Programmation Unix 1 – cours n°2

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°2

1. Système de gestion de fichiers (SGF) Unix
2. Développement des noms de fichiers
3. Développement des accolades
4. Branchements multiples avec `case`
5. Boucle `for`
6. Arguments

1 - Système de gestion de fichiers (SGF) Unix

Structure hiérarchique (arbre) :

- Répertoire racine /
- Chaque répertoire peut contenir
  - des fichiers
  - des sous-répertoires
- Chemin :
  - absolu : `/home/thiel/ens/c2.txt`
  - relatif par rapport au répertoire courant : `../ens/c2.txt`
- Composition

.	le répertoire lui-même
..	le parent
//	= /

- Exemple : ceci est un chemin valide

  `/home/dupont/../thiel/./rech/..//ens/c2.txt`
- Pas de chemin canonique : composition, liens

Hiérarchie standard

`/etc`	Fichiers configuration système
`/etc/init.d`	scripts démarrage
`/bin` , `/sbin`	Commandes principales, du super-user
`/usr/bin`	Commandes secondaires
`/usr/sbin`	du super-user
`/usr/local/bin`	Applications
`/usr/local/sbin`	du super-user
`/lib`	Bibliothèques principales (équiv DLL)
`/usr/lib`	secondaires
`/var`	spoolers, mails, logs
`/home`	Répertoires principaux des utilisateurs
`/tmp`	Fichiers temporaires
`/dev`	Accès logique aux périphériques (devices)
`/proc`	Accès logique aux tables et infos du noyau

Droits sur fichiers et répertoires

- Afficher les droits :

```
$ ls -l ex1.sh
-rwxr-x--- 1 thiel prof 984 janv. 23 18:22 ex1.sh
```

(nature, droits, nb liens physiques, propriétaire, groupe, taille, date et heure modification, nom)

Caractère 1	`-` fichier régulier, `d` répertoire, `l` lien, etc
Caractères 2 à 10	`r`, `w`, `x` = droit accordé, `-` = droit absent.

- Trois niveaux de droits :

Caractère 2 à 4	user	le propriétaire du fichier
5 à 7	group	le groupe du fichier
8 à 10	others	les autres

Droits sur fichiers et répertoires (2)

- Droits sur les fichiers :

  r lire le fichier
  w modifier le fichier
  x exécuter le fichier

- Droits sur les répertoires :

  r lire le catalogue du répertoire → `ls`
  w modifier le catalogue du répertoire
  x traverser (*cross*) le répertoire → `cd`

Changer les droits avec `chmod`

- Usage :
  `chmod mode fichiers`

`mode` = chaîne de caractères donnant ou enlevant des droits

`$ chmod u=rwx,g=rx,o= f1.txt`	droits `rwxr-x---`
`$ chmod ugo=rw f1.txt`	droits `rw` pour tout le monde
`$ chmod ugo+x f1.txt`	rend le fichier exécutable

- Codage octal : `r` = 4, `w` = 2, `x` = 1
  `rwxr-x---` = (4+2+1)(4+1)(0) = 750 → `chmod 750 f1.txt`

- Droits par défaut pour fichiers : 666 - `umask`
  `$ umask 026` → Tous les fichiers créés avec droits 640

- Changer le propriétaire ou le groupe d'un fichier : `chown`, `chgrp`
  (il faut être administrateur)

2 - Développement des noms de fichiers

Motif = argument comportant des jokers

Jokers : `*` `?` `[...]`

Bash substitue chaque motif par la liste des fichiers

- correspondant au motif, et
- existant effectivement,

séparés par des blancs, dans l'ordre lexicographique.

Correspondances :

`*`	toute chaîne, y compris vide
`?`	1 caractère, n'importe lequel
$[c_1 c_2 \cdots c_n]$	1 caractère dans cet ensemble
$[c_1\text{-}c_2]$	1 caractère dans cet intervalle

Exemples

```
$ ls
bu2.c  bu2.h  ga.c  ga.h  ga.o meu1.c zo5.h

$ echo *.c
bu2.c ga.c meu1.c

$ echo *.[ch]
bu2.c bu2.h ga.c ga.h meu1.c zo5.h

$ echo ??[0-9].[ch]
bu2.c bu2.h zo5.h
```

Fonctionne aussi avec des noms de répertoires :

```
$ mv ../t[dp][1-5]/*.[ch] tmp
```

Caractères exclus

- `[^...]` ou `[!...]` correspond à 1 caractère $\notin$ `[...]`

```
$ ls
essai.txt prog1.c resume.txt
$ ls *[^0-9].*
essai.txt resume.txt
```

- Syntaxe :
  - On peut mélanger éléments et intervalles :
    `[A-Za-z0-9_. ]` `[^A-Za-z0-9_. ]`
  - Inclure ou exclure `']'` : le mettre en premier
    `[]abc]` `[^]abc]`
  - Inclure ou exclure `'-'` : le mettre en premier ou dernier
    `[-abc]` `[abc-]` `[^-abc]` `[^abc-]`
  - Inclure ou exclure `'^'` ou `'!'` : ne pas le mettre en premier
    `[abc^!]` `[^abc^!]`

Aucune correspondance

Si le motif ne correspond à aucun fichier ou répertoire existant, le motif n'est pas développé.

```
$ ls
essai.txt prog1.c resume.txt
$ echo *.txt
essai.txt resume.txt
$ echo *.h
*.h
$ ls *.h
ls: impossible d'accéder à *.h: Aucun fichier
ou dossier de ce type
```

Fichiers cachés

Les fichiers et répertoires commençant par '.' sont *cachés* y compris les répertoires '.' et '..'

```
$ ls
essai.txt prog1.c resume.txt
$ ls -a
. .. .ancien.txt essai.txt prog1.c resume.txt
```

Les motifs commençant par * ou ? ne correspondent pas aux fichiers cachés.

```
$ ls *.txt
essai.txt resume.txt
$ ls .*.txt
.ancien.txt
```

Fichiers cachés habituels

La plupart des fichiers cachés sont dans `$HOME` :

```
$ ls -a $HOME
./
../
.bash_history       historique des commandes
.bashrc             lancé au démarrage de bash
.profile            lancé au login
.config/            XDG, configuration des programmes
.cache/             XDG, données non essentielles
.local/share/       XDG, données utilisateur
.gconf/             réglages via gconf-editor
.mozilla/           données de firefox
.thunderbird/       données de thunderbird
.ssh/               configuration de ssh
```

3 - Développement des accolades

Permet la création de chaînes indépendamment de l'existence des fichiers.

Syntaxe : `{mot1,...,motn}` *sans blanc*

→ bash substitue la chaîne initiale en faisant le produit cartésien des listes

```
$ echo {ga,bu,meu}
ga bu meu
$ echo a{ga,bu,meu}z
agaz abuz ameuz
$ echo {ga,bu,meu}-{eggs,ham}
ga-eggs ga-ham bu-eggs bu-ham meu-eggs meu-ham
```

Imbrication

- On peut imbriquer les {}

```
$ echo ga{zo,pti{foo,bar}lo}bu
                    ↓
$ echo ga{zo,ptifoolo,ptibarlo}bu
                    ↓
gazobu gaptifoolobu gaptibarlobu
```

Protection

- On peut protéger {},␣ avec \ ou '' ou "":

```
$ echo {\{hop\},ploum,a\,b\␣c}
{hop} ploum a,b␣c
$ echo {"{hop}",ploum,'a,b␣c'}
{hop} ploum a,b␣c
```

- Il faut au moins une virgule :

```
$ echo {ga}
{ga}
$ echo {ga,}
ga
```

→ pas de développement

Mélange motif et accolades

Lorsqu'un motif contient des accolades :

- le développement des {} est fait en premier, les motifs sont dupliqués littéralement ;
- puis le développement des motifs est fait sur les fichiers existants.

```
$ ls /home/{thiel,guest}/tp[1-7]/*.c
                        ↓
$ ls /home/thiel/tp[1-7]/*.c /home/guest/tp[1-7]/*.c
                        ↓
$ ls /home/thiel/tp2/ga.c /home/thiel/tp3/bu.c
     /home/guest/tp1/zo.c
```

4 - Branchements multiples avec case

Syntaxe :

```
case mot in
    motif1) instructions ;;
    motif2) instructions ;;
    ...
esac
```

Le `mot` est développé (substitution des variable, etc), puis mis en correspondance avec chaque motif.

Le premier motif qui correspond $\rightarrow$ exécution des instructions puis saut après `esac`.

Les motifs dans case

Règles pour les motifs :

- idem motifs de fichiers : `* ? []`
- sans correspondance à des fichiers effectifs
- pas de développement des `{}`
- + liste motifs avec `|`

```
case "$i" in
    "un")         echo "1" ;;
    deux|trois)   echo "2 ou 3" ;;
    q*)           echo "commence par q" ;;
    *)            echo "rien" ;;
esac
```

`*)` en fin de `case` = cas par défaut

Exemples (1/2)

```
echo -n "Démarrer l'installation ? "
read rep

case "$rep" in
    [oO] | [oO][uU][iI] )
        echo "L'installation va démarrer"
        ;;
    [nN] | [nN][oO][nN] )
        echo "Installation abandonnée"
        exit 0
        ;;
    *)
        echo "Erreur: oui ou non attendu" > /dev/stderr
        exit 1
esac
```

Exemples (2/2)

```
echo "Entrez un chemin :"
read chemin

case "$chemin" in
    *"Mes images"* ) flag="vrai" ;;
    *              ) flag="faux" ;;
esac

if test "$flag" = "vrai" ; then
    echo "Le chemin contient \"Mes images\""
fi
```

5 - Boucle `for`

```
for mot in un deux trois
do
  echo "$mot"
done
```

→ la variable `mot` prend successivement la valeur de chaque élément de la liste.

Il faut un `;` ou un RC avant `do` et `done`

```
for mot in un deux trois ; do echo "$mot" ; done
```

Marche sur une liste de mots (séparateur blanc), pas un intervalle !

Exemples (1/3)

On peut itérer sur un développement de fichiers :

```
for fichier in "Mes progs"/t[dp]??-*.[ch] ; do
    echo -n "Fichier $fichier : "
    cat "$fichier" | wc -l
done
```

Exemples (2/3)

On peut itérer sur une liste contenue dans une variable :

```
liste="bijou caillou chou genou hibou joujou pou"
for mot in $liste ; do
    echo "un $mot, des ${mot}x"
done
```


Pas de "" après `in`, pour découper.

Exemples (3/3)

On peut itérer sur un développement d'accolades :

```
for mot in {{do,re,mi}fa{sol,la},si}{ga,bu}lol ; do
    echo "$mot"
done | column -x          # -x : ligne à ligne
```

affiche :

```
dofasolgalol    dofasolbulol    dofalagalol
dofalabulol     refasolgalol    refasolbulol
refalagalol     refalabulol     mifasolgalol
mifasolbulol    mifalagalol     mifalabulol
sigalol         sibulol
```

Commandes `true` et `false` (1/2)

Commandes `true`, `false` :

Rappel : n'affichent rien, réussit / échoue immédiatement

Essayer : `type true ; help true ; which true`

Usages :

- pour satisfaire la syntaxe

```
if ... ; then true ; else ... ; fi
```

- flags : écritures équivalentes

```
x=true
if test "$x" = "true" ; then ... ; fi
if $x ; then ... ; fi
```

Commandes `true` et `false` (2/2)

- Sortie d'une boucle avec un flag

```
continuer=true
while $continuer ; do
    ...
    continuer=false
    ...
done
```

- Boucle infinie

```
while true ; do
    ...
done
```

Contrôle de boucles

Dans les boucles `for ... ; do ... ; done`
`while ... ; do ... ; done`

`break [n]` arrêt boucle
`continue [n]` itération suivante

`[n]` : pour les boucles imbriquées,
arrêt ou itération suivante de `n` niveaux

6 - Arguments

Ils permettent

- d'éviter de faire des `read` dans un script ;
- de rappeler plus facilement des commandes déjà paramétrées ;
- d'appeler des scripts dans des scripts.

Syntaxe

- En C : `int main (int argc, char *argv[])`
- En bash, variables spéciales :

`$0`	chemin absolu commande	`argv[0]`
`$1`	argument numéro 1	`argv[1]`
`${10}`	argument numéro 10	`argv[10]`
`$#`	nombre d'arguments	`argc-1`
`$*`	liste des arguments séparés par un espace	
`"$@"`	liste des arguments protégés par des ""	

Exemple

Script `kietu` qui récupère un nom et un prénom en argument

```
#! /bin/bash

if test $# -ne 2 ; then
    echo "Erreur : 2 arguments attendus" > /dev/stderr
    exit 1
fi
nom="$1"
prenom="$2"

echo "Vous vous appelez $prenom $nom"
```

Essai :

```
$ ./kietu Lagroseille Jules
Vous vous appelez Jules Lagroseille
```

Modifier les arguments

- Écraser les arguments :

  `set arg1 .. argn`

- Décaler les arguments :

  `shift [n]` en C : `argc -= n ; argv += n;`

Exemple :

```
$ set ga bu zo
$ echo "$# $1 $2"
3 ga bu
$ shift
$ echo "$# $1 $2"
2 bu zo
```

Boucle `while` sur les arguments

```
set le lundi au soleil

while test $# -gt 0 ; do
  echo "$# $1"
  shift
done
```

Affiche :

```
4 le
3 lundi
2 au
1 soleil
```

Différence entre $* et "$@"

`$*` liste des arguments séparés par un ␣
`$@` liste des arguments séparés par un "␣"

```
$ set le "lundi au" soleil
```

`$*`	→	`le␣lundi au␣soleil`	4 mots
`"$*"`	→	`"le lundi au soleil"`	1 mot
`$@`	→	`le"␣"lundi au"␣"soleil`	4 mots
`"$@"`	→	`"le"␣"lundi au"␣"soleil"`	3 mots

Boucle `for` sur les arguments

```
for arg in $* ; do echo "$arg" ; done
```

Risque de re-séparation des arguments

Bonne méthode :

```
for arg in "$@" ; do echo "$arg" ; done
```

Raccourci équivalent (cf `help for`) :

```
for arg ; do echo "$arg" ; done
```

Ne pas faire de `shift` dans un `for`