Initiation à l'environnement Unix
CM9 (objets trouvés)
`read`, logique des statuts de sortie (suite), *job control*, développements de variable modifiés, *job control*

Pierre Rousselin

Université Sorbonne Paris Nord
L1 informatique et DL
novembre 2025

Commande `read`

Mots-clés `&&`, `||` et `!`

Exécution d'une commande *en arrière-plan*

Job control

Développements de variables modifiés

La commande `read`

```
$ read foo bar baz
    les petits poissons dans l'eau
$ echo $?
0
$ printf '%s\n' "$foo" "$bar" "$baz"
les
petits
poissons dans l'eau
```

- `read` réussit si elle parvient à lire une ligne depuis son entrée standard
- les arguments de `read` sont des noms de variable
- la première variable reçoit le premier mot de la ligne, la deuxième le deuxième, *etc*
- si plus de mots que de variable, la dernière reçoit la fin de la ligne
- si moins de mots que de variables, la fin de la ligne [1] est affectée à la dernière variable
- séparation de la ligne en mots : suivant l'IFS (soupirs...)

1. éventuellement privée de ses blancs qui sont dans l'IFS...

read, \ et option -r

```
$ read foo bar baz
       les\ petits poi\
> ssons\ dans l'eau\\\\
$ printf '%s\n' "$foo" "$bar" "$baz"
les petits
poissons dans
l'eau\\
```

- Par défaut, les contre-obliques rendent le caractère suivant « littéral » (inhibent les séparateurs de champ ou la contre-oblique elle-même)
- ou « annulent une fin de ligne »
- Avec l'option `-r` (pour *raw*, c'est-à-dire « brut », sans traitement), la contre-oblique redevient normale :

```
$ read -r foo bar
\\\\\ a b c\
$ printf '%s\n' "$foo" "$bar"
\\\\\
a b c\
```

`read`, `IFS` et blancs en fin ou début de ligne

- On change souvent l'IFS uniquement pour la commande **`read`** (et pas pour le shell courant) :

```
$ IFS=/ read foo bar baz
/usr/bin/
$ printf '%s\n' "$foo" "$bar" "$baz"

usr
bin
```

- Avec une seule variable on devrait lire toute la ligne

```
$ read foo
  un deux trois
$ printf '%s\n' "$foo"
un deux trois
```

- mais les « blancs » contenus dans l'IFS au début et à la fin de la ligne sont supprimés...
- donc en vidant l'IFS, on a notre ligne :

```
$ IFS= read foo
  un deux trois
$ printf '%s\n' "$foo"
  un deux trois
```

Lire une ligne tranquillement

- Sauf cas particuliers ou petites bidouilles rapides, il vaut mieux lire les lignes d'entrée sans aucun traitement.
- En effet, `read` (dans sa version standard [2]) ne permet que d'affecter des mots à un nombre de variables connu à l'avance, ...
- ... alors qu'on parle d'entrée et donc (presque par définition) ce nombre de mots est le plus souvent inconnu.
- Quant au comportement des contre-obliques, le cas « par défaut » est certainement le moins utile.
- Donc en général, on lit notre ligne en vidant l'IFS, et avec l'option `-r`

  ```
  IFS= read -r ligne
  ```

- et après, on fait des traitements éventuels sur cette ligne

  ```
  set -- $ligne; printf '%d mots\n' $#
  ```

2. voir l'option `-a` (resp. `-A`) pour `read` dans `bash` (resp. `zsh`)

Lire le contenu d'un fichier ligne par ligne

```
#!/bin/sh
case $# in
0)
        printf "pas d'argument, on sort" >&2
        exit 1
esac
if ! [ -f "$1" ]; then
        printf '%s\n' "le fichier $1 n'existe pas" >&2
        exit 2
fi
if ! [ -r "$1" ]; then
        printf '%s\n' "le fichier $1 n'est pas lisible" >&2
        exit 3
fi
fichier=$1
while IFS= read -r ligne; do
        printf '%s\n' "$ligne"
done <"$fichier"
```

Lire le contenu d'un fichier ligne par ligne

- Attention au piège classique :

```
while IFS= read -r ligne <"$fichier"; do
        printf '%s\n' "$ligne"
done
```

  est une boucle infinie dès que "$fichier" contient au moins une ligne
- car le fichier est ouvert (au début donc) à chaque commande `read`.
- mais utiliser une duplication de descripteur de fichier fonctionne :

```
$ printf 'foo\nbar\n' >baz.txt
$ exec 3<baz.txt # ouverture en lecture dans fd 3
$ while IFS= read -r ligne <&3; do printf '%s\n' "$ligne"; done
foo
bar
```

Commande `read`

Mots-clés `&&`, `||` et `!`

Exécution d'une commande *en arrière-plan*

Job control

Développements de variables modifiés

Négation

Avant une commande, le mot-clé `!` (point d'exclamation) nie simplement le code de retour de la commande :

- Si le code de retour de `cmd` est 0 (succès, vrai), alors celui de `! cmd` est 1.
- Si le code de retour de `cmd` est > 0 (échec, faux), alors celui de `! cmd` est 0.

```
$ ls /usr/
bin games include lib lib64 libexec local sbin share src tmp
$ echo $?
0
$ ! ls /usr/
bin games include lib lib64 libexec local sbin share src tmp
$ echo $?
1
$ ! ls /ust/
ls: impossible d'accéder à '/ust/'...
$ echo $?
0
```

Conjonction court-circuit

Rappel de logique : table de vérité de « et » (P et Q sont des variables propositionnelles).

P	Q	P et Q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

Conjonction court-circuit

Rappel de logique : table de vérité de « et » (P et Q sont des variables propositionnelles).

P	Q	P et Q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

On remarque que si P est faux, ce n'est pas la peine de considérer Q, le résultat sera de toute façon faux !
C'est le principe de l'évaluation *court-circuit* des opérations logiques (*short-circuit evaluation*).

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

```
cmd1 && cmd2 && ... && cmdn
```

et est exécutée de la façon suivante :

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé `&&`.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.
- En cas de succès, la commande suivante (`cmd2`) est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd2`.
- En cas de succès, etc
- Si toutes les commandes réussissent, le code de retour de la liste-et est 0.

Le mot-clé &&

En shell, l'opérateur de conjonction (le « et » logique) est le mot-clé &&.

Une « liste-et » (*AND-list*) de commandes a la forme suivante :

`cmd1 && cmd2 && ... && cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd1`.
- En cas de succès, la commande suivante (`cmd2`) est exécutée. En cas d'échec (code de retour > 0), l'exécution de la liste-et est terminée et son code de retour est celui de `cmd2`.
- En cas de succès, etc
- Si toutes les commandes réussissent, le code de retour de la liste-et est 0.

```
if [ -e "$rep" ] && ! [ -d "$rep" ]; then
    printf "$rep existe et n'est pas un répertoire !\n"
    exit 1
fi
```

Disjonction court-circuit

Rappel de logique : table de vérité de « ou » (P et Q sont des variables propositionnelles).

P	Q	P ou Q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

Disjonction court-circuit

Rappel de logique : table de vérité de « ou » (P et Q sont des variables propositionnelles).

P	Q	P ou Q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

On remarque que si P est vrai, ce n'est pas la peine de considérer Q, le résultat sera de toute façon vrai !

Le mot-clé `||`

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

```
cmd1 || cmd2 || ... || cmdn
```

et est exécutée de la façon suivante :

Le mot-clé `||`

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

`cmd1 || cmd2 || ... || cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.

Le mot-clé ||

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé ||.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

`cmd1 || cmd2 || ... || cmdn`

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, la commande suivante (`cmd2`) est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, etc
- Si toutes les commandes échouent, le code de retour de la liste-ou est celui de la dernière commande.

Le mot-clé ||

En shell, l'opérateur de disjonction (le « ou » logique) est le mot-clé `||`.
Une « liste-ou » (*OR-list*) de commandes a la forme suivante :

```
cmd1 || cmd2 || ... || cmdn
```

et est exécutée de la façon suivante :

- la commande `cmd1` est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, la commande suivante (`cmd2`) est exécutée. En cas de succès (code de retour 0), l'exécution de la liste-ou est terminée et son code de retour est 0.
- En cas d'échec, etc
- Si toutes les commandes échouent, le code de retour de la liste-ou est celui de la dernière commande.

```
mkdir rep_temp || exit 1
```

Commande `read`

Mots-clés `&&`, `||` et `!`

Exécution d'une commande *en arrière-plan*

Job control

Développements de variables modifiés

Attente de la fin d'une commande

Dans l'exemple suivant, le système attend 5 secondes, puis la commandes `sleep 5` se termine avec le code de retour 0 (succès), ce qui entraîne l'impression de `prêt !` sur la sortie standard.
Une fois la « liste-ET » terminée, le shell peut lancer la commande suivante, affichant `voilà !` sur la sortie standard.

```
$ sleep 5 && echo 'prêt !'; echo 'voilà !'
prêt !
voilà !
```

Attente de la fin d'une commande

Dans l'exemple suivant, le système attend 5 secondes, puis la commandes `sleep 5` se termine avec le code de retour 0 (succès), ce qui entraîne l'impression de `prêt !` sur la sortie standard.
Une fois la « liste-ET » terminée, le shell peut lancer la commande suivante, affichant `voilà !` sur la sortie standard.

```
$ sleep 5 && echo 'prêt !'; echo 'voilà !'
prêt !
voilà !
```

Autre exemple énervant, si vous avez un éditeur de texte en mode graphique (par exemple `gedit`, `kate`, ...) et que vous voulez le lancer depuis le terminal :

```
$ gedit chef_doeuvre.c
Je ne peux plus
lancer de commandes GRRRRR
```

Vous ne reprendrez la main sur le shell que lorsque vous aurez quitté votre éditeur de texte car le shell *attend* que cette commande se termine !

Non attente de la fin d'une commande

Si on termine la commande `sleep 5 && echo 'prêt !'` par une esperluette (`&`) à la place de `;`, le shell n'attend pas la fin de cette commande pour lancer la suivante.

```
$ sleep 5 && echo 'prêt !'& echo 'voilà !'
[1] 7829
voilà !
$ prêt !
```

Non attente de la fin d'une commande

Si on termine la commande `sleep 5 && echo 'prêt !'` par une esperluette (`&`) à la place de `;`, le shell n'attend pas la fin de cette commande pour lancer la suivante.

```
$ sleep 5 && echo 'prêt !'& echo 'voilà !'
[1] 7829
voilà !
$ prêt !
```

- Le shell a imprimé deux nombres : le premier, entre crochets est le *numéro de job* de la commande en arrière-plan et le second est son *PID* (*Process ID*, c'est-à-dire identifiant de processus).
- L'affichage de `prêt !` qui faisait partie de la commande en arrière-plan peut se faire de façon assez désordonnée... (ici juste après l'invite de commande, mais pas d'inquiétude, cela ne gêne pas pour entrer de nouvelles commandes).

Non attente de la fin d'une commande

```
$ firefox&
[1] 8912
$ gedit idees_geniales.tex& evince idees_geniales.pdf &
[2] 8914
[3] 8915
$ pdflatex idees_geniales.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.20 (TeX Live
2019) (preloaded format=pdflatex)
...
```

Tant que ces commandes ne sont pas terminées :

- `firefox` a pour *numéro de job* 1 et pour *PID* 8912 ;
- `gedit idees_geniales.tex` a pour *numéro de job* 2 et pour *PID* 8914 ;
- `evince idees_geniales.pdf` a pour *numéro de job* 3 et pour *PID* 8915 ;

Non attente de la fin d'une commande

```
$ firefox&
[1] 8912
$ gedit idees_geniales.tex& evince idees_geniales.pdf &
[2] 8914
[3] 8915
$ pdflatex idees_geniales.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.20 (TeX Live
2019) (preloaded format=pdflatex)
...
```

Tant que ces commandes ne sont pas terminées :

- `firefox` a pour *numéro de job* 1 et pour *PID* 8912 ;
- `gedit idees_geniales.tex` a pour *numéro de job* 2 et pour *PID* 8914 ;
- `evince idees_geniales.pdf` a pour *numéro de job* 3 et pour *PID* 8915 ;

Le processus `pdflatex idees_geniales.tex` a un numéro de job (sans doute 4 ?) et un *PID* (peut-être 8916 ?) mais pour l'instant on ne les connaît pas.

Le paramètre spécial $!

Le paramètre spécial `$!` se développe en le *PID* du dernier processus lancé en arrière-plan.

```
$ sleep 1000&
[1] 13155
$ echo $!
13155
$ sleep 1337&
[2] 13183
$ echo $!
13183
```

Le terminateur de commande &

- Une commande terminée par le caractère « esperluette » (`&`) est exécutée *en arrière-plan*.
- Cela signifie que le shell *n'attend pas* la fin de la commande avant de lancer la suivante (ou d'être prêt à recevoir la commande suivante de l'utilisateur).

On l'utilise souvent pour :

- des commandes qui ouvrent des fenêtres dans l'environnement graphique (`$ firefox &` ou `$ gedit&` par exemple) et qui ne seront donc pas terminées avant que l'utilisateur ne les quitte manuellement.
- des commandes qui prennent beaucoup de temps à se terminer (par exemple la compilation d'un gros programme `$ make &` ou de gros calculs `$ ./gros_calc&`).

Une commande lancée par un shell interactif a un *numéro de job pour ce shell* (et un *PID* comme tous les processus).

Commande `read`

Mots-clés `&&`, `||` et `!`

Exécution d'une commande *en arrière-plan*

Job control

Développements de variables modifiés

Arrêter ou suspendre une commande au premier plan

Après la première commande `sleep`, on a tapé sur `Ctrl-c` et après la suivante sur `Ctrl-z`.

```
$ sleep 60000 # oups !
^C
$ sleep 60
^Z
[1]+  Stoppé                  sleep 60
```

Arrêter ou suspendre une commande au premier plan

Après la première commande `sleep`, on a tapé sur `Ctrl-c` et après la suivante sur `Ctrl-z`.

```
$ sleep 60000 # oups !
^C
$ sleep 60
^Z
[1]+  Stoppé                  sleep 60
```

Depuis le terminal,

- `Ctrl-c` envoie le signal `SIGINT` (*interrupt*) au processus au premier plan, ce qui en général le *tue* ;
- `Ctrl-z` envoie le signal `SIGTSTP` (*terminal stop*) au processus au premier plan, ce qui en général le met en *pause*.

Pour le processus ayant reçu `SIGTSTP`, le shell imprime son *numéro de job*, suivi du signe `+` et de *l'état* de ce processus : il est *stoppé*.

Jobs

La commande interne `jobs` liste les jobs en cours pour le shell interactif. Les seules commandes que l'on pourra voir sont [3] :

- les commandes en arrière-plan ;
- les commandes stoppées (par exemple avec `Ctrl-z`) ;
- les commandes terminées (les processus morts) alors qu'elles étaient stoppées ou en arrière-plan et que le shell n'a pas encore rapportées comme telles.

```
$ firefox& sleep 3&
[1] 18425
[2] 18426
$ sleep 100
^Z
[3]+  Stoppé                  sleep 100
$ jobs
[1]   En cours d'exécution    firefox &
[2]-  Fini                     sleep 3
[3]+  Stoppé                  sleep 100
```

3. si une commande s'exécute au premier plan, on ne peut pas lancer la commande `jobs`...

`bg` et `fg`

- La commande `bg` (*background*, arrière-plan) reprend l'exécution d'un job stoppé en le mettant en arrière-plan ;
- La commande `fg` (*foreground*, premier plan) reprend l'exécution d'un job stoppé en le mettant au premier plan.

Ces commandes prennent comme argument un numéro de job, précédé de `%`, ou bien sans argument s'appliquent au *job courant*, qui est le dernier job stoppé, s'il y en a, sinon le dernier job lancé en arrière-plan.

bg et fg

```
$ gedit
GRR j'ai oublié de le mettre en arrière-plan !!!!
^Z
[1]+  Stoppé                 gedit
$ bg
$ jobs
[1]+  En cours d'exécution   gedit &
$ sleep 100&
[2] 21089
$ jobs
[1]-  En cours d'exécution   gedit &
[2]+  En cours d'exécution   sleep 100 &
$ fg
sleep 100
^Z
$ bg %2
[2]+ sleep 100 &
```

`bg` et `fg`

Utilisation courante de `fg` :

- Exécution d'une commande qui prend toute la fenêtre du terminal, par exemple un éditeur visuel dans le terminal comme `vi`, `nano` ou `emacs -nw` ou bien les commandes `man`, `less`, ...
- Mise en pause de la commande avec `Ctrl-z` ; retour au terminal pour, par exemple, compiler un programme, essayer une commande, etc.
- Commande `fg` (sans argument) pour revenir au programme précédent (par exemple l'éditeur de texte), dans l'état où on l'avait laissé.

Commande `read`

Mots-clés `&&`, `||` et `!`

Exécution d'une commande *en arrière-plan*

Job control

Développements de variables modifiés

Supprimer un préfixe

```
$ foo=poisson-chat; printf '%s\n' "${foo#poisson-}"
chat
$ printf '%s\n' "${foo#*-}"
chat
$ printf '%s\n' "${foo#*o}"
isson-chat
$ printf '%s\n' "${foo##*o}"
n-chat
$ printf '%s\n' "${foo#*}" # ne sert à rien
poisson-chat
$ printf '%s\n' "${foo##*}"
$
$ bar=pois
$ printf '%s\n' "${foo#$bar}"
son-chat
$ printf '%s\n' "${foo#bar}"
poisson-chat
```

Supprimer un préfixe

```
${parameter#word} # supprimer le plus petit préfixe
${parameter##word} # supprimer le plus grand préfixe
```

- D'abord `word` subit le développement du tilde, les substitutions de commande, les développements de variables et arithmétiques ainsi que la suppression des caractères inhibiteurs. Ceci est valable pour toute cette section, dès que la valeur de `word` est nécessaire (ce qui est le cas ici).
- Puis `word` est interprété comme un motif shell.
- Avec un seul croisillon (`#`), le plus petit préfixe qui correspond au motif shell est supprimé du développement de la variable `parameter`.
- Avec deux croisillons (`##`), c'est le plus grand préfixe qui correspond au motif shell qui est supprimé.

Supprimer un suffixe

```
${parameter%word} # supprimer le plus petit suffixe
${parameter%%word} # supprimer le plus grand suffixe
```

```
$ foo=poisson-chat; printf '%s\n' "${foo%-chat}"
poisson
$ printf '%s\n' "${foo%-*}"
poisson
$ printf '%s\n' "${foo%o*}"
poiss
$ printf '%s\n' "${foo%%o*}"
p
$ printf '%s\n' "${foo%${foo#????}}"
pois
```

- Supprimer un suffixe ou un préfixe est très utile !

  ```
  $ for f in /usr/include/*; do printf '%s' "${f%.*}"; done
  ```

- ```
  $ foo=/home/bob/hello.c
  $ printf '%s\n' "${foo##*/}" "${foo##*.}" "${foo%.*}
  hello.c
  c
  /home/bob/hello
  ```
  ```

Valeurs par défaut

`${parameter:-word}`
`${parameter-word}`

```
$ foo=bar; echo ${foo:-lol}
bar
$ unset foo; echo ${foo:-lol}
lol
$ foo=; echo ${foo:-lol}
lol
$ foo=; echo ${foo-lol}

$ unset foo; echo ${foo-lol}
lol
```

- le développement de `${parameter:-word}` donne `word` si la variable `parameter` est non définie ou vide
- le développement de `${parameter-word}` donne `word` si la variable `parameter` est non définie

Valeurs par défaut avec affectation

`${parameter:=word}`
`${parameter=word}`

```
$ foo=bar; echo ${foo:=lol} $foo
bar bar
$ unset foo; echo ${foo:=lol} $foo
lol lol
$ foo=; echo ${foo:=lol} $foo
lol lol
$ foo=; echo ${foo=lol} $foo

$ unset foo; echo ${foo=lol} $foo
lol lol
```

- Équivalent de `:-` et `-`, mais en plus, `word` est affecté à `parameter` si elle est non définie ou,
- juste pour la version `:=`, si elle est vide.
- Mais, et c'est spécifique à `:=` et `=`, pas possible avec les paramètres positionnels.

Erreur si non définie (ou vide)

```
${parameter:?word}
${parameter?word}
```

- Dans les deux cas, si `parameter` est non définie, le message d'erreur `word` est écrit sur la sortie des erreurs (avec éventuellement d'autres renseignements),
- idem dans le cas `:?` si `parameter` est vide,
- et dans les deux cas, si dans un script, fin du script, avec statut de sortie non nul (1 sur les shells testés).
- ```
  #!/bin/sh
  # interrogation : version nulle de ls et du
  ls "${1?argument manquant}"
  du -sh "$1"
  ```
- ```
  $ ./interrogation
  ./interrogation: ligne 5: 1: argument manquant
  $ echo $?
  1
  ```