Scripts et Redirections

Système et environnement de programmation

Université Grenoble Alpes

Plan

1 Scripts shell

2 Redirections des E/S

3 Quelques commandes utiles

Script shell

Commmandes UNIX combinées dans un **fichier ASCII** nommé fichier de commande ou script shell.

Extension `.sh` (convention)

Langage de programmation complet

- Variables
- Structures de contrôle (conditionelle, itération)
- Fonctions

Particularités

- Combine des commandes Unix et des structures du langage
- Syntaxe
- Interprété (pas de compilation)

Exécution, interprétation

Pour exécuter un script

- le munir du droit en exécution (x)
- donner son nom comme commande à l'interpréteur

Il est en suite interprété

- chaque ligne est lue puis exécutée avant de passer à la suivante
- les erreurs de syntaxe ne sont vues qu'en les rencontrant

Avantages / inconvénients

- plus lent
- développement plus rapide et plus simple

Exemple de fichiers de commandes : `sauvegarde.sh` (TP1)

```
#!/bin/bash
cd $HOME/INF203
cp -r TP1 sauve_TP1
cd sauve_TP1
echo Sauvegarde effectuée dans
pwd
echo il contient
ls -l
```

Exemple de fichiers de commandes : `sauvegarde.sh` (TP1)

```
#!/bin/bash
cd $HOME/INF203
cp -r TP1 sauve_TP1
cd sauve_TP1
echo Sauvegarde effectuée dans
pwd
echo il contient
ls -l
```

Exécution en séquence

Certaines commandes connues
`cd`, `pwd`, ...

Commandes séparées par les retours à la ligne (`;` est également séparateur de commandes)

Exemple de fichiers de commandes : `sauvegarde.sh` (TP1)

```
#!/bin/bash
cd $HOME/INF203
cp -r TP1 sauve_TP1
cd sauve_TP1
echo Sauvegarde effectuée dans
pwd
echo il contient
ls -l
```

`#!/bin/bash` : Commentaire spécial (#!), programme à utiliser pour interpréter ce fichier, ici ***/bin/bash***

Les commentaires ordinaires débutent par # et se terminent à la fin de la ligne

Exemple de fichiers de commandes : `sauvegarde.sh` (TP1)

```
#!/bin/bash
cd $HOME/INF203
cp -r TP1 sauve_TP1
cd sauve_TP1
echo Sauvegarde effectuée dans
pwd
echo il contient
ls -l
```

`$HOME` est la valeur de la variable d'environnement `HOME`, chemin absolu vers le répertoire principal

Pause : démo Caséine

Variables

Pas de déclaration, variables créées lors de l'affectation

Pas de type particulier

- leur représentation textuelle est stockée
- convertie dans le bon type selon le contexte

Affectation : *nom_var*=`TP1`
(attention, pas d'espace autour du =)

Valeur : `$nom_var`
`cd $nom_var` → `TP1` devient le répertoire courant

Remarque : si `nom_inconnu` n'a jamais été affecté, `$nom_inconnu` est remplacé par la chaine vide

Arguments de la ligne de commande

L'exécution d'un fichier de commande peut être paramétrée par des arguments fournis sur la ligne de commande

Dans le script ce sont des variables spéciales

- $0 : le nom du script
- $1 : le premier argument
- $2 : le 2ème argument
- ...
- $# : nombre d'arguments donnés ($0 exclus)
- $* : la liste de tous les arguments.

Exemple utilisant ses arguments

Exemple, contenu d'un script ***affiche_args.sh***

```
#!/bin/bash
echo le nom du script est $0
echo $# arguments lui ont été donnés :
echo le premier argument est $1
echo le deuxième argument est $2
echo le troisième argument est $3
echo ...
```

Pour l'exécuter

```
./affiche_args.sh un 2 trois 4 et voila
```

Substitution de commande

Une commande entre `$( )` est exécutée et remplacée par l'affichage qu'elle produit durant son exécution

```
repertoire=$(pwd)
cd
echo le répertoire de login contient $(ls)
# retour au répertoire initial
cd $repertoire
```

C'est grâce à ce mécanisme qu'on fait du calcul en shell

- `expr 7 + 6` affiche 13
- `toto=$(expr 7 + 6)` met 13 dans la variable `toto`

Remarque : une commande entre `` `` `` est également substituée par son affichage, mais les `` `` `` sont plus difficiles à imbriquer que `$()`

Expansion des métacaractères

Dans chaque ligne de commande, l'interpréteur transforme certains caractères avant de découper et d'exécuter le résultat

Nous avons vu `*`, `?` et `[]`

Le $ qui récupère la valeur d'une variable ou débute une substitution de commande est aussi un métacaractère, il y a aussi
`(`, `)`, `|`, `<`, `>`, `&`, `...`

Exemple

```
command=ls
echo L\'exécution de $command sur \$HOME donne :
$command $HOME
```

Métacaractères de protection contre l'expansion

Ces métacaractères rendent d'autres caractères ordinaires

- entre ' ', aucun caractère n'est sujet à l'expansion
- le caractère qui suit \ n'est pas sujet à l'expansion
- entre " ", seul le $ et le \ sont sujets à l'expansion

Exemple

```
truc=machin
echo entre \", il faut '\' pour afficher "'\$truc=$truc'"
```

produit l'affichage

```
entre ", il faut \ pour afficher '$truc=machin'
```

Remarque

Il y a de multiples manières d'arriver au même résultat

```
echo entre \", il faut '\' pour afficher "'\$truc=$truc'"

echo 'entre ", il faut \ pour afficher '\''$truc='$truc\'

echo "entre \", il faut \\ pour afficher '\$truc=$truc'"

echo entre \", il faut \\ pour afficher \'\$truc=$truc\'
```

Seconde remarque

La protection est souvent nécessaire

- pour l'affichage (bien sur)

```
echo On peut ainsi afficher des parenthèses : \(\)
```

- pour donner les bons arguments à certaines commandes

```
expr 6 \* 7
```

- pour avoir des espaces dans la valeur d'une variable

```
toto="$(ls $HOME)"
```

- pour fabriquer des commandes à exécuter

```
demi=21; var42=41
eval toto=\$\(expr \$var$(expr $demi \* 2) + 1\)
# ou
eval toto=\`expr \$var`expr $demi \* 2` + 1\`
```

Plan

1 Scripts shell

2 Redirections des E/S

3 Quelques commandes utiles

Définition

Lorsque l'on exécute une commande, ses données (entrées/sorties)

- sont lues sur l'entrée standard, depuis le clavier
- sont écrites sur la
  - sortie standard (sorties normales)
  - sortie d'erreur standard (messages d'erreur)

  à l'écran dans les deux cas

Rediriger

- l'entrée standard signifie ne pas lire depuis le clavier mais depuis
  - un fichier
  - les sorties d'une autre commande
- une sortie standard signifie ne pas écrire à l'écran mais vers
  - un fichier
  - les entrées d'une autre commande

Redirection des sorties vers un fichier

Ecrire les sorties d'une commande dans un fichier

- sortie standard

  ```
  ls > resultat
  ```

- sortie d'erreur standard

  ```
  cp 2> erreur.log
  ```

Dans les deux exemples n'affiche rien mais

- crée le fichier de nom donné si besoin
- écrase son contenu par la sortie de la commande
  - pour `ls` : `resultat` va contenir la liste des fichiers
  - pour `cp` : `erreur.log` va contenir

    ```
    cp: missing file operand
    Try 'cp --help' for more information.
    ```

On peut concaténer à la suite de l'ancien contenu avec >> au lieu de >

```
ls >> resultat
cp 2>> erreur.log
```

Redirection des entrées depuis un fichier

Lire les entrées d'une commande depuis un fichier

```
# Afficher le compte des lignes sans le nom du fichier
wc -l < mon_programme.c
```

Par rapport au clavier

- on ne voit pas à l'écran ce qui est lu dans le fichier
- pas de blocage, si la fin de fichier est atteinte, les lectures echouent

Redirection d'entrées, exemple détaillé

fichier	addition2.sh

contenu	`echo Saisir deux entiers`
	`read A # stocke dans A une ligne lue`
	`read B`
	`expr $A + $B`

fichier	deux_entiers	un_entier
contenu	10 32	42

- `./addition2.sh < deux_entiers`
  affiche `42`
- `./addition < un_entier`
  fin de fichier pour le `read B` la variable `B` n'est pas créée
  affiche `expr: syntax error`

Quelques combinaisons surprenantes

Redirection des entrées et des sorties

```
./addition2.sh > resultats < deux_entiers
# ou
./addition2.sh < deux_entiers > resultats
```

Attention à l'ordre

- démarre toujours par une commande
- l'argument est à droite des opérateurs < ou >

Un programme peut sembler ne rien faire

```
./addition2.sh > resultat
```

Redirection des entrées/sorties entre commandes

Transmettre les sorties d'une commande comme entrées d'une autre

`./pgm1 | ./pgm2`

`|` est un opérateur nommé *pipe* (tuyau en anglais)

Similaire à :

```
./pgm1 > nom_fichier
./pgm2 < nom_fichier
```

Sauf qu'avec l'opérateur `|`, le système transmet directement les données d'un programme à l'autre sans créer de vrai fichier

Divers

Redirection de la sortie d'erreur standard vers la sortie standard

```
cp 2>&1
```

... pour tout regrouper dans un seul fichier

```
cp >resultat.txt 2>&1
```

attention à l'ordre...

Le trou noir `/dev/null`

```
cp 2>/dev/null
```

pour jeter les messages d'erreur

Plan

1 Scripts shell

2 Redirections des E/S

3 Quelques commandes utiles

Avertissement

Selection non-exhaustive des commandes les plus utiles

Seules les options les plus utiles sont présentées

Pour aller plus loin, consulter le manuel

Filtres

Beaucoup de commandes, sans argument, sont des filtres

- travaillent sur les données lues sur l'entrée standard
- écrivent leur résultat sur la sortie standard

On peut les combiner facilement avec un *pipe* |

`tr`, `head`, `tail`, `cut`, `grep`, `sed`, et beaucoup d'autres...

Quand vous écrirez vos propres commandes, pensez à faire de même.

Rappel : `grep`

`grep [-c | -v] motif [fichier ...]`
affiche les lignes de ses entrées correspondant à un `motif` donné

Pour l'ensemble des lignes de `Candide_chapitre1.txt`

`grep Candide Candide_chapitre1.txt`
affiche celles contenant `Candide`

`grep -c Candide Candide_chapitre1.txt`
affiche le nombre de celles contenant `Candide`

`grep -v Candide Candide_chapitre1.txt`
affiche celles ne contenant pas `Candide`

`grep -v -c Candide Candide_chapitre1.txt`
affiche le nombre de celles ne contenant pas `Candide`

sed

`sed commande [fichier ...]`
affiche les lignes de ses entrées en les filtrant et/ou modifiant selon les instructions données dans la partie `commande`

Pour l'ensemble des lignes de `Candide_chapitre1.txt`

`sed /Candide/d Candide_chapitre1.txt`
affiche celles ne contenant pas `Candide`

`sed s/Candide/Toto/ Candide_chapitre1.txt`
les affiche en remplaçant le premier `Candide` par `Toto`

`sed s/Candide/Toto/g Candide_chapitre1.txt`
les affiche en remplaçant toutes les `Candide` par `Toto`

tr

`tr chaîne1 chaîne2`
`tr -s chaîne`
copie l'entrée standard sur la sortie standard en substituant

- les caractères présents dans `chaîne1` par le caractère de position correspondante dans `chaîne2`
- les répétitions des caractères présents dans `chaîne` par une occurrence unique

`echo toto | tr o a`		`tata`
`echo maman | tr man yoo`	affiche	`yoyoo`
`echo tommme | tr -s m`		`tome`

`head` et `tail`

`head [-n nbl | -c nbc] [fichier ...]`

affiche le début de ses entrées

- les `nbl` premières lignes (option par défaut, 10)
- les `nbc` premiers octets

`head toto.txt`

affiche les 10 premières lignes du fichier `toto.txt`

`head -n 5`

affiche les 5 premières lignes de l'entrée standard

`head -c 42 toto.txt`

affiche les 42 premiers caractères du fichier `toto.txt`

`head` et `tail`

`head [-n nbl | -c nbc] [fichier ...]`
affiche le début de ses entrées

- les `nbl` premières lignes (option par défaut, 10)
- les `nbc` premiers octets

`head toto.txt`
affiche les 10 premières lignes du fichier `toto.txt`

`head -n 5`
affiche les 5 premières lignes de l'entrée standard

`head -c 42 toto.txt`
affiche les 42 premiers caractères du fichier `toto.txt`

De même `tail [-n nbl | -c nbc] [fichier ...]`
affiche la fin de ses entrées

cut

`cut [-c liste | -f liste [-d caractère]] [fichier ...]`
affiche des colonnes de ses entrées

- à des positions (`-c`, indices de caractères) dans chaque ligne
- ou bien de numéros donnés (`-f`), en utilisant `caractère` pour délimiter les colonnes successives

Affiche les colonnes :

`cut -c 3-5 toto.txt`
  situées entre les positions 3 et 5 de `toto.txt`

`cut -c 5-`
  situées entre la position 5 et la fin de ligne sur l'entrée standard

`cut -f 2-5 toto.txt`
  2 à 5 avec la tabulation comme séparateur de colonnes

`cut -f 2,4 -d ' ' toto.txt`
  2 et 4 avec le caractère `' '` comme séparateur de colonnes

sort

`sort [-n | -k pos1[,pos2]] [fichier ...]`

affiche les lignes de ses entrées triées par ordre croissant, avec

- `-n`, selon un ordre numérique (plutôt que lexicographique)
- `-k`, en spécifiant les positions de caractères à utiliser pour trier

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l
```

```
total 1184
-rw-r--r--  1 zebulon  staff    9422 Feb 12 11:20 Slides.aux
-rw-r--r--  1 zebulon  staff  104812 Feb 12 11:20 Slides.log
-rw-r--r--  1 zebulon  staff    6078 Feb 12 11:20 Slides.nav
-rw-r--r--  1 zebulon  staff    1845 Feb 12 11:20 Slides.out
-rw-r--r--@ 1 zebulon  staff  266813 Feb 12 11:20 Slides.pdf
-rw-r--r--  1 zebulon  staff       0 Feb 12 11:20 Slides.snm
-rw-r--r--  1 zebulon  staff   24347 Feb 12 11:27 Slides.tex
-rw-r--r--  1 zebulon  staff   27762 Feb  7 11:35 Slides.tex~
-rw-r--r--  1 zebulon  staff     190 Feb 12 11:20 Slides.toc
-rw-r--r--  1 zebulon  staff     204 Feb 12 11:20 Slides.vrb
-rwxr-xr-x  1 zebulon  staff      40 Feb  7 11:35 addition2.sh
-rw-r--r--  1 zebulon  staff       5 Feb  7 11:35 deux_entiers
-rwxr-xr-x  1 zebulon  staff     206 Feb  7 11:35 exemple_args.sh
-rwxr-xr-x  1 zebulon  staff     177 Feb  7 11:35 exemple_backquotes.sh
-rwxr-xr-x  1 zebulon  staff      96 Feb  7 11:35 exemple_expansions.sh
-rwxr-xr-x  1 zebulon  staff     134 Feb  7 11:35 exemple_vars.sh
-rw-r--r--  1 zebulon  staff       3 Feb 10 18:24 un_entier
-rw-r--r--@ 1 zebulon  staff       5 Feb 10 18:24 un_entier~
```

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l | tr -s ' '
```

```
total 1184
-rw-r--r-- 1 zebulon staff 9422 Feb 12 11:20 Slides.aux
-rw-r--r-- 1 zebulon staff 104812 Feb 12 11:20 Slides.log
-rw-r--r-- 1 zebulon staff 6078 Feb 12 11:20 Slides.nav
-rw-r--r-- 1 zebulon staff 1845 Feb 12 11:20 Slides.out
-rw-r--r--@ 1 zebulon staff 266813 Feb 12 11:20 Slides.pdf
-rw-r--r-- 1 zebulon staff 0 Feb 12 11:20 Slides.snm
-rw-r--r-- 1 zebulon staff 24347 Feb 12 11:27 Slides.tex
-rw-r--r-- 1 zebulon staff 27762 Feb 7 11:35 Slides.tex~
-rw-r--r-- 1 zebulon staff 190 Feb 12 11:20 Slides.toc
-rw-r--r-- 1 zebulon staff 204 Feb 12 11:20 Slides.vrb
-rwxr-xr-x 1 zebulon staff 40 Feb 7 11:35 addition2.sh
-rw-r--r-- 1 zebulon staff 5 Feb 7 11:35 deux_entiers
-rwxr-xr-x 1 zebulon staff 206 Feb 7 11:35 exemple_args.sh
-rwxr-xr-x 1 zebulon staff 177 Feb 7 11:35 exemple_backquotes.sh
-rwxr-xr-x 1 zebulon staff 96 Feb 7 11:35 exemple_expansions.sh
-rwxr-xr-x 1 zebulon staff 134 Feb 7 11:35 exemple_vars.sh
-rw-r--r-- 1 zebulon staff 3 Feb 10 18:24 un_entier
-rw-r--r--@ 1 zebulon staff 5 Feb 10 18:24 un_entier~
```

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l | tr -s ' ' | cut -d' ' -f 5,9
```

```
9422 Slides.aux
104812 Slides.log
6078 Slides.nav
1845 Slides.out
266813 Slides.pdf
0 Slides.snm
24347 Slides.tex
27762 Slides.tex~
190 Slides.toc
204 Slides.vrb
40 addition2.sh
5 deux_entiers
206 exemple_args.sh
177 exemple_backquotes.sh
96 exemple_expansions.sh
134 exemple_vars.sh
3 un_entier
5 un_entier~
```

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l | tr -s ' ' | cut -d' ' -f 5,9 | sort -n
```

```
0 Slides.snm
3 un_entier
5 deux_entiers
5 un_entier~
40 addition2.sh
96 exemple_expansions.sh
134 exemple_vars.sh
177 exemple_backquotes.sh
190 Slides.toc
204 Slides.vrb
206 exemple_args.sh
1845 Slides.out
6078 Slides.nav
9422 Slides.aux
24347 Slides.tex
27762 Slides.tex~
104812 Slides.log
266813 Slides.pdf
```

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l | tr -s ' ' | cut -d' ' -f 5,9 | sort -n |
   tail -n 4
```

```
24347 Slides.tex
27762 Slides.tex~
104812 Slides.log
266813 Slides.pdf
```

Exemple : afficher les 4 plus gros fichiers du répertoire

```
ls -l | tr -s ' ' | cut -d' ' -f 5,9 | sort -n |
   tail -n 4 | cut -d' ' -f2
```

```
Slides.tex
Slides.tex~
Slides.log
Slides.pdf
```

Une solution plus simple...

```
ls -S | head -n 4
```

```
Slides.pdf
Slides.log
Slides.tex~
Slides.tex
```

basename

`basename chaîne [suffixe]`
affiche `chaîne` privée de son chemin (tout ce qui se trouve avant le dernier `/` significatif) et de son éventuel `suffixe` (si `chaîne` se termine par `suffixe`)

```
basename /enseignement/inf203/TD6.pdf
basename /enseignement/inf203/TD6.pdf .pdf
basename /enseignement/inf203/TD6.pdf .c
```

affiche

```
TD6.pdf
TD6
TD6.pdf
```

dirname

`dirname chaîne`
affiche `chaîne` privée de son nom de base (complémentaire de `basename`)

```
dirname /enseignement/inf203/TD6.pdf
dirname inf203/TD6.pdf
dirname TD6.pdf
```

affiche

```
/enseignement/inf203
inf203
.
```

diff

`diff fichier1 fichier2` trouve les différences entre deux fichiers

- les lignes de `fichier1` sont précédées par $<$
- les lignes de `fichier2` sont précédées par $>$
- deux formes pour indiquer les différences
  - [intervalle]`d`[position] ou [position]`d`[intervalle]
    disparition d'une partie d'un des fichiers dans l'autre
  - [intervalle]`c`[intervalle]
    changement, deux portions diffèrent entre les deux fichiers

find

`find repertoire [expression]`

parcourt et **agit** sur l'arborescence du système de fichiers à partir de `repertoire` en fonction de l'expression donnée

`find . -atime 24h`

affiche tous les fichiers ayant été accédés dans les dernières 24h

`find ~ -name "*.c"`

affiche tous les fichiers C à partir du répertoire principal

find

`find repertoire [expression]`

parcourt et **agit** sur l'arborescence du système de fichiers à partir de `repertoire` en fonction de l'expression donnée

`find . -atime 24h`

affiche tous les fichiers ayant été accédés dans les dernières 24h

`find ~ -name "*.c"`

affiche tous les fichiers C à partir du répertoire principal

`find TP1 -name "*.o" -delete`

suppression récursive de tous les `.o` à partir du répertoire `TP1`

`find . -name "*.c" -exec wc -l {} \;`

exécution d'une commande pour chaque fichier trouvé

- `{}` est remplacé par le nom du fichier
- `\;` sert à terminer le `-exec`