ESIEE PARIS

Introduction à Unix

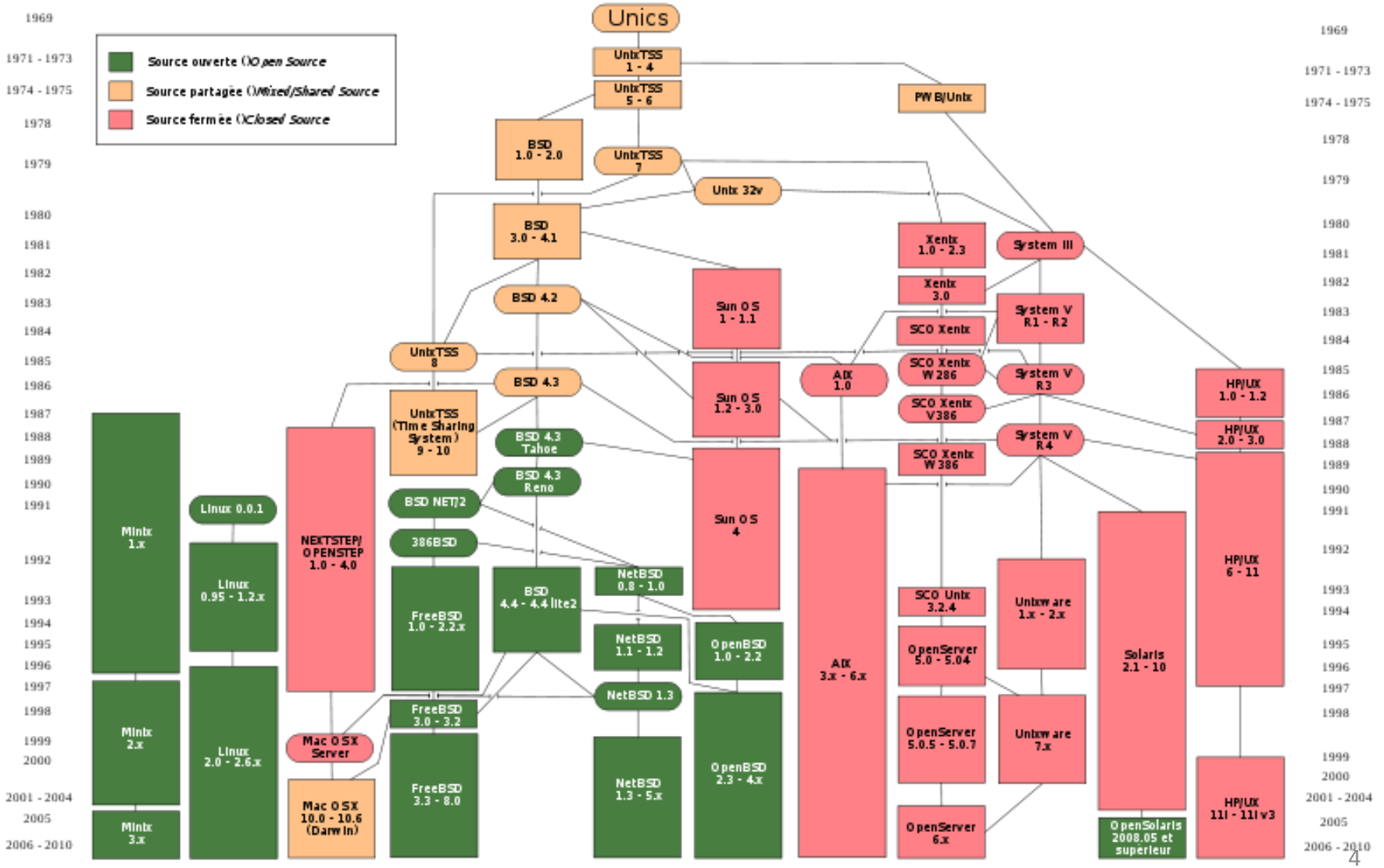
Benjamin Perret

1) Introduction générale

Qu'est qu'Unix

- Historiquement:
  - Système d'exploitation créé en 1969 chez Bell
- Aujourd'hui:
  - Famille de systèmes d'exploitation qui se conforment à la « *Single Unix Specification* » *(3700 pages de spécifications)*
- Quelques exemples: Linux, Mac OS X, *BSD,…

Une famille nombreuse

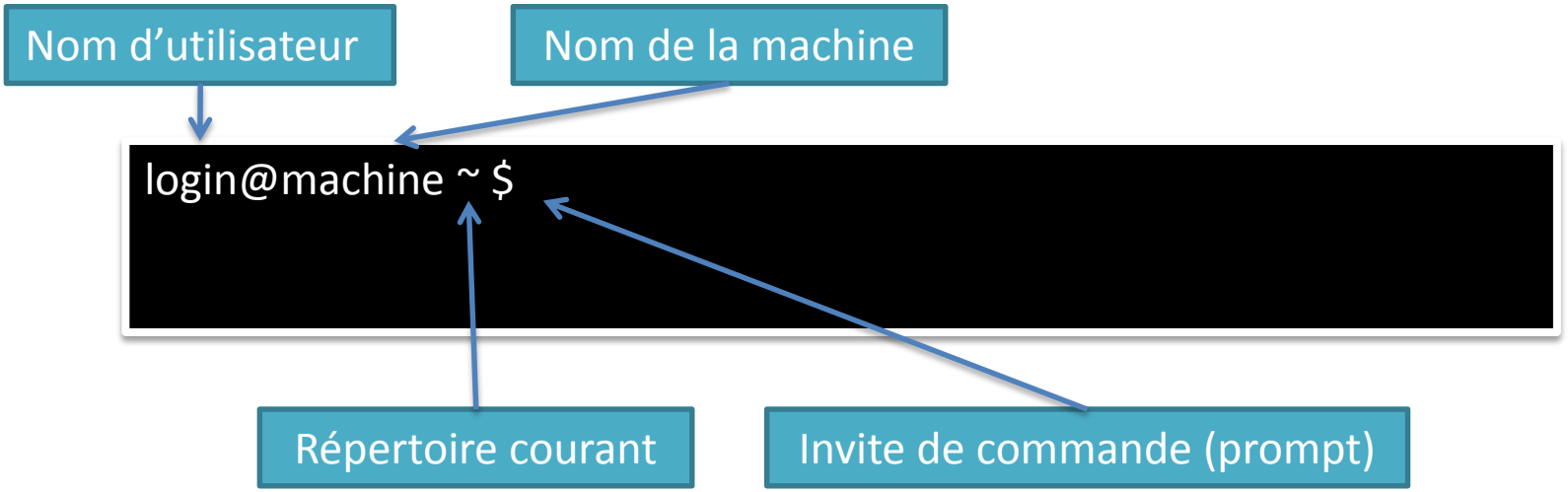


Philosophie
2 grands principes

- Une tâche ⇔ un outil
  - Grands nombres de fonctions/outils simples
  - Les opérations complexes sont obtenues par combinaison
- Tout est fichier
  - L'ensemble du système logiciel et matériel est accessible par manipulation de fichiers

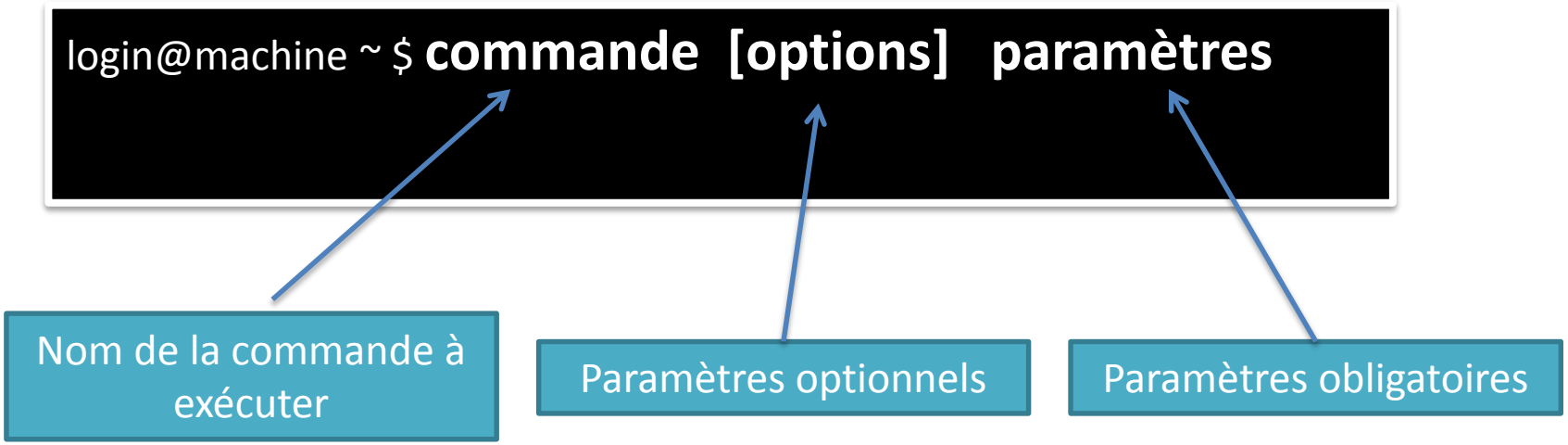
Le shell

- Interface utilisateur du système
- Autres noms : console, terminal
- Il existe différents shells
  - bash, sh, ksh, csh, zsh, ash,…
- Permet d'exécuter des commandes en mode texte



Commande

- Une commande consiste à appeler une fonction du système ou un programme
- Syntaxe générale :



- La commande est exécutée lors de l'appui sur la touche entrée (↵)

Commande

- Exemple: la commande « wc »
  - wc = « word count »
  - Prend en paramètre le nom d'un fichier texte
  - Compte le nombre de lignes, de mots et de caractères dans ce fichier.

```
login@machine ~ $ wc germinal.txt ↵
 18317  166516 1051650 germinal.txt
```

Il y a donc 18317 lignes, 166516 mots, 10… caractères dans le fichier germinal.txt

Commande

- Passage d'options
  - Si on ne souhaite connaitre que le nombre de mots on peut passer l'option –w à la commande wc

```
login@machine ~ $ wc -w germinal.txt ↵
166516 germinal.txt
```

Il y a donc 166516 mots dans le fichier germinal.txt

Le manuel (1)

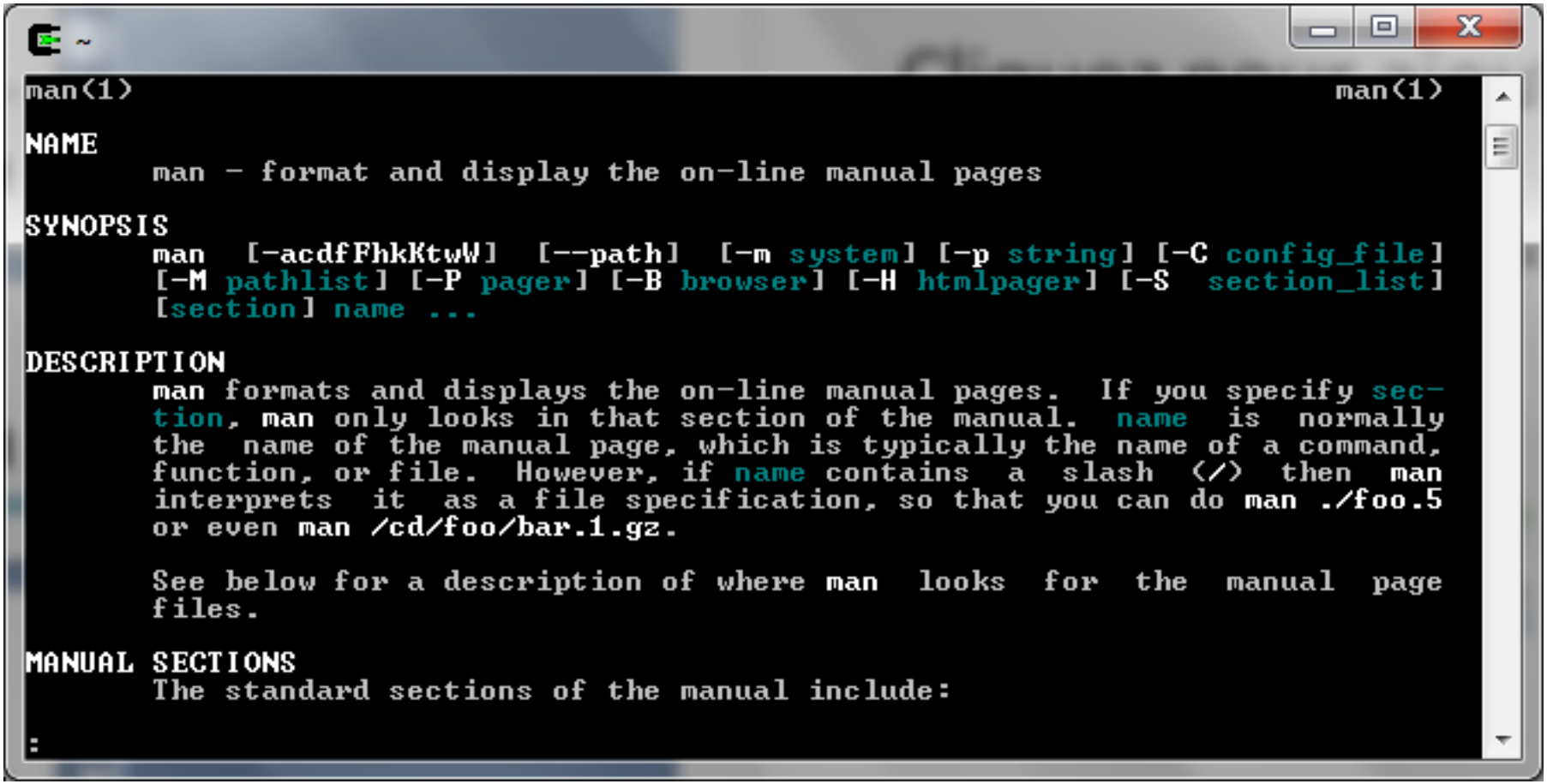
- Commande fondamentale :

  man [section] commande

- Obtenir de l'aide sur l'utilisation d'une commande
- Structurée en section (2 pour les appels systèmes, 3 pour la librairie C, …)
- Exemple: comment se servir du manuel:

```
login@machine ~ $ man man ↵
```

Le manuel (2)


~
man(1) man(1)
NAME
man - format and display the on-line manual pages
SYNOPSIS
man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file] [-M pathlist] [-P pager] [-B browser] [-H htmlpager] [-S section_list] [section] name ...
DESCRIPTION
man formats and displays the on-line manual pages. If you specify section, man only looks in that section of the manual. name is normally the name of the manual page, which is typically the name of a command, function, or file. However, if name contains a slash (/) then man interprets it as a file specification, so that you can do man ./foo.5 or even man /cd/foo/bar.1.gz.
See below for a description of where man looks for the manual page files.
MANUAL SECTIONS
The standard sections of the manual include:
:

Le manuel (3)

- L'aide est structurée en sections. Ex:
  - **Name:** nom de la commande et description rapide
  - **Synopsis:** résumé des options et paramètres
  - **Description:** description détaillée de la commande, des options et des paramètres
  - **Example:** exemples typiques d'utilisation
  - **See also:** autres commandes affiliées
- Pour naviguer dans l'aide: flèches haut et bas, barre d'espace (page suivante), q (quitter).

Le manuel (4)

- Recherche dans le manuel
  - Pour chercher un mot dans une page du manuel il utiliser le caractère / suivi du mot recherché et appuyer sur entrée
  - Le manuel vous amène la position de la première occurrence trouvée
  - Pour passer à l'occurrence suivante il faut taper / et appuyer sur entrée

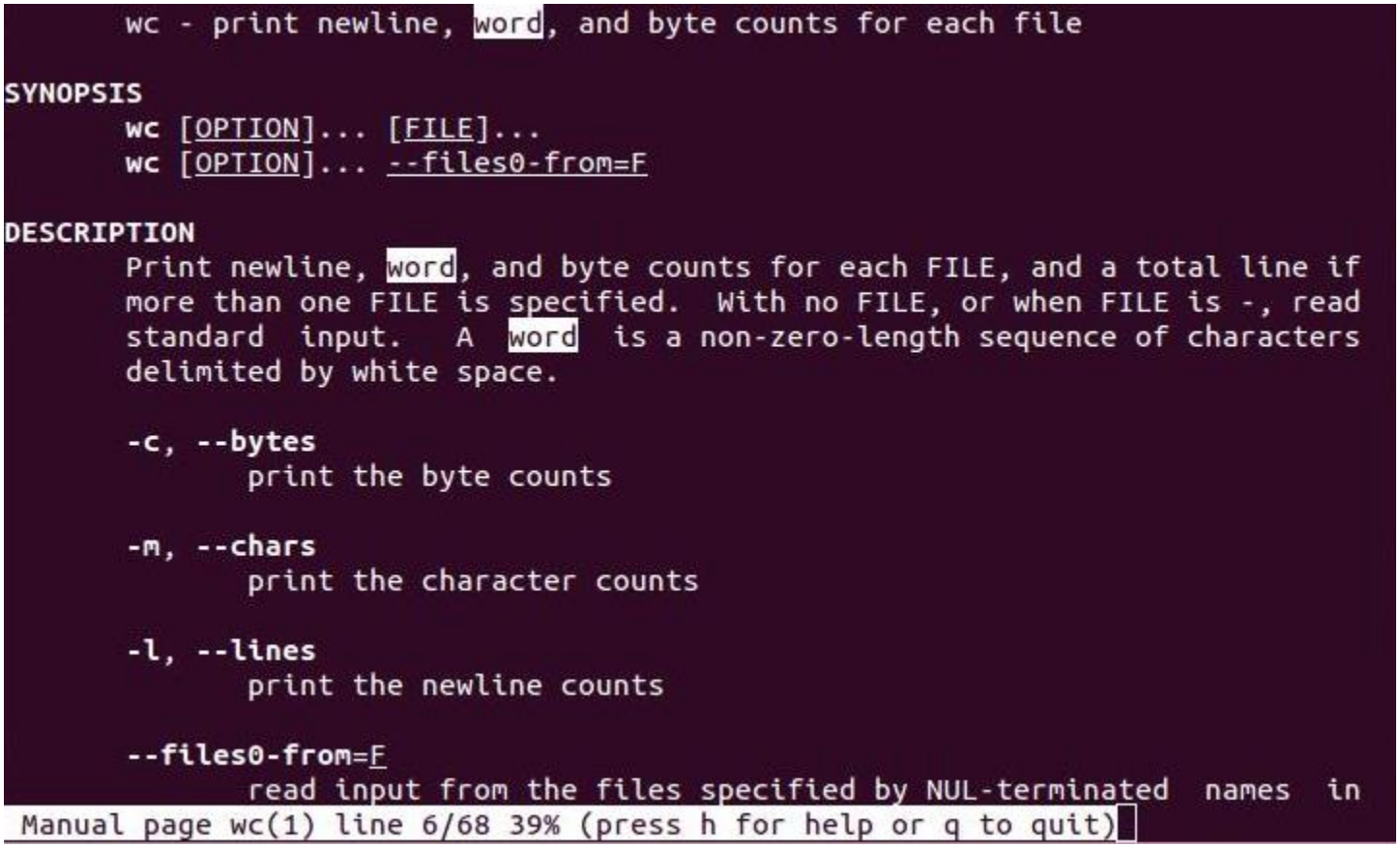
Le manuel (5)

- Exemple

```
login@machine ~ $ man wc ↵
WC(1)                    User Commands                    WC(1)

NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [OPTION]... [FILE]...
       wc [OPTION]... --files0-from=F

…
Manual page wc(1) line 1 (press h for help or q to quit)

/word↵
```

Le manuel (5)

- Exemple (suite)



Auto-complétion du shell

- Fonction très puissante (à utiliser absolument): touche Tabulation (⇆)
- Le shell va chercher à compléter automatiquement le mot que vous êtes en train de taper
- Pour cela il cherche les différentes possibilités dans la liste des fichiers et des commandes accessibles

Historique

- Permet de naviguer dans la liste des commandes déjà utilisées
- Les touches haut (↑) et bas (↓) reviennent à la commande précédente et suivante
- La commande « history » affiche la liste des commandes dernièrement utilisées
- La commande « !n » rappelle la n-ième commande utilisée (en comptant depuis la dernière)
- La commande « !abc » rappelle la dernière commande utilisée commençant par abc.

2) Système de fichier

Système de fichier

- Un fichier est un ensemble de données désigné par un nom
- Le système de fichier désigne l'ensemble des logiciels qui permettent de gérer les fichiers accessibles sur une machine.
- Sous Unix les systèmes de fichiers sont hiérarchiques:
  - Les dossiers sont des fichiers spéciaux qui contiennent d'autres fichiers

Chemin absolu

- Un fichier est désigné par son chemin d'accès:

```
/users/info/i3m/etudiant1/cv.txt
```

  - Le premier **/** désigne le répertoire racine (chemin absolu)
  - Puis on rentre dans le répertoire **users** qui se trouve dans **/**
  - Puis on rentre dans le répertoire **info** qui se trouve dans **users**
  - **…**
  - On désigne finalement le fichier **cv.txt** du répertoire **etudiant1**

•Le **/** tout à gauche désigne la racine
•Le **/** entre deux noms de répertoires désigne la relation « ... est le répertoire père de ... »

Chemin relatif (1)

- Quelques répertoires particuliers:

Symbole	Description
.	Répertoire courant (celui dans lequel le shell est en train de travailler)
..	Répertoire parent du répertoire courant
~	Répertoire personnel de l'utilisateur courant
~login	Répertoire personnel de l'utilisateur « login »

Chemin relatif (2)

- Permet de raccourcir les chemins d'accès: (on parle de chemin relatif)
  - Par exemple si etudiant1 lance un terminal

  - Le répertoire de travail courant est **~**,donc le répertoire personnel de etudiant1
  - Tous les chemins suivant désignent le même fichier:
    - cv.txt
    - ./cv.txt
    - ~/cv.txt
    - ../etudiant1/cv.txt
    - ./../../cv.txt

Naviguer dans le système de fichier

- Les commandes fondamentales sont :
  - **pwd** : donne le chemin absolu du répertoire courant

```
Benjamin@Perseus ~ $ pwd
/home/Benjamin
```

  - **ls** : liste les fichiers contenu dans le répertoire courant

```
Benjamin@Perseus / $ ls
Cygwin.bat  Cygwin.ico  bin  cygdrive  dev  etc  home  lib  proc  tmp  usr  var
```

  - **cd [nom_de_repertoire]:** change le répertoire courant

```
Benjamin@Perseus / $ cd home
Benjamin@Perseus /home $ cd ..
Benjamin@Perseus /
```

Si aucun répertoire n'est précisé, cd vous ramène dans votre répertoire personnel ($cd ⇔ $cd ~)

Manipuler les fichiers

- **cp [-r] origine destination:** copie le fichier « origine » dans « destination »
  - si « origine » est un répertoire l'option –r permet de faire une copie récursive (copie du répertoire et de tout son contenu)
- **mv origine destination:** déplace origine dans destination (util également pour renommer un fichier)
- **rm [-r] fichier [fichiers,…]**: supprime le fichier ou les fichiers désignés, l'option –r permet de supprimer un répertoire et son contenu.
  - Attention **rm *** supprime tous les fichiers du répertoire courant
- **rmdir répertoire**: supprime le répertoire (doit être vide)

Manipuler les fichiers

- **mkdir répertoire:** créé un nouveau répertoire
- **touch fichier:** si fichier existe: modifie sa date de dernière modification, sinon le fichier est créé
- **more fichier**: affiche le contenu d'un fichier page par page
- **less fichier**: affiche le contenu d'un fichier (plus souple que more)
- **head fichier** ou **tail fichier :** affichent respectivement le début et la fin d'un fichier

Caractères Joker

- Le caractère * (wildcard) désigne n'importe quelle chaine de caractères
  - *.txt désigne tous les fichiers du répertoire courant se terminant par .txt
  - * désigne tous les fichiers du répertoires courant
- Le caractère ? désigne n'importe quel caractère
  - fichier?.txt peut désigner fichier1.txt, fichier2.txt, fichiera.txt, …

Trouver des fichiers

- **find location –name pattern :** recherche les fichiers dont le nom correspond au motif donné dans le répertoire location

```
Benjamin@Perseus ~ $ find / -name .bashrc
/etc/defaults/etc/skel/.bashrc
/etc/skel/.bashrc
/home/Ben/.bashrc
```

```
Ben@Perseus ~ $ find / -name *.txt
/usr/share/doc/common-licenses/ReadMe.txt
/usr/share/doc/dos2unix/BUGS.txt
/usr/share/doc/dos2unix/ChangeLog.txt
…
```

Find est une commande complexe qui propose énormément d'options !

3) Utilisateurs et droits

Utilisateurs

- Unix est un système multi-utilisateurs
- Chaque utilisateur doit s'identifier pour pouvoir accéder à la machine, il est désigné par un numéro unique, l'UID (user identifier)
- Les utilisateurs sont regroupés dans des groupes identifiés par des numéros uniques, les GID (Group Identifier)
- Chaque ressource (et donc fichier) appartient à un utilisateur et un ou plusieurs groupes
- Les commandes que vous lancez hérite de vos droits

Utilisateurs : commandes

- whoami : donne votre login

```
Benjamin@Perseus ~ $ whoami
Benjamin
```

- who : donne la liste des utilisateurs connectés
- passwd : change votre mot de passe
- exit : quitte la console
- su [utilisateur] : change d'utilisateur (root si aucun utilisateur n'est spécifié)
- sudo [utilisateur] commande : exécute une commande avec les droits d'un autre utilisateur

Gestion des droits

- Sous Unix le monde est divisé en trois classes d'utilisateurs:
  - user (u) : le propriétaire du fichier
  - group (g) : le groupe (ou les groupes) associés au fichier
  - others (o) : le reste du monde
- Il est possible de définir des droits d'accès différents pour chacune de ces 3 catégories.
- L'administrateur (généralement root) conserve le droit de faire ce qu'il veut…

Gestion des droits

- Comment voir les droits associés à un fichier

```
Benjamin@Perseus / $ ls -l
-rwxr-xr-x  1 root root   57 Jun 28 15:18 Cygwin.bat
-rw-r--r--  1 root root 7022 Jun 28 15:18 Cygwin.ico
drwxr-xr-x  1 root root    0 Jun 28 15:16 bin
dr-xr-xr-x  4 root root    0 Sep  2 16:01 cygdrive
 …
```

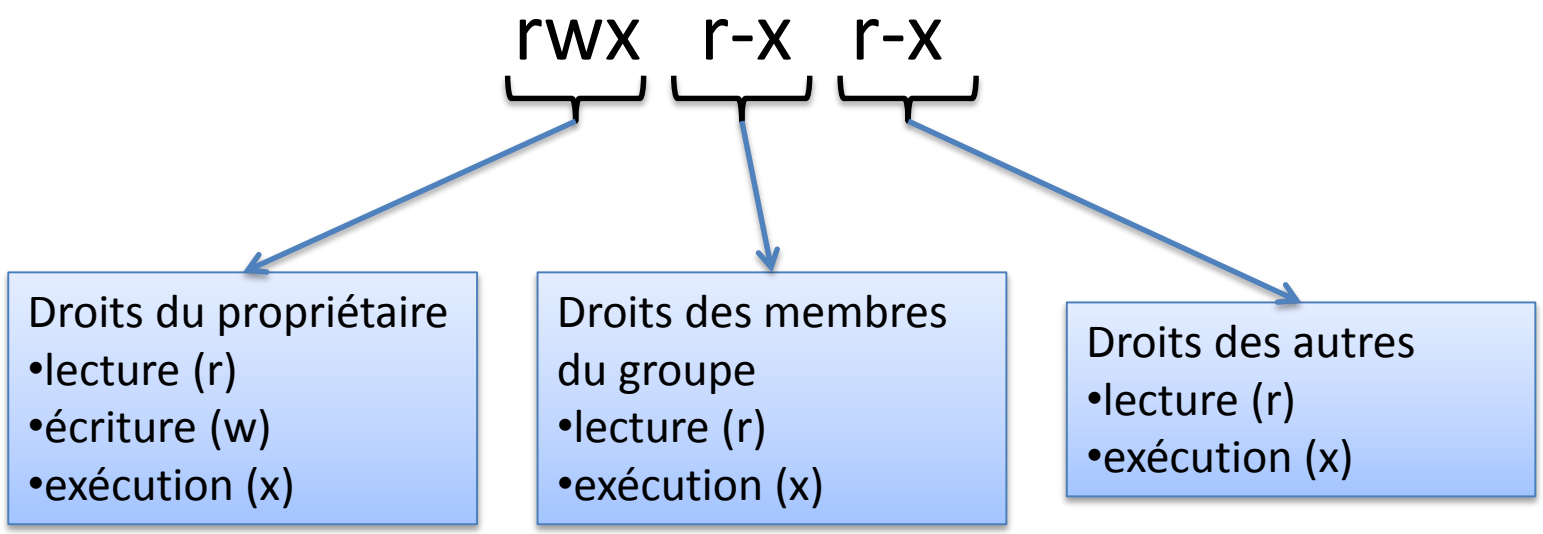
- Chaque ligne correspond à un fichier:
  - le premier caractère donne son type (- pour fichier « normal », d pour un répertoire, …)
  - les 9 caractères suivants donnent les droits
  - nombre de liens pointant sur ce fichier
  - nom du propriétaire et du groupe
  - taille en octet
  - date de dernière modification
  - nom du fichier

Gestion des droits sur les fichiers

- Cas d'un fichier « normal »

```
-rwxr-xr-x  1 root root   57 Jun 28 15:18 Cygwin.bat
```

- Les droits associés à ce fichier sont



Gestion des droits sur les fichiers

- Cas d'un répertoire

```
drwxr-xr-x  1 root root    0 Jun 28 15:16 bin
```

- La sémantique de r, w et x est modifiée
  - r est le droit de lister les fichiers contenu dans le répertoire
  - w est le droit de créer un nouveau fichier dans le répertoire
  - x est le droit de traverser le dossier
- Le droit x permet d'autoriser un utilisateur à accéder à un fichier d'un répertoire sans pour autant le laisser voir le contenu du répertoire

Gestion des droits sur les fichiers

- Intérêt du droit de traverser

```
Benjamin@Perseus /home $ ls -l
drwx-----x 1 Benjamin Benjamin 0 Sep  2 16:27 Benjamin
Benjamin@Perseus /home $ cd Benjamin
Benjamin@Perseus ~ $ ls -l
drwx---r-x 1 Benjamin Benjamin 0 Sep  2 16:27 public_html
```

- Mon serveur Web préféré ne pourra pas lire le contenu de mon répertoire personnel mais il pourra accéder au sous-répertoire public_html qui contient ma page Web personnelle

```
apache@Perseus /home $ cd Benjamin
cd Benjamin: Permission denied!
apache@Perseus /home $ cd Benjamin/public_html/
apache@Perseus /home/Benjamin/public_html $
```

Modification des droits

- **chown utilisateur fichier** : change le propriétaire du fichier

  ! On ne peut pas reprendre un fichier qui a été donné à un autre utilisateur

- **chgrp groupe fichier** : change le groupe du fichier

Modification des droits

- **chmod droits fichiers** : change les droits associés aux fichiers.
- 2 façon de spécifier les droits

Droits	Binaire	Décimal
rwx	111	7
rw-	110	6
r-x	101	5
r--	100	4
-wx	011	3
-w-	010	2
--x	001	1
---	000	0

Pour donner les droits rwxr-x--- sur le fichier cv.txt

```
Benjamin@Perseus ~ $ chmod 750 cv.txt
```

Modification des droits

- 2e approche, plus simple mais moins efficace
  - quels groupes d'utilisateurs sont affectés par le changement de droits (**u** pour le propriétaire, **g** pour le groupe, **o** pour les autres et **a** pour tout le monde)
  - veut-on ajouter (+) ou enlever (-) des droits
  - quels droits sont concernés (r, w ,x)
- Par exemple pour autoriser le groupe et les autres à lire et modifier le fichier cv.txt

```
Benjamin@Perseus ~ $ chmod go+rw cv.txt
```

4) Processus et Jobs

Processus

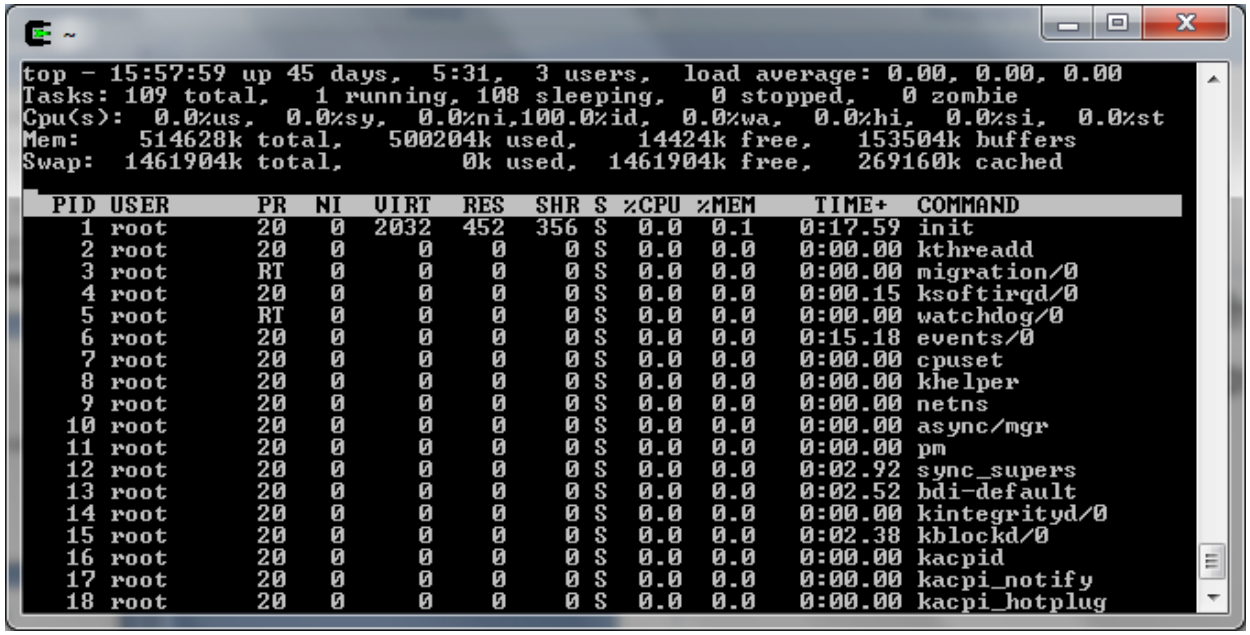
- Un processus est un programme en cours d'exécution
- Unix est un système multi-tâches, de nombreux processus peuvent s'exécuter simultanément (ou presque)
- Chaque processus est identifié par un numéro unique le PID (Processus Identifier)
- Chaque processus est lié au processus qui l'a lancé: le processus père identifié par le PPID

Arrière plan, avant plan

- Par défaut une commande lance un processus d'avant plan : le shell se bloque jusqu'à la fin du processus
- On peut tuer un processus d'avant plan en appuyant sur **Ctrl+C**
- On peut lancer un processus en arrière plan en ajoutant **&** à la fin de la commande, le shell reste alors accessible même si le processus n'est pas terminé
- En cas d'oubli de **&**, on peut obtenir le même résultat avec la manipulation suivante:
  - **Ctrl+Z** suspend le processus de premier plan
  - la commande **bg** relance les processus suspendus et le fait passé en arrière plan

Gestion des processus

- La commande **top** affiche la liste des processus actif en temps réel:



- Les infos sont (entre autres): pid, propriétaire, priorité, occupation CPU, mémoire, temps d'exécution, commande utilisée

Gestion des processus

- La commande **ps** affiche un instantané des processus
- Par défaut **ps** n'affiche que les processus rattaché au shell que vous utilisez
  - « ps eaux » pour avoir la liste de tous les processus
- **ps** est très utile pour retrouver des processus bloqués

Signaux (1)

- La commande **kill** permet d'envoyer des signaux aux processus
- Les signaux sont identifié par un nom et un numéro :

Signal	Numéro	Action	Description
SIGINT	2	Terminate	Interruption depuis le clavier
SIGQUIT	3	Core	Quitter depuis le clavier
SIGKILL	9	Terminate	Signal tuer
…			

- En général on utilise le signal numéro 9 (SIGKILL)

Signaux (2)

- Exemple :

```
Benjamin@Perseus ~ $ top &
Benjamin@Perseus ~ $ ps
PID   TTY          TIME CMD
15171 pts/1    00:00:00 bash
15177 pts/1    00:00:00 top
15178 pts/1    00:00:00 ps
Benjamin@Perseus ~ $ kill -9 15177
Benjamin@Perseus ~ $ ps
PID   TTY          TIME CMD
15171 pts/1    00:00:00 bash
15179 pts/1    00:00:00 ps
[1]  + Killed                  top
```

5) Commandes « filtres »

Grep

- Grep [-n] motif [fichiers] : affiche les lignes contenant le motif indiqué
  - l'option –n permet de donner les numéros de ligne également
  - motif correspond a une expression régulière

```
Benjamin@Perseus ~ $ grep –n 'Isaac Asimov' bibliotheque.txt
2 Les Robots, Isaac Asimov, 1967
3 Fondation, Isaac Asimov, 1957
6 Fondation et Empire, Isaac Asimov, 1965
```

Expression régulière

- Quantificateur:
  - x? : 0 ou une fois le caractère x
  - x* : 0 ou plusieurs fois le caractère x
  - x+ : au moins une fois le caractère x

Exemple: a*b?c+

Valide	Invalide
abc	ab
c	bbc
bc	abcd
aabc	lfzlkdsvqjdf
aabcc	

Expression régulière

- Groupe de caractères:
  - [abc] : a, b ou c
  - [a-z] : tous les caractères de a à z
  - [^abc] : tous les caractères sauf a, b et c
  - (abc)\{n,m\} : entre n et m fois la chaine abc
- Caractères spéciaux:
  - . : n'importe quel caractère
  - ^ : début de la ligne
  - $: fin de la ligne
  - | : opérateur de choix
  - \ : caractère d'échappement

Expression régulière

- Exemples
  - chat|chien : le mot chat ou le mot chien
  - [cC]hat|[cC]hien : les mots chat, Chat, chien ou Chien
  - peu[xt]? : les mots peu, peux ou peut
  - ‘^[0-9]\{2,3\} .*’ : toutes les lignes commençant par un nombre à 2 ou 3 chiffres
  - .*\]$: toutes les lignes se terminant par]

6) Redirections

Redirections

- Chaque programme connait trois « flux » d'entrées/sorties qui sont par défaut:
  - l'entrée standard: le clavier (0)
  - la sortie standard: l'écran (1)
  - la sortie d'erreur: l'écran (2)
- On peut rediriger chacun de ces flux vers des fichiers ou un autre programme

Redirection depuis/vers un fichier

- Rediriger la sortie standard

```
Benjamin@Perseus ~ $ commande > fichier1
```

  - si fichier1 n'existe pas il est créé, sinon il est écrasé

```
Benjamin@Perseus ~ $ commande >> fichier1
```

  - si fichier1 n'existe pas il est créé, sinon les nouvelles informations sont mise à la suite

- Rediriger la sortie d'erreur

```
Benjamin@Perseus ~ $ commande 2> fichier1
```

- Rediriger la sortie standard et la sortie d'erreur

```
Benjamin@Perseus ~ $ commande >& fichier1
```

- Rediriger l'entrée standard

```
Benjamin@Perseus ~ $ commande < fichier1
```

Redirection depuis/vers un fichier

- Exemple: trier un fichier et enregistrer le résultat:

```
Benjamin@Perseus ~ $ sort < fichier1 > fichier1_trie
```

- Chercher un fichier et ignorer les erreurs (droits d'accès insuffisants)

```
Benjamin@Perseus ~ $ find / -name *.jpg 2> /dev/null
```

  - Le fichier /dev/null est un fichier spécial du système qui n'enregistre rien. Toute information écrite dedans est perdue.

Pipe

- Les pipes (tubes) permettent de relier la sortie d'un programme à l'entrée d'un autre
- Pour utiliser le résultat d'une commande avec une autre on peut écrire:

```
Benjamin@Perseus ~ $ commande1 > mon_fichier
Benjamin@Perseus ~ $ commande2 < mon_fichier
```

- Ou plus simplement:

```
Benjamin@Perseus ~ $ commande1 | commande2
```

Pipe

- Exemple: Afficher tous les fichiers de / appartenant à root:
  - liste des fichiers de / avec le nom du propriétaire : « ls –l / »
  - les lignes contenant le mot root : « grep root »
  - supprime les espaces inutils: « tr –s ’ ’ »
  - la 9é colonne de chaque ligne (les colonnes sont séparée par un espace) : « cut –d’’ ’’ –f 9 »

```
Benjamin@Perseus ~ $ ls -l / | grep root | tr -s " " | cut -d" " -f9
var
usr
tmp
lib
…
```

Exécution séquentielle

- Exécuter des commandes l'une après l'autre
- Séparer les commandes par des ;

`Cmd1 ; Cmd2 ; Cmd3 ; ...`

- Exemple :

```
Benjamin@Perseus ~ $ echo 'Hello World!' > hello ; write Benjamin < hello

Message from Benjamin@Perseus on pts/1 at 11:20 ...
Hello World!
EOF
Benjamin@Perseus ~ $
```

Exécution conditionnelle

- Chaque commande retourne un code d'erreur
  - 0 signifie "succès"
  - tout autre valeur signifie "échec"
- On peut vouloir réaliser des actions différentes selon le code d'erreur d'une commande:
  - Exécuter commande2 si commande1 s'est terminée normalement:

```
Benjamin@Perseus ~ $ commande1 && commande2
```

  - Exécuter commande3 si commande1 ne s'est pas terminée normalement:

```
Benjamin@Perseus ~ $ commande1 || commande3
```

Exécution conditionnelle

- Exemple créer un répertoire et se déplacer dedans si la création à fonctionner, afficher une erreur sinon:

```
Benjamin@Perseus ~ $ (mkdir monrep && cd monrep) || echo ’erreur’
```

Autres commandes utiles

- split : découpage
- wc : comptage des caractères/mots/lignes
- sort: trier les lignes
- tr: conversion de caractères
- cat: concaténation
- cut: extraction de champs
- cmp: comparaison
- diff: différence
- sed: éditeur de texte non interactif
- awk: sed en plus puissant
- ...

6) Shell script

Principe

- On utilise le shell en mode batch (opposé du mode interactif que l'on utilisait jusqu'alors)
- Les instructions à exécuter sont enregistrées dans un fichier appelé « shell script » qui sera interprété par le shell.

Remarque

- Les shell scripts ne sont pas très agréable à coder :
  - syntaxe stricte, pas toujours évidente
  - lisibilité mauvaise
  - résultat généralement illisible pour un non initié
- Alors pourquoi les utiliser ?
  - facile à mettre en œuvre (rien à installer)
  - entrée/sortie faciles
  - accès direct au système et à l'ensemble des commandes
- Quand les utiliser ?
  - Automatisation d'une tâche répétitive
  - Création de nouvelles commandes
  - (administration système)

Script minimal

- Fichier: HelloWorld.sh

```
#! /bin/bash
# ceci est un commentaire
# la première ligne indique l'interpréteur de script à utiliser
msg="Hello World !"
echo $msg
```

- Un script commence par le nom de l'interpréteur à utiliser
- Pour l'exécuter (pensez à mettre les droits)

```
Benjamin@Perseus ~ $ ./helloWorld.sh
Hello World !
```

Les variables (1)

- Toutes les variables sont de type chaine de caractères
- Création/initialisation

```
maVariable=“valeur”
```

- Remarque : pas d’espace à gauche ou à droite du signe =
- Utilisation avec le signe $

```
echo $maVariable
```

- De manière équivalente:

```
echo ${maVariable}
```

Variables (2)

- On peut utiliser une variable pour désigner une autre variable (méta-programmation)

```
a="2"
b="5"
prix="b"
echo "Le prix est ${$prix}"
# affiche : Le prix est 5
```

- Initialisation à partir du résultat d'une commande:
  - Avec des back quotes (Alt Gr+7)

```
a=`cat /etc/passwd | cut –d: -f1,2`
```

  - Avec des parenthèses

```
a=$(ps eaux | grep `whoami`)
```

- Attention: on peut utiliser des variables non initialisées qui sont alors égales à la chaine vide.

Single quote, double quotes, back quotes

- Les variables sont interprétées à l'intérieur des doubles quotes:

```
a="2"
b="${a}5"
echo "b contient $b"
# affiche: b contient 25
```

- Les variables ne sont pas interprétées à l'intérieur des simple quotes:

```
a="25"
echo 'b contient $b'
# affiche: b contient $b
```

- Les variables sont interprétées à l'intérieur des back quotes

```
a="root"
echo `ps eaux | grep "$a"`
# affiche la liste des processus appartenant à root
```

L'instruction read

- read permet de lire une ligne depuis l'entrée standard (clavier par défaut) et la met dans une variable

```
echo "Entrez votre nom et prénom"
read nomEtPrenom
echo "Vous vous nommez : $nomEtPrenom"
```

- On peut également séparer les mots dans des variables différentes

```
echo "Entrez votre nom et prénom"
read nom prenom
echo "Votre nom est : $nom et votre prenom : $prenom"
```

- On peut bien sûr rediriger l'entrée standard:

```
read ligne < germinal.txt
echo "Première ligne de germinal : $ligne"
```

Arithmétique

- Comment faire des maths si tout est chaine de caractères?
- Les expressions contenues dans $((...)) sont traitées comme des expressions arithmétiques

```
i=2
j=$((2*i+1))
echo $j
# affiche : 5
```

- Dans l'expression arithmétique la variable n'est pas préfixée par $

Variables spéciales

- Certaines variables sont prédéfinies et ne sont pas modifiable par l'utilisateur:

Nom	Description
$#	Nombre de paramètres reçus
$0	Nom du script utilisé
$*	Ensemble des paramètres reçus sous la forme d'une unique chaine de caractères
$@	Ensemble des paramètres reçus sous la forme d'un tableau
$1, $2,...,$9,${10}	Désigne le i-ième paramètre reçu
$?	Résultat de la dernière commande exécutée
$$	PID du script
$!	PID de la dernière commande exécutée

Variables spéciales: exemple

- On créé un fichier script.sh

```
#!/bin/bash
echo "Bonjour, mon nom est $0, je porte le numéro $$"
echo "J'ai reçu $# paramètres"
echo "Le premier est \"$1\""
```

- Exécution:

```
Benjamin@Perseus ~$ ./script.sh unParametre unAutre
Bonjour, mon nom est ./script.sh, je porte le numéro 6080
J'ai reçu 2 paramètres
Le premier est "unParametre"
```

Structures conditionnelles

```
if condition1
then
        action1
elif condition2
then
        action2
else
        action4
fi
```

- Condition peut-être n'importe quelle commande (simple ou composée), c'est la valeur de retour de la commande ($?) qui est testée
- Si la commande réussit ($?=0) alors la condition est remplie, l'action correspondante est exécutée
- Toutes les autres valeurs de retour sont considérées comme équivalente à false

La commande test

- La commande test permet d'effectuer tous les tests basiques, elle est symbolisée par les caractères [ ]

```
if test condition1   # appel à la fonction test
then
        action1
elif [ condition2 ]  # appel à la fonction test (équivalent)
then
        action2
else
        action4
fi
```

- Il est nécessaire de respecter les espaces avant et après chaque crochet !

Test [] (1)

Conditions sur les chaines de caractères		
-z *chaine*	vrai si *chaine* est vide	`[ -z "$var" ]`
-n *chaine*	vrai si *chaine* non vide	`[ -n "$var" ]`
chaine1 = *chaine2*	vrai si les 2 chaines sont égales	`[ "$var" = "lundi" ]`
chaine1 != *chaine2*	vrai si les 2 chaines sont différentes	`[ "$var" != "lundi" ]`

Conditions numériques		
num1 –eq *num2*	égalité (==)	`[ $n -eq 42 ]`
num1 –ne *num2*	inégalité (!=)	`[ $n -ne 42 ]`
num1 –lt *num2*	inférieur (<)	`[ $n -lt 42 ]`
num1 –le *num2*	inférieur ou égal (<=)	`[ $n -le 42 ]`
num1 –gt *num2*	supérieur (>)	`[ $n -gt 42 ]`
num1 –ge *num2*	supérieur ou égal (>=)	`[ $n -ge 42 ]`

Test [] (2)

Conditions numériques		
-e *filename*	vrai si *filename* existe	`[ -e "$file" ]`
-d *filename*	vrai si *filename* est un répertoire	`[ -d "$file" ]`
-f *filename*	vrai si *filename* est un fichier ordinaire	`[ -f "$file" ]`
-L *filename*	vrai si *filename* est un lien symbolique	`[ -L "$file" ]`
-r *filename*	vrai si *filename* est lisible (r)	`[ -r "$file" ]`
-w *filename*	vrai si *filename* est modifiable (w)	`[ -w "$file" ]`
-x *filename*	vrai si *filename* est exécutable (x)	`[ -x "$file" ]`
file1 –nt *file2*	vrai si *file1* est plus récent que *file2*	`[ "$f1" -nt "$f2" ]`
file1 –ot *file2*	vrai si *file1* est plus ancien que *file2*	`[ "$f1" -ot "$f2" ]`

Opérateurs logiques		
! *condition*	vrai si *condition* est fausse	`[ ! ("$var" = "abc") ]`
cond1 –a *cond2*	vrai si *cond1* ET *cond2* sont vraies	`[ (-e "$f") -a (-w "$f") ]`
cond1 –o *cond2*	vrai si *cond1* OU *cond2* est vraie	`[ ($n -ge 2) -o ($n -le 4)]`

Conditionnelle : exemple (1)

- Utilisation d'une commande

```
#!/bin/bash
if grep $USER /etc/passwd &>/dev/null
then
       echo "Vous apparaissez dans /etc/passwd"
else
        echo "Vous n'apparaissez pas dans /etc/passwd"
fi
```

- Grep retourne 0 si au moins une ligne a été trouvée et 1 sinon

Conditionnelle : exemple (2)

- pasdeparametre.sh

```
#!/bin/bash
if [ $# -ne 0 ]
then
        echo "Je n'accepte aucun parametre !"
        echo "usage : $0"
        exit 1
fi
exit 0
```

Conditionnelle : exemple (3)

- Comparer deux nombres

```
#!/bin/bash
if [ $# -ne 2 ]
then
        echo "Parametres incorrects"
        echo "usage : $0 nombre1 nombre2"
        exit 1
fi
if [ $1 -lt $2 ]
then
        echo "$1 est plus petit que $2"
elif [ $1 -gt $2 ]
then
        echo "$1 est plus grand que $2"
else
        echo "$1 et $2 sont égaux"
fi
```

Boucles for

```
for x in liste
do
        action
done
```

- liste désigne un ensemble de valeurs
- x prend successivement toutes les valeurs de liste

Boucle for : exemple 1

```
#!/bin/bash
a="un deux trois"
for mot in $a
do
        echo "$mot"
done
```

- Affiche les mots "un", "deux" et "trois" l'un après l'autre.
- Les éléments de la liste sont donc séparés par des espaces.

Boucle for : exemple (2)

- monls.sh : affichage de liste des fichiers

```
#!/bin/bash
for file in *
do
        echo "$file"
done
```

- * est remplacé par la liste des fichiers présents dans le répertoire courant

Boucle for : exemple (3)

- monwc.sh : comptage de mots

```
#!/bin/bash
n=0
texte=`cat $1`
for mot in $texte
do
        n=$((n+1))
done
echo "Nombre de mots: $n"
```

Boucle while

```
while condition
do
        action
done
```

- L'action est exécuté tant que condition est vraie

While : exemple (1)

```
#!/bin/bash
while true
do
        echo "Je ne m'arrete jamais !"
done
```

- true est toujours vrai...
- Le programma boucle à l'infini

While : exemple (2)

- Lecture d'un fichier ligne par ligne

```
#!/bin/bash
n=1
while read line
do
        echo "$n : $line"
        n=$((n+1))
done < fichier.txt
```

- Le programme ajoute le numéro de ligne devant chaque ligne

While : exemple (3)

- Interagir avec l'utilisateur:

```
#!/bin/bash
while [ "$reponse" != "quitter" ]
do
        echo 'Ecrivez "quitter" pour quitter'
        read reponse
done
```

- Tant que l'utilisateur ne donne pas une réponse valide le programme continue de poser la même question.

Définition de fonctions 1

- Vous pouvez définir des fonctions dans un script

```
#!/bin/bash
unefonction() {
        echo "on m'appelle"
}
unefonction
unefonction
# affiche 2 fois "on m'appelle"
```

- Les fonctions d'utilisent comme une nouvelle commande

Définition de fonctions 2

- Une fonction doit être définie avant d'être utilisée
- Les fonctions ne déclarent pas de paramètre ni de type de retour
- Les paramètres sont gérés comme pour le script ($1, $2, ...)
- Il est possible de retourner une valeur avec l'instruction return
- La valeur retournée est alors placée dans la variable $? du script appelant

Fonction : exemple

- Script affichant le maximum de deux nombres

```
#!/bin/bash
max() {
        if [ "$1" -ge "$2" ]
        then
                v=$1
        else
                v=$2
        fi
        return $v
}
echo "Donnez deux nombres :"
read n1 n2
max $n1 $n2
echo "Le max est $?"
```