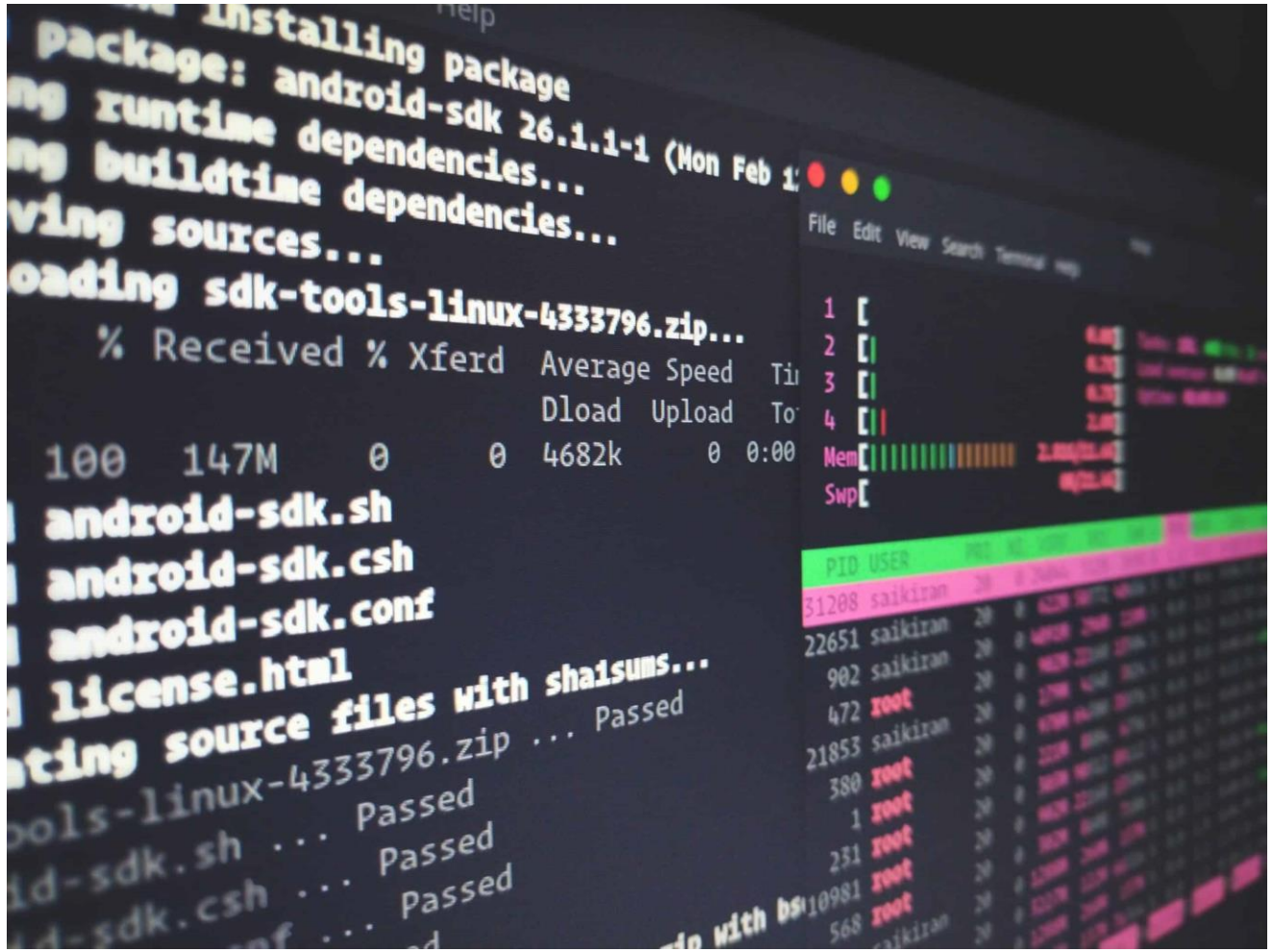
v1.01

# INFORMATIQUE 3

## VI. UNIX & SCRIPT SHELL



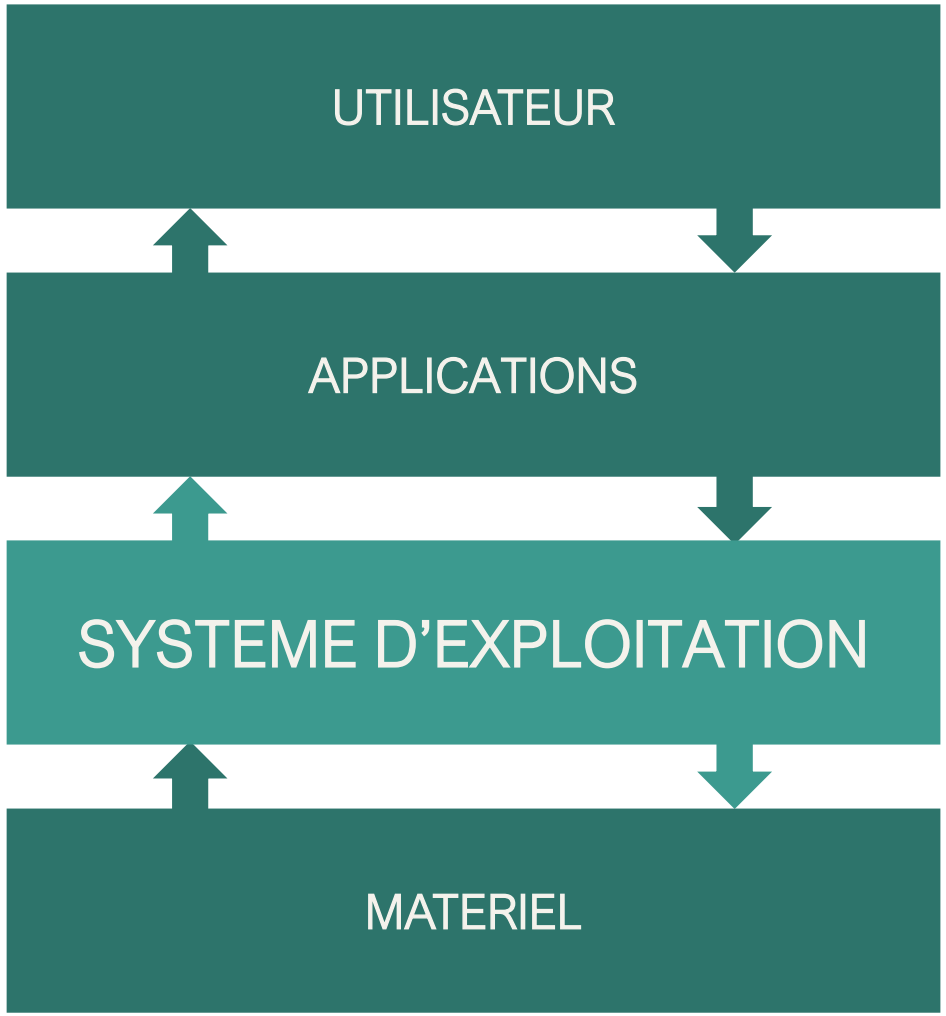

Eva ANSERMIN & Renaud VERIN & Romuald GRIGNON

CY TECH

# I. Contexte

# Contexte

- “Système d’exploitation” : logiciel permettant de faire le lien entre le matériel de la machine et les applications



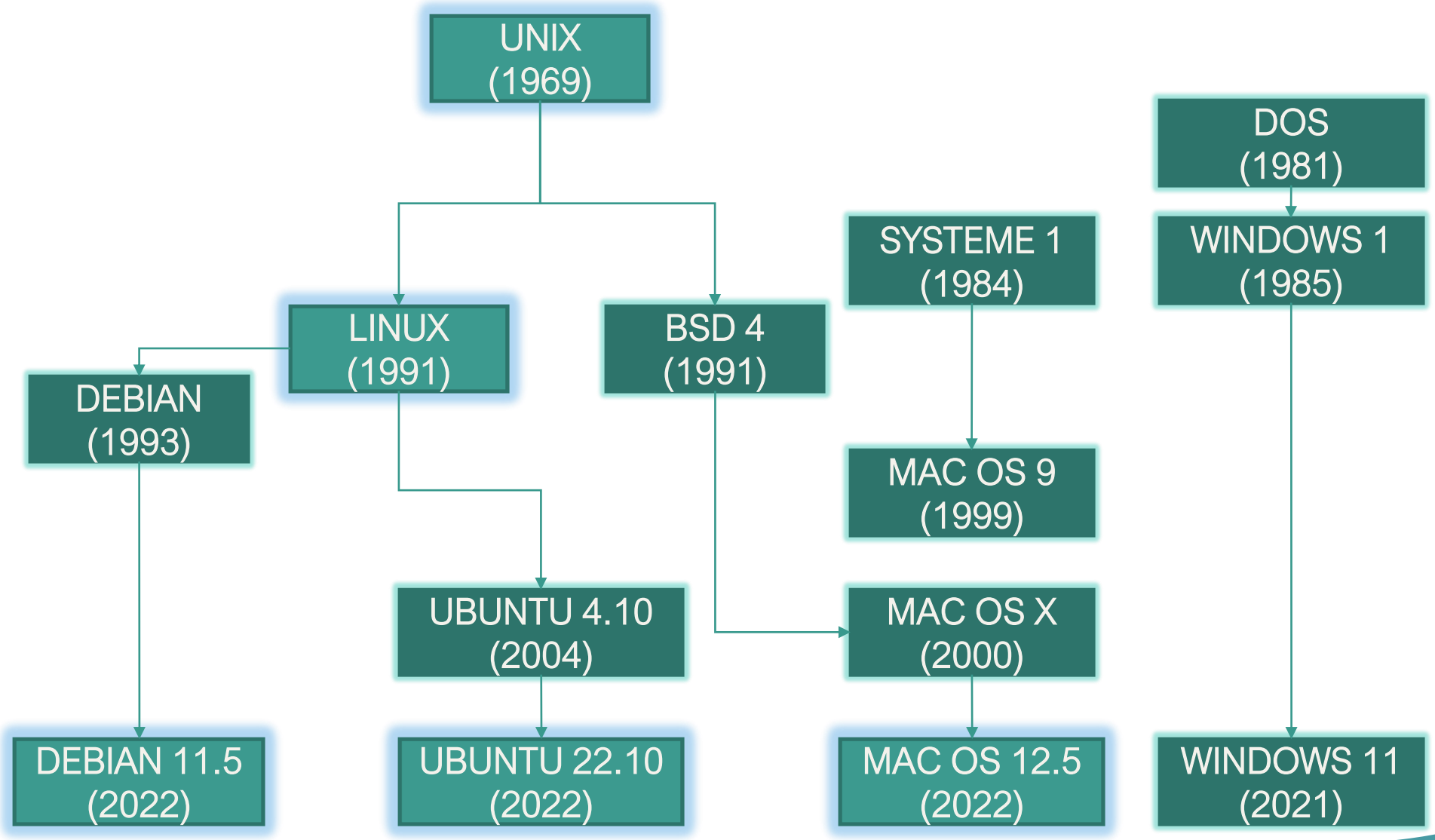
# Contexte

Le système d'exploitation permet de gérer plusieurs tâches nécessaires au bon fonctionnement de l'ordinateur :

- **Interface utilisateur**
  - Interprète de commandes
  - Interface graphique
- **Gestion de la mémoire vive**
  - Mémoire vive (RAM)
  - Allocation / Libération / Segmentation
  - Pour le noyau et pour les autres tâches
- **Gestion de la mémoire secondaire**
  - Mémoire non volatile (disque dur, clef USB, …)
  - Structure de stockage des données (fichiers)
- **Noyau**
  - Gestion des "processus"
  - Allocation de CPU
  - Gestion des interruptions
- **Gestion des périphériques**
  - Pilotes

# Contexte

- “Distribution” : logiciel complet regroupant un système d’exploitation, des pilotes et des applications



# II. Présentation des concepts

# UNIX

- Système d'exploitation
  - multi-tâches
  - multi-utilisateurs

- 4 concepts élémentaires :
  - les fichiers
  - les droits d'accès
  - les processus
  - la communication inter-processus

# UNIX

- Système d'exploitation
  - multi-tâches
  - multi-utilisateurs

- 4 concepts élémentaires :
  - les fichiers
  - les droits d'accès
  - les processus
  - la communication inter-processus

# UNIX : les fichiers

- **Unité élémentaire** de gestion des ressources
- Référencé dans un **système de fichiers** par :
  - le propriétaire
  - le groupe d'utilisateurs
  - les droits d'accès (lecture / écriture / exécution)
  - la date de dernière modification
  - la taille dans la mémoire secondaire
  - les références des blocs de données dans la mémoire secondaire
- Sous Unix, tout est vu comme un fichier (fichier, dossier, raccourci, flux de données entrant ou sortant, …)

# UNIX : les fichiers

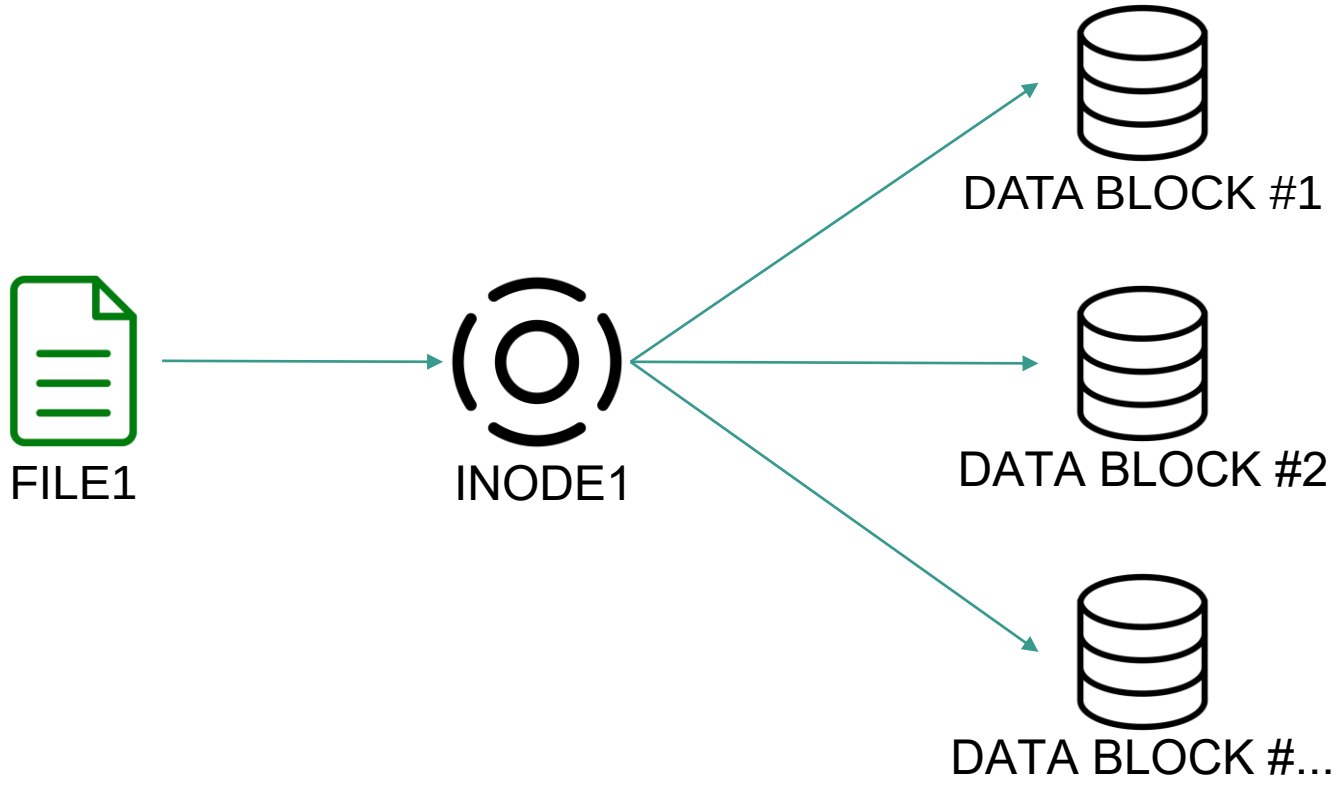
- Ce n'est **pas** le **fichier** en lui-même qui **contient** les **informations** décrites précédemment
- Elles sont stockées dans ce que l'on nomme un ***inode***

  *inode* ⇔ *index of node*

- Le fichier est une zone mémoire qui contient seulement le **nom** du fichier ainsi que le **numéro** d'**inode** associé
- Chaque inode possède un **numéro unique**
- Lorsque l'on souhaite afficher le contenu d'un fichier, en réalité, on accède à l'inode correspondant, qui lui-même accède au contenu

# UNIX : les fichiers

- Illustration d'un fichier dans la mémoire du système



# UNIX : les fichiers

- Il est possible de créer des liens vers des fichiers

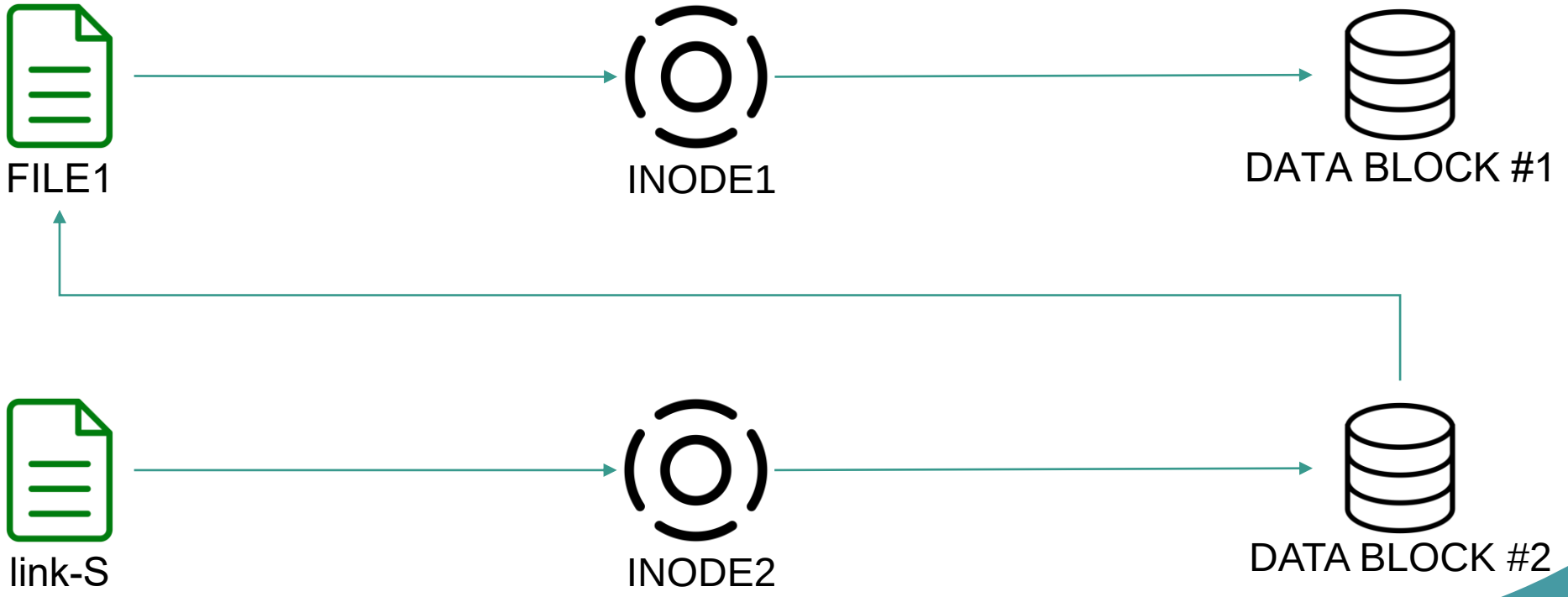
- Un peu à l'image des raccourcis, ces liens permettent d'accéder à du contenu qui est ailleurs dans l'arborescence des fichiers

- Il existe plusieurs types de liens
  - Les liens symboliques (liens souples)
  - Les liens physiques (liens durs)

# UNIX : les fichiers

- Liens symboliques : fichier avec son propre inode
- Si le fichier cible est supprimé, le lien ne fonctionne plus

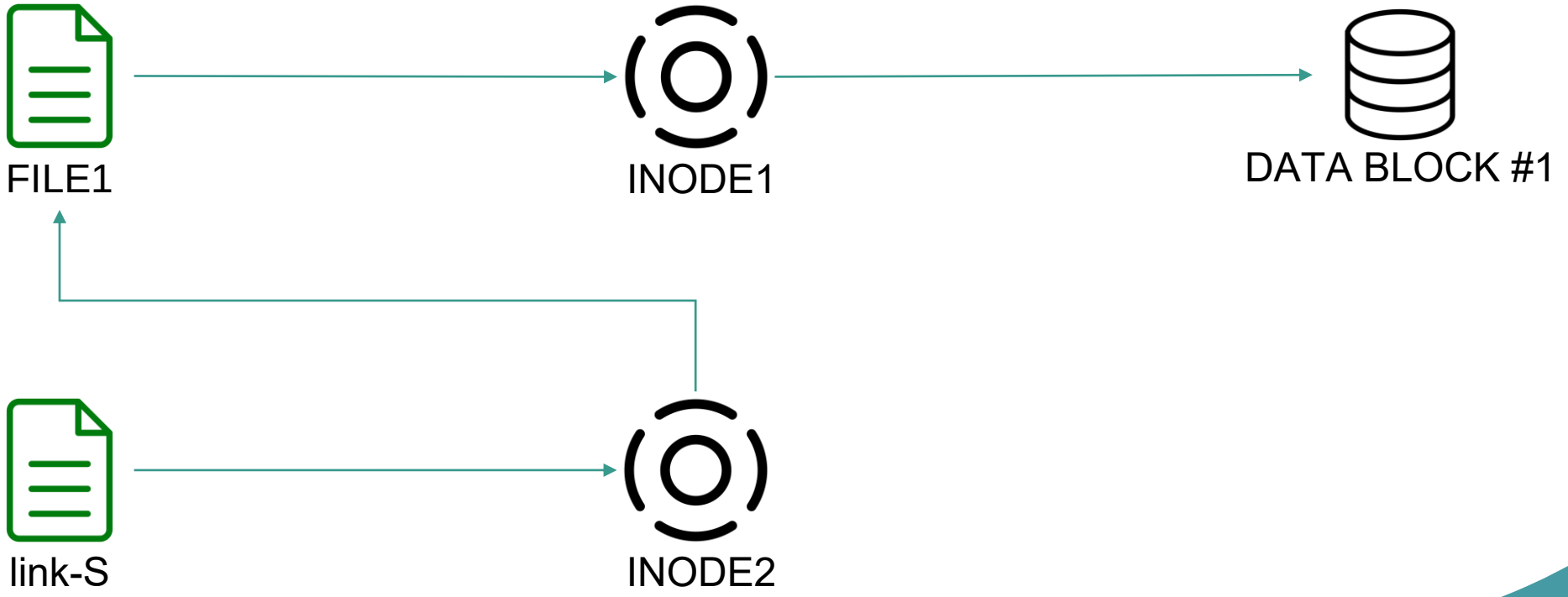
- Schéma de lien symbolique (fast)



# UNIX : les fichiers

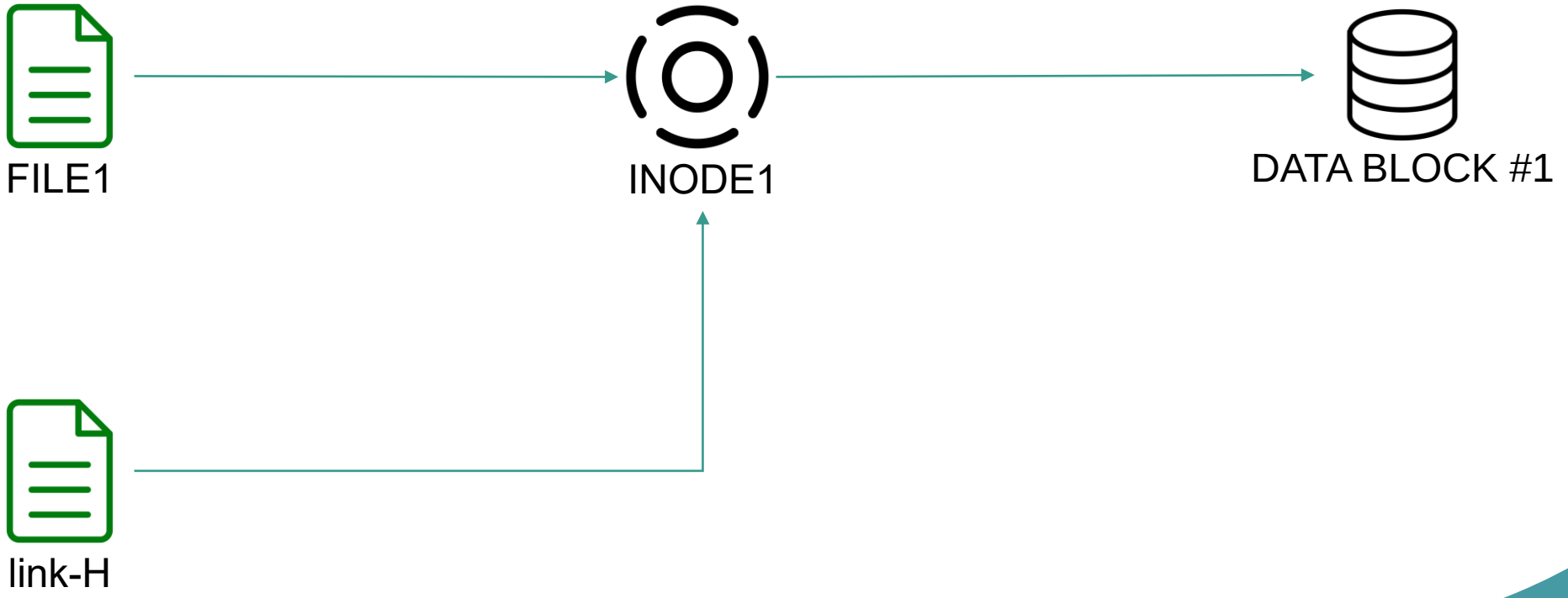
- Liens symboliques : fichier avec son propre inode
- Si le fichier cible est supprimé, le lien ne fonctionne plus

- Schéma de lien symbolique (fast)



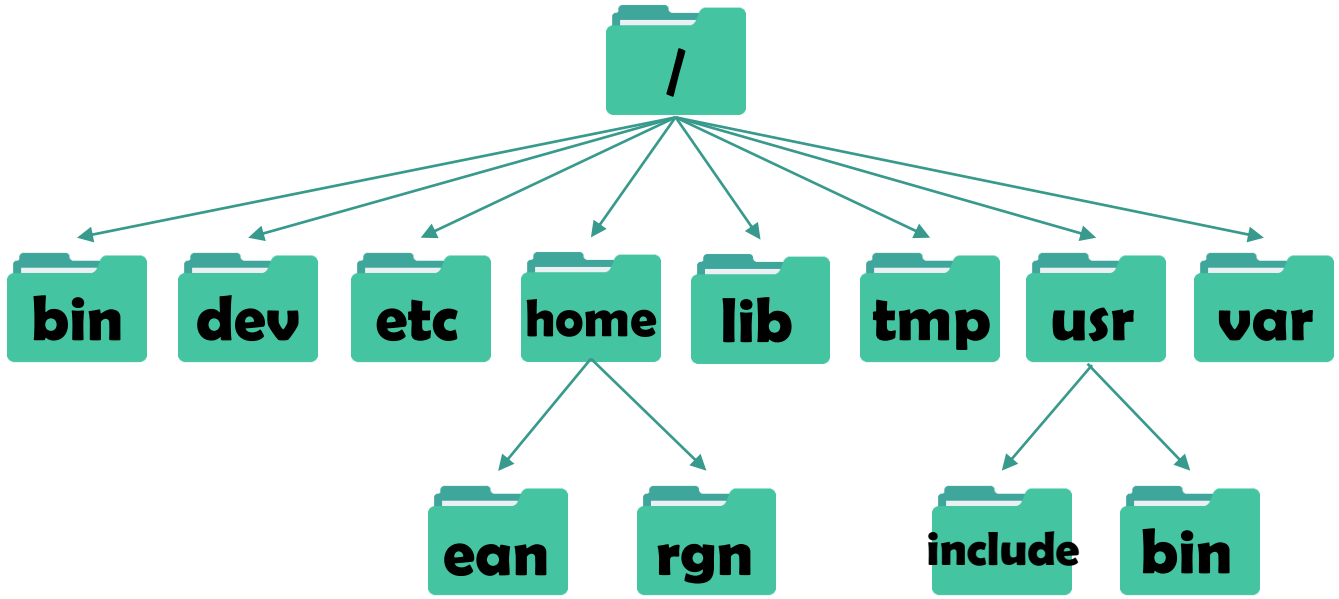
# UNIX : les fichiers

- Liens physiques : fichier avec un lien vers l'inode de la cible
- Si le fichier cible est supprimé, le lien fonctionne toujours
- Si l'inode n'a plus aucun fichier qui y fait référence, les données seront supprimées de la mémoire secondaire



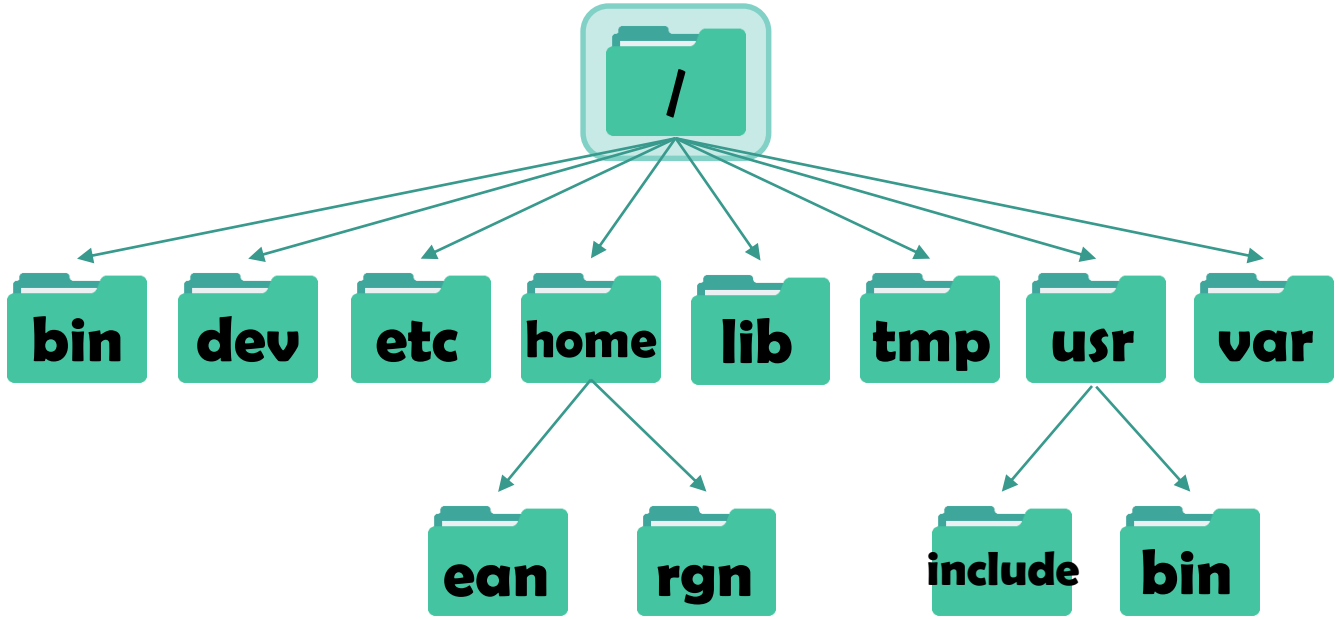
# UNIX : les dossiers

- Les fichiers peuvent être regroupés hiérarchiquement dans un répertoire (dossier)
- Un dossier est un fichier dont le contenu est la liste des références des fichiers qu'il contient
- Il a les mêmes propriétés qu'un fichier



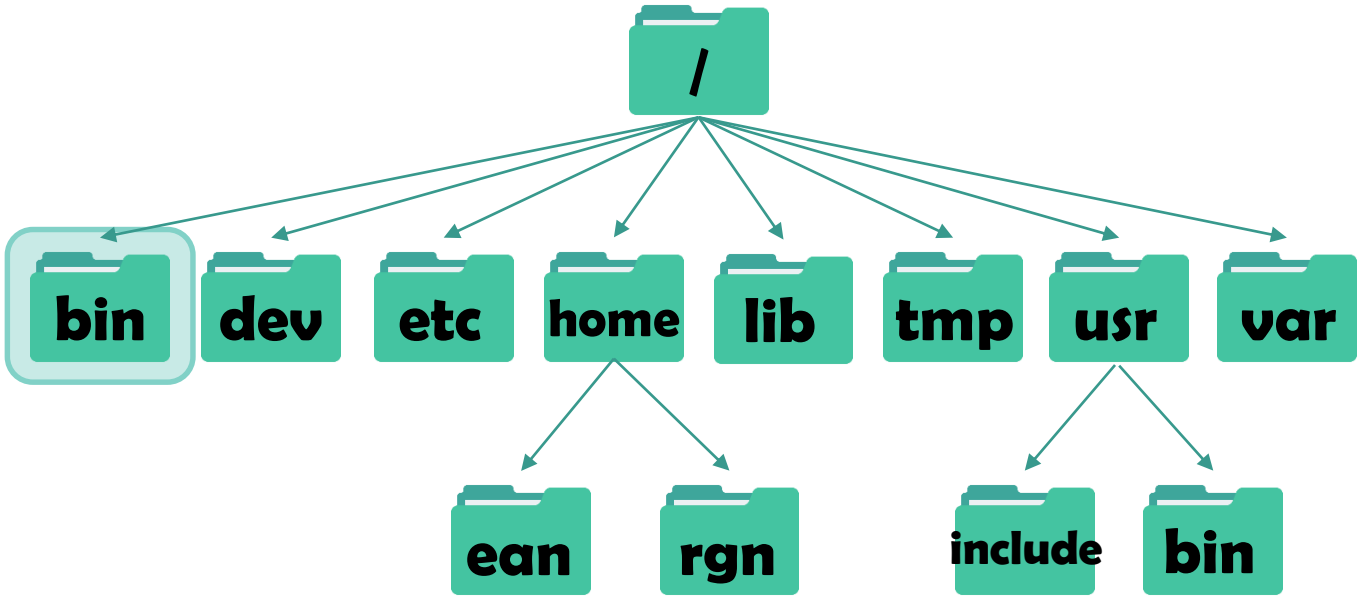
# UNIX : les dossiers

- / : répertoire racine (root)



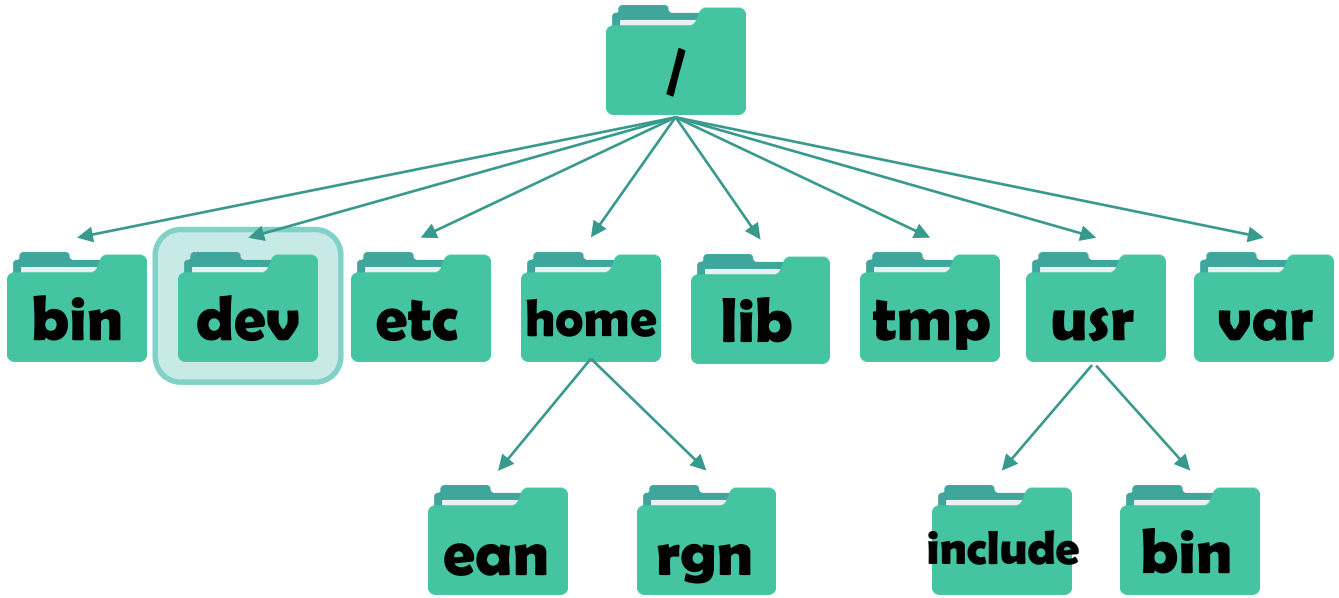
# UNIX : les dossiers

- /bin : commandes de base pour tous les utilisateurs



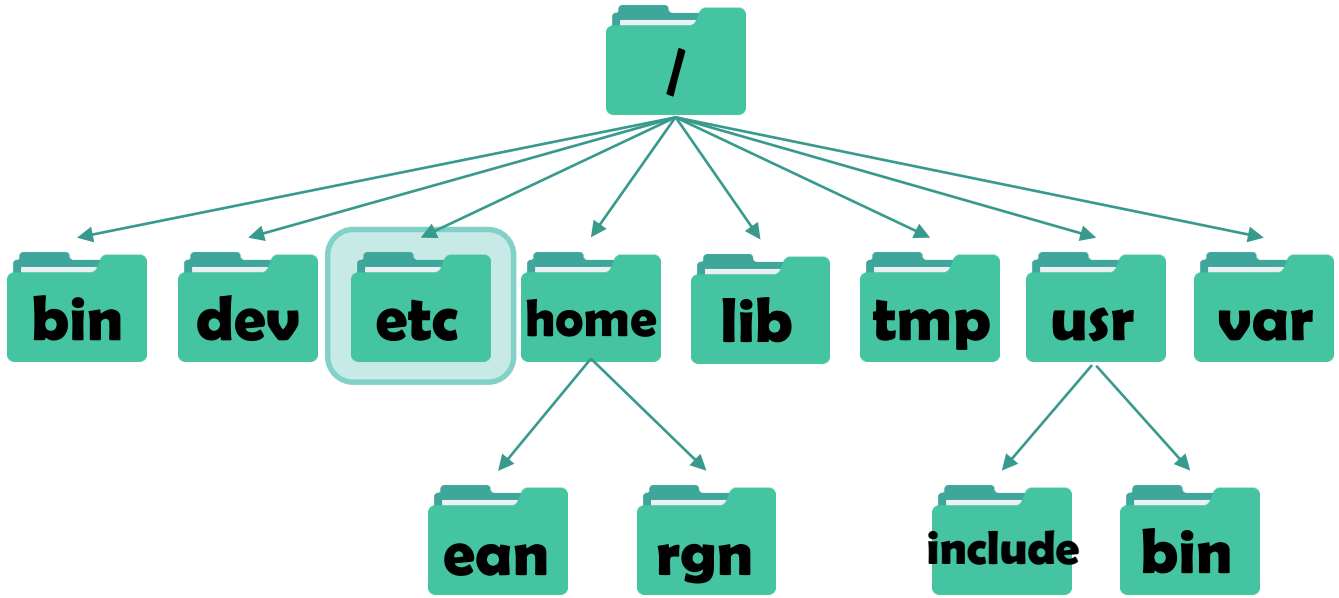
# UNIX : les dossiers

- /dev : fichiers pilotant les périphériques matériels



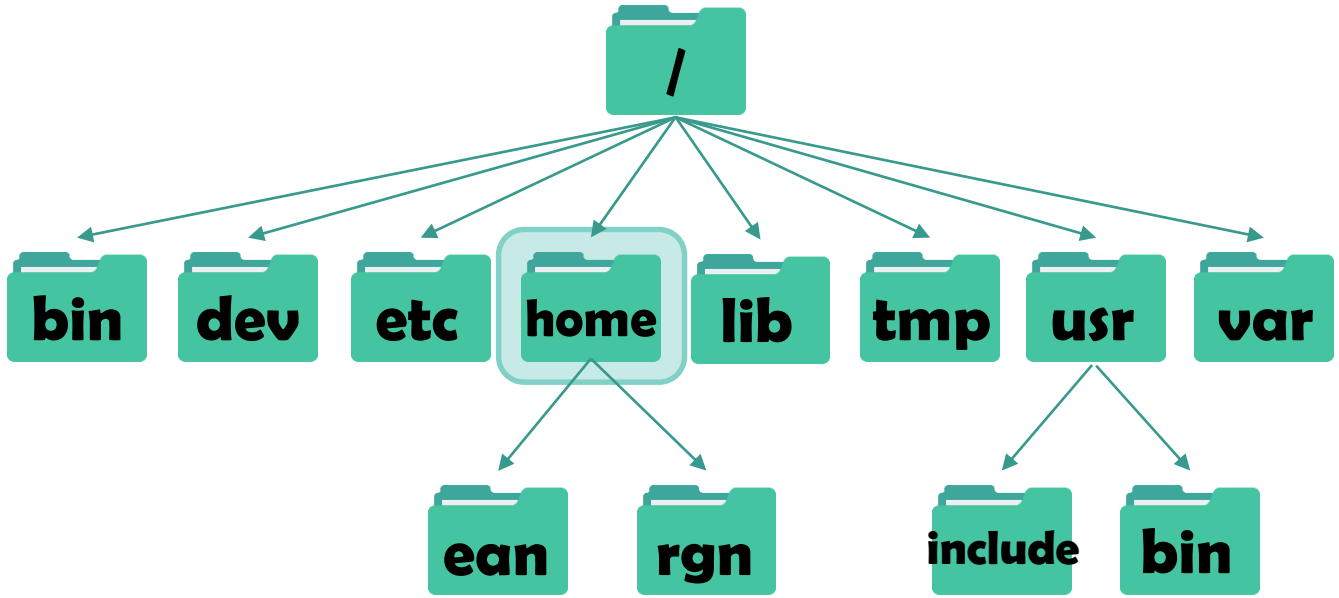
# UNIX : les dossiers

- **/etc** : fichiers de configuration (fichiers de démarrage)



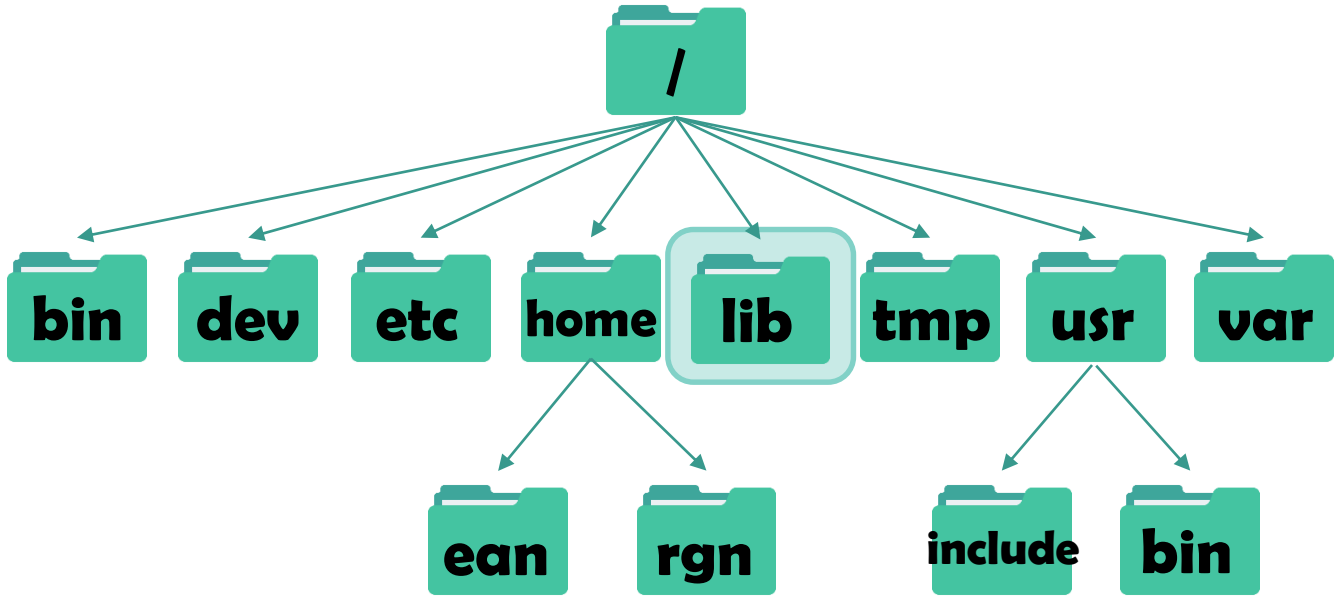
# UNIX : les dossiers

- /home : fichiers des utilisateurs



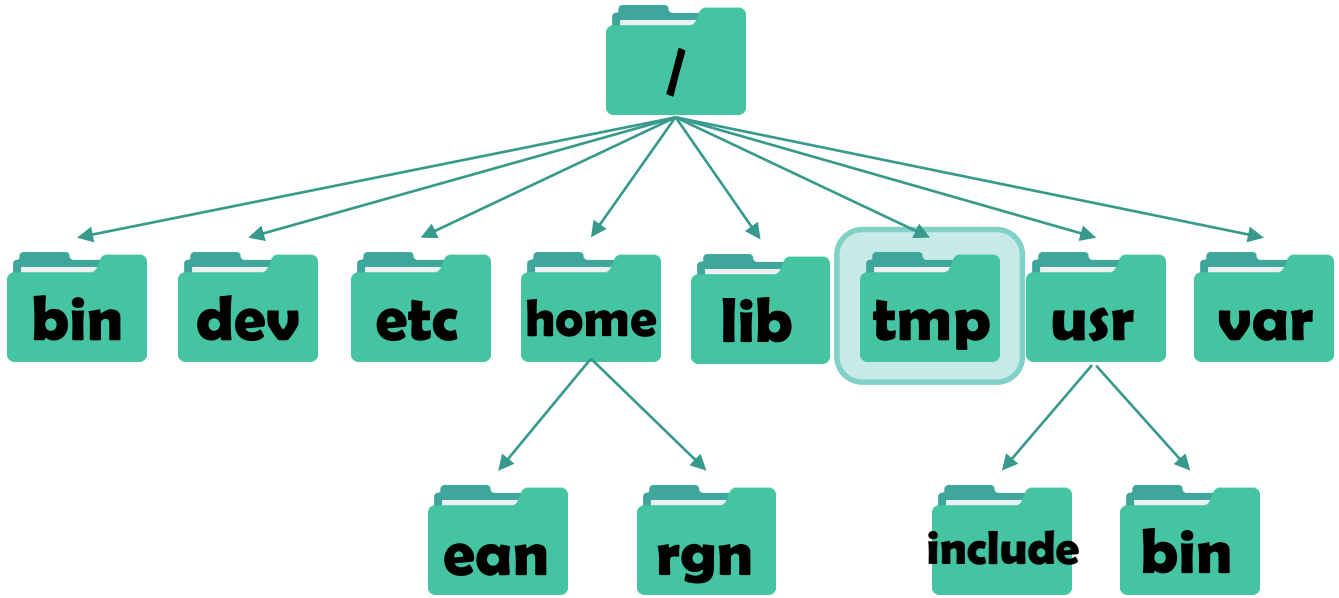
# UNIX : les dossiers

- **/lib** : bibliothèques compilées (nécessaires pour les binaires dans /bin)



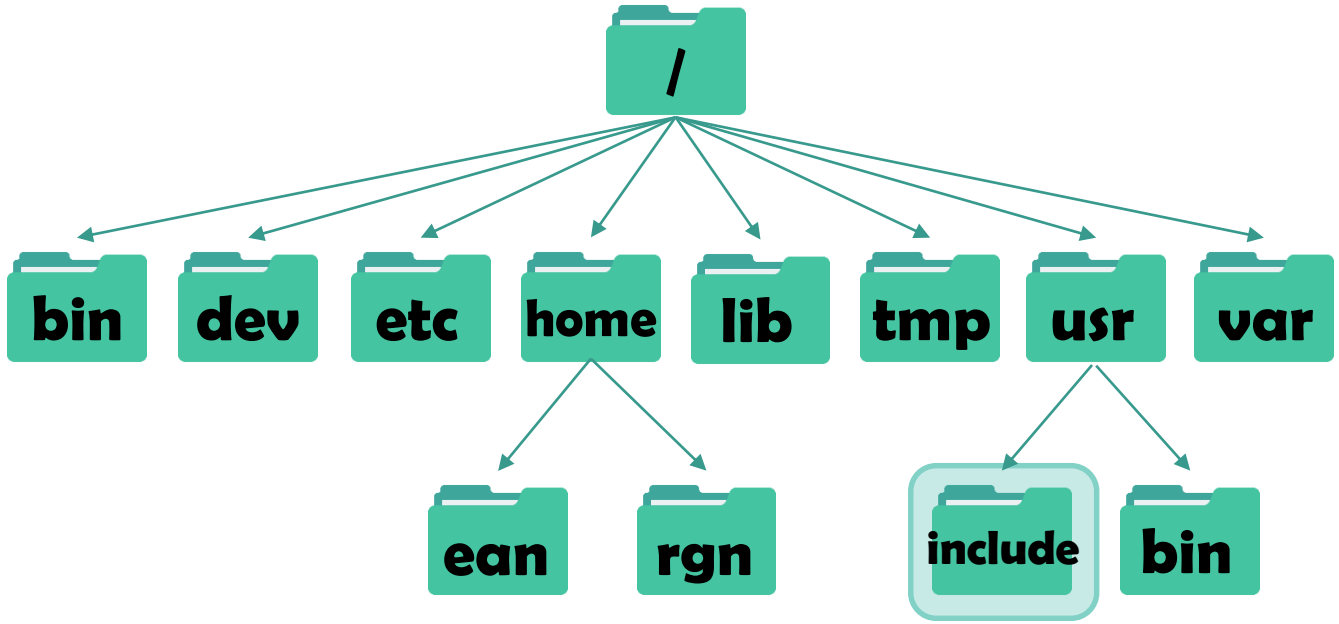
# UNIX : les dossiers

- /tmp : fichiers de travail temporaires



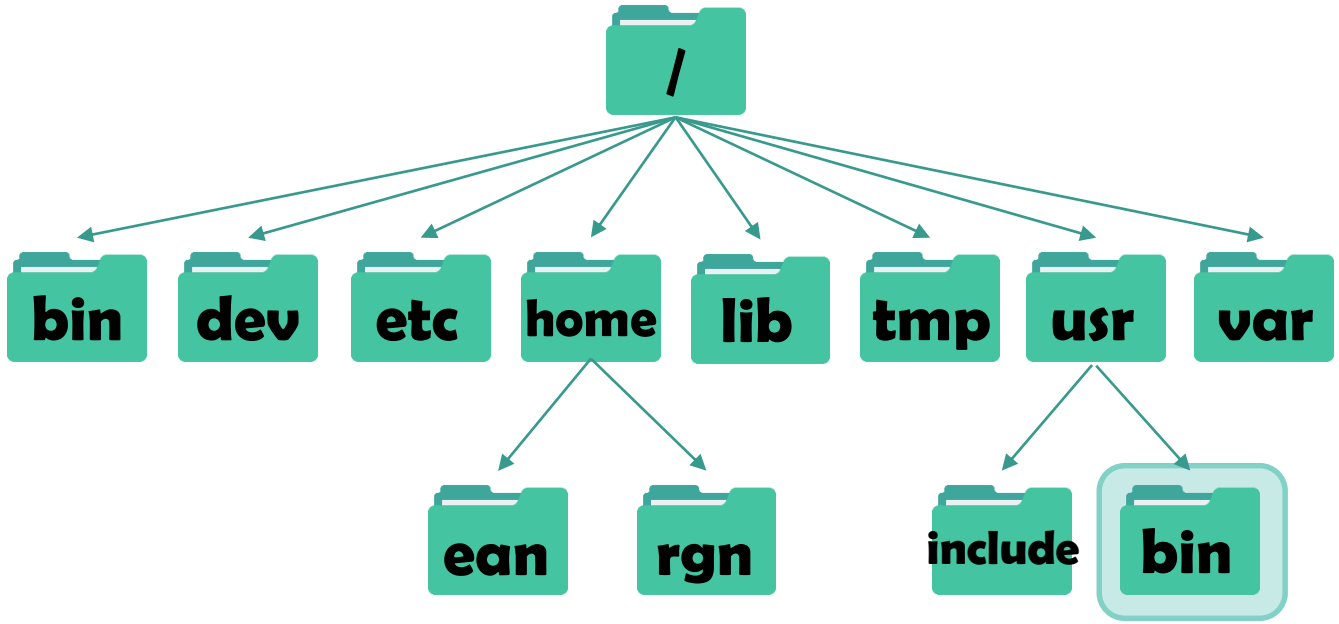
# UNIX : les dossiers

- **/usr/include** : fichiers d'entête des langages



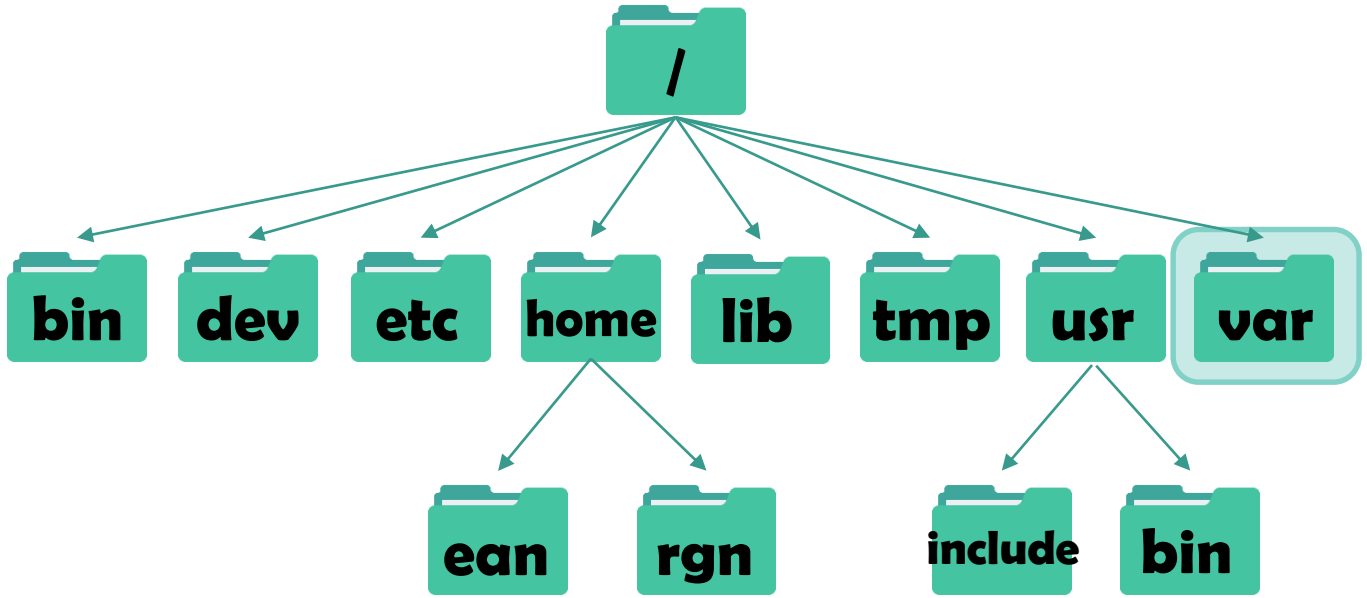
# UNIX : les dossiers

- **/usr/bin** : commandes complémentaires de l'utilisateur



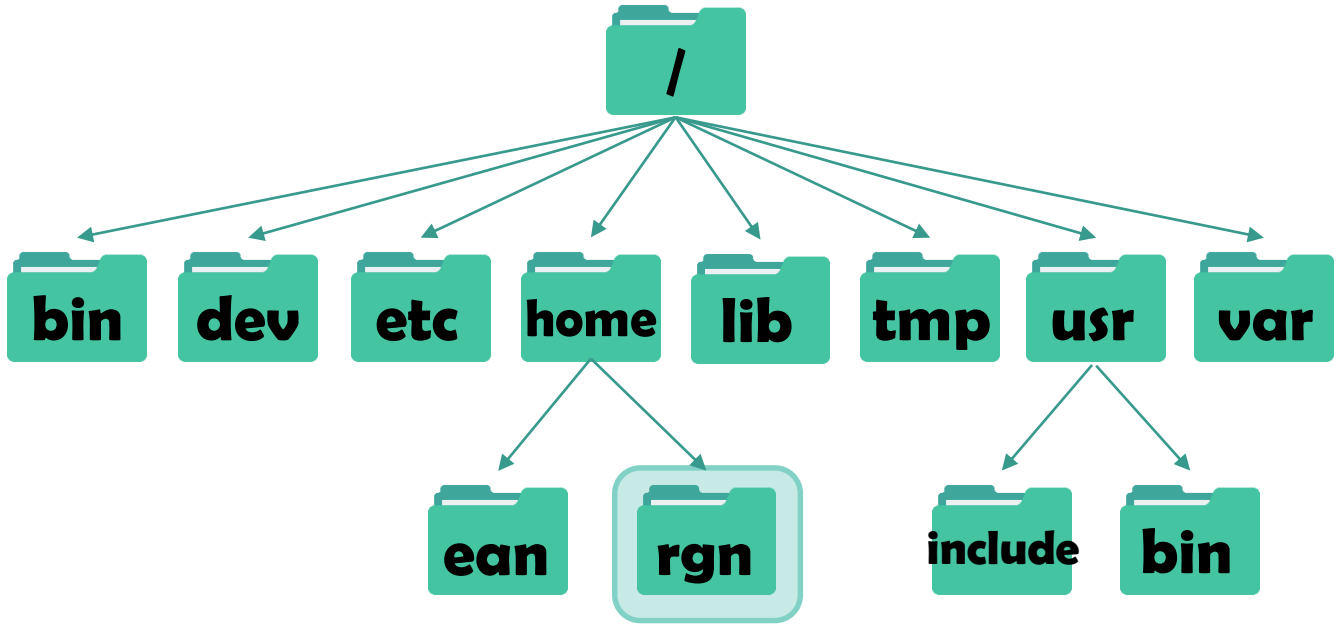
# UNIX : les dossiers

- /var : fichiers divers (boîte email, fichiers temporaires, fichiers journaux, …)



# UNIX : les dossiers

- /home/rgn (~) : dossier personnel de l'utilisateur « rgn »



# UNIX : les dossiers

- Un chemin commençant par “ / “ est un chemin **absolu**
- Tout autre chemin est un chemin **relatif** par rapport repertoire **courant**

- Le dossier racine du disque dur (root) se nomme “ / “
- Il existe des noms génériques pour certains dossiers :
  - dossier home utilisateur : “ ~ “
  - dossier courant : “ . “
  - dossier parent : “ .. “
  - dossier précédent : “ - “

# UNIX

- Système d'exploitation
  - multi-tâches
  - multi-utilisateurs

- 4 concepts élémentaires :
  - les fichiers
  - les droits d'accès
  - les processus
  - la communication inter-processus

# UNIX : les droits d'accès

- Tout fichier possède un **identificateur** (nom) :
  - Le caractère " / " n'est pas autorisé
  - Pas de structure particulière : des noms de fichiers sans suffixe ou sans préfixe sont possibles
  - Un nom de fichier **sans préfixe** (commençant par un point, suivi par des caractères) sera considéré comme un **fichier caché** qui ne sera pas visible par défaut
  - Les **caractères spéciaux** tels que les espaces, ?, &, *, …, sont **autorisés** mais en pratique ils compliquent la manipulation des noms des fichiers

# UNIX : les droits d'accès

- Tout fichier possède un **propriétaire** :
  - Très souvent le propriétaire d'un fichier est son **créateur**
  - Le propriétaire est le seul utilisateur classique à pouvoir **modifier les droits** d'accès du fichier
  - L'administrateur (**root**) peut également modifier les droits d'accès car c'est un utilisateur qui a tous les pouvoirs
  - Le propriétaire peut **transférer la propriété** du fichier (perdant ainsi ses privilèges associés) à un autre utilisateur qui deviendra à son tour le nouveau propriétaire

# UNIX : les droits d'accès

- Tout fichier possède des **droits d'accès** :
  - lecture/écriture/éxecution → read/write/exec → r/w/x
  - lecture : le fichier peut être lu
  - écriture : le fichier peut être modifié/supprimé
  - exécution : le fichier peut être executé par le système

- Dans le cas d'un **dossier** :
  - lecture : le contenu est visible
  - écriture : le contenu est modifiable (création de sous-dossiers, effacement de fichiers, …)
  - exécution : l'utilisateur peut rentrer dans le dossier

# UNIX : les droits d'accès

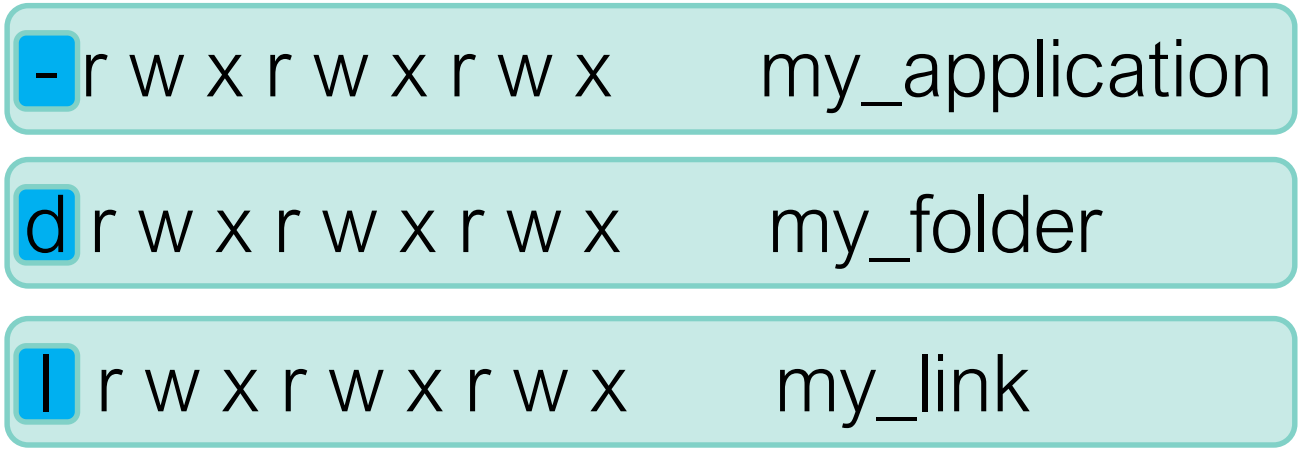
- Les différents droits sont déclinés en fonction des utilisateurs qui accèdent au fichier :
  - le propriétaire
  - les utilisateurs du même groupe que le propriétaire
  - les autres utilisateurs

```
- r w x r - x - - -        my_application
```

- Cette différenciation permet de configurer finement l'accès aux différents fichiers.
- Même si le propriétaire n'a aucun droit d'accès à un fichier, il peut toujours en modifier les droits

# UNIX : les droits d'accès

- En plus des informations concernant le propriétaire, le groupe, les droits d'accès, il est possible de savoir à quel type de fichier on a à faire :
  - "-" : un fichier classique
  - "d" : un répertoire
  - "l" : un lien ("raccourci")



# UNIX

- Système d'exploitation
  - multi-tâches
  - multi-utilisateurs

- 4 concepts élémentaires :
  - les fichiers
  - les droits d'accès
  - les processus
  - la communication inter-processus

# UNIX : les processus

- Unité élémentaire de gestion des **traitements**
  - un ensemble **d'instructions** à executer
  - un espace **mémoire** dédié pour les données de travail et les instructions
  - dispose d'autres **ressources** comme des descripteurs de fichiers, des ports reseau, …
- **Thread** ("processus léger")
  - Fil d'execution partageant des données et différentes ressources
  - On peut avoir plusieurs threads dans un processus
- Chaque processus possède un **identifiant** unique (**PID**) et stocke l'identifiant du processus "parent" qui lui a donné naissance (**PPID**)

# UNIX

- Système d'exploitation
  - multi-tâches
  - multi-utilisateurs

- 4 concepts élémentaires :
  - les fichiers
  - les droits d'accès
  - les processus
  - la communication inter-processus

# UNIX : la communication inter-processus

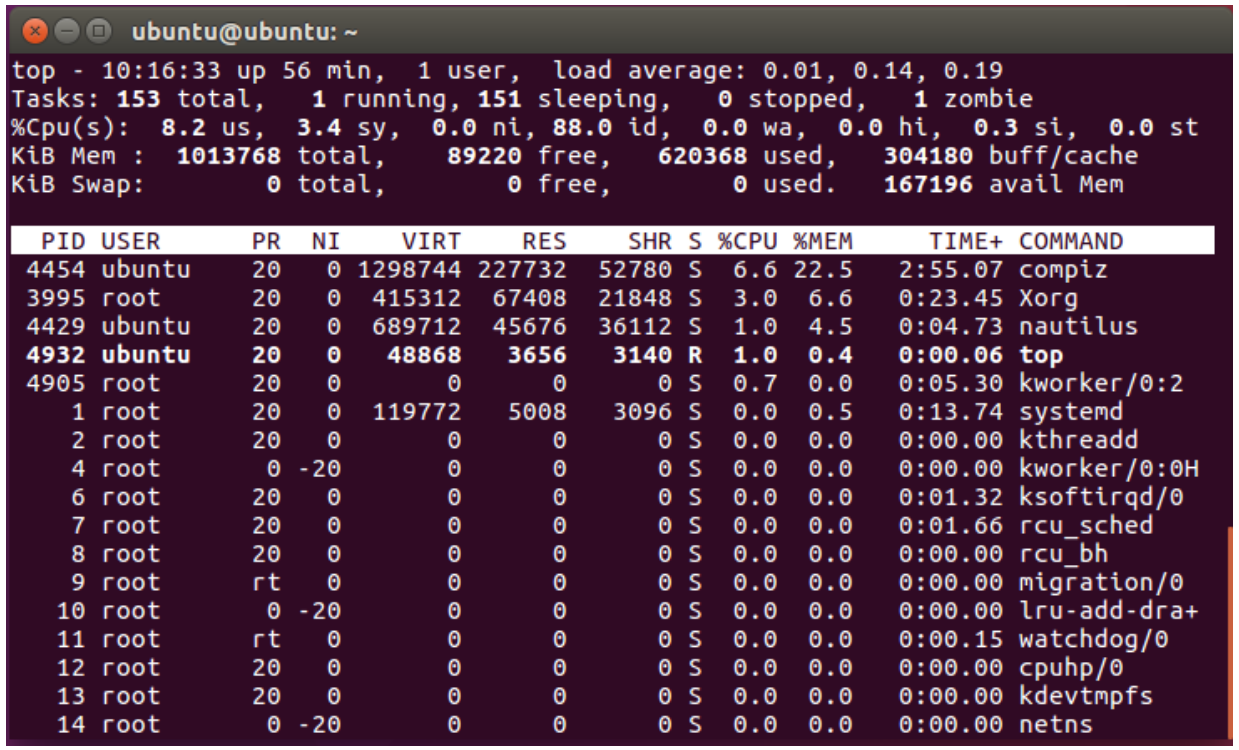
- Plusieurs processus peuvent **communiquer** entre eux via :
  - des **signaux**
  - une **mémoire partagée**
  - un **tube** (“pipe”)
  - une communication **réseau** (“socket”)

- Pour arbitrer l’accès à une ressource partagée, quelle qu’elle soit, il existe des services permettant d’obtenir et/ou d’attendre l’autorisation :
  - **mutex**
  - **sémaphore**

# III. Commandes UNIX

# Terminal de commandes :

- A partir d'un **terminal** de commandes, on peut effectuer diverses actions sur le système en appellant des programmes executables déjà presents




- Un **interpréteur** va s'occuper d'extraire les informations de la commande afin de l'executer

# Commandes :

- Le format de base des **commandes** est le suivant :
  - *<nom_commande> <options> <arguments>*

- Chaque commande peut (optionnellement) posséder des **options** et/ou des **arguments**

- Option avec une seule lettre : *-o -l -r -f*
- Option avec plusieurs lettres : *--color --version*

- Les arguments sont généralement des noms de fichiers, des expressions, …

# Commandes :

- Pour obtenir de l'**aide détaillée** sur une commande :
  - ***man*** *<nom_commande>*
  - *permet d'afficher le **manuel** de la commande afin d'avoir une explication concrète de son utilisation*
  - *détaille l'ensemble des options possibles et leurs rôles*
- Pour obtenir une **aide succincte** sur une commande :
  - *<nom_commande>* ***--help***
  - ***aide-mémoire** sur les principales options*
- Une liste des commandes usuelles est présentée à la fin de ce document (chapitre XI)

# Commandes :

- Chaque commande est un programme qui renvoie un code d'erreur en fin d'exécution
  - *0* : le traitement s'est terminé avec succès
  - *autre valeur* : une erreur est survenue

- Une **variable** shell (cf. chapitre VI) contient ce code d'erreur :
  - la commande **echo** permet de l'afficher
    - `ls myDir   # affiche le contenu du dossier`
    - `echo $?   # affiche le code retour de ls`

# Commandes :

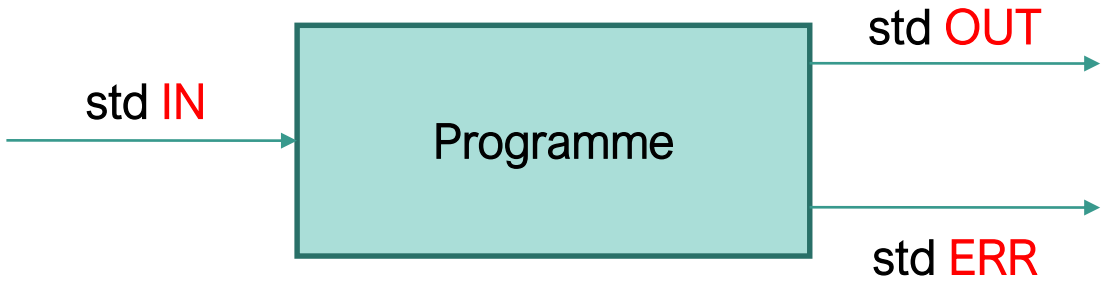
- Il est possible d'indiquer des valeurs de caractères de manière littérale dans les arguments des commandes :
  - " ? " : remplace 1 caractère quel qu'il soit
  - " * " : remplace plusieurs caractères quels qu'ils soient
  - " [ ] ": remplace 1 caractère parmi une liste définie

```
# affiche tous les fichiers PNG
ls "*.png"
# affiche les fichiers PNG avec 3 caractères
ls "???.png"
# affiche les fichiers PNG commençant par d ou f
ls "[df]*.png"
```

# IV. Redirection des entrées/sorties

# Redirection des entrées/sorties :

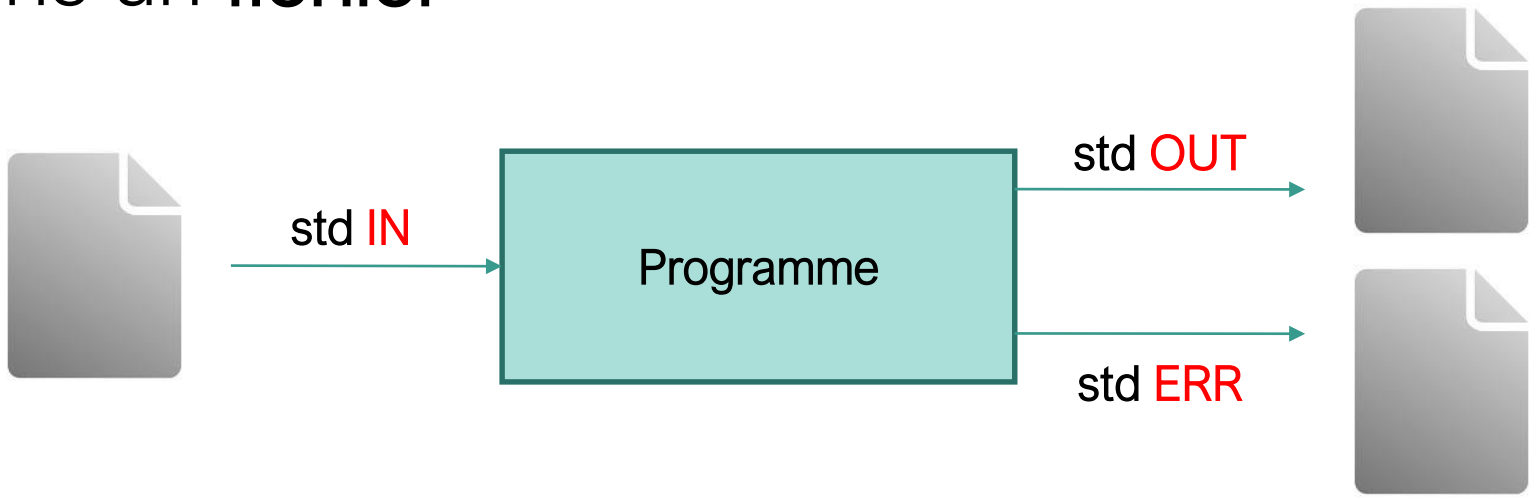
- Toutes les commandes sont des exécutables (programmes) qui peuvent donc recevoir des données depuis **l'entrée standard**, et/ou envoyer des données vers la **sortie standard** et/ou **l'erreur standard**



# Redirection des entrées/sorties :

- Il est possible d'effectuer **simplement** des **redirections** de ces entrées/sorties les unes vers les autres

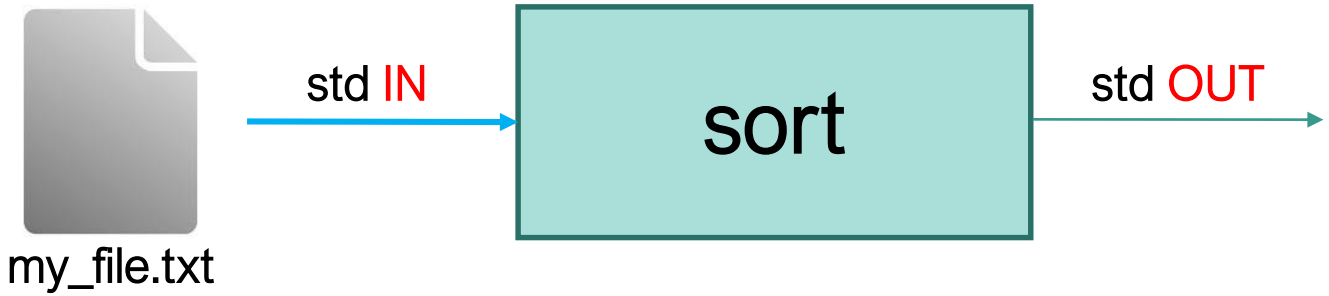
  - Dans le cas où l'on souhaite **lire** les données d'un **fichier** pour les traiter, et/ou **écrire** le résultat d'une commande dans un **fichier**



# Redirection des entrées/sorties :

- Rediriger le contenu d'un fichier vers un programme

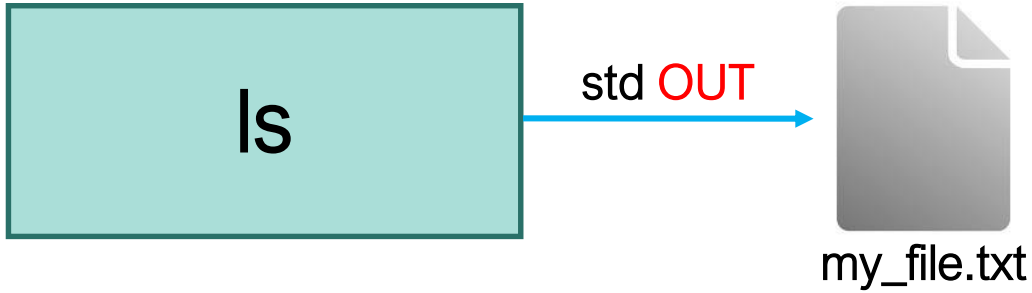
- ```
  # la commande sort trie les lignes reçues en entrée avant de les renvoyer sur sa sortie
  sort < "my_file.txt"
  ```


  ```

# Redirection des entrées/sorties :

- Rediriger la sortie d'un programme vers un fichier
- Mode **remplacement** :
  - création du fichier cible si inexistant
  - effacement du contenu initial du fichier cible si existant

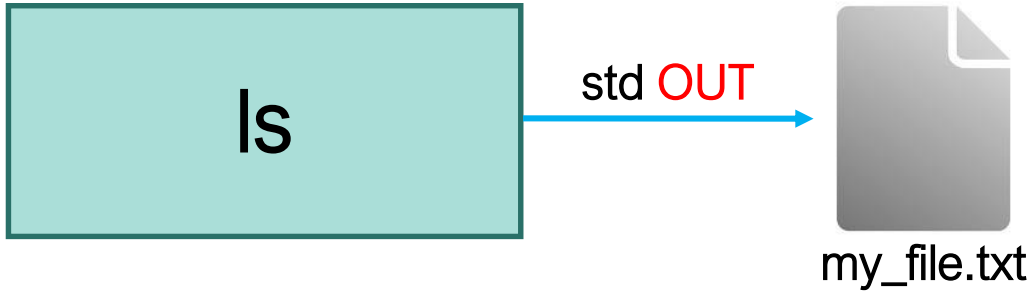
- ```
  # la commande ls affiche le contenu d'un dossier
  ls > "my_file.txt"
  ```


  ```

# Redirection des entrées/sorties :

- Rediriger la sortie d'un programme vers un fichier
- Mode **ajout** :
  - création du fichier cible si inexistant
  - ajout après le contenu initial du fichier cible si existant

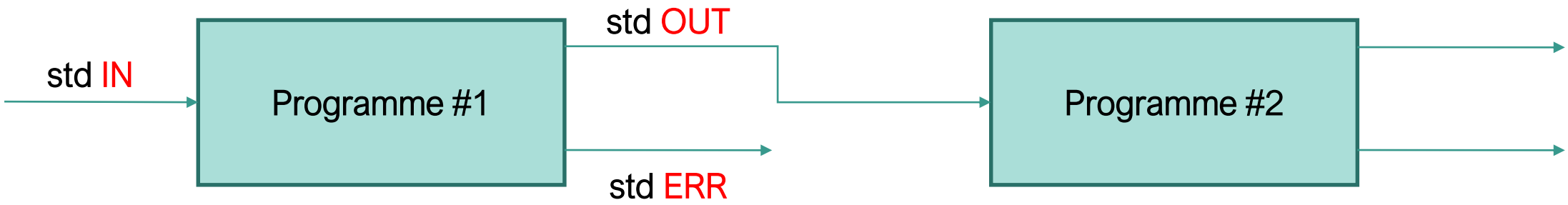
- ```
  # la commande ls affiche le contenu d'un dossier
  ls >> "my_file.txt"
  ```


  ```

# Redirection des entrées/sorties :

- Il est possible d'effectuer **simplement** des **redirections** de ces entrées/sorties les unes vers les autres

  - Dans le cas où l'on souhaite **transférer le résultat** d'une commande à la commande suivante : les *tubes* (eng:*pipes*)

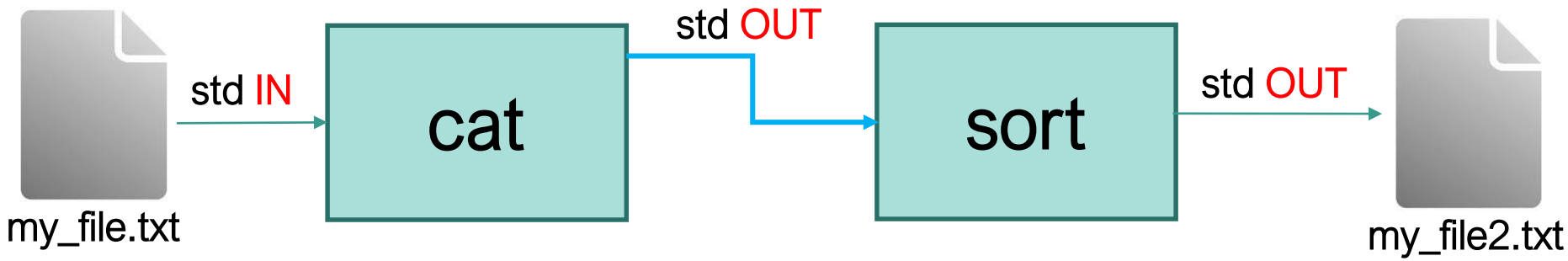


# Redirection des entrées/sorties :

- # la commande ***cat*** redirige l'entrée standard vers la sortie standard

- # la commande ***sort*** trie les lignes en entrées avant de les renvoyer vers la sortie

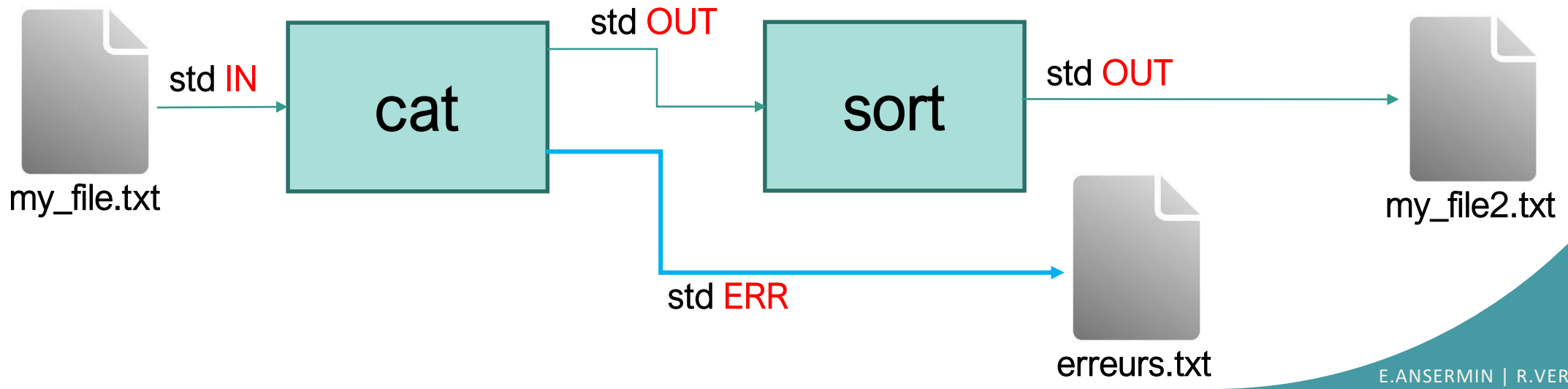
```
cat < my_file.txt | sort > my_file2.txt
```



# Redirection des entrées/sorties :

- L'erreur standard est utilisée **uniquement** quand le programme détecte une **anomalie** : de fait cette sortie **n'est pas envoyée** à la commande **suivante**
- On **peut forcer la redirection** de **l'erreur** standard vers **un fichier**

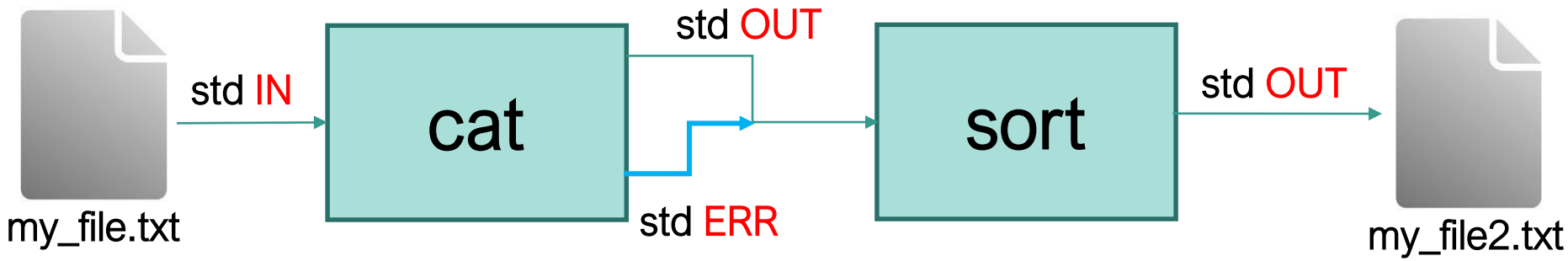
```
cat < my_file.txt 2>erreurs.txt | sort > my_file2.txt
```



# Redirection des entrées/sorties :

- L'erreur standard est utilisée **uniquement** quand le programme détecte une **anomalie** : de fait cette sortie **n'est pas envoyée** à la commande **suivante**
- On **peut forcer la redirection** de **l'erreur** standard vers la **sortie** standard (et ainsi **fusionner les 2 sorties** en 1)

```
cat < my_file.txt 2>&1 | sort > my_file2.txt
```



# V. Script Shell - Introduction

# Langages de scripts :

- En programmation, il existe 2 grandes familles de langages :
  - Les langages compilés (C/C++, Java, …)
  - Les langages interprétés (Python, PHP, JavaScript, …)

- C'est dans la 2ème catégorie que se situe le Shell

- Un interpréteur est donc utilisé dans la console du terminal pour prendre en compte les commandes

- Le script en lui même n'est qu'un fichier texte dont les lignes seront interprêtées pendant l'exécution

# Script Shell :

- **Automatise** un ensemble d'**instructions**
- **Ecriture rapide**
- **Typage faible (**chaînes de caractères par défaut)
- Utilise toutes les possibilités des **commandes UNIX**
- Peu d'interactions avec l'utilisateur en général
- Peut prendre des **paramètres** utilisateurs
- **Pas de compilation**
- Exécuté via un interpréteur (Shell)
- Différents types de Shell existent
  - Par conséquent, des langages différents et/ou des langages similaires avec des particularités propres

# Script Shell (famille) :

- sh : le Shell **d'origine**
- csh : C-Shell, proche de la **syntaxe du C** (ancien)
- bsh : Bourne Shell, propriété de Bell (mécanisme des **tubes**)
- tcsh : Tenex C-Shell, **complétion** automatique, **historique**, …
- ksh : fusion de csh et bsh, **historique des commandes**

- bash : Bourne Again Shell, **open-source**, reprend toutes les **avancées**, souvent celui par défaut dans les distributions Linux

- zsh : fusion des shells précédents
- et d'autres…

# Script Shell - Exécution:

- A partir de votre terminal, la commande suivante lance votre script :

```
bash my_script.sh
```

- Si votre fichier de script (un simple fichier texte) possède les droits d'exécution, il est possible de le lancer comme tout autre exécutable

```
./my_script.sh
```

- En réalité, votre script est passé en paramètre à l'interpréteur. Pour être certain de la bonne exécution de votre fichier de script, il convient d'ajouter le "hash-bang" **en première ligne**, en indiquant le bon interpréteur

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

# VI. Script Shell – Les variables

# Shell – Les variables :

- Typage faible (**chaînes de caractères**)
- Pas besoin de déclarer les variables
- La **déclaration** est **implicite** lors de l'**initialisation**

- L'opérateur d'affectation est le caractère '='

```
a='Hello'
b=5
c=32.759
```

⚠ Notez l'**absence** d'**espace** autour de l'opérateur '='

# Shell – Les variables :

- Utilisation d'une variable (lecture/affichage)

- La commande **echo** permet d'afficher une chaîne de caractères passée en paramètre
- Le caractère '**$**' doit toujours précéder le nom de la variable

```
echo $a
```

- Il est possible d'encadrer le nom de la variable par des accolades pour le borner avec précision

```
echo ${a}
```

- Si vous avez des variables qui partagent un même préfixe, n'oubliez pas les accolades pour éviter les bugs

# Shell – Les chaînes de caractères :

- En Shell il existe 3 types de chaînes :

  - Les **chaînes littérales**
  - Elles s'affichent telles qu'elles sont décrites sans aucune modification
  - Elles sont entourées des caractères : `'…'`

```
a=987
b='$a'          # b vaut : '$a'
```

# Shell – Les chaînes de caractères :

- En Shell il existe 3 types de chaînes :

  - Les **chaînes substituées**
  - Les noms des variables sont remplacés par leurs valeurs respectives
  - Elles sont entourées des caractères : "…"

```
a=987
b="$a"          # b vaut : '987'
```

# Shell – Les chaînes de caractères :

- En Shell il existe 3 types de chaînes :

  - Les **chaînes exécutées**
  - Il s'agit de commandes dont les sorties seront utilisées comme chaînes
  - Elles sont entourées des caractères : `…`

```
b=`ls`        # b vaut la sortie de ls
```

# Shell – Les chaînes de caractères :

- En Shell il existe 3 types de chaînes :

  - Les **chaînes exécutées**
  - Une syntaxe différente existe et permet d'être plus polyvalent dans l'écriture des scripts
  - Les commandes sont entourées des caractères : **$(…)**

✅ `varA=$(ls 'my_file_'$(date '+%Y%m%d'))`

❌ `varB=`ls -lai 'my_file_'`date '+%Y%m%d'``

# Shell – Les chaînes de caractères :

- Il est possible de filtrer une chaîne de caractères pour ne récupérer qu'une sous partie en recherchant un motif

```
my_str='A.coucou.Z'

# … à partir de la 1ère occurence
echo ${my_str#*cou}           # affiche 'cou.Z'
# … à partir de la dernière occurence
echo ${my_str##*cou}          # affiche '.Z'

# … jusqu'à la dernière occurence
echo ${my_str%cou*}           # affiche 'A.cou'
# … jusqu'à la 1ère occurence
echo ${my_str%%cou*}          # affiche 'A.'
```

# Shell – Les chaînes de caractères :

- Pour la concaténation, il n'y a pas d'opérateur particulier
- Il suffit de "coller" les chaînes pour qu'elles soient fusionnées

```
a='ABC'
b='DEF'
# par défaut les variables sont substituées
c="$a$b"  ou  c=$a$b
echo $c
```

# Shell – Les entiers :

- Un entier n'est rien d'autre qu'une chaîne de caractères représentant une valeur entière

```
a=51
b=-37
```

- Lors de l'affichage, rien ne différencie la valeur numérique sous-jacente de la chaîne :

```
echo "la variable a vaut : $a"
echo 'et b vaut : '${b}
```

# Shell – Les entiers :

- Les opérations **arithmétiques** sur les **entiers** nécessitent une syntaxe particulière avec les doubles parenthèses pour que le script comprenne qu'il a une opération de conversion de *chaîne* → *entier* à faire :

```
a=51
b=-37
add=$(( a + b ))
sub=$(( a - b ))
mul=$(( a * b ))
div=$(( a / b ))
mod=$(( a % b ))
echo $add $sub $mul $div $mod
```

# Shell – Les réels :

- Les opérations **arithmétiques** sur les **réels** ne sont pas prises en compte dans le Shell
- Il est nécessaire de faire appel à des commandes tierces pour effectuer les opérations
- Il existe une commande ***bc*** qui prend en entrée une expression arithmétique avec des réels, et renvoie la chaîne résultante du calcul

```
a=51.65
b=-37.98
r=`echo "$a+$b" | bc`
```

# VII. Script Shell – Les branchements conditionnels

# Shell – Les branchements conditionnels :

- Conceptuellement, un branchement prend une condition booléenne
  - Le type **booléen** n'**existe pas** en Shell
  - On teste le **retour** d'une **commande**

- Toute **expression** testée dans un branchement doit être une **commande** exécutée
  - Si la **valeur de retour 0** est renvoyée, on exécute le bloc d'instructions '**SI**'
  - Pour toute **autre valeur**, on exécute le bloc d'instructions '**SINON**'

# Shell – Les branchements conditionnels :

- Syntaxe :

```
if <commande>
then
    <instructions A>
else
    <instructions B>
fi
```

# Shell – Les branchements conditionnels :

- Possibilité de mettre les mot-clés ‘**if**’ et ‘**then**’ sur la même ligne s’ils sont séparés par un ‘**;**’

```
if <commande> ; then
    <instructions A>
else
    <instructions B>
fi
```

- Le mot-clé ‘else’ et le bloc d’instructions associé sont optionnels

```
if <commande> ; then
    <instructions A>
fi
```

# Shell – Les branchements conditionnels :

- Il est possible d'effectuer plusieurs branchements avec le test de plusieurs conditions successives
- On utilise pour cela le mot-clé '**elif**' :

```
if <commande1> ; then
    <instructions A>
elif <commande2> ; then
    <instructions B>
…
else
    <instructions C>
fi
```

# Shell – Opérateurs logiques :

- Il est possible de **combiner** le résultat de plusieurs **commandes** à l'aide d'**opérateurs logiques**
- Il existe 2 opérateurs logiques en Shell
  - l'opérateur '**ET**'
  - l'opérateur '**OU**' (inclusif)

- Ces opérateurs sont interprétés en mode "**évaluation paresseuse**" (*lazy evaluation*)
- La 2ème commande ne sera **pas exécutée** si le résultat de la 1ère **satisfait** déjà le **résultat final**

# Shell – Opérateurs logiques :

- L'opérateur '**ET**' :

```
if <commande1> && <commande2> ; then
    …
else
    …
fi
```

- En évaluation paresseuse, la <*commande2*> ne sera **pas exécutée** SI la <*commande1*> **échoue** (valeur de retour différente de 0)

# Shell – Opérateurs logiques :

- L'opérateur '**OU**' :

```
if <commande1> || <commande2> ; then
    …
else
    …
fi
```

- En évaluation paresseuse, la <*commande2*> ne sera **pas exécutée** SI la <*commande1*> **réussit** (valeur de retour égale à 0)

# Shell – Opérateurs de comparaison :

- Même si le type booléen n'existe pas, il est possible d'effectuer des comparaisons avec des opérateurs usuels
- La commande ***test*** prend en paramètre une expression de comparaison
- Utile pour comparer des chaînes ou des valeurs numériques

```
if test <expression> ; then
    <instructions A>
else
    <instructions B>
fi
```

# Shell – Opérateurs de comparaison :

- Il existe une autre syntaxe ne faisant plus apparaitre le nom de la commande ***test***, plus proche de celles utilisées dans les branchements conditionnels d'autres langages

- Cette syntaxe utilise des crochets pour contenir l'expression

```
if [ <expression> ] ; then
    …
fi
```

⚠ Notez les **espaces autour** de l'**expression** entre crochets

# Shell – Opérateurs de comparaison :

- Cette syntaxe remplace donc l'appel de la commande ***test*** dans la structure de branchement conditionnel :

```
if [ … ] ; then
    <instructions A>
else
    <instructions B>
fi
```

- Si la comparaison est correcte alors le bloc A sera effectué, sinon le bloc B

# Shell – Opérateurs de comparaison :

- On peut combiner ces comparaisons avec les opérateurs logiques vus précédemment :

```
if [ … ] && [ … ] ; then
    <instructions A>
elif [ … ] || [ … ] ; then
    <instructions B>
else
    <instructions C>
fi
```

# Shell – Comparaison de valeurs entières :

- Pour comparer des valeurs numériques entières il existe des opérateurs pour toutes les comparaisons usuelles :

```
a=5
b=12
[ $a -eq $b ]   # égalité          (equal)
[ $a -ne $b ]   # différence       (not equal)
[ $a -ge $b ]   # sup. ou égal     (great. or eq.)
[ $a -gt $b ]   # sup. strict.     (greater than)
[ $a -le $b ]   # inf. ou égal     (less. or eq.)
[ $a -lt $b ]   # inf. strict.     (lesser than)
```

⚠ Notez les **espaces autour** de l'**opérateur**

# Shell – Comparaison de valeurs entières :

- Il est possible d'utiliser des opérateurs de comparaison dont les symboles sont plus proches de leur représentation mathématique
- Pour cela, il faut utiliser la syntaxe des doubles parenthèses vue précédemment :

```
if (( $a == $b )) …    # égalité
if (( $a != $b )) …    # différence
if (( $a >= $b )) …    # sup. ou égal
if (( $a >  $b )) …    # sup. strict.
if (( $a <= $b )) …    # inf. ou égal
if (( $a <  $b )) …    # inf. strict.
```

# Shell – Comparaison de chaînes :

- Pour comparer des chaînes de caractères il existe plusieurs opérateurs :
  - égalité
  - différence
  - supérieur (suivant dans l'ordre lexicographique)
  - inférieur (précédent dans l'ordre lexicographique)
  - chaîne vide
  - chaîne non vide

# Shell – Comparaison de chaînes :

- Pour comparer des chaînes de caractères il existe plusieurs opérateurs :
  - égalité
  - différence

```
a='Coucou'
b='Hello’
[ "$a" = "$b" ]  # egalité
[ "$a" == "$b" ] # egalité (synonyme de =)
[ "$a" != "$b" ] # différence
```

⚠ L’**opérateur** d’**égalité** est un **simple** ‘=‘

⚠ Notez les **espaces autour** de l’**opérateur**

# Shell – Comparaison de chaînes :

- Pour comparer des chaînes de caractères il existe plusieurs opérateurs :
  - supérieur (suivant dans l'ordre lexicographique)
  - inférieur (précédent dans l'ordre lexicographique)

```
a='Coucou'
b='Hello'
[ "$a" > "$b" ]        # supérieur
[ "$a" < "$b" ]        # inférieur
```

⚠ Notez les **espaces autour** de l'**opérateur**

# Shell – Comparaison de chaînes :

- Pour comparer des chaînes de caractères il existe plusieurs opérateurs :
  - chaîne vide
  - chaîne non vide

```
a='Coucou'
b=''
[ -n "$a" ]          # test non-vide (VRAI ici)
[ -z "$b" ]          # test vide (VRAI ici)
```

⚠ Notez les **espaces autour** de l'**opérateur**

# Shell – Tests sur les propriétés des fichiers :

```
my_file='mon_programme.c'
```

- teste si la variable est une (**s**)tring déclarée ET non vide

```
[ -s $my_file ]
```

- teste si la cible (**e**)xiste

```
[ -e $my_file ]
```

- teste si la cible est un (**f**)ichier

```
[ -f $my_file ]
```

- teste si la cible est un (**d**)ossier

```
[ -d $my_file ]
```

- teste si la cible est un (**L**)ien

```
[ -L $my_file ]
```

# Shell – Tests sur les propriétés des fichiers :

```
my_file='mon_programme.c'
```

- teste si la cible possède un accès de type (**r**)ead

```
[ -r $my_file ]
```

- teste si la cible possède un accès de type (**w**)rite

```
[ -w $my_file ]
```

- teste si la cible possède un accès de type e(**x**)ecution

```
[ -x $my_file ]
```

- teste si la cible a été modifiée depuis la dernière lecture

```
[ -N $my_file ]
```

⚠ Par défaut, les droits d'accès sont testés par rapport à l'utilisateur qui a lancé le script (utilisateur courant)

# Shell – Opérateurs de comparaison :

- Il est possible de '**complémenter**' logiquement le retour d'une comparaison

- **Pas nécessairement utile** pour les **entiers** ou les **chaînes** de caractères car tous les opérateurs existent

- Peut être **utile** pour tester l'absence de **propriétés** des **fichiers**

```
[ ! s1 = s2 ]
[ ! 5 -eq 7 ]
[ ! -d 'mon_fichier.txt' ]
```

⚠ Notez l'espace entre le '**!**' et l'**expression** qui suit

# Shell – Branchement de type « case » :

- Lorsque l'on souhaite vérifier la valeur d'une variable parmi plusieurs valeurs fixes, la structure "Selon … Cas …" est souvent utilisée en algorithmie
- Les mots clés **case** … **in** … **esac** sont utilisés pour former ce type de structure

```
case $a in
   …
esac
```

# Shell – Branchement de type « case » :

- On peut donc comparer la chaîne originale avec n'importe quelle chaîne constante
- Chaque valeur de comparaison est suivie du caractère ')' qui indique le début du bloc d'instructions associé
- Chaque bloc d'instructions se termine par un double ";"

```
case $a in
  1)            … ;;
  2)            … ;;
  'Hello')      … ;;
esac
```

# Shell – Branchement de type « case » :

- Il est possible de comparer la chaîne d'origine en utilisant les caractères spéciaux des chaînes (*, ?, …) ou le contenu des autres variables
- Le cas "défaut" se note donc implicitement "***)**"

```
case $a in
   *toto*)     … ;;
   ???.txt)    … ;;
   "$b.png")   … ;;
   *)          … ;;
esac
```

# VIII. Script Shell – Les boucles

# Shell – Les boucles :

- Les boucles permettent d'exécuter un bloc d'instructions plusieurs fois en fonction de conditions particulières

- Plusieurs utilisations sont possibles :
  - boucle "POUR" qui va soit utiliser une variable dont elle va modifier la valeur (compteur), soit qui va parcourir une liste d'éléments (itérateur)
  - boucle "TANT QUE" qui va itérer tant qu'une condition reste valide

# Shell – La boucle POUR (compteur):

- La syntaxe pour ce type de boucle est similaire à celle du langage C

```
for (( i=5 ; i<=12 ; i++ ))   ; do
do
   echo $i
   …
done
```

- Comme pour le branchement conditionnel, il est possible d'écrire les mot-clés '**for**' et '**do**' sur la même ligne s'il sont séparés par un '**;**'

# Shell – La boucle POUR (compteur/itérateur):

- Il existe une syntaxe qui utilise un opérateur “**plage**”, et le compteur va **itérer** dans cette plage pour prendre chaque valeur l’une après l’autre

```
for k in {5..12} ; do
   echo $k
   …
done
```

- Les valeurs de **début** et de **fin** sont **inclues**

# Shell – La boucle POUR (itérateur):

- En Shell, une **chaîne** de caractères est **considérée** comme une **liste** de valeurs séparées par des caractères d'**espacement** (espace, tabulation, retour à la ligne, …)
- Il est donc possible d'itérer sur chacune des valeurs contenues dans cette chaîne (liste)

```
liste='abc def ghi jkl'
for var in $liste ; do
  echo $var
  …
done
```

# Shell – La boucle POUR (itérateur):

- La liste fournie dans la boucle peut provenir d'une commande exécutée
- La commande ls permet d'obtenir une liste de fichiers dans le répertoire courant : on peut itérer sur cette liste si l'on récupère le résultat de la commande (chaîne exécutée)

```
fichiers=`ls`
for fic in $fichiers; do
  echo $fic
  …
done
```

# Shell – Les caractères de séparation :

- Les **espaces et les tabulations** dans les données récupérées depuis les fichiers ou les commandes, peuvent **créer des problèmes** dans l'exécution des scripts

- Les **boucles** peuvent "**couper**" les **listes** aux **mauvais endroits** à cause de ces caractères d'espacements

- Il faudrait pouvoir indiquer à l'itérateur quels sont les caractères pouvant servir de séparateur : c'est le but d'une **variable** pré-existante **$IFS**

# Shell – Les caractères de séparation :

- Par défaut la variable **$IFS** contient tous les caractères d'espacement (espaces, tabulations, retours à la ligne)
- Il est possible de la modifier pendant l'exécution d'un script, mais il est toujours prudent de conserver sa valeur d'origine pour la restaurer plus tard dans le script en cours
- Tous vos scripts sont exécutés dans des contextes mémoire séparés, donc ce n'est pas trop impactant si la valeur d'origine de $IFS n'est pas restaurée

# Shell – Les caractères de séparation :

- Dans le cas où certains noms de fichiers contiendraient des espaces ou des tabulations, on met à jour $IFS pour ne tenir compte que des retours à la ligne pour découper la chaîne :

```
oldIFS=$IFS          # sauvegarde de IFS
IFS=$'\n'            # mise à jour de IFS
fichiers=`ls`        # liste des fichiers
for fic in $fichiers; do
  echo $fic
  …
done
IFS=$oldIFS          # restauration de IFS
```

# Shell – Les boucles :

- Les boucles permettent d'exécuter un bloc d'instructions plusieurs fois en fonction de conditions particulières

- Plusieurs utilisations sont possibles :
  - boucle "POUR" qui va soit utiliser une variable dont elle va modifier la valeur (compteur), soit qui va parcourir une liste d'éléments (itérateur)
  - boucle "TANT QUE" qui va itérer tant qu'une condition reste valide

# Shell – La boucle TANT QUE :

- Comme pour les branchements conditionnels, cette boucle prend en paramètre une **condition** qui est le résultat d'une **commande** ou d'une **comparaison**
- Tant que cette condition est **vraie**, les itérations s'exécutent les une après les autres

```
while [ … ] ; do
do
   …
done
```

# Shell – La boucle TANT QUE :

- Utilisation de la boucle avec une condition provenant d'une comparaison de compteur

```
a=1
while [ a -le 10 ] ; do
   echo $a
   a=$((a+1))
done
```

# Shell – La boucle TANT QUE :

- Utilisation de la boucle avec une commande retournant un code d'erreur différent de 0 pour stopper les itérations
- Ici la commande **read** permet de retourner les lignes d'un fichier, **une par une**, à partir d'un flux de données (ici l'opérateur '<' )

```
fichier='my_file.txt'
while read row ; do
   echo $row
done < $fichier
```

# IX. Script Shell – Les arguments

# Shell – Les arguments :

- Tout script peut prendre des arguments comme n'importe quel programme
- Il existe des variables prédéfinies qui contiennent des informations sur ces arguments :
  - Le nom du script
  - Le nombre d'arguments
  - La chaîne contenant tous les arguments
  - Des variables pour chaque argument

# Shell – Les arguments :

- Le nom du script en cours d'exécution :

```
nom_script=$0
```

- Le nombre d'arguments :

```
nb_args=$#
```

- Le valeurs de chaque argument :

```
arg1=$1
arg2=$2
arg3=$3
```

- La chaîne complète contenant tous les arguments :

```
all_args=$*
all_args=$@   # identiques en bash
```

# X. Script Shell – Les fonctions

# Shell – Les fonctions :

- En Shell, une fonction n'est rien d'autre qu'un sous-script qui sera lancé par le script courant
- La fonction est donc considérée comme n'importe quelle autre commande
- Une fonction devra donc tester ses propres arguments, effectuer ses calculs, affichages, …, puis retourner un code d'erreur

# Shell – Les fonctions :

- Syntaxes de déclaration d'une fonction :

```
# Syntaxe 1
function test {
    …
}

# Syntaxe 2
test() {
    …
}
```

# Shell – Les fonctions :

- Exemple de fonction :

```
test() {
    if [ $# -ge 1 ] ; then
        echo 'Il y a 1+ argument(s)'
        return 0
    else
        echo 'argument manquant'
        return 1
    fi
}
```

# Shell – Les fonctions :

- Exemple d'utilisation de la fonction/commande :

```
if test ; then
     echo 'OK'
else                                  > argument manquant
     echo 'ERROR'                     > ERROR
fi

if test 1 2 3 ; then
     echo 'OK'
else                                  > Il y a 1+ argument(s)
     echo 'ERROR'                     > OK
fi
```

# Shell – Les fonctions :

- La variable $? est une variable prédéfinie et contient le code retour de la dernière commande exécutée
- Exemple de récupération de la sortie et du code d'erreur d'une fonction/commande

```
var1=`test`
ret1=$?
var2=`test 1 2 3`
ret2=$?

echo $var1          > argument manquant
echo $ret1          > 1
echo $var2          > il y a 1+ argument(s)
echo $ret2          > 0
```

# XI. Commandes courantes

# Affichage :

- ls <*options*> <*chemin*> :
  - affichage des fichiers
  - options :
    - -l : format d'affichage long (taille, date, droits, …)
    - -a : affichage des fichiers cachés
    - -h : affichage des données en unités lisibles facilement
  - chemin :
    - fichier : affiche sa description
    - dossier : affiche son contenu

# Affichage :

- **pwd** :
  - affichage du chemin absolu du dossier courant
- **who <options> <arguments>** :
  - liste les utilisateurs connectés au système
- **date <options> <format>** :
  - affiche la date et l'heure
- **file <options> <fichier>** :
  - fourni des informations sur un fichier
- **echo <*valeur*>** :
  - affiche une chaîne de caractères

# Manipulation de chaînes :

- **cat** <options> <fichier> :
  - affiche le contenu complet d'un fichier
- **head** <options> <fichier> :
  - affiche les premières lignes d'un fichier
- **tail** <options> <fichier> :
  - affiche les dernières lignes d'un fichier
- **more** <options> <fichier> :
  - affiche le contenu d'un fichier, page par page
- **wc** <options> <fichier> :
  - compte le nombre de caractères d'un fichier
  - permet de compter le nombre de lignes

# Manipulation de fichiers :

- **cd <options> <dossier> :**
  - changement de répertoire courant
- **touch <options> <fichier> :**
  - création de fichier vide
  - si le fichier existe déjà, met à jour sa date de dernière modification
- **mkdir <options> <dossier> :**
  - création de dossier
- **ln <options> <cible> <nom_lien> :**
  - création de lien (symbolique ou physique)

# Manipulation de fichiers :

- **mv** <options> <source> <destination> :
  - déplacement/renommage de fichiers/dossiers

- **cp** <options> <source> <destination> :
  - copie de fichiers

- **rm** <options> <fichier> :
  - suppression de fichiers/dossiers

- **chmod** <options> <mode> <fichier> :
  - modification des droits d'accès d'un fichier/dossier

# Archivage :

- **tar <options> <fichiers> :**
  - permet d'archiver / désarchiver des fichiers
  - permet de compresser ou non l'archive
  - options :
    - -c : créer une archive à partir de fichiers
    - -x : extraire les fichiers d'une archive
    - -v : affichage détaillé des opérations sur la console
    - -f : spécification du fichier d'archive
    - -z : activation de la (dé)compression de l'archive

# Filtrage :

- **grep** *<options> <motif> <fichiers>* :
  - cherche des lignes contenant le motif dans des fichiers
  - filtrage “en ligne”
- **cut** *<options> <fichier>* :
  - extrait des “colonnes” pour chacun des lignes d’un fichier
  - permet de définir la chaîne qui sert de séparateur
  - permet de spécifier quelles colonnes conserver
  - filtrage “en colonne”
- **sort** *<options> <fichiers>* :
  - Trie les lignes du fichier afin de les afficher dans l’ordre
  - ordres alphabétique ou numérique
  - ordre croissant ou décroissant

# Processus :

- ps <options> :
  - affiche les processus en cours d'utilisation

- top <options> :
  - affiche la liste des processus les plus actifs en temps réel

- kill <options> <pid>:
  - envoie un signal à un processus (souvent pour y mettre fin)