

# R1.04 : Introduction aux systèmes d'exploitation et à leur fonctionnement

BUT Informatique – Semestre 1

Jean-François ANNE, Athénaïs VAGINAY

[jean-francois.anne@unicaen.fr](mailto:jean-francois.anne@unicaen.fr), [athenaïs.vaginay@unicaen.fr](mailto:athenaïs.vaginay@unicaen.fr)



# Avant-propos

Athénaïs Vaginay, mcf IUT / GREYC, [athenais.vaginay@unicaen.fr](mailto:athenais.vaginay@unicaen.fr)

Jean-Francois Anne, [jean-francois.anne@unicaen.fr](mailto:jean-francois.anne@unicaen.fr)

- Organisation R1.04 :
  - 8 **cours magistraux** (CM) – 1h amphi– promo
  - 8 séances de **travaux dirigés** (TD) – 2h sur machine – en groupe
  - 4 séances de **travaux pratiques** (TP) – 2h sur machine – en demi-groupe
- Documents sur <https://nanls.github.io/classes/index.html>, mais **prenez des notes**.
- Une note individuelle (semaine du 11 nov) + une note en binôme (QCM à chaque TP)
- Questions posables en cours et/ou par mail (objet + **du contexte**)

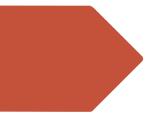


# Sommaire (prévision)

	CM (8)	TD (8)	TP (4)
<b>Caractéristiques et structure des systèmes d'exploitations</b> (couches noyaux, shell, apps)	1		
<b>Historique et types des systèmes d'exploitations</b>	2		
<b>Langage de commandes : commandes de base</b>	2 (nope)	1, 2	1, 2, 3, 4
- Gestion des utilisateurs (caractéristiques, création, suppression, etc.)	3	1	1
- Gestion des fichiers (arborescence, types, droits, etc.)	3, 4	1, 2, 3, 4	1
- Gestion des processus (création, destruction, suivi, communications etc.)	5	5, 6	1
- Installation et configuration d'un système	6		1
<b>Langage de commandes : introduction à la programmation des scripts</b>	7, 8	7, 8	2, 3, 4

Focus sur Linux mais TP2, 3, 4 sur Windows

Installation d'un système d'exploitation (Linux et Windows) : R1.03, avec JFA



# Bibliographie et Webographie

- Richard W. Stevens. Advanced Programming in the Unix Environment. Addison-Wesley, 2013.
- Michael Kerrisk. The Linux Programming Interface : A Linux and UNIX System Programming Handbook., No Starch Press, 2010
- Graham Glass. Unix for Programmer and Users. Prentice Hall, 1993.
- Joëlle Delacroix. Linux, Programmation système et réseau. Dunod, 2003
- **Les slides de JFA 2023-2024**
  - <http://e-classroom.over-blog.com/les-systemes-d-exploitation>
  - <https://distrowatch.com/>
  - <https://www.linuxfromscratch.org>
  - <https://www.lpi.org/our-certifications/>
  - [http://n.grassa.free.fr/sysrezo/systeme/Cours\\_Systeme.pdf](http://n.grassa.free.fr/sysrezo/systeme/Cours_Systeme.pdf)



```
00010100100001001010  
10001010010101010110  
010011000010001010001  
010100001010010011  
10011010010000001010  
010100100100100100  
100100101010101100  
10101010000100010011  
00101000100101001001
```



## Présentation des systèmes d'exploitation

Tout au long de cette ressource, nous apprendrons **comment fonctionnent les systèmes d'exploitation**. Mais, avant d'approfondir nos connaissances de **Linux** et **Windows**, deux des plus fascinants systèmes d'exploitation, nous allons définir **ce qu'est un système d'exploitation** et en voir **les composantes**.



# Deux catégories de programmes

- Les **programmes d'application** : résolvent les problèmes des utilisateurs.
- Les **programmes systèmes** : pour le fonctionnement des ordinateurs.

Le programme « **système d'exploitation** » est le programme fondamental des programmes systèmes. Il contrôle les ressources de l'ordinateur et fournit la base sur laquelle seront construits les autres programmes.



# **Système d'Exploitation (SE)**

## **= Operating System (OS)**

Vous en connaissez déjà plusieurs... des idées ? :)))

- L\*\*\*\*
- W\*\*\*\*\*

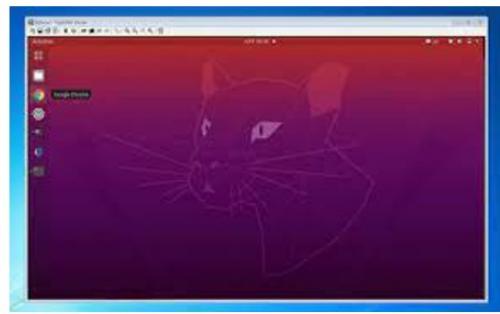
# Système d'Exploitation (SE) = Operating System (OS)



Apple Mac OS X Leopard



Microsoft Windows 10



Linux – Ubuntu



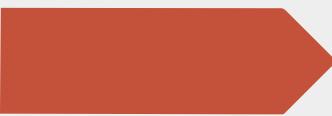
Chrome Os

Des guéguerres entre entreprises :

- « Passées », entre Unix (cours 2), entre Microsoft et Linux
- Présentes, [entre Microsoft et Google](#)

Des débats sans fin pour savoir « lequel est le meilleur ».

Android, iOS ... et plein d'autres !



# **Fonctions des systèmes d'exploitation**



# Exécuter un programme sur un ordinateur « nu »

Pour **exécuter un programme**, il faut :

- Aller le chercher sur le disque dur :
  - Trouver sa position
  - Lire les mots qui le décrivent
- Le mettre en mémoire
  - Lui allouer un espace
- L'exécuter...
  - Gestion du clavier par ce programme ?
  - Gestion de l'écran ?

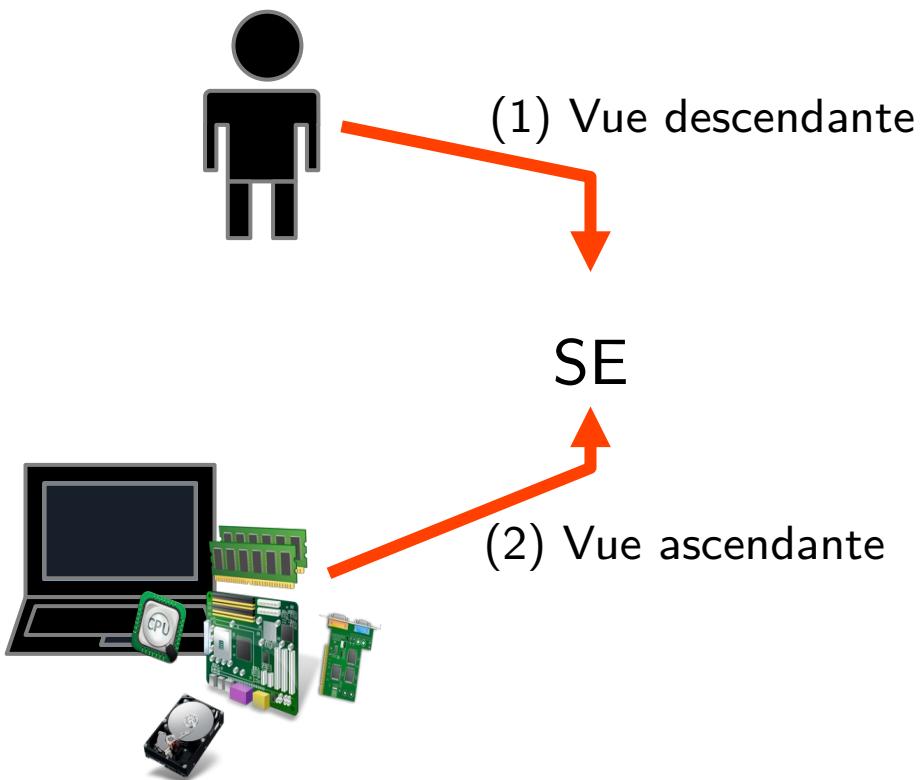
Mais un ordinateur nu, c'est :

- Programmation en **langage binaire** seulement
- Accès aux périphériques (clavier, écran, ...) très **difficiles**
- Exécution d'**un seul programme** à la fois

⇒ **Un ordinateur nu est « inutilisable »**

# Les deux vues du SE

Intermédiaire entre l'utilisateur.ice et le système informatique

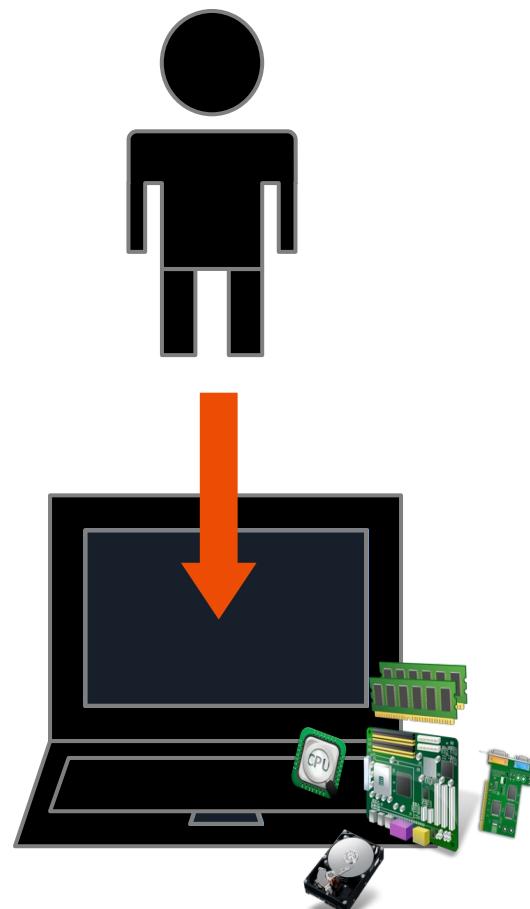


(1) Présenter **une machine virtuelle** : Son rôle est de masquer les éléments fastidieux liés au matériel et permettre à l'utilisateur.ice une exploitation simple et efficace de la machine ;

(2) Être **un gestionnaire de ressources** : Gérer l'**ordonnancement** et le contrôle de l'allocation des processeurs, des mémoires et des périphériques d'E/S entre les différents programmes qui y font appel.

E/S = Entrées/Sorties

# Vue descendante : Le SE est une machine virtuelle (= machine étendue)



Un ordinateur contient du **matériel complexe** : un ou plusieurs processeurs, une mémoire principale, des horloges, des terminaux, des disques, des interfaces de connexion à des réseaux, des périphériques d'entrées/sorties....

Via un mécanisme d'**abstraction** (= masquer les éléments complexes fastidieux à gérer), le système d'exploitation transforme cet assemblage de chips et de circuits en un appareil plus utilisable.

⇒ **simplification de l'interface humain-machine** (notée « IHM »)

## Vue descendante :

Le SE est une machine virtuelle (= machine étendue)

Fonction	Ressource matérielle	Concept abstrait
Gestion des données persistantes (accès, modification, stockage, partage)	Disques avec tête de lecture, disque SSD, clés USB, ...	Fichier, répertoire (via le <b>système de fichier</b> ), instructions « lire_fichier »
Gestion des communications avec l'utilisateur (entrée/sortie)	Clavier, écran, souris	GUI (fenêtres) / terminal (ligne de commandes)
Gestion des activités (création, suivi, destruction, erreur)	Processeur, programme sur disque, mémoire centrale	Processus (cours dédié plus tard)
Mémoire de travail	RAM	Mémoire virtuelle

GUI = Graphical User Interface

# Vue descendante : Le SE est une machine virtuelle (= machine étendue)

Exemple vis-à-vis de l'utilisation de la mémoire : **la virtualisation de l'allocation mémoire**

Allocation des adresses mémoires disponibles aux **processus** (= programme en cours d'exécution).

Un processus demande plus de mémoire → il lui est alloué les **prochaines** adresses disponibles (en RAM, ou sur disque = swap). On se retrouve donc **potentiellement** avec des **blocs pas consécutifs**, ce qui est **fastidieux** à gérer pour le programmeuseuse.

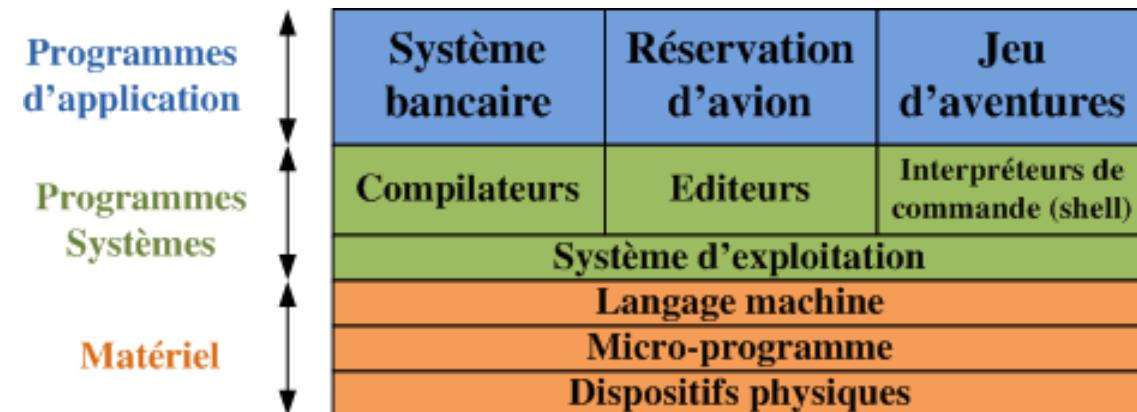
→ **pour cacher la complexité, le SE travaille de concert avec le MMU** (Memory Management Unit, un circuit sur la carte mère) : il font une **abstraction** et laisse le processus croire que la mémoire qui lui est allouée est consécutive.

PHYSICAL MEMORY ADDRESSES	VIRTUAL MEMORY ADDRESSES	ALLOCATED TO:
0–999	0–999	PROGRAM A
1000–1999	0–999	PROGRAM B
2000–2999	1000–1999	PROGRAM A
3000–3999	(NOT ALLOCATED)	(NOT ALLOCATED)
...	...	...

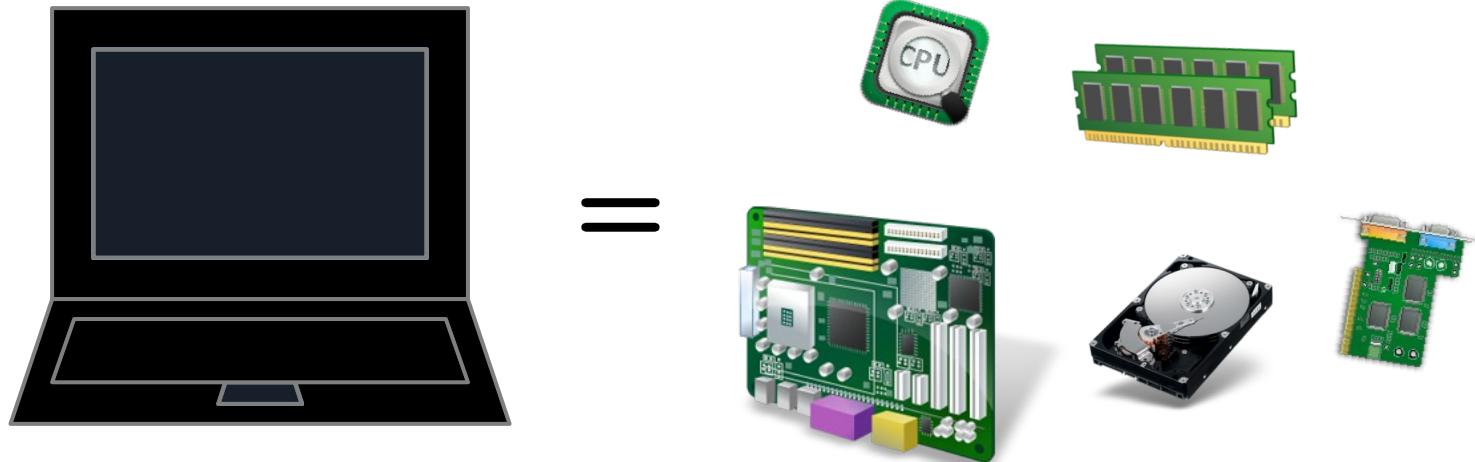
# Vue descendante : Le SE est une machine virtuelle (= machine étendue)

Un ordinateur...

- contient des **dispositifs physiques** : ils se composent de circuits intégrés, de fils électriques, de périphériques physiques ...
- contient un **microprogramme** = un logiciel de contrôle des périphériques (interprète).
- utilise le **langage machine** = un ensemble d'**instructions élémentaires (bas niveau)** (entre 50 et 300) pour effectuer le déplacement des données, des calculs, ou la comparaison de valeurs.
- utilise un **système d'exploitation**, qui propose un ensemble d'**instructions plus haut niveau**, comme LIRE\_FICHIER que peuvent utiliser les **programmes d'application**).



# Vue ascendante : le SE est un gestionnaire de ressources



- Mémoire
- Processeur
- Périphériques
- Fichiers
- Stockage
- ...

Un ordinateur se compose de **plusieurs ressources** qu'il faut **partager** entre les différents utilisateurs (multi-utilisateur) et programmes (multi-tâche).



# **Vue ascendante : le SE est un gestionnaire des ressources**

- Partage **entre les programmes (processus)** : le rôle de policier du SE permet d'éviter les conflits d'utilisation de la mémoire, des périphériques d'entrées/sorties, des interfaces réseau... etc.
- Partage **entre les usagers** : le partage de la mémoire et surtout sa protection demeure une priorité absolue.

**⇒ En tout temps, un bon système d'exploitation connaît l'utilisateur d'une ressource, ses droits d'accès, son niveau de priorité.**

**Exemples :**

- 3 programmes essaient d'imprimer simultanément leurs résultats sur une même imprimante ⇒ recours à un fichier tampon sur disque.
- accès simultané à une donnée ; lecture et écriture concurrentes (par deux processus) sur un même compteur.

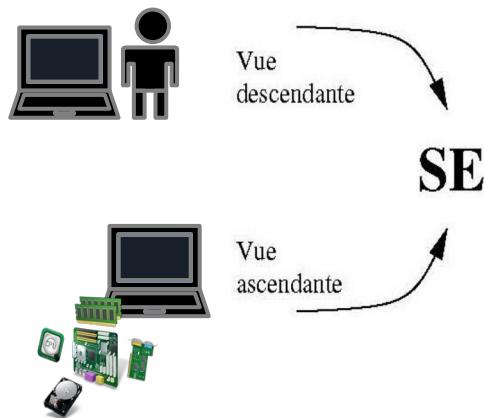


# Rôles du système d'exploitation

- **Transformer une machine matérielle en une machine utilisable**, c-à-d fournir des outils adaptés aux besoins indépendamment des caractéristiques physiques.
  - **Gérer les ressources**.
- Mais il faut également la garantie d'un bon niveau en matière de :
- **Sécurité** : intégrité, contrôle des accès confidentialité...
  - **Fiabilité** : degré de satisfaction des utilisateurs même dans des conditions hostiles et imprévues,
  - **Efficacité** : performances et optimisation du système pour éviter tout surcoût en termes de temps et de places consommées par le système au détriment de l'application.

# Rôles du système d'exploitation

- **Transformer une machine matérielle en une machine utilisable**, c-à-d fournir des outils adaptés aux besoins indépendamment des caractéristiques physiques. [vue descendante]
- **Gérer les ressources**. [vue ascendante]



Mais il faut également la garantie d'un bon niveau en matière de :

- **Sécurité** : intégrité, contrôle des accès confidentialité...
- **Fiabilité** : degré de satisfaction des utilisateurs même dans des conditions hostiles et imprévues,
- **Efficacité** : performances et optimisation du système pour éviter tout surcoût en termes de temps et de places consommées par le système au détriment de l'application.



# Divers rôles du système d'exploitation

machine virtuelle, gestionnaire de ressource (+ sécurité, fiabilité, efficacité)

- **Gestion du processeur** : le SE est chargé de gérer l'allocation du processeur entre les différents programmes grâce à un algorithme d'ordonnancement. Le type d'ordonnanceur est totalement dépendant du système d'exploitation, en fonction de l'objectif visé.
- **Gestion de la mémoire vive** : le SE est chargé de gérer l'espace mémoire alloué à chaque application et, le cas échéant, à chaque usager. En cas d'insuffisance de mémoire physique, le système d'exploitation peut créer une zone mémoire sur le disque dur. Cela permet de faire fonctionner des applications nécessitant plus de mémoire qu'il n'y a de mémoire vive disponible sur le système. En contrepartie cette mémoire est beaucoup plus lente.
- **Gestion des entrées/sorties** : le SE permet d'unifier et de contrôler l'accès des programmes aux ressources matérielles par l'intermédiaire des pilotes (appelés également gestionnaires de périphériques ou gestionnaires d'entrée/sortie).



# Divers rôles du système d'exploitation

machine virtuelle, gestionnaire de ressource (+ sécurité, fiabilité, efficacité)

- **Gestion de l'exécution des applications** : le système d'exploitation est chargé de la bonne exécution des applications en leur affectant les ressources nécessaires à leur bon fonctionnement. Il permet à ce titre de « tuer » une application ne répondant plus correctement.
- **Gestion des droits** : le système d'exploitation est chargé de la sécurité liée à l'exécution des programmes en garantissant que les ressources ne sont utilisées que par les programmes et utilisateurs possédant les droits adéquats.
- **Gestion des fichiers** : le système d'exploitation gère la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.
- **Gestion des informations** : le système d'exploitation fournit un certain nombre d'indicateurs permettant de diagnostiquer le bon fonctionnement de la machine.



# Autres qualités requises du système d'exploitation

Un système d'exploitation doit se « faire oublier » : la fonction d'un ordinateur est d'exécuter les applications, pas le système d'exploitation.

- Utilisation efficace des ressources
- Fiabilité
- Tolérance aux fautes (du matériel, des utilisateurs, des programmes)
- La qualité de l'interface (en particulier pour les systèmes interactifs)
- Convivialité
- Simplicité d'utilisation
- Documentation
- Bonne intégration au réseau
- Sécurité et protection
- Répertoire étendu des fonctions



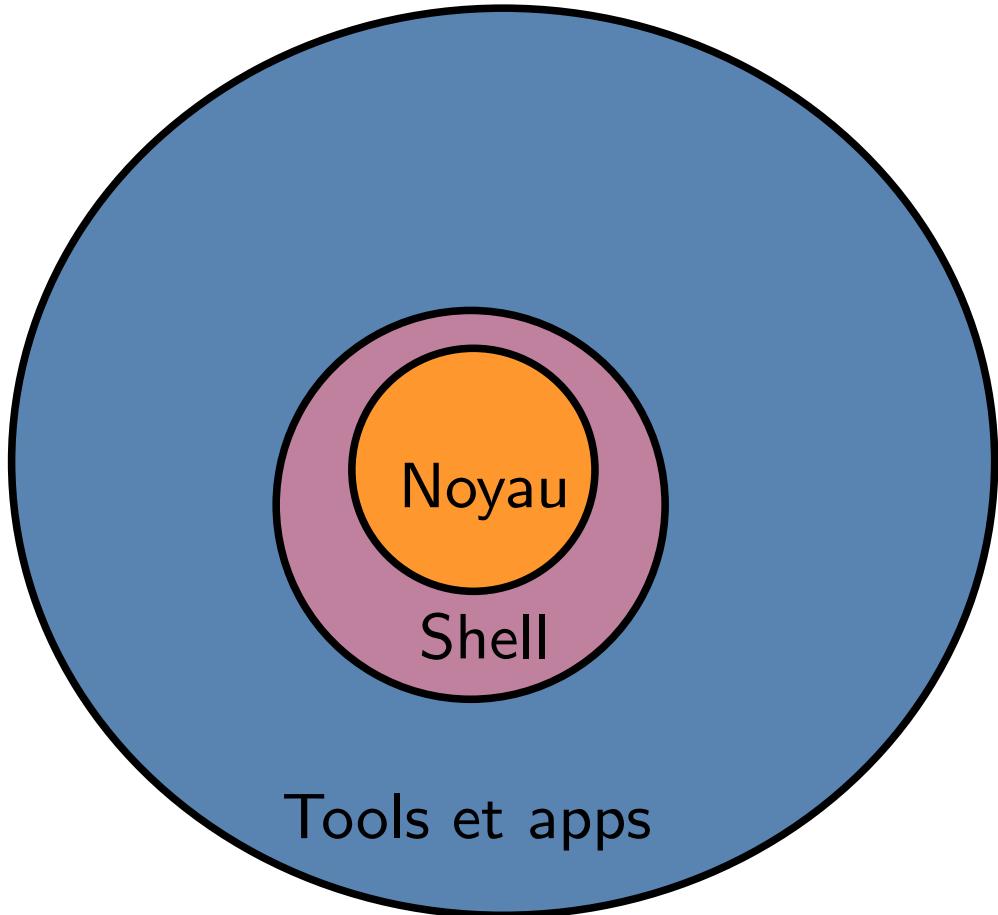
# Deux modes de fonctionnement de l'OS

- Le **mode noyau** ou **superviseur** (nécessaire pour l'exécution de certaines instructions qui ont besoin de droits)
  - Le **mode utilisateur** (le mode de base, pour des programmes tq un compilateur, un éditeur, programmes utilisateurs...).
- ⇒ selon le type de programme qu'il a à gérer (programme d'application ou programme système) et des droits, l'OS va le lancer dans un de ces deux modes.



## **Composants des systèmes d'exploitation**

# SE : segmentation en couches « d'oignon »

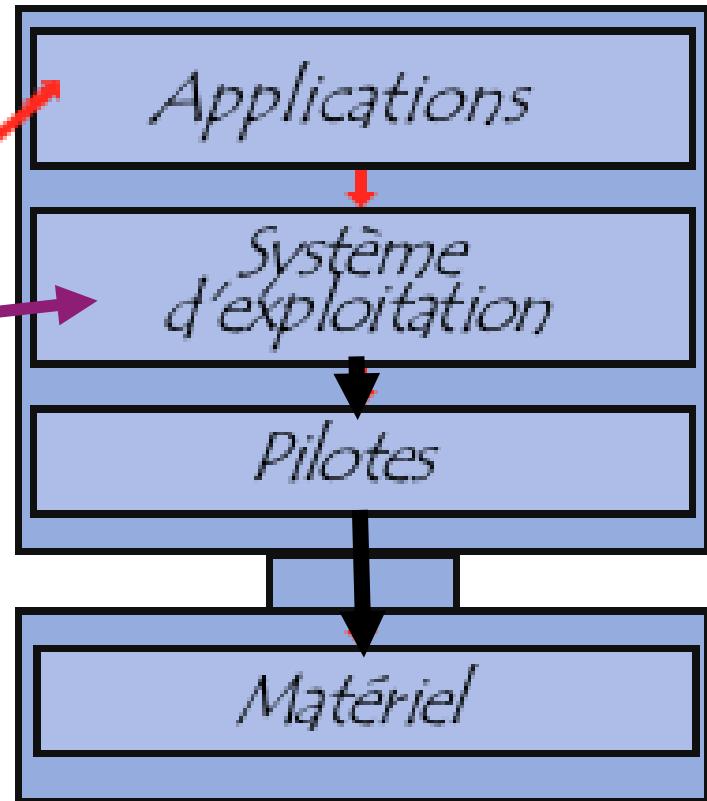
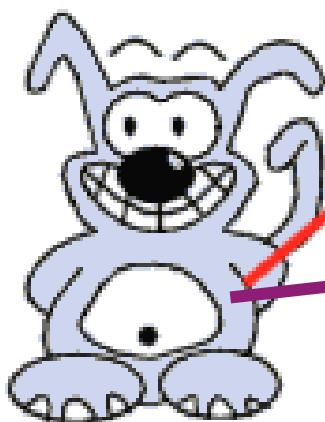


Les parties d'un système Unix

- Le **noyau** (kernel) représentant les **fonctions fondamentales** du SE, tq la gestion de la mémoire, des processus, des fichiers (système de fichiers), des entrées-sorties principales, et des fonctionnalités de communication.
- L'**interpréteur de commande** (en anglais Shell, traduisez « coquille » par opposition au noyau) permettant la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes (scripts).
- Un ensemble (plus ou moins minimal) de **tools et d'apps**.



# Recap : les différentes couches qui constituent un ordinateur



Votre interaction avec un SE :

- Via les **applications** (en GUI)
- Via le **shell** (langage de commandes)

Bientôt !



# Recap : Les fonctions d'un système d'exploitation

## Fonctions « visibles »

- Interface utilisateur
- Accès aux périphériques
- Gestion des disques
- Lancement des programmes

## Fonctions « cachés »

- Partage du processeur
- Partage de la mémoire
- Gestion d'événements
- Mécanismes de synchronisation entre les programmes





- Caractéristiques et types des systèmes d'exploitation
- Historique de la famille des UNIX
- Généralité sur les dérivés d'UNIX
- Quelques mots sur Microsoft windows.

# Récap CM1



Code d'événement  
**DBJAFM**

1 Allez sur [wooclap.com](https://wooclap.com)

2 Entrez le code d'événement dans le bandeau supérieur

1 Envoyez @DBJAFM au 06 44 60 96 62

2 Vous pouvez participer

Désactiver les réponses par SMS

A screenshot of a mobile device displaying a poll interface from Wooclap. The top right shows the event code "DBJAFM". Below it, two steps are outlined: one for web users to visit wooclap.com and enter the event code, and another for users to send a text message to 06 44 60 96 62. A "Désactiver les réponses par SMS" button is visible. The bottom part of the screen shows a speech bubble icon and the text "Vous pouvez participer".



# **Caractéristiques et types de systèmes d'exploitation**



# Différents types de systèmes d'exploitation, selon les services rendus

- **mono/multi-tâche** :
  - = capacité du système à pouvoir exécuter plusieurs processus **simultanément** ; par exemple effectuer une compilation et consulter le fichier source du programme correspondant.  
**Préemptif vs collaboratif**  
**Unix, OS/2 d'IBM, Windows**
- **mono/multi-utilisateurs** :
  - = capacité à pouvoir gérer un panel d'utilisateurs utilisant **simultanément** les mêmes ressources matérielles.  
**Unix/Linux, MacOS, MVS, Gecos, ...**



# Différents types de systèmes d'exploitation, selon les services rendus

Système	Codage	Mono-Utilisateur	Multi-Utilisateur	Mono-Tâche	Multi-Tâche	Type MT
DOS	8/16 bits	X		X		Mono
Windows 3.1	16/32 bits	X			X	Coopératif
Windows 95/98/Me	32 bits	X			X	Coopératif/ Préemptif
Windows NT/2000	32 bits		X		X	Préemptif
Windows XP/10	32/64 bits		X		X	Préemptif
Unix/LInux	32/64 bits		X		X	Préemptif
MacOs	32/64 bits		X		X	Préemptif
VMS	32 bits		X		X	Préemptif



# Différents types de systèmes d'exploitation, selon leur capacité à évoluer

## Systèmes fermés (ou propriétaires) :

- Extensibilité réduite : Quand on veut rajouter des fonctionnalités à un système fermé, il faut remettre en cause sa conception et refaire une archive (système complet).

Unix, MS-Dos ...

- Il n'y a aucun ou peu d'échange possible avec d'autres systèmes de type différent, voir même avec des types identiques.

Unix, BSD et SystemV.

Android et iOS

## Systèmes ouverts :

- Extensibilité accrue : Il est possible de rajouter des fonctionnalités et des abstractions sans avoir à repenser le système et même sans avoir à l'arrêter sur une machine.

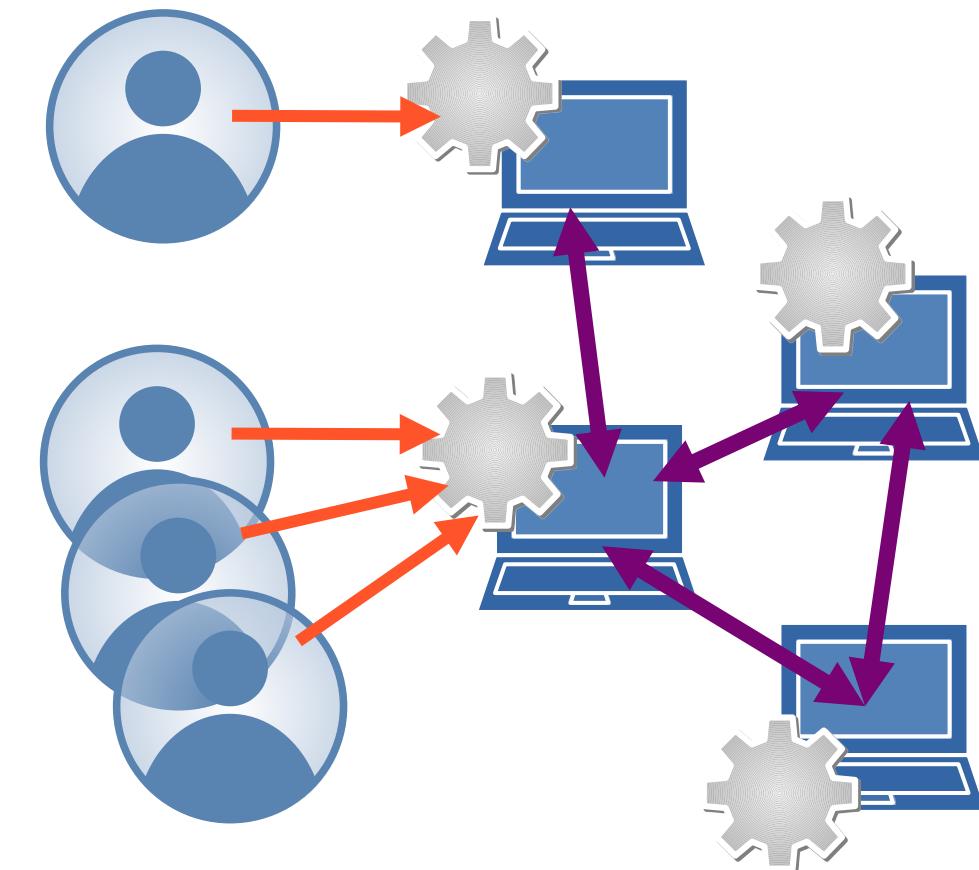
Linux

- Cela implique souvent une conception modulaire basée sur le modèle « client-serveur ».
- Cela implique aussi une communication entre systèmes, nécessitant des modules spécialisés.

# Différents types de systèmes d'exploitation, selon leur architecture : systèmes centralisé

- L'ensemble du système est entièrement présent sur la machine considérée.
- Les machines éventuellement **réliées** sont vues comme des entités étrangères disposant elles aussi d'un système centralisé.
- Le système ne gère que les ressources de la machine sur laquelle il est présent.

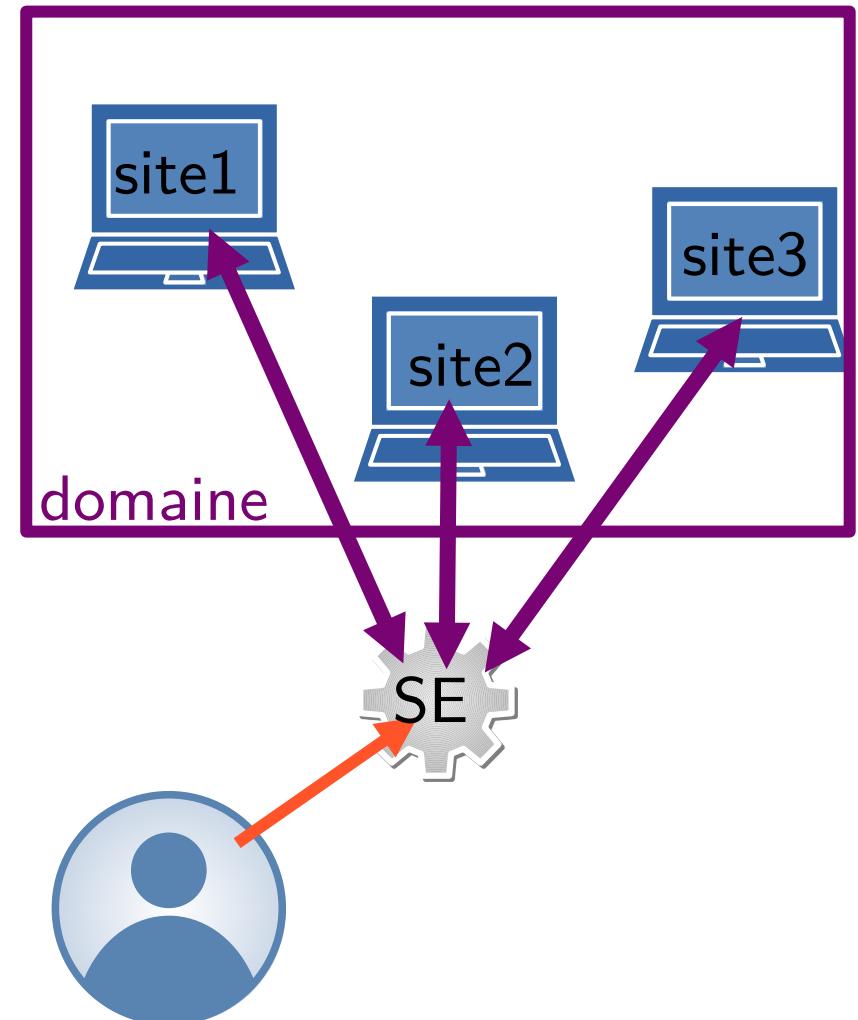
**UNIX**, même si les applications réseaux (X11, FTP, Mail...) se sont développées.



# Différents types de systèmes d'exploitation, selon leur architecture : système répartis = distributed systems

- Le SE contrôle un ensemble (**domaine**) de machines (**site**).
- L'utilisateur.ice n'a pas à se soucier de la localisation exacte des ressources. Quand iel lance un programme, iel n'a pas à connaître le nom de la machine qui l'exécutera. Le SE apparaît à ses yeux comme **une machine virtuelle monoprocesseur** même lorsque cela n'est pas le cas.
- Ces systèmes exploitent au mieux les capacités de parallélisme d'un domaine, et ils offrent des solutions aux problèmes de la **résistance aux pannes**.

Mach, Amoeba, Andrew, Athena, Locus, ...



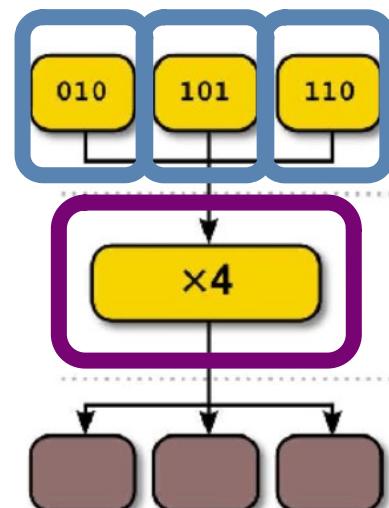
# Différents types de systèmes d'exploitation, selon l'architecture matérielle qui les supporte

**Architecture monoprocesseur** : (pseudo)-parallélisme possible grâce à la **commutation** rapide entre les différents processus (pour donner l'illusion d'un parallélisme)

**Architecture multiprocesseur** (parallélisme natif) : grande variété d'architectures [Flynn, 1972] :

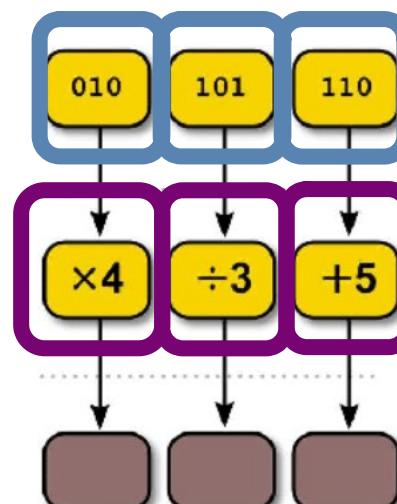
## SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata)

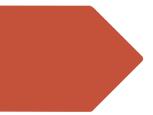
Tous les processeurs exécutent la même instruction, mais sur des données différentes.



## MIMD (**M**ultiple **I**nstructions **M**ultiple **D**ata)

Chaque processeur est complètement indépendant des autres et exécute des instructions différentes sur des données différentes.





# Architecture : appellations alternatives

- **Architecture fortement couplée** : Ce sont principalement des architectures à mémoire commune.
- **Architecture faiblement couplée** : Ce sont des architectures où chaque processeur possède sa propre mémoire locale ; c'est le cas d'un réseau d'ordi.
- **Architecture mixte** : Ce sont des architectures à différents niveaux de mémoire (commune et privée).

**Remarque :** Il n'y a pas de système universel pour cette multitude d'architectures. Les constructeurs de supercalculateurs ont toujours développé leurs propres systèmes. Aujourd'hui, compte tenu de la complexité croissante des systèmes d'exploitation et du coût inhérent, la tendance est à l'harmonisation notamment via le développement de systèmes polyvalents.



## Différents types de systèmes d'exploitation, selon leur relation au temps : systèmes temps-réel

- Généralement dans les **systèmes embarqués** (satellites, sondes, avions, trains...).
- Exécution des programmes soumise à des **contraintes temporelles** (absolues ou relatives) : les résultats de l'exécution d'un programme n'est **plus valide au delà d'un certain temps** connu et déterminé à l'avance.
- Système temps réel **strict** : aucun dépassement de contrainte n'est toléré
- Système temps réel **souple** : s'accommode des dépassements de contraintes dans certaines limites

Linux-RT, RTX, Windows CE, Embedded Linux, Symbian OS, Palm OS et VxWorks

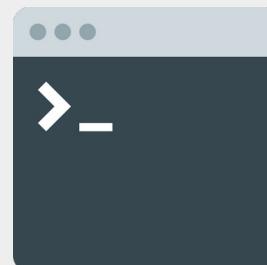


# Différents types de systèmes d'exploitation : 5 générations successives

Quand	Nom	Description
années 50'	traitements par lots (batch)	calculs les uns après les autres.
années 60'	multiprogrammation	exécution simultanée de plusieurs programmes visant l'utilisation efficace de la puissance de calcul du processeur.
1970	temps partagé	techniques avancée de multiprogrammation pour donner l'impression à chaque utilisateur.ice du système qu'iel est lea seul.e à utiliser l'ordi
milieu 70'	temp réel	avec garantie que toute opération se termine dans un délai donné
années 90'	distribués	utilisation des ressources de plusieurs ordinateurs à la fois



# Généralités sur UNIX et ses dérivés



# Historique 1969 - 1979 : les premiers pas universitaires

- Eté 1969 : **Ken Thompson**, chercheur aux **BELL** Laboratories, écrit la **version expérimentale d'UNIX** : système de fichiers exploité dans un environnement **mono-utilisateur, multi-tâche**, le tout étant écrit **en assembleur** sur un ordi de récupération **PDP 7** de 1964.
  - 1ère justification officielle : traitement de texte pour secrétariat (écriture de brevets)
  - Puis : étude des principes de programmation, de réseaux et de langages.



Le PDP 7 Oslo

<https://en.wikipedia.org/wiki/PDP-7>



# Historique 1969 - 1979 : les premiers pas universitaires

- **Eté 1969** : **Ken Thompson**, chercheur aux **BELL** Laboratories, écrit la **version expérimentale d'UNIX** : système de fichiers exploité dans un environnement **mono-utilisateur, multi-tâche**, le tout étant écrit **en assembleur** sur un ordi de récupération **PDP 7** de 1964.
  - 1ère justification officielle : traitement de texte pour secrétariat (écriture de brevets)
  - Puis : étude des principes de programmation, de réseaux et de langages.
- **Eté 1973** : réécriture du noyau et des utilitaires d'UNIX **en C** (langage compilé implémenté en 1972 par **Dennis Ritchie** à partir du langage interprété B, écrit par Ken Thompson).
- **En 1974** distribution d'UNIX aux **Universités** (**Berkeley** et **Columbia** notamment). Il se compose alors :
  - d'un système de fichiers modulaire et simple,
  - d'une interface unifiée vers les périphériques par l'intermédiaire du système de fichiers,
  - du multi-tâche
  - et d'un interpréteur de commandes (**shell**) flexible et interchangeable.

# Historique 1979 - 1984 : les premiers pas commerciaux

1979, avec la version 7, UNIX se développe commercialement :

- Par des sociétés privées comme [Microport](#) (1985), [Xenix-Microsoft](#) (1980) ... qui achetèrent les sources et le droit de diffuser des binaires.
- Des **UNIX-like** apparaissent ; le noyau est entièrement réécrit.
- L'[université de Berkeley](#) fait un portage sur les ordis **VAX** (ordi avec mémoire virtuelle) : **BSD UNIX 32V**.
- [AT&T](#) vend la version 7 sur les ordinateurs de la gamme **PDP 11**.

⇒ Se développent plusieurs produits différents, tous markétés « UNIX », et pas compatibles entre eux... Chaque industriel ajoute des fonctionnalités et pas grand monde partage les infos.

# Historique 1984 - 1993 ... : « UNIX war » et la standardisation

**1984** un groupe de vendeurs (dont Siemens, Phillips et Ericson) met en place **X/Open**, chargé de définir un standard UNIX pour permettre la **portabilité**.

**1987** **AT&T** et **Sun Microsystems** pactisent pour mettre en place un système UNIX unifié (system V x BSD)

**1988** : Contre-attaque d'autres industriels pro-BSD (**OSF/1** via **Open Software Foundation**).  
Contre-contre-attaque d'**AT&T** (**System V** via **UNIX international**).  
D'autres standards sont développés en parallèle (**POSIX**), d'autres OS aussi (Windows)

**Mai 1993** : des membres des 2 camps annoncent l'initiative **COSE** (Common Open Software Environment) = accord pour le développement d'applications dans un environnement commun.

**1994** : fusion avec X/Open → the **Open Group**,  
**aujourd'hui**, le **Open Group** est le seul à décerner le label UNIX (**The Single UNIX Specification**)

**Au final** : guerres techniques et culturels qui ont fini par passer au second plan, en faveur d'autres OS (Windows, macOS (dérivé de BSD), Linux).



# La normalisation d'UNIX

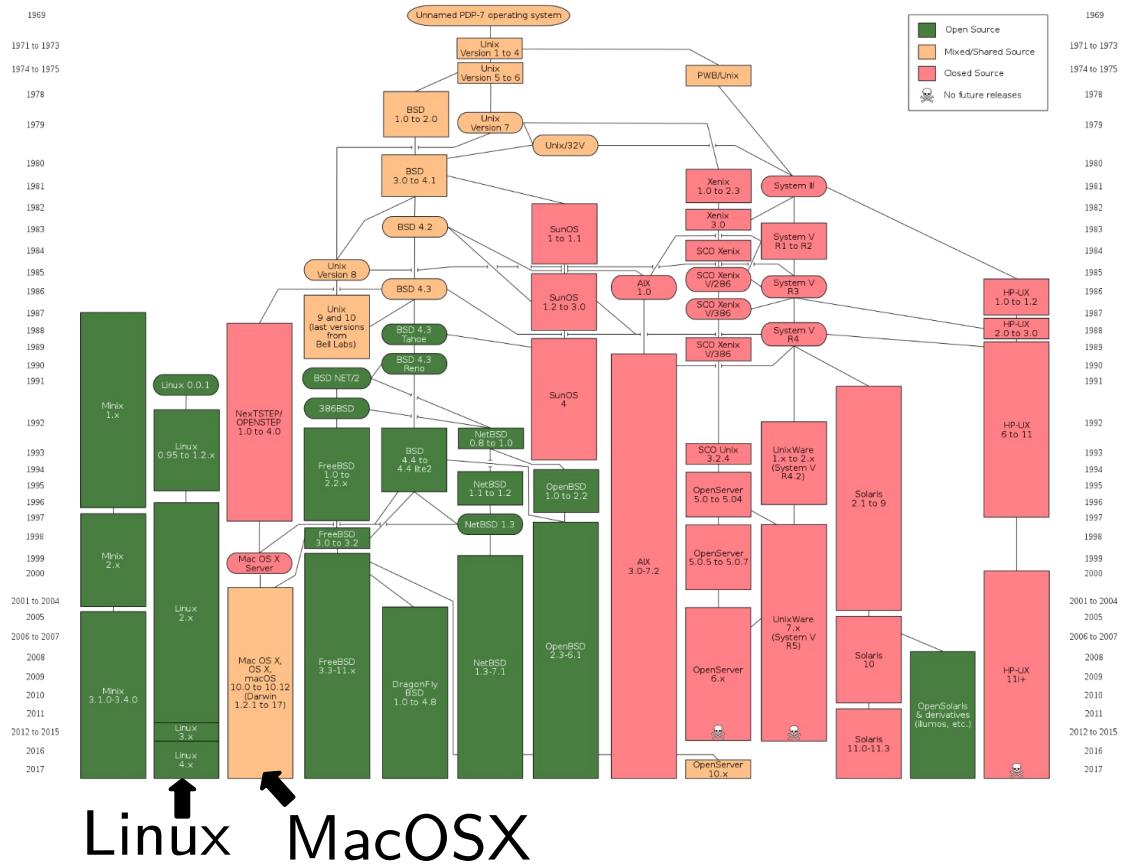
Historique :

- 1985 : Le **SVID** (System V Interface Definition) d'AT&T, qui définit l'interface d'application du Système V.2 et non pas son implémentation.

Toujours utiles :

- Pour les programmeur.euses et les system designers : les standards nous donnent un cadre qui rendent nos codes portables
- Pour les utilisateur.ices : les standards nous donnent un cadre qui rendent les outils facile à utiliser, « ça va marcher ».
  - **POSIX** (Portable Operating System Interface X), définit l'interface avec le système. Interface labellisée par l'ANSI (American National Standard Institute) et l'ISO (International Standard Organisation). [[Linux](#), [macOS](#)]
  - les **Spec1170**, devenue Single UNIX Specification (**SUS**) de l'Open Group. [[macOS](#)]

# Arbre généalogique de la famille des UNIX



Unix est l'ancêtre de bien des systèmes d'exploitation, notamment **Linux, BSD, Solaris, AIX, MacOSX**.



# Caractéristiques générales du noyau UNIX

- **Multi-tâche / multi-utilisateur**
  - Plusieurs utilisateur.ices peuvent travailler en même temps ; chacun.e peut effectuer une ou plusieurs tâches en même temps.
  - Une tâche ou un processus = programme s'exécutant dans un environnement spécifique.
  - Les tâches sont protégées ; certaines peuvent communiquer, c-à-d échanger ou partager des données, se synchroniser dans leur exécution ou le partage de ressources. Certaines tâches peuvent être « temps réel ».
- **Système de fichiers arborescent**
  - Arborescence unique de fichiers, même avec plusieurs périphériques (disques) de stockage.
- **Entrée/Sorties compatible fichiers, périphériques et processus**
  - Les périphériques sont manipulés comme des fichiers ordinaires.
  - Les canaux de communication entre les processus (pipe) s'utilisent avec les mêmes appels systèmes que ceux destinés à la manipulation des fichiers.

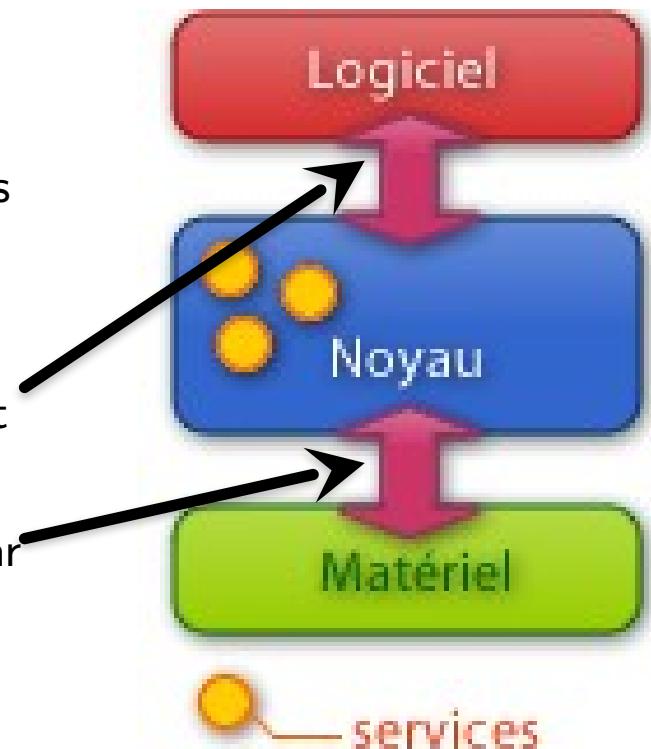


# Les qualités du système UNIX

- Code source facile à lire et à modifier ; disponible commercialement.
- Interface utilisateur simple et très puissante (mais « non-conviviale » ?).
- Le système est construit sur un petit nombre de primitives de base ; de nombreuses combinaisons possibles entre programmes. (modularité)
- Les fichiers ne sont pas structurés au niveau des données, ce qui favorise une utilisation simple.
- Toutes les interfaces avec les périphériques sont unifiées (système de fichier).
- Le programmeur.euse n'a jamais à se soucier de l'architecture de la machine sur laquelle iel travaille.
- C'est un système disponible sur de nombreuses machines, allant du super-calculateur au micro-ordinateur (PC), en passant par les smartphones.
- Les utilitaires et programmes proposés en standard sont très nombreux.

# Caractéristiques du noyau UNIX

- UNIX comprend un **noyau (kernel)** et des **utilitaires (services)**.
- Irremplaçable par l'utilisateur, le noyau gère les processus, les ressources (mémoires, périphériques ...) et les fichiers.
- Tout autre traitement doit être pris en charge par des **utilitaires** ; c'est le cas de l'*interprète de commande* (sh, csh, ksh, tcsh ...).
  - Interfaces du noyau
    - L'**interface** entre le **noyau UNIX** et les **programmes utilisateurs** est assurée par un ensemble d'**appels systèmes**.
    - L'**interface** entre le **noyau UNIX** et les **périphériques** est assurée par les **gestionnaires de périphériques (devices driver)**



[https://fr.wikipedia.org/wiki/Noyau\\_de\\_système\\_d%27exploitation.](https://fr.wikipedia.org/wiki/Noyau_de_système_d%27exploitation)

# UNIX : interface



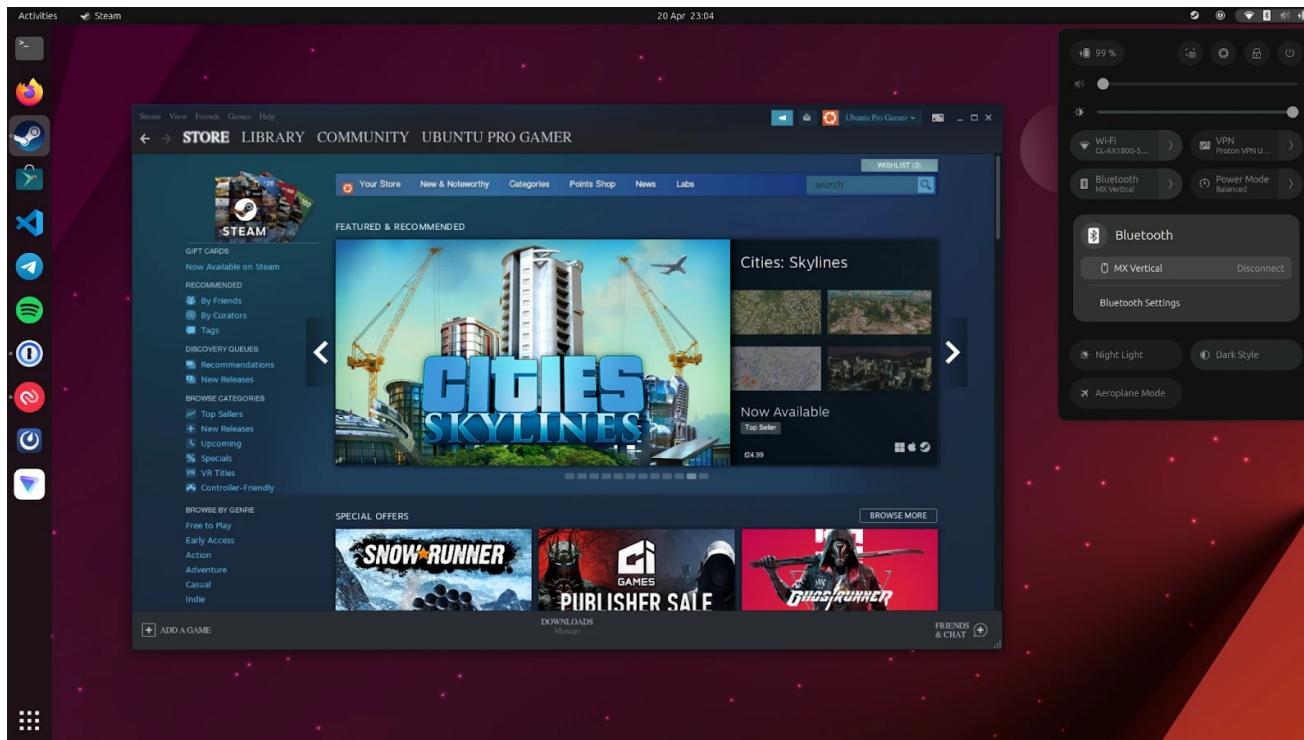
## Bureau Unix vers le début des années 1990.

[http://www-ens.iro.umontreal.ca/~gottif/bdeb/infc32/c1\\_fichiers/image049.jpg](http://www-ens.iro.umontreal.ca/~gottif/bdeb/infc32/c1_fichiers/image049.jpg)

# Deux environnements très différents sous Unix

- l'environnement graphique (des boutons et des fenêtres)
- l'environnement console / terminal, en ligne de commande

Et il existe plusieurs environnements console : les **Shells**.



```
jfa@jfa-VirtualBox:~$ cd  
jfa@jfa-VirtualBox:~$ pwd  
/home/jfa  
jfa@jfa-VirtualBox:~$ █
```



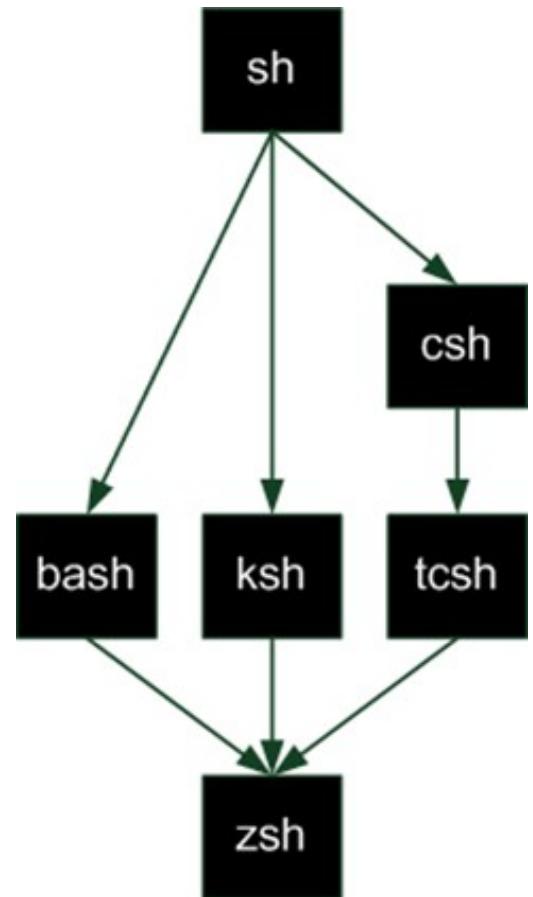
# Qu'est-ce qu'un Shell Unix ?

Un **Shell Unix** est un **interpréteur de commandes** destiné aux systèmes d'exploitation Unix (et de type Unix) qui permet d'accéder aux fonctionnalités internes du système d'exploitation. Il se présente sous la forme d'une **interface en ligne de commande** accessible depuis la **console** ou un **terminal**. L'utilisateur lance des commandes sous forme d'une **entrée texte exécutée ensuite par le shell**.

Dans les différents systèmes d'exploitation Microsoft Windows, le programme analogue est par exemple **powershell**.

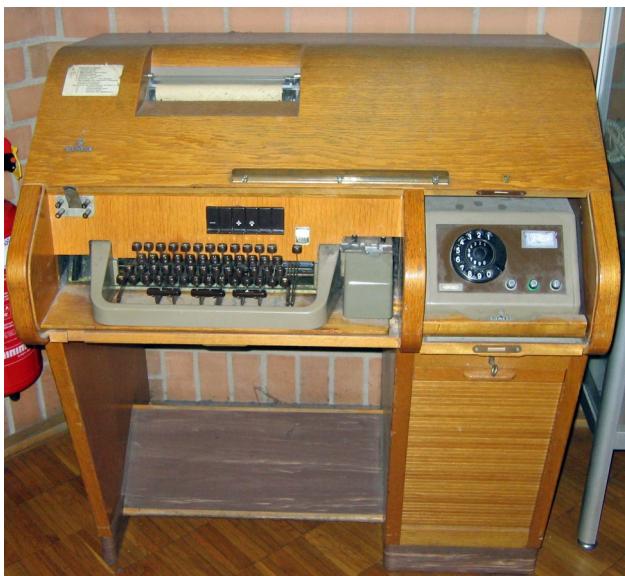
# Les principaux Shells Unix

- **Sh : Bourne Shell.** L'ancêtre de tous les Shells. Sa syntaxe des commandes est proche de celle des premiers UNIX
- **Bash : Bourne Again Shell.** Une amélioration du Bourne Shell augmenté de la plupart des fonctionnalités avancées du C shell, un script Bourne shell sera correctement interprété avec un Bash, disponible par défaut sous GNU/Linux et Mac OS X.
- **ksh : Korn Shell.** Un Shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec Bash.
- **csh : C Shell.** Un Shell utilisant une syntaxe proche du langage C.
- **tcsh : Tenex C Shell.** C'est une extension et amélioration du C shell d'origine.
- **zsh : Z Shell.** Shell assez récent reprenant les meilleures idées de Bash, ksh et tcsh



# Terminal / TTY / Console / Shell

- Terminal = extrémité
- Téléscripteur = télétype = TTY
- Console = « pupitre de commandes »
- émulateur de terminal = console virtuelle

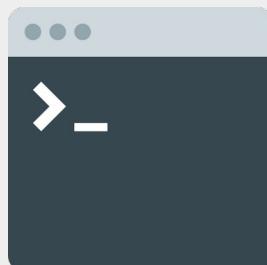


```
(jammy-base r5828)nanis@C302L-G24P07: ~
(base) (jammy-base r5828)nanis@C302L-G24P07:~$ ls
baseconfig          Desktop   miniconda3  ownCloud
baseconfig-gitlab  Documents  Modèles    Public
bluetooth-pods.sh  Images     Musique    signal-desktop-keyring.gpg
(base) (jammy-base r5828)nanis@C302L-G24P07:~$ pwd
/home/nanis
(base) (jammy-base r5828)nanis@C302L-G24P07:~$ █
```



# Généralités sur UNIX et ses dérivés

## Dérivé n°1 : Linux





# **Historique 1991 - ... : LINUX, le renouveau d'UNIX**

**1991** : **Linus B. Torvalds** (étudiant de 21 ans à l'univ. D'Helsinki, Finlande) étudie MINIX (un OS UNIX-like écrit par A. Tannenbaum)

**Août 1991** : 1ère version de LINUX 0.01. C'est une réécriture de MINIX, avec des ajouts de nouvelles fonctionnalités et la diffusion des sources sur « Internet »  
→ une **version instable**

**Mars 1994** : 1ère **version stable**, 176 250 lignes de code.

**Janvier 2004** : v.2.6.0 qui respecte la norme POSIX (code source portable) et le code source est gratuit.

**Novembre 2013** : v.3.12.0 écrite en C et en assembleur, sous licence GNU GPL 2.

**2022** : 27,8 millions lignes de code (projet libre qui en contient le plus).

# Linux

Post de Linus Torvalds pour partager son OS :

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)  
Newsgroups: comp.os.minix  
Subject: What would you like to see most in minix?  
Summary: small poll for my new operating system  
Message-ID:  
Date: 25 Aug 91 20:57:08 GMT  
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Source : l'archive Usenet (l'ancêtre d'internet)

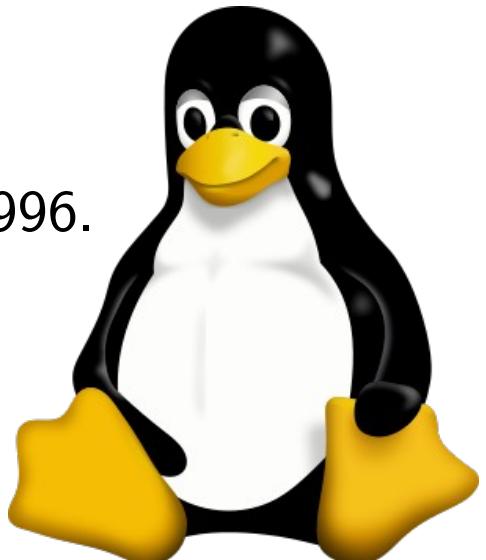
# LINUX

Accronyme récursif :

**LINUX = LINUX Is Not UniX = ...**

Mascotte :

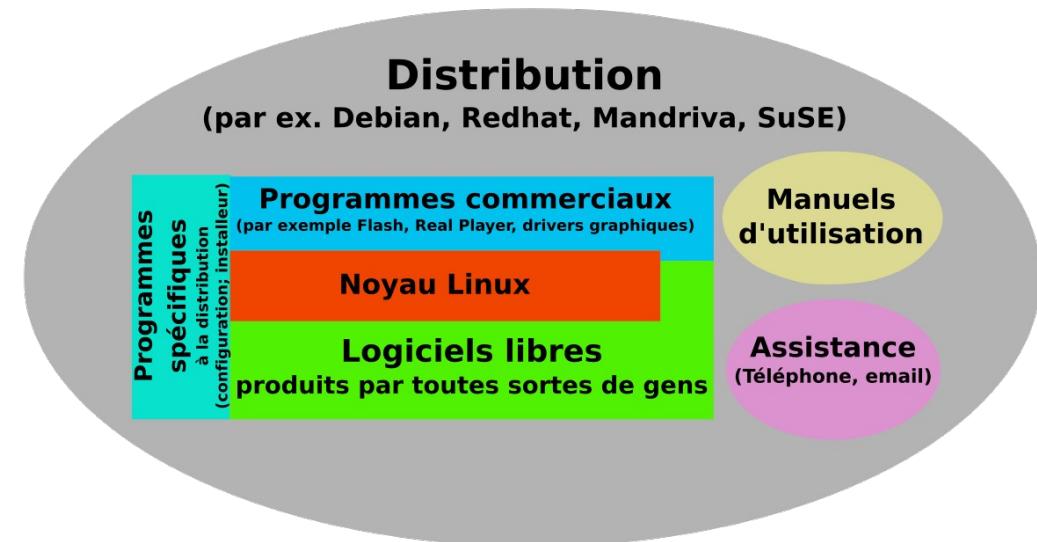
le **pingouin manchot TUX** (« Torvalds Unix »), créé par Larry Ewing en 1996.



# Distributions LINUX (ou « distro »)

## Définition :

Ensemble cohérent de logiciels assemblé autour d'un **noyau Linux** (des pilotes, les bibliothèques, les utilitaires, ... et une surcouche custom de logiciels).





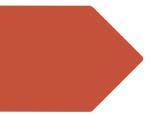
## « Linux » vs « GNU » vs « GNU/Linux »

À l'origine :

- À sa création (dans les années 90) le **noyaux Linux** n'était pas directement utilisable : un noyau n'est pas un système d'exploitation complet.
- En parallèle (depuis 1983), le **projet GNU (Richard Stallman)** travaillait à la mise en place d'un système d'exploitation compatible UNIX, et basé sur la **philosophie GNU** (logiciel libre). Il avait plein de composants logiciels mais il lui manquait un noyau.

Donc, selon le contexte :

- « **GNU** » se rapporte aux logiciels du projet GNU, à un système d'exploitation dont le noyau est très probablement Linux (d'où le fait que le SE soit parfois appelé **GNU/Linux**), mais aussi à une licence de code (**GNU GPL**)...
- « **Linux** » se rapporte à un noyau, ou à un système d'exploitation dont le noyau est Linux, et qui peut (ou pas) avoir sa couche logicielle issue (tout, ou partie) du projet GNU.



# Distributions LINUX

- **Slackware** : la plus vieille distribution encore en activité (1993)
- **Fedora** : entreprise RedHat
- **S.uS.E** : grande robustesse
- **Mandrake** : *Caennaise* basée sur **RedHat**
- **Caldera** : inclut des produits commerciaux
- **Gentoo** : système entièrement compilé à partir des sources, gestionnaire de paquetage Portage
- **Trinux** : fonctionne uniquement en mémoire, outils d'audit des réseaux
- **TurboLinux** : version en cluster, payante, pour gros serveurs
- **Knoppix** : très populaire, sans disque dur
- **Arch** : se veut simple (principe KISS – Keep it Simple Stupid)
- **Ubuntu** : entreprise Canonical, très populaire, basée sur **Debian**
- **Debian** : non commerciale et de grande qualité (genre vraiment).  
et bien d'autres... (dont **Android** sur nos smartphones).

Arbre des distributions GNU/Linux sur [http://fr.wikipedia.org/wiki/Distribution\\_Linux](http://fr.wikipedia.org/wiki/Distribution_Linux) .

# Distributions LINUX

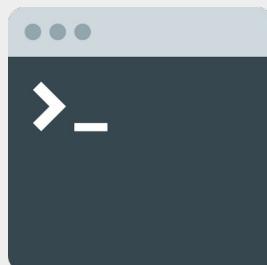


<https://linux.developpez.com/actu/137043/Quelles-sont-vos-distributions-Linux-preferees-Et-pour-quelles-utilisations-Merci-de-partager-votre-experience-avec-les-distributions-Linux/>



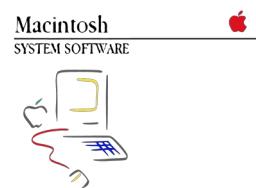
# Généralités sur UNIX et ses dérivés

## Dérivé n°2 : MacOS



# Mac OS (renommé OS X puis MacOS)

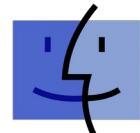
- Le système d'exploitation des ordinateurs Macintosh d'Apple → intimement **lié au matériel**.
- **Partiellement propriétaire** : certaines parties de FreeBSD (une saveur d'Unix) ont été réutilisées par Apple dans ce système d'exploitation.



1984 - 1988



Macintosh



MacOS



Mac OS X



Mac OS X

OSX

2001 - 2003

2003 - 2012

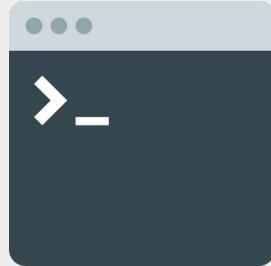
2012 - 2016

macOS

macOS

2016 - 2017

2017 - now



Windows :  
**la vache à lait de Microsoft**

# Windows

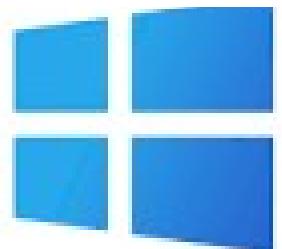
- Propriétaire, développé par Microsoft

Historique :

- 1981 : MS-DOS « Microsoft Disk Operating System »:  
Système monotâche, mono-utilisateur, et pas de mode protégé.  
Interface en ligne de commande (langage CMD).
- 1985 : Windows v1 = interface graphique pour MS-DOS
- 1993 : Nouveau kernel Windows NT (« New technology »).  
Système d'exploitation multitâche préemptif, multi-utilisateur, multiprocesseur.  
Langage : powershell.

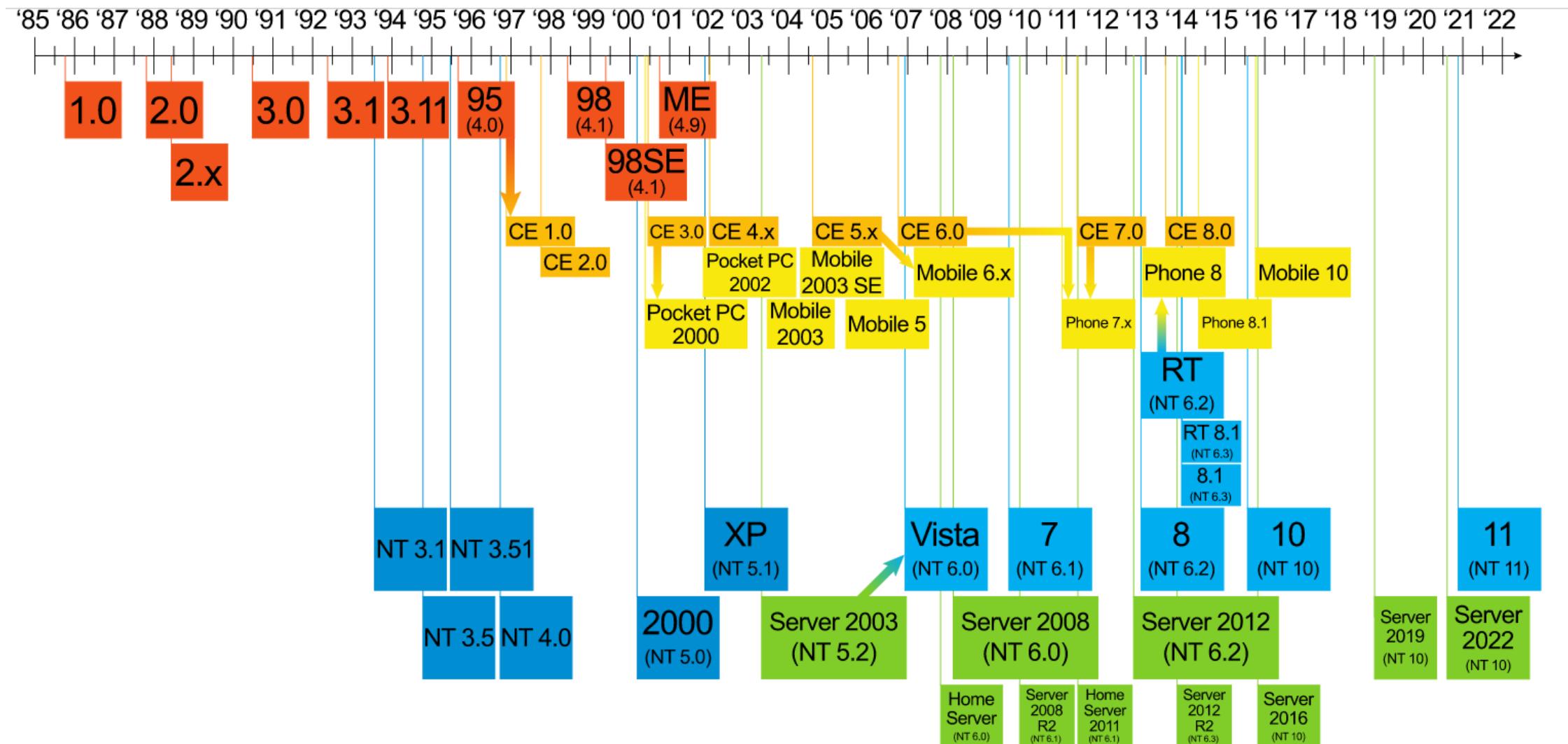


Écran de démarrage MS-DOS 6.22.  
Démarrage de MS-DOS...  
  
Vérification de la mémoire étendue par HIMEM...  
Vérification terminée.  
  
C:\>C:\DOS\SMARTDRV.EXE /X  
Fonction MODE PREPARE pour la page de codes terminée  
Fonction MODE SELECT pour la page de codes terminée  
C:\>\_



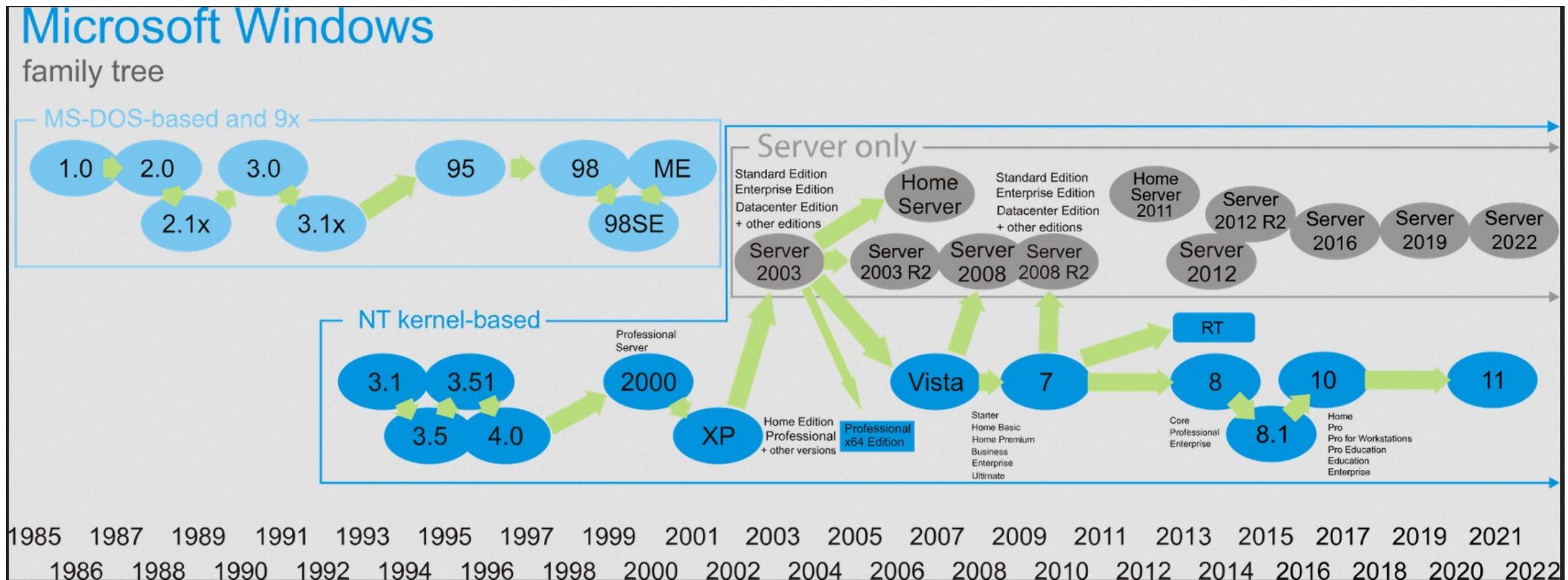
Sous-système Windows pour Linux (WSL) : plus besoin de dual boot.

# Windows : l'arbre familial, versions



[https://upload.wikimedia.org/wikipedia/commons/1/17/Suite\\_des\\_versions\\_de\\_Windows.svg](https://upload.wikimedia.org/wikipedia/commons/1/17/Suite_des_versions_de_Windows.svg)

# Windows : l'arbre familial, noyau



[https://upload.wikimedia.org/wikipedia/commons/thumb/0/0e/Windows\\_Family\\_Tree.svg/1889px-Windows\\_Family\\_Tree.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/0/0e/Windows_Family_Tree.svg/1889px-Windows_Family_Tree.svg.png)

# Le quasi-monopole de Microsoft Windows

- « vente-liée » : Windows est (quasi) systématiquement installé quand on achète un ordinateur
- « racketiciel » (AFUL : Association francophone des utilisateurs de logiciels libres)



En cas d'exposition, contacter  
**www.april.org**





## Recap

- Il existe de nombreux OS, certains étant adaptés à des cas très spécifiques.
- Plusieurs OS (MacOS, Linux, Android) ont un ancêtre commun : **UNIX**, et partage donc une organisation proche.
- Les protagonistes des histoires de tous ces OS ne sont pas si vieux (par ex, **Ken Thompson** travaille chez **Google** depuis 2006 et est impliqué dans le développement du **langage Go**).



# Teaser

TD 1 et 2:

Première utilisation de Linux, via l'interface en lignes de commandes. :))))

CM3 et suivants :

Reprise des notions vues en TD : utilisateur, fichiers, droits, plein de lignes de commandes



## « ça marche pô »

- Vérifiez qu'il n'y a pas une faute de frappe (/!\ les commandes sont sensibles à la case).
- Vérifiez que vous avez les bons arguments, le bon nombre
- Regardez les indications du terminal (le terminal est votre ami)
- Préparez-vous à poser votre question à son.a voisin.e / le.a prof / internet :
  - Qu'est ce que voulais faire ?
  - Quelles sont mes entrées ?
  - Quelles sont les sorties attendues ?





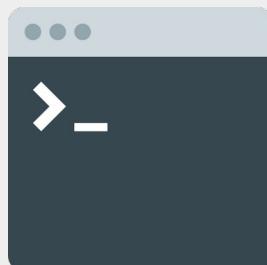
- Fin du CM2 : quelques mots sur MacOS et Windows
- mini-recap des TD 1 et 2
- les utilisateurs en UNIX
- Bonus : tips généraux

(pas de wooclap, désolée)



# Généralités sur UNIX et ses dérivés

## Dérivé n°2 : MacOS



# Mac OS (renommé OS X puis MacOS)

- Le système d'exploitation des ordinateurs Macintosh d'Apple, intimement **lié au matériel**.
- **Partiellement propriétaire** : certaines parties de FreeBSD (une saveur d'Unix) ont été réutilisées par Apple dans ce système d'exploitation.



1984 - 1988



Macintosh



MacOS



Mac OS X



Mac OS X

OSX

2001 - 2003

2003 - 2012

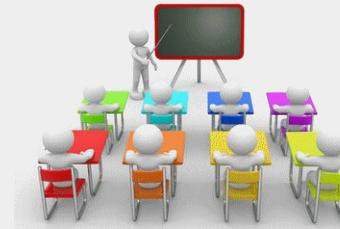
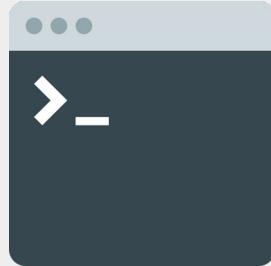
2012 - 2016

macOS

macOS

2016 - 2017

2017 - now



Windows :  
**la vache à lait de Microsoft**

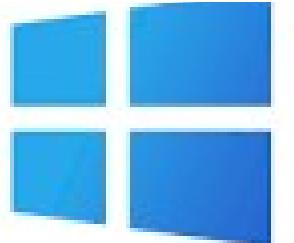
# Windows

- Propriétaire, développé par Microsoft

Historique :

- 1981 : **MS-DOS** « Microsoft Disk Operating System »:  
Système **monotâche**, **mono-utilisateur**, et **pas de mode protégé**.  
Interface en ligne de commande (langage CMD).
- 1985 : **Windows v1** = interface graphique pour MS-DOS
- 1993 : Nouveau kernel **Windows NT** (« New technology »).  
Système d'exploitation **multitâche préemptif**, **multi-utilisateur**, **multiprocesseur**.  
Langage : **powershell**.

Écran de démarrage MS-DOS 6.22.  
Démarrage de MS-DOS...  
  
Vérification de la mémoire étendue par HIMEM...  
Vérification terminée.  
  
C:\>C:\DOS\SMARTDRV.EXE /X  
Fonction MODE PREPARE pour la page de codes terminée  
Fonction MODE SELECT pour la page de codes terminée  
C:\>\_



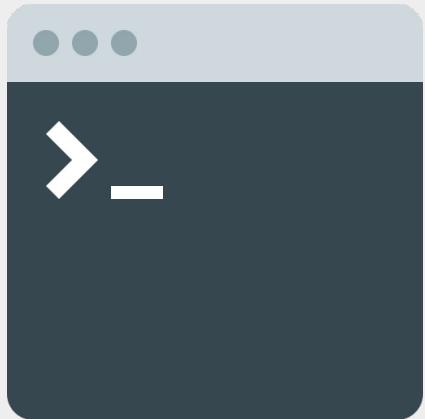
Sous-système Windows pour Linux (**WSL**) : plus besoin de dual boot.

# Le quasi-monopole de Microsoft Windows

- « vente-liée » : Windows est (quasi) systématiquement installé quand on achète un ordinateur
- « racketiciel » (AFUL : Association francophone des utilisateurs de logiciels libres)



En cas d'exposition, contacter  
**www.april.org**



# Linux :

## Le ~~petit~~ grand frère de UNIX





# Ce qu'on a vu concernant Linux

- Le **noyau** Linux a été initié par **Linus Torvald**.
- Les OS Linux (**distribution**) sont des OS **UNIX-like**. Ils partagent donc énormément de concepts.
- On passe par la ligne de commande, non pas qu'Unix/Linux n'ait pas d'interface graphique, mais car pour certaines tâches, l'**utilisation de la ligne de commande s'avère bien plus pratique et plus puissante que la souris**.
- Les fonctionnalités peuvent varier, notamment en fonction du Shell utilisé.
- En TD, vous avez appris à interagir avec Linux (distribution Ubuntu) via **la ligne de commandes** (shell Bash). Vous vous êtes connecté, avez navigué dans votre espace de travail, joué avec les fichiers, utilisé un éditeur de texte « archaïque » mais puissant : **vi(m)**.  
Commandes : **cd**, **ls**, **cat**, **grep**, ...



## « ça marche pô »

- Vérifiez qu'il n'y a pas une faute de frappe (/!\ les commandes sont sensibles à la case).
- Vérifiez que vous avez les bons arguments, le bon nombre
- Regardez les indications du terminal (le terminal est votre ami)
- Préparez-vous à poser votre question à son.a voisin.e / le.a prof / internet :
  - Qu'est ce que je voulais faire ?
  - Quelles sont mes entrées ?
  - Quelles sont les sorties attendues ?



# Objectif pour le cours

À retenir :

- Le nom des commandes qu'on vous présente
- À quoi elles servent, indépendamment de leurs arguments (= le synopsis du **man**, avec vos propres mots)
- Savoir à quoi servent les principaux arguments (ceux qu'on vous présente).

Ce qui n'est pas demandé :

- Réciter le manuel

**Astuces** : se concentrer en cours et en TD (apprentissage par exposition, répétition) et se faire des petites fiches (moyens mnémotechniques).

# Le prompt

- C'est ce qui s'affiche par défaut dans votre terminal / console une fois connecté.e.
- En général : « username@machinename \$ »
- Configurable, via la variable **PS1** (Prompt String)

```
nanis@jammy:[~] $ echo $PS1
\[\\e[34m\]nanis\[\\e[m\]@\[\\e[44m\]jammy\[\\e[m\]\[\w\] $
nanis@jammy:[~] $ PS1=toto
toto
toto
toto
toto
toto
totoecho $PS1
toto
totoPS1="`parse_git_branch`\[\\e[34m\]nanis\[\\e[m\]@\[\\e[44m\]jammy\[\\e[m\]\[\w\] $ "
nanis@jammy:[~] $ █
```

# Le prompt – exemples post configuration

```
(base) (jammy-base r5828)nanis@C302L-G24P07:~$ ls
baseconfig          Desktop   miniconda3  ownCloud
baseconfig-gitlab   Documents  Modèles    Public
bluetooth-pods.sh  Images     Musique    signal-desktop-keyring.gpg
(base) (jammy-base r5828)nanis@C302L-G24P07:~$ pwd
/home/nanis
(base) (jammy-base r5828)root@C302L-G24P07:/home/vaginay241 2024-09-18 19:50 (0) : (jammy-base r5853)
root@C302L-G24P07:/home/vaginay241 2024-09-18 19:50 (0) : (jammy-base r5853)
toto
bash: toto : commande introuvable
root@C302L-G24P07:/home/vaginay241 2024-09-18 19:50 (0) :( 127 (jammy-base r5853)

lin@Demeter ~ ~/Software/munin-pihole-plugins on 🖥 master ━ ✓ ⚡ 2.50 🔍 1 ↴ at 19:20:29 ⏱
```

```
nanis@jammy[~] $ ls
AV_sugg
baseconfig@
baseconfig-gitlab/
bluetooth-pods.sh@
nanis@jammy[~] $
```

```
21:22:57 79% chris@cruachan ~
$ dossh
chris@192.168.0.18's password:
Welcome to Arch Linux ARM

Website: http://archlinuxarm.org
Forum: http://archlinuxarm.org/forum
IRC: #archlinux-arm on irc.Freenode.net
Last login: Tue Sep 18 18:06:59 2018 from 192.168.0.1
21:23:05 chris@alarmpi 192.168.0.18 ~
$
```

# Syntaxe générale d'une commande

- Une commande s'appelle directement par son **nom**
- On peut spécifier un ou plusieurs **arguments** à une commande :
  - directement une valeur (**paramètre de commande**),
  - un **nom d'option** (« `--option` » ou « `-o` »), suivi si besoin d'une valeur (**paramètre d'option**).
- Exemples :

`ls` # juste une commande, son nom est « `ls` ».

`ls .` # 1 argument qui est un paramètre de commande

`ls -a .` # 2 arguments : une option et un paramètre

`ls -l` # juste une option

`ls -la` # 1 argument correspondant à 2 options combinées (oui, des fois c'est possible)

`ls --width 5 dossier/` # 3 args (une option, un param d'option, un param de commande)

`ls -w 5 dossier/` # idem, mais option en version courte

# Cas d'erreurs d'une commande

- La commande n'**existe pas**. ATTENTION à la casse et aux espaces.

```
nanis@jammy:~$ jfidosfhisdof
```

```
bash: jfidosfhisdof : commande introuvable
```

- Vous n'avez **pas le droit** d'exécuter cette commande.

```
nanis@jammy:~$ cd /root/home
```

```
bash: cd: /root/home: Permission non accordée
```

- Les **arguments** (options et/ou paramètres) de la commande sont **erronées**.

```
nanis@jammy:~$ mv file
```

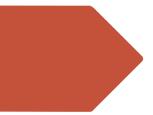
```
mv: opérande de fichier cible manquant après 'file'
```

```
Saisissez « mv --help » pour plus d'informations.
```

```
nanis@jammy:~$ mv f1 f2
```

```
mv: impossible d'évaluer 'f1': Aucun fichier ou dossier de ce nom
```

Que faire ? **Lire** le retour de la commande, **chercher** des explications dans l'aide (**man**), **demandez**.



# L'aide

**man** : le manuel de référence

Syntaxe de base : **man [option] [[section] page]**

Arguments :

- **option** : Facultatif
- **section** : Facultatif – Précise le n° de la section du manuel
- **page** : Page dont on souhaite consulter le manuel

- La documentation d'une commande indique sa **syntaxe générale**, ainsi que le **détail des arguments** possible : **commande [option ...] [paramètres ...]**
- Dans la syntaxe générale, si un argument est indiqué entre [ ], alors il est facultatif.



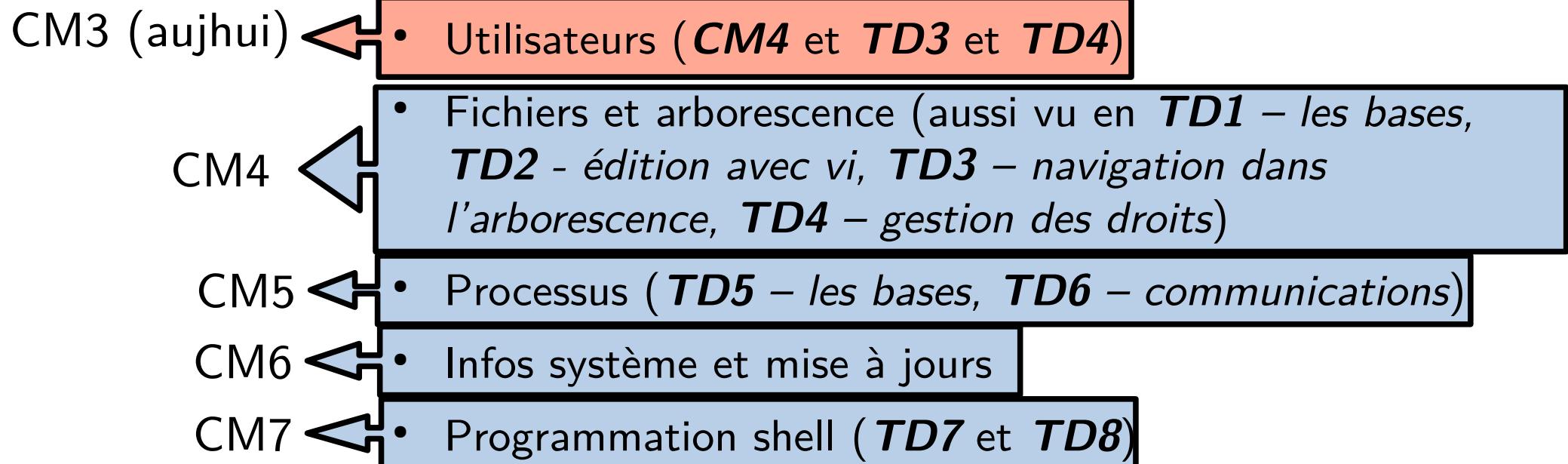
# Commandes utiles du terminal : man

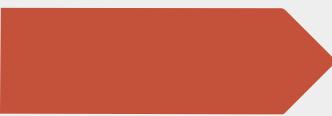
- Affichage toutes les pages **passwd** (de toutes les sections) du manuel:  
prompt> **man -a passwd**
- Affichage de la page **passwd** dans la section 5 du manuel :  
prompt> **man 5 passwd**
- Trouver toutes les rubriques contenant un mot clé donné :  
prompt> **man -k mot-clé-donné**
- Description succincte d'une commande :  
prompt> **man -f passwd**
- Plus d'info sur man :  
prompt> **man man**

# Les sections d'aide dans le man

Section	Type de commandes
1	commandes et applications utilisateur ( <a href="#">ls</a> , <a href="#">cd</a> , <a href="#">grep</a> , ..., <a href="#">passwd</a> )
2	appels système, codes erreurs noyau ( <a href="#">open</a> , <a href="#">read</a> , <a href="#">write</a> )
3	fonctions des bibliothèques (bib standard C : <a href="#">printf</a> , <a href="#">malloc</a> )
4	fichiers spéciaux : pilotes de périphériques et protocoles réseau ( <a href="#">/dev/null</a> )
5	formats de fichiers standard et autre conventions ( <a href="#">/etc/passwd</a> )
6	jeux
7	divers fichiers et documents
8	commandes d'administration système ( <a href="#">iptables</a> , <a href="#">mount</a> , <a href="#">apt-get</a> )
9	divers specs noyau et interfaces (non standard)

# Les notions et commandes qu'on va (re)voir





# **Linux : les utilisateurs**



# Vocabulaire

Linux est un **système multiutilisateur**

Toute entité (personne physique ou programme particulier) devant interagir avec un système Linux est authentifiée sur cet ordinateur par un **utilisateur.ice**.

Chaque utilisateur.ice est identifié par un nom unique (**login**) et un numéro unique : le **UID** – user ID).

Chaque utilisateur.ice fait partie d'**un ou plusieurs groupes** (par défaut un groupe du même nom).

Chaque groupe est également identifié par un numéro unique : le **GID** – group id. Les groupes servent à identifier des utilisateurs qui ont des caractéristiques communes (prof, étudiants, invités) et permettent une gestion efficace.

Il existe un compte utilisateur particulier qui **dispose de tous les droits** : **root** ou « **super utilisateur** », ou « **compte administrateur** » (d'UID 0).

# Connexion dans un terminal

**Compte** = nom de connexion + mot de passe.

Login : Jean

Password : \*\*\*\*

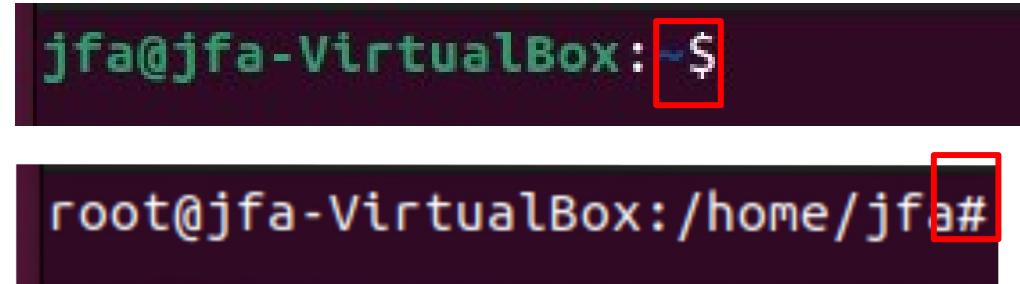
Bienvenue sur ...

Prompt>

« \$ » (ou « % », ou « # » - généralement pour root) est le **prompt** ou l'**invite** de l'**interprète de commande** utilisé (shell).

L'interprète attend que l'utilisateur tape une commande, exécute cette commande, réaffiche la chaîne d'invite, et attend une nouvelle commande, jusqu'à ce qu'on quitte...

**exit** ou **^D (CTRL-D)** en début de ligne ⇒ déconnexion (fin du shell)



```
jfa@jfa-VirtualBox:~$  
root@jfa-VirtualBox:/home/jfa#
```



# Où suis-je connecté.e ?

**tty** : renvoie le nom du terminal sur lequel on est connecté (en soit c'est juste un nom de fichier ; en UNIX tout est fichier).

Exemple :

```
nanis@jammy[~] $ tty  
/dev/pts/0
```

<https://www.geeksforgeeks.org/tty-command-in-linux-with-examples/>

<https://www.malekal.com/quest-ce-que-tty-comment-utiliser-commande-tty-sur-linux/>



# Qui d'autre est connecté ?

**who** : liste des utilisateurs connectés,

Exemple d'utilisation :

```
nanis@jammy:~$ who
vaginay241 :1          2024-09-15 12:34 (:1)
vaginay241 tty3        2024-09-18 16:02
```

<https://www.geeksforgeeks.org/who-command-in-linux/>

# Informations sur les utilisateur.ices

**/etc/passwd** est un fichier statégique qui rassemble des **infos sur toutes les entités** ayant un compte sur le système (user physique ou certains programmes spécifiques).

Exemple :

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

Une ligne pour chaque entités, 7 champs par ligne (séparés par le caractère « : »)

- **Nom** : login
- **Mot de passe** : soit rempli et crypté (sur 13 caractères) par le programme **passwd**, soit contient « x » avec un mot de passe crypté et déporté dans un autre fichier, accessible par l'administrateur.ice.
- **UID [= User Identification]** : numéro d'identification de l'utilisateur.ice (0 pour root).
- **GID [= Group Identification]** : numéro d'identification du groupe primaire de l'utilisateur.ice.
- **Commentaire** : champ facultatif.
- **Répertoire HOME** : répertoire d'accueil lors de la connexion de l'utilisateur.
- **Commande** : commande lancée au moment de la connexion.

# Informations sur les utilisateur.ices et groupes

Synthaxe : **id [user ...]** : Affiche l'UID, le GID et la liste des groupes d'un utilisateur

Paramètres : **user** : Nom de l'utilisateur dont on souhaite afficher les informations.

Si non-spécifié : affiche les informations de l'utilisateur exécutant la commande.

Il est également de spécifier plusieurs utilisateurs (séparés par des espaces).

Synthaxe : **groups [user ...]** : Affiche la liste des groupes d'un utilisateur

Paramètre : **user** : Nom de l'utilisateur dont on souhaite afficher les informations.

Si non-spécifié, affiche les informations de l'utilisateur exécutant la commande.

Il est possible de spécifier plusieurs utilisateurs (séparés par des espaces).

# Liste des groupes

L'ensemble des groupes est listé dans le fichier **/etc/group**

Exemple :

root:x:0:root

daemon:x:1:daemon

Les champs, dans l'ordre :

- nom du groupe,
- mot de passe du groupe,
- GID,
- entités du groupe (séparés par une virgule)

Lister uniquement les noms des groupes : **compgen -g**

# Gestion des utilisateur.ices et groupes

Ces opérations (création, suppression modification) nécessitent d'être **root** (administrateur).

	utilisateur	groupe
Ajouter	<b>adduser / useradd</b>	<b>addgroup / groupadd</b>
Supprimer	<b>deluser</b>	<b>groupdel</b>
Modifier	<b>usermod</b>	<b>groupmod</b>

Exemple :

ajouter un.e utilisateur.ice existant à un groupe existant : **adduser username groupname**

[https://doc.ubuntu-fr.org/tutoriel/gestion\\_utilisateurs\\_et\\_groupes\\_en\\_ligne\\_de\\_commande](https://doc.ubuntu-fr.org/tutoriel/gestion_utilisateurs_et_groupes_en_ligne_de_commande)

# Gestion des utilisateur.ices et groupes

- **groupadd *groupname*** : Ajoute un groupe d'utilisateurs.

Exemple :

```
prompt> groupadd but1a
```

- **groupdel *groupname*** : Détruit un groupe d'utilisateurs.

Exemple :

```
prompt> groupdel but1a
```

- **useradd *username*** : Ajoute un utilisateur

Exemple :

```
prompt> useradd jfa
```

- **usermod *username*** : Modifie les paramètres d'un compte utilisateur

Exemple :

```
prompt> usermod -c "This is the best user" jfa
```

```
prompt> usermod -d /home/jfahome jfa
```

```
prompt> usermod -e 2020-05-29 jfa
```

<https://www.geeksforgeeks.org/usermod-command-in-linux-with-examples/>

# Modifier son mot de passe

```
prompt> passwd
```

Changing password for Jean

Old password : \*\*\*\*\*

New password : \*\*\*\*\*

Re-enter new passwd : \*\*\*\*\*

Un mdp :

- Doit être difficile à trouver !  
(mélanger des chiffres, des lettres, des majuscules/minuscules, mettre des caractères spéciaux, et qu'il ait suffisamment de symboles (plus de 8))
- Ne doit pas être partagé

Mdp oublié ?

- Sous linux si vous avez oublié **votre** mot de passe utilisateur, l'administrateur (root) peut vous le changer.
- Si c'est le mot de passe de **root** que vous avez oublié, tout est perdu ! **Il faut réinstaller le système !**

# A propos de l'user root, de sudo et de su

Rappel : **root** est un utilisateur particulier qui a **tous les droits**. Certaines commandes nécessitent d'avoir les droit root. Quand on est un utilisateur autre que root et qu'on veut administrer, il faut « prendre les droits root ».

Souvent, l'utilisateur root n'a **pas de mot de passe dédié**, et on ne peut pas se connecter en tant que root (c'est sûre ! :)))

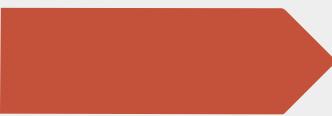
Si on en a le droit (liste de sudoers), on peut lancer une commande avec les droit administrateurs :  
**sudo commande** (demande \***notre**\* mdp utilisateur et lance **commande** en se faisant passer pour **root**)

Exemple :

```
$ cat /etc/group | grep sudo  
sudo:x:27:Tom,nanis,vaginay241  
$ whoami  
nanis  
$ sudo whoami  
root
```

Il faut (généralement) mettre **sudo** à chaque ligne. Si on a la flemme : **sudo bash** ouvre une session en tant que root (changement de prompt), et on peut lancer des commandes sans **sudo** alors qu'on aurait dû en temps normal). Exemple : **wc /etc/shadow**

Si on assigne un mdp à l'utilisateur root, alors on peut se connecter en tant que root (via la commande **su**). Mais c'est moins sûre, vu que tous les utilisateurs qui vont utiliser **su** doivent connaître le mdp de root (très mauvaise pratique de partager un mdp /!\).



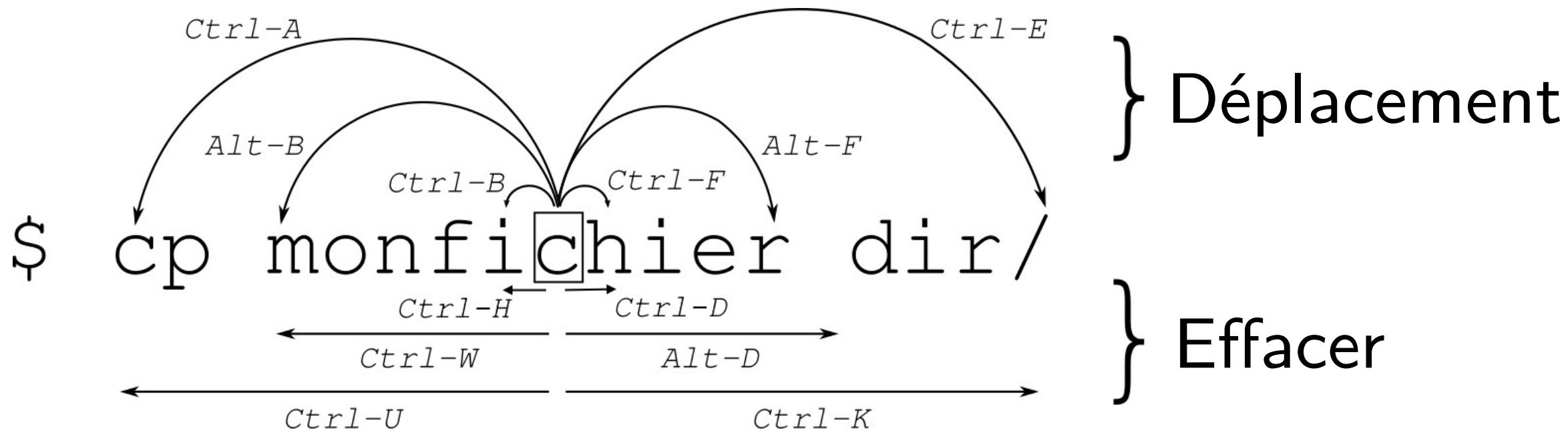
## **Bonus : protips sur l'utilisation du terminal**

# Utilisation du terminal

Bash (et d'autres CLI) utilise l'outil **readline** pour interagir via la ligne de commandes.  
À ce titre, on a accès à quelques raccourcis claviers :

- **tab** : autocompletion
- **^H** ou **DEL** : annule le dernier caractère frappé,
- **^L** : vide la fenêtre du terminal (commande **clear**)
- **^U** : annule tous les caractères de la ligne,
- **^C** : interrompt la commande en cours
- **^D** : simule une fin de fichier (eof)
- **↑** : parcourt l'historique de commandes
- **^R** : rechercher dans l'historique de commandes
- **^Z** : suspend l'application (retour à Bash ; faire **fg** pour rendre la main à la commande suspendue)

# Utilisation du terminal



Raccourcis inspirés de l'éditeur de texte **Emacs**, mais un mode **vi(m)** est possible ;)  
**set -o vi** pour essayer.

Plus de commandes : <https://ss64.com/bash/syntax-keyboard.html>



**Teaser TD3** : arborescence du système de fichiers Linux (+ un peu de gestion d'utilisateurs et groupes)





Aujourd'hui : système de fichier

- Définition de l'arborescence
- Sur le disque
- Racine UNIX
- Commandes associées



# Système de fichiers

Ce qui fait correspondre comment les fichiers sont physiquement stockés sur les **disques** (via des séquences de 0 et de 1) et ce avec quoi l'utilisateur interagit (une **arborescence**)

Selon le contexte :

- **organisation hiérarchique des fichiers au sein d'un système d'exploitation**
- **organisation des fichiers sur un médium** (ext[2-4], NTFS, FAT, FAT32, ...)

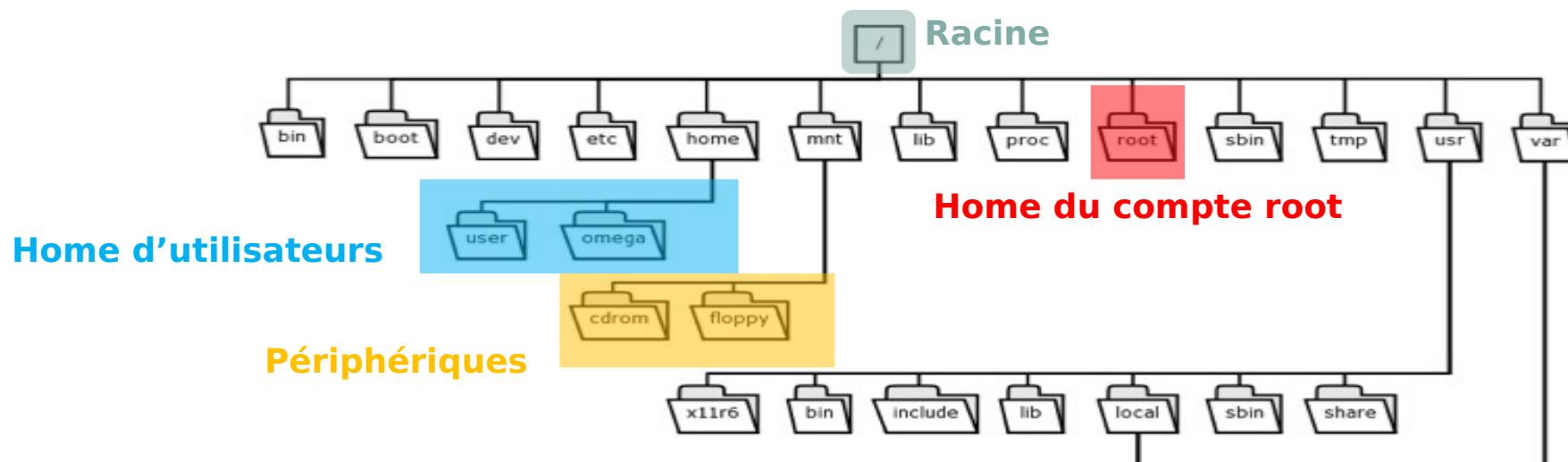


# Système de fichiers UNIX : L'arborescence

# Arborescence UNIX

Les **fichiers** (fichiers systèmes et fichiers utilisateur) sont **tous** rangés dans des **répertoires** organisés dans **une** structure hiérarchique : l'**arborescence**

- **Root** (« racine » en français, notée « `/` ») = le répertoire initial  
/!\ à ne pas confondre avec le **compte root**, ni avec le « `/` » utilisé dans les **chemins**.
- **Home** (« maison » en français) = un dossier réservé à chaque compte du système.
- On peut « **monter** » des périphériques extérieurs (clés USB, DDE, ...) dans l'arborescence (`/mnt`, `/media`)  
→ pas de disques différents comme sur Windows (A:, C:, D:...).



# Chemins UNIX

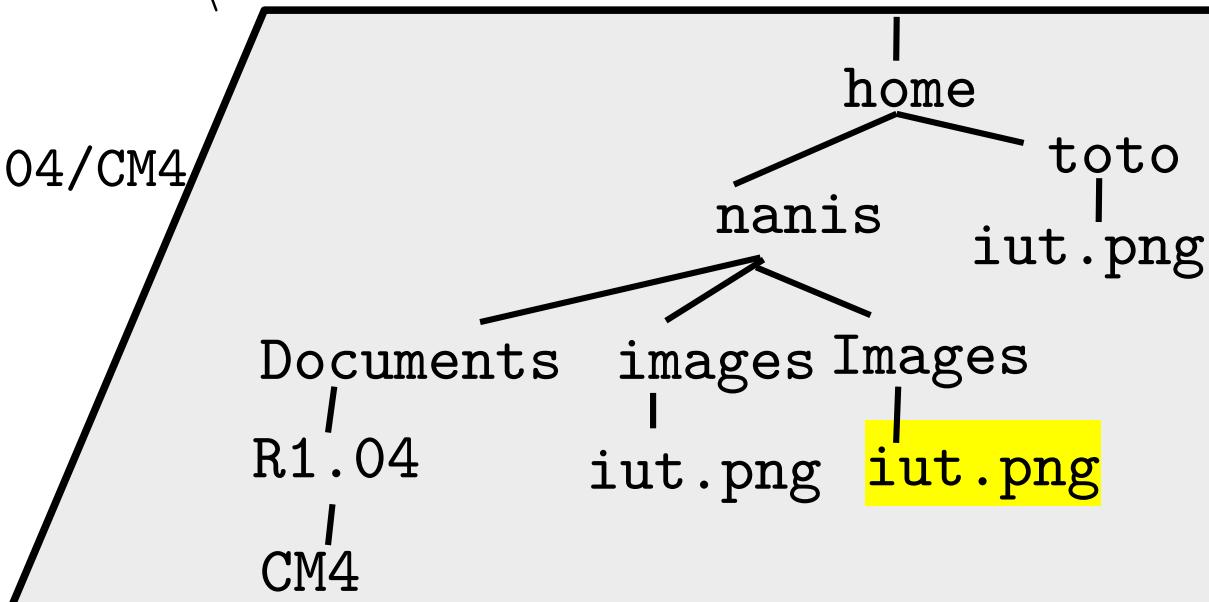
Un chemin est **chaîne de caractère qui décrit l'emplacement d'un fichier**.

- **absolu** : défini depuis la racine du système de fichiers « / » (ou home « ~ »)
- **relatif** : défini à partir de notre localisation en utilisant « . » (répertoire courant) et « .. » (répertoire parent).

/!\ le séparateur de chemin est « / » sous UNIX mais « \ » sous Windows.

Exemple : on est dans /home/nanis/Documents/R104/CM4  
et on veut faire référence au fichier **iut.png**.

Wooclap



# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code  
d'événement dans le  
bandeau supérieur



Activer les réponses par SMS

Code d'événement  
**XZBBRK**

# Chemins UNIX

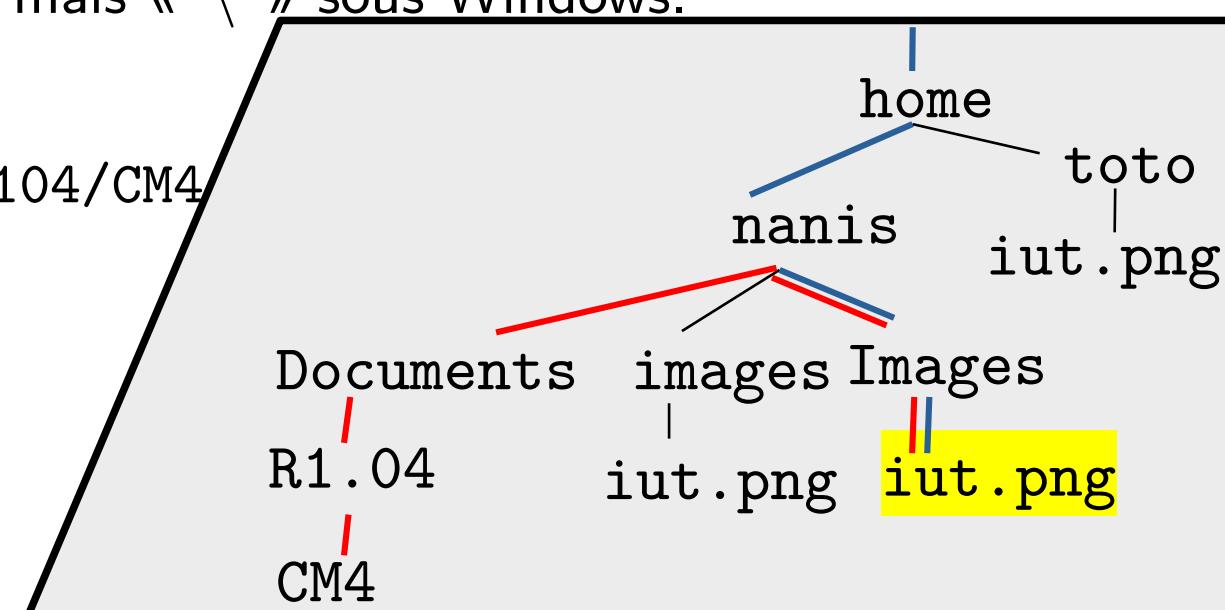
Un chemin est **chaîne de caractère qui décrit l'emplacement d'un fichier.**

- **absolu** : défini depuis la racine du système de fichiers « / » (ou home « ~ »)
- **relatif** : défini à partir de notre localisation en utilisant « . » (répertoire courant) et « .. » (répertoire parent).

/!\ le séparateur de chemin est « / » sous UNIX mais « \ » sous Windows.

Exemple : on est dans /home/nanis/Documents/R104/CM4  
et on veut faire référence au fichier **iut.png**.

- /home/nanis/Images/iut.png (**absolu**)
- ../../../../Images/iut.png (**relatif**)



Note : Toutes les commandes UNIX qui manipulent les fichiers acceptent toutes les formes de chemins  
<https://doc.ubuntu-fr.org/chemins>

# Noms de fichiers

- Idéalement, composés exclusivement de caractères alphanumériques (accents possible), du tiret « - », du souligné « \_ », et du point « . ».
- Ils sont sensibles à la casse : `toto.txt`, `TOTO.txt`, `toto.TXT`
- Certains caractères nécessitent d'être « échappés » quand on les utilise dans un nom de fichier en ligne de commande. On les fait précéder de « \ ». : « \\$ », « \\ »
- Les caractères interdits ou à éviter sont :
  - « / » : **Interdit**, sert à séparer les noms de répertoire dans un chemin
  - « \ » : À **échapper**, et peut poser des problèmes de compatibilité avec d'autres systèmes d'exploitation comme Windows. À éviter.
  - « - » : À **éviter** en début de nom (le shell va l'interpréter comme une option de commande... )
  - « \* », « ? », « : », « ' », « " », « # », « \$ », « ; », « ! », « & », « | », parenthèses « () », accolades « {} », chevrons « <> », crochets « [] » : **À éviter**.
  - Espace : À **échapper**.
  - « . » en début de nom indique que le fichier est caché.

# Noms génériques de fichiers

Un **nom générique** est un nom qui contient un ou plusieurs caractères spéciaux (**méta-caractères ou joker**). Il permet de désigner un ensemble d'objets.

- « ? » signifie n'importe quel caractère

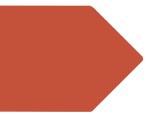
Exemple : « t?t? » correspond entre autre à « toto » qu'à « titi » mais pas à « tooto ».

- « \* » signifie n'importe quelle chaîne de caractères (y compris vide).

Exemple :

« bon\*.txt » peut correspondre entre autre aux noms de fichiers suivants : « bonjour.txt », « bonsoir.txt », « bon\_à\_rien.txt », « bon\_j\_ai\_pas\_d'autre\_idée\_mais\_vous\_avez\_compris\_hein.txt », « bon.txt », mais pas à « bonjour », « bon », ni « jour.txt ».

- **Exception** : Le caractère « . » ne peut pas être remplacé par un joker s'il est au début d'un nom de fichier.  
Conséquence directe : une chaîne avec joker porte soit sur les fichiers non-cachés, soit sur les fichiers cachés, mais pas les deux en même temps.



# Noms génériques de fichiers

- **[ ]** signifient un caractère appartenant à un ensemble de valeurs décrites dans les crochets, si les fichiers résultants existent  
 $t[oi]to \rightarrow toto, tito$ 
  - utilisé avec les crochets permet de définir un intervalle, plutôt qu'un ensemble de valeurs.  
 $t[a-d]to \rightarrow tato, tbto, tcto, tdto$   
! ou  $\wedge$  utilisé entre crochets en première position, signifie tout caractère excepté ceux spécifiés entre crochets.  
 $t[!o]to \rightarrow pas\ toto$
- **{}** signifient un caractère appartenant à un ensemble de valeurs décrites dans les crochets, même si les fichiers résultants n'existent pas.  
 $t\{o,i\}to \rightarrow toto, tito$ 
  - .. utilisé avec les accolades, permet de définir un intervalle, plutôt qu'un ensemble de valeurs.  
 $t\{a..d\}to \rightarrow tato, tbto, tcto, tdto$



# Noms génériques de fichiers – Exemple

```
prompt> ls test*
```

```
prompt> touch test_[ab].txt  
prompt> ls test*
```

```
prompt> touch test_{a..j}.txt  
prompt> ls test*
```

```
test_a.txt  test_c.txt  test_e.txt  test_g.txt  test_i.txt  
test_b.txt  test_d.txt  test_f.txt  test_h.txt  test_j.txt
```

```
prompt> ls test_[ab].txt  
test_a.txt  test_b.txt
```

# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code  
d'événement dans le  
bandeau supérieur



Activer les réponses par SMS

Code d'événement  
**XZBBRK**



# Noms génériques de fichiers – Exemples

Tous les fichiers dont le nom...

**f\*** commence par 'f'.

**f?** commence par 'f', suivi d'un seul caractère quelconque.

**f[12xy]** commence par 'f', suivi d'un caractère à choisir parmi '1', '2', 'x' ou 'y'.

**f[a-z]** commence par 'f', suivi d'un caractère dont le code ASCII est compris entre le code 'a' et le code 'z', donc une lettre minuscule.

**\*.c** fini par .c (= dont l'extension est .c)

**?.**c est formé d'un caractère quelconque, suivi de '.c'

**??** est formé de deux caractères.

**\*.[A-Za-z]** se termine par un '.' suivi d'une seule lettre majuscule ou minuscule.

**\*.[ch0-9]** se termine par un '.' suivi d'un seul caractère à choisir parmi 'c', 'h', ou un chiffre entre '0' et '9'.

**[!f]\*** ne commence pas par 'f'

**\*[!0-9]** ne se termine pas par un chiffre.

# Noms génériques de fichiers – les classes

La **norme POSIX** définit des classes (groupes) de caractère suivantes :

`[:upper:]` pour les majuscules

`[:lower:]` pour les minuscules

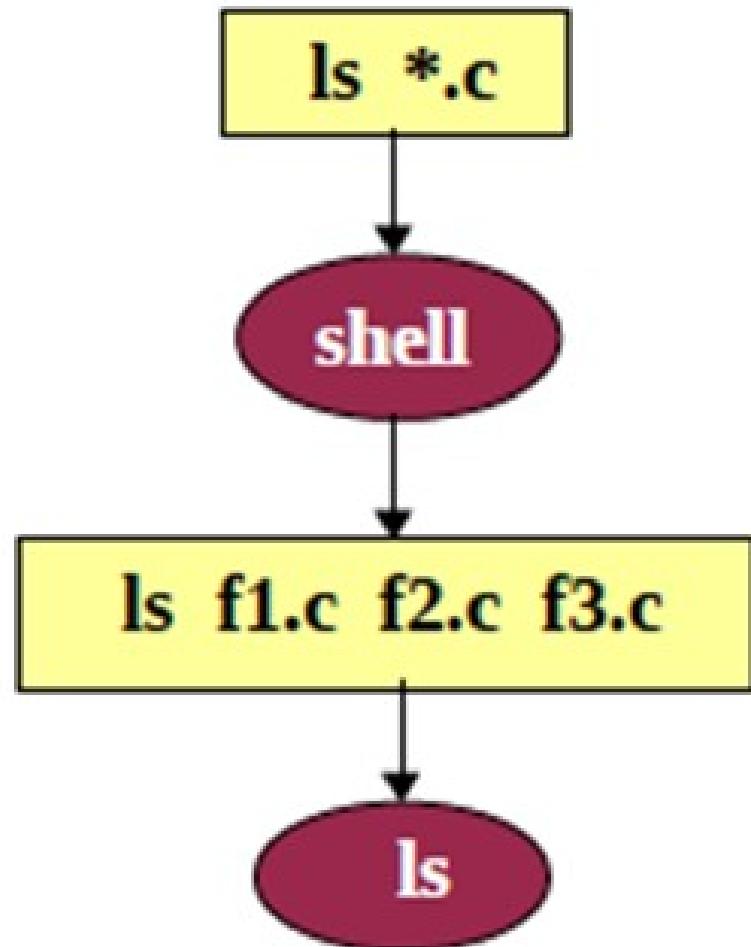
`[:digit:]` pour les chiffres de 0 à 9

`[:alnum:]` pour les caractères alphanumériques

Exemple :

`ls ./[:upper:]*` → tous les noms de fichiers qui commencent par une lettre majuscule.

# Noms génériques de fichiers – substitution



Le traitement des méta-caractères est indépendant de la commande. Il est effectué par le Shell, avant l'exécution de la commande.

Chaque ligne est donc analysée deux fois :

**1) Par le Shell :**

si la ligne contient un joker, le shell remplace le motif par les valeurs possibles.

Exemple : le shell lit `ls f*.c`. Il enlève « `f*.c` » et le remplace par « `f1.c f2.c f3.c` ». Le résultat est donc `ls f1.c f2.c f3.c`.

**2) Par la commande :** lorsque la commande `ls` est exécutée, elle reçoit « `f1.c f2.c f15.c` ». la commande va donc travailler sur les 3 fichiers.



# **Le système de fichier : correspondance avec le disque**



# Différents systèmes de fichiers

Comment les fichiers sont stockés sur les disques

- Sous windows : FAT, FAT32, NTFS
- Sous Linux : ext2, ext3

Implémentations et caractéristiques différentes (tailles max d'un fichier, gestion des droits, etc).

# Système de fichiers : généralités

Dans un **flat file system** (« plat ») :

- Pas de hiérarchie, tous les fichiers sont à la racine, et rangés à la queue leu sur le disque (arg si changement de taille...).
- un fichier **directory file (DF)** stocke l'adresse où lire les données d'un fichier (~ sommaire d'un livre)



Dans un système de fichiers moderne :

- Les fichiers sont mis dans des **blocs** de taille standardisée (ça laisse un peu de marge : **slack space**). Ils sont mis dans plusieurs block s'ils sont plus gros qu'un bloc (fragmentation).
- S'il y a niveaux de dossiers, chaque dossier a un **directory file** qui stockent le bloc (ou plutôt une liste de blocs) où est rangés chacun de ses éléments (fichier ou dossier).

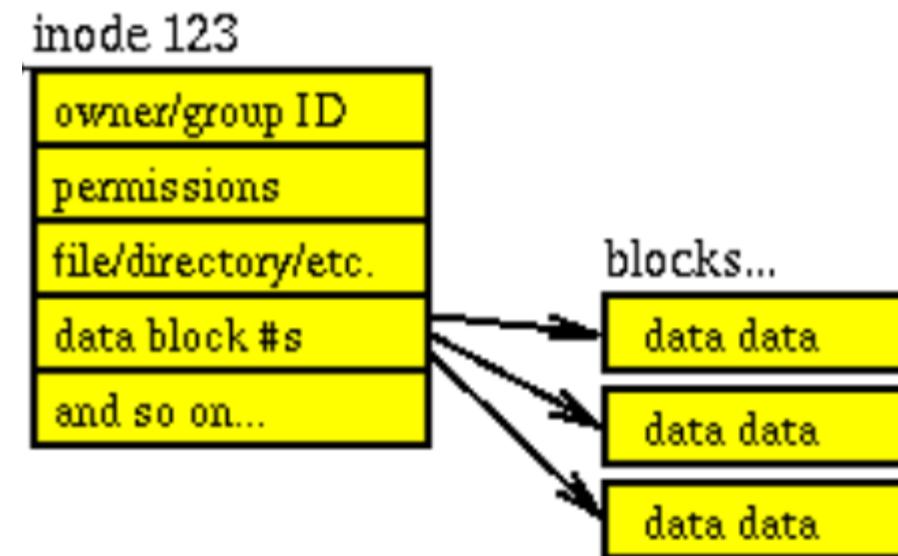


# Système de fichiers : inodes de fichiers

Structure de données qui contient les **métadatas** d'un fichier :

- un identifiant (**inode number**),
- la taille du fichier,
- l'identifiant du périphérique contenant le fichier,
- l'identifiant du propriétaire du fichier, et du groupe,
- le mode du fichier (ses droits d'accès),
- l'horodatage du fichier (date de modification de l'inode, du fichier et de dernier accès)
- l'adresse des datas du fichiers

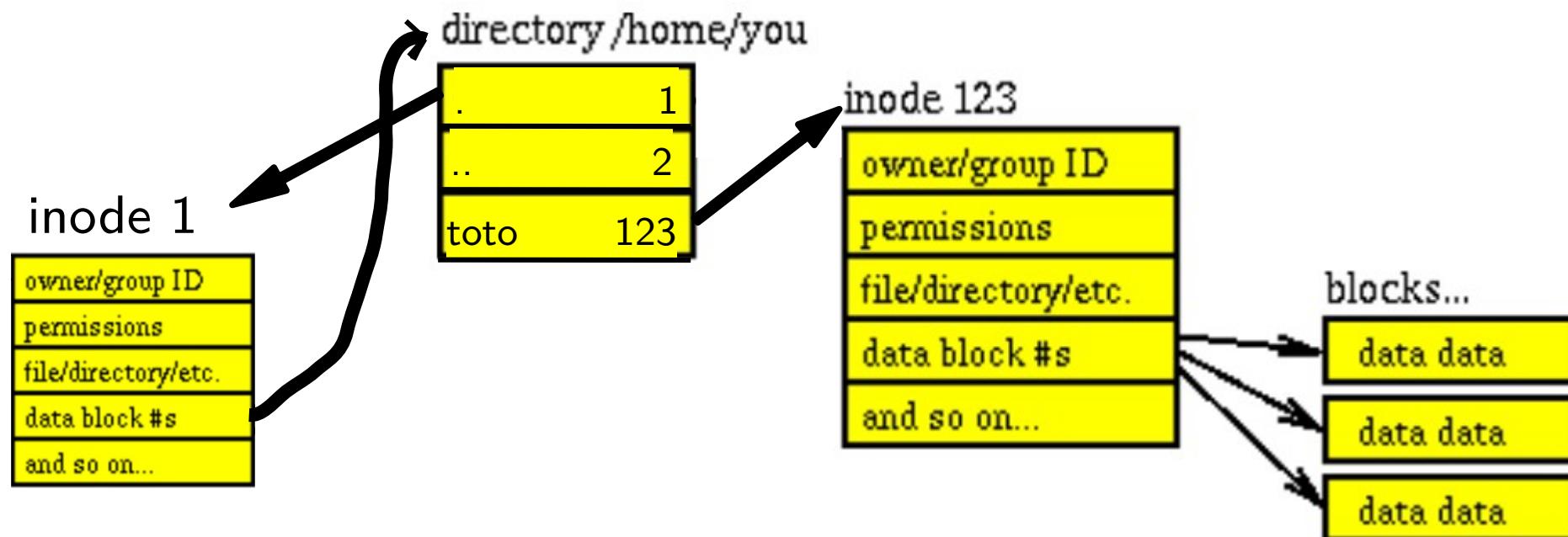
Attention ! L'inode d'un fichier ne contient **pas son nom** ! (Un fichier peut en effet être présent à plusieurs endroits dans l'arborescence de fichiers, sous différents noms ; on en reparle plus tard dans les slides sur les liens physiques et symboliques, )



Similaires aux **tables d'allocation de fichiers** dans les systèmes de type FAT (Windows)

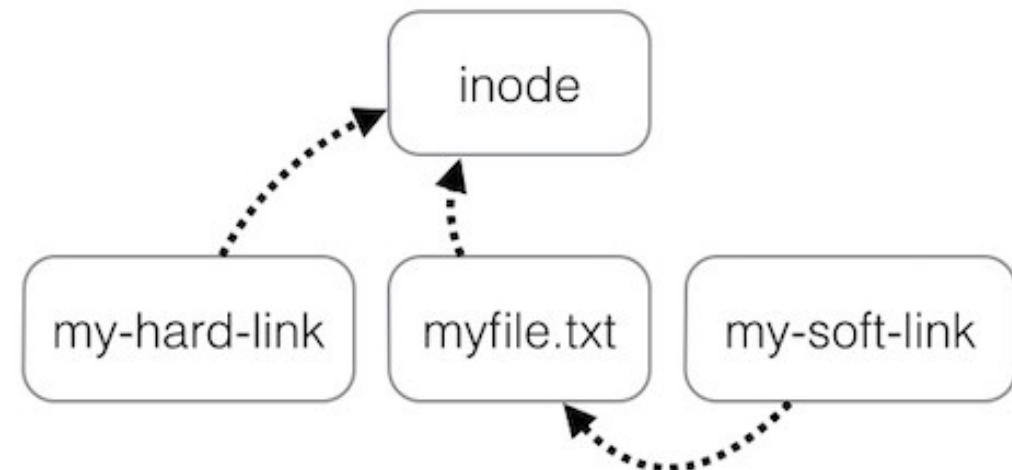
# Système de fichiers : les répertoires

Un répertoire est comme un fichier ordinaire : il possède un inode (contenant les métadatas), et ses données (table de liens [chaîne de caractère, i-nombre]) sous forme de suite d'octets.



# Système de fichiers : les liens

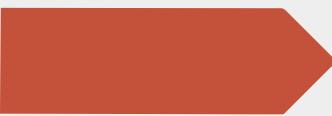
- Lien dur « hard link » : un fichier qui pointe **vers** le même **inode** qu'un autre fichier.
- Lien symbolique « soft link » : un fichier qui pointe **vers un chemin donné**.



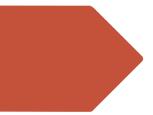
# Différentes catégories de fichiers

En UNIX, « **tout est fichier** »

- fichiers **normaux** : documents odp, sources des programmes, fichier textes de configuration, images, archives zip, **exécutables** (programmes en code binaire).
- fichiers **répertoires** (les **dossiers**), qui peuvent contenir d'autres fichiers.
- fichiers **liens symboliques**
- fichiers **spéciaux**
  - dans `/dev`, le système d'exploitation prépare des canaux de communication avec les périphériques. Démo : `tty` ; `echo toto > /dev/pts/0`



# **Système de fichiers : La racine des UNIX**



# L'organisation à la racine

```
cd / ; ls
```

**Standard** Filesystem Hierarchy Standard – FHS maintenu par la fondation Linux.

[https://refspecs.linuxfoundation.org/FHS\\_3.0/fhs-3.0.pdf](https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf)

- A changé au fil des années
- Pas forcément respecté par toutes les distribs

# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

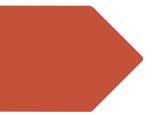
2

Entrez le code  
d'événement dans le  
bandeau supérieur



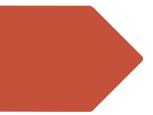
Activer les réponses par SMS

Code d'événement  
**XZBBRK**



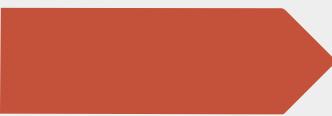
# L'organisation à la racine

- **/bin** : [binaries] les exécutables
- **/boot**: le noyau et les fichiers de démarrage
- **/dev** : [device file] le répertoire des fichiers spéciaux pour communiquer avec les périphériques (« tout est fichier », même le hardware : disque dev/disk/sda, webcam, clavier...)
- **/etc** : [etc -- D. Ritchie / Edit To Configure] les fichiers de config au niveau du système (« system-wide »)
  - **/etc/rc.d** scripts de démarrage du système
  - **/etc/cron** description des tâches périodiques à effectuer
- **/home** : la racine des répertoires personnels des utilisateurs
- **/lib** [library] les bibliothèques et les modules du noyau, équivalent des DLL de Windows
- **/mnt** [mount] et **/media** : la racine des points de montage des systèmes de fichiers périphériques ou extérieurs (cd, clé-usb, ...).



# L'organisation à la racine

- **/opt** [optional] : lieu d'installation d'applications supplémentaires (Chrome, Signal, OwnCloud, ...)
- **/proc** [processus] : contient des infos sur l'état du système et les différents processus en fonction.
- **/root** : répertoire personnel du super-utilisateur root
- **/sbin** : [system binaries] les exécutables pour l'administration du système (commandes de démarrage et d'arrêt du système, ...)
- **/tmp** [temporary] : les fichiers temporaires
- **/usr** : programmes accessibles à tout utilisateur; sa structure reproduit celle de la racine /
- **/var** [variable] : contient des fichiers dont on s'attend qu'ils grossissent en taille. (**var/crash**, **/var/mail**, **/var/log**, file d'impression dans **/var/spool/lpd**)



# **Système de fichiers : Les commandes associées**



# Commandes de base sur les fichiers et répertoires

- **ls, cat, cp, mv, rm, touch**
- **pwd, cd, mkdir, rmdir, rm -r**

# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code  
d'événement dans le  
bandeau supérieur



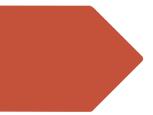
Activer les réponses par SMS

Code d'événement  
**XZBBRK**



# Commandes de base sur les fichiers

- **ls *fichier* ... [list]** affiche le contenu des répertoires (à un niveau) et les noms des fichiers passés en arguments, ou s'il n'y a pas d'arguments, tous les fichiers du répertoire courant (« . ») sauf ceux commençant par un point.
- **cat *fichier* ... [concatenate]** affiche le contenu des fichiers donnés en arguments
- **cp *fichier1* *fichier2* [copy]** copie ***fichier1*** dans ***fichier2***
- **mv *fichier1* *fichier2* [move]** renomme ***fichier1*** en ***fichier2***
- **rm *fichier* [remove]** détruit le fichier ***fichier***



# Commandes de base sur les répertoires

- **pwd** [print working directory] affiche le chemin absolu du répertoire courant.
- **cd *répertoire*** [change directory] change de répertoire courant. Sans argument, rapatrie dans le répertoire de connexion (votre home).
- **mkdir *répertoire*** [make directory] crée un répertoire.
- **rmdir *répertoire*** [remove directory] détruit le répertoire s'il est vide et si ce n'est pas votre répertoire courant.
- **rm -r *répertoire*** [remove] détruit le répertoire récursivement (même non vide).
- **cp -r *répertoire*** [copy recursively]
- **mv -r *répertoire*** [move recursively]

# Commandes de base sur les répertoires - Exemples

- Créer un répertoire vide et le supprimer

```
prompt> mkdir rep
```

```
prompt> rmdir rep
```

```
prompt> rmdir rep
```

```
rmdir: impossible de supprimer 'rep': Aucun fichier ou dossier de ce nom
```

- Crée un répertoire non vide et le supprimer :

```
prompt> mkdir rep ; touch rep1/toto
```

```
prompt> rmdir rep1
```

```
rmdir : 'rep1' : Le répertoire n'est pas vide.
```

```
prompt> rm -r rep1
```

- Copier un répertoire ou un fichier

```
prompt> cp -r repertoire repertoire1
```

```
prompt> cp fichier fichier1
```

# Commandes de base sur les répertoires - Exemples

- Copier une arborescence

```
prompt> ls -R /tmp/r1  
/tmp/r1 :  
f1 f2 r11 r12  
/tmp/r1/r11 :  
f11  
/tmp/r1/r12 :  
f12  
prompt> cp -r /tmp/r1 .  
prompt> ls -R r1 r1 :  
f1 f2 r11 r12  
r1/r11 :  
f11 r1/r12 :  
f12
```

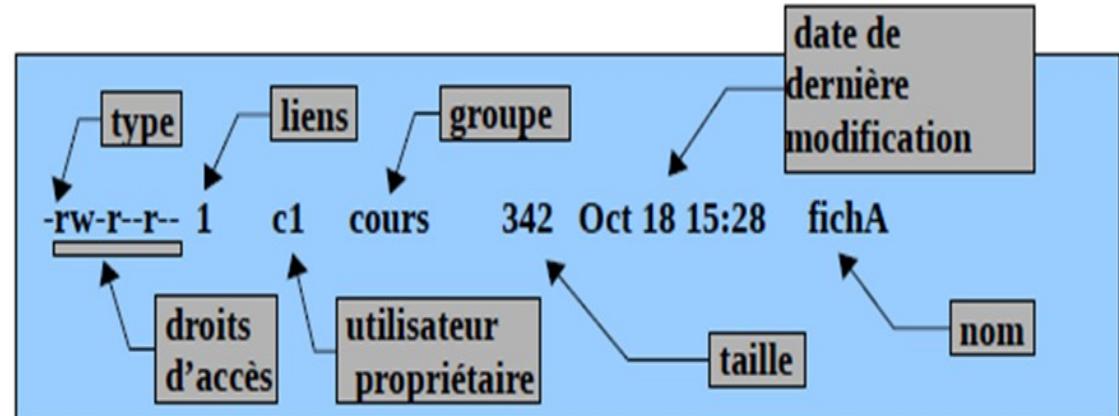
- Supprimer une arborescence : Attention il n'y a pas de demande de confirmation !

```
prompt> rm -rf r1  
prompt> ls r1
```

ls : r1 : Aucun fichier ou répertoire de ce type.

# La commande ls -l

ls -l (long) affiche de nombreuses informations sur le fichier :



- **Le type du fichier :**
  - 'd' : pour répertoire,
  - '-' pour fichier ordinaire,
  - 'b' pour périphérique bloc,
  - 'c' pour périphérique caractère,
  - 'l' lien symbolique,
  - 'p' tube nommé (IPC),
  - 's' socket locale (IPC).
- **Le nom de fichier :** Limité à 14 (ou 255) caractères parmi le jeu ASCII. Le système n'impose aucun format. On évite les caractères invisibles et les méta-caractères (\*, ?, [ et ]).
- **La taille du fichier :** C'est son nombre d'octets. Elle sert à déterminer la fin du fichier. Il n'y a donc pas de marque de fin de fichier.
- **Droits d'accès :** Trois groupes d'autorisation :
  - l'utilisateur propriétaire,
  - les personnes appartenant au groupe propriétaire et
  - les autres.



## La commande ls -i

Pour voir les numéros d'i-nodes par la commande :

```
prompt> ls -i
```

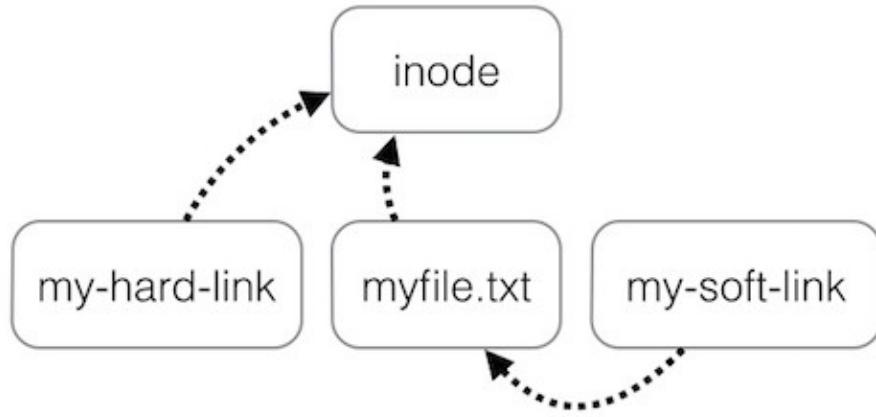
```
423  fichA
```

```
666  fichB
```

```
759  fichC
```

# La commande ln [link]

- Lien dur « hard link » : un fichier qui pointe **vers le même inode** qu'un autre fichier.
- Lien symbolique « soft link » : un fichier qui pointe **vers un chemin** donné.



```
echo 'Hello, World!' > myfile.txt
```

```
ln myfile.txt my-hard-link % hard link
```

```
ls -i myfile.txt my-hard-link % ils ont les même inode
```

```
ln -s myfile.txt my-soft-link % lien symbolique
```

```
ls -i myfile.txt my-soft-link % ils n'ont pas les mêmes inode
```

```
rm myfile.txt % n'affecte pas le lien dur, alors que le lien soft est cassé : il renvoie vers un fichier qui n'existe plus
```



## La commande du

Connaître la taille [disk usage] d'une arborescence et de chacun de ses sous-réertoires et fichiers

```
prompt> du .
```

```
8 ./kde/Autostart
```

```
8 ./kde
```

```
4 ./rep
```

```
56 .
```

Connaître le total (-s, --summarize) avec la taille exprimée en K, M et G (-h, --human-readable)

```
prompt> du -hs /home
```

```
620 M /home
```



# La commande file

**file fichier ...** affiche le type du fichier. À utiliser avant de visualiser le contenu d'un fichier pour éviter d'afficher un contenu binaire. ;-)

Exemple :

```
prompt> file /bin/ls /etc/passwd /usr/bin  
bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),dynamically linked (uses shared libs),  
for GNU/Linux 2.6.15, stripped. /bin/ls: demand paged pure executable  
/etc/passwd: ASCII text  
/usr/bin: directory
```

# La commande grep

**grep** recherche dans un ou plusieurs fichiers les lignes qui correspondent à un **motif** (expression régulière) :

**grep [options] motif chemin**

Avec :

**options**: les options possibles (voir man grep)

**motif** : le terme à rechercher, entre guillemets

**chemin** : le chemin du fichier (ou dossier) où faire la recherche

Rq : guillemets facultatifs, mais conseillés si le motif contient des caractères autres qu'alphanumériques.

Exemple :

```
prompt> grep c1 /etc/passwd # Afficher toutes les lignes du fichier /etc/passwd contenant la chaîne c1.
```

```
c1:vs9Fi0TbjD6xg:208:2001:eleve 1:/usr/c1:/bin/ksh c10:vs9Fi9bjD6xg:209:2001:eleve
```

```
10:/usr/c10:/bin/ksh
```

```
prompt> grep "^m" /etc/passwd # Les lignes qui commencent par « m »
```

```
prompt> grep "[0-9]\{10\}" * # recherche des suites de 10 chiffres dans tous les fichiers
```

<https://wodric.com/commande-grep/>

<https://www.geeksforgeeks.org/grep-command-in-unixlinux/>



# Commandes d'affichage de fichiers

**cat**

**head**

**less**

**more**

**pg** : affiche le contenu du fichier passé en argument par pages de 23 lignes

**pr** : formatte le texte du fichier passé en argument pour l'impression.

<https://www.geeksforgeeks.org/pr-command-in-linux/>



## La commande wc

**wc** compte le nombre de lignes, de mots et de caractères contenus dans le fichier passé en argument,

Exemple:

```
prompt> wc fichier  
8 48 208 fichier
```

→ Le fichier « **fichier** » possède 8 lignes, 48 mots et 208 caractères.



## La commande sum

**sum** calcule et affiche une **somme de contrôle** (checksum) pour un fichier (intégrité des fichiers).

Exemple :

```
prompt> sum fichier  
08860    1
```

# La commande diff

**diff** affiche les lignes différentes devant être modifiées pour que les deux fichiers soient identiques.

Exemple :

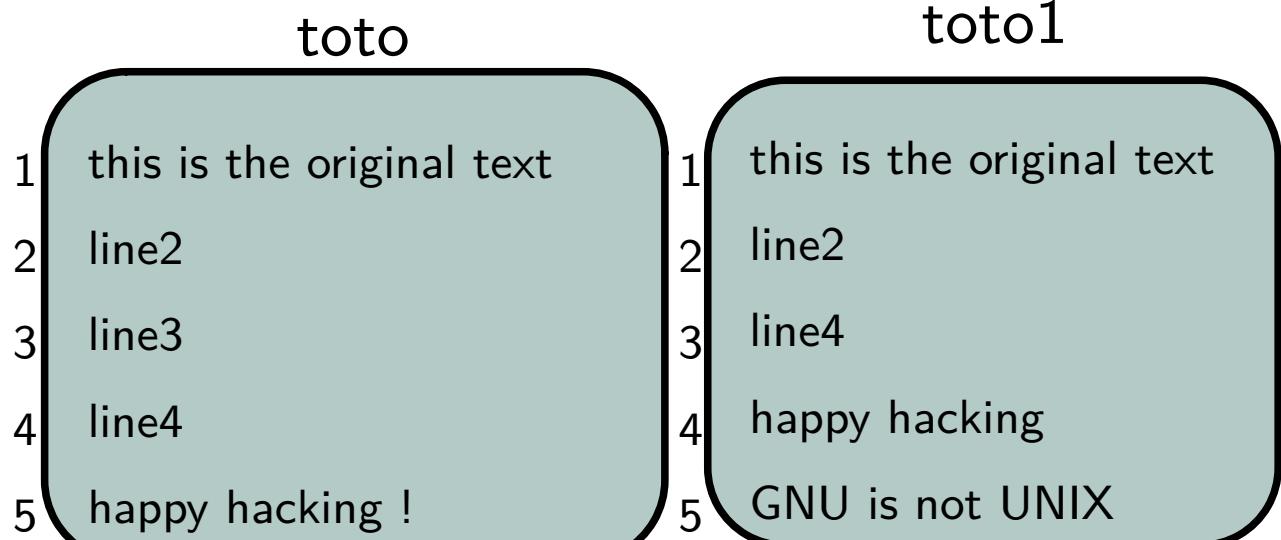
```
prompt> diff toto toto1
```

```
3d2
```

```
< line3
```

```
5a5
```

```
> GNU is not UNIX
```



# La commande diff

**diff** affiche les lignes différentes devant être modifiées pour que les deux fichiers soient identiques.

Exemple :

```
prompt> diff -u toto toto1
```

```
--- toto 2024-09-22 18:10:46.370960519 +0200
```

```
+++ toto1    2024-09-22 18:10:14.937189801 +0200
```

```
@@ -1,5 +1,5 @@
```

```
this is the original text
```

```
line2
```

```
-line3
```

```
line4
```

```
happy hacking !
```

```
+GNU is not UNIX
```

toto

```
1 this is the original text  
2 line2  
3 line3  
4 line4  
5 happy hacking !
```

toto1

```
1 this is the original text  
2 line2  
3 line4  
4 happy hacking  
5 GNU is not UNIX
```

# La commande **cmp**

**cmp** compare octet par octet les deux fichiers passés en paramètres. Cette commande renvoie « 0 » si les fichiers sont identiques, « 1 » sinon.

Exemple :

```
prompt> cmp toto toto1
```

toto toto1 sont différents: octet 41, ligne 3

```
prompt> cmp toto toto
```

```
prompt>
```

/\*les fichiers sont identiques\*/

toto		toto1	
1	this is the original text	1	this is the original text
2	line2	2	line2
3	line3	3	line4
4	line4	4	happy hacking
5	happy hacking !	5	GNU is not UNIX

# La commande touch

**touch** met à jour la date de dernière modification du fichier. Si le fichier n'existe pas encore, il sera créé (et de taille nulle) sauf si l'option **-c** (no create) est utilisée.

Exemple :

```
prompt> ls -l .
-rw-r--r-- 1 c1 cours 0 Oct 9 1991 toto
prompt> touch toto
prompt> ls -l toto
-rw-r--r-- 1 c1 cours 0 Sep 23 2024 toto
prompt> touch titi
prompt> ls -l .
-rw-r--r-- 1 c1 cours 0 Sep 23 2024 toto
-rw-r--r-- 1 c1 cours 0 Sep 23 2024 titi
```

# La commande `compress`

`compress` opère une compression visant à diminuer l'espace occupé par les différents fichiers référencés (algo de Lempel–Ziv–Welch). Chaque fichier est remplacé par un nouveau fichier dont la référence est obtenue en suffixant la référence d'origine avec l'extension `.Z`. Les caractéristiques du fichier sont conservées. Le fichier d'origine est supprimé.

**`compress [options] liste_fichiers`**

Exemples :

```
prompt> compress -v exemple.xls
```

```
exemple.xls : -- replaced with exemple.xls.Z Compression: 24.57%
```

le fichier **exemple.xls** est compressé et remplacé par le fichier **exemple.xls.Z**

```
prompt> compress -rv abc
```

comprime tous les fichiers contenus dans **abc** et ses sous répertoires de manière récursive



# La commande **uncompress**

**uncompress** permet la **décompression** et la reconstruction d'une série de fichiers à partir de leurs formes compressées avec la commande **compress**.

**uncompress [options] liste\_fichiers**

Exemple d'utilisation :

```
prompt> uncompress -v exemple.xls.Z
exemple.xls : 24.6% -- replaced with exemple.xls
le fichier exemple.xls.Z est décompressé et remplacé par le fichier exemple.xls
```

# La commande zcat

**zcat** permet d'afficher de manière lisible le contenu d'un fichier compressé par la commande **compress**.

***zact [options] liste\_fichiers.Z***

Exemple d'utilisation :

```
prompt> cat toto
```

```
tototototototototototototototototo
```

```
prompt> compress toto
```

```
prompt> cat toto.Z
```

```
??t?(???"
```

```
??
```

```
prompt> zcat toto.Z
```

```
totototototototototototototototo
```

# La commande tar

**tar** (**tape archiver**) permet de gérer des archives de fichiers. La **clé** définit les actions de la commande. Elle est constituée d'une suite de caractères définissant la **fonction** (crtux) et des **qualificatifs** de cette commande (Abfhmov).

**tar [clé] liste\_fichiers.tar [chemin]**

Exemples d'utilisation :

prompt> **tar cvf archive\_cible.tar /etc**

Place tous les fichiers du répertoire **/etc** dans le fichier **archive\_cible.tar**

prompt> **tar xvf archive\_cible.tar**

Extraction de **archive\_cible.tar** dans le répertoire courant.

prompt> **tar tvf archive\_cible.tar**

Liste les fichiers contenus dans **archive\_cible.tar**



# La commande tar – les fonctions (crtux)

- c** : création d'une nouvelle archive. Sur bande, l'écriture de l'archive a lieu en début de bande et non à la suite du dernier fichier ;
- r** : fonction de remplacement permettant d'écrire en fin d'archive les fichiers de références données ;
- t**: liste des références de fichiers dans l'archive sans restitution ;
- u** : les fichiers sont ajoutés en fin d'archive s'ils n'y figurent pas encore ou si la date de modification de la dernière version archivée est antérieure à la version du fichier sur le disque ;
- x**: fonction d'extraction de l'archive. Si la référence examinée est une référence de répertoire, son contenu est extrait de manière récursive. Si aucune référence de fichier n'est donnée, tous les fichiers de l'archive sont extraits.

# La commande tar – les qualificatifs (Abfhmov)

- A : les messages d'avertissement sont supprimés ;
  - f : l'argument suivant est interprété comme une référence de fichier correspondant au nom de l'archive (au lieu d'une référence par défaut qui est en général celle d'un fichier spécial associé à un dérouleur de bande). Si cet argument est -, la commande lit sur l'entrée standard ou écrit sur la sortie standard ;
  - h : les liens symboliques sont suivis (par défaut, ils ne le sont pas) ;
  - v : option « verbeuse ».
- ... et les autres → **man tar**



# La commande find

**find** parcourt récursivement l'arborescence en sélectionnant des fichiers selon des **critères de recherche**, et **exécute des actions** sur chaque fichier sélectionné.

**find *répertoire\_de\_départ* [*critère\_de\_recherche*] *action\_à\_exécuter***

Exemple :

```
$ find ~ -print
```

parcoure toute l'arborescence à partir du home (~), sélectionne tous les fichiers (puisque n'y a aucun critère de recherche), et affiche le nom de chaque fichier trouvé.

# La commande `find` – critère de recherche

`-name modèle` sélectionne uniquement les fichiers dont le nom correspond au modèle.

**Attention !** Le modèle doit être interprété par la commande `find` et non par le shell, donc s'il contient des caractères spéciaux pour le shell (par exemple `*`), ceux-ci doivent être échappés.

Mauvais exemple : \$ `find /usr/c1 -name *.c -print`

Le shell remplace `*.c` par la liste des fichiers finissant par `.c` du répertoire `/usr/c1`, puis va chercher dans l'arborescence donnée ces noms de fichiers.

Cela reviendra à : \$ `find /usr/c1 -name f1.c f2.c f3.c -print`

Problème : `-name` n'accepte qu'un seul argument !

Par contre dans : \$ `find /usr/c1 -name '*'.c -print`

C'est bien `*.c` qui sera passé en argument de l'option `-name` de la commande `find`. La recherche se fera donc bien sur les trois fichiers `f1.c`, `f2.c` et `f3.c`.



## La commande **find** – critère de recherche

**-perm *nombre\_octal*** sélectionne les fichiers dont les droits d'accès sont ceux indiqués par le nombre octal.

Exemple : Afficher tous les fichiers qui sont autorisés en lecture, écriture et exécution pour l'utilisateur propriétaire, les personnes du groupe propriétaire et tous les autres.

```
$ find /usr/c1 -perm 777 -print
```



# La commande `find` – critère de recherche

`-type caractère` sélectionne les fichiers dont le type est celui indiqué par le caractère.

C'est-à-dire :

- **c** pour un fichier spécial en mode caractère
- **b** pour un fichier spécial en mode bloc
- **d** pour un répertoire
- **f** pour un fichier normal
- **l** pour un lien symbolique

Exemple : afficher tous les répertoires et sous-répertoires de `/usr/c1`.

```
$ find /usr/c1 -type d -print
```

# La commande `find` – critère de recherche

`-links nombre_décimal` sélectionne les fichiers dont le nombre de liens est donné par le nombre décimal. Si le nombre est précédé d'un + (d'un -) cela signifie supérieur (inférieur) à ce nombre.

Exemple : afficher tous les fichiers qui ont plus de deux liens.

```
$ find /usr/c1 -links +2 -print
```

`-user n[ou]m_utilisateur` sélectionne les fichiers dont l'utilisateur propriétaire est *nom\_utilisateur* ou dont le numéro d'utilisateur (UID) est *num\_utilisateur*.

Exemple : Afficher tous les fichiers spéciaux appartenant à l'utilisateur *c1*

```
$ find /dev -user c1 -print
```



# La commande `find` – critère de recherche

**-inum *nombre\_décimal*** sélectionne les fichiers ayant pour numéro d'i-noeud *nombre\_décimal*.

**-newer *fichier*** sélectionne les fichiers qui sont plus récents que celui passé en argument.

**-atime *nombre\_décimal*** sélectionne les fichiers qui ont été accédés dans les *nombre\_décimal* derniers jours.

**-mtime *nombre\_décimal*** sélectionne les fichiers qui ont été modifiés dans les *nombre\_décimal* derniers jours.

**-size *nombre\_décimal[c]*** sélectionne les fichiers dont la taille est de *nombre\_décimal* blocs. Si on postfixe le *nombre\_décimal* par le caractère **c**, alors la taille sera donnée en nombre de caractères.

# La commande `find` – critère de recherche

Plusieurs critères peuvent être groupés (combinés) par les opérateurs `(` et `)`.

**Attention** : pour le shell, ce sont des caractères spéciaux, ils doivent être échapés.

- Le **ET** logique est implicite : on met plusieurs critères à la suite et `find` sélectionne les fichiers qui répondent à tous les critères.

Exemple : afficher les fichiers se terminant par `.c` **ET** modifiés dans les 3 derniers jours.

```
$ find /usr/c1 \(\ -name '*.c' -mtime -3 \) -print
```

- Le **OU** logique est représenté par l'opérateur `-o`

Exemple : affiche tous les fichiers se terminant par `.txt` **OU** `.doc`.

```
$ find /usr/c1 \(\ -name '*.txt' -o -name '*.doc' \) -print
```

- Le **NON** logique est l'opérateur `!`

Exemple : afficher tous les fichiers **n'appartenant PAS** à `c1`, mais qui se trouvent dans son arborescence.

```
$ find /usr/c1 ! -user c1 -print
```

# La commande `find` – les actions possibles

- **-print** affiche le nom des fichiers sélectionnés sur la sortie standard.

Exemple : afficher toute l'arborescence de `c1`.

```
$ find /usr/c1 -print
```

- **-exec *commande* \;** exécute *commande* sur tous les fichiers sélectionnés. Dans la commande shell, « `{}` » sera remplacé par le nom du fichier sélectionné.

Exemple : rechercher tous les fichiers se terminant par l'extension `.o` dans l'arborescence `/usr/c1` et les détruit, puis recherche les fichiers se terminant par l'extension `.o` pour vérifier

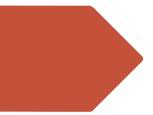
```
$ find /usr/c1 -name '*.o' -exec rm {} \;
$ find /usr/c1 -name '*.o' -print
```

Exemple : rechercher dans les répertoires `/dev` et `/home`, tous les fichiers appartenant à `hugo`, en format long.

```
$ find /dev /home -user hugo -exec ls -l {} \;
```



TD3 :  
L'arborescence du système de  
fichiers d'UNIX



# Recap

- En Linux, tout est **fichier**, même les commandes du terminal qui sont stockées dans **/bin**
- En Linux, l'utilisateur **root** a tous les droits.
- **cd** [change directory], **ls** [list], **cat** [concatenate], **cp** [copy], **rm** [remove]

**cd /bin**

**ls**

**cat ls**

**cp ls machintruc**

**sudo cp ls machintruc**

**sudo rm ls**

**ls**

**machintruc**





## **Menu du jour :**

- wooclap de recap du CM4
- encore quelques petites commandes sur les fichiers (compression, décompression, recherche)
- les droits sur les dossiers/fichiers
- les redirections sur fichiers

# Wooclap de recap du CM3/4



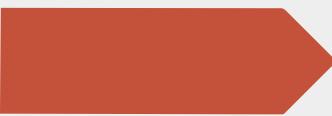
1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code d'événement  
dans le bandeau supérieur

Code d'événement  
**MSLPYC**



## **Encore quelques commandes sur les fichiers**

# La commande **cmp**

**cmp** compare octet par octet les deux fichiers passés en paramètres. Cette commande renvoie « 0 » si les fichiers sont identiques, « 1 » sinon.

Exemple :

```
prompt> cmp toto toto1
```

toto toto1 sont différents: octet 41, ligne 3

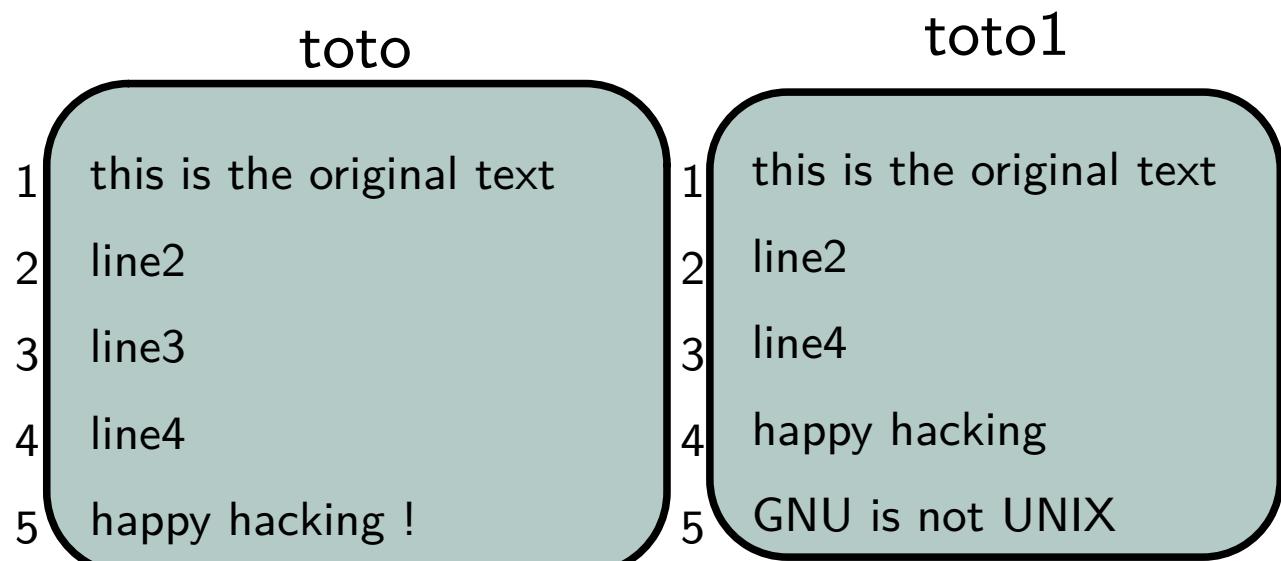
```
prompt> echo $?
```

1

```
prompt> cmp toto toto
```

```
prompt> echo $?
```

0



# La commande `compress`

`compress` opère une compression visant à diminuer l'espace occupé par les différents fichiers référencés (algo de Lempel–Ziv–Welch). Chaque fichier *nomfichier* à compresser est remplacé par un nouveau fichier *nomfichier.Z* qui conserve les caractéristiques du fichier initial.

**`compress [options] liste_fichiers`**

Exemples :

prompt> **`compress -v exemple.xls`**

exemple.xls : -- replaced with exemple.xls.Z Compression: 24.57%

le fichier **`exemple.xls`** est compressé et remplacé par le fichier **`exemple.xls.Z`**

prompt> **`compress -rv abc`**

comprime tous les fichiers contenus dans **`abc`** et ses sous répertoires de manière récursive (**`-r`**)

<https://www.geeksforgeeks.org/compress-command-in-linux-with-examples/>

# La commande **uncompress**

**uncompress** permet la **décompression** et la reconstruction d'une série de fichiers à partir de leurs formes compressées avec la commande **compress**.

**uncompress [options] liste\_fichiers**

Exemple d'utilisation :

```
prompt> uncompress -v exemple.xls.Z
exemple.xls : 24.6% -- replaced with exemple.xls
le fichier exemple.xls.Z est décompressé et remplacé par le fichier exemple.xls
```

# La commande zcat

**zcat** permet d'afficher de manière lisible le contenu d'un fichier compressé par la commande **compress**.

**zact [options] liste\_fichiers.Z**

Exemple :

prompt> **cat toto**

totototototototototototototototototo

prompt> **compress toto**

prompt> **cat toto.Z**

??t?(???"

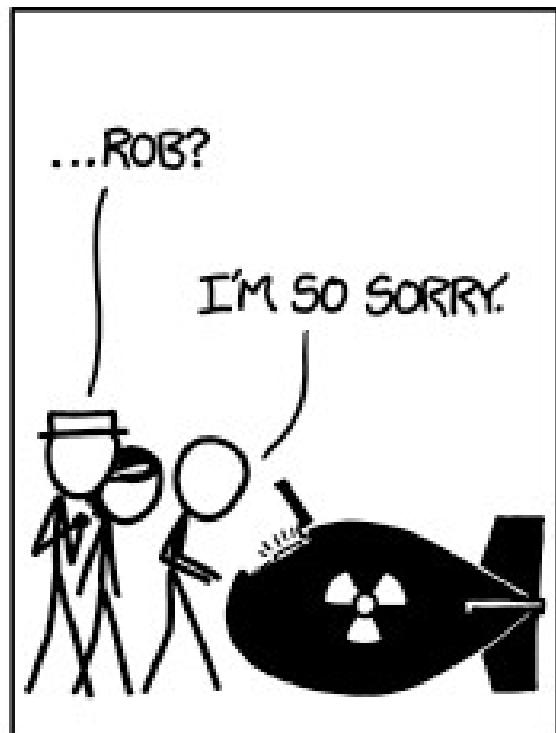
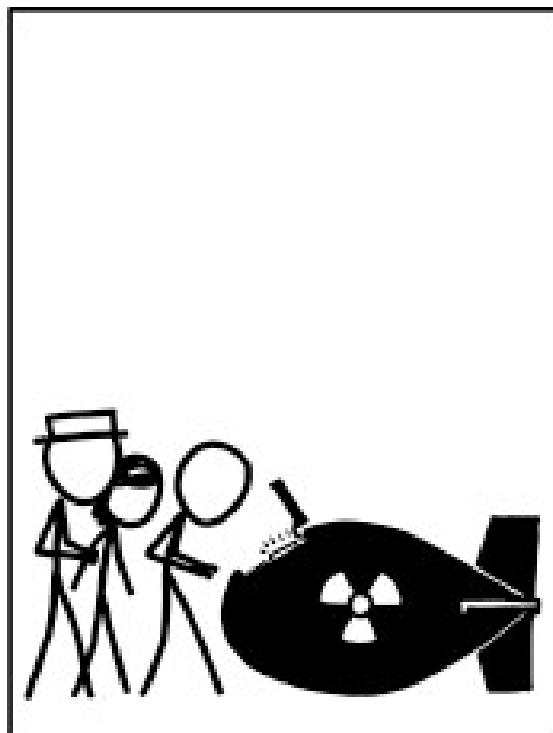
?

prompt> **zcat toto.Z**

tototototototototototototototototo

<https://www.computerhope.com/unix/uzcat.htm>

# La commande tar



<https://xkcd.com/1168/>

# La commande tar

**tar** (tape archiver) gère des archives de fichiers. La **clé** définit l'action de la commande. Elle est constituée d'un caractère définissant la **fonction** et d'autres définissant des **qualificatifs**.

**tar [clé] liste\_fichiers.tar [chemin]**

Exemples :

prompt> **tar cfv archive\_cible.tar /etc**

Crée une archive des fichiers du répertoire **/etc** dans le fichier **archive\_cible.tar**

prompt> **tar fxv archive\_cible.tar**

Extraction de **archive\_cible.tar** dans le répertoire courant.

prompt> **tar tvf archive\_cible.tar**

Liste les fichiers contenus dans **archive\_cible.tar**



# La commande tar – les fonctions

Il faut au moins une fonction :

**-c** : création d'une nouvelle archive

**-r** : fonction de remplacement permettant d'écrire en fin d'archive les fichiers de références données

**-d**: trouve les différences entre les archives

**--delete** : supprime de l'archive

**-u** : (update) les fichiers sont ajoutés en fin d'archive s'ils n'y figurent pas encore ou si la date de modification de la dernière version archivée est antérieure à la version du fichier sur le disque

**-x**: fonction d'extraction de l'archive. Si la référence examinée est une référence de répertoire, son contenu est extrait de manière récursive. Si aucune référence de fichier n'est donnée, tous les fichiers de l'archive sont extraits

... et les autres → **man tar**



# La commande tar – les qualificatifs

- f : l'argument suivant est interprété comme une référence de **fichier** correspondant au nom de l'archive
- h : les liens symboliques sont suivis (par défaut, ils ne le sont pas)
- z : compresse avec **gzip**
- Z : compresse avec **compress**
- v : option « **verbeuse** »

... et les autres → **man tar**



# La commande find

**find** parcourt récursivement l'arborescence en sélectionnant des fichiers selon des **critères de recherche**, et **exécute des actions** sur chaque fichier sélectionné.

**find *répertoire\_de\_départ* [*critère\_de\_recherche*] *action\_à\_exécuter***

Exemple :

```
$ find ~ -print
```

parcours toute l'arborescence à partir du home (~), sélectionne tous les fichiers (puisque n'y a aucun critère de recherche), et affiche le nom de chaque fichier trouvé.

# La commande `find` – les actions possibles

- `-print` affiche le nom des fichiers sélectionnés sur la sortie standard.

Exemple : afficher toute l'arborescence de `c1`.

```
$ find /usr/c1 -print
```

- `-exec commande \;` exécute `commande` sur tous les fichiers sélectionnés. Dans la commande shell, « `{}` » sera remplacé par le nom du fichier sélectionné.

Exemple : rechercher tous les fichiers se terminant par l'extension `.o` dans l'arborescence `/usr/c1` et les détruit, puis recherche les fichiers se terminant par l'extension `.o` pour vérifier

```
$ find /usr/c1 -name '*.o' -exec rm {} \;
$ find /usr/c1 -name '*.o' -print
```

Exemple : rechercher dans les répertoires `/dev` et `/home`, tous les fichiers appartenant à `nanis`, et afficher des infos en format long.

```
$ find /dev /home -user nanis -exec ls -l {} \;
```

# La commande `find` – critère de recherche

`-name modèle` sélectionne uniquement les fichiers dont le nom correspond au modèle donné.

**Attention !** Le modèle doit être interprété par la commande `find` et non par le shell, donc s'il contient des caractères spéciaux pour le shell (par exemple `*`), ceux-ci doivent être échappés.

Mauvais exemple : \$ `find /usr/c1 -name *.c -print`

Le shell remplace `*.c` par la liste des fichiers finissant par `.c` du répertoire `/usr/c1`, puis va chercher dans l'arborescence donnée ces noms de fichiers.

Cela reviendra à : \$ `find /usr/c1 -name f1.c f2.c f3.c -print`

Problème : `-name` n'accepte qu'un seul argument !

Par contre dans : \$ `find /usr/c1 -name '*'.c -print`

C'est bien `*.c` qui sera passé en argument de l'option `-name` de la commande `find`. La recherche se fera donc bien sur les trois fichiers `f1.c`, `f2.c` et `f3.c`.

## La commande **find** – critère de recherche

**-perm *nombre\_octal*** sélectionne les fichiers dont les droits d'accès sont ceux indiqués.

Exemple : Afficher tous les fichiers qui sont autorisés en lecture, écriture et exécution pour l'utilisateur propriétaire, les personnes du groupe propriétaire et tous les autres.

```
$ find /usr/c1 -perm 777 -print
```



# La commande `find` – critère de recherche

`-type caractère` sélectionne les fichiers dont le type est celui indiqué.

C'est-à-dire :

- **f** pour un **fichier normal**
- **l** pour un **lien symbolique**
- **d** pour un **répertoire**
- **c** pour un fichier spécial en mode caractère
- **b** pour un fichier spécial en mode bloc

Exemple : afficher tous les répertoires et sous-répertoires de `/usr/c1`.

```
$ find /usr/c1 -type d -print
```

## La commande **find** – critère de recherche

**-links *nombre\_décimal*** sélectionne les fichiers qui ont le nombre donné de liens. Si le nombre est précédé d'un + (d'un -) cela signifie supérieur (inférieur) à ce nombre.

Exemple : pour afficher tous les fichiers qui ont plus de deux liens

```
$ find /usr/c1 -links +2 -print
```

**-user *n[ou]m\_utilisateur*** sélectionne les fichiers dont l'utilisateur propriétaire est *nom\_utilisateur* ou dont le numéro d'utilisateur (UID) est *num\_utilisateur*.

Exemple : pour afficher tous les fichiers spéciaux appartenant à l'utilisateur **c1**

```
$ find /dev -user c1 -print
```



# La commande `find` – critère de recherche

**-inum *nombre\_décimal*** sélectionne les fichiers ayant pour numéro d'i-noeud *nombre\_décimal*.

**-newer *fichier*** sélectionne les fichiers qui sont plus récents que celui passé en argument.

**-atime *nombre\_décimal*** sélectionne les fichiers qui ont été accédés dans les *nombre\_décimal* derniers jours.

**-mtime *nombre\_décimal*** sélectionne les fichiers qui ont été modifiés dans les *nombre\_décimal* derniers jours.

**-size *nombre\_décimal[c]*** sélectionne les fichiers dont la taille est de *nombre\_décimal* blocs. Si on postfixe le *nombre\_décimal* par le caractère **c**, alors la taille sera donnée en nombre de caractères.

# La commande `find` – critère de recherche

Plusieurs critères peuvent être groupés (combinés) par les opérateurs `(` et `)`.

**Attention** : pour le shell, ce sont des caractères spéciaux, ils doivent être échapés.

- Le **ET** logique est implicite : on met plusieurs critères à la suite et `find` sélectionne les fichiers qui répondent à tous les critères.

Exemple : afficher les fichiers se terminant par `.c` **ET** modifiés dans les 3 derniers jours.

```
$ find /usr/c1 \(-name '*.c' -mtime -3 \) -print
```

- Le **OU** logique est représenté par l'opérateur `-o`

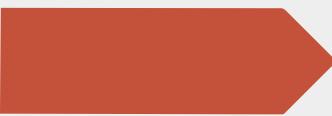
Exemple : affiche tous les fichiers se terminant par `.txt` **OU** `.doc`.

```
$ find /usr/c1 \(-name '*.txt' -o -name '*.doc' \) -print
```

- Le **NON** logique est l'opérateur `!`

Exemple : afficher tous les fichiers **n'appartenant PAS** à `c1`, mais qui se trouvent dans son arborescence.

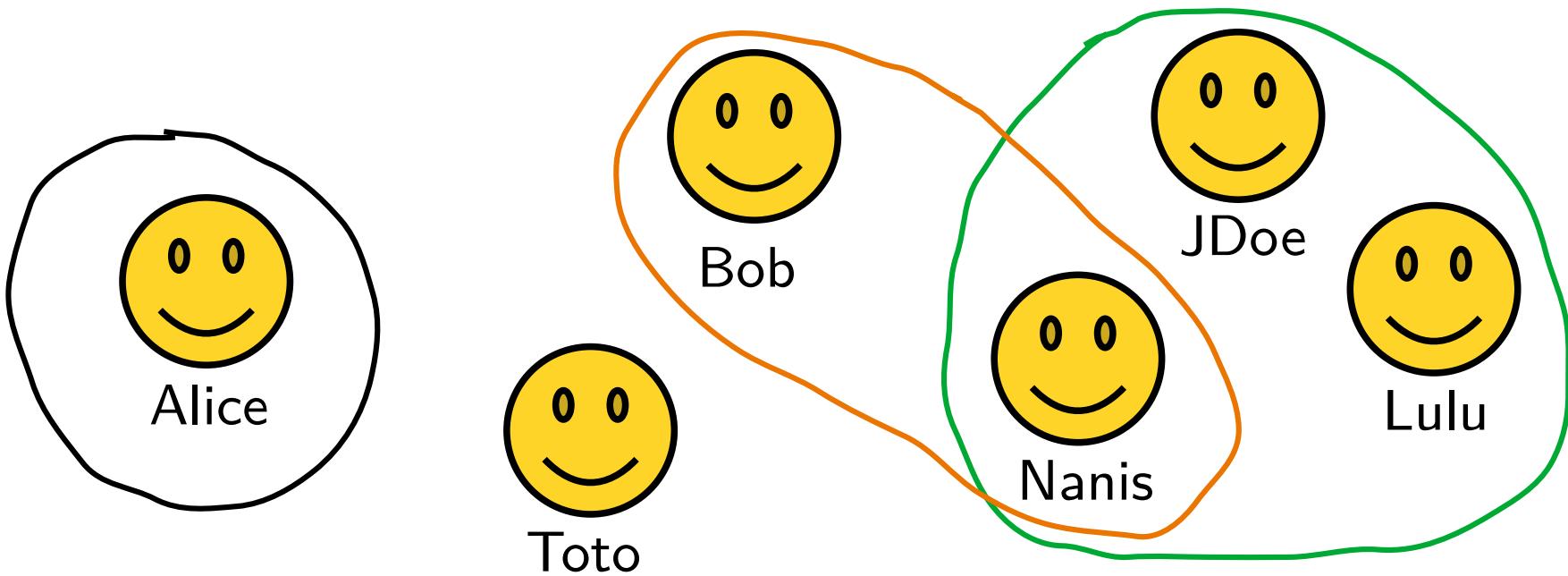
```
$ find /usr/c1 ! -user c1 -print
```



## **Fichiers : les droits**

# Rappels concernant les utilisateur.ices

- Les **utilisateur.ices** du système ont chacun.es leur **compte**.
- Leurs **droits** peuvent être administrés individuellement, ou en lot via des **groupes**.
- Il y a un compte « admin » appelé **root**, qui a **tous les droits**.
- Certain.es utilisateur.ices ont le droit d'utiliser **sudo** pour prendre les droits root.





# Appartenance d'un fichier / dossier

Un fichier / dossier a deux types de propriétaires :

- Un.e utilisateur.ice noté.e **u**
- Un groupe noté **g**

Les utilisateur.ices qui ne sont ni **u** ni dans **g** sont référencés par « autre », noté **o** (other).

⇒ une personne donnée est soit l'utilisateur.ice **u**, soit dans le groupe **g**, soit dans **o**.



# Types de modes d'accès

Type de mode d'accès	Fichier	Dossier
Lecture (r – read)	Lire le fichier (l'afficher : <b>cat</b> , le copier : <b>cp</b> )	Lister son contenu ( <b>ls</b> )
Écriture (w – write)	L'éditer avec <b>vim</b>	Modifier les attributs du dossier et son contenu (créer un fichier, le renommer, le supprimer) /!\ ces modifs nécessitent l'accès à un inode donc faut aussi le droit x.
Exécution (x)	<b>./nomfichier</b>	Droit de passage ( <b>cd</b> ) / Donne l'accès à linode d'un contenu dont on connaît le nom ( <b>ls -l fichier, stat fichier</b> )

**cat /home/nanis/toto** nécessite que la commande **cat** ouvre **toto** en **mode lecture**, mais aussi qu'elle ait le droit d'**exécution** sur « / », « **home** », et « **nanis** » pour localiser/traverser chaque dossier via les inodes.

# Permissions d'utiliser un mode : notations

- **Notation chaîne de caractère :**

Chaque groupe de 3 caractères correspond à une catégorie (**u**, **g**, **o**).

Si le droit pour un mode n'est pas attribué, on met un tiret.

Exemple : **rwx rwx r-x**

- **Notation binaire :**

On met 1 si le droit est donné, 0 sinon. (→ 0 s'il y a un tiret, 1 sinon).

Exemple : **111 111 101**

- **Notation octale :**

À partir de la notation binaire : on convertit chaque groupe de trois bits dans sa forme décimale/octale.

À partir de la chaîne de caractère : « **r** » vaut 4, « **w** » vaut 2 et « **x** » vaut 1.

Exemple : **775**

$$7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 4 + 2 + 1$$

$$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 0 + 1$$

lettres	binaire	octal
---	000	0
--x	001	1
-w-	010	2
-wx	011	3
r--	100	4
r-x	101	5
rw-	110	6
rwx	111	7

# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code d'événement  
dans le bandeau supérieur

Code d'événement  
**MSLPYC**



# La commande ls -l

**ls -l** affiche les noms de fichiers et dossier par ligne avec tout un tas d'infos, dont les droits

```
prompt> cd /home # contient un fichier « toto » et un dossier « dossier »
```

```
prompt> ls -l toto
```

```
-rw-r--r-- 1 c1 cours 342 Oct 18 15:28 toto
```

```
prompt> ls -l dossier
```

```
total 4
```

```
-rw-r--r-- 1 vaginay241 utilisateurs_du_domaine 18 sept. 22 18:45 contenu1-dans-dossier
```

```
prompt> ls -ld dossier
```

```
drwxr-xr-x 2 vaginay241 utilisateurs_du_domaine 20 sept. 22 18:47 dossier
```

```
prompt > ls -l | grep dossier
```

```
drwxr-xr-x 2 vaginay241 utilisateurs_du_domaine 20 sept. 22 18:47 dossier
```

# Les permissions : attention !

```
[rw] .
└ [rwx] f_tous
$ cat f_tous
cat: f_tous: Permission non accordée
→ on ne peut pas lire un fichier si on n'a pas
le droit « x » sur le dossier qui le contient.
```

```
[rwx] .
└ [x] toto.exe
$ ./toto.exe
bonjour
→ pas besoin du droit « r » sur le binaire
pour l'exécuter.
```

```
[rwx] .
└ [ ] f_aucun
$ rm f_aucun
→ supprimer un fichier ne nécessite pas
d'avoir les droits dessus.
```

```
[rwx] .
└ [x] script.sh
$ ./script.sh
/bin/bash: ./script.sh: Permission non
accordée
→ le shell a besoin de lire un script bash pour
pouvoir l'exécuter
```

# La commande chmod

**chmod** [change mode] sert à modifier les droits d'accès sur un fichier ou un répertoire

Syntaxe : **chmod code chemin** où **code** est soit :

- un code octal (slide précédente)
- un code symbolique **[ugoa][-+=[modes]**, relatif aux droits actuels.

Exemple d'utilisation :

```
prompt> ls -l toto
-rw-r--r-- 1 nanis cours 342 sept. 24 15:28 toto
prompt> chmod 770 toto; ls -l toto
-rwxrwx--- 1 nanis cours 342 sept. 24 15:28 toto
prompt> chmod gu-x toto; ls -l toto
-rw-rw---- 1 nanis cours 342 sept. 24 15:28 toto
```

Personne concernée	
propriétaire	u
groupe	g
autres	o
tous	a
Action	
ajouter	+
enlever	-
initialiser	=
Accès autorisés en	
lecture	r
écriture	w
exécution/traverse	x

# Wooclap



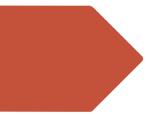
1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code d'événement  
dans le bandeau supérieur

Code d'événement  
**MSLPYC**



# La commande chmod – Exemple

```
prompt> ls -l fichA  
-rw-rw-rw- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod go-w fichA; ls -l fichA  
-rw-r--r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod u+x fichA; ls -l fichA  
-rwxr--r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod g-r fichA; ls -l fichA  
-rwx---r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod ug+rw fichA; ls -l fichA  
-rwxrw-r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod g-rw fichA; ls -l fichA  
-rwx---r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

```
prompt> chmod ug=rw fichA; ls -l fichA  
-rw-rw-r-- 1 c1 cours 342 Oct 18 15:28 fichA
```

# La commande umask

**umask** sert à gérer les droit accordés par défaut à un création des dossier et fichiers.

Syntaxe : **umask [-S] [masque]**

Le **masque**, c'est ce qui va être soustrait aux droits par défaut à la création des futurs éléments (777 pour les dossiers et 666 pour les fichiers).

Sans l'argument masque, **umask** renvoie la valeur actuelle du masque.

Exemple :

<b>droits demandés :</b>	<b>rwx rwx rwx ou encore 777</b>
<b>- masque :</b>	<b>--- -w- rwx ou encore 027</b>
<hr/>	
<b>droits accordés :</b>	<b>rwx r-x --- ou encore 750</b>

# Wooclap



1

Allez sur [wooclap.com](https://wooclap.com)

2

Entrez le code d'événement  
dans le bandeau supérieur

Code d'événement  
**MSLPYC**

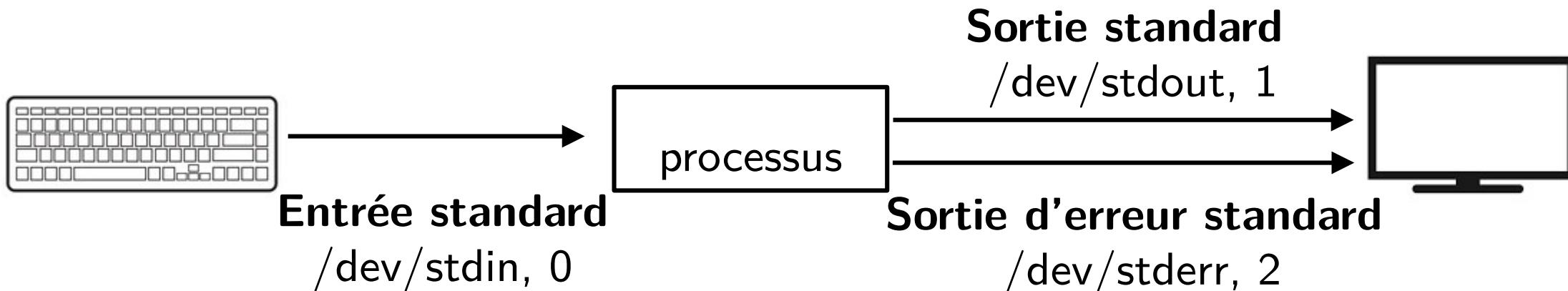


## **Utilisation des fichiers pour la redirection de commandes**

# Flux de données

Tout programme qui s'exécute est par défaut associé à trois fichiers.

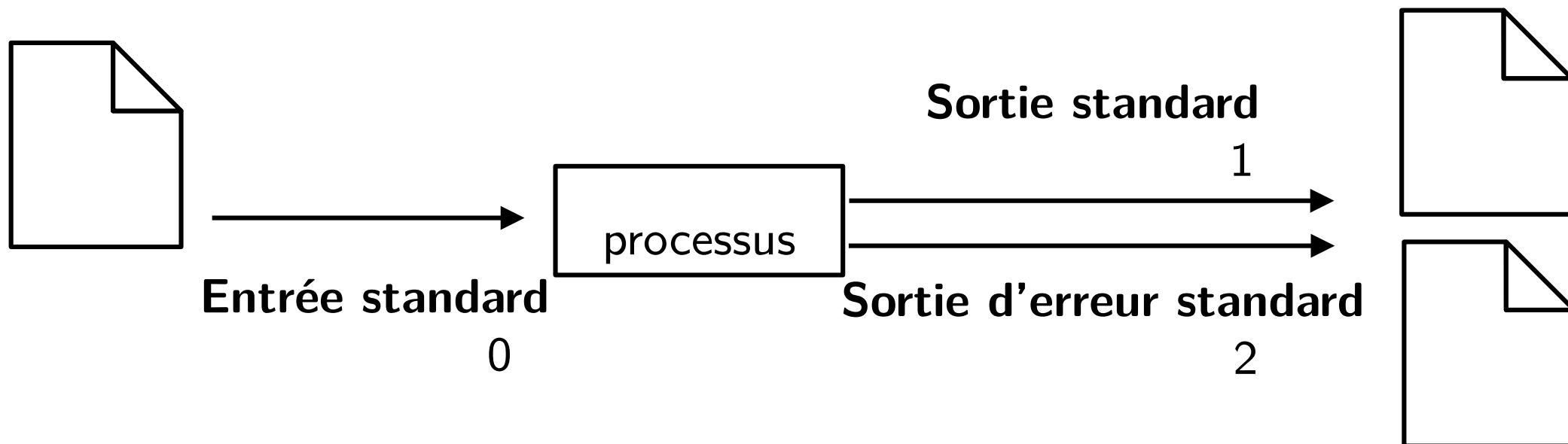
Chaque fichier ouvert est associé à un nombre : le **descripteur de fichier**.



# Flux de données

Tout programme qui s'exécute est par défaut associé à trois fichiers.

Chaque fichier ouvert est associé à un nombre : le **descripteur de fichier**.



# Redirection de la sortie standard, via >

Syntaxe : **commande > fichier**

Si le fichier n'existe pas, il est créé par le Shell. S'il existe déjà, le Shell détruit son contenu pour le remplacer par la sortie de la commande (« clobbering »).

Exemple :

```
prompt> who # liste les personnes connectées au système
c1 tty1 Sept 25 8:16
c2 tty3 Apr 19 2:55
c3 tty6 July 1 20:33
prompt> who > toto.txt # la commande n'affiche rien
prompt> cat toto.txt # le fichier toto.txt contient la sortie de la commande précédente
c1 tty1 Sept 25 8:16
c2 tty3 Apr 19 2:55
c3 tty6 July 1 20:33
```

# Redirection de la sortie standard, via >

Pour rediriger plus d'une commande dans un fichier :

*(commande1 ; commande2) > fichier*

Exemple :

prompt> **(date; who) > who.txt**

→ Les sorties de **date** et **who** seront redirigées dans le même fichier : **who.txt**

prompt> **date; who > who.txt**

→ le Shell exécute **date**, affiche le résultat sur le terminal et lance ensuite **who**, en redirigeant le résultat sur le fichier **who.txt**

## Redirection de la sortie standard, via >>

Pour ne pas écraser le contenu de fichier, mais rajouter la sortie de la commande à la fin d'un fichier (crée si besoin) : **commande >> fichier**

Exemple :

```
prompt> date > date.t ; cat date.t
```

```
Fri Sept 27 10:45:21 MET 2023
```

```
prompt> date >> date.t ; cat date.t
```

```
Fri Sept 27 10:45:21 MET 2023
```

```
Fri Sept 27 10:45:23 MET 2023
```

## Redirection de la sortie d'erreur standard, via 2>

Syntaxe : **commande 2> fichier**

Exemple :

```
prompt> cat fhsdofh
```

```
cat: fhsdofh: Aucun fichier ou dossier de ce nom
```

```
prompt> cat fhsdofh 2> erreur.txt % la commande n'affiche rien
```

```
prompt> cat erreur.txt
```

```
cat : fhsdofh : Aucun fichier ou dossier de ce nom
```

# Redirection de l'entrée standard, via <

Syntaxe : **commande < fichier**

La commande va lire ses données du fichier donné en paramètre.

Exemple :

prompt> **mail nanis < reponse**

Mail envoyé à nanis

→ La commande **mail** lit le texte à envoyer depuis le fichier **reponse**, grâce à la redirection <, au lieu de lire les données à partir du terminal.

# Redirection de l'entrée standard, via <<

Syntaxe : **commande << délimiteur**

La commande va lire ses données du fichier donné en paramètre.

Exemple :

```
prompt> mail nanis <<theend  
blablabalablablatheend  
theendblabla  
theend
```

Mail envoyé à nanis

→ La commande **mail** lit le texte à envoyer depuis le terminal, tant que le délimiteur n'a pas été rencontré.

# Redirection : syntaxe générale

*[n]redir-op word*

*n* : descripteur de fichier (un nombre)

*redir-op* : un opérateur de redirection (parmi `>`, `>>`, `<`, `<<`, ...)

*word* : un chemin, ou un délimiteur, selon le cas.

Le standard posix :

[https://pubs.opengroup.org/onlinepubs/009695399/utilities/xcu\\_chap02.html#tag\\_02\\_07](https://pubs.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_07)

# Redirection : droits

Soient **n** et **m** deux descripteurs de fichiers (des entiers) et **fichier** un chemin vers un fichier.

- **n < fichier** redirige en lecture le descripteur **n** sur **fichier** (qui doit exister).  
**n = 0** (l'entrée standard) par défaut.
- **n > fichier** redirige en écriture le descripteur **n** sur **fichier** (qui est écrasé ou crée).  
**n = 1** (la sortie standard) par défaut.
- **n >> fichier** redirige en écriture le descripteur **n** à la fin de **fichier** (qui est créée si besoin).  
**n = 1** (sortie standard) par défaut.

...

⇒ Il est **nécessaire d'avoir les droits adéquats** sur les fichiers utilisés en redirection !



# Double redirection

Il est possible de rediriger à la fois l'entrée et la sortie :

```
prompt> wc < /etc/passwd > tmp
```

```
prompt> cat tmp
```

```
20 21 752
```



# Redirection : remarques

- Une commande se contente de lire et d'écrire sur des descripteurs. Elle ne connaît pas la provenance et la destination exactes des données qu'elle lit et écrit.
- C'est l'interpréteur de commandes qui traite les demandes de redirection **avant** d'appeler la commande.
- Les redirections sont indépendantes du contexte : les caractères spéciaux « > » et « < » peuvent être situés n'importe où sur une ligne de commande.

```
prompt> who > tmp; grep 'c[12]' < tmp  
c1 tty4 Jul 31 09:46  
c2 tty2 Jul 31 09:17
```

```
prompt> > tmp who; < tmp grep 'c[12]'  
c1 tty4 Jul 31 09:46  
c2 tty2 Jul 31 09:17
```

# Redirections multiples

- Un processus ne possède qu'**une seule entrée, une seule sortie et une seule sortie d'erreur**. Donc chaque descripteur ne peut être redirigé qu'une seule fois par commande !

prompt> **commande > fichier1 > fichier2**

→ **fichier1** est créé mais reste vide, **fichier2** contient la sortie standard de **commande**

- La commande **tee** lit l'entrée standard et l'écrit **à la fois** dans la sortie standard et dans un ou plusieurs fichiers

prompt> **commande | tee fichier1 fichier2**

→ **fichier1** et **fichier2** sont créés et contiennent la sortie standard de **commande**

# Redirection : attention aux typos

Le Shell traite les séparateurs avant les redirections ; par conséquent il y a une différence importante entre les deux commandes suivantes :

```
prompt> cmd 2> fichier
```

```
prompt> cmd 2 > fichier
```

- Dans le premier cas, on redirige la sortie d'erreur de la commande *cmd* vers *fichier*.
- Dans le second cas, on redirige la sortie standard de la commande *cmd* vers *fichier* et **2** sera considéré comme un argument de *cmd* (dû à l'espace entre le **2** et le **>**) !



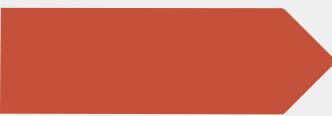
TD 4 :  
La politique d'accès aux  
fichiers d'UNIX



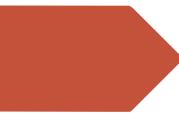


### **Menu du jour : les processus**

- ce qu'ils sont
- comment ils sont gérés par le SE
- ce qu'il se passe à leur naissance, pendant leur vie, et à leur mort.
- comment on peut les faire communiquer (signaux et pipes)



## **Les processus : définitions**

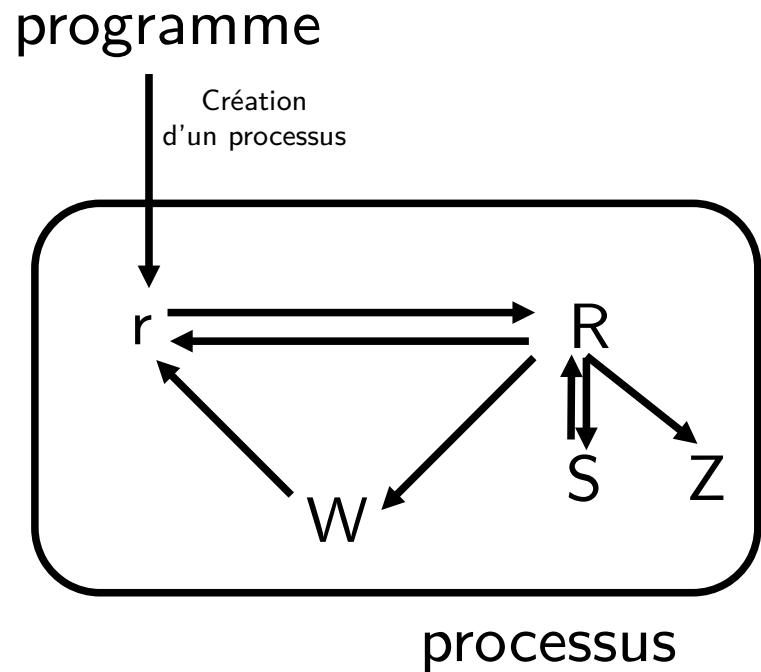


# Introduction aux processus

- Un **programme** est un fichier contenant du code pouvant être exécuté. Exemple : le **a.out**
- Un **processus** (process) = une **instance** d'un programme en cours d'exécution
- Pour être exécuté, un programme est chargé dans la mémoire vive, ses instructions sont exécutées par le processeur. Le système d'exploitation lui fourni un **espace d'addressage** (une zone de mémoire fournie pour qu'il puisse puisse écrire dedans).
- Un système d'exploitation **multitâche** doit traiter plusieurs processus en même temps.
- **ps -aux [ | grep nomprocessus ]**

# L'ordonnanceur et les états possibles

- L'**ordonnanceur (scheduler)** = partie du noyau qui choisit quel processus doit s'exécuter à un moment donné
- Il maintient :
  - Le processus en train de s'exécuter (**R** – Running)
  - Une file de processus prêts à s'exécuter (**r** – runnable)
  - Un ensemble de processus en attente d'un événement (**W** – waiting)
  - Un ensemble de processus endormis (**S** – sleeping)
  - Un ensemble de processus morts (**Z** – zombie)
- Les infos sur les processus sont gérés dans la **table de processus**



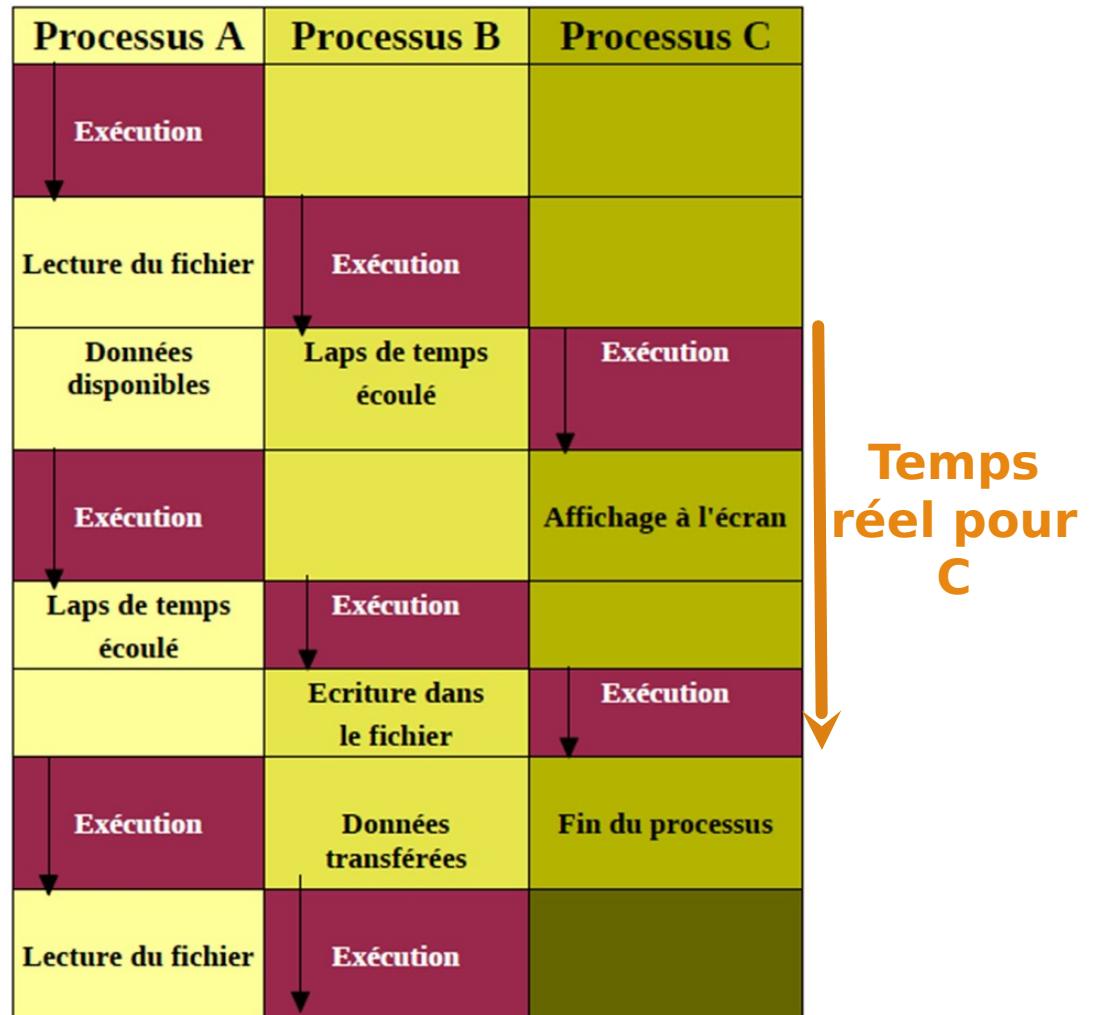
# Table de processus et process control block

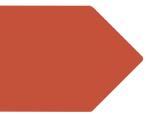
Table de processus = un tableau où chaque case (appelée un process control block) conserve, pour un processus donnée, les infos qui doivent toujours être accessibles par le noyau

- Identité du processus : **PID** : process identifier, **PPID** : parent PID
- **Propriétaires** : réels (qui on est vraiment) et effectif (de qui on a les droits) (cf Exo 1 TD 4)
- **État du processus** (cf diagramme d'avant)
- Le répertoire courant = une référence à un i-node
- **Avancement** du processus = l'état de la mémoire utilisée, la valeur de ses variables, l'adresse de la prochaine instructions, la liste des fichiers ouverts ...
- **Priorité**,
- Autre : événements attendus par le processus, vecteur de signaux que le processus n'a pas encore géré, ...

# Changement de processus

- UNIX est un **système multitâche à temps partagé** (cours 2). Comme il n'a (en général) qu'un seul processeur, il ne peut traiter qu'une tâche à la fois. Pour donner l'illusion du parallélisme, il **commute** rapidement entre les tâches.
- Sur un intervalle de temps assez grand, tous les processus ont progressé, mais à un instant donné **un seul processus est actif**.
- **Temps réel** écoulé != **temps CPU**





# Commutation de contexte

Chaque commutation entre deux processus P1 et P2 nécessite de sauvegarder l'état d'avancement de P1, et de reprendre où P2 s'était arrêté

Si pour une raison x ou y, le système d'exploitation a décidé de changer quel processus est exécuté, le noyau :

- Arrête l'exécution du processus de x,
- Copie les valeurs des registres hardware dans le process control block (sauvegarde du **mot d'état**)
- Charge le mot d'état du processus y (màj de registres avec les valeurs du processus y)
- Lance l'exécution de y

Temps nécessaire pour changer de contexte : **overhead**

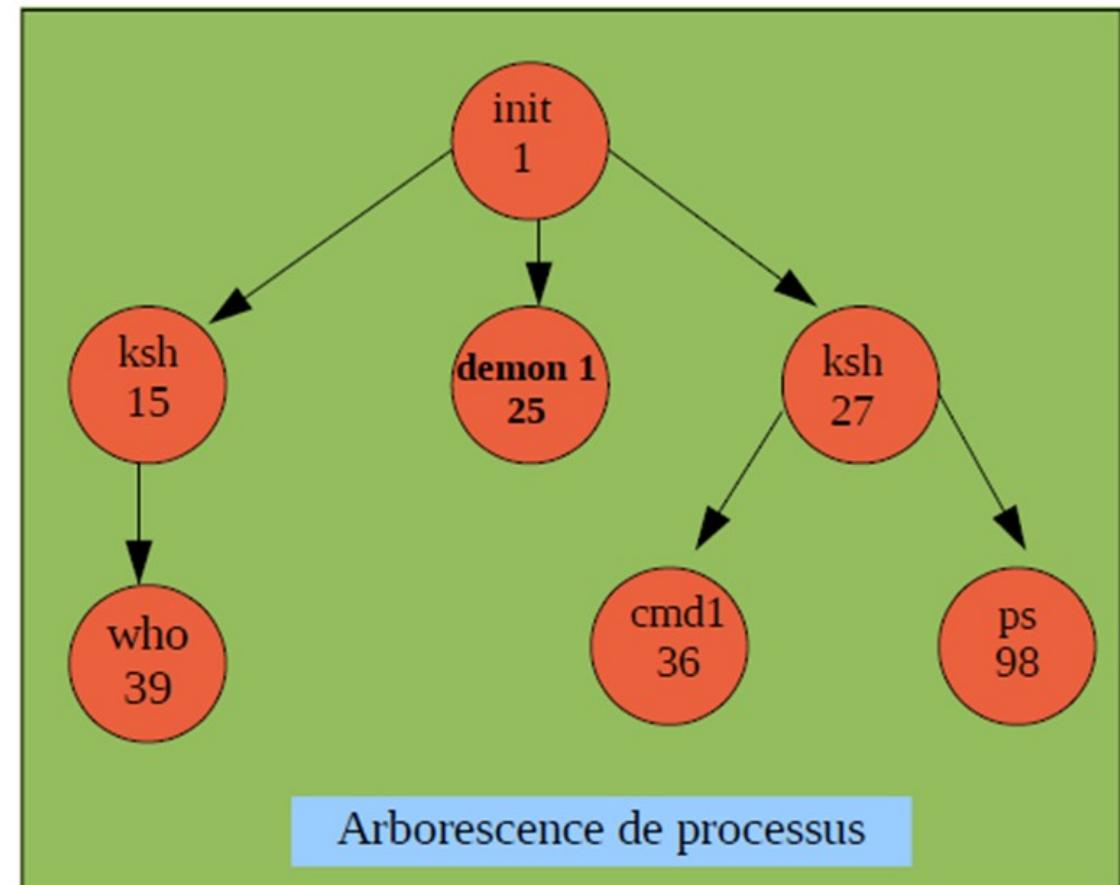


# init, le processus originel

- PID = 1
- Lancé par le kernel pendant la phase de boot du système (**systemd** / **sysVinit** / **upstart** / ...)
- Lance les scripts de démarrage (fichier rc dans /etc), monte le système de fichiers, démarre tous les services nécessaires (démons), crée un processus par terminal (tty) qui attend une connexion de l'utilisateur.ice.

# L'arborescence des processus

- Si une connexion réussie, le processus de login exécute un shell qui peut accepter des commandes.
- Ces commandes peuvent lancer d'autres processus, ...
- Donc : **les processus lancent des processus qui lancent des processus qui ... → un arbre** (/!\ ne pas confondre avec l'arborescence des fichiers)
- Tous les processus dérivent de **init**.





# **Les processus**

## **Comment ils naissent, vivent, et meurent**



# Cycle de vie des processus

- Sauf pour init : un processus parent génère un nouveau processus (mécanisme de **fork** ou de **clone**)
- Le processus fait sa vie
- Le processus meurt
  - Soit par sa propre volonté : fin d'exécution normale (retourne 0), ou erreur non critique (retourne une valeur  $\neq 0$ )
  - Soit involontairement : via une erreur critique, ou une interruption provoquée par un autre processus



# Lancement d'un processus

- Dessin : un processus fait un fork
- Il y a donc 2 processus :
  - Le **processus père P**, qui exécute le programme Shell,
  - Le **processus fils F**, qui exécute la commande.
- Le fils **hérite de tout l'environnement** du processus file père, **sauf** du **PID**, du **PPID** et des **temps d'exécution**.
- Cas 1 : F finit avant P, le SE notifie P qui est tâché de gérer
- Cas 1' : P était en wait mode.
- Cas 2 : P finit avant F, le SE rattache F à init

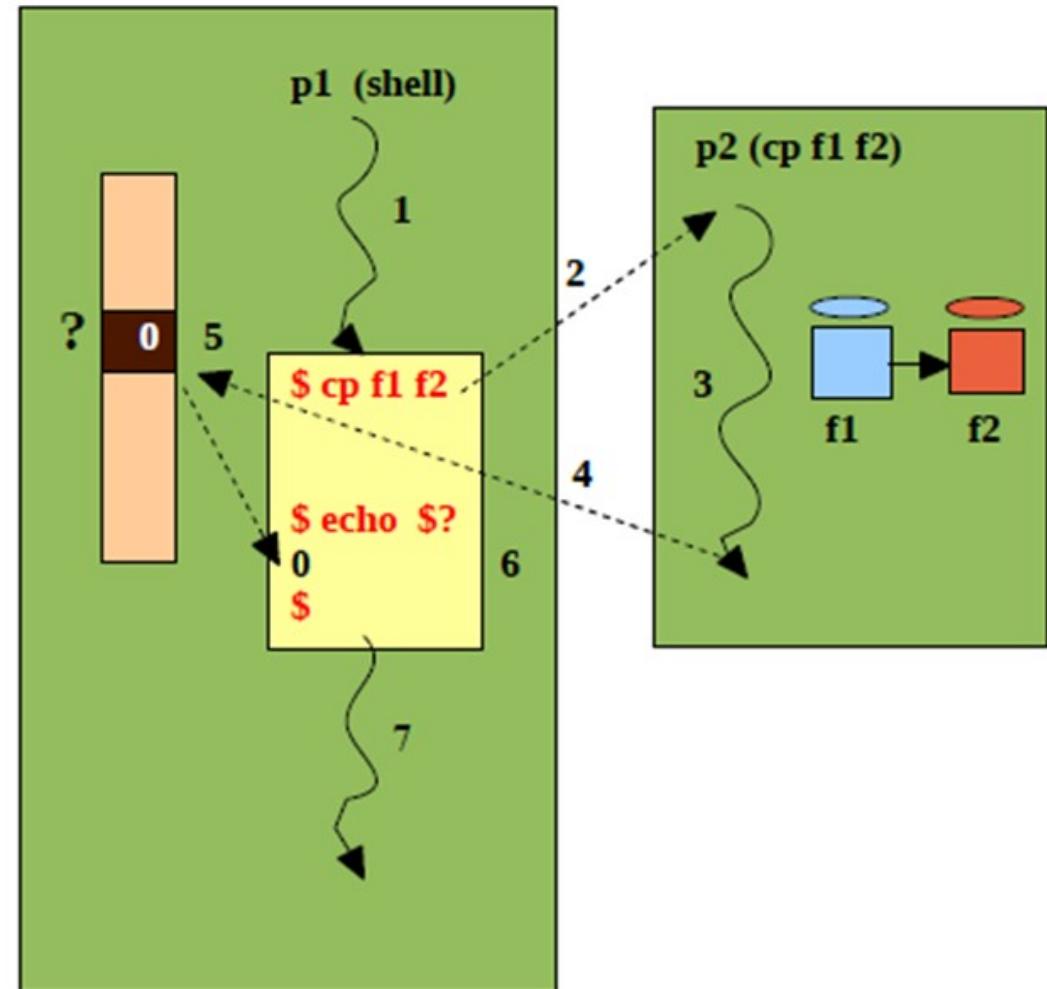
# Cycle de vie des processus : illustration dans le shell

Pour chaque commande lancée\*, le Shell crée automatiquement un nouveau processus et se met en attente.

Exemple :

```
prompt> cp f1 f2  
prompt> echo $?  
0
```

\* : sauf les primitives, qui sont directement intégrées au shell, voir TD...



# Commande shell en premier plan

- C'est le mode par défaut : on entre une commande, on attend la réponse : **le shell nous redonne la main quand le processus est fini.**
- Tant que le processus fils qui correspond à la commande n'est pas fini, le processus père (le shell) est en attente → on ne peut pas utiliser le shell.

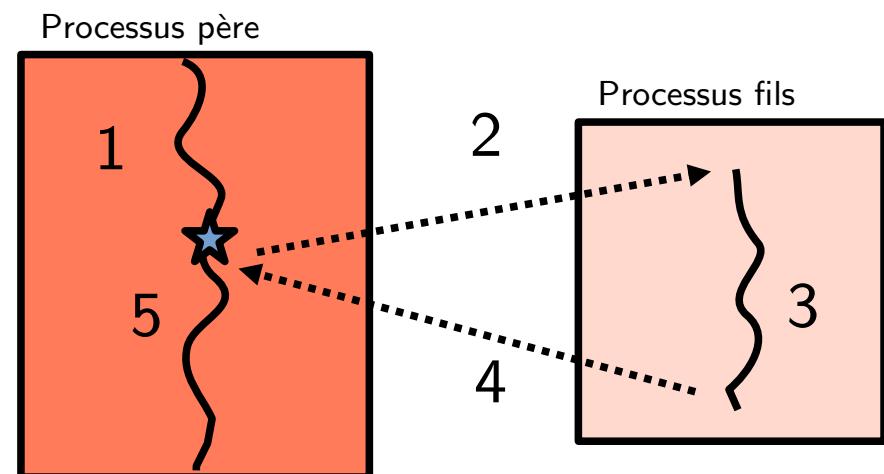
prompt> cmd1

... résultat de la commande cmd1

ça peut être long

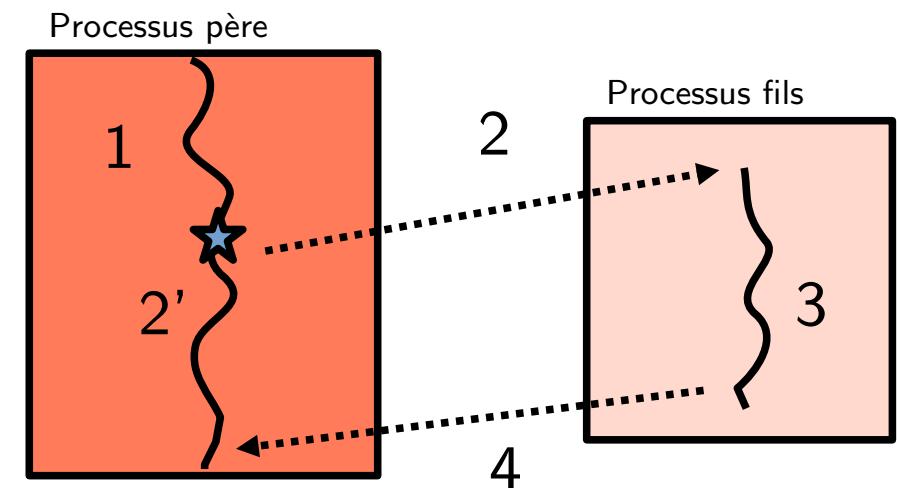
trèèèèèès lonnnnng.

trop long ? (faites Ctrl-C si vous en avez marre)



# Commande shell en arrière plan

On utilisera cette solution (processus lancés en parallèle) par exemple pour lancer un traitement très long, et continuer à travailler en même temps. Dans ce cas, on dit que le père a lancé un fils en **tâche de fond (background)** ou encore en **mode asynchrone**.



# Commande shell en arrière plan

```
prompt> cmd1 & # le nom de la commande suivi de '&'  
[1] 127  
prompt >
```

Le Shell affiche un **numéro de tâche** entre « [ ] » et le **PID** de cette tâche de fond, puis continue à travailler (→ donc affichage de la chaîne d'invite et attente de la prochaine commande).

Pour lancer plusieurs commandes successives (« ; ») en arrière plan :

```
prompt> (cmd1; cmd2) &  
[2] 128  
prompt>
```

→ La commande **cmd2** ne sera lancée que lorsque la commande **cmd1** sera terminée. L'utilisateur.ice récupère la main tout de suite. Le Shell détecte la présence du '**&**' partout sur la ligne.

# Commande shell en arrière plan

- Dans le cas suivant, la commande **cmd1** est lancée en arrière plan et la commande **cmd2** est tout de suite lancée derrière, en direct (en parallèle).

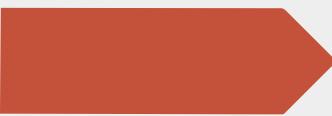
```
prompt> cmd1 & cmd2
[3] 130
résultat commande 2
prompt>
```

- La commande « **wait n** » permet d'attendre la mort de la tâche de fond dont le PID est « **n** »..

```
prompt> cmd1 &
[4] 132
prompt> wait 132 # rester bloqué jusqu'à ce que cmd1 se termine
```

Si « **n** » n'est pas précisée, **wait** attend la mort de toutes les tâches de fond. **wait** ne s'applique qu'aux processus lancés dans le shell lui-même.

- Pour lister les processus lancé dans la session en cours : **jobs**



# **La communication entre les processus**



# Communications inter-processus

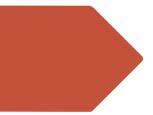
Deux paradigmes :

- par structures de données partagées.

Exemple : via des fichiers en lecture-écriture concurrentes, via des bases de données, ...

- par messages

Exemple : via les signaux et les tubes

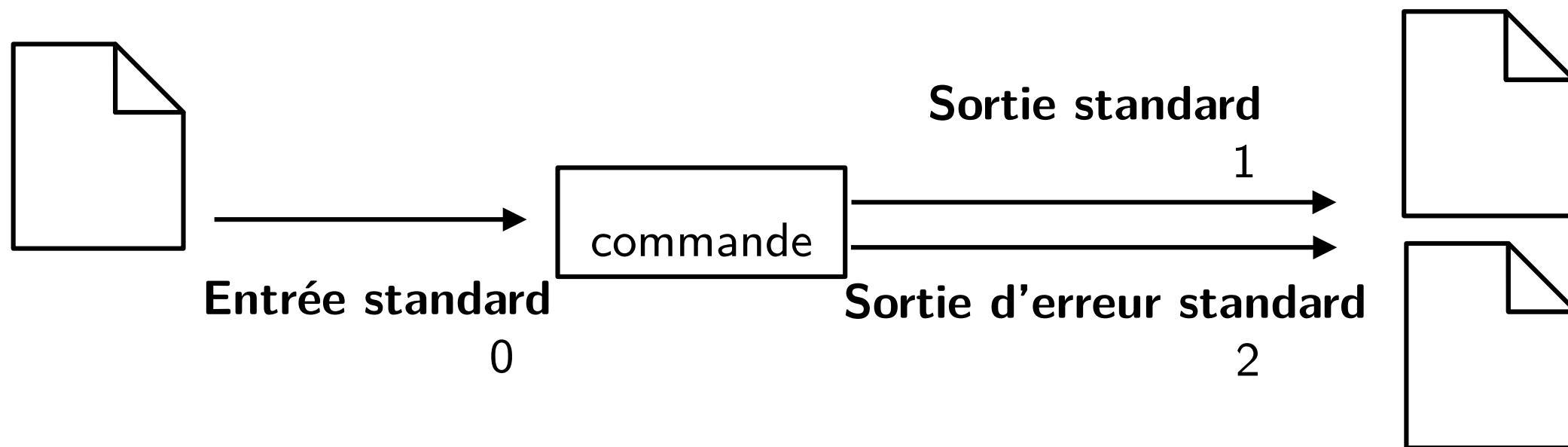


# Communication via signaux

- Les processus communiquent (entre eux et avec le SE) via des signaux
- Liste de signaux : [man7.org/linux/man-pages/man7/signal.7.html](http://man7.org/linux/man-pages/man7/signal.7.html)
  - exit** et **return** : « coucou kernel, j'ai fini » → fin d'exécution normale
  - SIGTERM** : sommation d'interruption
  - SIGINT** : interrompu en douceur (généralement CTRL-C dans le terminal)
  - SIGKILL** : interruption violente
- Dans un processus, un **handler** c'est une fonction qui attrape les signaux et les gère.  
Exception : **SIGKILL** et **SIGSTOP**  
le CTRL-C dans vim ne tue pas le processus

# Communication via pipe

## Rappel sur les entrée / sortie des commandes



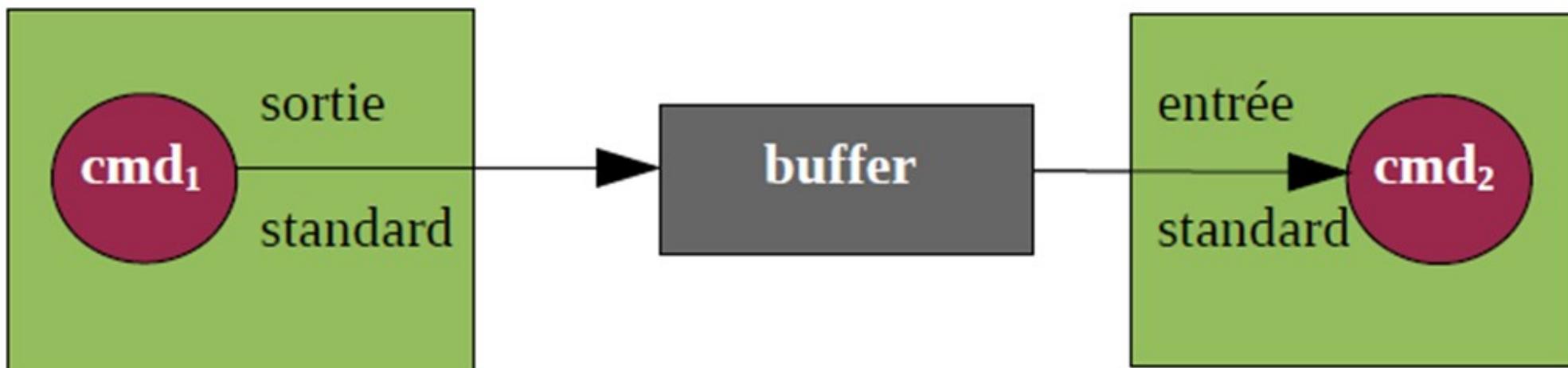
# Communication via pipe

Un pipe (tube) permet de faire communiquer deux processus en pluggant la sortie de A sur l'entrée de B.

Les deux processus s'exécutent en parallèle.

En shell : **commande<sub>1</sub> | commande<sub>2</sub> [ | ... | commande<sub>n</sub>]**

Exemple : **ls | grep toto.txt**



# Communication entre processus via un pipe

```
prompt> who | grep cours
```

cours	ttya4	Jul 31 10:50
cours	ttyc6	Jul 31 09:34
cours	ttya2	Jul 31 09:02

→ La sortie produite par la commande **who** est associée à l'entrée de la commande **grep**. **who** donne la liste des personnes connectées au système à un moment donné ; **grep** cherche si la chaîne cours est présente dans le flot de données qu'elle reçoit. On peut donc considérer que la commande **grep** joue le rôle de filtre.

Le pipe est plus court et compact que :

```
prompt> who > tmp
```

```
prompt> grep cours < tmp
```

cours	ttya4	Jul 31 10:50
cours	ttyc6	Jul 31 09:34
cours	ttya2	Jul 31 09:02

```
prompt> rm tmp # pour ne pas conserver le fichier intermédiaire
```

# Communication entre processus via un pipe

```
prompt> ps -a | wc -l  
9
```

Création de deux processus concurrents. Un tube est créé dans lequel le premier (**ps -a**) écrit ses résultat et le deuxième (**wc -l**) lit.

Lorsque le processus écrivain se termine et que le processus lecteur dans le tube a fini d'y lire (le tube est donc vide et sans lecteur), ce processus détecte une fin de fichier sur son entrée standard et se termine.

Le système assure la **synchronisation de l'ensemble** dans le sens où :

- il bloque le **processus lecteur** du tube lorsque le tube est vide en attendant qu'il se remplisse (s'il y a encore des processus écrivains);
- il bloque (éventuellement) le **processus écrivain** lorsque le tube est plein (si le lecteur est plus lent que l'écrivain et que le volume des résultats à écrire dans le tube est important).

# Communication entre processus via un pipe

## La commande tee

- `cmd1 | cmd2 | cmd3 | .... | cmdn`
- On ne voit pas les résultats intermédiaires
- `tee` : lit dans son entrée standard (0), écrit dans sortie standard (1) et dans un fichier
- Exemple : `ps -l | tee /dev/tty | wc -l`

F S UID PID PPID PRI ... CMD

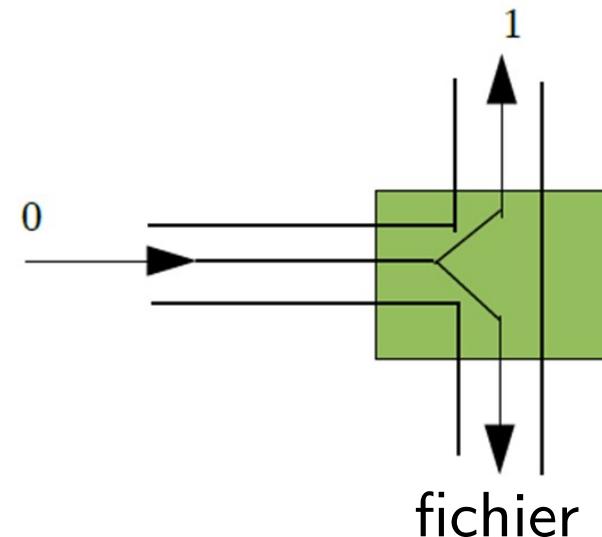
1 S 102 241 234 158 -bash

1 R 102 294 241 179 ps

1 S 102 295 241 154 tee

1 S 102 296 241 154 wc

5





# Communication entre processus via un pipe

## Les filtres

Un **filtre** est une commandes ayant la propriété **à la fois de :**

- lire sur leur entrée standard et
- d'écrire sur leur sortie standard.

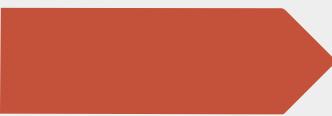
Commandes filtres : **cat, wc, sort, grep, sed, sh, awk, head, tail, ....**

Commande non filtres : **echo, ls, ps...**



- En annexe : liste de commandes utiles pour monitorer et gérer les processus
- En TD : gestion des processus – les bases





## **Annexes : liste de commandes utiles pour la gestion des processus**

## Récupérer le PID de la session shell courante

Le **PID** du shell courant est stocké dans une pseudo-variable spéciale que l'on appelle « `$` ». On peut le consulter grâce à : `echo $$`

Le 1er "`$`" définit le contenu de la pseudo-variable. Le second "`$`" correspond à la variable stockant le PID du Shell courant.

# La commande ps

La commande **ps** permet de visualiser les processus que lancés. Il y a plein d'options possible → **man ps**

```
prompt> echo $$  
527  
prompt> cmd1 &  
prompt>  
prompt> ps  
PID TTY TIME COMMAND  
527 tttyp4 1:70 -ksh  
536 tttyp4 0:30 cmd1  
559 tttyp4 0:00 ps  
prompt>
```

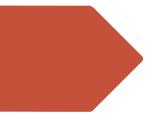
**PID** identifie le processus,  
**TTY** est le numéro du terminal associé,  
**TIME** est le temps cumulé d'exécution du processus,  
**COMMAND** est le nom du fichier correspondant au programme exécuté par le processus.

# La commande ps – les options

Sans option, la commande concerne les processus associés au terminal depuis lequel elle est lancée.

```
ps          # liste des processus du shell courant  
ps -ef      # liste de tous les processus  
ps -ef | grep firefox # Firefox est-il actif ?  
ps -aux     # affiche les ressources utilisées  
ps -u root   # les processus associés à un UID donné
```

L'option « **--forest** » permet de d'afficher en supplément l'arborescence des processus.



# La commande type

**type *commande* ...** donne le chemin absolu du fichier exécuté lorsque vous tapez ***commande***. Sinon, indique que la commande est interne au shell.

**Exemples :**

```
prompt> type find pg
```

```
find is /bin/find
```

```
pg is /usr/bin/pg
```

```
prompt> type umask
```

```
umask is a shell builtin
```

```
umask est une primitive du shell
```



# Monitorer les processus

- Commandes : **top**, **htop**, **jobs**

# La commande top

Affiche **en temps réel** les processus qui consomment le plus de ressources systèmes. Dans les premières lignes, elle affiche des informations globales sur le système (charge, mémoire, nombre de processus, ...).

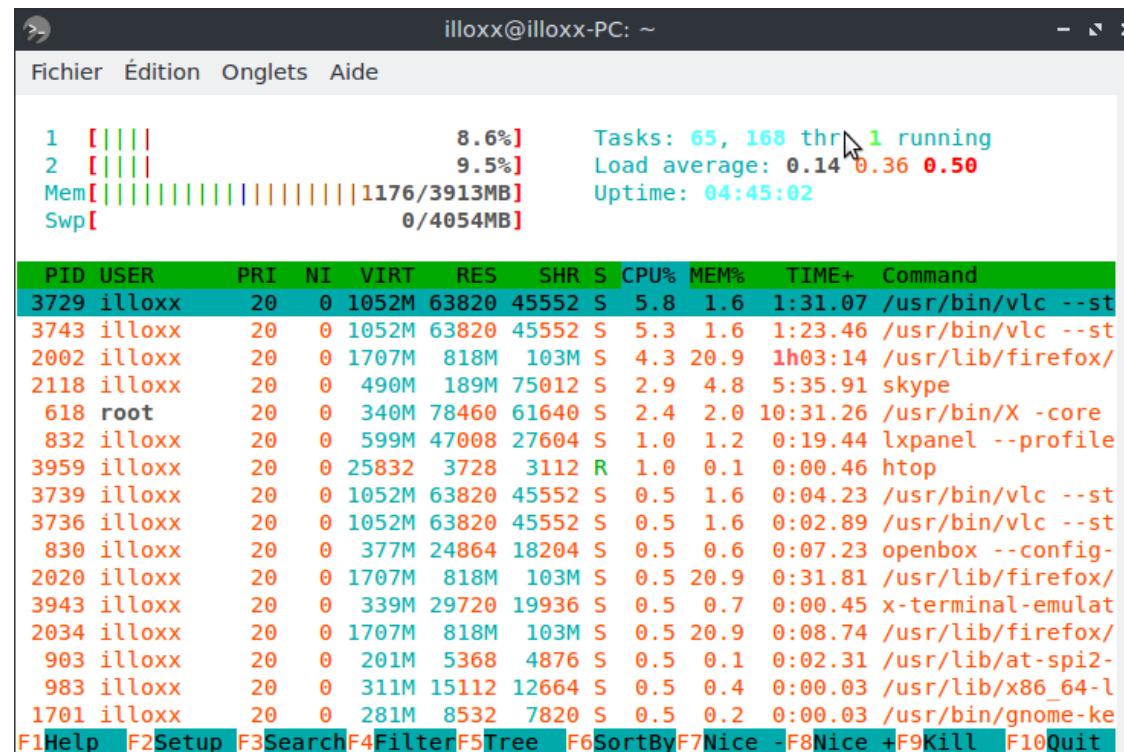
```
top - 11:14:18 up 1:02, 1 user, load average: 0,05, 0,09, 0,08
Tasks: 209 total, 1 running, 208 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,1 us, 0,2 sy, 0,0 ni, 99,8 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
MiB Mem : 7933,1 total, 6381,5 free, 797,0 used, 754,6 buff/cache
MiB Swap: 2048,0 total, 2048,0 free, 0,0 used. 6866,3 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM TIME+ COMMAND
        607 systemd+ 20   0  16004  6300  5472 S  0,3  0,1  0:03.81 systemd+
       1327 jfa     20   0 4053620 269836 129020 S  0,3  3,3  0:37.19 gnome-s+
       2104 jfa     20   0  227344  2432  2072 S  0,3  0,0  0:06.34 VBoxCli+
       2207 jfa     20   0  563848  54248  41684 S  0,3  0,7  0:02.81 gnome-t+
       2718 jfa     20   0 2819336  64620  49212 S  0,3  0,8  0:00.68 gjs
         1 root     20   0  167916 12340  8576 S  0,0  0,2  0:01.75 systemd
         2 root     20   0      0     0      0 S  0,0  0,0  0:00.00 kthreadd
         3 root     0 -20      0     0      0 I  0,0  0,0  0:00.00 rcu_gp
         4 root     0 -20      0     0      0 I  0,0  0,0  0:00.00 rcu_par+
         5 root     0 -20      0     0      0 I  0,0  0,0  0:00.00 netns
         6 root     20   0      0     0      0 I  0,0  0,0  0:01.01 kworker+
         7 root     0 -20      0     0      0 I  0,0  0,0  0:00.00 kworker+
         9 root     0 -20      0     0      0 I  0,0  0,0  0:00.06 kworker+
        10 root     0 -20      0     0      0 I  0,0  0,0  0:00.00 mm_perc+
        11 root     20   0      0     0      0 I  0,0  0,0  0:00.00 rcu_tas+
        12 root     20   0      0     0      0 I  0,0  0,0  0:00.00 rcu_tas+
        13 root     20   0      0     0      0 I  0,0  0,0  0:00.00 rcu_tas+
```

<https://manpages.ubuntu.com/manpages/xenial/fr/man1/top.1.html>

# La commande htop

Similaire à top, mais interface un peu plus évoluée



<https://doc.ubuntu-fr.org/htop>

# La commande jobs

La commande **jobs** est une commande des systèmes d'exploitation Unix et Unix-like pour lister les processus lancés ou suspendus en arrière-plan.

Elle **liste es processus en cours d'exécution ainsi que leur état : running ou stopped ou done.**

**Syntaxe générale** **jobs [option] [jobID]**

**Exemple :**

```
$ nano f1 &
```

```
$ firefox &
```

```
$ jobs
```

```
[1]- Stopped
```

```
nano f1
```

```
[2]+ Running
```

```
firefox &
```

```
$
```

# La commande fg

La commande **fg** est la commande qui permet de remettre un processus au premier plan (**foreground**)

**Syntaxe générale :** **fg [options] %[jobID]**

Le **jobID** est le numéro fournit par la commande **jobs**

**Exemple :**

**\$ jobs**

[1]- Stopped	nano f1
[2]+ Running	firefox &

**\$ fg %2**

→ Affiche la fenêtre Firefox

# La commande bg

La commande **bg** est la commande qui permet de remettre un processus au arrière plan (**background**)

**Syntaxe générale : bg [options] %[jobID]**

Le **jobID** est le numéro fournit par la commande **jobs**

**Exemple :**

```
$ jobs  
[1]- Stopped          nano f1  
[2]+ Running          firefox &  
$ bg %2
```

→ Remet la fenêtre Firefox en arrière plan !



# Les commandes pour tuer des processus

- **kill, killall**

# La commande kill

**kill** envoie un **signal** à un (des) processus ou groupes de processus spécifiés, les obligeant à agir en fonction du signal. Lorsque le signal n'est pas spécifié, il est défini par défaut sur **-15 (-TERM)**.

**Syntaxe générale : kill [option] [PID]**

Les signaux habituellement utilisés :

- 1 (HUP)** - Reload a process.
- 9 (KILL)** - Kill a process.
- 15 (TERM)** - Gracefully stop a process.

**kill -l** liste les signaux disponibles

**Exemple :**

```
$ ps
```

PID	TTY	TIME	CMD
2226	pts/0	00:00:00	bash
3614	pts/0	00:00:00	vi
3617	pts/0	00:00:00	ps

```
$ kill -15 3614
```

```
$ ps
```

PID	TTY	TIME	CMD
2226	pts/0	00:00:00	bash
3635	pts/0	00:00:00	ps
[1]+	Killed	vi toto	

```
$
```

# La commande killall

La commande **killall** est similaire à **kill** mais pour tous les processus qui exécutent une commande spécifique : elle envoie un signal à tous les processus ou groupes de processus dont le nom est spécifié, les obligeant à agir en fonction du signal. Lorsque le signal n'est pas spécifié, il est défini par défaut sur **-15 (-TERM)**.

**Syntaxe générale : killall [option] Processus**

Les signaux les plus communs :

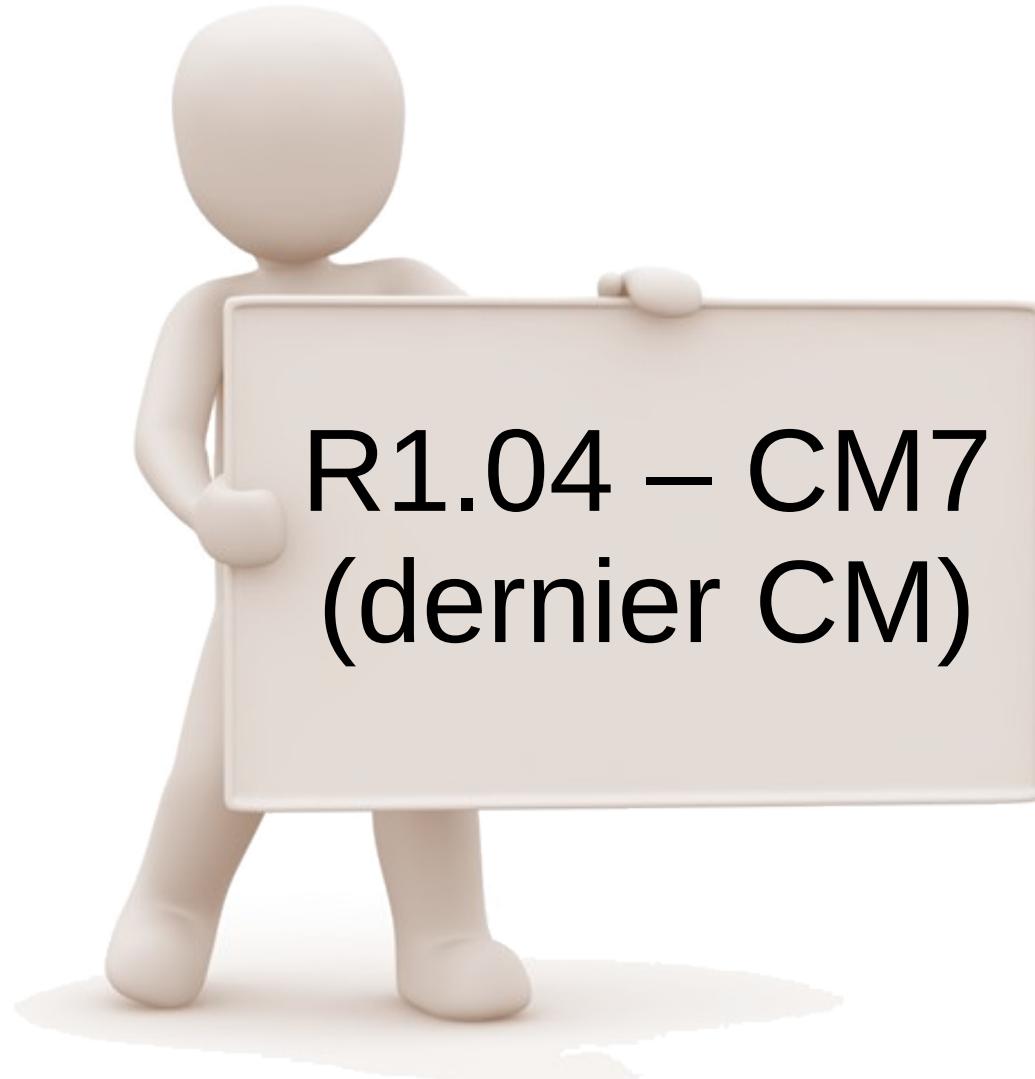
**1 (HUP)** - Reload a process.

**9 (KILL)** - Kill a process.

**15 (TERM)** - Gracefully stop a process.

**killall -l** liste les signaux disponibles





## Menu du jour :

- recap du dernier CM et TD
- installation et mise-à-jour d'un système Linux basé sur Debian
- programmation en shell (bash, pwsh, cmd)



# Recap du dernier CM et TD

- La différence entre un programme et un processus
- La mythologie des processus
- Les successions de pipes
- Exécuter vs sourcer vs remplacer
- Les commandes internes et externes

# Programme, processus, commutation : métaphore

Une informaticienne prépare un gâteau d'anniversaire pour sa fille. Elle a une recette pour faire le gâteau et dispose de farine, d'oeufs, de sucre ...



The image shows the interface of a Wooclap poll. On the left, there is a green circular icon with a globe symbol. To its right, two numbered steps are listed: "1 Allez sur [wooclap.com](https://wooclap.com)" and "2 Entrez le code d'événement dans le bandeau supérieur". On the right side of the poll interface, the event code "MYKUYR" is displayed in large green capital letters, preceded by the text "Code d'événement". At the bottom right of the poll interface, there is a button labeled "Activer les réponses par SMS" with a small SMS icon.

# Programme, processus, commutation : métaphore

Une **informaticienne** prépare un gâteau d'anniversaire pour sa fille. Elle a une **recette** pour faire le gâteau et dispose de farine, d'oeufs, de sucre ...

- La **recette** représente le **programme** (algorithme traduit en une suite d'instructions).
- L'**informaticienne** joue le rôle du **processeur** (CPU)
- Les **ingrédients** sont les **données** à fournir
- Le **processus** est l'activité de notre cordon bleu qui lit la recette, trouve les ingrédients nécessaires et fait cuire le gâteau.

Si le fils de l'informaticienne arrive en pleurant parce qu'il a été piqué par une guêpe, sa mère marque l'endroit où elle en était dans la **recette** (l'état du processus en cours est sauvegardé), cherche un **livre sur les premiers soins** et commence à **soigner son fils**.

Le **processeur** passe donc d'un **processus** (**la cuisine**) à un autre plus prioritaire (**les soins médicaux**), chacun d'eux ayant un **programme** propre (**la recette** et le **livre des soins**).

Lorsque la piqûre de la guêpe aura été soignée, l'**informaticienne** reprendra sa **recette** à l'endroit où elle l'avait abandonnée.



# Recap : la mythologie des processus



The image shows a screenshot of a Wooclap poll interface. On the left, there is a green circular icon with a globe symbol. To its right, two numbered steps are listed:

- 1 Allez sur [wooclap.com](https://wooclap.com)
- 2 Entrez le code d'événement dans le bandeau supérieur

To the right of the steps, the event code "MYKUYR" is displayed in a large, bold, green font. At the bottom right, there is a button labeled "Activer les réponses par SMS" with a small SMS icon.

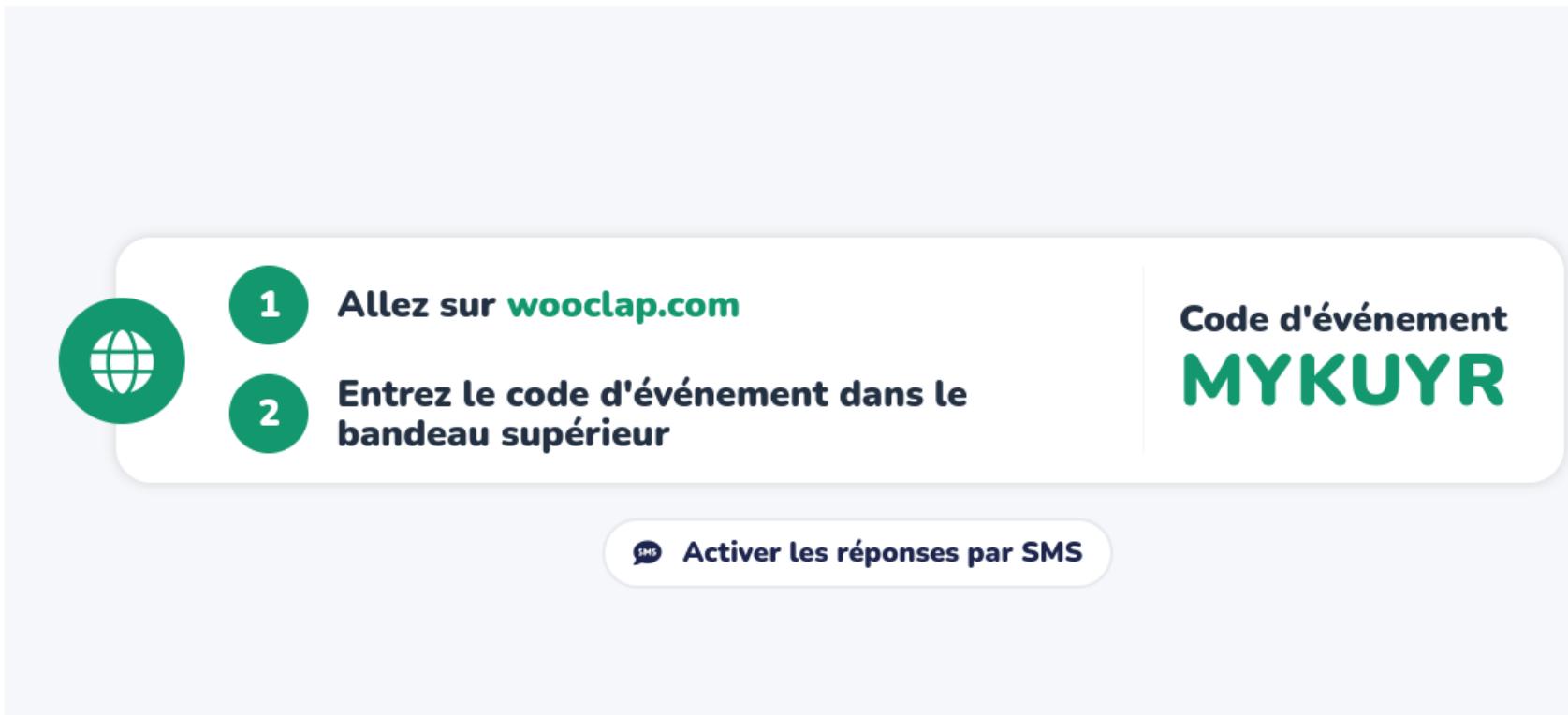
# Recap : la mythologie des processus

-  **Zombie** Achevé, défunt. Doit être traité par son père.
-  **Orphelin** A perdu son père, va être adopté par **init**
-  **Démon** N'est pas créée par l'utilisateur

# Communication entre processus

Suite de tubes pour le filtrage des données :

```
prompt> df -k . | tail -1 | sed "s/ */ /g" | cut -d " " -f 4  
60833156
```



The image shows a screenshot of a Wooclap poll interface. On the left, there are two green circular icons with white numbers 1 and 2. To the right of each icon is a step description. Step 1 says "Allez sur [wooclap.com](https://wooclap.com)". Step 2 says "Entrez le code d'événement dans le bandeau supérieur". To the right of these steps is a large green text box containing the event code "Code d'événement MYKUYR". At the bottom right of the interface is a button labeled "Activer les réponses par SMS" with a small SMS icon.

- 1 Allez sur [wooclap.com](https://wooclap.com)
- 2 Entrez le code d'événement dans le bandeau supérieur

Code d'événement  
**MYKUYR**

Activer les réponses par SMS

# Communication entre processus

- Affichage des statistiques en kilo-octet (option **-k**) sur le répertoire courant (.) :  
prompt> **df -k .**

```
Filesystem 1K-blocks Used Available Use% Mounted on  
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde qu'une seule ligne en partant de la fin :

```
prompt> df -k . | tail -1  
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde qu'un seul espace entre chaque mot :

```
prompt> df -k . | tail -1 | sed "s/ */ /g"  
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde que le quatrième champ de la ligne, l'option « **-d** » précise le séparateur à prendre en compte (l'espace) :

```
prompt> df -k . | tail -1 | sed "s/ */ /g" | cut -d " " -f 4  
60833156
```

⇒ La commande affiche l'espace libre (en kilo-octet) sur la partition qui contient le répertoire de travail.

# Recap : sourcer vs exécuter vs remplacer



The right side of the slide displays the Wooclap poll interface. At the top, there is a green circular icon with a globe symbol. Below it, two numbered steps are listed:

- 1 Allez sur [wooclap.com](https://wooclap.com)
- 2 Entrez le code d'événement dans le bandeau supérieur

To the right of these steps, the event code "MYKUYR" is displayed in a large, bold, green font. Below the code, there is a button labeled "Activer les réponses par SMS" with a small SMS icon.



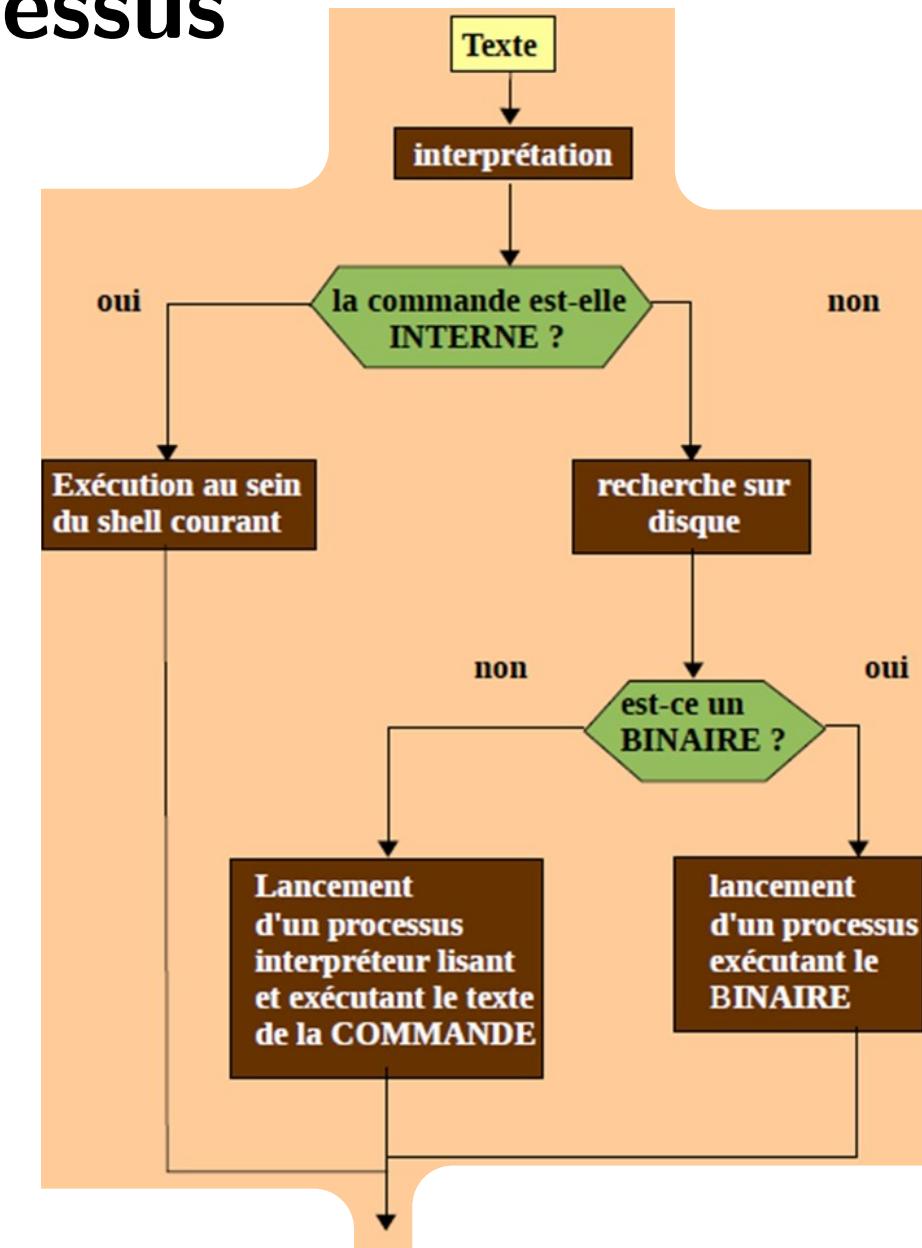
## Recap : sourcer vs exécuter vs remplacer

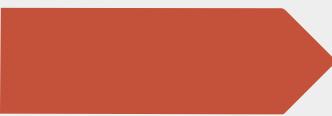
- Sourcer : ne crée pas de nouveau processus (`source f ; . f`)
- Exécuter : crée un nouveau processus (`./f`, `chemin/vers/f`, `f` – si dans le PATH)
- Exec : remplace le processus courant → on ne finira pas les instruction du « père »

# Création ou non de nv processus

Nouveau processus

Oui	Non
<code>./f</code> <code>bash f</code> <code>f</code>	<code>source f,</code> <code>. f</code> <code>exec f</code> (remplace le processus courant)
<b>Commandes externes</b>	<b>Commandes internes</b>





# **Linux : les infos systèmes et la mise à jour**

# Info système : versions Linux

## Coté noyaux :

- convention de numérotation **x.y.z** :
  - **x** : numéro de version.
  - **y** : si pair, désigne une version stable,  
sinon, désigne une version en Bêta-test.
  - **z** : incrémenté à chaque correction de bug.
- Pour connaître la version du noyau en cours :

```
prompt> uname -r  
5.15.0-47-generic
```

## Coté système :

Pour connaître la distribution utilisée :

```
prompt> cat /etc/issue  
Ubuntu 22.04.1 LTS
```

# La commande `uname`

Affiche les informations relatives à la version du système. L'option `-a` (all) affiche toutes les informations.

Exemple :

```
nanis@jammy:~$ uname -a
Linux C302L-G24P07.png.unicaen.fr 6.8.0-45-generic #45~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC
Wed Sep 11 15:25:05 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
```

<https://www.geeksforgeeks.org/uname-command-in-linux-with-examples/>



# Mise à jour du système Linux

- Il faut **mettre régulièrement son système linux à jour**
  - pour corriger des bugs,
  - ajouter des logiciels,
  - supprimer les logiciels obsolètes, ...
- Vocabulaire :
  - Un **paquet** (package) est un fichier qui permet d'installer un logiciel sur une distribution. Le fichier exécute des scripts afin de placer les fichiers (de configuration et de l'application) au bon endroit sur le système.
  - Les paquets sont créés et maintenus par des **mainteneurs**
  - Les paquets sont (souvent) stockés sur des serveurs, appelés **dépôts** (**repository**).  
dépôts officiels vs dépôts tiers (third-party)
  - Un **gestionnaire de paquets** est un outil qui permet de gérer des paquets logiciel d'une distribution depuis les dépôts.



# Mise-à-jour sur les distributions Debian

Distribution tq Ubuntu, Mint, ...

Gestionnaire de paquets : **Advanced Package Tool (APT)**

Paquet : fichiers **.deb**.

- **Installation/ mise-à-jour d'un logiciel :**
  - APT se connecte à l'un repository
  - APT télécharge le .deb
  - APT installe le .deb
  - De manière sous-jacente : commande **dpkg**.
- **Mise à jour de la distribution :**
  - Télécharger tous les .deb de la nouvelle version et les installer.



# APT : fichiers de configuration

- **/etc/apt/sources.list** : stocke les sources avec l'adresse des dépôts
- **/etc/apt/sources.list.d/** : sources additionnels. Ainsi on peut ajouter des dépôts non officielles.
- **/etc/apt/apt.conf** : fichier de configuration APT
- **/etc/apt/apt.conf.d/** : fichiers de configuration additionnels.
- **/etc/apt/preferences.d/** : Les fichiers de préférences additionnels.
- **/var/cache/apt/archives/** : Stocke les deb déjà téléchargés (évite de retélécharger si réinstallation)
- **/var/lib/apt/lists/** : stocke la liste et informations sur les packages du systèmes.



# Les commandes APT

Apt regroupe différentes commandes, selon les besoins.

On modifie la configuration système → ces opérations **nécessite des droits root** :

- s'identifier en root (**su**)
- utiliser **sudo**

**dpkg** : le programme qui installe les fichiers .deb

**apt-\* (install/cache/key)**: programmes qui traquent les paquets disponibles, les téléchargent et les filent à **dpkg**

**apt** : wrapper pour **apt-\*** ← partez du principe que c'est **apt que vous aller utiliser**

**aptitude** : un frontend pour APT



## La commande `sudo apt update`

- Met à jour (resynchronise) l'**indexation** du dépôt sur **vos** Linux.
- En effet, indexation trop ancienne = plus synchronisée avec l'indexation en ligne. Ainsi, vous pouvez demander une version qui n'existe plus et obtenir une erreur.
- Enfin on utilise **apt update** lorsque l'on modifie le sources afin de télécharger les nouveaux index.



## La commande sudo apt upgrade

Met à jour **les paquets** de la distribution Linux de votre machine (pas la version de la distro en elle-même...).

En effet, des mises à jour de sécurité sont publiées chaque jour.

**Lancement** de la commande, **affichage** de la liste des mises à jour (potentiellement très longue si la dernière mise à jour remonte à très longtemps), **validation** de l'utilisateur (**o/y**), **téléchargement** des paquets (la vitesse et le délai s'affichent en bas à droite de l'écran), phase **d'installation**.

apt peut poser des questions sur des actions à effectuer durant la mise à jour.



## Autre commandes apt

Pour supprimer les fichiers qui ne sont anciens ou plus nécessaires.

- **sudo apt autoremove**

supprime les paquets installés dans le but de satisfaire les dépendances d'autres paquets et qui ne sont plus nécessaires.

- **sudo apt clean**

vide le dossier **/var/cache/apt/archives** qui contient les .deb téléchargés.

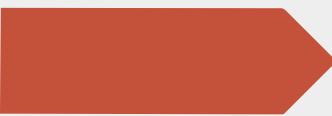
- **sudo apt autoclean**

nettoie le référentiel local des paquets récupérés. La différence avec clean est qu'il supprime uniquement les paquets qui ne peuvent plus être téléchargés et qui sont inutiles.

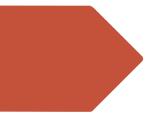


## La commande `sudo apt dist_upgrade`

- Met la **distribution à niveau** (fait passez à la version suivante de votre distribution).
- Ensuite on lance **apt upgrade** ou encore **apt-full upgrade** pour finir la mise à jour,



# **Les scripts shell**



# Définitions et concepts généraux

- **Script** = série de commandes dans un fichiers.
- **Langage interprété** : exécute les instruction directement, sans avoir besoin de compilation. Les langages shell (**sh**, **bash**, **ksh**, **zsh**, **pwsh**) sont interprétés (comme aussi **python**, mais pas comme **C**).
- Les **extensions** sont des *conventions* (contrairement à Windows, ce ne sont pas les extension qui détermine le type de fichiers...).
- **Shebang** : première ligne d'un script, indiquant l'interprète à utiliser « shebang »  
**`#!/bin/bash`**    **`#!/usr/bin/env pwsh`**    **`#!/usr/bin/env python3`**
- **Code de retour** : *par convention*, 0 signifie que tout s'est bien passé. Les valeurs supérieures à 0 représentent différents cas d'erreurs (sémantique à documenter)
- **Syntaxe** : syntaxe variant en fonction du shell (→ utiliser des **cheatsheets**), mais un script qui respecte la norme POSIX est en principe compréhensible par n'importe quel shell...



# Éléments de langages

- Les variables.  
Un contenant nommé ayant une valeur. Doit être **déclarée avant d'être utilisée** (affichage, calcul, ...)  
Type de variable : numérique, chaîne, tableau, ...
- Les structures conditionnelles. **Si/Sinon/Si**
- Les structures itératives.
  - **Pour**
  - **Tant que**
- Les fonctions.

# Exécution d'un script shell

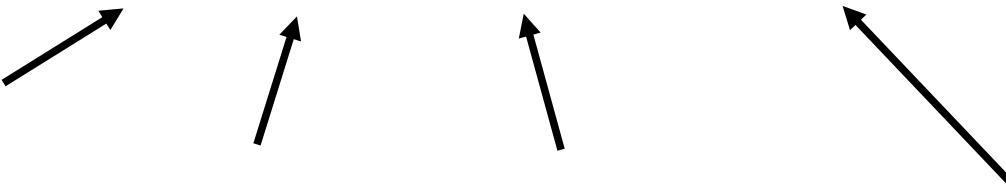
Le shell suit grossièrement les étapes suivantes ; il :

- Lit le script **ligne à ligne**, et **charge l'interpréteur** spécifié par le **shebang**.
- Coupe la ligne en morceaux (**tokens**) et se demande :
  - Quelle est la **commande** ? (fonction / commande intégrée (built-in) / exécutable / script)
  - Quels sont les arguments / options / paramètres ?
- Exécute les **expansions** (\*.**c** transformé en liste de chemins, par exemple).
- Exécute les actions de **redirection** (>**toto** ouvre en lecture le fichier **toto**)
- Exécute la commande ; les arguments sont numérotés de 1 à n.
- Attend que la commande soit finie et récupère le statut en sortie.

# Les scripts bash

- Extension **.sh** par *convention* et shebang : **#!/bin/bash**
- Langage respectant la **norme POSIX**
- Exécution : **bash file**, **/chemin/vers/le/file**, **./file**, **file**, **. file**, **source file**  
/!\ à la **syntaxe**, aux **droits** et au **PATH** (cf TD)
- Code de retour : max 255, par défaut : 0, mots clés **return** ou **exit**
- **Valider ses scripts** pour éviter les erreurs courantes et mauvaises pratiques :

<https://www.shellcheck.net/>





# **Script bash : les variables**

# Script bash : les variables chaîne de caractères

- Déclaration, via le `=`, ou le `(( ))` :

une variable appelée « `s` » qui contient un « `toto` » → type chaîne

```
s='toto' # /!\ pas d'espace !!!!!!!  
((s = "toto" ))
```

- Utilisation en affichage :

```
echo s # affiche « s »
```

```
echo $s # affiche le contenu de s, donc « toto »
```

```
echo "M. $s" # utiliser " et pas ' pour que la variable soit interprétée
```

```
echo "M. \"\$s\" " # /!\ Il faut échapper les guillemets si nécessaire
```

```
echo "tototo${s}tototo" # /!\ Il faut délimiter la variable si nécessaire
```

# Script bash : les variables numériques

Déclaration, via le `=`, ou le `(( ))`, (ou le `let`) :

une variable appelée « `a` » qui contient un `1` → type numérique

```
$ v=1 # /!\ pas d'espace !!!!!!!!
$ (( v = 1 ))
$ let "v=1" # préférer la syntaxe (( )) que le let, voir shellcheck ;)
```

Utilisation en affichage :

```
$ echo $v # affiche (echo) le contenu de la variable v
$ echo "Taille du disque : ${v}To"
$ echo "blablabla $(( v ))"
```

# Script bash : les variables numériques

## Utilisation en calcul :

- Bash ne peut pas faire directement de calculs mathématiques :

```
$ a=2 ; echo $a+2  
$a+2
```

- Utiliser `$(( ))` (ou les mot-clé `let` et `expr`) pour des opérations sur les entiers

Opération prises en charge : +, -, \*, /, \*\* (puissance), % (modulo)

```
$ a=$((2 + 2)) ; echo $a # 4, /!\ pas d'espace pour l'assignement  
$ b=$((a / 3)) ; echo $b # 3 /!\ nombres entiers...
```

- Contraction d'opérations (à l'instar de nombreux langages de prog) :

```
$ a=1 ; ((a+=1)) ; echo $a # affiche 2
```

```
$ a=1 ; echo $((a+=1)) # on utilise le $ pour récupérer l'output directement
```

- Autres possibilités pour effectuer des calculs (pas forcément avec les nombres entiers) : `bc`.

[https://fr.wikibooks.org/wiki/Programmation\\_Bash/Calculs](https://fr.wikibooks.org/wiki/Programmation_Bash/Calculs), <https://github.com/koalaman/shellcheck/wiki/SC2219>



# Script bash : les variables tableaux

- **Déclaration :**

```
t=("un et " "deux et " "trois et " "quatre")
```

- **Utilisation :**

```
echo ${t} # affiche juste le premier élément...
```

```
echo ${t[@]} # affiche tout le tableau
```

```
echo ${t[0]}
```

```
echo ${t[1]}
```

- **Redéfinition :**

```
t[1]="toto"
```

# Script bash : les trois types de quotes

- Simples quote « ' ' » : contenu pas analysé et traité de façon brute

```
$ echo 'je m\'appelle $prenom.'
```

Je m'appelle \${prenom}.

- Doubles quotes « " " » : le contenu est analysé et traité (pour \$, \ et `).

```
$ echo "je m'appelle $prenom."
```

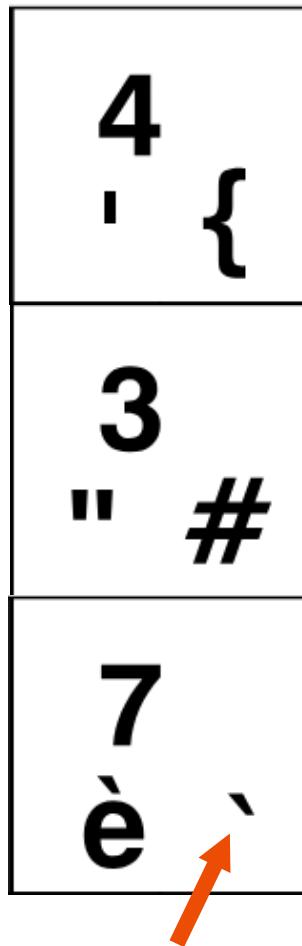
Je m'appelle Athénaïs

- Back quotes « ` ` » : utilisation du résultat (sortie standard) de l'exécution de la commande entre backquotes

Alternative : \$( ) (à préférer aux « ` ` », en fait...)

```
$ echo "Vous êtes sur `uname`"
```

```
$ echo "Vous êtes sur $(uname)"
```



# Script bash : les arguments

Quand on lance un script avec des arguments *scriptname arg1 arg2 arg3*, le shell fait les assignments de **variables spéciales** : *1=arg1, 2=arg2, 3=arg3*

Aussi :

- *0=scriptname*
- *#* : nombre d'arguments du script (sans compter *\$0*)
- *@* : liste des arguments du script
- *?* : code de retour de la dernière commande

# Script bash : les arguments

Exemple :

```
--- dans un script :  
if test "$#" -eq "0" then  
    echo "usage: $0 arg1 arg2" >&2 exit 1  
fi  
---
```

→ Si le nombre d'arguments passés à l'appel de la commande vaut 0, on quitte la commande avec un message d'erreur sur la sortie standard (sortie 1 recopiée sur la sortie 2).

# Script bash : saisie de l'utilisateur.ice

La commande **read** permet de récupérer la saisie de l'utilisateur.ice

```
$ read nom # Demande à l'utilisateur.ice de saisir une valeur  
$ echo "Bonjour ${nom}!"  
  
$ read nom prenom # Saisir plusieurs variables d'affilée  
$ echo "Bonjour $prenom $nom!"
```

**read** lit **mot par mot** (le séparateur est l'espace). Chaque mot est affecté aux variables spécifiées dans l'ordre où elles sont données. La dernière variable récupère tous les mots restants s'il y en a plus que spécifié.

## Quelques options :

- **-p** : Affiche en plus un message pour l'utilisateur.  
`read -p 'Entrez votre nom : ' nom  
echo "Bonjour $nom!"`
- **-s** : Masque le texte saisi (pratique pour un mot de passe, par exemple)
- **-t s** : Renvoie une valeur vide dans la variable au bout de **s** secondes.  
`read -t 15 -p 'Entrez votre nom dans les 15 secondes qui suivent : ' nom`



## **Script bash : les tests**

# Les tests : syntaxes POSIX

## Syntaxes :

- Avec la commande « **test condition** » : `test "${login}" = "toto"`
- Avec les « [ **condition** ] » : `[ "${login}" = "toto" ] /!\` [ ] ``

## Retour :

- 0 si le test est vrai,
- 1 si le test est faux,
- 2 ou plus si erreur.

### sur nombres

**N1 -eq N2** : Vrai si les nombres sont égaux (equal, `==`)

**N1 -ne N2** : Vrai si les nombres sont différents (not equal, `!=`)

**N1 -lt N2** (less than, `<`)

**N1 -le N2** (less equal, `<=`)

**N1 -gt N2** (greater than, `>`)

**N1 -ge N2** :(greater equal , `>=`)

### sur fichiers

**-e FICHIER** : Le fichier existe

**-f FICHIER** : C'est un fichier ordinaire

**-d FICHIER** : C'est un répertoire

**-L FICHIER** : C'est un lien symbolique

**-r FICHIER** : Le fichier est lisible (read)

**-w FICHIER** : Fichier modifiable (write)

**-x FICHIER** : Fichier exécutable (execute)

**FICHIER1 -nt FICHIER2** : F1 plus récent que F2

**FICHIER1 -ot FICHIER2** : F1 plus ancien que F2

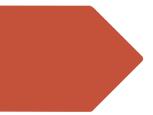
### sur chaînes de caractère :

`=, !=, <, etc.`

**-z CHAINE** : Vrai si la chaine est vide

**-n CHAINE** : Vrai si la chaine n'est pas vide

Exemple : `[ "${nom}" = "prenom" ]`



# Les tests : exemples simples

```
prompt> ls -l
```

```
drwxr-x--- 11 c1 cours 17 Aug 1 09:00 save  
-rw-r----- 1 fun axis 21 Jul 25 17:05 data
```

```
prompt> who am i
```

```
c1 term/c4 Aug 2 09:01
```

```
prompt> test -f save; echo $?
```

```
1
```

```
prompt> test -d save; echo $?
```

```
0
```

```
prompt> test -r save; echo $?
```

```
0
```

```
prompt> test -f data; echo $?
```

```
0
```

```
prompt> test -w data; echo $?
```

```
1
```

# Les tests : combinaisons

Avec **test** ou **en dehors** des [ ] :

- « Et » logique : **&&**  
**test EXPR1 && test EXPR2**  
**[EXPR1 && EXP2]**
- « Ou » logique : **||**  
**test EXPR1 || test EXPR2**  
**[EXPR1 || EXPR2]**

Directement à l'intérieur des [ ] :

- « Et » logique : **-a** : **[EXPR1 -a EXPR2]**
- « Ou » logique : **-o** : **[EXPR1 -o EXPR2]**
- Inverser un test : **!** : **[ ! EXPR]**

---

Combinaisons avancées :

- avec { }  
**if [ "true" ] || { [ -e /does/not/exist ] && [ -e /does/not/exist ] ;} ; then echo true; else echo false; fi**
- avec ( ) : il faut **les échapper** ou les mettre entre '' → **pas très lisible !**  
**if [ \("true" -o -e /does/not/exist \) -a -e /does/not/exist ]; then echo true; else echo false; fi**  
**if [ '(' "true" -o -e /does/not/exist ')' -a -e /does/not/exist ]; then echo true; else echo false; fi**

# Les tests : syntaxes alternatives

[ *condition* ] et test *condition* sont équivalents (et **POSIX**, donc portable ! :D)

[[ *condition* ]] : amélioration de [ *condition* ]. Dispo en ksh, bash, zsh...

((*condition*)) : utilise l'expansion arithmétique. Dispo en ksh, bash, zsh.

Discussion sur les différences entre [ ] et [[ ]] : <https://forum.ubuntu-fr.org/viewtopic.php?id=398332>

<https://unix.stackexchange.com/questions/306111/what-is-the-difference-between-the-bash-operators-vs-vs-vs>



## **Script bash : les structures de contrôle**

- Les conditions
- Les boucles
  - Tant que
  - Pour



# Les conditions

**si / si-sinon / si - sinon si - [sinon si ...] - sinon**

**if *condition* ; then**

***instructions***

**elif *condition* then # autant de elif qu'on veut**

***instructions***

**else # max un else par if, et en dernier !**

***instructions***

**fi**



# Les conditions : exemple de si-sinon

```
----  
if test -d "$1" -a -x "$1" then  
    echo chemin accessible  
    cd $1  
else  
    echo chemin inaccessible  
fi  
----
```

→ **Si** la valeur du premier argument est un répertoire **et** que l'on est autorisé à se déplacer dedans (**-x**), **alors** on y va. **Sinon** on affiche un message.

# Les conditions : un cas pratique

Consigne : Vérifier si une variable est un nombre :

```
[ $mavariable -eq 1 ] 2> /dev/null  
if [ $? -eq 0 -o $? -eq 1 ]  
then  
    echo "C'est un nombre."  
  
else  
    echo "Ce n'est pas un nombre."  
  
fi
```

## Explications :

[ \$mavariable -eq 1 ] 2> /dev/null

Compare la valeur de la variable au nombre « 1 ». Trois cas pour le code de retour du test :

- 0 : La valeur de la variable est égale à 1, donc elle vaut 1.
- 1 : La valeur de la variable est différente de 1, mais la comparaison s'est bien réalisée → la valeur de la variable est un nombre != 1
- 2 ou + : La comparaison a échoué → la valeur de la variable n'est pas un nombre.

if [ \$? -eq 0 -o \$? -eq 1 ]

si le code de retour est 0 ou 1, on a un nombre.  
sinon, ce n'est pas un nombre.

/!\ Il faut impérativement utiliser le « -o » et non « || ».

if [ \$? -eq 0 ] || [ \$? -eq 1 ] ne fonctionne pas car on effectue deux tests différents :

- Au premier test, \$? contient le code de retour du test de la variable,
- Au 2nd test, \$? contient le code de retour du 1er test du « if » actuel. On ne testerait donc pas le bon code retour.

# Conditions et combinaison de commandes - ET

Exécuter *cmd2* uniquement si la commande *cmd1* se termine correctement : *cmd1 && cmd2*

Exemple : S'il existe un répertoire *tmp* dans le répertoire courant, alors aller dans ce répertoire.

```
$ pwd  
/home/c1  
$ mkdir tmp  
$ test -d $HOME/tmp && cd $HOME/tmp  
$ pwd  
/home/c1/tmp  
  
$ cd  
$ rmdir tmp  
$ test -d $HOME/tmp && cd $HOME/tmp  
$ pwd  
/home/c1
```

En utilisant la structure de contrôle **if ... then ...fi** :

```
$ pwd  
/home/c1  
$ mkdir tmp  
$ if test -d $HOME/tmp  
> then cd tmp  
> fi  
$ pwd  
/home/c1/tmp  
  
$ cd  
$ rmdir tmp  
$ if test -d $HOME/tmp  
> then cd tmp  
> fi  
$ pwd  
/home/c1
```

# Conditions et combinaison de commandes - OU

Exécuter *cmd2* uniquement si la commande *cmd1* ne se termine correctement : *cmd1* || *cmd2*

Exemple : S'il n'existe pas de répertoire *tmp* dans le répertoire courant, alors afficher un message.

```
$ pwd  
/home/c1  
$ mkdir tmp  
$ test -d $HOME/tmp || echo $HOME/tmp inexistant
```

```
$ rmdir tmp  
$ test -d $HOME/tmp || echo $HOME/tmp inexistant  
/home/c1/tmp inexistant
```

En utilisant la structure de contrôle **if ... then ...fi** :

```
$ pwd  
/usr/c1  
$ mkdir tmp  
$ if test ! -d $HOME/tmp  
> then  
>   echo $HOME/tmp inexistant  
> fi
```

```
$ rmdir tmp  
$ if test ! -d $HOME/tmp  
> then  
>   echo $HOME/tmp inexistant  
> fi  
/usr/c1/tmp inexistant
```

# Les conditions : enchainements de cas avec `case`

```
case $nom in
    "Athénaïs") # un cas se termine par )
        echo "Salut Athénaïs !" # Tout le code s'exécute jusqu'au prochain ;;
    ;;
    "Jean-François")
        echo "Bonjour JFA"
    ;;
    M*) # utilisation, de jokers. Ici : tous les noms commençants par « M » ; on fait rien
    ;;
    "Riri" | "Fifi" | "Loulou") # on combine plusieurs valeurs
        echo "Bonjour neuveau"
    ;;
    *) # cas par défaut, si aucun autre test ne valide la valeur de la variable.
        echo "Hein, mais t'es qui ?"
    ;;
esac
```

# Les boucles tant que

## Syntaxe :

```
while condition ; do  
    instructions  
done
```

## Exemple :

```
#!/bin/bash  
while [ -z "$reponse" ] || [ "$reponse" != 'oui' ]  
do  
    read -p 'Dites oui : ' reponse  
done
```

Les conditions sont les mêmes que pour les if



# Les boucles for

Syntaxe générale :

```
for name [ in [word ... ]]
```

```
do
```

```
    instruction
```

```
done
```

```
for v in 'var1' 'val2' 'val3' ; do
```

```
    echo $v
```

```
done
```

```
for i in $@ ; do
```

```
    echo $i
```

```
done
```

```
for (( i=0 ; i<10 ; ++i )) ; do
```

```
    echo $i
```

```
done
```

Pour parcourir le résultat d'une commande :

```
liste_fichiers=`ls`  
for fichier in $liste_fichiers ; do  
    echo "Fichier trouvé : $fichier"  
done
```

Pour renommer les fichiers d'un répertoire :

```
for fichier in `ls` ; do  
    mv $fichier $fichier-old  
Done
```

La commande **seq x y** génère tous les nombres allant de **x** à **y**

```
for i in $(seq 1 10); do  
    echo "n°$i"  
done
```

**Alternative** (à partir de la version 3 de bash) : **{x..y}**

```
for i in {1..15}; do  
    echo "n°$i"  
done
```

# Les fonctions

Déclaration AVANT le premier appel :

```
nomfonction() # les paramètre ne sont pas déclaré
{
    instructions
    return x # ou exit (facultatif, par défaut 0)
}
```

Appel :

```
nomfonction param_1 param_2 ... param_n
```

Remarques :

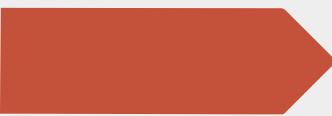
Les fonctions sont similaires aux scripts.

- Les paramètres sont facultatifs.
- Accès au nombre de paramètres transmis via \$#
- Accès au paramètre transmis via \$0, \$1, \$2, ...
- \$0 est le nom de la fonction

Exemples :

```
show() {
    echo 'Hello $1'
    return 5
}
show toto # Hello toto
```

```
isFruitRouge(){
if [ "$1" == "fraise"
-o "$1" == "framboise"
-o "$1" == "groseille" ]
then
    return 0 # Exécution correcte (ou vrai, en l'occurrence)
else
    return 1 # Exécution incorrecte (ou faux, en l'occurrence)
fi
}
```



# **Powershell et cmd**

# PowerShell (et CMD)

**CMD** initialement pour DOS (l'ancêtre de Windows NT)

Extension : .bat ou .cmd

**Powershell** : le successeur, depuis 2006. **Orienté objet** (comme python et javascript)

Extension : .ps1

Commandes de la forme **prefixe-objet**

- exemples de préfixes : **get, set, add, clear, import, export, new, write**
- exemples d'objet : **command, item, content, ...**

**Compatible POSIX** → **cd, ls, help, mkdir**, les redirections, les pipes

Tourne sur Windows (nativement...) mais aussi sur Linux ! ;)

<https://learn.microsoft.com/fr-fr/powershell/scripting/overview?view=powershell-7.4>

<https://simvil.github.io/files/noSQL/CM-4-powershell.pdf>



TD 7 et 8 :  
programmation en shell

4 TP :

- shell (bash)
- windows CMD x2
- powershell









