --> fichier ada";;
        *.o)      echo "$i --> fichier objet";;
        *.java)   echo "$i --> fichier Java";;
        *)        echo "$i --> extension non reconnue";;
    esac
done
```

Structure `while`

```
while liste_de_commandes_1  
    do liste_de_commandes_2  
done
```

- ▶ À chaque tour de boucle, la *liste_de_commandes_1* est exécutée :
 - ▶ si le résultat de la dernière commande de cette liste est 0, la *liste_de_commandes_2* est exécutée ;
 - ▶ sinon, la structure `while` est terminée.

Structure **until**

```
until liste_de_commandes_1  
    do liste_de_commandes_2  
done
```

- ▶ À chaque tour de boucle, la *liste_de_commandes_1* est exécutée :
 - ▶ si le résultat de la dernière commande de cette liste n'est pas 0, la *liste_de_commandes_2* est exécutée ;
 - ▶ sinon, la structure **until** est terminée.

Shell – Outils

Les commandes internes

- ▶ Les commandes internes sont des commandes qui sont réalisées de manière interne au shell (l'exécution des autres commandes est confiée à un nouveau processus).
- ▶ Les commandes internes de `bash` peuvent être :
 - ▶ héritées du Bourne Shell, par exemple `cd`, `eval`, `exit`, `export`, `pwd`, `shift`, `test`, ...
 - ▶ spécifiques à (ou étendue dans) `bash` , par exemple `echo`, `read`, `source` , ...

En tout état de cause, `bash` respecte la norme POSIX définissant le « standard » pour les shell Unix.

Regroupement de commandes

Il existe deux possibilités de regrouper les commandes :

- ▶ `{ commande1 ; commande2 ; ... ; commanden ; }` dans ce cas, la liste de commandes est simplement exécutée ;
- ▶ `(commande1 ; commande2 ; ... ; commanden)` dans ce cas, la liste de commande est exécutée par un nouveau processus.

Exemple.

```
[~/rep]$ { cd ../; ls; }
rep          toto1          toto2
[~/]$ cd rep
[~/rep]$ ( cd ../;ls )
rep          toto1          toto2
[~/rep]$
```

N.B. Les parenthèses sont des caractères spéciaux, les accolades sont des mots réservés.

Évaluation

La commande interne suivante permet de « forcer » l'évaluation des arguments.

```
eval [argument ...]
```

- ▶ Les arguments sont évalués et concaténés en une seule commande qui est exécutée par le shell.
- ▶ La valeur de retour de la commande est retournée comme valeur du `eval`. S'il n'y a pas d'argument ou uniquement des arguments vides, la valeur de retour est 0.

Exemple.

```
bash$ x=2
bash$ y=' $x'
bash$ echo $y
 $x
bash$ eval echo $y
2
```

Récurtivité ?

Puisqu'une commande peut être appelée dans un script, il est possible d'écrire des scripts « récursifs ».

```
#!/bin/bash
tmp=$1
while test $tmp -ne 0
do
    echo $tmp
    tmp=`expr $tmp - 1`
    $0 $tmp
done
echo "fin"
```

Exécution

```
bash$  essai  2
2
1
fin
fin
1
fin
fin
```

Plusieurs processus sont lancés (un par appel de la commande). Si la commande est suspendue on peut obtenir une liste du type :

```
bash$  ps
  PID  TTY          TIME CMD
  680  ttyp0      00:00:00 tcsh
  968  ttyp0      00:00:00 bash
 1480  ttyp0      00:00:00 essai
 1483  ttyp0      00:00:00 essai
 1490  ttyp0      00:00:00 essai
 1491  ttyp0      00:00:00 ps
```

Signaux

La commande interne

```
trap [ argument ] numero_signal
```

- ▶ permet à un processus de spécifier le traitement qu'il veut exécuter quand un signal particulier lui est adressé ;
- ▶ le *numero_signal* spécifie le (ou les) signal concerné par le traitement ;
- ▶ l'argument spécifie le traitement à adopter :
 - ▶ si *argument* vaut "" le signal est ignoré,
 - ▶ si *argument* vaut -, le comportement par défaut est réinstallé,
 - ▶ si *argument* est une commande ou une fonction, celle-ci sera exécutée à la réception du signal.

Exemples

```
#!/bin/bash
trap 'echo "je suis tue"; exit' 15
while test "" = ""; do echo "coucou"; done
```

```
bash$  essai  > toto &
[1] 11697
bash$  ps
  PID TTY          TIME CMD
 5307 ttyp2        00:00:00 bash
11697 ttyp2        00:00:01  essai
11698 ttyp2        00:00:00   ps
bash$  kill  -15 11697
bash$  cat  toto
coucou
coucou
je suis tue
[1]+  Done                  essai >toto
```

Les fonctions

Il est possible de définir des fonctions shell :

```
[function]  nom  ()  {  liste_de_commandes  ;  }
```

- ▶ La fonction est exécutée comme une commande normale dans le contexte du shell, il n'y a PAS de nouveau processus créé pour exécuter cette fonction.
- ▶ La fonction est appelée comme une commande.
- ▶ Lorsqu'une fonction est appelée, les paramètres de la fonctions deviennent les paramètres de position (uniquement durant l'exécution) à l'exception de \$0 qui demeure inchangé.
- ▶ Le statut de retour de la fonction est le retour de la dernière commande exécutée dans la fonction ou l'argument de la commande `return` si il est présent.

Exemples

```
bash$ more scriptTTTT
#!/bin/bash
TTTT=essai2
export TTTT
bash$ function toto () { echo "coucou"; }
bash$ toto
coucou
bash$
bash$ function defTOTO () { TOTO=essai; export TOTO; }
bash$ defTOTO
bash$ echo $TOTO
essai
bash$ scriptTTTT
bash$ echo $TTTT

bash$
```

Commande `cut`

La commande

```
cut [OPTION]... [FILE]...
```

- ▶ affiche sur la sortie standard certaines parties de chaque ligne de chaque fichier en entrée (ou l'entrée standard s'il n'y a pas de fichier donné ou pour le nom `-`).

Exemples d'options :

- ▶ l'option `-f` suivie d'un ou plusieurs nombres ou des intervalles séparés par des virgules permet de n'afficher que les champs correspondant. Par défaut, les champs sont séparés par des tabulations ;
- ▶ l'option `-d` suivie d'un délimiteur permet de choisir le premier caractère du délimiteur comme séparateur pour l'option `-f` .

Exemples

Afficher la liste des utilisateurs déclarés dans le fichier `/etc/passwd`

```
bash$ more /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
dupont:x:8305:83::/home/dupont:/bin/tcsh
toto:x:8306:83::/home/toto:/bin/tcsh
durant:x:8308:83::/home/durant:/bin/tcsh
bash$ cat /etc/passwd | cut -f 1 -d:
root
...
dupont
toto
durant
```

Exemples

Afficher la liste des utilisateurs et de leur répertoire :

```
bash$ cat /etc/passwd | cut -f 1,6 -d:  
root:/root  
...  
dupont:/home/dupont  
toto:/home/toto  
durant:/home/durant
```

Afficher la liste des champs 2 à 5 :

```
bash$ cat /etc/passwd | cut -f 2-5 -d:  
x:0:0:root  
...  
x:8305:83:  
x:8306:83:  
x:8308:83:
```

Commande `sed`

La commande `sed` est un *stream editor*. Elle permet d'éditer un flot de données : elle prend des lignes en entrée, leur applique une requête.

Syntaxe :

```
sed [-n] [[-e] requetes] [-f fichier-req] [fichiers]
```

- ▶ `-n` : n'afficher que les lignes dont l'affichage est explicitement demandé par une requête `p`.
- ▶ `-f fichier-req` : lire les requêtes dans le fichier `fichier-req` .
- ▶ `-e requetes` : traiter les requêtes de la chaîne `requetes` .

N.B. l'intérêt de cette commande est que seule la ligne courante est mémorisée, on peut donc traiter de très gros fichiers avec très peu de ressources.

sed présentation

Il y a équivalence entre :

```
bash$ sed 'requete1\  
> requete2\  
> requete3' fichier
```

```
bash$ sed -e 'requete1' -e 'requete2' -e 'requete3' fichier
```

```
bash$ cat toto  
requete1  
requete2  
requete3  
bash$ sed -f toto fichier
```

sed requêtes

Forme

[ligne1[,ligne2][!]] action-requete [arguments]

Pour spécifier quelles seront les lignes à traiter :

- ▶ si aucune ligne n'est spécifiée, tout le fichier est traité ;
- ▶ si une ligne est spécifiée, elle est traitée ;
- ▶ si deux lignes sont spécifiées, le bloc délimité par ces deux lignes sera traité ;
- ▶ si le ! est présent, l'ensemble complémentaire des lignes spécifiées sera traité.

sed requêtes (2)

Les lignes sont spécifiées par :

- ▶ `nombre` , le numéro de la ligne ;
- ▶ `$` la dernière ligne du fichier ;
- ▶ `/expr-reg/` la première ligne qui vérifie l'expression régulière ;
- ▶ `+nombre` adressage relatif.

Exemples.

`13` ligne `13`

`/^processus/ , +3` la première ligne qui commence par « processus » et les trois lignes suivantes.

`1,/fin/+2` de la ligne 1 jusqu'à deux lignes après la première occurrence de « fin ».

Requêtes de base

- ▶ `p` (print) écrit le contenu du tampon sur la sortie standard.
- ▶ `n` (next) idem puis remplace le contenu du tampon par la prochaine ligne d'entrée.
- ▶ `=` écrit le numéro de la ligne courante sur la sortie standard.

Exemples.

`sed 'p' toto` affiche le contenu de toto en doublant les lignes.

`sed -n 'p' toto` ou `sed " " toto` affichent tel quel le contenu du fichier.

`sed -n -e '=' -e 'p' toto` affiche le numéro de ligne avant chaque ligne. (Identique à `sed -e '=' toto`).

Autres requêtes

- ▶ `d` (delete) détruit le contenu du tampon.
- ▶ `q` (quit) termine l'exécution, `sed` lit l'entrée standard sans la traiter.
- ▶ `y/chaine1/chaine2/` les deux chaînes doivent être de même longueur, traduit chaque occurrence du *i*ème caractère de `chaine1` par le *i*ème caractère de `chaine2` .
- ▶ `s/expr-reg/chaine/[mod]` (substitute) recherche toute chaîne correspondant à l'expression régulière `expr-reg` et la remplace par la chaîne `chaine` . L'option `mod` peut être `p` (print) pour afficher le tampon s'il a subi une modification ou `g` (global) pour effectuer la substitution sur tout le tampon. Dans la chaîne, on peut rappeler la valeur correspondant à l'expression régulière avec `&`.

Pour le reste... `man sed` .

Exemples

```
bash$ cat /tmp/toto
aaa
bbb
ccc
bash$ sed -e 'd' /tmp/toto
bash$ sed -e '=' -e 'd' /tmp/toto
1
2
3
bash$ sed -e 'y/abc/xyz/' /tmp/toto
xxx
yyy
zzz
```

Exemples

```
bash$ cat /tmp/titi
tralala il fait beau tralala
tralala le soleil brille tralala
bash$ sed -e 's/tralala/youpi/' /tmp/titi
youpi il fait beau tralala
youpi le soleil brille tralala
bash$ sed -e 's/tralala/youpi/g' /tmp/titi
youpi il fait beau youpi
youpi le soleil brille youpi
```

Expressions–Régulières – Quoi ? Pourquoi ?

- ▶ Les expressions régulières définissent des « motifs » qui permettent de rechercher des chaînes dans un texte.
- ▶ Elles sont utilisées par de nombreuses commandes comme `sed` , `grep` , `find` , ...
- ▶ Deux types d'expressions régulières sont définies en POSIX, les « obsolètes » qui sont les anciennes expressions régulières et les expressions dites « étendues ».
- ▶ On dit qu'une chaîne « correspond » ou « matche » ou « est appariée à » une expression régulière.
- ▶ Attention : aux problèmes d'incompatibilités et de portage !

Exemple : afficher les lignes du fichier `/etc/services` qui commencent par un 't'.

```
egrep ^t /etc/services
```

Expression régulière – Définition

- ▶ Au plus haut niveau, une expression régulière (étendue) est une alternative (symbole |).
- ▶ Chaque opérande de l'alternative est une concaténation de « pièces ».

Exemples.

Les lignes commençant par 'f' ou 'g' :

```
egrep "^f|^g" /etc/services
```

Les lignes commençant par 'fo' ou 'gd' :

```
egrep "^fo|^gd" /etc/services
```

Pièce

- ▶ Une « pièce » est un atome (noté a ici) éventuellement suivi d'un symbole spécial :
 - ▶ a^* correspond à la répétition de 0 ou plusieurs fois a ;
 - ▶ a^+ correspond à la répétition de 1 ou plusieurs fois a ;
 - ▶ $a^?$ correspond à la répétition de 0 ou 1 fois a ;
 - ▶ $a\{n\}$ correspond à une séquence de n matches de a ;
 - ▶ $a\{n, \}$ correspond à une séquence d'au moins n matches de a ;
 - ▶ $a\{n, m\}$ avec $n \leq m$ correspond à une séquence k matches de a avec $n \leq k \leq m$;

Exemple

```
bash$ cat toto
aaa
abab
aaaaa
bbbbbb
bash$ egrep "(a|b){4}" toto
abab
aaaaa
bbbbbb
bash$cat toto
acaaacc
aa
ababaaaaa
aaaaaaaaaaa
bash$ egrep "a{3,5}" toto
acaaacc
ababaaaaa
aaaaaaaaaaa
```

Atome

- ▶ Une expression régulière entre parenthèses (l'expression `()` correspond à la chaîne vide).
- ▶ Un caractère spécial :
 - ▶ `'.'` qui représente n'importe quel caractère (sauf entre `[` et `]`);
 - ▶ `'^'` qui représente un début de ligne lorsqu'il est le premier caractère d'une expression ;
 - ▶ `'$'` qui représente une fin de ligne lorsqu'il est le dernier caractère d'une expression ;
 - ▶ `'\'` suivi d'un caractère quelconque représente ce caractère,
 - ▶ `\<` et `\>` correspondent respectivement au début et à la fin d'un mot. Un mot est une suite de caractères alpha-numériques et `'_'`.
- ▶ Un seul caractère sans signification spéciale.
- ▶ `'{'` non suivie d'un chiffre est considéré comme le caractère « accolade ouvrante ».
- ▶ une expression entre crochets.

Exemple

```
bash$ cat toto
.{aaaaaa}
.{aaaaaa}b
.{aabaaaa}
c.{aaaaaa}
bash$ egrep "(" toto
.{aaaaaa}
.{aaaaaa}b
.{aabaaaa}
c.{aaaaaa}
bash$ egrep ^.{a *}$ toto
.{aaaaaa}
```

Crochets

Une expression entre crochets correspond à un caractère de l'ensemble décrit. Elle peut être :

- ▶ $[c_1c_2 \dots c_n]$ correspond à un des caractères c_i ;
- ▶ $[\wedge c_1c_2 \dots c_n]$ correspond à un caractère du complémentaire de l'ensemble $\{c_1, \dots, c_n\}$;
- ▶ c_1-c_2 dans une suite de caractères décrivent tous les caractères compris entre c_1 et c_2 (inclus) ;
- ▶ Cas particuliers :
 - ▶ $']'$ dans une suite doit être placé en premier caractère,
 - ▶ $' -'$ dans une suite doit être placé en premier ou en dernier caractère.

Exemple

```
bash$ cat toto
bonjour il fait beau !
tralala la lere
aaa[bbb]ccc
bash$ egrep b[aeiouy] toto
bonjour il fait beau !
bash$ egrep l[^ea] toto
bonjour il fait beau !
bash$ egrep [b-d] toto
bonjour il fait beau !
aaa[bbb]ccc
bash$ egrep []\![] toto
bonjour il fait beau !
aaa[bbb]ccc
```

Exemple

```
bash$ cat toto
Bonjour, il fait beau
C'est un grand jour !
operation : 2+2=4 !
bash$ egrep "\<jour\>" toto
C'est un grand jour !
bash$ egrep "[-=]" toto
operation : 2+2=4 !
bash$ egrep "jour\>" toto
Bonjour, il fait beau
C'est un grand jour !
```

Classes de caractères

- ▶ `[: class :]` correspond (entre crochets) à un caractère de la classe de caractères ainsi désignée, `class` pouvant être :
 - ▶ `alnum` pour les caractères alphanumériques,
 - ▶ `digit` pour les chiffres décimaux,
 - ▶ `punct` pour les caractères de ponctuation,
 - ▶ `alpha` pour les lettres,
 - ▶ `graph` pour les caractères imprimables sauf espace,
 - ▶ `space` pour les caractères d'espacement,
 - ▶ `blank` pour espace ou tabulation,
 - ▶ `lower` pour les lettres minuscules,
 - ▶ `upper` pour les lettres majuscules,
 - ▶ `cntrl` pour les caractères de contrôle,
 - ▶ `print` pour les caractères imprimables,
 - ▶ `xdigit` pour les chiffres hexadécimaux ;

Exemple

```
bash$ cat toto
aaa 0xAF12 bbb
ABDFE'ZFFDFD
aCsDeFgBtHfD
afbv 12 fdlk 14
bash$ egrep 0x[[:xdigit:]] * toto
aaa 0xAF12 bbb
bash$ egrep "^[[:upper:]] *$" toto
ABDFE'ZFFDFD
bash$ egrep "^([:lower:][:upper:]) *$" toto
aCsDeFgBtHfD
```

Sous-expressions

```
bash$ cat toto
aaa toto azerrt
aaa djfhldksfh      aaa fkdldgjmfdl
sldkjfksfjk
toto titi titi toto
toto toto titi titi
bash$ egrep "(..)\1" toto
aaa toto azerrt
toto titi titi toto
toto toto titi titi
bash$ egrep "([[:alpha:]]{3,}).*\1" toto
aaa djfhldksfh      aaa fkdldgjmfdl
toto titi titi toto
toto toto titi titi
bash$ egrep "([[:alpha:]]{4})[[:space:]]*\1" toto
toto titi titi toto
toto toto titi titi
```