INSTITUT Mines-Télécom

TELECOM SudParis

Le shell bash

CSC3102 - Introduction aux systèmes d'exploitation

Élisabeth Brunet et Gaël Thomas

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Le terminal

- Porte d'entrée d'un ordinateur

- Un terminal offre :
  - un canal pour entrer des données (clavier, souris, écran tactile…)
  - un canal pour afficher des données (écran, imprimante, haut-parleur…)

Le terminal

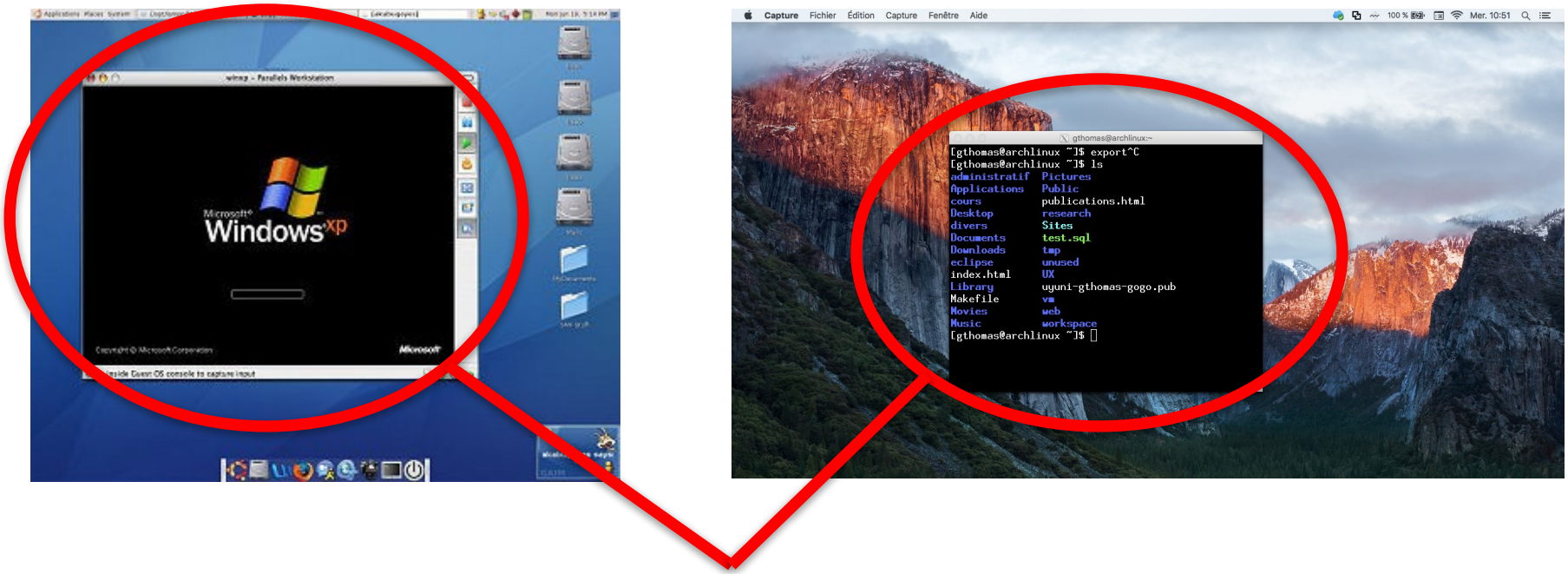
Un ordinateur n'a pas toujours un terminal intégré

Bien que ce soit souvent le cas

(smartphone, tablette, ordinateur portable…)

Un terminal peut être virtualisé

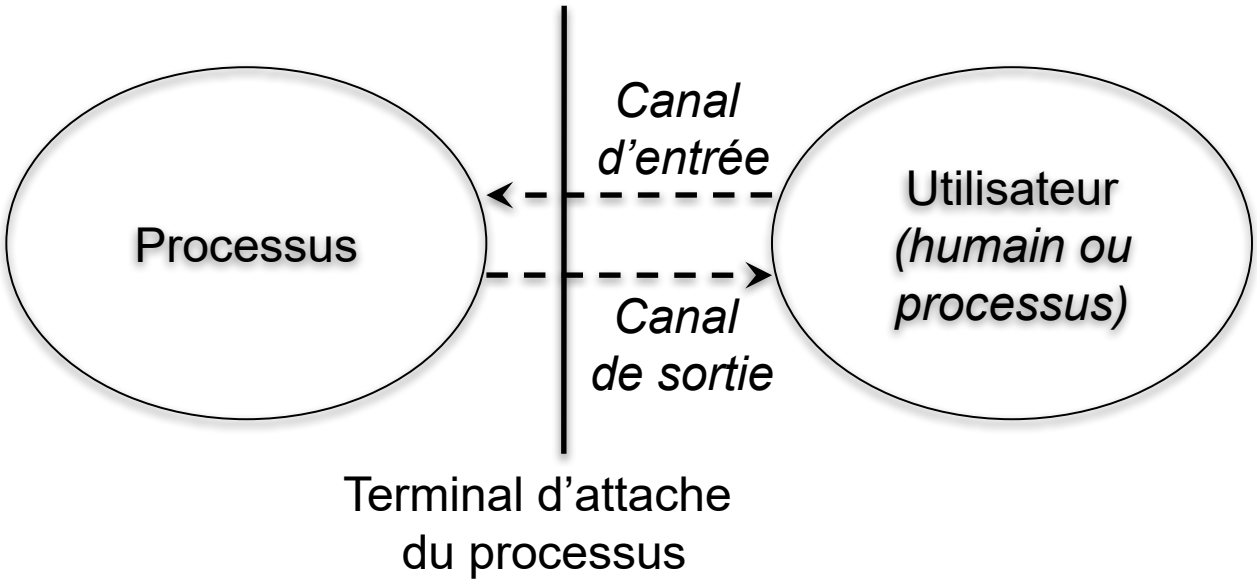
- Un terminal virtuel émule le comportement d'un terminal physique dans un autre terminal (virtuel ou physique)



Terminaux virtuels

Un processus communique avec l'utilisateur via un terminal

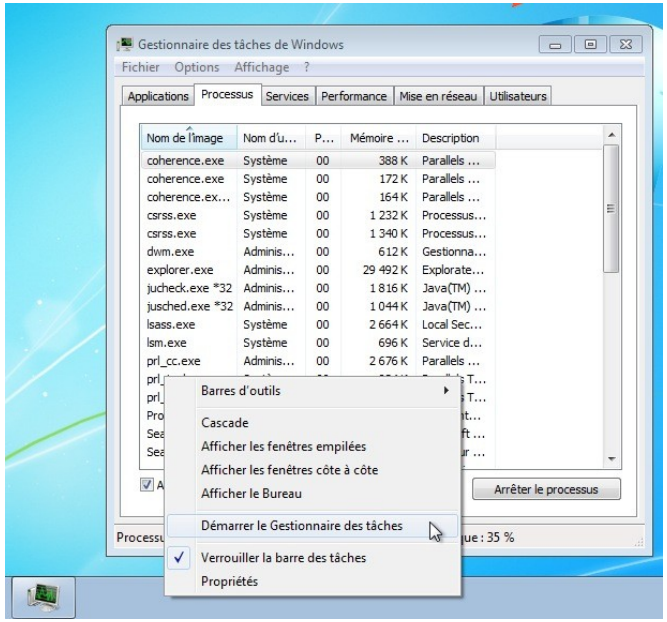
- On dit que le processus est attaché à un (et un seul) terminal



Remarque : lorsqu'un terminal est fermé,
tous les processus attachés au terminal sont détruits

Le shell

Le shell est un programme permettant d'interagir avec les services fournis par un système d'exploitation



Shell en mode graphique
(Bureau windows, X-windows…)

```
[gthomas@archlinux ~]$ ls
administratif  Pictures
Applications   Public
cours          publications.html
Desktop        research
divers         Sites
Documents      test.sql
Downloads      tmp
eclipse        unused
index.html     UX
Library        uyuni-gthomas-gogo.pub
Makefile       vm
Movies         web
Music          workspace
[gthomas@archlinux ~]$
```

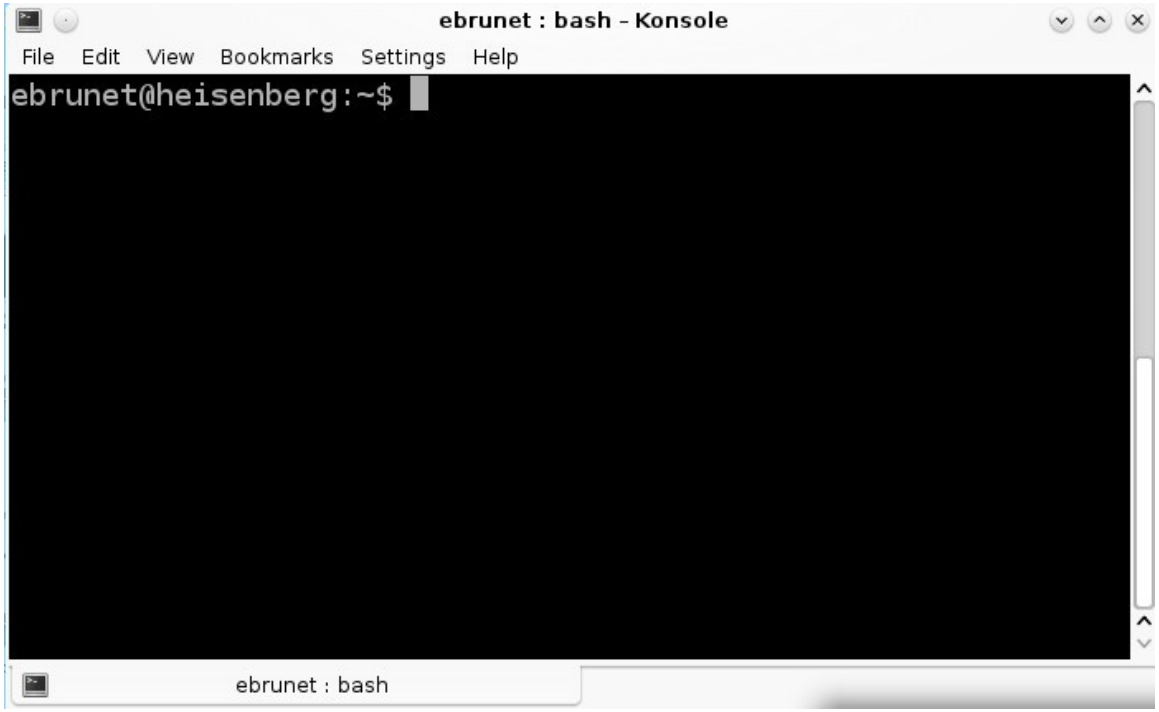
Shell en mode texte
(bash, tcsh, zsh, ksh, cmd.exe…)

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Le Bourne-Again Shell (`bash`)

- Dans ce cours, nous étudions le shell en **mode texte** `bash`
  - En mode texte car permet d'écrire des scripts !

- Attaché à un terminal virtuel en mode texte



Remarque importante

Dans la suite du cours, nous utiliserons souvent le terme « shell » pour désigner le « Bourne-Again shell »

Mais n'oubliez pas que `bash` n'est qu'un shell parmi de nombreux autres shells (`bash`, `tcsh`, `zsh`, `ksh`, `cmd.exe`...)

Bash

- Interpréteur de commandes
  - Lit des commandes (à partir du terminal ou d'un fichier)
  - Exécute les commandes
  - Écrit les résultats sur son terminal d'attache

- `Bash` définit un langage, appelé le langage bash
  - Structures de contrôle classiques
    (if, while, for, etc.)
  - Variables

- Accès rapide aux mécanismes offert par le noyau du système d'exploitation (tube, fichiers, redirections, …)

Un texte `bash`

- Un **texte** est formé de **mots `bash`**

- Un **mot `bash`** est
  - Formé de **caractères** séparés par des **délimiteurs** (délimiteurs : espace, tabulation, retour à la ligne)
    Exemple : `Coucou=42!*` est un unique mot
  - Exceptions :
    - `;` `&` `&&` `|` `||` `(` `)` `` ` `` sont des mots ne nécessitant pas de délimiteurs
    - Si une chaîne de caractères est entourée de `""` ou `''`, bash considère un unique mot

`bash` est sensible à la casse (c.-à-d., minuscule ≠ majuscule)

Un texte bash

- Un **texte** est formé de **mots**

```
Ici    nous    avons   5         mots

" En bash, ceci est un unique "mot" y compris mot milieu"

Voici, trois, mots

" zip "@é§èçà°-_¨^$%ù£,.:+=' est un autre unique mot'

Nous|avons;NEUF&&mots&ici
```

Un texte `bash`

- Un **texte** est formé de **mots**

Attention :

Ce n'est pas parce qu'on écrit des mots que ces mots ont un sens pour bash

Exemple : `echo yop!3:bip` est constitué de deux mots mais n'est pas compréhensible par `bash`

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2…` **`cmd`** `arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$
```

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo Salut tout le monde
```

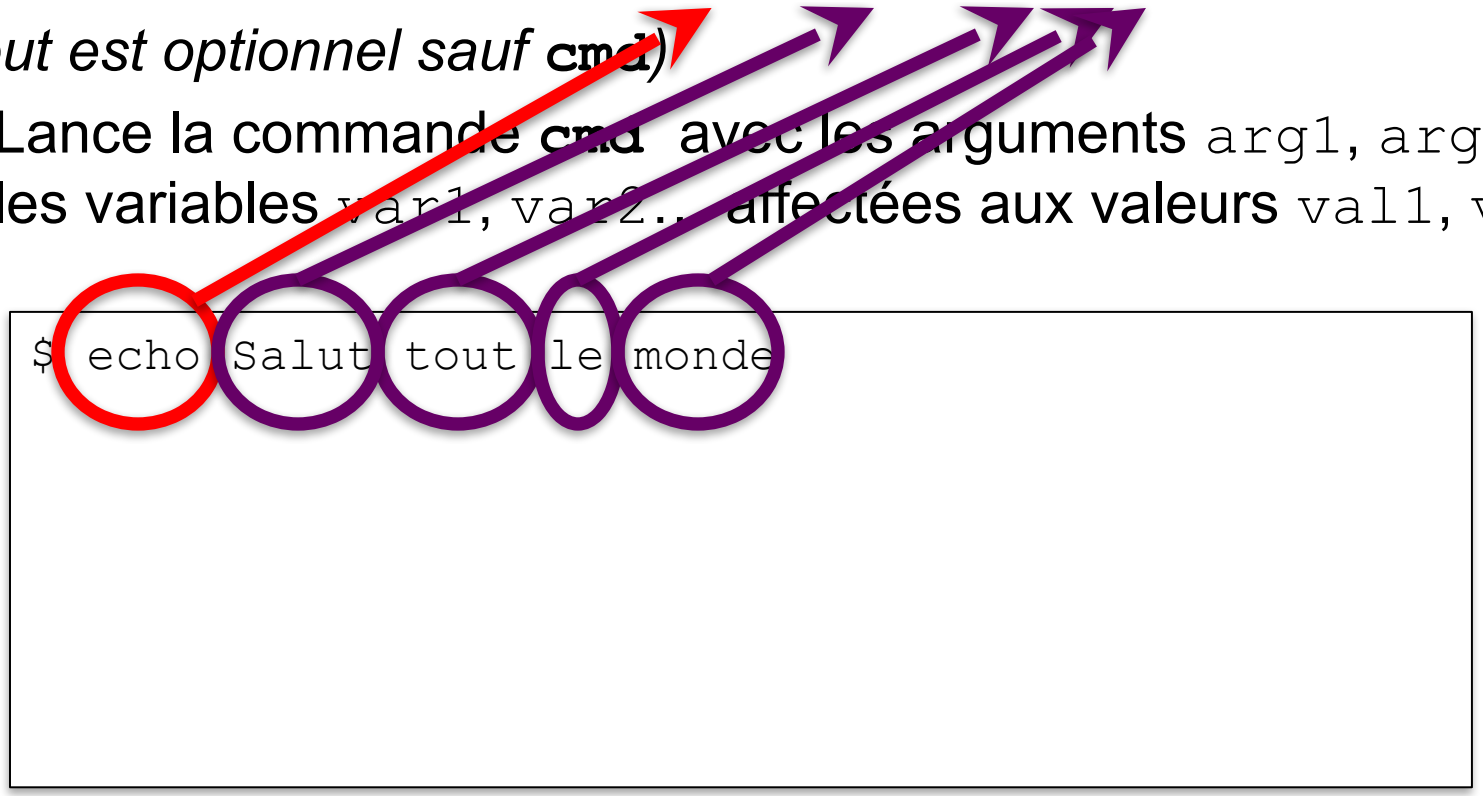
Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…



Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo Salut tout le monde
Salut tout le monde
```

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo "Salut   tout le   monde"
```

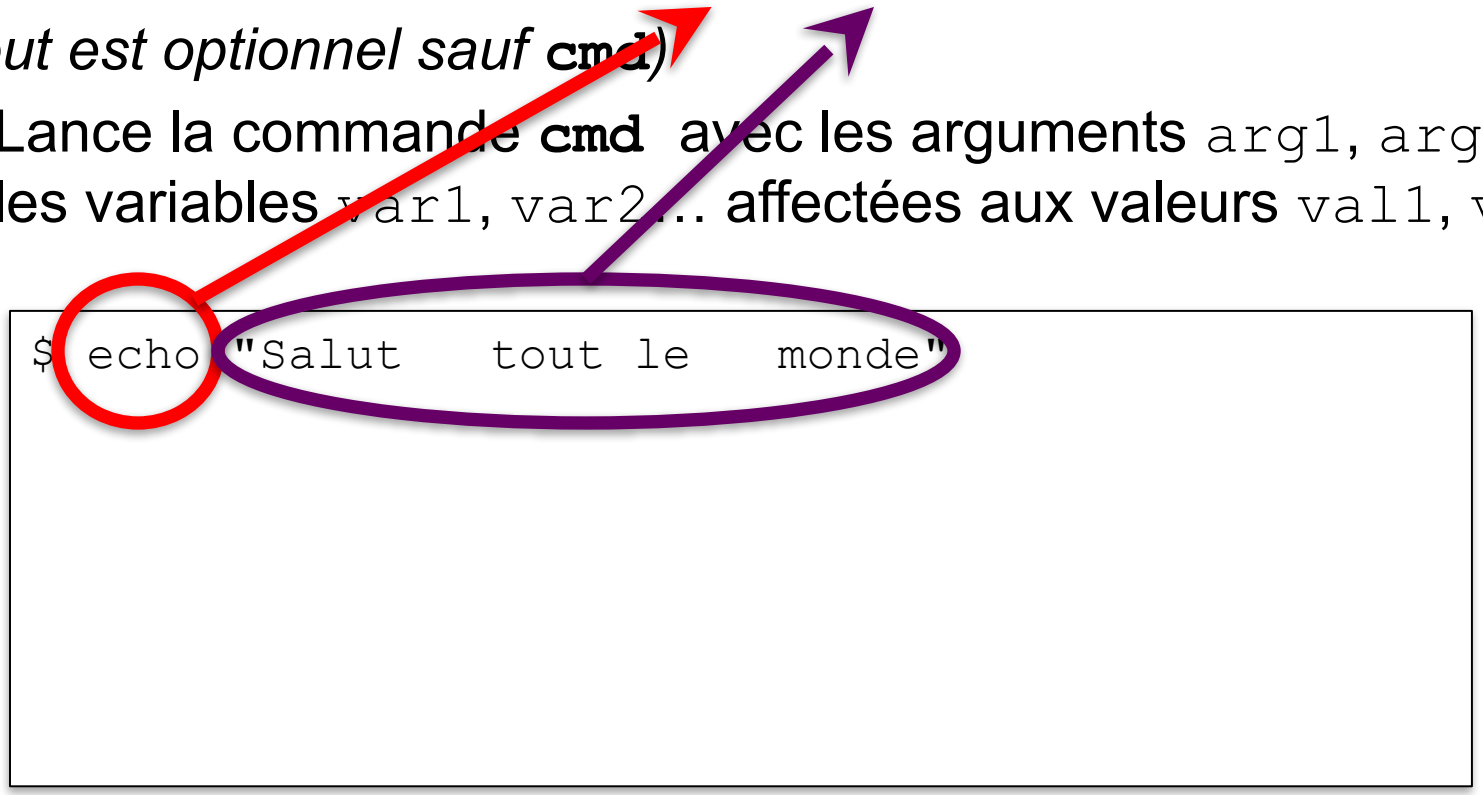
Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…



Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo "Salut   tout le   monde"
Salut   tout le   monde
```

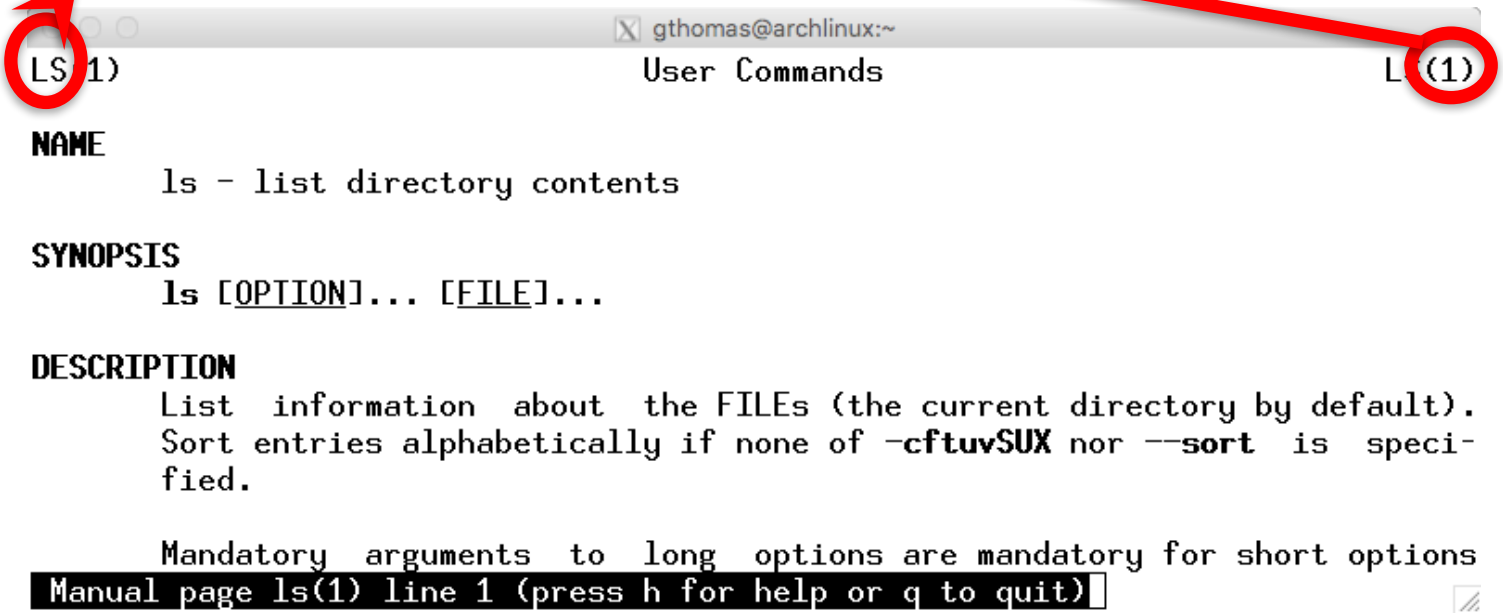
La première commande à connaître

- `man 1 cmd`
  - `man` pour manuel : donne de l'aide
  - `1` (optionnel) indique la section d'aide de la commande
    - `1` : commandes
  - `cmd` est la commande dont on veut consulter le manuel

```
$ man ls
```

La première commande à connaître

- `man 1 cmd`
  - `man` pour manuel : donne de l'aide
  - `1` (optionnel) indique la section d'aide de la commande
    - `1` : commandes
  - `cmd` est la commande dont on veut consulter le manuel



Caractères spéciaux de `bash`

- Caractères spéciaux
  - `\ ' ` " > < $ # * ~ ? ; ( ) { }`
    (`'` est appelé quote ou apostrophe
    alors que `` ` `` est appelé antiquote ou accent grave)
  - Explication de chacun donnée dans la suite du cours

- Désactiver l'interprétation des caractères spéciaux
  - `\` désactive l'interprétation spéciale du caractère suivant
  - `'…'` ⇒ désactive l'interprétation dans toute la chaîne
  - `"…"` ⇒ seuls sont interprétés les caractères $ \ ` (accent grave)

Script `bash`

- Programme `bash` = texte `bash` dans un fichier texte
  - Interprétable par `bash` au lancement par l'utilisateur
  - Modifiable par un éditeur de texte (p. ex. `emacs`, `vi`, mais pas `word` !)
  - Un programme bash doit être rendu exécutable avec :

    `chmod u+x mon_script.sh`
    (notion vue dans le CI2 sur le système de fichiers)
  - Par convention, les noms de script sont suffixés par l'extension « .sh »
    - p. ex., `mon_script.sh`

- Invocation du script nommé `mon_script.sh` avec
  - `./mon_script.sh`
  - Avec ses arguments :

    `./mon_script.sh  arg1 arg2`

`./` indique que le script se trouve dans le répertoire courant (notion vue dans le CI2)

Structure d'un script bash

- Première ligne : `#! /bin/bash`
  - `#!` : indique au système que ce fichier est un ensemble de commandes à exécuter par l'interpréteur dont le chemin suit
    - par exemple : `/bin/sh`, `/usr/bin/perl`, `/bin/awk`, etc.
  - `/bin/bash` lance `bash`

- Puis séquence structurée de commandes shell

```
#! /bin/bash

commande1
commande2
...
```

mon_script.sh

- Sortie implicite du script à la fin du fichier
  - Sortie explicite avec la commande `exit`

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Variables `bash`

- Déclaration/affectation avec = (exemple `ma_var=valeur`)
- Consultation en préfixant du caractère `$` (exemple `$ma_var`)
- Saisie interactive : `read var1 var2 ... varn`
  - Lecture d'une ligne saisie par l'utilisateur (jusqu'au retour chariot)
  - Le premier mot va dans `var1`
  - Le second dans `var2`
  - Tous les mots restants vont dans `varn`

Variables `bash`

- Déclaration/aff
- Co
- Sa
  -
  -
  - L
  - Tous les mots restants vont dans `varn`

Attention :

Pas de blanc dans `ma_var=valeur`
Pas de blanc dans `$ma_var`

(dans les deux cas, `bash` interprète de façon spéciale un unique mot)

r chariot)

Variables `bash` - exemple

```
$
```

Variables `bash` - exemple

```
$ a=42
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$
```

Saisi par l'utilisateur

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$
```

Saisi par l'utilisateur

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$ echo $x
Ceci
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$ echo $x
Ceci
$ echo $y
est une phrase
```

Premier mot

Tous les mots qui suivent

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Schéma algorithmique séquentiel

- Suite de commandes les unes après les autres
  - Sur des lignes séparées
  - Sur une même ligne en utilisant le caractère point virgule (`;`) pour séparateur

Schéma alternatif (`if`)

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

Conditions de test

- Tests sur des valeurs numériques
  - `[ n1 -eq n2 ]` : vrai si `n1` est égal à `n2`
  - `[ n1 -ne n2 ]` : vrai si `n1` est différent de `n2`
  - `[ n1 -gt n2 ]` : vrai si `n1` supérieur strictement à `n2`
  - `[ n1 -ge n2 ]` : vrai si `n1` supérieur ou égal à `n2`
  - `[ n1 -lt n2 ]` : vrai si `n1` inférieur strictement à `n2`
  - `[ n1 -le n2 ]` : vrai si `n1` est inférieur ou égal à `n2`
- Tests sur des chaînes de caractères
  - `[ mot1 = mot2 ]` : vrai si `mot1` est égale à `mot2`
  - `[ mot1 != mot2 ]`: vrai si `mot1` n'est pas égale à `mot2`
  - `[ -z mot ]` : vrai si `mot` est le mot vide
  - `[ -n mot ]` : vrai si `mot` n'est pas le mot vide

Conditions de test

- Tests sur des valeurs numériques
  - `[ n1 -eq n2 ]`
  - [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
- Tes[illegible]
  - [illegible]
  - `[ mot1 != mot2 ]`: vrai si `mot1` n'est pas égale à `mot2`
  - `[ -z mot ]` : vrai si `mot` est le mot vide
  - `[ -n mot ]` : vrai si `mot` n'est pas le mot vide

Attention :

Les blancs sont essentiels !

Remarque sur les conditions

- `[ cond ]` est un raccourci pour la commande **`test`** `cond`

- **`test`** est une commande renvoyant vrai (valeur 0) ou faux (valeur différente de 0) en fonction de l'expression qui la suit

```
if [ $x -eq 42 ]; then
   echo coucou
fi
```

Équivaut à

```
if test $x -eq 42; then
   echo coucou
fi
```

Schéma alternatif (`if`)

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

```
x=1
y=2
if  [ $x -eq $y ]; then
   echo "$x = $y"
elif [ $x -ge $y ]; then
   echo "$x >= $y"
else
  echo "$x < $y"
fi
```

Schéma alternatif (case)

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
case mot in
   motif1)
    …;;
   motif2)
    …;;
   *)
     …;;
esac
```

- Schéma alternatif multiple
  - Si `mot` vaut `motif1` …
    Sinon si `mot` vaut `motif2` …
    Sinon …
  - Motif : chaîne de caractères pouvant utiliser des méta-caractères (voir CI3)
  - *) correspond au cas par défaut

Schéma alternatif (case)

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif multiple
  - Si `mot` vaut `motif1` …
    Sinon si `mot` vaut `motif2` …
    Sinon …
  - Motif : chaîne de caractères pouvant utiliser des méta-caractères (voir CI3)
  - *) correspond au cas par défaut

```
case mot in
   motif1)
    …;;
   motif2)
    …;;
   *)
     …;;
esac
```

```
res="fr"
case $res in
   "fr")
     echo "Bonjour";;
   "it")
     echo "Ciao";;
   *)
     echo "Hello";;
esac
```

Schémas itératifs

```
while cond; do
    cmds
done
```

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
   read x
   echo $x
done
```

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle
  - **for**
    - Pour chaque … dans … faire …
    - `var` correspond à la variable d'itération
    - `liste` : ensemble sur lequel `var` itère

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
    read x
    echo $x
done
```

```
for var in liste; do
    cmds
done
```

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle
  - **for**
    - Pour chaque … dans … faire …
    - `var` correspond à la variable d'itération
    - `liste` : ensemble sur lequel `var` itère

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
    read x
    echo $x
done
```

```
for var in liste; do
    cmds
done
```

```
for var in 1 2 3 4; do
    echo $var
done
```

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Arguments d'une commande

- `mon_script.sh arg1 arg2 arg3 arg4 …`
  ⇒ chaque mot est stocké dans une variable numérotée

mon_script.sh	arg1	arg2	arg3	arg4	…
"$0"	"$1"	"$2"	"$3"	"$4"	…

- `"$0"` : toujours le nom de la commande
- `"$1" … "$9"` : les paramètres de la commande
- `$#` : nombre de paramètres de la commande
- `"$@"` : liste des paramètres : `"arg1" "arg2" "arg3" "arg4" …`
- `shift` : décale d'un cran la liste des paramètres

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$./mon_echo.sh toto titi
toto
titi
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$./mon_echo.sh toto titi
toto
titi
$./mon_echo "fin de" la demo
fin de
la
demo
$
```

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$
```

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$ echo "Nous sommes le $(date). "
Nous sommes le lundi 27 juillet 2015, 12:47:06
(UTC+0200).
$
```

Rappel : avec "…", seuls sont interprétés les caractères $ \ `

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$ echo "Nous sommes le $(date). "
Nous sommes le lundi 27 juillet 2015, 12:47:06
(UTC+0200).
$ echo "Nous sommes le $date."
Nous sommes le .
$
```

Attention, récupère la variable date et non le résultat de la commande date

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Conclusion

- Concepts clés
  - Terminal, shell
  - Interpréteur de commande `bash`
    - Commandes, langage `bash`
  - Documentation
  - Caractères spéciaux de `bash`
  - Script `bash`
- Commandes clés
  - `man, bash, echo, read`
- Commandes à connaître
  - `date`

En route pour le TP !