Initiation à l'environnement Unix
CM1 : Historique et fichiers

Pierre Rousselin

Université Paris 13
L1 informatique
septembre 2021

Le système Unix

Ken Thompson et Dennis Ritchie (assis) travaillant sur un PDP-11

Par Peter Hamer, téléversé par Magnus Manske, CC BY-SA 2.0,
https://commons.wikimedia.org/w/index.php?curid=24512134

Système d'exploitation

Unix est un *système d'exploitation* (en anglais *operating system* ou O.S.). Un (noyau de) système d'exploitation est un programme :

- qui se charge tout seul au démarrage de la machine ;
- qui gère entièrement l'accès au *matériel* :
  - processeur et mémoire vive
  - autres composants de calcul (GPU, ...)
  - périphériques d'entrée : clavier, souris, tablette graphique, microphone, webcam, ...
  - périphériques de sortie : écran, imprimante, enceintes, ...
  - connexions réseau (à la fois entrée et sortie) : par ethernet, wifi, ...
- qui charge les autres programmes en mémoire ;
- et leur donne la possibilité de lui demander gentiment l'accès à une ressource matérielle : ce sont les *appels système*.

Organisation du système Unix

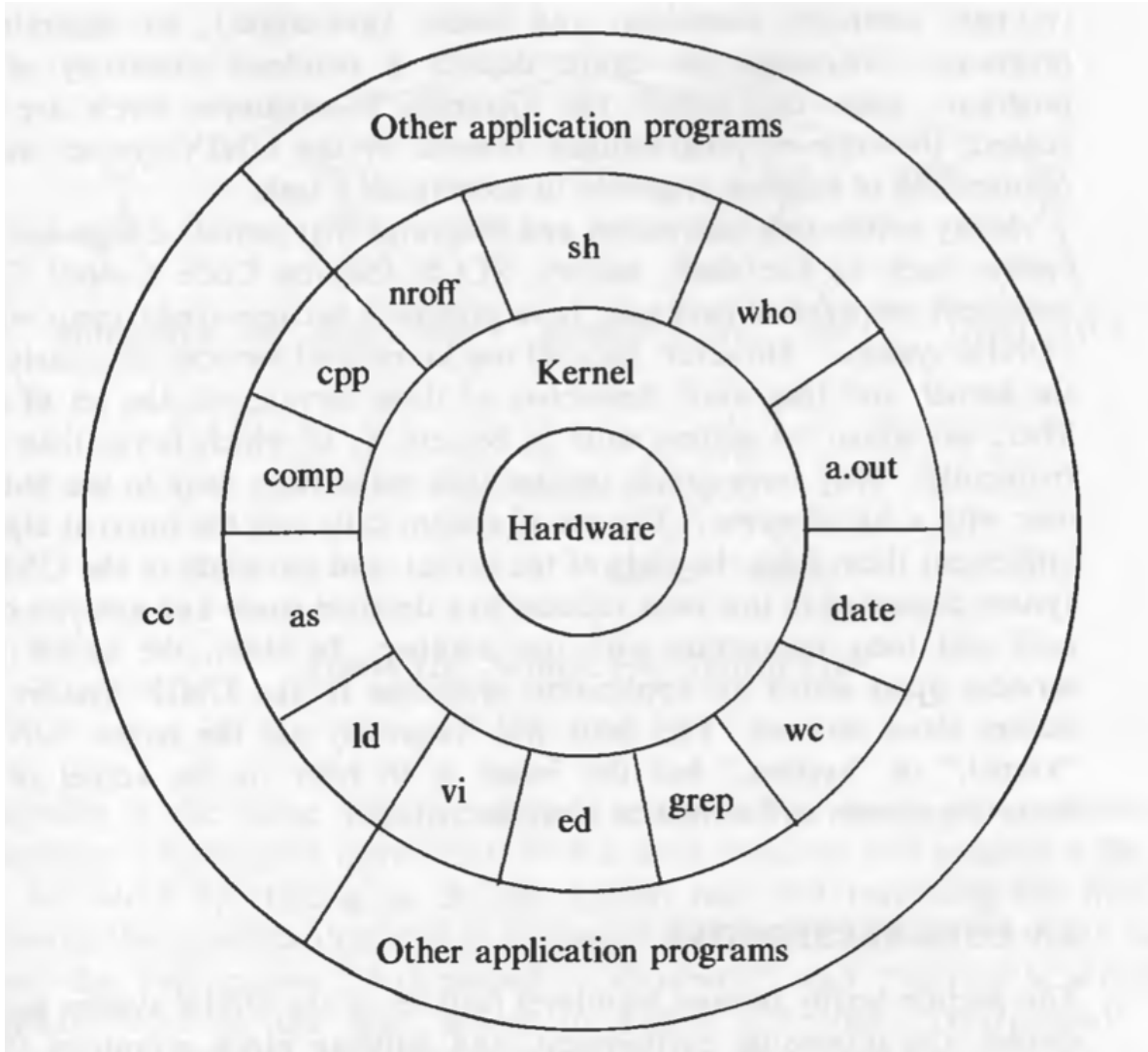


FIGURE – Source : Design of the Unix Operating System, Maurice Bach, 1986

Système Unix

- Multi-utilisateur
- L'utilisateur communique avec l'ordinateur par le biais d'un *terminal*.
- Multitâche (illusion que plusieurs programme tournent en même temps), à *temps partagé* (*time-sharing*)
- Système de fichiers et permissions
- Concept de processus : programme en cours d'exécution
- Les fichiers ont une « place ».
- Les processus ont une « vie ».
- + un environnement de développement à la fois minimaliste et complet (compilateur C, shell, utilitaires, ...)

Bref historique

1964 Système d'exploitation Multics (*MULTiplexed Information and Computing Service*) développé par le MIT, les laboratoires Bell (AT&T) et General Electrics

1969 Les laboratoires Bell quittent le projet Multics qui met trop longtemps à aboutir et Ken Thompson commence à travailler sur un nouveau système appelé Unics puis Unix.

1973 Unix est réécrit en langage C, inventé pour l'occasion par Dennis Ritchie et devient portable.

1974 L'université de Berkeley obtient un système Unix et commence à le modifier (début de BSD pour *Berkeley Software Distribution*).

1977 Steve Bourne (laboratoires Bell) crée le *Bourne shell* qui sera le shell par défaut de tous les Unix développés par les laboratoires Bell.

Bref historique (2)

1983 Unix System V est commercialisé par AT&T

1983 Richard Stallman lance le projet GNU (*GNU is Not Unix*) dont le but est d'écrire un système d'exploitation entièrement libre et compatible avec Unix.

1987 GNU C compiler (`gcc`, qui deviendra ensuite GNU Compiler Collection)

1989 Bourne again shell (`bash`) pour le projet GNU

1988 Débuts du standard POSIX pour uniformiser les nombreux Unix

1991 Linus Torvalds commence à travailler sur ce qui deviendra le noyau *Linux*.

1993 L'*Open Group*, en charge de la standardisation POSIX acquiert les droits sur le nom Unix et définit ce qui est ou non un système Unix.

La famille des Unix

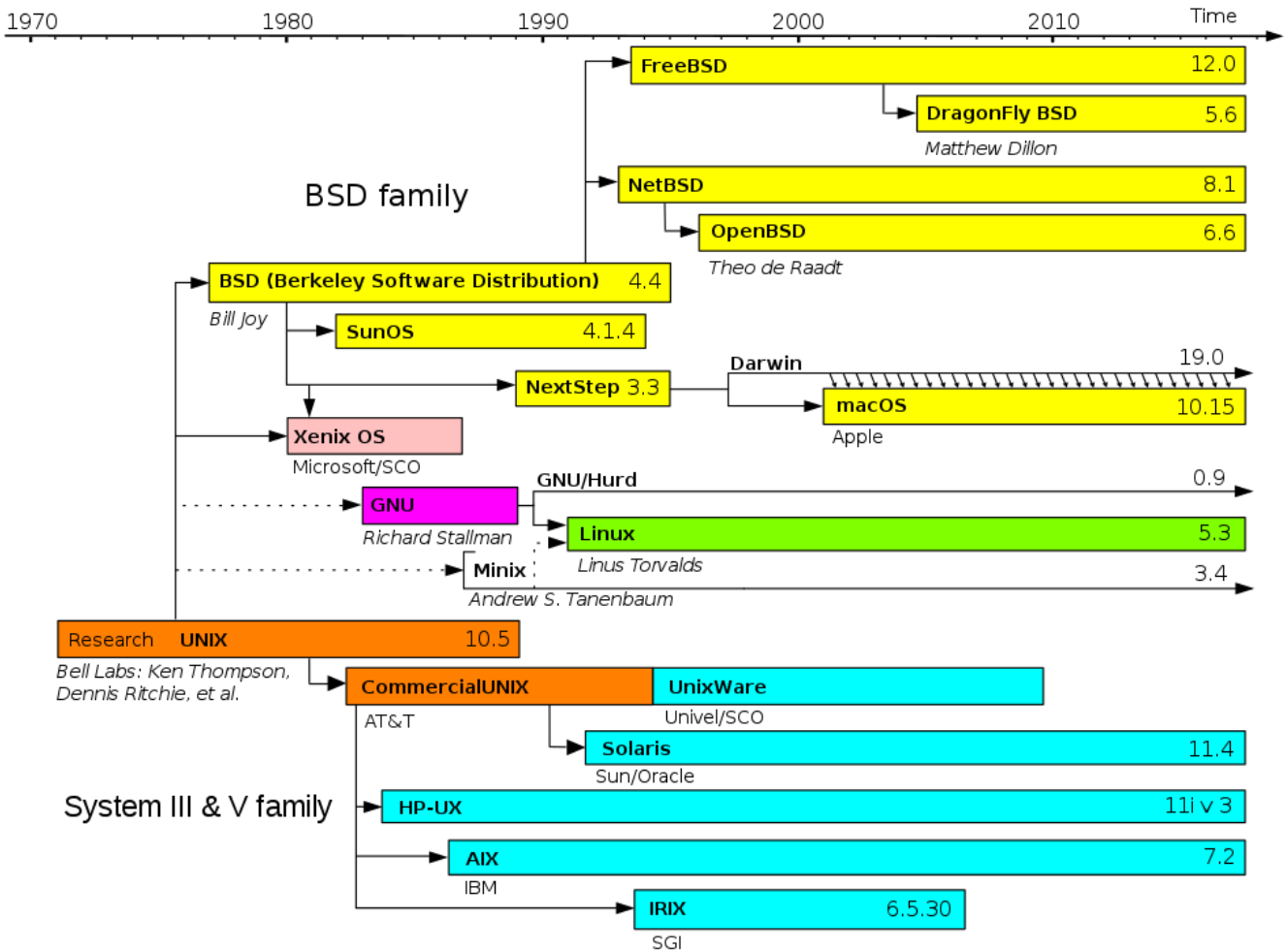


FIGURE – Guillem, Wereon, Hotmocha (copied from old version's history)Christoph S. (redrew the image with Inkscape), Public domain, via Wikimedia Commons

Unix aujourd'hui

GNU/Linux est un système Unix qui équipe la quasi-totalité des supercalculateurs, une part importante des serveurs et entre 1 et 2% des ordinateurs individuels.

FreeBSD est un système d'exploitation de type Unix, dont dérivent, par exemple, MacOS et le système d'exploitation des PlayStation 3 et 4 (5 ?)

Android équipe plus de 80% des smartphones et est basé sur le noyau Linux.

MacOS et iOS sont présents sur les ordinateurs personnels et les smartphones de la marque Apple.

Le standard POSIX

- Devant la multiplication des différents Unix, il était devenu nécessaire de *standardiser* les interfaces de ces différents systèmes d'exploitation pour pouvoir écrire des programmes (en C ou en shell par exemple) qui seraient *portables* sur les différents systèmes Unix.
- POSIX : *Portable Operating System Interface* + le X pour rappeler la filiation avec Unix.
- Standard public et mis à jour par un groupe totalement ouvert regroupant des industriels et des enthousiastes.
- La section *Shell and Utilities* du standard définit ce qu'est un shell Unix et décrit les commandes standard (comme `ls`, `cp`, etc).
- Le standard est plus ou moins bien respecté...

`https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/`

Le terminal vidéo VT 100

Par Jason Scott - Flickr : IMG_9976, CC BY 2.0,

https://commons.wikimedia.org/w/index.php?curid=29457452

Interface

« Une interface est la couche limite par laquelle ont lieu les échanges et les interactions entre deux éléments. »(Wikipedia)
Interface physique :

- Téléscripteur, en anglais *teletypewriter* ou *teletype* (`tty` sous UNIX)
- Terminal vidéo (par exemple le VT100)
- Écran, clavier, souris
- Écran tactile

Interface logicielle :

- shell (interpréteur de commandes)
- interface graphique
- interface graphique des smarphones

Interface

« Une interface est la couche limite par laquelle ont lieu les échanges et les interactions entre deux éléments. »(Wikipedia)
Interface physique :

- Téléscripteur, en anglais *teletypewriter* ou *teletype* (`tty` sous UNIX)
- Terminal vidéo (par exemple le VT100)
- Écran, clavier, souris
- Écran tactile

Interface logicielle :

- shell (interpréteur de commandes)
- interface graphique
- interface graphique des smarphones

Le shell est une interface logicielle textuelle entre l'utilisateur et le noyau.

Le shell aujourd'hui

Il est souvent plus rapide et plus facile de communiquer avec une machine par le biais d'un shell plutôt qu'en utilisant une interface graphique.

- *Bourne shell* (`sh`), écrit par Steven Bourne dans les laboratoires Bell, shell par défaut d'Unix à partir de 1977
- *Korn shell* (`ksh`, `ksh88`, `ksh93`), écrit par David Korn (laboratoires Bell) s'inspire du *Bourne shell* et l'étend
- *C shell* (`csh`), écrit par Bill Joy à l'université de Berkeley, très différent, et son descendant le `tcsh`
- *Bourne Again Shell* (`bash`) du projet GNU
- *Z shell* (`zsh`), écrit par Paul Falstad (université de Princeton), reprend une partie de tous les shells précédemment cités
- *Debian Almquist Shell* (`dash`) : très proche du standard POSIX, minimaliste
- ...

Dans ce cours

- En mode intéractif, on utilisera le shell `bash`.
- Mais les scripts respecteront le standard POSIX car :
  - plus petit (moins de choses à apprendre, donc on peut mieux les apprendre)
  - standard donc plus portable (devrait fonctionner avec tous les shells qui respectent le standard)
  - standard donc bien documenté

Dans ce cours

- En mode intéractif, on utilisera le shell `bash`.
- Mais les scripts respecteront le standard POSIX car :
  - plus petit (moins de choses à apprendre, donc on peut mieux les apprendre)
  - standard donc plus portable (devrait fonctionner avec tous les shells qui respectent le standard)
  - standard donc bien documenté

Dans les salles de TP de l'institut Galilée :

- le système est une distribution GNU/Linux appelée *Debian* ;
- l'environnement de bureau est *Gnome* ou *Xfce* ;
- l'émulateur de terminal par défaut est *gnome-terminal* ;
- le shell interactif par défaut est *bash* ;
- le shell qui interprétera vos scripts est *dash*.

Commande

- Suite de mots qui se finit par un terminateur de commande (le plus souvent, fin de ligne : touche entrée).
- Le premier mot est le *nom de commande*.
- Les éventuels mots suivants sont les *arguments de la commande*.
- Les différents mots sont séparés par des *blancs* (espaces et/ou tabulations).
- Le nombre de blancs entre les mots n'a pas d'importance.
- Les blancs avant le nom de commande et avant le terminateur de commande n'ont pas d'importance.

Commande

- Suite de mots qui se finit par un terminateur de commande (le plus souvent, fin de ligne : touche entrée).
- Le premier mot est le *nom de commande*.
- Les éventuels mots suivants sont les *arguments de la commande*.
- Les différents mots sont séparés par des *blancs* (espaces et/ou tabulations).
- Le nombre de blancs entre les mots n'a pas d'importance.
- Les blancs avant le nom de commande et avant le terminateur de commande n'ont pas d'importance.

Exemple :

```
echo   1    deux trois    4
```

Commande

- Suite de mots qui se finit par un terminateur de commande (le plus souvent, fin de ligne : touche entrée).
- Le premier mot est le *nom de commande*.
- Les éventuels mots suivants sont les *arguments de la commande*.
- Les différents mots sont séparés par des *blancs* (espaces et/ou tabulations).
- Le nombre de blancs entre les mots n'a pas d'importance.
- Les blancs avant le nom de commande et avant le terminateur de commande n'ont pas d'importance.

Exemple :

```
echo   1    deux trois    4
```

Nom de commande : `echo`, 4 arguments.

Qu'est-ce qu'une commande ?

On distingue quatre types de noms commandes :

- les programmes externes, qui peuvent être des programmes compilés (par exemple depuis des fichiers sources C), ou interprétés (par exemple par python, perl, etc, ou le shell lui-même !) ;
- les primitives du shell, c'est-à-dire des fonctionnalités internes au shell ;
- les fonctions shell, que l'on abordera plus tard ;
- les alias, qui sont des raccourcis pour d'autre commandes.

type type

La commande `type` permet de connaître le type d'un nom de commande :

```
$ type who date echo cd ls mkcd type
who est /usr/bin/who
date est /bin/date
echo est une primitive du shell
cd est une primitive du shell
ls est un alias vers "ls --color=auto"
mkcd est une fonction
mkcd ()
{
    mkdir "$1";
    cd "$1"
}
type est une primitive du shell
```

man man

Obtenir de l'aide :

- pour un programme externe : `man`
- pour une primitive du shell, dans bash, `help`

```
$ type type
type est une primitive du shell
$ help type
type: type [-afptP] nom [nom ...]
  Affiche des informations sur le type de commande.

  Pour chaque NOM, indique comment il serait
  interprete s il etait utilise comme un nom de
  commande.
  ...
$ type id
id est /usr/bin/id
$ man id
$ type man
man est /usr/bin/man
$ man man
```

Section, nom et synopsis

- Le manuel est organisé en sections (voir `man man` et TP).
- La section est entre parenthèses après le nom de commande en haut à gauche de la page.
- La partie « NOM » décrit en une ligne ce que fait la commande.
- La partie « SYNOPSIS » décrit les syntaxes acceptées par la commande.

  entre crochets optionnel
  avant ... peut être répété
  | indique un « ou exclusif »

- Les pages de manuel sont plus ou moins compréhensibles...
- ... mais on a rarement besoin de les comprendre en entier.

Aide, encore

- La plupart des commandes `GNU` ont une option `--help` qui donne une aide succinte.

```
$ who --help
Usage: who [OPTION]... [ FICHIER | ARG1 ARG2 ]
Afficher des informations sur les utilisateurs
connectés.
...
```

Aide, encore

- La plupart des commandes `GNU` ont une option `--help` qui donne une aide succinte.

```
$ who --help
Usage: who [OPTION]... [ FICHIER | ARG1 ARG2 ]
Afficher des informations sur les utilisateurs
connectés.
...
```

- `man -k mot` imprime les commandes dont la description succinte contient ce mot.

```
$ man -k chercher
apropos (1)           - Chercher le nom et la
  description des pages de manuel
badblocks (8)         - Rechercher des blocs
  défectueux sur un périphérique
...
```

Aide, encore

- Si elle est installée, la commande `whatis` permet d'obtenir une description brève d'une commande ou d'une fonction, ainsi que sa section de manuel.

```
$ whatis printf
printf (1)    - Formatter et afficher des donnees
printf (3)    - formatted output conversion
```

Répertoires et fichiers normaux

Parmi les différents types de fichiers, on trouve :

les répertoires (*directories*) qui contiennent seulement des informations sur les fichiers qu'ils « contiennent » ;

les fichiers normaux (*regular files*) qui contiennent des données (fichier source C, image au format jpg, programme compilé, ...).

Répertoires et fichiers normaux

Parmi les différents types de fichiers, on trouve :

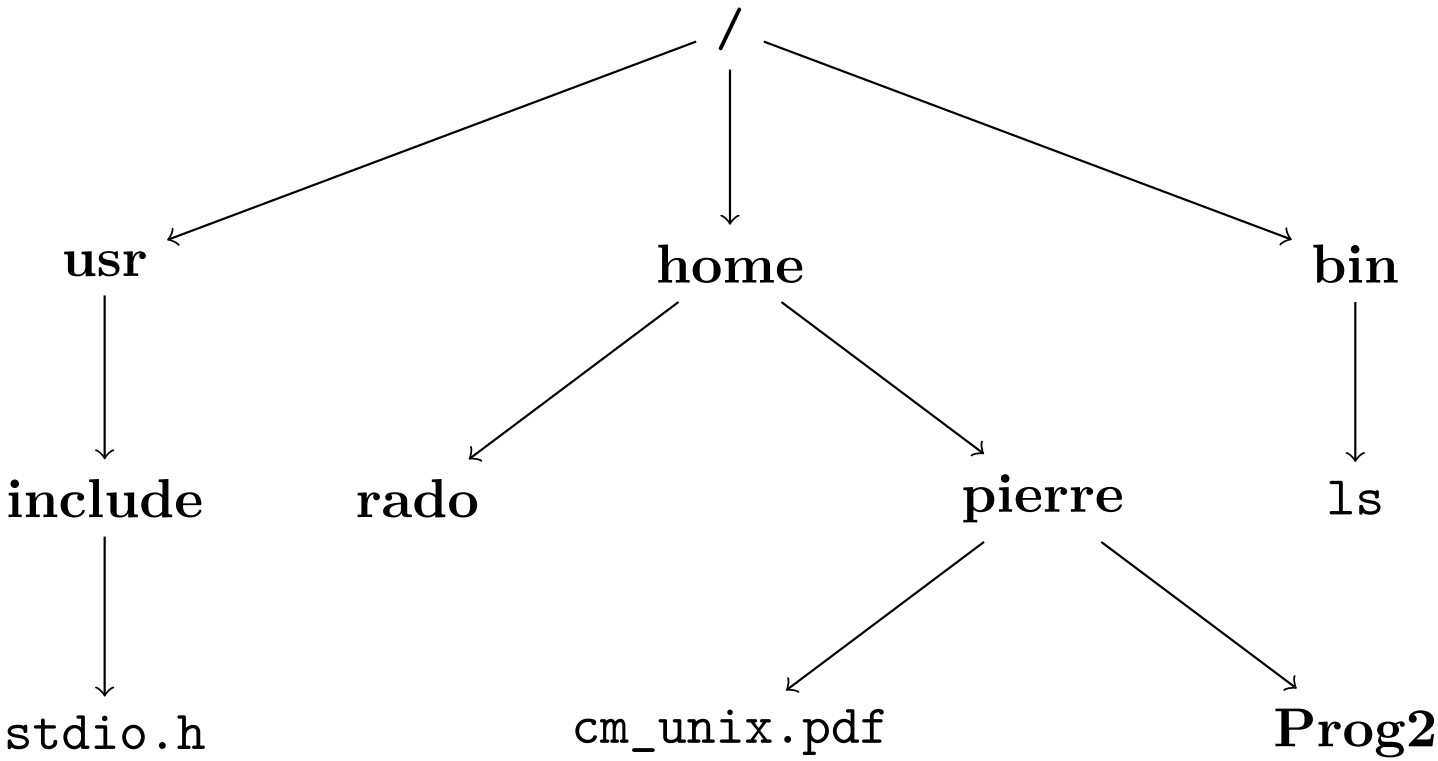
les répertoires (*directories*) qui contiennent seulement des informations sur les fichiers qu'ils « contiennent » ;

les fichiers normaux (*regular files*) qui contiennent des données (fichier source C, image au format jpg, programme compilé, ...).

Ces fichiers sont

- organisés sous la forme d'une unique arborescence ;
- dont les nœuds internes sont des répertoires ;
- dont les fichiers non répertoires sont les feuilles ;
- dont la racine de l'arborescence est le répertoire `/` ;
- dans laquelle chaque répertoire a un répertoire parent ;
- où la racine est le seul répertoire qui est son propre parent.

Un exemple



Chemin absolu

Un chemin absolu est

- une chaîne de caractères ;
- qui commence par `/` ;
- où les différents éléments du chemin sont séparés par `/` ;
- où chaque fichier du chemin est situé dans le répertoire qui le précède immédiatement dans ce chemin.

Exemples :

- `/home/pierre/cm_shell.pdf`
- `/usr`
- `/usr/`
- `/usr/include/`
- `/`

Si un répertoire se trouve à la fin du chemin, il peut être suivi ou non du caractère `/`

. et ..

Dans un chemin, le point (`.`) désigne le répertoire qui le précède immédiatement, tandis que deux points successifs (`..`) désigne le parent du répertoire qui le précède. Exemples :

- `/home/pierre/../rado` → `/home/rado`
- `/usr/include/../..` → `/`
- `/usr/./include/./` → `/usr/include`
- `/usr/../home/../bin/./` → `/bin`

Répertoire courant et chemin relatif

À tout moment, le shell enregistre le chemin absolu d'un répertoire qu'on appelle répertoire courant (*working directory*).
La commande `pwd` (pour *print working directory*) permet de l'afficher dans le terminal.

- Un chemin qui ne commence pas par `/` est un chemin relatif au répertoire courant.
- Le chemin absolu correspondant est obtenu en concaténant le chemin absolu du répertoire courant et le chemin relatif.

Exemples, si `/home/pierre` est le répertoire courant :

- `cm_shell.pdf` → `/home/pierre/cm_shell.pdf`
- `Prog2/` → `/home/pierre/Prog2/`
- `../rado` → `/home/rado`
- `./cm_shell.pdf` → `/home/pierre/cm_shell.pdf`
- `../../usr/include/stdio.h` → `/usr/include/stdio.h`

Répertoire courant et chemin relatif

Exemples, si `/home/pierre` est le répertoire courant :

- `cm_shell.pdf` → `/home/pierre/cm_shell.pdf`
- `Prog2/` → `/home/pierre/Prog2/`
- `..` → `/home/`
- `../rado` → `/home/rado`
- `.` → `/home/pierre`
- `./cm_shell.pdf` → `/home/pierre/cm_shell.pdf`
- `../../usr/include/stdio.h` → `/usr/include/stdio.h`

Répertoire courant et chemin relatif

- Pour changer de répertoire courant :
  `cd chemin_rep`
- exemples :
  - `cd Prog2`
  - `cd ../../rado`
  - `cd /usr/include`

Partout où un chemin est attendu, on peut utiliser soit un chemin relatif, soit un chemin absolu !

Répertoire personnel

- Chaque utilisateur du système a un répertoire personnel (*home directory*).
- Au démarrage, le répertoire courant est le répertoire personnel.
- La commande `cd` sans argument fait du répertoire personnel le répertoire courant.
- S'il est en début de mot, le caractère `~` (tilde, `Alt Gr-2`) est remplacé par le shell par le chemin absolu du répertoire personnel de l'utilisateur : c'est le développement du tilde.

Répertoire personnel

- Chaque utilisateur du système a un répertoire personnel (*home directory*).
- Au démarrage, le répertoire courant est le répertoire personnel.
- La commande `cd` sans argument fait du répertoire personnel le répertoire courant.
- S'il est en début de mot, le caractère `~` (tilde, `Alt Gr-2`) est remplacé par le shell par le chemin absolu du répertoire personnel de l'utilisateur : c'est le développement du tilde.

Il arrive que le shell transforme la ligne de commande avant que la commande soit lancée.

Lister le contenu d'un répertoire

La commande `ls` (pour *list*) permet de lister le contenu d'un ou plusieurs répertoire et/ou d'afficher des informations sur les fichiers.

- Sans argument, elle liste le contenu du répertoire courant.
- Pour chaque argument qui est un chemin de répertoire, elle liste le contenu de ce répertoire.
- Pour chaque argument qui est un chemin de fichier non répertoire, elle l'affiche.
- Des tonnes (trop !) d'options...
- À utiliser énormément en mode interactif, mais (presque) jamais dans les scripts à cause de problèmes de portabilité (par exemple le `ls` qu'on trouve sous GNU/Linux est différent de celui de MacOS).

Quelques options standard de `ls`

- `-l` : affichage *long*, chaque fichier sur une ligne, avec les permissions, le propriétaire, la taille, etc ;
- `-a` : pour *all*, afficher aussi les fichiers cachés, c'est-à-dire ceux dont le nom commence par un point (`.`) ;
- `-h` : pour *human-readable*, les tailles affichées avec l'option `l` sont données avec des unités (K, M, G, etc) pour être plus lisibles plutôt qu'en octets ;
- `-d` : pour *directory*, traiter les répertoires en argument comme des fichiers normaux, ne pas lister leur contenu ;
- `-t` : pour *time*, trier en fonction de la date de dernière modification ;
- `-r` : pour *reverse*, inverser le tri.

On peut combiner les options (c'est souvent le cas), par exemple :

```
ls -lhrt ~ /usr
```

Créer et détruire des répertoires

- `mkdir rep...` (pour *make directory*)
  crée les répertoires dont les chemins sont donnés en arguments
- échoue si le chemin désigne déjà un fichier existant
- `rmdir rep...` (pour *remove directory*)
  détruit les répertoires vides dont les chemins sont donnés en arguments
- échoue si le chemin ne désigne pas un répertoire vide.

Ces commandes sont sûres.

Supprimer des fichiers

- `rm fichier...` (pour *remove*)
  supprime les fichiers *normaux* dont les chemins sont donnés en argument
- échoue si un des chemins désigne un répertoire
- `rm -r fichier...` ou `rm -R fichier...` (pour *récursif*)
  supprime les fichiers ou répertoires (et tout ce qu'ils contiennent !) dont les chemins sont donnés en arguments.
- `rm` demande confirmation avant de supprimer un fichier protégé en écriture, sauf avec l'option `-f` (pour *force*).
- Les fichiers et répertoires supprimés le sont définitivement.

Cette commande est indispensable mais très dangereuse, surtout avec l'option `-r`.
Avec l'option `-i` (pour *interactif*), `rm` demande confirmation avant de supprimer chaque fichier.

Copier et déplacer des fichiers

- `cp src dest`
  crée une copie du fichier dont le chemin est `src` vers le chemin `dest`
  échoue si src désigne un répertoire, sauf si option `-R` (pour récursif).
- `cp sources... rep/`
  Crée des copies des fichiers sources (avec le même nom de base) dans le répertoire `rep`, qui doit être un répertoire existant.

Copier et déplacer des fichiers

- `cp src dest`
  crée une copie du fichier dont le chemin est `src` vers le chemin `dest`
  échoue si src désigne un répertoire, sauf si option `-R` (pour récursif).
- `cp sources... rep/`
  Crée des copies des fichiers sources (avec le même nom de base) dans le répertoire `rep`, qui doit être un répertoire existant.
- `mv src dest`
  Déplace (*move*) le fichier dont le chemin est `src` vers le chemin `dest`
  N'échoue pas si src est un répertoire.
  C'est en fait un renommage.
- `mv sources... rep/`
  Déplace les fichiers `sources` dans le répertoire `rep`, qui doit être un répertoire existant.

Copier et déplacer des fichiers

- `cp src dest`
  crée une copie du fichier dont le chemin est `src` vers le chemin `dest`
  échoue si src désigne un répertoire, sauf si option `-R` (pour récursif).
- `cp sources... rep/`
  Crée des copies des fichiers sources (avec le même nom de base) dans le répertoire `rep`, qui doit être un répertoire existant.
- `mv src dest`
  Déplace (*move*) le fichier dont le chemin est `src` vers le chemin `dest`
  N'échoue pas si src est un répertoire.
  C'est en fait un renommage.
- `mv sources... rep/`
  Déplace les fichiers `sources` dans le répertoire `rep`, qui doit être un répertoire existant.
- Commandes dangereuses car peuvent écraser silencieusement un fichier déjà existant (mais option `-i`).

Caractères joker et motifs du shell (1)

`*` `?` `[...]` `[!...]`

- L'étoile `*` correspond à (en anglais *match*) n'importe quelle chaîne de caractère.
- `a*`

Caractères joker et motifs du shell (1)

`*` `?` `[...]` `[!...]`

- L'étoile `*` correspond à (en anglais *match*) n'importe quelle chaîne de caractère.
- `a*` ↔ toute chaîne commençant par un `a` (a, abricot, ...)
- `*b*a*`

Caractères joker et motifs du shell (1)

`*` `?` `[...]` `[!...]`

- L'étoile `*` correspond à (en anglais *match*) n'importe quelle chaîne de caractère.
- `a*` ↔ toute chaîne commençant par un `a` (a, abricot, ...)
- `*b*a*` ↔ toute chaîne contenant un au moins un `a` après au moins un `b` (ba, babar, débobinage)
- Le point d'interrogation `?` correspond à n'importe quel caractère.
- `a?c`

Caractères joker et motifs du shell (1)

`*` `?` `[...]` `[!...]`

- L'étoile `*` correspond à (en anglais *match*) n'importe quelle chaîne de caractère.
- `a*` ↔ toute chaîne commençant par un `a` (a, abricot, ...)
- `*b*a*` ↔ toute chaîne contenant un au moins un `a` après au moins un `b` (ba, babar, débobinage)
- Le point d'interrogation `?` correspond à n'importe quel caractère.
- `a?c` ↔ toute chaîne de trois caractères commençant par `a` et se terminant par `c` (abc, arc, a.c, ...)

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]`

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]` $\leftrightarrow$ toute chaîne se terminant par `.c` ou `.h` (`stdio.c`, `cat.c`, ...).
- `[aeiouy]*`

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]` ↔ toute chaîne se terminant par `.c` ou `.h` (`stdio.c`, `cat.c`, ...).
- `[aeiouy]*` ↔ toute chaîne commençant par une voyelle

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]` ↔ toute chaîne se terminant par `.c` ou `.h` (`stdio.c`, `cat.c`, ...).
- `[aeiouy]*` ↔ toute chaîne commençant par une voyelle
- On peut abréger les suites de caractères consécutifs avec un tiret `-` : `[a-d]` remplace `[abcd]`.
- `[0-9]*`

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]` ↔ toute chaîne se terminant par `.c` ou `.h` (`stdio.c`, `cat.c`, ...).
- `[aeiouy]*` ↔ toute chaîne commençant par une voyelle
- On peut abréger les suites de caractères consécutifs avec un tiret `-` : `[a-d]` remplace `[abcd]`.
- `[0-9]*` ↔ toute chaîne commençant par un chiffre décimal

Caractères joker et motifs du shell (2)

`*` `?` `[...]` `[!...]`

- Un ensemble de caractères entre crochets `[...]` correspond à un seul caractère présent dans les crochets.
- `*.[ch]` ↔ toute chaîne se terminant par `.c` ou `.h` (`stdio.c`, `cat.c`, ...).
- `[aeiouy]*` ↔ toute chaîne commençant par une voyelle
- On peut abréger les suites de caractères consécutifs avec un tiret `-` : `[a-d]` remplace `[abcd]`.
- `[0-9]*` ↔ toute chaîne commençant par un chiffre décimal
- Avec le caractère `!` *en première position* `[!...]` correspond à un seul caractère non présent dans les crochets.
- `[!a-zA-Z]` ↔ un caractère qui n'est pas une lettre alphabétique (minuscule ou majuscule)

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` $\leftrightarrow$

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔ un caractère qui n'est ni `[` ni `]`
- Un `!` ailleurs qu'en première position est littéral.
- `[?!.]`

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔ un caractère qui n'est ni `[` ni `]`
- Un `!` ailleurs qu'en première position est littéral.
- `[?!.]` ↔ un des caractères `?`, `!` ou `.`
- `[!!]`

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔ un caractère qui n'est ni `[` ni `]`
- Un `!` ailleurs qu'en première position est littéral.
- `[?!.]` ↔ un des caractères `?`, `!` ou `.`
- `[!!]` ↔ un caractère qui n'est pas un point d'exclamation.
- Un `-` en première position est littéral (deuxième si `!` ou `]` en première position, troisième si ...)
- `[-a-z]`

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔ un caractère qui n'est ni `[` ni `]`
- Un `!` ailleurs qu'en première position est littéral.
- `[?!.]` ↔ un des caractères `?`, `!` ou `.`
- `[!!]` ↔ un caractère qui n'est pas un point d'exclamation.
- Un `-` en première position est littéral (deuxième si `!` ou `]` en première position, troisième si ...)
- `[-a-z]` ↔ un tiret ou une lettre minuscule.
- `[!]-!]`

Caractères joker et motifs du shell (3)

`*` `?` `[...]` `[!...]`

Et si je veux mettre `]` ou `!` ou `-` entre les crochets ?

- On doit mettre `]` en première position (deuxième si `[!...]`).
- `[][]` ↔ l'un des caractères `[` ou `]`
- `[!][]` ↔ un caractère qui n'est ni `[` ni `]`
- Un `!` ailleurs qu'en première position est littéral.
- `[?!.]` ↔ un des caractères `?`, `!` ou `.`
- `[!!]` ↔ un caractère qui n'est pas un point d'exclamation.
- Un `-` en première position est littéral (deuxième si `!` ou `]` en première position, troisième si ...)
- `[-a-z]` ↔ un tiret ou une lettre minuscule.
- `[!]-!]` ↔ un caractère qui n'est ni `]` ni `-` ni `!`

Développement de noms de chemins

- Si un mot de la ligne de commande contient au moins un des caractères `*`, `?` ou `[`, le shell essaie de trouver des chemins de fichiers correspondant à un motif.
- En cas de succès, le mot est remplacé par tous les chemins correspondants.
- Attention : en cas d'échec, le mot est laissé tel quel.
- C'est le développement de noms de chemins (*pathname expansion*).

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`
  `exo1.c exo2.c truc.c`
- `echo *.tex`

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`
  `exo1.c exo2.c truc.c`
- `echo *.tex`
  `*.tex`
- `echo ./exo[0-9].c`

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`
  `exo1.c exo2.c truc.c`
- `echo *.tex`
  `*.tex`
- `echo ./exo[0-9].c`
  `./exo1.c ./exo2.c`
- `echo ?ruc.c`

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`
  `exo1.c exo2.c truc.c`
- `echo *.tex`
  `*.tex`
- `echo ./exo[0-9].c`
  `./exo1.c ./exo2.c`
- `echo ?ruc.c`
  `truc.c`
- `echo *.[!ch]`

Exemple : si le répertoire courant contient les fichiers `exo1.c`, `exo2.c`, `truc.c` et `exo1.h`.

- `echo *.c`
  `exo1.c exo2.c truc.c`
- `echo *.tex`
  `*.tex`
- `echo ./exo[0-9].c`
  `./exo1.c ./exo2.c`
- `echo ?ruc.c`
  `truc.c`
- `echo *.[!ch]`
  `*.[!ch]`

Deux subtilités

Lors du développement de noms de chemins,

- tous les caractères `/` doivent être explicitement écrits ;
- tous les points (`.`) en début de chaîne ou juste après un `/` doivent être explicitement écrits.

Les fichiers dont le nom commence par un point sont appelés des fichiers cachés.

Deux subtilités

Lors du développement de noms de chemins,

- tous les caractères `/` doivent être explicitement écrits ;
- tous les points (`.`) en début de chaîne ou juste après un `/` doivent être explicitement écrits.

Les fichiers dont le nom commence par un point sont appelés des fichiers cachés.

- `echo /*` affiche tous les fichiers présents dans le répertoire racine.
- `echo /*/*` affiche tous les fichiers présents dans un sous-répertoire de la racine.
- `echo .*` affiche tous les fichiers cachés présents dans le répertoire courant.

Démonstration

Fichiers cachés

- dans chaque répertoire, `.` (lien vers lui-même) et `..` (lien vers son parent)
- fichiers de configuration dans le répertoire personnel, par exemple `.bashrc`, `.vimrc`, ...
- voire des répertoires entiers cachés, par exemple `~/.mozilla`, ...

Question

Que fait la commande suivante ?

```
rm -rf ~/*
```