Initiation à l'environnement Unix
CM4 : fichiers, inodes, répertoires et recherche de commandes

Pierre Rousselin

Université Sorbonne Paris Nord
L1 informatique et DL
octobre 2024

Fichiers texte

- Le système Unix favorise depuis le début les *fichiers textes.*
- `comptine.txt` est un exemple de tel fichier.

Définition

Un fichier texte est un fichier destiné à être interprété comme *une suite de caractères* dans un certain *jeu de caractères codés* (UTF-8, ASCII, Latin-1, ...) de façon à être *lisible et facilement éditable par un humain.*

Fichiers texte

Exemples :

- Un fichier texte sans forme particulière, comme `comptine.txt`.
- Un fichier source C, java, OCaml, FORTRAN, ... destiné à être compilé donc à être compris également par un programme appelé compilateur.
- Un script shell, python, perl, awk, ... destiné à être interprété donc à être compris également par un programme appelé interprète (ou interpréteur calque de l'anglais *interpreter*).
- Un document HTML destiné à être compris par un navigateur web pour produire un affichage dans sa fenêtre.
- Les fichiers de configuration sous Unix, pouvant avoir chacun leur format, par exemple `~/.ssh/config` pour la configuration du client `ssh` de l'utilisateur, les fichiers de configuration du système dans `/etc` (pour *editable text configuration*).

Fichiers dits « binaire »

Définition

Un fichier qui n'est pas un fichier texte est un *fichier binaire.*

Cette appellation est assez malheureuse (les fichiers textes sont aussi codés en binaire...) mais c'est celle qui est répandue. Un humain ne peut pas comprendre son contenu avec un simple éditeur de texte. Ce fichier n'aura de sens que pour le ou les programmes à qui il est destiné.

Exemples :

- Un exécutable produit par un compilateur (C, java, OCaml, etc).
- Une image (jpg, png, etc), une vidéo, des sons...
- Une archive compressée.
- Un fichier utilisé par un logiciel de traitement de texte (comme LibreOffice ou Microsoft Word).

Fichiers binaires exécutables

- Suivant le contexte, un *binaire* peut aussi vouloir dire plus spécifiquement un programme compilé, par opposition au code source du programme qui lui est un fichier texte.
- Raison pour laquelle la plupart des programmes du système Unix sont traditionnellement dans `/bin` et/ou `/usr/bin` (bien qu'ils contiennent aussi des scripts qui sont des fichiers texte !)
- L'existence de binaires (exécutables) dont le code source n'est pas public pose problème :
  - le binaire n'est pas lisible par un humain ;
  - dès lors, comment savoir ce que fait réellement le programme ? Est-il dangereux ?

Le logiciel libre (*free software*)

- Lancé par Richard Stallman : GNU (1983) puis Free Software Foundation (1985).
- Outils logiciels mais aussi juridiques (licence GPL).

Les 4 libertés fondamentales que doit respecter un logiciel libre :

- la liberté de faire fonctionner le programme comme vous voulez, pour n'importe quel usage (liberté 0) ;
- la liberté d'étudier le fonctionnement du programme, et de le modifier pour qu'il effectue vos tâches informatiques comme vous le souhaitez (liberté 1) ; l'accès au code source est une condition nécessaire ;
- la liberté de redistribuer des copies, donc d'aider les autres (liberté 2) ;
- la liberté de distribuer aux autres des copies de vos versions modifiées (liberté 3) ; en faisant cela, vous donnez à toute la communauté une possibilité de profiter de vos changements ; l'accès au code source est une condition nécessaire.

Le logiciel libre

Quelques exemples :

- noyau Linux (licence GPLv2), implémentation libre d'un noyau Unix (entre 25 et 30 millions de lignes de code, plusieurs milliers de contributeurs chaque année) ;
- FreeBSD, implémentation libre d'Unix (noyau et utilitaires), 16 millions de ligne de code ;
- Projet GNU, par exemple `ls`, `sed`, `bash`, `emacs`, `gimp`, `gcc`, . . . ;
- Firefox (navigateur), Thunderbird (client mail) chez Mozilla ;
- La suite LibreOffice de la Document Foundation ;
- Chromium (navigateur) et le système Android (en partie) chez Google ;
- Visual Studio Code chez Microsoft.

Quelques contre-exemples : MS Windows, MacOS, MS Office, Google Chrome, Chrome OS, beaucoup de pilotes de périphériques.

Fichiers texte et fichiers binaires

Répertoires et inodes

Permissions associées aux fichiers

Résolution des noms de commande

Exécution de programmes interprétés : le *shebang*

Permissions et répertoires

Ce que ne contient *pas* un fichier

- son (ou ses) nom(s) dans l'arborescence ;
- ses permissions ;
- le nom du périphérique qui le contient et son emplacement sur celui-ci (informations permettant de trouver physiquement le contenu du fichier)
- ...

Bref, un fichier ne contient que les données écrites dedans et rien d'autre.

Alors où sont ces données ?

- Le ou les noms sont dans des répertoires ;
- les autres attributs du fichier sont dans une structure gérée par le système appelée inode ;
- un répertoire n'est qu'un tableau permettant d'associer à un nom un numéro appelé numéro d'inode qui permet de récupérer les informations de l'inode.

Ce sont les répertoires qui nomment les fichiers.

Nom, inodes et attributs

- La commande `ls`, les développements de noms de chemins, etc permettent de consulter les noms contenus dans un répertoire ;
- la commande `ls` avec l'option `-i` affiche le numéro d'inode ;
- la commande `ls` avec l'option `-l` affiche des attributs contenus dans l'inode ;
- ce que fait aussi la commande non standard `stat`.

Attributs de fichier

- numéro d'inode ;
- utilisateur propriétaire ;
- groupe propriétaire ;
- permissions ;
- dates :
  - de création (le fichier a été créé) ;
  - de dernière modification (le fichier a été ouvert en écriture) ;
  - de dernier accès (le fichier a été ouvert en lecture) ;
  - de dernier changement (l'inode du fichier a changé, par exemple ses permissions).
- taille ;
- périphérique sur lequel se trouve le fichier ;
- nombre de liens physiques, c'est-à-dire de noms dans l'arborescence.

Liens physiques

- Un lien physique est une association
  nom (dans un répertoire) $\leftrightarrow$ inode.
- La commande `ln` (comme *link*, lier en anglais) permet de donner un nouveau nom dans l'arborescence à un fichier.
- On peut donc avoir des noms différents mais un seul fichier.

Lien physique, exemple

```
$ ls -l foo
drwxr-xr-x. 1 pierre pierre 8 12 oct.  13:13 bar
-rw-r--r--. 2 pierre pierre 7 12 oct.  13:11 baz.txt
$ cat foo/baz.txt
hop
$ ln foo/baz.txt foo/bar/truc
$ ls -li foo/baz.txt foo/bar/truc
2357623 -rw-r--r--. 2 pierre pierre 7 12 oct. 13:11 foo/bar/truc
2357623 -rw-r--r--. 2 pierre pierre 7 12 oct. 13:11 foo/baz.txt
$ echo plop >>foo/bar/truc
$ cat foo/baz.txt
hop
plop
$ chmod a+x foo/baz.txt
$ ls -li foo/bar/truc
2357623 -rwxr-xr-x. 2 pierre pierre 4 12 oct. 13:19 foo/bar/truc
```

Suppression d'un nom, suppression d'un inode

- La commande `rm` supprime un lien et c'est tout (donc elle est très rapide !)
- Lorsque le nombre de liens tombe à 0, l'inode est supprimé.
- Le système sait que l'espace anciennement occupé par le fichier est maintenant disponible et peut être réutilisé.
- Il est alors impossible (à part avec de grands moyens et beaucoup de chance et de patience) de retrouver les données contenues dans le fichier.

```
$ ls -li foo/baz.txt
2357623 -rw-r--r--. 2 pierre pierre 7 12 oct. 13:11 foo/baz.txt
$ rm foo/bar/truc
$ ls -li foo/baz.txt
2357623 -rw-r--r--. 1 pierre pierre 7 12 oct. 13:11 foo/baz.txt
$ rm foo/baz.txt # inode supprimé
```

Manipuler un inode

Condition pour le faire : être propriétaire du fichier (ou `root`).

- Modifier les dates : `touch`.
- Modifier l'utilisateur propriétaire : `chown`.
- Modifier le groupe propriétaire : `chgrp`.
- Modifier les permissions `chmod`.

Permissions

ugo : user, group, others

rwx : read, write, execute

```
$ ls -ld hop plop /root
--w-r--r--. 1 pierre dialout   0 25 oct.  14:55 hop
-rw-r-----. 1 alice  wheel     0 25 oct.  14:55 plop
dr-xr-x---. 1 root   root    258 19 oct.  13:09 /root
$ id
uid=1000(pierre) gid=1000(pierre) groupes=pierre,wheel,dialout
```

Ici, l'utilisateur a les permissions :

Permissions

ugo : user, group, others
rwx : read, write, execute

```
$ ls -ld hop plop /root
--w-r--r--. 1 pierre dialout   0 25 oct.  14:55 hop
-rw-r-----. 1 alice  wheel     0 25 oct.  14:55 plop
dr-xr-x---. 1 root   root    258 19 oct.  13:09 /root
$ id
uid=1000(pierre) gid=1000(pierre) groupes=pierre,wheel,dialout
```

Ici, l'utilisateur a les permissions :

- `w` (et c'est tout) sur `hop` car il est utilisateur propriétaire, donc ce sont les permissions de `u` qui s'appliquent ;

Permissions

ugo : user, group, others
rwx : read, write, execute

```
$ ls -ld hop plop /root
--w-r--r--. 1 pierre dialout   0 25 oct.  14:55 hop
-rw-r-----. 1 alice  wheel     0 25 oct.  14:55 plop
dr-xr-x---. 1 root   root    258 19 oct.  13:09 /root
$ id
uid=1000(pierre) gid=1000(pierre) groupes=pierre,wheel,dialout
```

Ici, l'utilisateur a les permissions :

- `w` (et c'est tout) sur `hop` car il est utilisateur propriétaire, donc ce sont les permissions de `u` qui s'appliquent ;
- `r` (et c'est tout) sur `plop` car il n'est pas utilisateur propriétaire du fichier et l'un des groupes auxquels il appartient (`wheel`) est groupe propriétaire du fichier `plop` donc ce sont les permissions `g` qui s'appliquent ;

Permissions

ugo : user, group, others
rwx : read, write, execute

```
$ ls -ld hop plop /root
--w-r--r--. 1 pierre dialout   0 25 oct.  14:55 hop
-rw-r-----. 1 alice  wheel     0 25 oct.  14:55 plop
dr-xr-x---. 1 root   root    258 19 oct.  13:09 /root
$ id
uid=1000(pierre) gid=1000(pierre) groupes=pierre,wheel,dialout
```

Ici, l'utilisateur a les permissions :

- `w` (et c'est tout) sur `hop` car il est utilisateur propriétaire, donc ce sont les permissions de `u` qui s'appliquent ;
- `r` (et c'est tout) sur `plop` car il n'est pas utilisateur propriétaire du fichier et l'un des groupes auxquels il appartient (`wheel`) est groupe propriétaire du fichier `plop` donc ce sont les permissions `g` qui s'appliquent ;
- aucune sur le répertoire `/root` car il n'est pas son utilisateur propriétaire et n'appartient pas à son groupe propriétaire, ce sont les permissions `o` qui s'appliquent.

Permissions et fichiers normaux

Sur les fichiers normaux (en fait les fichiers de tous types sauf les répertoires) :

- `r` permet *d'ouvrir un fichier en lecture*, donc de demander au noyau de pouvoir *lire* (*read*) les octets qu'il contient. Exemples : `cat message.txt`, `eog image.jpg`, ...
- `w` permet *d'ouvrir un fichier en écriture*, donc de pouvoir *écrire* (*write*) des octets dedans. Exemple : `printf 'lol\n' >>fichier.txt` écrit les octets correspondants à la chaîne `lol` (suivi de l'octet `0x0A`) à la fin de `fichier.txt`, si on peut l'ouvrir en écriture.
- `x` permet *d'exécuter le fichier*, c'est-à-dire de l'envoyer au noyau pour qu'un processus exécute les instructions qu'il contient. Ces instructions sont en langage machine (ou presque) dans le cas d'un programme compilé, ou bien dans un langage interprété (shell, python, ...) dans le cas d'un script commençant par un shebang.

Changer les permissions

Possible si utilisateur propriétaire du fichier (ou root).
On utilise `chmod MODE[, MODE]... FICHIER...` où `MODE` a la forme `[ugoa...][-+=][rwx...]`

- pour qui sont changées les permissions (`user`, `group`, `others`, ou `all` : tout le monde) ;
- `-` pour enlever, `=` pour fixer, `+` pour ajouter.

Exemple

- `chmod og-r secret.txt` : enlever la permission `r` (si elle était présente, sinon ça ne change rien) à `g` et `o` ;
- `chmod a+rwx truc` donner tous les droits, à tout le monde, sur truc ;
- `chmod a=r truc` donner le droit `r` et seulement celui-là à tout le monde (les autres permissions que pouvaient éventuellement avoir des utilisateurs sur `truc` sont ôtées) ;
- `chmod u=rwx,og= truc` : tous les droits pour l'utilisateur propriétaire, aucun droit pour tous les autres utilisateurs.

Fichiers texte et fichiers binaires

Répertoires et inodes

Permissions associées aux fichiers

Résolution des noms de commande

Exécution de programmes interprétés : le *shebang*

Permissions et répertoires

Noms de commandes

- Rappel : le nom de commande est le premier mot de la ligne de commande.
- Rappel : il y a des primitives du shell et des commandes externes.

```
$ type cd rm
cd est une primitive du shell
rm est /usr/bin/rm
```

- Qu'est-ce qu'une commande externe ?
  - de la puissante magie ?
  - un fichier ?

Commandes externes

- Une commande externe n'est rien d'autre qu'un **fichier** contenant un **programme.**
- Par exemple, `rm` correspond souvent au fichier `/usr/bin/rm` qui est, le plus souvent :
  - un programme, au départ écrit en C [1] ;
  - puis compilé ;
  - et installé dans le répertoire `/usr/bin` : le fichier *compilé* y est copié ou déplacé.

1. Code source de GNU `rm` :
`http://git.savannah.gnu.org/cgit/coreutils.git/tree/src/rm.c`

Commandes externes

- Une commande externe n'est rien d'autre qu'un **fichier** contenant un **programme**.
- Par exemple, `rm` correspond souvent au fichier `/usr/bin/rm` qui est, le plus souvent :
  - un programme, au départ écrit en C [1] ;
  - puis compilé ;
  - et installé dans le répertoire `/usr/bin` : le fichier *compilé* y est copié ou déplacé.
- Comment le système sait-il où sont les programmes ?

1. Code source de GNU `rm` :
`http://git.savannah.gnu.org/cgit/coreutils.git/tree/src/rm.c`

Résolution des noms de commandes : une mécanique simple

En simplifiant un tout petit peu pour des raisons pédagogiques...

- Si le nom de commande est une primitive du shell, le shell lance la commande lui-même.
- Ensuite, si le nom de commande contient un `/`, le shell cherche simplement un fichier dont le chemin est ce nom !
- Exemple :

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, world!\n"); return 0; }
$ gcc -Wall hello.c -o hello
$ ./hello
Hello, world!
$
```

- Le shell voit un `/` dans le nom de commande. Il cherche le fichier `./hello` (simplement `hello` dans le répertoire courant).
- Il le trouve, puis lance un nouveau processus avec ce programme dedans (en l'occurrence compilé).

PATH

Mais que se passe-t-il avec, par exemple, la commande

```
$ rm foo.txt
```

PATH

Mais que se passe-t-il avec, par exemple, la commande

```
$ rm foo.txt
```

- Lorsque le shell reçoit un nom de commande sans `/` dedans, par exemple `rm`, qui n'est pas une primitive...
- il cherche un programme dans les répertoires de la variable `PATH`.

PATH

Mais que se passe-t-il avec, par exemple, la commande

```
$ rm foo.txt
```

- Lorsque le shell reçoit un nom de commande sans `/` dedans, par exemple `rm`, qui n'est pas une primitive...
- il cherche un programme dans les répertoires de la variable `PATH`.

La variable `PATH` contient simplement une liste de répertoires séparés par des `:`, par exemple :

```
$ printf '%s\n' "$PATH"
/usr/local/bin:/usr/bin:/bin
```

PATH

Mais que se passe-t-il avec, par exemple, la commande

```
$ rm foo.txt
```

- Lorsque le shell reçoit un nom de commande sans `/` dedans, par exemple `rm`, qui n'est pas une primitive...
- il cherche un programme dans les répertoires de la variable `PATH`.

La variable `PATH` contient simplement une liste de répertoires séparés par des `:`, par exemple :

```
$ printf '%s\n' "$PATH"
/usr/local/bin:/usr/bin:/bin
```

Alors, il cherche un fichier exécutable appelé `rm` :

- dans `/usr/local/bin` (pas trouvé), puis
- dans `/usr/bin` : trouvé !

Il crée un nouveau processus avec le programme `/usr/bin/rm`

PATH

Mais que se passe-t-il avec, par exemple, la commande

```
$ rm foo.txt
```

- Lorsque le shell reçoit un nom de commande sans `/` dedans, par exemple `rm`, qui n'est pas une primitive...
- il cherche un programme dans les répertoires de la variable `PATH`.

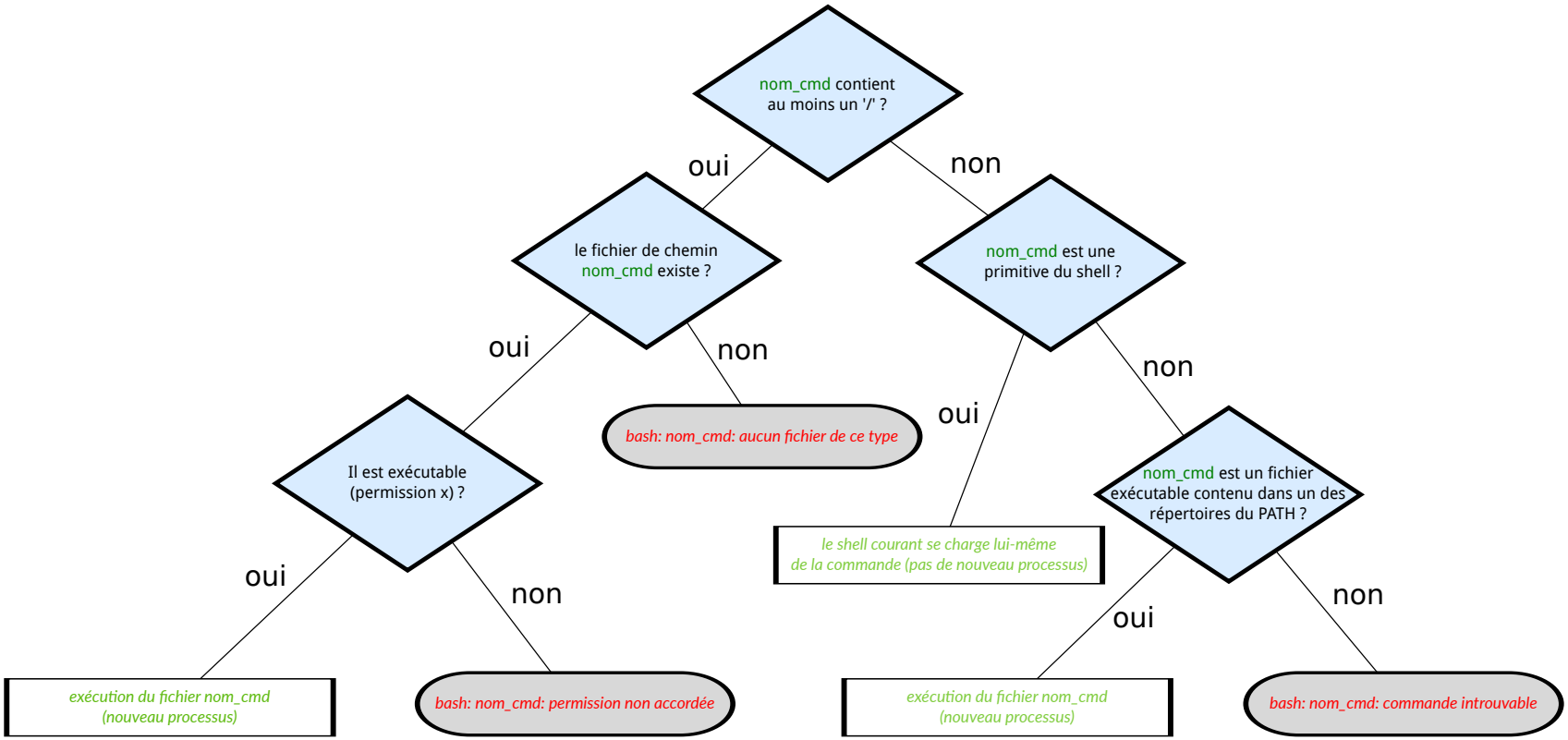
La variable `PATH` contient simplement une liste de répertoires séparés par des `:`, par exemple :

```
$ printf '%s\n' "$PATH"
/usr/local/bin:/usr/bin:/bin
```

Alors, il cherche un fichier exécutable appelé `rm` :

- dans `/usr/local/bin` (pas trouvé), puis
- dans `/usr/bin` : trouvé !

Il crée un nouveau processus avec le programme `/usr/bin/rm` en lui donnant pour argument la chaîne `foo.txt`.


nom_cmd contient au moins un '/' ?
oui
non
le fichier de chemin nom_cmd existe ?
nom_cmd est une primitive du shell ?
oui
non
oui
non
bash: nom_cmd: aucun fichier de ce type
Il est exécutable (permission x) ?
le shell courant se charge lui-même de la commande (pas de nouveau processus)
nom_cmd est un fichier exécutable contenu dans un des répertoires du PATH ?
oui
non
oui
non
exécution du fichier nom_cmd (nouveau processus)
bash: nom_cmd: permission non accordée
exécution du fichier nom_cmd (nouveau processus)
bash: nom_cmd: commande introuvable

Fichiers texte et fichiers binaires

Répertoires et inodes

Permissions associées aux fichiers

Résolution des noms de commande

Exécution de programmes interprétés : le *shebang*

Permissions et répertoires

Cas d'un programme interprété

- Le programme `rm` de `/usr/bin/` est compilé en langage machine : le noyau sait l'exécuter tout seul.
- Mais il y a de très nombreux langages de programmation qui sont *interprétés*.
- Un programme (souvent compilé, lui) sait lire des fichiers texte appelés scripts ;
- par exemple : `python`, `awk`, le shell lui-même : `sh`, en calcul numérique `octave` ou `matlab`, ...
- l'utilisateur écrit un programme dans le langage associé à ce programme ;
- et peut appeler le programme explicitement pour lire son script.

Exemples de scripts

- Un script `python` dans le fichier `hello.py`

```
print("Hello, world!")
```

- Exécution en nommant l'interprète :

```
$ python hello.py
Hello, world!
```

- Un script `awk` dans le fichier `plop.awk`

```
BEGIN { print "plop" }
```

- Exécution en nommant l'interprète :

```
$ awk -f plop.awk
plop
```

Homogénéité du lancement de programmes

- Dans le système Unix, il ne doit pas y avoir de différence pour l'utilisateur entre les programmes compilés et interprétés.
- Il faut donc une convention pour dire, dans le cas des programmes en texte, interprétés, quel programme interprète utiliser.
- Ceci est fait dans la première ligne du programme.
- Les deux premiers octets sont `#!`
- suivis éventuellement de blancs
- puis d'un chemin absolu vers le programme interprète
- avec éventuellement une option

Exemple avec python

```
$ cat hello.py
#!/bin/python
print("Hello, world!")
$ chmod u+x hello.py
$ ./hello.py # le noyau trouve /bin/python
Hello, world!
```

Exemple avec python

```
$ cat hello.py
#!/bin/python
print("Hello, world!")
$ chmod u+x hello.py
$ ./hello.py # le noyau trouve /bin/python
Hello, world!
```

Exactement comme si on avait entré la commande

```
$ /bin/python hello.py
```

Exemple avec `awk`

```
$ cat plop.awk
#!/bin/awk -f
BEGIN { print "plop" }
$ chmod u+x plop.awk
$ ./plop.awk # le noyau trouve /bin/awk
plop
```

Exemple avec awk

```
$ cat plop.awk
#!/bin/awk -f
BEGIN { print "plop" }
$ chmod u+x plop.awk
$ ./plop.awk # le noyau trouve /bin/awk
plop
```

Exactement comme si on avait entré la commande

```
$ /bin/awk -f plop.awk
```

Fichiers texte et fichiers binaires

Répertoires et inodes

Permissions associées aux fichiers

Résolution des noms de commande

Exécution de programmes interprétés : le *shebang*

Permissions et répertoires

Permissions et répertoire

Image mentale : un répertoire n'est qu'un tableau à deux colonnes : nom et numéro d'inode.

- r permet de *lire les noms* (et seulement les noms), donc en pratique `ls` sans option, autocomplétion dans `bash` et développement de noms de chemins ;
- w permet de *modifier le tableau* en ajoutant, supprimant ou modifiant des noms, mais sans `x`, `w` ne permet pas de faire quoi que ce soit ;
- x permet d'associer un numéro inode à un nom, ce qui est indispensable pour toute opération sur un fichier à partir du nom donné par ce répertoire.

Permissions, exemples

- `ls rep/` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` : besoin de `wx` sur `rep/` : modification du répertoire et accès à l'inode pour diminuer le nombre de liens
  Aucune permission sur le fichier `rep/a` n'est nécessaire, c'est la permission sur le répertoire qui permet de `rm` !
- `cat rep/a` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` : besoin de `wx` sur `rep/` : modification du répertoire et accès à l'inode pour diminuer le nombre de liens
  Aucune permission sur le fichier `rep/a` n'est nécessaire, c'est la permission sur le répertoire qui permet de `rm` !
- `cat rep/a` : besoin de `x` sur `rep/` et `r` sur `rep/` (besoin d'accéder à l'inode pour accéder aux permissions et aux données !)
- `cp rep/a rep/b` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` : besoin de `wx` sur `rep/` : modification du répertoire et accès à l'inode pour diminuer le nombre de liens
  Aucune permission sur le fichier `rep/a` n'est nécessaire, c'est la permission sur le répertoire qui permet de `rm` !
- `cat rep/a` : besoin de `x` sur `rep/` et `r` sur `rep/` (besoin d'accéder à l'inode pour accéder aux permissions et aux données !)
- `cp rep/a rep/b` : besoin de `wx` sur `rep/`, `r` sur `rep/a` et `w` sur `rep/b`
- `rm rep/*` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` : besoin de `wx` sur `rep/` : modification du répertoire et accès à l'inode pour diminuer le nombre de liens
  Aucune permission sur le fichier `rep/a` n'est nécessaire, c'est la permission sur le répertoire qui permet de `rm` !
- `cat rep/a` : besoin de `x` sur `rep/` et `r` sur `rep/` (besoin d'accéder à l'inode pour accéder aux permissions et aux données !)
- `cp rep/a rep/b` : besoin de `wx` sur `rep/`, `r` sur `rep/a` et `w` sur `rep/b`
- `rm rep/*` : besoin de `rwx` sur `rep/`, `r` pour le développement de noms de chemins, `w` pour supprimer les noms contenus dans le répertoire et `x` pour l'accès aux inodes des fichiers de `rep/`
- `/usr/bin/cal` :

Permissions, exemples

- `ls rep/` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `echo rep/*` : besoin de `r` sur `rep/` (`x` aussi sur certains systèmes).
- `rm rep/a` : besoin de `wx` sur `rep/` : modification du répertoire et accès à l'inode pour diminuer le nombre de liens
  Aucune permission sur le fichier `rep/a` n'est nécessaire, c'est la permission sur le répertoire qui permet de `rm` !
- `cat rep/a` : besoin de `x` sur `rep/` et `r` sur `rep/` (besoin d'accéder à l'inode pour accéder aux permissions et aux données !)
- `cp rep/a rep/b` : besoin de `wx` sur `rep/`, `r` sur `rep/a` et `w` sur `rep/b`
- `rm rep/*` : besoin de `rwx` sur `rep/`, `r` pour le développement de noms de chemins, `w` pour supprimer les noms contenus dans le répertoire et `x` pour l'accès aux inodes des fichiers de `rep/`
- `/usr/bin/cal` : besoin de `x` sur `/` pour accéder à l'inode correspondant à `/usr`, de `x` sur `/usr` pour accéder à l'inode de `/usr/bin`, de `x` sur `/usr/bin` pour l'inode de `cal` et enfin de `x` sur `/usr/bin/cal` pour l'exécuter

Permissions, contenus, noms, inode : schéma

```
$ cat ~/docs/hop
lol
```