Cours Unix 3

Michel Mauny

INRIA ROCQUENCOURT

ETGL

Plan du cours 3

1. Analyse et exécution de la ligne de commande
2. Variables
3. Scripts
4. Types de commandes
5. Commandes composées
6. Structures de contrôle

Ksh: analyse et exécution de la ligne de commande

Analyse et exécution

1. Analyse syntaxique: séparation de la ligne en **mots**
   - **mot**: séquence de caractères séparés par
     - espaces non neutralisés
     - métacaractères: `< > | ; & ( )`
   - les métacaractères sont utilisés pour construire les redirections, pipelines, etc.

   Les espaces, métacaractères peuvent être neutralisés
   - individuellement par `\`
   - en groupe en les mettant à l'intérieur de `"..."` ou `'...'`

   Attention: d'autres caractères sont traités spécialement:

   `\ " ' # $ ` ~ { } * ? [`

   et doivent être neutralisés pour être utilisés pour eux-mêmes.

Analyse et exécution

2. Expansion:
   - des variables (par ex. `$HOME`)
   - des répertoires (`~user`) est expansé en le répertoire personnel de l'utilisateur `user`, et `~` en la valeur de `$HOME`
   - des noms de fichiers (`*`, `?`, `[...]`)
3. Analyse des constructions (appels de commandes, structures de contrôle).
4. Exécution (interprétation)

Variables

Variables: petits exemples

```
$ V1=toto
$ echo $V1
toto
$ V2='titi tutu'
$ echo $V2
titi tutu
$ echo '$V2'
$V2
$ echo "$V1 $V2"
toto titi tutu
$ V3="$V1 $V2 tata"
$ echo $V3
toto titi tutu tata
```

Attention à la différence entre `"..."` et `'...'`!

Parenthèse: substitution de commandes

Stocker le résultat de l'appel à une commande dans une variable?

```
$ msg="Je travaille dans `pwd`"       # Old-style
$ echo $msg
 Je travaille dans /tmp
$ msg="Je travaille dans $(pwd)"       # ksh/bash-style
$ echo $msg
 Je travaille dans /tmp
$ wc -l $(find . -type f -name '*.c' -print)
 ...
```

Note: préférer `$(< foo)` à `$(cat foo)` pour efficacité.

Variables propres au processus courant

Variables locales (au processus courant) définies par:

```
VARIABLE=valeur
```

mais restent propres au processus courant (non héritées).

```
$ VAR=toto
$ ksh
  $ echo $VAR

  $ exit
$ echo $VAR
toto
$ echo ${VAR}titi
tototiti
$ echo "${VAR}titi"
tototiti
```


Variables exportées (copiées) vers les processus fils

`export VARIABLE=valeur`	`export VAR1 VAR2 ... VARn`
`typeset -x VARIABLE=valeur`	`typeset -x VAR1 VAR2 ... VARn`

Recopiées par les processus fils (ce ne sont *pas* des variables globales):

```
$ export VAR=toto
$ ksh
  $ echo $VAR
  toto
  $ VAR=tralala
  $ exit
$ echo $VAR
toto
```


Types de variables

Historiquement, tout est chaîne en shell. Les shells récents (ksh, bash) disposent de plusieurs types de variables:

- chaînes: `X='abc efg'`
- numériques: `typeset -i cnt=0; cnt=${cnt}+1`
- tableaux:
  `nom[0]='zero'; echo ${nom[0]}`

Les valeurs et opérations arithmétiques:

```
cnt=$(( 3+4 ))
cnt=${cnt}*5   # pas d'espace de chaque côté de *
cnt=$(( ${cnt} * 5 ))
```


Calculs numériques

Façon ksh/bash:

```
$(( ... ))
```

Façon "ancien shell": commande `expr` et substitution de commande:

```
$ expr 5 '*' 5
 25
$ cnt=$( expr 5 '*' 5 )
$ echo Il y a ${cnt} élèves, soit $(expr ${cnt} '*' 2) oreilles
 Il y a 25 élèves, soit 50 oreilles
```

Variables prédéfinies

Déjà aperçu: `HOME`, `PATH`, `USER`, PS1.

Aussi:

- `$IFS`: (*Internal Field Separator*) espaces. Par défaut, blanc, tab, newline.
- `$COLUMNS`, `$LINES`: taille du terminal courant.
- `$$`, le numéro de processus du shell.
- `$PPID`, la numéro du processus parent du shell.
- `$PS2`, `$PS3`: invites auxiliaires (commandes multilignes, interaction avec l'utilisateur).
- `$VISUAL`, `$EDITOR`: non nécessairement définies.

Lecture de variables

```
$ echo -n 'Enter your name: '; read name; echo "Hi, ${name}"
 Enter your name: robert
 Hi, robert
```

Ce schéma étant courant (question/lecture), `ksh` permet de le faire en une seule commande:

```
$ read position?"and what is ${name}'s position? "
 and what is robert's position? engineer
```


Lecture de variables

Lecture multiple: utilise `$IFS` pour séparer les éléments à lire:

```
$ read a b c
 1 2 3 4 5
$ echo $a
 1
$ echo $c
 3 4 5
```

On peut préfixer une commande par une affectation de variable: la modification de variable n'est opérée que durant l'exécution de la commande.

```
$ echo $IFS
 # espaces, donc invisibles
$ IFS=':' read user pass uid gid realname home shell </etc/passwd
$ echo $home
 /root
```


Substitution de variables

Variables définies ou non?

`${var:-mot}` si `var` est définie et non vide, vaut `$var`, sinon `mot`.
`${var:?mot}` le shell imprime `mot` et quitte (si non-interactif) si `var` indéfinie.
Note: `mot` peut être un appel de commande: `${dir:-$(pwd)}`

Effacer un segment initial de chaîne:

`${var#motif}` vaut `$var` où on a retiré le plus court segment initial filtré par `motif` (`##` pour le segment le plus long).

```
$ echo ${HOME#/home*}
 /tom
$ echo ${HOME##/home*}
```

Pour un segment terminal, `%` et `%%` au lieu de `#` et `##`.

Scripts

Un script shell est sous Unix un fichier

- dont la première ligne est `#!/bin/sh`
- contenant des commandes shell à exécuter
- et qui est exécutable

Si `script` non exécutable, alors:

- `ksh -v script` — tester la syntaxe
- `ksh -x script` — exécuter en imprimant chaque action au terminal
- `ksh script` — exécuter
- `. script` — inclure dans la session courante

Scripts: un exemple

```
$ ls -l fdate.sh
 -rwxrwxr-x    1 tom  users   49 Oct 26 11:17 fdate.sh
$ cat fdate.sh
 #!/bin/ksh
 export LANG=fr_FR
 /bin/date
$ ./fdate.sh
 dim oct 26 11:18:30 CET 2003
$ ksh -x fdate.sh
 + export LANG=fr_FR
 + /bin/date
 dim oct 26 11:18:49 CET 2003
```


Paramètres positionnels

Les paramètres d'un script sont numérotés de 0 à 9 (voire plus, selon les shells):

- `$0` le nom de la commande telle qu'elle a été exécutée (`fdate.sh`, `./fdate.sh`, qu'elle soit lancée directement ou via un interprète);
- `$1`, `$2` ... les arguments de la commande.

À l'intérieur du script:

- `$*` représente la liste des (éléments composant les) arguments
- `"$@"` représente la liste des arguments
- `$#` représente le nombre d'arguments
- `shift [n]` décale les paramètres positionnels de `n`
- `set mot ...` modifie la valeur de **tous** les paramètres positionnels

Paramètres positionnels: exemple

```
$ cat script.sh
 #!/bin/sh
 echo -n \$#=$# ''; echo \$0=$0; echo \$1=$1; echo \$2=$2
 echo \$*=$*; echo \$@="$@"
 ls $*
 ls "$@"
$ ./script.sh -l "mon repertoire"
 $#=2 $0=./script.sh
 $1=-l
 $2=mon repertoire
 $*=-l mon repertoire
 $@=-l mon repertoire
 ls: mon: No such file or directory
 ls: repertoire: No such file or directory
 ls: mon repertoire: No such file or directory
```

Types de commandes

Exécutables

Exécutables: scripts, binaires

- Sans chemin d'accès: situés dans un des éléments du `PATH`
- Avec chemin d'accès (`./date`, `/bin/date`)

*Commandes prédéfinies (*builtins*)*

Commandes prédéfinies et interprétées directement par le shell

- `cd`, `umask` et autres commandes modifiant des variables et comportements du shell
- `pwd`, `kill`, `echo`: pas de sous-processus à allouer
- `alias`, `unalias` gestion des abréviations (voir plus loin)
- `print`: similaire à `echo`

*Commandes prédéfinies (*builtins*)*

- `typeset`: affiche ou affecte les attributs d'identificateurs Principaux usages

  `-i` variables de type entier
  `-l` variables dont la valeur est convertie en minuscules
  `-f` identificateurs représentant des fonctions (voir plus loin)
  `-r` en lecture seule (non modifiable, *readonly*)
  `-x` dont la valeur est exportée
- `unset` libère des identificateurs de leur valeur en tant que variable ou fonction. Par exemple:

  ```
  test -x /bin/uname -a "$(/bin/uname)" = Linux && unset MANPATH
  ```

  libère la variable `MANPATH` si le système est Linux.
- `shift` décale les paramètres positionnels

*Commandes prédéfinies (*builtins*)*

Gestion des processus (*job control*, dans un shell interactif). Lorsqu'une commande (ou un pipeline) est lancé en arrière-plan (avec `&`), un numéro de **job** lui est affecté.

On identifie généralement les numéros de jobs par $\%n$ dans les commandes ci-dessous:

- `jobs`: liste les jobs en cours
- `fg`, `bg`, ramène au premier plan ou renvoie en arrière-plan les jobs dont les numéros sont en arguments de la commande
- `wait`: attend la terminaison des jobs en cours (ou indiqués sur la ligne de commande)

*Commandes prédéfinies (*builtins*)*

Valeurs (codes) de retour:

- `return [n]` donne une valeur de retour (un entier) à une fonction ou à un script inclus par "."
- `exit [n]` termine un script (ou une session) en lui donnant une valeur de retour

Convention: $0 ==$ succès, $n > 0 ==$ échec.

*Commandes prédéfinies (*builtins*)*

Évaluation d'expressions conditionnelles:

```
test expression
[ expression ]
```

- *-option fichier* (`-e`, `-r` `-w`, `-x`, `-f`, `-d`, `-h`, `-b`, `-c`)
- *-option chaîne* (`-z`: chaîne vide?, `-n`: chaîne non vide?)
- *chaîne op chaîne* (*op* peut être `=`, `==`, `!=`)
- *nombre op nombre* (*op* peut être `-eq`, `-ne`, `-ge`, `-gt`, `-le`, `-lt`)
- *expression* `-a` *expression*
- *expression* `-o` *expression*
- `(` *expression* `)` Attention: les parenthèses doivent être neutralisées

*Commandes prédéfinies (*builtins*)*

Évaluation d'expressions conditionnelles (suite):

```
[[ expression ]]
```

Similaire à `test` et `[...]`, mais

- `-a` et `-o` sont remplacés par `&&` et `||`
- l'opérande droite de `=` et `!=` est interprétée comme un **motif**
- deux opérateurs additionnels sur les chaînes: `<` et `>`
- les parenthèses ne doivent pas être neutralisées

Pour les expressions conditionnelles, ≪être vrai≫ signifie ≪retourner l'entier 0≫.

Quelques mots sur les chaînes

Comme dans tous les langages de programmation, les chaînes de caractères peuvent contenir des «échappements» permettant de coder ces caractères:

`\n` retour à la ligne

`\t` tabulation

`\d` caractère *backspace*

`\0xxx` code octal

Ces caractères sont interprétés par `echo` (builtin) et par `print`.

Expressions arithmétiques

`ksh` attend des expressions arithmétiques:

- à droite d'une affectation de variable entière (`typeset -i`)
- comme opérande des opérateurs arithmétiques de `expr`, `[...]` et `[[...]]`
- à l'intérieur de `$(( ... ))` (synonyme de `let "`*expression*`"`)

Expressions arithmétiques

Les expressions arithmétiques sont construite à partir de:

- constantes positives
- variables:
  - dont la valeur est une constante entière
  - ou bien dont la valeur est la *syntaxe* d'une expression, auquel cas `ksh` procède à son évaluation lorsque c'est nécessaire
  - une expression cponstruite à l'aide des opérateurs arithmétiques ou booléens suivants
    - Unaires: `+ - ! ++ --`
    - Binaires: `, = *= /= %= += -= <<= >>= &= ^= |= || && | ^ & == != < <= >= > << >> + - * / %`
    - Ternaires: `_?_:_`
    - Groupes: `( ... )`

Les opérateurs arithmétiques (extrait de `man pdksh`)

The operators are evaluated as follows:

`unary +` result is the argument (included for completeness).

`unary -` negation.

`!` logical not; the result is 1 if argument is zero, 0 if not.

`~` arithmetic (bit-wise) not.

`++` increment; must be applied to a parameter (not a literal or other expression) - the parameter is incremented by 1. When used as a prefix operator, the result is the incremented value of the parameter, when used as a postfix operator, the result is the original value of the parameter.

`--` similar to ++, except the parameter is decremented by 1.

Les opérateurs arithmétiques (extrait de `man pdksh`*)*

`,` separates two arithmetic expressions; the left hand side is evaluated first, then the right. The result is the value of the expression on the right hand side.

`=` assignment; variable on the left is set to the value on the right.

`*= /= %= += -= <<= >>= &= ^= |=` assignment operators; `<var> <op>= <expr>` is the same as `<var> = <var> <op> ( <expr> )`.

`||` logical or; the result is 1 if either argument is non-zero, 0 if not. The right argument is evaluated only if the left argument is zero.

`&&` logical and; the result is 1 if both arguments are non-zero, 0 if not. The right argument is evaluated only if the left argument is non-zero.

Les opérateurs arithmétiques (extrait de `man pdksh`*)*

`|` arithmetic (bit-wise) or.

`^` arithmetic (bit-wise) exclusive-or.

`&` arithmetic (bit-wise) and.

`==` equal; the result is 1 if both arguments are equal, 0 if not.

`!=` not equal; the result is 0 if both arguments are equal, 1 if not.

`<` less than; the result is 1 if the left argument is less than the right, 0 if not.

`<= >= >` less than or equal, greater than or equal, greater than. See `<`.

`<< >>` shift left (right); the result is the left argument with its bits shifted left (right) by the amount given in the right argument.

Les opérateurs arithmétiques (extrait de `man pdksh`*)*

`+ - * /` addition, subtraction, multiplication, and division.

`%` remainder; the result is the remainder of the division of the left argument by the right. The sign of the result is unspecified if either argument is negative.

`<arg1> ? <arg2> : <arg3>` if `<arg1>` is non-zero, the result is `<arg2>`, otherwise `<arg3>`.

Alias

Permettent de définir des synonymes pour des commandes:

- `alias nom=mot`
- `unalias nom`
- `alias`, sans argument, liste les aliases connus

Exemples:

```
alias ll='ls -l'

alias line='IFS= read && echo "${REPLY}"'
```

Fonctions

Permettent de définir des sous-programmes (fonctions, procédures), de sorte à mieux structurer les scripts.

```
fname() {
  ...
  ...
}
```

Les paramètres des fonctions sont `$1`, `$2`, ... et leur nombre est `$#`.

Les variables modifiées sont des variables globales, sauf si elles sont déclarées par `typeset`.

Fonctions: exemples

```
error() {
    echo "$*" >&2
    exit 1
}

usage() {
    error "Usage: $(basename $0) [-a] file ..."
}
```


Quelques mesures de prudence

Éviter:

- de définir des alias ou des fonctions ou des commandes du même nom que des builtins/alias/commandes déjà définies. Se trouver ses propres noms.
- d'avoir ≪.≫ dans le `PATH`, ou alors le mettre **à la fin**: si vous l'avez au début, vous pouvez, dans certaines circonstances, exécuter une autre commande que celle que vous pensez exécuter.

Commandes composées

Groupes

`{ commande ; commande ; ... ; commande ; }` Utile pour rediriger les entrées-sorties de plusieurs commandes.

Exemple:

```
{ echo Bonjour; echo Monsieur; } >/tmp/output
```

Attention: les ≪;≫ sont équivalents à des fins de ligne en shell.

La commande ci-dessus est équivalente à:

```
{ echo Bonjour; echo Monsieur
  } >/tmp/output
```


Groupes

`( commande ; commande ; ... ; commande )` comme ci-dessus, mais exécuté dans un **sous-shell**.

Exemple:

```
( echo Bonjour; echo Monsieur ) >/tmp/output
```


Listes

On peut composer des commandes grâce aux opérateurs `&&` (conjonction), `||` (disjonction) et `;` (séquence):

`commande1 ; commande2`: les deux commandes sont exécutées en séquence (quelque soit le code de retour de `commande1`)

`commande1 && commande2`: si `commande1` renvoie le code 0 (réussit), alors `commande2` est exécutée

`commande1 || commande2`: si `commande1` renvoie un code non nul (échoue), alors `commande2` est exécutée

Exemple:

```
test -x /bin/date && /bin/date || exit 1
{ test -x /bin/date && /bin/date ; } || exit 1
```


Pipelines

`commande1 | commande2 | commande3` les 3 commandes sont exécutées en connectant la sortie standard de la commande $n-1$ à l'entrée standard de la commande n

Les processus sont synchronisés: le noyau Unix ralentit ceux qui produisent trop vite, et ceux qui consomment trop vite sont mis en attente.

Les pipelines sont donc généralement préférables aux fichiers temporaires.

Exemple:

```
{ ls -l / | grep '^d' | sort >/tmp/macommande.out; } \
    2>/tmp/macommande.err
```

Structures de contrôle

Généralités

Les structures de contrôle du shell sont assez classiques: conditionnelles, analyse de cas et boucles.

On peut sortir des boucles par la commande `break`. On peut sortir de plusieurs niveaux de boucles imbriquées en lui adjoignant un argument entier.

On peut passer immédiatement à l'itération suivante de la boucle suivante par la commande `continue`. On passe à l'itération suivante de la $n$ième boucle imbriquée en lui adjoignant un argument numérique n.

Les conditionnelles

`if liste0 then liste [elif liste then liste] ... [else liste] fi`
si l'exécution de `liste0` retourne 0, alors c'est la partie "then" qui est exécutée, sinon ...

Exemple:

```
if [ -x /bin/fdate ]; then
    /bin/fdate
elif [ -x /bin/date ]; then
    /bin/date
else
    echo 'Desole' >&2 ;  exit 1
fi
```

Attention à la syntaxe: il est raisonnable (par souci de compatibilité) de suivre le schéma ci-dessus.

Analyse de cas

`case mot in motif [| motif] ... ) liste ;; ... esac` exécute la première `liste` dont le `motif` filtre `mot`. Attention: les `motifs` sont au préalable soumis à des expansions par le shell (variables, commandes).

Exemple:

```
case "$HOME" in
   /) echo 'Must be root, then' ;;
   /home/*|/users/*) echo 'Hmm, regular user?' ;;
   '') echo 'Homeless?' ;;
   *) echo 'Cannot determine your class' ;;
esac
```

Boucle `for`

`for nom [ in mot ... ] ; do liste done`: classique... sauf si la partie «`in mot ...`» est absente: dans ce cas, l'itération se fait sur les paramètres positionnels ($1, $2, etc).

Exemple:

```
for f in $(ls); do
  case ‘file $f‘ in
    ’ASCII text’) TEXTFILES="$f $TEXTFILES";;
  esac
done

for i in 0 1 2 3 4 5 6 7 8 9; do ...
for f in *; do ...
for arg in "$@"; do ...
for arg ; do ...
```


Boucle `while`

`while liste do liste done`: classique.

Exemples:

```
while [ $# -gt 0 ] ; do
  case "$1" in
     -l) LONG=1;;
     -*) usage;;       # Prints usage and aborts
  esac
  shift
done

(IFS=’:’
 while read user rest; do
    echo $user
 done </etc/passwd)
```


Sélection

`select nom [ in mot ... ] ; do liste done`: propose un menu (texte) permettant à l'utilisateur de choisir une option.

Exemple:

```
$ select rep in Dormir Travailler ; do
>   case "$REPLY" in
>     1) echo Bonne nuit "($rep)"; break ;;
>   esac
> done
 1) Dormir
 2) Travailler
 #? 2
 #? 1
 Bonne nuit (Dormir)
```


Exemple de script comptage de fichiers

Exemple de script

Cahier des charges:

- `countdx [dir ...]`: compte les fichiers exécutables et les répertoires dans ses répertoires arguments.
- Étapes:
  1. Arguments: 0 ou plusieurs répertoires (rép. courant si 0)
  2. Messages d'erreurs (pas un répertoire, pas d'accès)
  3. Lister les répertoires, et compter exécutables et sous-répertoires (ne pas compter ≪`.`≫ et ≪`..`≫)
  4. Afficher le résultat
  5. Terminer