Cours 1 : Shell et commandes

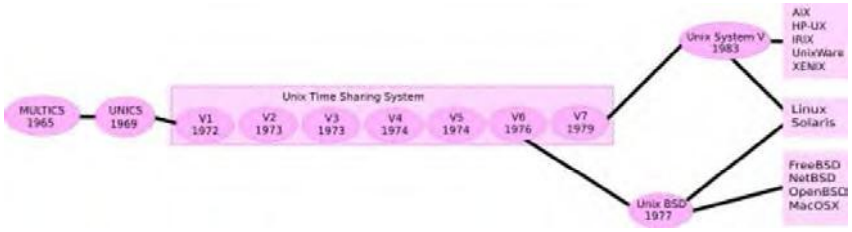
Rabii El Ghorfi

Plan

Historique

- 1965: *Multics* (laboratoires Bell - AT&T, MIT, General Electric)
- 1969: *Unics* (Ken Thompson, laboratoires Bell, développé en langage d'assemblage)
- 1971: publication de *The UNIX Programmer's manual*
- 1973: réécriture de *Unix* en langage C (Dennis Ritchie, Brian Kernighan)
- fin des années 70: reprise par le monde académique (Université de Californie à Berkeley)

Historique (suite)



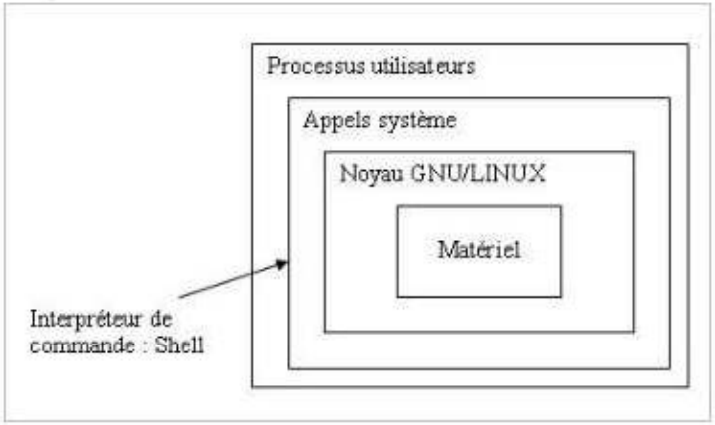
(source: Wikipedia)

Définition

UNIX est un système d'exploitation permettant de contrôler un PC et ses différents périphériques. UNIX se distingue par les caractéristiques suivantes :

- **multi-utilisateurs** (qui peut être utilisé simultanément par plusieurs personnes)
- **multi-tâches** (un utilisateur peut exécuter plusieurs programmes en même temps)
- repose sur un **noyau** (kernel) utilisant 4 concepts principaux **fichiers**, **droits d'accès**, **processus** et **communication interprocessus** (IPC)

Schéma d'UNIX



Plan

L'interpréteur de commande

- **Shell** : interface entre l'utilisateur et le système d'exploitation ("coquille")
- Application (fichier exécutable) chargé d'interpréter les commandes des utilisateurs et de les transmettre au système
- Différents types de shell, les principaux étant :
  - **sh** (Bourne shell)
  - **bash** (Bourne again shell)
  - **csh** (C shell)
  - **Tcsh** (Tenex C shell)
  - **ksh** Korn shell
  - **zsh** Zero shell
- Le nom du shell correspond généralement au nom l'exécutable :
  `% /bin/bash`

Utilisation du shell

- Le shell correspond à une fenêtre présentant un *prompt*, encore appelé *invite de commande*. Celle-ci est paramétrable et par défaut en bash se compose comme suit :

  `login@machine$`

  (suffixe \$ → utilisateur normal,
  suffixe # → super-utilisateur – administrateur)
- On saisit les commandes à la suite du prompt
- Pour stopper la commande en cours : `Ctrl-C`
- Pour mettre en attente la commande en cours : `Ctrl-Z`
- Pour terminer l'entrée standard (les éventuelles par ` donnés par l'utilisateur via le clavier) : `Ctrl-D`

Utilisation du shell (suite)

- Le shell est personnalisable au moyen des fichiers suivants :
  1. le fichier `/etc/profile`, s'il existe
  2. le fichier `$HOME/.bash_profile`, s'il existe
  3. le fichier `$HOME/.bash_login`, s'il existe
  4. le fichier `$HOME/.profile`, s'il existe
  5. le fichier système `/etc/bashrc`
  6. le fichier caché `.bashrc`, s'il existe

Les entrées-sorties standards

- Lors de l'exécution d'une commande, un processus est créé. Celui-ci va alors ouvrir trois flux :
  - stdin : l'**entrée standard**, par défaut le clavier, identifiée par l'entier **0** (descripteur)
  - stdout : la **sortie standard**, par défaut l'écran, identifiée par l'entier **1**
  - stderr : la **sortie d'erreur standard**, par défaut l'écran, identifée par l'entier **2**

Les redirections

Il est possible de redigirer les flux d'entée-sortie au moyen d'opérateurs spécifiques :

>	redirection de la sortie standard (par exemple dans un fichier)
<	redirection de l'entrée standard
>>	redirection de la sortie standard avec **concaténation**
>&	redirection des sorties standard et d'erreur
>!	redirection avec écrasement de fichier
|	redirection de la sortie standard vers l'entrée standard (pipe)

Exemple: la commande echo

```
$ echo "ca va"
ca va
$ java toto
Exception in thread "main"
java.lang.NoClassDefFoundError:  toto
$ java toto > erreur.txt
Exception in thread "main"
java.lang.NoClassDefFoundError:  toto
$java toto > & erreur.txt
```


Plan

Le système de fichiers

- Le système de fichier correspond à une arborescence que l'on parcourt de la racine (root) vers les feuilles
- La racine se note / (slash)
- Il s'agit d'un répertoire contenant les sous-répertoires suivants :

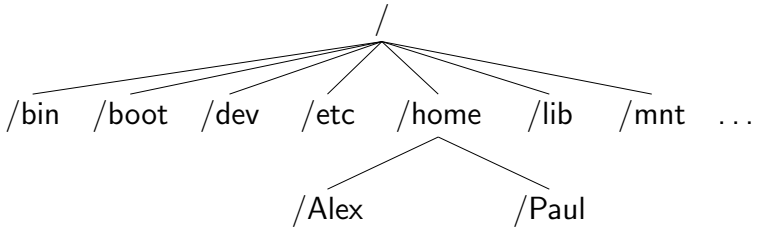
/bin	exécutables essentiels pour le système, directement utilisable par les utilisateurs
/boot	contient les fichiers permettant à Linux de démarrer
/dev	contient les points d'entrée des périphériques (=device)
/etc	configuration du réseau → contient les commandes et les fichiers à l'administrateur du système (fichiers pa group, inittab, ld.so.conf, lilo.conf, ...)

Le système de fichiers (suite)

- Sous-répertoires de la racine (suite) :

/home	répertoire personnel des utilisateurs
/lib	contient des bibliothèques partagées essentielles au système lors du démarrage
/mnt	contient les points de montage des partitions temporaires (cd-rom, disquette, ...), parfois nommé *media*
/opt	contient des packages d'applications supplémentaires
/proc	fichiers content des info sur la mémoire, E/S, périphérique, compatibilité pour le noyau, ...
/root	répertoire de l'administrateur root
/usr	hiérarchie secondaire (utilisateurs)
/var	contient des données variables
/tmp	contient les fichiers temporaires

Représentation graphique



Plan

Les commandes fondamentales

- Aide
  `$ man commande`
  → Manuel pour les commandes
- Où suis-je dans l'arborescence ?
  `$ pwd`
  NB: chemin absolu vs chemin relatif

  Exemple:
  `yannick@nausicaa:~/toto $ pwd`
  `/home/yannick/toto`

Les commandes fondamentales (suite)

- Comment se déplacer dans l'arborescence ?
  `cd [chemin]`
  → Permet de changer de répertoire (**c**hange **d**irectory)

  Alias :
  . → répertoire courant
  .. → répertoire parent

  Exemples :
  `$ pwd` → `/home/yannick/toto`
  `$ cd ..` → `/home/yannick/`
  `$ cd projet` → `/home/yannick/projet`
  `$ cd /usr/local` → `/usr/local`

Les commandes fondamentales (suite)

- Lister le contenu d'un répertoire ?
  `ls [option] [chemin]`
  → Liste le contenu d'un répertoire avec plus ou moins de détails

  Exemples :
  `$ ls l*` → liste tous les fichiers commençant par l
  `$ ls -l` → liste tous les fichiers du répertoire courant, en donnant les attributs des fichiers (droits, taille, etc)
  `$ ls -a` → liste tous les fichiers du répertoire courant (y compris les fichiers cachés dont le nom commence par un ".")
  `$ man ls` → affiche la page de manuel de la commande ls

Les commandes fondamentales (suite)

- Visualiser le contenu d'un fichier ?
  `cat [option] [chemin vers le fichier1, fichier 2, etc]`
  → affiche le contenu d'un fichier

  Exemples :
  `$ cat .bash_profile` → affiche le contenu du fichier caché `.bash_profile`
  `$ cat toto > tata` → écrit le contenu du fichier toto dans un fichier nommé tata

- Visualiser le contenu d'un fichier page à page ?
  `more [fichier]`

- Visualiser le contenu d'un fichier dans un flux ?
  `less [fichier]`

Les commandes fondamentales (suite)

- Obtenir des statistiques sur le contenu d'un fichier ?
  `wc [option] [chemin vers le fichier]`
  → affiche le nombre de mots / lignes / caractères d'un fichier

  Exemples :
  `$ wc -l toto` → affiche le nombre de lignes du fichier toto
  `$ wc -c toto` → affiche le nombre de caractères du fichier toto
  `$ ls | wc -l` → affiche le nombre de fichiers dans le répertoire courant

- Editer un fichier ?
  `emacs [fichier]`
  `vim [fichier]`
  `gedit [fichier]`
  ...

Les commandes fondamentales (suite)

- Copier un fichier ?
  `cp [option] [chemin vers fichier source] [chemin vers fichier destination]`
  → copie un fichier source en le renommant si le chemin du fichier destination contient un nom de fichier

  Exemples :
  `$ cp toto /tmp/` → copie le fichier local toto dans /tmp (toujours nommé toto)
  `$ cp toto /tmp/tata` → copie le fichier local toto dans /tmp en le nommant tata
  `$ cp -r projet /tmp` → copie le contenu du répertoire projet dans le répertoire /tmp/projet

Les commandes fondamentales (suite)

- Déplacer un fichier ?
  `mv [option] [chemin vers fichier source] [chemin vers fichier destination]`
  → déplace un fichier source en le renommant si le chemin du fichier destination contient un nom de fichier

  Exemples :
  `$ mv toto /tmp/` → déplace le fichier local toto dans /tmp (toujours nommé toto)
  `$ mv toto /tmp/tata` → déplace le fichier local toto dans /tmp en le nommant tata
  `$ mv -i toto /tmp` → déplace le fichier toto dans /tmp en prévenant l'utilisateur s'il existe déjà un fichier /tmp/toto

Les commandes fondamentales (suite)

- Supprimer un fichier ?
  `rm [option] [chemin vers fichier]`
  → supprime un fichier

  Exemples :
  `$ rm toto` → supprime le fichier toto
  `$ rm -i toto` → supprime le fichier toto en demandant confirmation à l'utilisateur
  `$ rm -f toto*` → supprime les fichiers dont le nom commence par toto, sans demander confirmation à l'utilisateur
  `$ rm -r projet` → efface récursivement le contenu du répertoire projet

Les commandes fondamentales (suite)

- Créer / supprimer un répertoire ?
  `mkdir [chemin vers répertoire]`
  `rmdir [chemin vers répertoire]`
  → crée / supprime un répertoire *vide*

  Exemples :
  `$ mkdir toto` → crée le répertoire toto
  `$ rmdir toto` → supprime le répertoire vide toto
  `$ rmdir projet` → `rmdir: projet/: Directory not empty`

Les commandes fondamentales (suite)

- Retrouver un fichier ?
  `find [options]`
  → effectue une recherche à partir des informations données en option

  Exemples :
  `$ find . -name toto` → cherche, dans le répertoire courant et ses sous-répertoires, un fichier nommé toto
  `$ find /tmp/ -type d` → cherche tous les sous-répertoires du répertoire /tmp
  `$ find /tmp -type d -exec ls '{}' \;` → affiche le contenu des sous-répertoires du répertoire /tmp

Cours 2 : Système de fichiers

Rabii El Ghorfi

Un système d'exploitation

Exemples connus :

▷ Windows, ▷ Linux, ▷ OS X

Qu'apportent-ils ?

- ▷ La possibilité d'utiliser l'ordinateur par une interface graphique ou plutôt une interface homme-machine
  - lancer des programmes
  - copier/déplacer/$\cdots$ des fichiers
- ▷ Permettre aux programmes de fonctionner quelque soit le matériel
  - jouer à un jeu vidéo quelque soit la carte vidéo et sa performance, avec plus ou moins d'options

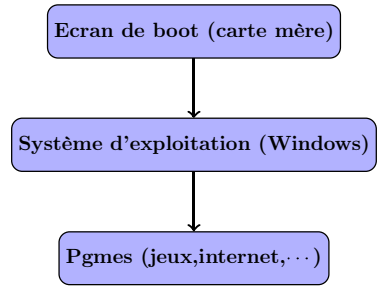
Un système d'exploitation

Pour aller plus loin :

L'OS (Operating System) gère

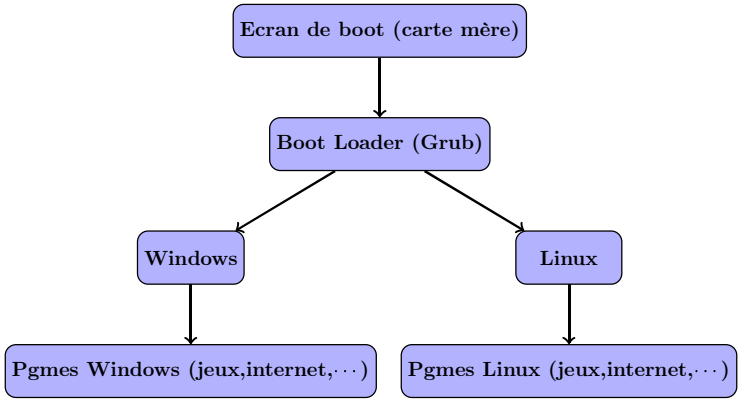
- ▷ **La mémoire** : il la partage entre tous les programmes
- ▷ **Les périphériques** : écran, imprimante, disque dur, réseau. Il s'assure que les programmes puissent les utiliser de façon standard.
- ▷ **Le processeur** : il le partage entre tous les programmes pour qu'ils aient l'air de fonctionner parallèlement
- ▷ **Les utilisateurs** : gérer les droits d'accès aux fichiers, comme au matériel
- ▷ **La standardisation des programmes** : offre des interfaces de programmation simplifiées et standardisées

Vue générale



Mais pas que!!! Peut on avoir 2 OS sur son ordinateur???

Vue générale



Système d'exploitation

Définition (Système d'Exploitation)

Un système d'exploitation (**SE**) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications informatiques de l'utilisateur (traitement de textes, vidéo,...).
Il fournit aux programmes applicatifs des points d'entrées génériques pour les périphériques.

Le système Unix est un système d'exploitation multi-utilisateur et multi-tâche

Unix est multi-utilisateurs

Multi-User : Plusieurs utilisateurs sous Unix. Chacun dispose de l'ensemble des ressources du système. Comme tout système multi-utilisateur, Unix comporte des mécanismes d'identification et de protection permettant d'éviter toute interférence entre utilisateurs.

2 types de Users :

1. Users *normaux* : compte avec
   - Login
   - password
   - Espace de travail protégé (**rep.** privé -`home directory`)
   - mail box
2. *Super-User* `root` gère tout le système

Unix est multi-tâche

Multi-tâches : Unix est multi-tâche car plusieurs programmes peuvent être en cours d'exécution en même temps sur une même machine.
Un **processus** est une tâche en train de s'exécuter. On appelle **processus**, l'image de l'état du processeur et de la mémoire au cours de l'exécution du programme.
En fait, à chaque instant, le processeur ne traite qu'au plus un seul des programmes lancés. La gestion des processus est effectuée par le système.

Fonctions principales d'Unix

- Partage des ressources équitables : veiller au partage équitable des ressources entre tous les processus.
- Interface avec le matériel :passage par des fichiers spéciaux gérés par le SE. pour accéder à une ressource matériel (disque dur, lecteur de disquettes,...)
- Gestion de la mémoire :partage correct de la RAM entre processus.
- Gestion des fichiers : Unix fournit un mécanisme de protection des fichiers.

Unix fonctionne par couche

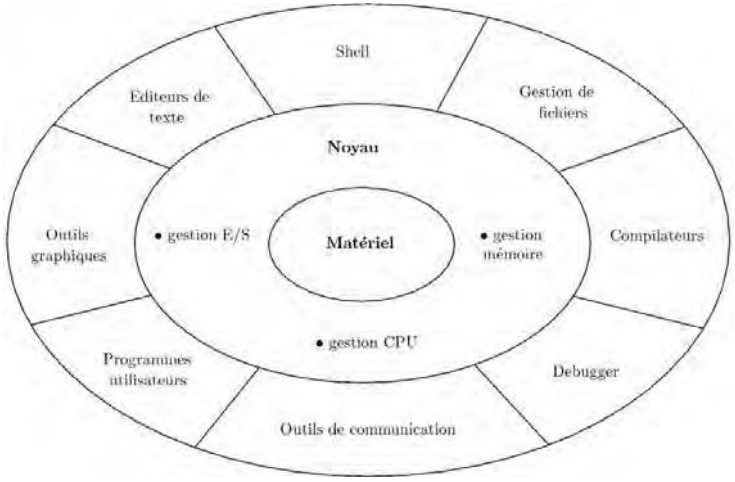
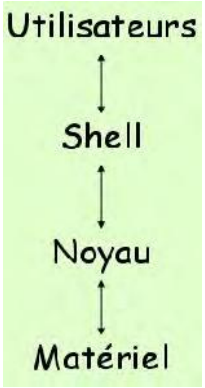


Schéma d'exploitation de la machine



- **shell :** interpréteur de commandes Unix (vérifie, interprète les commandes, exécute et renvoie les réponses). Le Shell envoie des appels au noyau en fonction des requêtes des utilisateurs
- **noyau :** couche logicielle la plus interne du S.E Unix dédiée à la gestion des composants matériels : processeur, mémoire, périph.
- Autour du noyau gravite un certain nombre d'utilitaires.

Connexion-Déconnexion

Connexion : S'identifier pour ouvrir une session (de travail) :

- Entrer nom de connexion aprés le message `login`
- Entrer mot de passage aprés le message `password`

$\rightsquigarrow$ L'utilisateur de trouve alors dans son répertoire privé correspondant à son `login` (`home directory`)

Déconnexion : En l'absence d'environnement graphique, une simple commande `exit` suffit pour terminer ma session de travail.

Choisir son mot de passe

Un bon mot de passe :

- posséder entre 7 et 8 caractères
- posséder au moins une lettre majuscule
  au moins un chiffre
  et un caractère de ponctuation
- ne pas contenir de donnés relatives à votre identité
- ne pas appartenir à un dictionnaire
- ne pas contenir de répétition de caractères
- ...

Commande Unix en console

Unix fonctionne en mode ligne de commandes et non en mode graphique $\Rightarrow$ permet des opérations plus complexes.
Une commande est un programme. Pour l'exécuter $\Rightarrow$ taper son nom éventuellement suivi d'options et d'arguments.
Syntaxe :

```
nom_commande [-liste_options][liste_arguments]
```

Exemple : `ls -l` $\hookleftarrow$
Lors de l'appui sur la touche `Entrée`, le shell analyse la ligne de commande et l'interprète.
Différence entre majuscules et minuscules. : On dit que la console Unix est sensible à la casse.

Commande Unix en console

Aide en ligne : Doc. de référence organisée en 9 sections

1	Commande users
2	Appels système
3	Fct. bib. standard
4	Formats fichier
5	Tables
6	Jeux
7	Drivers périph.
8	Commandes admin.
9	Commandes locales

Visualiser une page du manuel :
`man[-s section] nom_commande`
Recherche page qui se rapporte a un mot clé :
`man -k mot-clé`
Quelques commandes :

- `who` Affiche les users actuellement connectés
- `date` Consulter date et heure
- `cal [mois[année]]` Affiche calendrier

Les fichiers sous Unix

Définition (fichier)

fichier : objet recevant et délivrant des données, constitué d'une chaîne de caractères non structurée.

Type de Fichiers :

- Fich. ordinaire : données stockées sur un disque
- Répertoire : ensemble d'informations permettant l'accés à d'autres fichiers
- Fich. spécial : dispositif d'entrée/sortie (terminal, lecteur,...)

Description de Fichiers : dans un **i-nœud** (**inode**) comportant

- type de fichier, mode de protection, nb. liens, num. propriétaire
- num groupe, taille fichier, adr.physique direct
- date et heure dernière modif., date heure dernier accés,....

Les fichiers sous Unix

`ls -i fich` : numéro i-œud du fichier fich.

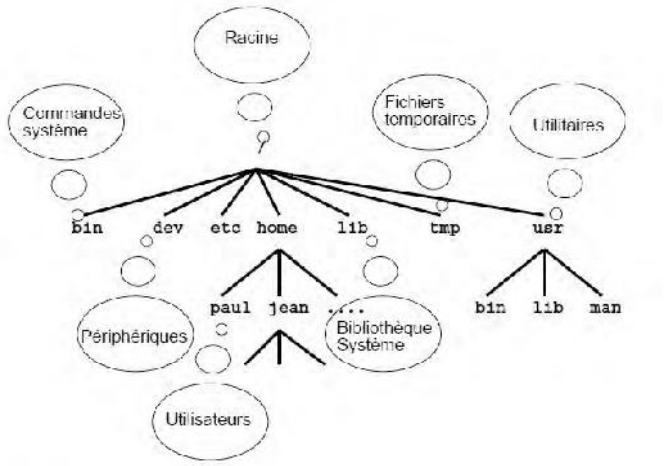
`ls -l rep` : liste contenu repertoire rep `-l` fournit des détails des fichiers contenu dans le répertoire rep.

```
-rw-rw-r- 1 nicolas nicolas 3205 août 24 09:53 demenagement.org
-rw-rw-r- 1 nicolas nicolas 2453 juil. 18 16:07 recherche.org
```

Opérations élémentaires sur les fichiers

`cat fich`	affiche sur la sortie standard le contenu de `fich`
`more fich`	affiche contenu de `fich` page par page
`head fich`	affiche début de `fich`
`tail fich`	affiche fin de `fich`
`sort fich`	trie lignes de `fich`
`ls rep`	affiche contenu de `rep`

Arborescence des fichiers



".." désigne le répertoire parent.

Accès aux fichiers

Atteindre un fichier :

- Référence absolue : chemin à partir de la racine (`/usr/local/bin`)
- Réference relative : chemin à partir du répertoire de travail

Commandes :

`pwd`	indique la réf. absolue du rep de travail
`cd`	`cd ..` permet de remonter dans l'arbre
`ls -R`	liste récursivement les sous-rep. et leur contenu
`mkdir`	création d'un rep

À sa création, un rep contient deux liens :
(index, .) : un lien sur lui-mème
(index, ..) : un lien sur son père
Le nombre de liens sur un rep. est $\geqslant 2$
(index, nom) dans le rep. père
(index, .) dans lui-mème

Opérations sur les fichiers

cp - to copy - copier

`cp f_source f_dest` recopie physique de `f_source` dans `f_dest`

rm - to remove - supprimer

`rm fich` suppression de `fich`

`rm -r fich` suppression du rep `fich` et de son contenu

mv - to move - déplacer

`mv f_source f_dest` renommer le fichier `f_source` en `f_dest`

`mv f rep_accueil` déplace le fichier `f`

Chemin absolu et relatif

Notion de chemin d'accès :

- Pour identifier un fichier : suite de noms étiquetant les arêtes le long de l'arborescence.
- racine absolue : /
- / sert aussi de séparation entre sous-répertoires.

Référence absolue : chemin d'accès `pathname` depuis la racine (permettant le repérage sans ambiguité)
Exemple : `/home/prot1/formation/M1IR`
Référence relative : Selon l'endroit où l'on se situe (répertoire de travail = `working directory`), repérer un fichier peut s'effectuer de manière relative.
Exemple : `../../DESS`

Des fichiers physiques différents appartenant à des disques logiques distincts peuvent avoir le même index de i-nœud ⇒ impossible de créer des liens
Le système Unix permet de créer des liens symboliques entre des fichiers.

Définition (Lien symbolique)

fichier contenant la référence absolue d'un autre fichier. Toute opération sur ce fichier (lecture, écriture, ...) s'effectue sur le fichier référencé. Un lien est créé pour pouvoir accéder au même fichier à différents endroits de l'arborescence.

Commandes : (à utiliser dans un rÃ©pertoire de travail)

`ln -s f_cible lien_nom` crée un lien symbolique `lien_nom` contenant la référence à `f_cible`

`ls -l` fait apparaître le lien sous la forme `fich_dest -> fich_source`

Cours 3 : Droits d'accès

Rabii El Ghorfi

Chaque fichier (ou répertoire) possède un ensemble d'attributs définissant les droits d'accès à ce fichier pour tous les utilisateurs du système.

3 types d'utilisateurs

le propriétaire	`u`
le groupe	`g`
les autres	`o`

3 types de droits

lécture	`r`
écriture	`w`
exécution	`x`

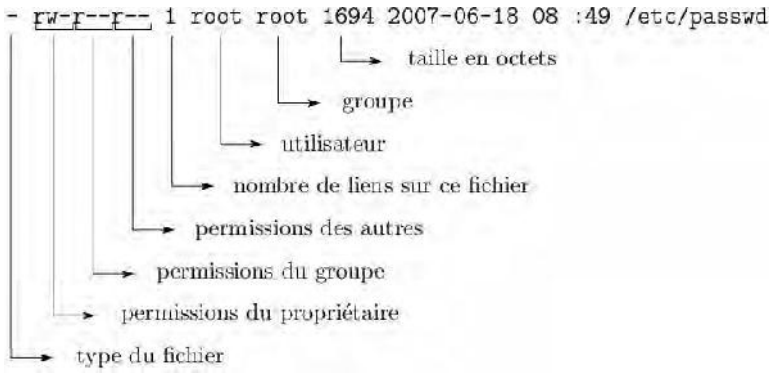
Pour les fichiers, les droits sont exprimés par une chaîne de 10 caractères :`tuuugggooo`

`t` :type du fichier :

Fichier ordinaire	`-`
Répertoire	`d`
Lien symbolique	`l`
Fichier spécial	`c` ou `b`
Socket	`s`

Le super-utilisateur (`root`) a tous les droits

Pour visualiser les droits, on utilise `ls -l`.
Présence du droit si une lettre `r`, `w` ou `x`. Abscence du droit si `-`
Exemple
`ls -l /etc/passwd` donne



On peut représenter la protection d'un fichier par trois chiffres (un nombre octal).

Exemple :`rwxrw-r-x` est représenté par 765.
Une lettre est ≡ à 1 et un tiret à 0
⇒
`rwxrw-r-x` = 111 110 101 = 765
car
$111 = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$
$110 = 2^2 + 2^1 = 4 + 2 = 6$
$101 = 2^2 + 2^0 = 4 + 1 = 5$

Droits	valeur octale
`---`	0
`--x`	1
`-w-`	2
`-wx`	3
`r--`	4
`r-x`	5
`rw-`	6
`rwx`	7

`chmod` : changer le mode de protection d'un fichier

Syntaxe

```
chmod mode nom_fichier
```

Deux modes d'utilisation :

Mode absolu : `mode` représenté par un nombre octal

Exemple `chmod 765 fich`

```
nicolas@lancelot: /Bureau$ touch ess.txt
nicolas@lancelot: /Bureau$ ls -l ess.txt
-rw-rw-r- 1 nicolas nicolas 0 août 29 23:42 ess.txt
nicolas@lancelot: /Bureau$ chmod 764 ess.txt
nicolas@lancelot: /Bureau$ ls -l ess.txt
-rwxrw-r- 1 nicolas nicolas 0 août 29 23:42 ess.txt
```


mode symbolique :

`mode` indique de quelle façon les droits d'accès doivent être modifiés. Il se décompose en `[qui] op accès` où

- `qui` (optionnel) indique quelles classes sont concernées par `chmod`, est composé de une ou plusieurs lettres (`u`, `g` et `o`). Si aucune lettre alors tous les types d'utilisateurs sont concernés (ou `a` pour all).
- `op` peut être :
  `+` pour ajouter des droits d'accès
  `-` pour enlever des droits d'accès
  `=` pour affecter des droits d'accès.
- `accès` est une combinaison des lettres `r`, `w` et `x` qui spécifient les types d'accès.

Exemple : `chmod a=r,u+w fich`

Exemple précédent :

```
nicolas@lancelot: $ ls -l ess.txt
-rwxrw-r- 1 nicolas nicolas 0 sept. 1 11:50 ess.txt
nicolas@lancelot: $ chmod u-x,g-w ess.txt
nicolas@lancelot: $ ls -l ess.txt
-rw-r-r- 1 nicolas nicolas 0 sept. 1 11:50 ess.txt
```


Vous pouvez "donner" un fichier vous appartenant à un autre utilisateur.

```
chown nouveau_propriétaire nom_fichier
```

ou changer le groupe auquel le fichier est rattachÃ©

```
chgrp nouveau_groupe nom_fichier.
```

Si vous êtes à la recherche d'un fichier qui commence par la lettre *a*, en faisant `ls`, vous voudriez voir que les fichiers commençant par *a*. De même si vous voulez appliquer une commande à certains fichiers mais pas à d'autres.
C'est le but des métacaractères, ils vous permettent de faire une sélection de fichiers suivant certains critères.

Les métacaractères

- sont des caractères génériques permettant de désigner un ensemble d'objets et
- s'appliquent aux arguments des commandes qui désignent des noms de fichiers.

Le Shell permet de générer une liste de noms de fichier en utilisant les caractères spéciaux suivants :

`*` toutes chaînes de caractères, y compris la chaîne vide
ex : `a*b` tous les noms de fichiers commençant par `a` et finissant par `b`

`?` caractère quelconque
ex : `a?b` tous les noms de fichier commençant par `a`, suivi d'un caractère et finissant par `b`

`[...]` un caractère quelconque ∈ à la liste donnée entre crochets
Le `-` permet de représenter un intervalle.
ex : `a[a-z0-9A-Z]b` désigne tous les noms de fichiers commençant par `a` suivi d'un caractère alphanumérique et finisant par `b`

`[!...]` une liste de caractères à exclure
ex : `a[!a-z]b` tous les noms de fichiers commençant par `a` suivi d'un caractère autre qu'un caractère alphabétique en minuscule et finissant par `b`

Exemple Si le répertoire courant contient :
`fich1.bin fich1.txt fich2.txt fich10.txt fichier.txt readme zzz` Alors :

`fich1*`	
`fich*.txt`	
`fich[0-9]*.txt`	
`???`	

Exemple Si le répertoire courant contient :
`fich1.bin fich1.txt fich2.txt fich10.txt fichier.txt readme zzz` Alors :

`fich1*`	`fich1.bin fich1.txt fich10.txt`
`fich*.txt`	
`fich[0-9]*.txt`	
`???`	

Exemple Si le répertoire courant contient :
`fich1.bin fich1.txt fich2.txt fich10.txt fichier.txt readme zzz` Alors :

`fich1*`	`fich1.bin fich1.txt fich10.txt`
`fich*.txt`	`fich1.txt fich2.txt fich10.txt fichier.txt`
`fich[0-9]*.txt`	
`???`	

Protection de fichiers
Métacaractères et expressions régulières
Métacaractères
Expression régulières
Recherche de chaîne dans un fichier : grep
Recherche d'un fichier find
La commande sed

Exemple Si le répertoire courant contient :
`fich1.bin fich1.txt fich2.txt fich10.txt fichier.txt readme zzz` Alors :

`fich1*`	`fich1.bin fich1.txt fich10.txt`
`fich*.txt`	`fich1.txt fich2.txt fich10.txt fichier.txt`
`fich[0-9]*.txt`	`fich1.txt fich2.txt fich10.txt`
`???`	

Protection de fichiers
Métacaractères et expressions régulières
Métacaractères
Expression régulières
Recherche de chaîne dans un fichier : grep
Recherche d'un fichier find
La commande sed

Exemple Si le répertoire courant contient :
`fich1.bin fich1.txt fich2.txt fich10.txt fichier.txt readme zzz` Alors :

`fich1*`	`fich1.bin fich1.txt fich10.txt`
`fich*.txt`	`fich1.txt fich2.txt fich10.txt fichier.txt`
`fich[0-9]*.txt`	`fich1.txt fich2.txt fich10.txt`
`???`	`zzz`

Protection de fichiers
Métacaractères et expressions régulières
Métacaractères
Expression régulières
Recherche de chaîne dans un fichier : grep
Recherche d'un fichier find
La commande sed

Les expressions régulières (comme les métacaractères) sont aussi des suites de caractères permettant de faire des sélections.
Une expression régulière peut être aussi simple qu'un mot exact à rechercher, par exemple `'Bonjour'`, ou aussi complexe que `'^ [a-zA-Z]'` qui correspond à toutes les lignes commençant par une lettre minuscule ou majuscule.
La syntaxe des expressions régulières utilise les notations suivantes :

- `c` correspond au caractère c
- `\c` banalise le métacaractère c
  ex : `\.`, `\*`, ...
- `.` caractérise n'importe quel caractère
- `[...]` n'importe quel caractère de l'ensemble spécifié
  - pour définir un intervalle

`[^ ...]`	n'importe quel caractère hors de l'ensemble spécifié
`^`	caractérise le début de ligne (`^` $\neq$ `[^ ...]`)
	ex : `^ abc` désigne une ligne commençant par `abc`
`$`	caractérise la fin de ligne
	ex : `abc$` désigne une ligne finissant par `abc`
`^ $`	ligne vide
`*`	0 à n fois le caractère la précédent
	ex : `a*` représente de 0 à n fois `a`
	`aa*` représente de 1 à n fois `a`
	`.*` désigne n'importe quelle chaîne même vide
`\{n \}`	nombre de répétition `n` du caractère placé devant

Exemple : `[0-9]\{4 \} $` : du début à la fin du fichier `$`, recherche les nombres `[0-9]` de 4 chiffres `\{4 \}`

La commande `grep` permet de rechercher une chaîne de caractères dans un fichier.
Syntaxe : `grep [option] motif nom_fichier`
Les options sont les suivantes :

- `-v` affiche les lignes ne contenant pas la chaîne
- `-c` compte le nombre de lignes contenant la chaîne
- `-n` chaque ligne contenant la chaîne est numérotée
- `-x` ligne correspondant exactement à la chaîne
- `-l` affiche le nom des fichiers qui contiennent la chaîne

Exemple : le fichier carnet-adresse :

```
olivier:29:0298333242:Brest
marcel:13:0466342233:Gardagnes
myriam:30:0434214452:Nimes
yvonne:92:013344433:Palaiseau
```

On peut utiliser les expressions régulières avec `grep`. Si on tape
`grep ^ [a-d] carnet-adresse`
On va obtenir tous les lignes commençant par les caractères compris entre `a` et `d`. Dans notre exemple, on n'en a pas, d'où l'absence de sortie.
`grep Brest carnet-adresse`
Permet d'obtenir les lignes contenant la chaîne de caractère `Brest` :

```
olivier:29:0298333242:Brest
```

Il existe aussi les commandes `fgrep` et `egrep` équivalentes.

La commande `find` permet de retrouver des fichiers à partir de certains critères.
Syntaxe :
`find <répertoire de recherche> <critères de recherche>`

critères de recherche :

`-name`	recherche sur le nom du fichier
`-perm`	recherche sur les droits d'accès
`-link`	recherche sur le nombre de liens
`-user`	recherche sur le propriétaire
`-group`	recherche sur le groupe auquel appartient le fichier
`-type`	recherche sur le type
`-size`	recherche sur la taille
`-atime`	recherche sur la date de dernier accès en lecture
`-mtime`	recherche sur la date de dernière modification du fichier
`-ctime`	recherche sur la date de création du fichier

On peut combiner les critères avec des opérateurs logiques :

- `critère1 critère2` ou `critère1 -a critère2` ≡ au ET logique,
- `!critère` ≡ NON logique,
- `\(critère1 -o critère2\)` ≡ OU logique,

L'option `-print` est indispensable pour obtenir une sortie.

Remarque

La commande `find` est récursive, *i.e.* scruter dans les répertoires, et les sous répertoires qu'il contient.

Recherche par nom de fichier

Pour chercher un fichier dont le nom contient la chaîne de caractères `toto` à partir du répertoire `/usr` :

```
find /usr -name toto -print
```

- Si le(s) fichier(s) existe(nt) ⇒ sortie : `toto`
- En cas d'échec, vous n'avez rien.

Pour rechercher tous les fichiers se terminant par `.c` dans le répertoire `/usr` :

```
find /usr -name " *.c " -print
```

⇒ toute la liste des fichiers se terminant par `.c` sous les répertoires contenus dans `/usr` (et dans `/usr` lui même).

Recherche suivant la date de dernière modification

Ex :Les derniers fichiers modifiés dans les 3 derniers jours dans toute l'arborescence `(/)` : `find / -mtime 3 -print`

Recherche suivant la taille

Ex :Connaître dans toute l'arborescence, les fichiers dont la taille dépasse 1Mo (2000 blocs de 512Ko) :

```
find / -size 2000 -print
```

Recherche combinée

Ex :Chercher dans toute l'arborescence, les fichiers ordinaires appartenant à olivier, dont la permission est fixée à 755 :

```
find / -type f -user olivier -perm 755 -print
```

Ex : Recherche des fichiers qui ont pour nom `a.out` et des fichiers se terminant par `.c` :

```
find . \ ( -name a.out -o -name " *.c " \ ) -print
```


Commandes en option :

En dehors de `-print` on dispose de l'option `-exec`. Le `find` couplé avec `exec` permet d'exécuter une commande sur les fichiers trouvés d'après les critères de recherche fixés. Cette option attend comme argument une commande, suivie de {}\ .

Ex : recherche des fichiers ayant pour nom `core`qu'on efface

```
find . -name core -exec rm {}\
```

Ex : les fichiers ayant pour nom `core` seront détruits, pour avoir une demande de confirmation avant l'exécution de `rm` :

```
find . -name core -ok rm {}\
```

Autres subtilités : Une fonction intéressante de `find` est de pouvoir être utilisé avec d'autres commandes

Ex : `find . -type f -print | xargs grep toto`

Rechercher dans le répertoire courant tous les fichiers normaux (sans fichiers spéciaux), et rechercher dans ces fichiers tous ceux contenant la chaîne `toto`.

`sed` est éditeur ligne non interactif, il lit les lignes d'un fichier une à une (ou provenant de l'entrée standard) leur applique un certain nombre de commandes d'édition et renvoie les lignes résultantes sur la sortie standard. Il ne modifie pas le fichier traité, il écrit tout sur la sortie standard.
Syntax `sed -e 'programme sed' fichier-a-traiter`
ou `sed -f fichier-programme fichier-a-traiter`
On disposez de l'option `-n` qui supprime la sortie standard par défaut, `sed` va écrire uniquement les lignes concernées par le traitement (sinon il écrit tout même les lignes non traitées).
L'option `-e` n'est pas nécessaire quand on a une seule fonction d'édition.
`sed` est une commande très riche (pour plus de détails `man sed`)

La fonction de substitution :`s`

`s` permet de changer la 1^{re} ou toutes les occurences d'une chaîne par une autre.
Syntaxe :

- `sed "s/toto/TOTO/" fichier` va changer la 1^{re} occurence de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/3" fichier` va changer la 3^{me} occurence de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/g" fichier` va changer toutes les occurences de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/p" fichier` en cas de remplacement imprime les lignes concernées
- `sed "s/toto/TOTO/w resultat" fichier` en cas de substitution la ligne en entrée est inscrite dans un fichier résultat

La fonction de substitution peut être utilisée avec une expression régulière.
`sed -e "s/[Ff]raise/FRAISE/g" fichier` substitue toutes les chaînes `Fraise` ou `fraise` par `FRAISE`

La fonction de suppression :`d`

La fonction de suppression `d` supprime les lignes comprises dans un intervalle donné.
Syntaxe : `sed "20,30d" fichier`
Cette commande va supprimer les lignes 20 à 30 du fichier `fichier`. On peut utiliser les expressions régulières :

- `sed "/toto/d" fichier` : supprime les lignes contenant la chaîne `toto`
- `sed "/toto/!d" fichier` : supprime toutes les lignes ne contenant pas la chaîne `toto`

En fait les lignes du fichier d'entrée ne sont pas supprimées, elles le sont au niveau de la sortie standard.

Les fonctions : `p`, `l` et `=`

- `p` (print) affiche la ligne sélectionnée sur la sortie standard. Elle invalide l'option `-n`.
- `l` (list) affiche la ligne sélectionnée sur la sortie standard avec en plus les caractères de contrôles en clair avec leur code ASCII (deux chiffres en octal).
- `=` donne le num de la ligne sélectionnée sur la sortie standard.

Ces trois commandes sont utiles pour le débogage, (mise au point des programmes `sed`)
`sed "/toto/=" fichier` : afficher le numéro de la ligne contenant la chaîne `toto`.

Les fonctions : `q`, `r` et`w`

- `q` (quit) interrompre l'exécution de `sed`, la ligne en cours de traitement est affichée sur la sortie standard (uniquement si `-n` n'a pas été utilisée).
- `r` (read) lit le contenu d'un fichier et écrit le contenu sur la sortie standard.
- `w` (write) écrit la ligne sélectionnée dans un fichier.

`sed "/^ toto/w resultat" fichier` : Ecrire dans le fichier `resultat` toutes les lignes du fichier `fichier` commençant par la chaîne `toto`.

Cours 4 : Commandes find, sed, sort, wc, uniq et cut

Rabii El Ghorfi

La commande `find` permet de retrouver des fichiers à partir de certains critères.

Syntaxe :

```
find <répertoire de recherche> <critères de recherche>
```

critères de recherche :

`-name`	recherche sur le nom du fichier
`-perm`	recherche sur les droits d'accès
`-link`	recherche sur le nombre de liens
`-user`	recherche sur le propriétaire
`-group`	recherche sur le groupe auquel appartient le fichier
`-type`	recherche sur le type
`-size`	recherche sur la taille
`-atime`	recherche sur la date de dernier accès en lecture
`-mtime`	recherche sur la date de dernière modification du fichier
`-ctime`	recherche sur la date de création du fichier

On peut combiner les critères avec des opérateurs logiques :

- `critère1 critère2` ou `critère1 -a critère2` ≡ au ET logique,
- `!critère` ≡ NON logique,
- `\(critère1 -o critère2\)` ≡ OU logique,

L'option `-print` est indispensable pour obtenir une sortie.

Remarque

La commande `find` est récursive, *i.e.* scruter dans les répertoires, et les sous répertoires qu'il contient.

Recherche par nom de fichier

Pour chercher un fichier dont le nom contient la chaîne de caractères `toto` à partir du répertoire `/usr` :

```
find /usr -name toto -print
```

- Si le(s) fichier(s) existe(nt) ⇒ sortie : `toto`
- En cas d'échec, vous n'avez rien.

Pour rechercher tous les fichiers se terminant par `.c` dans le répertoire `/usr` :

```
find /usr -name " *.c " -print
```

⇒ toute la liste des fichiers se terminant par `.c` sous les répertoires contenus dans `/usr` (et dans `/usr` lui même).

Recherche suivant la date de dernière modification

Ex :Les derniers fichiers modifiés dans les 3 derniers jours dans toute l'arborescence `(/)` : `find / -mtime 3 -print`

Recherche suivant la taille

Ex :Connaître dans toute l'arborescence, les fichiers dont la taille dépasse 1Mo (2000 blocs de 512Ko) :

```
find / -size 2000 -print
```

Recherche combinée

Ex :Chercher dans toute l'arborescence, les fichiers ordinaires appartenant à olivier, dont la permission est fixée à 755 :

```
find / -type f -user olivier -perm 755 -print
```

Ex : Recherche des fichiers qui ont pour nom `a.out` et des fichiers se terminant par `.c` :

```
find . \ ( -name a.out -o -name " *.c " \ ) -print
```

Commandes en option :
En dehors de `-print` on dispose de l'option `-exec`. Le `find` couplé avec `exec` permet d'exécuter une commande sur les fichiers trouvés d'après les critères de recherche fixés. Cette option attend comme argument une commande, suivie de {}\ .
Ex : recherche des fichiers ayant pour nom `core`qu'on efface
`find . -name core -exec rm {}\`
Ex : les fichiers ayant pour nom `core` seront détruits, pour avoir une demande de confirmation avant l'exécution de `rm` :
`find . -name core -ok rm {}\`
Autres subtilités : Une fonction intéressante de `find` est de pouvoir être utilisé avec d'autres commandes
Ex : `find . -type f -print | xargs grep toto`
Rechercher dans le répertoire courant tous les fichiers normaux (sans fichiers spéciaux), et rechercher dans ces fichiers tous ceux contenant la chaîne `toto`.

Appliquons ! ! !

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)

Appliquons ! ! !

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
   `find /var/log/ -name ''syslog''`

Appliquons ! ! !

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. $\cdots$ Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
   `find / -name "syslog"`

Recherche d'un fichier find
La commande sed
La commande sort
La commande wc
La commande uniq
La commande cut

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`

Recherche d'un fichier find
La commande sed
La commande sort
La commande wc
La commande uniq
La commande cut

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`
   `find ~ -size +10M`

Recherche d'un fichier find
La commande sed
La commande sort
La commande wc
La commande uniq
La commande cut

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`
4. uniquement les rep qui s'appellent `syslog` (et pas les fichiers)

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`
4. uniquement les rep qui s'appellent `syslog` (et pas les fichiers)
   `find /var/log -name ‘‘syslog‘‘ -type d`

Recherche d'un fichier find
La commande sed
La commande sort
La commande wc
La commande uniq
La commande cut

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`
4. uniquement les rep qui s'appellent `syslog` (et pas les fichiers)
5. Imaginons que je souhaite mettre un `chmod` à 600 pour chacun de mes fichiers `jpg`, pour que je sois le seul à pouvoir les lire

Unix/Linux III

Recherche d'un fichier find
La commande sed
La commande sort
La commande wc
La commande uniq
La commande cut

Appliquons !!!

1. Retrouver tous les fichiers qui s'appellent `syslog` situés dans `/var/log` (et ses sous-rep)
2. ··· Et si je veux rechercher sur tout le disque dur, et pas seulement dans un dossier ?
3. rechercher tous les fichiers qui font plus de 10 Mo dans mon `home`
4. uniquement les rep qui s'appellent `syslog` (et pas les fichiers)
5. Imaginons que je souhaite mettre un `chmod` à 600 pour chacun de mes fichiers `jpg`, pour que je sois le seul à pouvoir les lire
   `find ~ -name "*.jpg" -exec chmod 600 {} +`

Unix/Linux III

`sed` est éditeur ligne non interactif, il lit les lignes d'un fichier une à une (ou provenant de l'entrée standard) leur applique un certain nombre de commandes d'édition et renvoie les lignes résultantes sur la sortie standard. Il ne modifie pas le fichier traité, il écrit tout sur la sortie standard.

Syntax `sed -e 'programme sed' fichier-a-traiter`
ou `sed -f fichier-programme fichier-a-traiter`

On dispose de l'option `-n` qui supprime la sortie standard par défaut, `sed` va écrire uniquement les lignes concernées par le traitement (sinon il écrit tout même les lignes non traitées).
L'option `-e` n'est pas nécessaire quand on a une seule fonction d'édition.
`sed` est une commande très riche (pour plus de détails `man sed`)

La fonction de substitution : `s`

`s` permet de changer la 1^{re} ou toutes les occurences d'une chaîne par une autre.

Syntaxe :

- `sed "s/toto/TOTO/" fichier` va changer la 1^{re} occurence de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/3" fichier` va changer la 3^{me} occurence de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/g" fichier` va changer toutes les occurences de la chaîne `toto` par `TOTO`
- `sed "s/toto/TOTO/p" fichier` en cas de remplacement imprime les lignes concernées
- `sed "s/toto/TOTO/w resultat" fichier` en cas de substitution la ligne en entrée est inscrite dans un fichier résultat

La fonction de substitution peut être utilisée avec une expression régulière.
`sed -e "s/[Ff]raise/FRAISE/g" fichier` substitue toutes les chaînes `Fraise` ou `fraise` par `FRAISE`

La fonction de suppression : `d`

La fonction de suppression `d` supprime les lignes comprises dans un intervalle donné.

Syntaxe : `sed "20,30d" fichier`

Cette commande va supprimer les lignes 20 à 30 du fichier `fichier`. On peut utiliser les expressions régulières :

- `sed "/toto/d" fichier` : supprime les lignes contenant la chaîne `toto`
- `sed "/toto/!d" fichier` : supprime toutes les lignes ne contenant pas la chaîne `toto`

En fait les lignes du fichier d'entrée ne sont pas supprimées, elles le sont au niveau de la sortie standard.

Les fonctions : `p`, `l` et `=`

- `p` (print) affiche la ligne sélectionnée sur la sortie standard. Elle invalide l'option `-n`.
- `l` (list) affiche la ligne sélectionnée sur la sortie standard avec en plus les caractères de contrôles en clair avec leur code ASCII (deux chiffres en octal).
- `=` donne le num de la ligne sélectionnée sur la sortie standard.

Ces trois commandes sont utiles pour le débogage, (mise au point des programmes `sed`)
`sed "/toto/=" fichier` : afficher le numéro de la ligne contenant la chaîne `toto`.

Les fonctions : `q`, `r` et `w`

- `q` (quit) interrompre l'exécution de `sed`, la ligne en cours de traitement est affichée sur la sortie standard (uniquement si `-n` n'a pas été utilisée).
- `r` (read) lit le contenu d'un fichier et écrit le contenu sur la sortie standard.
- `w` (write) écrit la ligne sélectionnée dans un fichier.

`sed "/^ toto/w resultat" fichier` : Ecrire dans le fichier `resultat` toutes les lignes du fichier `fichier` commençant par la chaîne `toto`.

La commande `sort` se révèle bien utile lorsqu'on a besoin de trier le contenu d'un fichier
sort : `sort sort.txt`
Le contenu du fichier est trié alphabétiquement et le résultat est affiché dans la console. Vous noterez que `sort` ne fait pas attention à la casse (majuscules / minuscules).
Le fichier en lui-même n'a pas été modifié lorsque nous avons lancé la commande. Seul le résultat était affiché dans la console. Vous pouvez faire en sorte que le fichier soit modifié en précisant un nom de fichier avec l'option `-o` :
`sort -o nomstries.txt sort.txt`

La commande `wc` signifie *Word Count*. C'est donc a priori un compteur de mots, mais en fait on lui trouve plusieurs autres utilités : compter le nombre de lignes (très fréquent) et compter le nombre de caractères.
Comme les précédentes, la commande `wc` travaille sur un fichier. Sans paramètres, les résultats renvoyés par `wc` sont un peu obscurs :

`wc sort.txt` $\Longrightarrow$ `27 27 220 sort.txt`

Ces 3 nombres signifient, dans l'ordre :

1. Le nombre de lignes
2. Le nombre de mots
3. Le nombre d'octets

Les options :

1. `-l` : compter le nombre de lignes
2. `-w` : compter le nombre de mots
3. `-c` : compter le nombre d'octets
4. `-m` : compter le nombre caractères

Parfois, certains fichiers contiennent des lignes en double et on aimerait pouvoir les détecter ou les supprimer. La commande `uniq` est toute indiquée pour cela. Nous devons travailler sur un fichier trié. En effet, la commande `uniq` ne repère que les lignes successives qui sont identiques.

```
uniq doubl.txt
```

Vous pouvez demander à ce que le résultat sans doublons soit écrit dans un autre fichier plutôt qu'affiché dans la console :

```
uniq doubl.txt sansdoubl.txt
```


Les options :

1. `-d` : afficher uniquement les lignes présentes en double
2. `-c` : compter le nombre d'occurences

Vous avez déjà coupé du texte dans un éditeur de texte, non ? La commande `cut` vous propose de faire cela au sein d'un fichier, afin de conserver uniquement une partie de chaque ligne.

Couper selon le nombre de caractères

Par exemple, si vous souhaitez conserver uniquement les caractères 2 à 5 de chaque ligne du fichier :

```
cut -c 2-5 sort.txt
```

Pour conserver du 1er au 3me caractère

```
cut -c -3 sort.txt
```

pour conserver du 3me au dernier caractère

```
cut -c 3- sort.txt
```


Couper selon un délimiteur

Faisons maintenant quelque chose de bien plus intéressant. Plutôt que de s'amuser à compter le nombre de caractères, on va travailler avec ce qu'on appelle un délimiteur. Prenons un cas pratique : les fichiers notes.

Imaginons que nous souhaitons extraire de ce fichier la liste des prénoms. Comment nous y prendrions-nous ?

Couper selon un délimiteur

Faisons maintenant quelque chose de bien plus intéressant.
Plutôt que de s'amuser à compter le nombre de caractères, on va travailler avec ce qu'on appelle un délimiteur. Prenons un cas pratique : les fichiers notes.
Imaginons que nous souhaitons extraire de ce fichier la liste des prénoms. Comment nous y prendrions-nous ?
Nous allons donc nous servir du fait que nous savons que la virgule sépare les différents champs dans ce fichier. Vous allez avoir besoin d'utiliser 2 paramètres :

- `-d` :indique quel est le délimiteur dans le fichier
- `-f` :indique le numéro du ou des champs à couper

Dans notre cas, le délimiteur qui sépare les champs est la virgule. Le numéro du champ à couper est 1 (c'est le premier).

```
cut -d , -f 1 notes.txt
```

Après le `-d`, nous avons indiqué quel était le délimiteur (à savoir la virgule).
Après le `-f`, nous avons indiqué le numéro du champ à conserver (le premier).

Cours 5 : Les flux de redirection

Rabii El Ghorfi

Vous devriez maintenant avoir l'habitude d'un certain nombre de commandes que propose la console de `Linux`. Le fonctionnement est toujours le même :

1. Vous tapez la commande (par exemple `ls`)
2. Le résultat s'affiche dans la console

Toutefois, au lieu d'afficher le résultat dans la console, vous allez pouvoir l'envoyer ailleurs.
Où ? Dans un fichier, ou en entrée d'une autre commande pour "chaîner des commandes". Ainsi, le résultat d'une commande peut en déclencher une autre !
Comment ? A l'aide de petits symboles spéciaux, appelés flux de redirection

La manipulation la plus simple que nous allons voir va nous permettre d'écrire le résultat d'une commande dans un fichier, au lieu de l'afficher simplement dans la console.
Préparatifs : Prenons par exemple la commande `cut` et le fichier `telephone.txt`
La commande `cut` nous avait permis de découper une partie du fichier et d'afficher le résultat dans la console. Par exemple

```
cut -f 2 telephone.txt
```

Ce résultat s'est affiché dans la console. C'est ce que font toutes les commandes par défaut... à moins que l'on utilise un flux de redirection !

> : rediriger dans un nouveau fichier

Supposons que nous souhaitions écrire la liste des prénoms dans un fichier, afin de garder sous le coude la liste des prénoms des personnes inscrit dans l'annuaire `telephone.txt`.
C'est là qu'intervient le petit symbole magique > (appelé chevron)
Ce symbole permet de rediriger le résultat de la commande dans le fichier de votre choix

```
cut -f 2 telephone.txt > people.txt
```

Attention : si le fichier existait déjà il sera écrasé sans demande de confirmation !

» : rediriger à la fin d'un fichier

Le double chevron » sert lui aussi à rediriger le résultat dans un fichier, mais cette fois à la fin de ce fichier.
Avantage : vous ne risquez pas d'écraser le fichier s'il existe déjà. Si le fichier n'existe pas, il sera créé automatiquement.
Normalement, on devrait avoir créé un fichier `people.txt` lors des manipulations précédentes. Si vous faites :

```
cut -f 3 telephone.txt » people.txt
```

les commandes produisent 2 flux de données différents :

1. La sortie standard : pour tous les messages (sauf les erreurs)
2. La sortie d'erreurs : pour toutes les erreurs

Lorsque l'on applique la commande `cat` sur un fichier `test.txt` pour afficher son contenu. Il y a 2 possibilités :

- Si tout va bien : le résultat (le contenu du fichier) s'affiche sur la sortie standard
- S'il y a une erreur : celle-ci s'affiche dans la sortie d'erreurs.

Par défaut, tout s'affiche dans la console : la sortie standard comme la sortie d'erreurs
Alors...Essayons

```
cut -d , -f 1 fichier_inexistant.txt > standard.txt
```


Rediriger les erreurs dans un fichier à part

On pourrait souhaiter "logger" les erreurs dans un fichier d'erreurs à part pour ne pas les oublier et pour pouvoir les analyser ensuite. Pour cela, on utilise l'opérateur `2>`.

```
cut -d , -f 1 fich_inexist.txt > standard.txt 2>
erreurs.log
```

Il y a deux redirections ici :

1. `> standard.txt` : redirige le résultat de la commande (sauf les erreurs) dans le fichier eleves.txt. C'est la sortie standard.
2. `2> erreurs.log` : redirige les erreurs éventuelles dans le fichier erreurs.log. C'est la sortie d'erreurs.

Fusionner les sorties

Parfois, on n'a pas envie de séparer les informations dans 2 fichiers différents. Heureusement, il est possible de fusionner les sorties dans un seul et même fichier.
Comment ? Il faut utiliser le code suivant :`2>&1` Cela a pour effet de rediriger toute la sortie d'erreurs dans la sortie standard.

```
cut -d , -f 1 fichier_inexistant.txt > eleves.txt 2>&1
```


Pour le moment, nous avons redirigé uniquement la **sortie** des commandes. Nous avons décidé où envoyer les messages issus de ces commandes.
Maintenant, faisons l'inverse, *i.e.* de décider **d'où vient l'entrée d'une commande**!!!.
Jusqu'alors, l'entrée venait des paramètres de la commande... mais on peut faire en sorte qu'elle vienne d'un fichier ou d'une saisie au clavier!!!

< : lire depuis un fichier

Le chevron ouvrant < permet d'indiquer d'où vient l'entrée qu'on envoie à la commande.

```
cat < telephone.txt
```

On faisait pas pareil avant en écrivant juste `cat telephone.txt` par hasard ?
Si. Le fait d'écrire `cat < telephone.txt` est strictement identique à écrire `cat telephone.txt` ... du moins en apparence.

Le résultat produit est le même, mais ce qui se passe derrière est très différent :

- `cat telephone.txt` : la commande `cat` reçoit en entrée le nom du fichier "telephone.txt" qu'elle doit ensuite se charger d'ouvrir pour afficher son contenu.
- Si vous écrivez `cat < telephone.txt` : la commande `cat` reçoit le contenu de "telephone.txt" qu'elle se contente simplement d'afficher dans la console. C'est le shell (le programme qui gère la console) qui se charge d'envoyer le contenu de "telephone.txt" à la commande `cat`

2 façons de faire la même chose mais de manière très différente.

« : lire depuis le clavier progressivement

Le double chevron ouvrant « fait quelque chose d'assez différent : il vous permet d'envoyer un contenu à une commande avec votre clavier. Cela peut s'avérer très utile.

```
sort -n « FIN
```

Cela évite d'avoir à créer un fichier dont on a pas besoin.
On peut faire la même chose avec une autre commande comme par exemple `wc` pour compter le nombre de mots ou de caractères.

```
wc -m « FIN
```

Rq : finir par un mot-clé qui sert à indiquer la fin de la saisie.

Nous pouvons tout à fait combiner ces symboles avec ceux qu'on a vus précédemment. Par exemple :

```
sort -n « FIN > nombres_tries.txt 2>&1
```

Définition

Un processus est un programme en cours d'exécution. On distingue deux types de processus :

- Le processus système (daemons) : assure des services généraux accessibles à tous les utilisateurs du système. Le propriétaire est le `root` et il n'est sous le contrôle d'aucun terminal.
- Le processus utilisateur : dédié à l'exécution d'une tâche particulière. Le propriétaire est l'utilisateur qui l'exécute et il est sous le contrôle du terminal à partir duquel il a été lancé.

Création

Toute exécution d'un programme déclenche la création d'un processus dont la durée de vie = la durée d'exécution du programme.

Le système alloue à chaque processus un numéro d'identification unique : `PID` (Process IDentifier).

Tout processus est créé par un autre processus : son processus père

Exemples :

Lorsqu'un utilisateur lance une commande, un processus est créé dont le père est le processus correspondant à l'exécution du shell.

États d'un processus :

- Prêt (R) : le processus attend que le processeur lui soit affecté.
- Actif (R) : le processeur exécute le processus.
- Endormi (S pour <20s ou I pour >20s) : le processus est en attente de l'arrivée d'un évènement (par exemple, une réponse du terminal).
- Zombi (Z) : le processus a pris fin sans libérer ses ressources.
- Suspendu (T) : le processus a été interrompu et attend l'arrivée d'un signal de reprise.

Exécution d'une commande :

Cinq modes d'exécution d'une commande sous Unix :

- mode interactif : commande lancée à partir d'un terminal. Le contrôle du terminal n'est rendu à l'utilisateur qu'à la fin de l'exécution de la commande.
  `<ctrl-c>` : interrompre la commande
  `<ctrl-z>` : suspendre la commande
- mode en arrière plan : permet de rendre immédiatement le contrôle à l'utilisateur (commande lancée suivie du caractère &). Si le terminal est fermé, la commande en arrière plan est interrompue automatiquement. Pour éviter ce problème, il faut lancer la commande sous le contrôle de la commande `nohup` (syntaxe : `nohup nom_commande &`).

- mode différé : `at` permet de déclencher l'exécution d'une commande à une date fixée.
  ex : `at 16:50 10/06/08 < commande`
  démarre le lancement du contenu de commande le 06 octobre 2008 à 16h50.
  (`at -l` pour lister et `at -r` pour supprimer)
- mode batch : permet de placer une commande dans une file d'attente.
- mode cyclique : tâche exécutée de façon cyclique.

La commande `ps`

Visualiser les processus avec la commande : `ps (options)`, les options les plus intéressantes

- `-e` : affichage de tous les processus
- `-f` : affichage détaillé

exemple : `ps -ef`

```
UID PID PPID C STIME TTY TIME COMMAND
root 1 0 0 Dec 6 ? 1:02 init
...
jean 319 300 0 10:30:30 ? 0:02 /usr/dt/bin/dtsession
olivier 321 319 0 10:30:34 ttyp1 0:02 csh
olivier 324 321 0 10:32:12 ttyp1 0:00 ps -ef
```

La signification des différentes colonnes :

`UID`	nom du user qui a lancé le process
`PID`	num. du process
`PPID`	num. du process parent
`C`	facteur de priorité : $\uparrow \Rightarrow$ prioritaire
`STIME`	heure de lancement du processus
`TTY`	nom du terminal
`TIME`	durée de traitement du processus
`COMMAND`	nom du process

exemple précédent : `ps -ef`
le 1^{er} p_i a pour `PID 321`, le 2^{me} 324. Le PPID du process `" ps -ef "` est 321 qui correspond au shell, par conséquent le shell est le process parent, de la commande qu'on vient de taper.
Pour voir les process d'un seul utilisateur : `ps -u olivier`

Les signaux

Il est possible d'agir sur le déroulement d'un p_i en lui envoyant un signal. Unix définit de façon standard un certain nombre de signaux dont :

- `SIGINT` (signal 2) : interrompre l'exécution d'un p_i,
- `SIGKILL` (signal 9) : arrêt définitif l'exécution d'un p_i (ne peut pas être ignoré),
- `SIGTSTP` (signal 24) : suspendre temporairement l'exécution d'un p_i,
- `SIGCONT` (signal 25) : reprendre l'exécution d'un p_i précédemment suspendu par l'envoi d'un signal `SIGTSTP`.

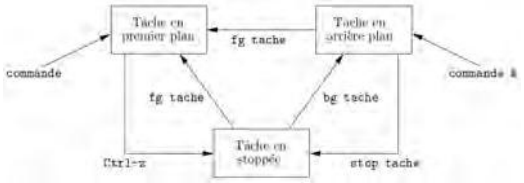
Un signal peut être envoyé par :

1. le système (ex : signaux d'erreur),
2. un autre p_i,
3. l'utiliateur :
   - soit l'utilisateur tape des caractères provoquant l'envoi d'un signal au p_i en cours d'exécution sur le terminal (ex : `<ctrl-z>` pour `SIGTSTP`, `<ctrl-c>` pour `SIGINT`),
   - soit l'utilisateur utilise la commande `kill` pour envoyer un signal à un ou plusieurs p_i lorsqu'il n'a pas accès au terminal de rattachement des p_i ou lorsque ces derniers sont exécutés en arrière plan.
     La syntaxe est la suivante :
     `kill -signal PID`
     `signal` : nom symbolique ou num.
     `PID` : donné par `ps`

Le job control

Un job est une ligne de commande shell. Chaque job est numéroté de 1 à N par le shell. Un job peut se trouver dans trois états :

- avant plan : Le job s'exécute et vous n'avez pas la main sur le shell.
- arrière plan : Le job s'exécute et vous avez la main sur le shell.
- suspendu : Le job est en attente, il ne s'exécute pas.



Quelques commandes

`jobs`	:	permet de lister les jobs en cours
`<ctrl-z>`	:	suspendre une commande en avant plan
`stop %num_job`	:	suspendre une commande en arrière plan
`bg %num_job`	:	basculer de suspendu à arrière plan
`fg %num_job`	:	passer en avant plan
`kill %num_job`	:	tuer un processus

Il est possible sur une même ligne de commande de lancer plusieurs p_i.

Lancement en séquence de plusieurs commandes

`commande1 ; commande2 ; commande3 > fich`

création du p_i qui exécute `commande1`
puis quand `commande1` est terminé,
création du p_i qui exécute `commande2`
...
Les différents p_i ne co-existent pas.
Seule la S standard de `commande3` est redirigée vers `fich`. Pour rediriger les trois commandes :
`(commande1 ; commande2 ; commande3) > fich`.

Lancement concurrent de p_i communiquant par un tube (pipe)
L'exécution simultanée de plusieurs p_i échangeant des données par l'intermédiaire de zones mémoires tubes.
Syntaxe : `com1 | com2 | ... | comn`
Le système crée $n-1$ tubes et n p_i.

- le p_i qui exécute `com1` a sa S standard redirigée vers le 1^{er} tube, son E standard étant celle définie par défaut (clavier),
- le p_i qui exécute `com2` a son E standard redirigée vers le 1^{er} tube et sa S standard redirigée vers le 2^{me} tube, ...
- le p_i qui exécute `comn` a son E standard redirigée vers le $n-1^{me}$ tube, sa S standard étant celle par défaut (l'écran).

Les n p_i coexistent et se partagent l'accès au processeur. Le système se charge de leur synchronisation : les p_i lecteurs sont mis en attente tant que leur tube en entrée est vide, les p_i écrivains sont mis en attente si leur tube en sortie est plein.
Exemple
`ls /bin /usr/local/bin | grep "cp" | wc -l`
permet de compter le nombre de fichiers dont le nom contient la chaîne `cp` dans les répertoires `/bin` et `/usr/local/bin`.
La commande `tee` permet de rediriger la sortie standard d'un processus vers un fichier et vers la sortie standard.
Exemple :
`ls /bin | grep "cp" | tee fich | wc -l`

la programmation shell. De quoi s'agit-il ?

Imaginez un mini-langage de programmation intégré à Linux. Ce n'est pas un langage aussi complet que peuvent l'être le `C`, le `C++` ou le `Java` par exemple, mais cela permet d'automatiser la plupart de vos tâches. Voici un aperçu de ce qu'on peut faire avec :

- Sauvegarde de vos données
- Surveillance de la charge de votre machine
- Système de gestion personnalisé de vos téléchargements
- ...etc

Pourquoi pas le C ?

Le gros avantage des **scripts shell**, c'est qu'ils sont totalement intégrés à Linux : il n'y a rien à installer et rien à compiler. Et surtout : vous avez trés peu de nouvelles choses à apprendre. En effet, toutes les commandes que l'on utilise dans les **scripts shells** sont des commandes du système que vous connaissez déjà : **ls, cut, grep, sort**

shell : Un interpréteur de commandes

Les fonctionnalités offertes par l'invite de commande peuvent varier en fonction du **shell** que l'on utilise.
Les principaux sont

- **sh** : Bourne Shell. L'ancêtre de tous les shells.
- **bash** : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.
- **ksh** : Korn Shell. Un shell puissant présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- **csh** : C Shell. Un shell utilisant une syntaxe proche du C.
- **tcsh** : Tenex C Shell. Amélioration du C Shell.
- **zsh** : Z Shell. Shell assez récent reprenant les meilleures idées de **bash**, **ksh** et **tcsh**.

A quoi sert un shell

Shell : programme qui gère l'invite de commandes. C'est donc le programme qui attend que vous rentriez des commandes :

- Se souvenir quelles étaient les dernières commandes tapées
- Auto-complétion d'une commande ou d'un nom de fichier lorsque vous appuyez sur **Tab**
- Gérer les processus (envoi en arrière-plan, mise en pause avec Ctrl + Z ...).
- Rediriger et chaîner les commandes (les fameux symboles $>$, $<$, | ...)

Avec quel shell écrire nos scripts alors ? bash

- On le trouve par défaut sous Linux et Mac OS X (cela couvre assez de monde !).
- Il rend l'écriture de scripts plus simple que **sh**.
- Il est plus répandu que **ksh** et **zsh** sous Linux.

En clair, le **bash** est un bon compromis entre **sh** (le plus compatible) et **ksh/zsh** (plus puissants).

- Commençons par créer un nouveau fichier pour notre script : `gedit essai.sh` $\longrightarrow$ fichier vide
- La première chose à faire dans un script shell est d'indiquer... quel shell est utilisé : Rajouter dans `essai.sh` la ligne

  `#!/bin/bash`

  le #! est appelé le sha-bang
- Après le sha-bang, nous pouvons commencer à coder. Le principe : Ecrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

Exple :

```
#!/bin/bash
ls
```


- Donner les droits d'exec au script

  `chmod +x essai.sh`
- Exécuter le script, en tapant "`./`" devant le nom du script

  `./essai.sh`

Que fait le script ? Il fait juste un `ls`, donc il affiche la liste des fichiers dans le répertoire.
On peut vouloir préciser en plus le rep courant :

```
#!/bin/bash
pwd
ls
```


Créer sa propre commande :

Actuellement, le script doit être lancé via `./essai.sh` et vous devez être dans le bon répertoire.
Comment font les autres programmes pour pouvoir être exécutés depuis n'importe quel répertoire sans "./" devant ? Ils sont placés dans un des rep. du `PATH`.
Def : Le `PATH` est une variable système qui indique où sont les programmes exec. Si vous tapez echo $PATH , vous aurez la liste de ces rep.$\longrightarrow$ déplacer ou copier le script dans un de ces rep, (`/bin, ou /usr/bin, ou /usr/local/bin`).
Rq : Il faut être `root` pour pouvoir faire ça.

Comme dans tous les langages de programmation, on trouve en `bash` ce qu'on appelle des variables.
$\longrightarrow$ stocker temporairement des informations en mémoire. C'est en fait la base de la programmation.
Les variables en `bash` sont particulières. Il faut être très rigoureux lorsqu'on les utilise $\longrightarrow$ différent du `C`

Variables :

- Un nom
- Une valeur

Exple

```
message='Bonjour tout le monde'
```

Rq :Pas d'espace autour de "="
Executons !!! `./var.sh`

`echo` : afficher une variable

Exple

- `echo "Salut tout le monde"`
- `echo -e "Message\n Autre ligne"`

`./varaffich.sh`

Les quotes

- Les apostrophes ' '(simples quotes)
  `./simplequote.sh`
- Les guillemets " " (doubles quotes)
  `./doublequote.sh`
- Les accents graves ` ` (back quotes)
  `./backquote.sh`

`read`

Demander au user de saisir du texte avec la commande `read`. La façon la plus simple de l'utiliser est d'indiquer le nom de la variable dans laquelle le message saisi sera stocké :

`./read1.sh`

La commande read propose plusieurs options intéressantes.

- `-p` : afficher un message de prompt `./readp.sh`
- `-n` : limiter le nombre de caractères `./readn.sh`
- `-s` : ne pas afficher le texte saisi `./reads.sh`

En `bash`, les var. sont toutes des chaînes de caractères
⇒Incapable de manipuler des nombres ⇒ pas opérations !!

la commande `let`

`./calcul1.sh`

Les opérations :

- L'addition : $+$
- La soustraction : $-$
- La multiplication : $*$
- La division : $/$
- La puissance : $**$
- Le modulo : $\%$

Les var. que crées dans scripts `bash` n'existent que dans ces scripts. *ie.* une variable définie dans un `pgme A` ne sera pas utilisable dans un `pgme B`.
Les var. d'environnement :var. utilisables n'importe quel pgme. On parle aussi parfois de var. globales. Afficher toutes celles actuellement en mémoire avec la commande `env`.

Quelques variables d'environnement

- `SHELL` : type de shell est en cours d'utilisation `(sh, bash, ksh...)`
- `PATH` : une liste rep qui contiennent des exec que vous souhaitez pouvoir lancer sans indiquer leur rep.
- `EDITOR` :Editeur de txt par défaut
- `HOME` : position du dossier `home`
- `PWD` : Dossier courant

Rq : En majuscule

Les scripts `bash` acceptent des paramètres

`./varparam.sh param1 param2 param3`

- `$#` : contient le nombre de param.
- `$0` : contient le nom du script exécuté (ici `"./varparam.sh"`)
- `$1` : contient 1^r param.
- ...
- `$9` : contient 9^m param.

Syntaxe

```
if [ test ]
then
echo "true"
fi
```

Rq : l'espace dans `[ test ]`

`./if1.sh`

Syntaxe

```
if [ test ]
then
echo "true"
else
echo "false"
fi
```

`./if2.sh param1`

Syntaxe

```
if [ test ]
then
echo "premier test a été verif"
elif [ autre_test ]
echo "second test a été verif"
elif [ encore_autre_test ]
echo "troisième test a été verif"
else
echo "Aucun des tests prec. n'a été vérifié"
fi
```

`./if3.sh param1`

3 types de tests différents en `bash` :

1. Tests sur des chaînes de caractères
2. Tests sur des nombres
3. Tests sur des fichiers

- `$chaine1 = $chaine2`
  teste si 2 chaînes sont identiques. `B` $\neq$ `b` (sensible à la casse...)
- `$chaine1 != $chaine2`
  Teste si 2 chaînes sont $\neq$
- `-z$chaine`
  Teste si 1 chaînes est vide
- `-n$chaine`
  Teste si 1 chaînes est non vide

`./test1.sh param1 param2`

Exo :Ecrire un script qui teste l'existence d'un paramètre

- `$num1 -eq $num2` Teste si les nombres sont égaux(equal) "=" compare les caractères.
- `$num1 -ne $num2` Teste si les nombres sont diff(non equal) "!=" compare les caractères.
- `$num1 -lt $num2` Teste si num1 est < num2 (lower than)
- `$num1 -le $num2` Teste si num1 est <= num2 (lower or equal)
- `$num1 -gt $num2` Teste si num1 est > num2 (greater than)
- `$num1 -ge $num2` Teste si num1 est >= num2 (greater or equal)

`./test2.sh param1`

- `-e $nomfich` Teste si le fich. existe
- `-d $nomfich` Teste si le fich. est un rep.
- `-f $nomfich` Teste si le fich. est un... fich. Un vrai fich. pas un dossier.
- `-L $nomfich` Teste si fich est un lien symbolique
- `-r $nomfich` Teste si fich est lisible (r)
- `$fich1 -nt $fich2` Teste si fich1 est plus récent que fich2 (newer than) | `$fich1 -ot $fich2` (older than)

EXO Ecrire un script qui demande au user de rentrer le nom d'un rep et de verifier si c'est bien un rep

Effectuer plusieurs tests à la fois

Dans un `if`, il est possible de faire plusieurs tests à la fois.

- Si un test est vrai ET qu'un autre test est vrai : `&&`
- Si un test est vrai OU qu'un autre test est vrai : `||`

Rq :encadrer chaque condition par des crochets
EXO Ecrire un script qui vérifie qu'il a au moins un param et la valeur du 1^r param est `asticot`

Syntaxe

```
while [ test ]
do
echo 'Action en boucle'
done
```

```
while [ test ]; do
echo 'Action en
boucle' done
```

`./while1.sh`

La boucle `for` permet de parcourir une liste de valeurs, et de boucler autant de fois qu'il y a de valeurs.

Syntaxe

```
for variable in 'valeur1' 'valeur2' 'valeur3'
do
echo "La variable vaut $variable"
done
```

`./for1.sh`

Rq : La liste des valeurs n'a pas besoin d'être définie directement dans le code :

`./for2.sh`

EXO : Script qui renomme tous les fichiers trouvé

Un cas plus classique du `for`

```
for i in `seq 1 10`;
do
echo $i
done
```

`./for3.sh`

Pour faire des sauts de 2 faire `for i in `seq 1 2 10`;`