UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

**Faculté des Sciences — Département d'Informatique**

ULB

INFO-F-201 – Systèmes d'exploitation

# Systèmes de fichiers et entrées / sorties

Joël Goossens (N8.107) Olivier Markowitch (N8.106)
Arnaud Leponce (N8.213) Yannick Molinghen (N8.210)
Alexis Reynouard (N8.215)

## 1 Le système de permissions

### 1.1 Description

Sous UNIX, la notion d'***utilisateur*** (*user*) sert à déterminer qui peut faire quelles opérations sur la machine. Toute action est faite au nom d'un utilisateur (c.-à-d. que tout processus est associé à un utilisateur) et tout fichier appartient à un utilisateur.

utilisateur

Les utilisateurs appartiennent à un ***groupe*** principal et potentiellement à d'autres groupes. Le plus souvent, le groupe principal d'un utilisateur « adam » est le groupe « adam » dont le seul utilisateur est « adam ». Cette organisation rend pratique la gestion des groupes et des utilisateurs.

groupe

Chaque fichier a un utilisateur ***propriétaire*** (*owner*) et un groupe (un groupe d'utilisateurs de la machine). Ces informations peuvent être consultées grâce à la commande `ls -l` (`-l` pour *long format*).

propriétaire

De plus, pour chaque fichier, on distingue les ***permissions*** :

permissions

- en ***lecture*** (*read*) qui permettent de consulter le fichier,
- en ***écriture*** (*write*) qui permettent de le modifier et
- en ***exécution*** (*execute*) permet d'exécuter le fichier le cas échéant.

lecture
écriture
exécution

Pour les répertoires,

- la permission en lecture permet de consulter le contenu (par la commande `ls` notamment),
- la permission en écriture permet de créer des fichiers dans ce répertoire et
- la permission en exécution permet de le ***traverser***, c.-à-d. soit de se positionner dedans grâce à `cd`, soit de le traverser (au travers d'un chemin de type `répertoire1/répertoire2/fichier`) pour accéder à l'un des fichiers qu'il contient.

traverser

Les systèmes UNIX maintiennent, pour chaque fichier, 3 ensembles de permissions :

1. les permissions accordées au propriétaire du fichier (*user*),
2. celles accordées aux utilisateurs appartenant au groupe du fichier (*group*),
3. et celles accordées aux autres utilisateurs (*other*, ni propriétaire ni appartenant au groupe).

Pour chacune de ces entités, on peut avoir comme permission n'importe quel sous-ensemble de l'ensemble {`r`, `w`, `x`} (`r` pour lecture, `w` pour écriture, `x` pour exécution). On peut consulter ces permission à l'aide de la commande `ls -l`.

## 1.2 Consultation et modification des permissions

`ls -l`

```
$ ls -l
total 184
-rw-rw-r-- 1 alexis www-data 122 sep 23 14:33 index.html
...
```

Cette commande permet de lister le contenu d'un répertoire avec les permissions, ainsi que le propriétaire et le groupe de chaque fichier. Pour chaque fichier, les permissions sont affichées sous le format suivant : permissions du propriétaire, permissions du groupe, permissions des autres. Pour chacune de ces trois entités, les permissions se composent de trois caractères qui indiquent, dans l'ordre, si l'utilisateur a les permissions en lecture, en écriture et en exécution. Le caractère « `r` », « `w` », ou « `x` » indique la présence d'une permission, et le caractère « `-` » son absence.

Dans l'exemple ci-dessus, le fichier `index.html` a pour propriétaire `alexis` et pour groupe `www-data`. Par conséquent, `alexis` et les membres du groupe `www-data` peuvent lire et modifier le fichier. Les autres peuvent seulement le lire. Dans ce cas, si un autre utilisateur tente de modifier le fichier `index.html`, il verra le message : `Permission denied`.

Avant les permissions du propriétaire, le `-` indique qu'il s'agit d'un fichier normal.

`stat`

La commande `stat` affiche des informations sur les fichiers.

```
$ stat documents/
File: documents/
Size: 4096       Blocks: 8       IO Block: 4096   directory
Device: 806h/2054d Inode: 4980748  Links: 28
Access: (0755/drwxr-xr-x)  Uid: ( 1000/  alexis)   Gid: ( 1000/  alexis)
Access: 2021-10-06 09:37:19.187369041 +0200
Modify: 2021-09-25 20:58:59.157211048 +0200
Change: 2021-09-25 20:58:59.157211048 +0200
Birth: -
```

L'option `--format` permet d'afficher exactement l'information voulue. Par exemple, `--format '%A'` permet d'obtenir le ***mode*** du fichier (les permissions précédées du type).

mode

```
$ stat --format '%A' documents/
drwxr-xr-x
```

Ici, le `d` indique qu'il s'agit d'un répertoire (*directory*).

`chown`

La commande `chown` (*change owner*) permet de changer le propriétaire d'un fichier. Seul le propriétaire actuel du fichier (ou le super-utilisateur) peut l'exécuter. Sa syntaxe est `chown nouveau_propriétaire fichier1 fichier2 ...`.

### `chgrp` et `groups`

La commande `chgrp` (*change group*) permet de changer le groupe associé à un fichier. Sa syntaxe est `chgrp nouveau_groupe fichier1 fichier2 ...`, où `nouveau_groupe` est soit le numéro du nouveau groupe soit son nom. La commande `groups user` permet de consulter les groupes auxquels appartient l'utilisateur `user`.

### `chmod`

La commande `chmod` (*change mode*) permet de changer les permissions d'un fichier. Elle n'est accessible qu'au propriétaire du fichier et au super-utilisateur. Sa syntaxe est `chmod mode fichier1 fichier2...`, où `mode` est de la forme `{a,u,g,o }{+,-, = }{r,x,w }`.

Les lettres `a`, `u`, `g`, `o` indiquent respectivement tous les utilisateurs (*all*), le propriétaire (*user*), le groupe (*group*) et les autres (*others*). On peut en spécifier plusieurs. Le caractère « + » dénote qu'on veut rajouter des permissions, le caractère « - » qu'on veut en supprimer et le caractère « `=` » désigne qu'on veut mettre exactement ces permissions. Finalement, on spécifie à quelles permissions précisément la commande s'applique, en spécifiant une ou plusieurs lettres parmi `{r,x,w }`.

Par exemple, la commande `chmod ug+rw brol` permet d'ajouter au fichier « `brol` » les permissions en lecture et en écriture, pour le propriétaire et les utilisateurs appartenant au groupe du fichier.

Les permissions peuvent aussi être indiquées en « octal ». Les permissions sont alors indiquées comme un nombre en base 8 à trois chiffres. Les chiffres indiquent dans l'ordre les permissions pour le propriétaire, pour les groupes et pour les autres. 4 signifie une permission en lecture seule, 2 en écriture et 1 en exécution. On combine les permissions en additionnant les valeurs. (Vous reconnaîtrez le fonctionnement d'un masque binaire.) Une valeur en octale s'écrit en C en commençant la valeur par un 0 (ex. : *0777*).

## 1.3 exercices

---

**Exercice 1.** *Se déplacer dans le répertoire `/home` et lire son contenu. Y créer un autre répertoire. Cela fonctionne-t-il ? Pourquoi ? Si non, pourquoi avez-vous quand même pu vous positionner dans ce répertoire et en lire son contenu ?*

---

**Exercice 2.** *Essayez de vous placer dans le répertoire personnel de l'un de vos camarades. Cela fonctionne-t-il ? Pourquoi ? Essayez de lister le contenu de ce répertoire ? Le pouvez-vous ? Pourquoi ? (Supposez que `/home` contient un dossier `qqun` avec les permissions 711.)*

---

**Exercice 3.** *Supposons que vous vouliez avoir un sous-répertoire public dans votre répertoire personnel, dans lequel vos camarades pourraient écrire des fichiers à votre attention, comment procéderiez-vous ? Imaginez et testez une solution. Vos amis peuvent-ils découvrir par eux-mêmes le nom de ce répertoire ou doivent-ils le connaître à l'avance ? Pourquoi ? Comment modifier cela ? Testez.*

```
root@workstation $ ls -la
total 348
drwxrwxr-x  3 adam   adam      4096 juin  15 13:33 .
drwxrwxrwt 26 root   root     20480 juin  17 17:29 ..
-rwxr-x--x  1 adam   eve         28 juin  16 19:14 chose
dr-xr-x---  2 adam   eve       4096 juin  17 17:59 documents
-rw-rw-r--  1 adam   logger 319502 juin  18 20:32 image.jpg
-rw-rw-r--  1 eve    logger      8 juin  18 17:33 log.txt
root@workstation $ touch machin # Créer le fichier machin
```

LISTING 1 – Session bash par le superutilisateur `root`

---

**Exercice 4.** *Considérons la session bash en figure 1.*

1. *Que signifie `rwxr-x--x` dans la sortie du `ls` ?*
2. *Le fichier `machin` a les attributs `rw-r--r--`. Quelle fonctionnalité / outil permet de modifier les attributs donnés aux fichiers à leur création ?*
3. *Adam a-t-il les permissions pour effectuer chacune des actions ci-dessous ? Si oui, indiquez grâce à quelle permission. Si non, indiquez la permission manquante. Si on ne peut pas le déterminer, indiquez les conditions nécessaires pour que l'action soit permise.*
   (a) *Créer un fichier avec `touch documents/message.txt`.*
   (b) *Créer un dossier avec `mkdir documents/messages/`.*
   (c) *Écraser `log.txt` avec `echo '' > log.txt`.*
   (d) *Supprimer `image.jpg` avec `rm image.jpg`.*

## 2 Lecture et écriture d'un fichier en C

Il s'agit d'un sujet que vous maîtrisez déjà. Alors voici juste quelques rappels. On ouvre un fichier avec la fonction `open`, on y lit et écrit avec `read` et `write` et on le ferme avec `close`.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
// Si O_CREAT est utilisé :
int open(const char *pathname, int flags, mode_t mode);

#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);

int dup2(int src_fd, int dst_fd);
```

**Pour `open`** il faut spécifier un des trois flags suivant : `O_RDONLY` (ouverture en lecture), `O_WRONLY` (ouverture en écriture), `O_RDWR` (ouverture en lecture et écriture). On peut indiquer d'autres flags à l'ouverture (par exemple `open("file", O_RDONLY | O_APPEND)`), dont voici les principaux :

`O_APPEND` effectuer tous les write() à la fin du fichier.

`O_CREAT` si le fichier n'existe pas, le créer. (Nécessite l'ajout d'un 3e paramètre contenant les permissions encodées en base 8.)

`O_PATH` ne pas ouvrir le fichier.

`O_TRUNC` si le fichier existe déjà, efface son contenu à l'ouverture.

**La fonction `read`** lit au plus `count` octets et les écrit à l'endroit pointé par `buf`. S'il y a moins d'octet à lire, `read` lit les octets disponibles. S'il n'y a pas eu d'erreur `read` retourne le nombre d'octets lus. En cas d'erreur, la fonction retourne -1.

**La fonction `write`** fonctionne d'une façon symétrique à `read`.

**La fonction `close`** ferme le fichier et retourne 0 en cas de réussite.

**La fonction `dup2`** duplique le descripteur de fichier `oldfd` dans le descripteur de fichier `newfd`. Si `newfd` était un descripteur de fichier valide, il est d'abord fermé silencieusement avant de référencer la même chose que `oldfd`. Fonctionne quelque soit la ressource référencée par `oldfd` (fichier, socket, pipe, flux clavier, etc). La fonction retourne le descripteur de fichier `newfd` en cas de succès et -1 sinon.

Cet appel système peut être utilisé pour rediriger un flux standard par exemple :

```
// À cause d'O_CREAT, il faut ajouter le paramètre 0644 spécifiant
// les permissions.
int nouveau_stdout = open("sortie.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (nouveau_stdout == -1) {
   perror("open()");
   exit(1);
}

// Remplacer stdout (fd #1) par un fichier
if (dup2(nouveau_stdout, 1) == -1) {
   perror("dup2()");
   exit(1);
}

close(nouveau_stdout); // Faclt
```

---

**Exercice 5.** *Créez un programme qui lance la commande `ls` mais en redirigeant sa sortie standard `stdout` vers le fichier `stdout.txt` à l'aide de `dup2`.*

## 3 Lecture et écriture d'un fichier en Bash

En bash, la façon habituelle de faire de la lecture et écriture de fichiers consiste à rediriger.

Les programmes utilitaires de base peuvent être vus comme des fonctions. Ces programmes lisent par défaut le contenu sur lequel ils doivent travailler à partir de leur *file descriptor* 0 et écrivent les résultats avec des `write` s sur le *file descriptor* 1.

La façon de faire est donc de modifier l'environnement des programmes appelés pour que leurs *file descriptors* se réfèrent aux fichiers voulus (ouverts pour eux avant leur lancement et fermés

quand ils se terminent). C'est ce que font les redirections que nous avions vues au premier TP. Vous pouvez aussi revoir le TP sur les pipes. Le fonctionnement pour des simples fichiers est similaire :

Conceptuellement, pour exécuter `echo "Hello" > fichier`, bash fait un `int fd = open("fichier", O_WRONLY | O_TRUNC)`. Il lance ensuite `echo` en s'assurant que l'entrée 1 de la table des fichiers de (cette instance de) `echo` soit la même chose que l'entrée `fd` de sa table de fichiers. Une fois qu'`echo` s'est terminé, bash exécute un `close(fd)`.

En ignorant le fait que `echo` est en général directement géré par l'interpréteur sans lancer `/bin/echo`, ainsi que d'autres détails, le code C en figure 2 illustre l'exécution de `echo "Hello" > fichier` par un shell.

```c
if ((pid = fork()) < 0) {
  // … Error
} else if (pid == 0) {
  int fd_out = open("fichier", O_WRONLY | O_TRUNC); // TODO handle error
  dup2(fd_out, STDOUT_FILENO); // TODO handle error
  close(fd_out);
  execlp("echo", "echo", "Hello", NULL);
  // … Error
} else {
  // … Shell
}
```

Listing 2 – Exécution de `echo "Hello" > fichier` par un shell

(Les quatre phrases suivantes ne font pas partie de la matière à connaître à l'examen.) Il est aussi possible d'ouvrir un fichier en écriture avec `exec {fd}> fichier` et en lecture avec `exec {fd}< fichier`, où `fd` est le numéro d'un descripteur de fichier. On écrit/lit ensuite avec les redirections habituelles, mais en utilisant `&{fd}` à la place du nom de fichier. On ferme ensuite le fichier avec `exec >&{fd}-` ou `exec <&{fd}-` suivant le cas. Pour ouvrir en lecture et écriture, on utilise la redirection `<>` dans les commandes avec `exec` ci-dessus.

# 4 Systèmes de fichiers

Les systèmes de fichiers permettent un stockage d'informations à long terme tout en abstrayant le type de support (disque dur, clef USB, bande magnétique, ...).

Ils doivent remplir certaines conditions essentielles : stocker de grandes quantités d'informations, assurer la persistance à long terme et pouvoir récupérer des informations stockées.

Pour ces exercices, nous considérerons le support de stockage comme une séquence linéaire de blocs de taille fixe où nous pouvons allouer et désallouer chaque bloc afin de stocker des informations.

## 4.1 Allocation contiguë

L'allocation contiguë est sans doute le système le plus simple car elle consiste simplement à ajouter des fichiers les uns à la suite des autres dans le premier emplacement suffisamment grand pour accueillir le fichier.

**Exercice 6.** *Supposons un stockage composé de 20 blocs et les fichiers suivants ;*

| *Fichier* | *taille* |
|---|---|
| *A* | *4* |
| *B* | *6* |
| *C* | *3* |
| *D* | *4* |

1. *Utilisez l'allocation contiguë pour dessiner l'état de la mémoire (c.-à-d. quels blocs sont alloués pour quel fichier) lors de l'ajout de tous les fichiers.*
2. *Supprimez le fichier B*
3. *Ajoutez un nouveau fichier E de trois blocs.*
4. *Quels problèmes voyez-vous venir sur le long terme avec l'allocation contiguë ?*

Lorsqu'on parle de stockage de fichiers, on a des problèmes de fragmentation similaire à ceux en mémoire principale.

**La fragmentation interne** vient de l'utilisation de blocs de plusieurs octets : pour un fichier dont la taille n'est pas un multiple de la taille d'un bloc, le système doit allouer trop d'espace.

**La fragmentation externe** a lieu lorsque l'espace libre total est répartie en petits segments entre les espaces alloués. Ces petits segments sont plus difficilement utilisable.

## 4.2 Inodes

L'***inode*** est une structure de données (utilisée dans un système de fichiers de style Unix) qui décrit un ***fichier*** ou tout autre objet du système de fichiers tel qu'un répertoire, un lien symbolique, un socket… L'inode stocke toutes les métadonnées de l'objet ainsi que les emplacements de ses données. Le nom n'est pas stocké dans l'inode. En effet, un inode peut avoir plusieurs noms. Ces noms sont en fait des données des répertoires le contenant.

inode

fichier

Les ***répertoires*** sont des fichiers spéciaux qui contiennent des listes de noms attribués aux inodes. Un répertoire contient une entrée (tuple (inode, type[1], nom)) pour lui-même, son parent et chacun de ses enfants.

répertoires

**Exercice 7.** *Combien d'opérations sur le disque sont nécessaires pour récupérer l'inode d'un fichier dont le chemin est* `/usr/ast/courses/os/handout.t`*? Nous supposerons que l'inode du répertoire racine est en mémoire mais qu'aucun autre élément le long du chemin n'est en mémoire. Nous supposerons également que tous les répertoires tiennent dans un bloc de disque.*

**Exercice 8.** *Étant donné un disque dur avec un système de fichiers de type Unix avec les spécifications suivantes : la taille du bloc est de 4 kio[2], la longueur de l'adresse du bloc est de 4 octets et les inodes ont une structure traditionnelle (10 pointeurs directs, 1 pointeur avec des indirections simples, 1 pointeur avec des doubles indirections et 1 pointeur avec des triples indirections).*

1. Généralement.
2. k- : kilo $\equiv \times 10^3$ ; ki- : kibi $\equiv \times 2^{10}$

a) *Quel est le nombre de blocs (y compris les blocs de données et d'adresse) qui contiennent les fichiers suivants :*
   - *Fichier A d'une taille de 20 kio*
   - *Fichier B d'une taille de 200 kio*
   - *Fichier C d'une taille de 2 000 kio*
   - *Fichier D d'une taille de 20 000 kio*

b) *Quelle est la taille maximale du fichier ?* [3]

3. En réalité, la taille maximale du fichier sera limitée par un attribut du fichier (`i_blocks`, enregistré dans l'inode), qui indique le nombre total de blocs de 512 octets utilisés et réservés pour le fichier, indépendamment de la taille *réelle* d'un bloc indiquée dans le super-bloc. ($512 \times 2^{32} = 2\,199\,023\,255\,552 = 2$ Tio)