Gestion des utilisateurs

Unix est un système d'exploitation multi-utilisateurs. Un tel système impose des contraintes de sécurité qui n'existent pas sur un système mono-utilisateur. Il est intéressant de passer en revue quelques unes de ces contraintes :

- il doit être possible d'identifier et/ou d'authentifier les utilisateurs du système
- il doit être possible d'exécuter des processus appartenant à plusieurs utilisateurs simultanément et de déterminer quel utilisateur est responsable de chaque opération
- le système d'exploitation doit fournir des mécanismes simples qui permettent de contrôler l'accès aux différentes ressources (mémoire, stockage, ...).
- il doit être possible d'allouer certaines ressources à un utilisateur particulier à un moment donné

Aujourd'hui, la plupart des systèmes informatiques demandent une authentification de l'utilisateur sous la forme d'un mot de passe, d'une manipulation particulière voire d'une identification biométrique comme une empreinte digitale. Cette authentification permet de vérifier que l'utilisateur est autorisé à manipuler le système informatique. Cela n'a pas toujours été le cas et de nombreux systèmes informatiques plus anciens étaient conçus pour être utilisés par un seul utilisateur qui était simplement celui qui interagissait physiquement avec l'ordinateur.

Les systèmes Unix supportent différents mécanismes d'authentification. Le plus simple et le plus utilisé est l'authentification par mot de passe. Chaque utilisateur est identifié par un nom d'utilisateur et il doit prouver son identité en tapant son mot de passe au démarrage de toute session sur le système. En pratique, une session peut s'établir localement sur l'ordinateur via son interface graphique par exemple ou à distance en faisant tourner un serveur tel que **sshd(8)** sur le système Unix et en permettant aux utilisateurs de s'y connecter via Internet en utilisant un client **ssh(1)**. Dans les deux cas, le système d'exploitation lance un processus **login(1)** qui permet de vérifier le nom d'utilisateur et le mot de passe fourni par l'utilisateur. Si le mot de passe correspond à celui qui est stocké sur le système, l'utilisateur est authentifié et son shell peut démarrer. Sinon, l'accès au système est refusé.

Lorsqu'un utilisateur se connecte sur un système Unix, il fournit son nom d'utilisateur ou *username*. Ce nom d'utilisateur est une chaîne de caractères qui est facile à mémoriser par l'utilisateur. D'un point de vue implémentation, un système d'exploitation préfère manipuler des nombres plutôt que des chaînes de caractères. Unix associe à chaque utilisateur un identifiant qui est stocké sous la forme d'un nombre entier positif. La table de correspondance entre l'identifiant d'utilisateur et le nom d'utilisateur est le fichier */etc/passwd*. Ce fichier texte, comme la grande majorité des fichiers de configuration d'un système Unix, comprend pour chaque utilisateur l'information suivante :

- nom d'utilisateur (*username*)
- mot de passe (sur les anciennes versions de Unix)
- identifiant de l'utilisateur (**userid**)
- identifiant du groupe principal auquel l'utilisateur appartient
- nom et prénom de l'utilisateur
- répertoire de démarrage de l'utilisateur
- shell de l'utilisateur

L'extrait ci-dessous présente un exemple de fichier `/etc/passwd`. Des détails complémentaires sont disponibles dans la page de manuel **passwd(5)**. Un utilisateur peut modifier les informations le concernant dans ce fichier avec la commande **passwd(1)**.

```
# Exemple de /etc/passwd
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
slampion:*:1252:1252:Séraphin Lampion:/home/slampion:/bin/bash
```

Il y a en pratique trois types d'utilisateurs sur un système Unix. L'utilisateur **root** est l'administrateur du système. C'est l'utilisateur qui a le droit de réaliser toutes les opérations sur le système. Il peut créer de nouveaux utilisateurs, mais aussi formatter les disques, arrêter le système, interrompre des processus utilisateurs ou accéder à l'ensemble des fichiers sans restriction. Par convention, cet utilisateur a l'identifiant `0`. Ensuite, il y a tous les utilisateurs *normaux* du système Unix. Ceux-ci ont le droit d'accéder à leurs fichiers, d'interagir avec leurs processus mais en général ne peuvent pas manipuler les fichiers d'autres utilisateurs ou interrompre leurs processus. L'utilisateur *slampion* dans l'exemple ci-dessus est un utilisateur *normal*. Enfin, pour faciliter l'administration du système, certains systèmes Unix utilisent des utilisateurs qui correspondent à un service particulier comme l'utilisateur *daemon* dans l'exemple ci-dessus. Une discussion de ce type d'utilisateur sort du cadre de ces notes. Le lecteur intéressé pourra consulter une référence sur l'administration des système Unix telle que **[AdelsteinLubanovic2007]** ou **[Nemeth+2010]**.

Unix associe à chaque processus un identifiant d'utilisateur. Cet identifiant est stocké dans l'entrée du processus dans la table des processus. Un processus peut récupérer son identifiant d'utilisateur via l'appel système **getuid(2)**. Outre cet appel système, il existe également l'appel système **setuid(2)** qui permet de modifier le **userid** du processus en cours d'exécution. Pour des raisons évidentes de sécurité, seul un processus appartenant à l'administrateur système (**root**) peut exécuter cet appel système. C'est le cas par exemple du processus **login(1)** qui appartient initialement à **root** puis exécute **setuid(2)** afin d'appartenir à l'utilisateur authentifié puis exécute **execve(2)** pour lancer le premier shell appartenant à l'utilisateur.

En pratique, il est parfois utile d'associer des droits d'accès à des groupes d'utilisateurs plutôt qu'à un utilisateur particulier. Par exemple, un département universitaire peut avoir un groupe correspondant à tous les étudiants et un autre aux membres du staff pour leur donner des permissions différentes. Un utilisateur peut appartenir à un groupe principal et plusieurs groupes secondaires. Le groupe principal est spécifié dans le fichier **passwd(5)** tandis que le fichier `/etc/group` décrit dans **group(5)** contient les groupes secondaires.

Systèmes de fichiers

Outre un processeur et une mémoire, la plupart des ordinateurs actuels sont en général équipés d'un ou plusieurs dispositifs de stockage. Les dispositifs les plus courants sont le disque dur, le lecteur de CD/DVD, la clé USB, la carte mémoire, ... Ces dispositifs de stockage ont des caractéristiques techniques très différentes. Certains stockent l'information sous forme magnétique, d'autres sous forme électrique ou en creusant via un laser des trous dans un support physique. D'un point de vue logique, ils offrent tous une interface très similaire au système d'exploitation qui veut les utiliser.

Dans un système de fichiers Unix, l'ensemble des répertoires et fichiers est organisé sous la forme d'un arbre. La racine de cet arbre est le répertoire `/`. Il est localisé sur un des dispositifs de stockage du système. Le système de fichiers Unix permet d'intégrer facilement des systèmes de fichiers qui se trouvent sur différents dispositifs de stockage. Cette opération est en général réalisée par l'administrateur système en utilisant la commande **mount(8)**. A titre d'exemple, voici quelques répertoires qui sont montés sur un système Linux.

```
$ df -k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             15116836   9425020   4923912  66% /
tmpfs                  1559516      4724   1554792   1% /dev/shm
/dev/sda2             19840924   4374616  14442168  24% /xendomains
xenstore               1972388        40   1972348   1% /var/lib/xenstored
david:/mnt/student   258547072 188106176  59935296  76% /etinfo/users
david:/mnt/staff     254696800 177422400  64136160  74% /etinfo/users2
```

Dans l'exemple ci-dessus, la première colonne correspond au dispositif de stockage qui contient les fichiers et répertoires. Le premier dispositif de stockage `/dev/sda1/` est un disque local et contient le répertoire racine du système de fichiers. Le système de fichiers `david://mnt/student` est stocké sur le serveur `david` et est monté via **mount(8)** dans le répertoire `/etinfo/users`. Ainsi, tout accès à un fichier dans le répertoire `/etinfo/users` se fera via le serveur `david`.

Chaque répertoire du système de fichiers contient un ou plusieurs répertoires et un ou plusieurs fichiers. A titre d'exemple, il est intéressant de regarder le contenu de deux répertoires. Le premier est un extrait au contenu du répertoire racine obtenu avec la commande **ls(1)**

```
$ ls -la /
dr-xr-xr-x.  26 root root 233472 Feb 23 03:18 .
dr-xr-xr-x.  26 root root 233472 Feb 23 03:18 ..
-rw-r--r--    1 root root      0 Feb 13 16:45 .autofsck
-rw-r--r--    1 root root      0 Jul 27  2011 .autorelabel
dr-xr-xr-x.   4 root root   4096 Dec 15 05:50 boot
drwxr-xr-x   19 root root   4160 Mar 22 12:04 dev
drwxr-xr-x. 125 root root  12288 Mar 22 12:04 etc
drwxr-xr-x    4 root root      0 Mar 22 10:22 etinfo
drwxr-xr-x.   2 root root   4096 Jan  6  2011 home
dr-xr-xr-x.  14 root root   4096 Mar 22 03:26 lib
dr-xr-xr-x.  10 root root  12288 Mar 22 03:26 lib64
drwx------.   2 root root  16384 Jul 27  2011 lost+found
...
drwxrwxrwt. 104 root root   4096 Mar 22 12:05 tmp
drwxr-xr-x.  13 root root   4096 Jul 19  2011 usr
drwxr-xr-x.  23 root root   4096 Jul 27  2011 var
```

Le répertoire racine contient quelques fichiers et des répertoires. Tout répertoire contient deux répertoires spéciaux. Le premier répertoire, identifié par le caractère `.` (un seul point) est un alias vers le répertoire lui-même. Cette entrée de répertoire est présente dans chaque répertoire dès qu'il est créé avec une commande telle que **mkdir(1)**. Le deuxième répertoire spécial est `..` (deux points consécutifs). Ce répertoire est un alias vers le répertoire parent du répertoire courant.

Les méta-données qui sont associées à chaque fichier ou répertoire contiennent, outre les informations de type, les bits de permission. Ceux-ci permettent d'encoder trois types de permissions et d'autorisation :

- `r` : autorisation de lecture
- `w` : autorisation d'écriture ou de modification
- `x` : autorisation d'exécution

Ces bits de permissions sont regroupés en trois blocs. Le premier bloc correspond aux bits de permission qui sont applicables pour les accès qui sont effectués par un processus qui appartient à l'utilisateur qui est propriétaire du fichier/répertoire. Le deuxième bloc correspond aux bits de permission qui sont applicables pour les opérations effectuées par un processus dont l'identifiant de groupe est identique à l'identifiant de groupe du fichier/répertoire mais n'appartient pas à l'utilisateur qui est propriétaire du fichier/répertoire. Le dernier bloc est applicable pour les opérations effectuées par des processus qui appartiennent à d'autres utilisateurs.

Les valeurs de ces bits sont représentés pas les symboles `rwx` dans l'output de la commande **ls(1)**. Les bits de permission peuvent être modifiés en utilisant la commande **chmod(1)** qui utilise l'appel système **chmod(2)**. Pour qu'un exécutable puisse être exécuté via l'appel système **execve(2)**, il est nécessaire que le fichier correspondant possède les bits de permission `r` et `x`.

Note

Manipulation des bits de permission avec **chmod(2)**

L'appel système **chmod(2)** permet de modifier les bits de permission qui sont associés à un fichier. Ceux-ci sont encodés sous la forme d'un entier sur 16 bits.

- `S_IRUSR (00400)` : permission de lecture par le propriétaire
- `S_IWUSR (00200)` : permission d'écriture par le propriétaire
- `S_IXUSR (00100)` : permission d'exécution par le propriétaire

- `S_IRGRP (00040)` : permission de lecture par le groupe hormis le propriétaire
- `S_IWGRP (00020)` : permission d'écriture par le groupe hormis le propriétaire
- `S_IXGRP (00010)` : permission d'exécution par le groupe hormis le propriétaire
- `S_IROTH (00004)` : permission de lecture par tout utilisateur hormis le propriétaire et son groupe
- `S_IWOTH (00002)` : permission d'écriture par tout utilisateur hormis le propriétaire et son groupe
- `S_IXOTH (00001)` : permission d'exécution par tout utilisateur hormis le propriétaire et son groupe

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

Ces bits de permissions sont généralement spécifiés soit sous la forme d'une disjonction logique ou sous forme numérique. A titre d'exemple, un fichier qui peut être lu et écrit uniquement par son propriétaire aura comme permissions `00600` ou `S_IRUSR|S_IWUSR`.

Le **nibble** de poids fort des bits de permission sert à encoder des permissions particulières relatives aux fichiers et répertoires. Par exemple, lorsque la permission `S_ISUID (04000)` est associée à un exécutable, elle indique que celui-ci doit s'exécuter avec les permissions du propriétaire de l'exécutable et pas les permissions de l'utilisateur. Cette permission spéciale est utilisée par des programmes comme **passwd(1)** qui doivent disposer des permissions de l'administrateur système pour s'exécuter correctement (**passwd(1)** doit modifier le fichier **passwd(5)** qui appartient à l'administrateur système).

Les exemples ci-dessous présentent le contenu partiel d'un répertoire.

```
$ ls -lai /etinfo/users/obo
total 1584396
drwx------ 78 obo  stafinfo       4096 Mar 17 00:34 .
drwxr-xr-x 93 root root           4096 Feb 22 11:37 ..
-rwxr-xr-x  1 obo  stafinfo      11490 Feb 28 00:43 a.out
-rw-------  1 obo  stafinfo       4055 Mar 22 15:13 .bash_history
-rw-r--r--  1 obo  stafinfo         55 Sep 18  1995 .bash_profile
-rw-r--r--  1 obo  stafinfo        101 Aug 28  2003 .bashrc
drwxr-xr-x  2 obo  stafinfo       4096 Nov 22  2004 bin
-rw-r--r--  1 obo  stafinfo        346 Feb 13 15:37 hello.c
drwxr-xr-x  3 obo  stafinfo       4096 Mar  2 09:30 sinf1252
drwxr-xr-x  2 obo  stafinfo       4096 May 17  2011 src
```

Dans un système Unix, que ce soit au niveau du shell ou dans n'importe quel processus écrit par exemple en langage C, les fichiers peuvent être spécifiés de deux façons. La première est d'indiquer le chemin complet depuis la racine qui permet d'accéder au fichier. Le chemin `/etinfo/users/obo` passé comme argument à la commande **ls(1)** ci-dessus en est un exemple. Le premier caractère `/` correspond à la racine du système de fichiers et ensuite ce caractère est utilisé comme séparateur entre les répertoires successifs. Ainsi, le fichier `/etinfo/users/obo/hello.c` est un fichier qui a comme nom `hello.c` qui se trouve dans un répertoire nommé `obo` qui lui-même se trouve dans le répertoire `users` qui est dans le répertoire baptisé `etinfo` dans le répertoire racine. La seconde façon de spécifier un nom de fichier est de préciser son nom relatif. Pour éviter de forcer l'utilisateur à spécifier chaque fois le nom complet des fichiers et répertoires auxquels il veut accéder, le noyau maintient dans sa table des processus le **répertoire courant** de chaque processus. Par défaut, lorsqu'un processus est lancé, son répertoire courant est le répertoire à partir duquel le programme a été lancé. Ainsi, lorsque l'utilisateur tape une commande comme `gcc hello.c` depuis son shell, le processus **gcc(1)** peut directement accéder au fichier `hello.c` qui se situe dans le répertoire courant. Un processus peut modifier son répertoire courant en utilisant l'appel système **chdir(2)**.

```
#include <unistd.h>

int chdir(const char *path);
```

Cet appel système prend comme argument une chaîne de caractères contenant le nom du nouveau répertoire courant. Ce nom peut être soit un nom complet (commençant par `/`), ou un nom relatif au répertoire courant actuel. Dans ce cas, il est parfois utile de pouvoir référer au répertoire parent du répertoire courant. Cela se fait en utilisant `..` Dans chaque répertoire, cet alias correspond au répertoire parent. Ainsi, si le répertoire courant est `/etinfo/users`, alors le répertoire `../../bin` est le répertoire `bin` se trouvant dans le répertoire racine. Depuis le shell, il est possible de modifier le répertoire courant avec la commande **cd(1posix)**. La commande **pwd(1)** affiche le répertoire courant actuel.

Il existe plusieurs appels systèmes et fonctions de la librairie standard qui permettent de parcourir le système de fichiers. Les principaux sont :

- l'appel système **stat(2)** permet de récupérer les méta-données qui sont associées à un fichier ou un répertoire. La commande **stat(1)** fournit des fonctionnalités similaires depuis le shell.
- les appels systèmes **chmod(2)** et **chown(2)** permettent de modifier respectivement le mode (i.e. les permissions), le propriétaire et le groupe associés à un fichier. Les commandes **chmod(1)**, **chown(1)** et **chgrp(1)** permettent de faire de même depuis le shell.
- l'appel système **utime(2)** permet de modifier les dates de création/modification associées à un fichier/répertoire. Cet appel système est utilisé par la commande **touch(1)**
- l'appel système **rename(2)** permet de changer le nom d'un fichier ou d'un répertoire. Il est utilisé notamment par la commande **rename(1)**
- l'appel système **mkdir(2)** permet de créer un répertoire alors que l'appel système **rmdir(2)** permet d'en supprimer un
- les fonctions de la librairie **opendir(3)**, **closedir(3)**, et **readdir(3)** permettent de consulter le contenu de répertoires.

Les fonctions de manipulation des répertoires méritent que l'on s'y attarde un peu. Un répertoire est un fichier qui a une structure spéciale. Ces trois fonctions permettent d'en extraire de l'information en respectant le format d'un répertoire. Pour accéder à un répertoire, il faut d'abord l'ouvrir en utilisant **opendir(3)**. La

fonction **readdir(3)** permet d'accéder aux différentes entrées de ce répertoire et **closedir(3)** doit être utilisée lorsque l'accès n'est plus nécessaire. La fonction **readdir(3)** permet de manipuler la structure `dirent` qui est définie dans **bits/dirent.h**.

```
struct dirent {
     ino_t          d_ino;       /* inode number */
     off_t          d_off;       /* offset to the next dirent */
     unsigned short d_reclen;    /* length of this record */
     unsigned char  d_type;      /* type of file; not supported
                                    by all file system types */
     char           d_name[256]; /* filename */
 };
```

Cette structure comprend le numéro de l'inode, c'est-à-dire la métadonnée qui contient les informations relatives au fichier/répertoire, la position de l'entrée `dirent` qui suite, la longueur de l'entrée, son type et le nom de l'entrée dans le répertoire. Chaque appel à **readdir(3)** retourne un pointeur vers une structure de ce type.

L'extrait de code ci-dessous permet de lister tous les fichiers présents dans le répertoire `name`.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

void exit_on_error(char *s) {
  perror(s);
  exit(EXIT_FAILURE);
}

int main (int argc, char *argv[])  {

  DIR *dirp;
  struct dirent *dp;
  char name[]=".";
  dirp = opendir(name);
  if(dirp==NULL) {
    exit_on_error("opendir");
  }
  while ((dp = readdir(dirp)) != NULL) {
    printf("%s\n",dp->d_name);
  }
  int err = closedir(dirp);
  if(err<0) {
    exit_on_error("closedir");
  }


}
```

La lecture d'un répertoire avec **readdir(3)** commence au début de ce répertoire. A chaque appel à **readdir(3)**, le programme appelant récupère un pointeur vers une zone mémoire contenant une structure `dirent` avec l'entrée suivante du répertoire ou `NULL` lorsque la fin du répertoire est atteinte. Si une fonction doit relire à nouveau un répertoire, cela peut se faire en utilisant **seekdir(3)** ou **rewinddir(3)**.

Note

readdir(3) et les threads

La fonction **readdir(3)** est un exemple de fonction non-réentrante qu'il faut éviter d'utiliser dans une application dont plusieurs threads doivent pouvoir parcourir le même répertoire. Ce problème est causé par l'utilisation d'une zone de mémoire `static` afin de stocker la structure dont le pointeur est retourné par **readdir(3)**. Dans une application utilisant plusieurs threads, il faut utiliser la fonction **readdir_r(3)** :

```
int  readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
               struct dirent **restrict result);
```

Cette fonction prend comme arguments le pointeur `entry` vers un buffer propre à l'appelant qui permet de stocker le résultat de **readdir_r(3)**.

Les appels systèmes **link(2)** et **unlink(2)** sont un peu particuliers et méritent une description plus détaillée. Sous Unix, un **inode** est associé à chaque fichier mais l'**inode** ne contient pas le nom de fichier parmi les méta-données qu'il stocke. Par contre, chaque **inode** contient un compteur (`nlinks`) du nombre de liens vers un fichier. Cela permet d'avoir une seule copie d'un fichier qui est accessible depuis plusieurs répertoires. Pour comprendre cette utilisation des liens sur un système de fichiers Unix, considérons le scénario suivant.

```
$ mkdir a
$ mkdir b
$ cd a
```

```
$ echo "test" > test.txt
$ cd ..
$ ln a/test.txt a/test2.txt
$ ls -li a
total 16
9624126 -rw-r--r--  2 obo  stafinfo  5 24 mar 21:14 test.txt
9624126 -rw-r--r--  2 obo  stafinfo  5 24 mar 21:14 test2.txt
$ ln a/test.txt b/test3.txt
$ stat --format "inode=%i nlinks=%h" b/test3.txt
inode=9624126 nlinks=3
$ ls -li b
total 8
9624126 -rw-r--r--  3 obo  stafinfo  5 24 mar 21:14 test3.txt
$ echo "complement" >> b/test3.txt
$ ls -li a
total 16
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test.txt
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test2.txt
$ ls -li b
total 8
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test3.txt
$ cat b/test3.txt
test
complement
$ cat a/test.txt
test
complement
$ rm a/test2.txt
$ ls -li a
total 8
9624126 -rw-r--r--  2 obo  stafinfo  16 24 mar 21:15 test.txt
$ rm a/test.txt
$ ls -li a
$ ls -li b
total 8
9624126 -rw-r--r--  1 obo  stafinfo  16 24 mar 21:15 test3.txt
```

Dans ce scénario, deux répertoires sont créés avec la commande **mkdir(1)**. Ensuite, la commande **echo(1)** est utilisée pour créer le fichier `test.txt` contenant la chaîne de caractères `test` dans le répertoire `a`. Ce fichier est associé à l'**inode** `9624126`. La commande **ln(1)** permet de rendre ce fichier accessible sous un autre nom depuis le même répertoire. La sortie produite par la commande **ls(1)** indique que ces deux fichiers qui sont présents dans le répertoire `a` ont tous les deux le même `inode`. Ils correspondent donc aux mêmes données sur le disque. A ce moment, le compteur `nlinks` de l'**inode** `9624126` a la valeur `2`. La commande **ln(1)** peut être utilisée pour créer un lien vers un fichier qui se trouve dans un autre répertoire **[4]** comme le montre la création du fichier `test3.txt` dans le répertoire `b`. Ces trois fichiers correspondant au même **inode**, toute modification à l'un des fichiers affecte et est visible dans n'importe lequel des liens vers ce fichier. C'est ce que l'on voit lorsque la commande `echo "complement" >> b/test3.txt` est exécutée. Cette commande affecte immédiatement les trois fichiers. La commande `rm a/test2.txt` efface la référence du fichier sous le nom `a/test2.txt`, mais les deux autres liens restent accessibles. Le fichier ne sera réellement effacé qu'après que le dernier lien vers l'**inode** correspondant aie été supprimé. La commande **rm(1)** utilise en pratique l'appel système **unlink(2)** qui en toute généralité décrémente le compteur de liens de l'inode correspondant au fichier et l'efface lorsque ce compteur atteint la valeur `0`.

Une description détaillée du fonctionnement de ces appels systèmes et fonctions de la librairie standard peut se trouver dans les livres de référence sur la programmation en C sous Unix **[Kerrisk2010]**, **[Mitchell+2001]**, **[StevensRago2008]**.

Utilisation des fichiers

Si quelques processus manipulent le système de fichiers et parcourent les répertoires, les processus qui utilisent des données sauvegardées dans des fichiers sont encore plus nombreux. Un système Unix offre deux possibilités d'écrire et de lire dans un fichier. La première utilise directement les appels systèmes **open(2)**, **read(2)**/ **write(2)** et **close(2)**. La seconde s'appuie sur les fonctions **fopen(3)**, **fread(3)**/ **fwrite(3)** et **fclose(3)** de la librairie **stdio(3)**. Seuls les appels systèmes sont traités dans ce cours. Des détails complémentaires sur les fonctions de la libraire peuvent être obtenus dans **[Kerrisk2010]**, **[Mitchell+2001]** ou **[StevensRago2008]**.

Du point de vue des appels systèmes de manipulation des fichiers, un fichier est une séquence d'octets. Avant qu'un processus ne puisse écrire ou lire dans un fichier, il doit d'abord demander au système d'exploitation l'autorisation d'accéder au fichier. Cela se fait en utilisant l'appel système **open(2)**.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char* pathname, int flags, mode_t mode);
```

Il existe deux variantes de l'appel système **open(2)**. La première permet d'ouvrir des fichiers existants. Elle prend deux arguments. La deuxième permet de créer un nouveau fichier et l'ouvre ensuite. Elle prend trois arguments. Le premier argument est le nom absolu ou relatif du fichier dont l'ouverture est demandée. Le deuxième argument est un entier qui contient un ensemble de drapeaux binaires qui précisent la façon dont le fichier doit être ouvert. Ces drapeaux sont divisés en

deux groupes. Le premier groupe est relatif à l'accès en lecture et/ou en écriture du fichier. Lors de l'ouverture d'un fichier avec **open(2)**, il est nécessaire de spécifier l'un des trois drapeaux d'accès suivants :

- `O_RDONLY` : indique que le fichier est ouvert uniquement en lecture. Aucune opération d'écriture ne sera effectuée sur le fichier.
- `O_WRONLY` : indique que le fichier est ouvert uniquement en écriture. Aucune opération de lecture ne sera effectuée sur le fichier.
- `O_RDWR` : indique que le fichier est ouvert pour des opérations de lecture et d'écriture.

En plus de l'un des trois drapeaux ci-dessus, il est également possible de spécifier un ou plusieurs drapeaux optionnels. Ces drapeaux sont décrits en détails dans la page de manuel **open(2)**. Les plus utiles sont probablement :

- `O_CREAT` : indique que si le fichier n'existe pas, il doit être créé lors de l'exécution de l'appel système **open(2)**. L'appel système **creat(2)** peut également être utilisé pour créer un nouveau fichier. Lorsque le drapeau `O_CREAT` est spécifié, l'appel système **open(2)** prend comme troisième argument les permissions du fichier qui doit être créé. Celles-ci sont spécifiées de la même façon que pour l'appel système **chmod(2)**. Si elles ne sont pas spécifiées, le fichier est ouvert avec comme permissions les permissions par défaut du processus définies par l'appel système **umask(2)**
- `O_APPEND` : indique que le fichier est ouvert de façon à ce que les données écrites dans le fichier par l'appel système **write(2)** s'ajoutent à la fin du fichier.
- `O_TRUNC` : indique que si le fichier existe déjà et qu'il est ouvert en écriture, alors le contenu du fichier doit être supprimé avant que le processus ne commence à y accéder.
- `O_SYNC` : ce drapeau indique que toutes les opérations d'écriture sur le fichier doivent être effectuées immédiatement sur le dispositif de stockage sans être mises en attente dans les buffers du noyau du système d'exploitation .. - `O_CLOEXEC` : ce drapeau qui est spécifique à Linux indique que le fichier doit être automatiquement fermé lors de l'exécution de l'appel système **execve(2)**. Normalement, les fichiers qui ont été ouverts par **open(2)** restent ouverts lors de l'exécution de **execve(2)**.

Ces différents drapeaux binaires doivent être combinés en utilisant une disjonction logique entre les différents drapeaux. Ainsi, `O_CREAT|O_RDWR` correspond à l'ouverture d'un fichier qui doit à la fois être créé si il n'existe pas et ouvert en lecture et écriture.

Lors de l'exécution de **open(2)**, le noyau du système d'exploitation vérifie si le processus qui exécute l'appel système dispose des permissions suffisantes pour accéder au fichier. Si oui, le système d'exploitation ouvre le fichier et retourne au processus appelant le **descripteur de fichier** correspondant. Si non, le processus récupère une valeur de retour négative et `errno` indique le type d'erreur.

Sous Unix, un **descripteur de fichier** est représenté sous la forme d'un entier positif. L'appel système **open(2)** retourne toujours le plus petit **descripteur de fichier** disponible. Par convention,

- `0` est le **descripteur de fichier** correspondant à l'entrée standard.
- `1` est le **descripteur de fichier** correspondant à la sortie standard.
- `2` est le **descripteur de fichier** correspondant à la sortie d'erreur standard.

Si l'appel système **open(2)** échoue, il retourne `-1` comme **descripteur de fichier** et `errno` donne plus de précisions sur le type d'erreur. Il peut s'agir d'une erreur liée aux droits d'accès au fichier (`EACESS`), une erreur de drapeau (`EINVAL`) ou d'une erreur d'entrée sortie lors de l'accès au dispositif de stockage (`EIO`). Le noyau du système d'exploitation maintient une table de l'ensemble des fichiers qui sont ouverts par tous les processus actifs. Si cette table est remplie, il n'est plus possible d'ouvrir de nouveau fichier et **open(2)** retourne une erreur. Il en va de même si le processus tente d'ouvrir plus de fichiers que le nombre maximum de fichiers ouverts qui est autorisé.

Note

Seul **open(2)** vérifie les permissions d'accès aux fichiers

Sous Unix, seul l'appel système **open(2)** vérifie qu'un processus dispose des permissions suffisantes pour accéder à un fichier qui est ouvert. Si les permissions ou le propriétaire d'un fichier change alors que ce fichier est ouvert par un processus, ce processus continue à pouvoir y accéder sans être affecté par la modification de droits. Il en va de même lorsqu'un fichier est effacé avec l'appel système **unlink(2)**. Si un processus utilisait le fichier qui est effacé, il continue à pouvoir l'utiliser même si le fichier n'apparaît plus dans le répertoire.

Toutes les opérations qui sont faites sur un fichier se font en utilisant le **descripteur de fichier** comme référence au fichier. Un **descripteur de fichier** est une ressource limitée dans un système d'exploitation tel que Unix et il est important qu'un processus n'ouvre pas inutilement un grand nombre de fichiers **[5]** et ferme correctement les fichiers ouverts lorsqu'il ne doit plus y accéder. Cela se fait en utilisant l'appel système **close(2)**. Celui-ci prend comme argument le **descripteur de fichier** qui doit être fermé.

```
#include <unistd.h>

int close(int fd);
```

Tout processus doit correctement fermer tous les fichiers qu'il a utilisé. Par défaut, le système d'exploitation ferme automatiquement les descripteurs de fichiers correspondant `0`, `1` et `2` lorsqu'un processus se termine. Les autres descripteurs de fichiers doivent être explicitement fermés par le processus. Si nécessaire, cela peut se faire en enregistrant une fonction permettant de fermer correctement les fichiers ouverts via **atexit(3)**. Il faut noter que par défaut un appel à **execve(2)** ne ferme pas les descripteurs de fichiers ouverts par le processus. C'est nécessaire pour permettre au programme exécuté d'avoir les entrées et sorties standard voulues.

Lorsqu'un fichier a été ouvert, le noyau du système d'exploitation maintient un **offset pointer**. Cet **offset pointer** est la position actuelle de la tête de lecture/écriture du fichier. Lorsqu'un fichier est ouvert, son **offset pointer** est positionné au premier octet du fichier, sauf si le drapeau `O_APPEND` a été spécifié lors de l'ouverture du fichier, dans ce cas l'**offset pointer** est positionné juste après le dernier octet du fichier de façon à ce qu'une écriture s'ajoute à la suite du fichier.

Les deux appels systèmes permettant de lire et d'écrire dans un fichier sont respectivement **read(2)** et **write(2)**.

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Ces deux appels systèmes prennent trois arguments. Le premier est le *descripteur du fichier* sur lequel l'opération doit être effectuée. Le second est un pointeur `void *` vers la zone mémoire à lire ou écrire et le dernier est la quantité de données à lire/écrire. Si l'appel système réussit, il retourne le nombre d'octets qui ont été écrits/lus et sinon une valeur négative et la variable `errno` donne plus de précisions sur le type d'erreur. **read(2)** retourne `0` lorsque la fin du fichier a été atteinte.

Il est important de noter que **read(2)** et **write(2)** permettent de lire et d'écrire des séquences contiguës d'octets. Lorsque l'on écrit ou lit des chaînes de caractères dans lesquels chaque caractère est représenté sous la forme d'un byte, il est possible d'utiliser **read(2)** et **write(2)** pour lire et écrire d'autres types de données que des octets comme le montre l'exemple ci-dessous.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

void exit_on_error(char *s) {
  perror(s);
  exit(EXIT_FAILURE);
}

int main (int argc, char *argv[])  {
  int n=1252;
  int n2;
  short ns=1252;
  short ns2;
  long nl=125212521252;
  long nl2;
  float f=12.52;
  float f2;
  char *s="sinf1252";
  char *s2=(char *) malloc(strlen(s)*sizeof(char )+1);
  int err;
  int fd;

  fd=open("test.dat",O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  if(fd==-1) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  if( write(fd, (void *) s, strlen(s)) == -1 )
    exit_on_error("write s");
  if (write(fd, (void *) &n, sizeof(int )) == -1)
    exit_on_error("write n");
  if (write(fd, (void *) &ns, sizeof(short int))==-1)
    exit_on_error("write ns");
  if (write(fd, (void *) &nl, sizeof(long int))==-1)
    exit_on_error("write nl");
  if (write(fd, (void *) &f, sizeof(float))==-1)
    exit_on_error("write f");
  if (close(fd)==-1)
    exit_on_error("close ");

  // lecture
  fd=open("test.dat",O_RDONLY);
  if(fd==-1) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  printf("Fichier ouvert\n");

  if(read(fd, (void *) s2, strlen(s))==-1)
    exit_on_error("read s");
  printf("Donnée écrite : %s, lue: %s\n",s,s2);

  if(read(fd, (void *) &n2, sizeof(int))==-1)
    exit_on_error("read n");
  printf("Donnée écrite : %d, lue: %d\n",n,n2);

  if(read(fd, (void *) &ns2, sizeof(short))==-1)
    exit_on_error("read ns");
  printf("Donnée écrite : %d, lue: %d\n",ns,ns2);
```

```
  if(read(fd, (void *) &nl2, sizeof(long))==-1)
    exit_on_error("read nl");
  printf("Donnée écrite : %ld, lue: %ld\n",nl,nl2);

  if(read(fd, (void *) &f2, sizeof(float))==-1)
    exit_on_error("read f");
  printf("Donnée écrite : %f, lue: %f\n",f,f2);
  err=close(fd);
  if(err==-1){
   perror("close");
    exit(EXIT_FAILURE);
  }

  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche la sortie ci-dessous.

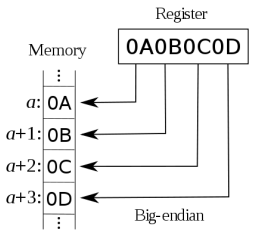
```
Fichier ouvert
Donnée écrite : sinf1252, lue: sinf1252
Donnée écrite : 1252, lue: 1252
Donnée écrite : 1252, lue: 1252
Donnée écrite : 125212521252, lue: 125212521252
Donnée écrite : 12.520000, lue: 12.520000
```

Si il est bien possible de sauvegarder dans un fichier des entiers, des nombres en virgule flottante voire même des structures, il faut être bien conscient que l'appel système **write(2)** se contente de sauvegarder sur le disque le contenu de la zone mémoire pointée par le pointeur qu'il a reçu comme second argument. Si comme dans l'exemple précédent c'est le même processus qui lit les données qu'il a écrit, il pourra toujours récupérer les données correctement.

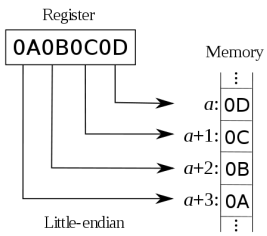
Par contre, lorsqu'un fichier est écrit sur un ordinateur, envoyé via Internet et lu sur un autre ordinateur, il peut se produire plusieurs problèmes dont il faut être conscient. Le premier problème est que deux ordinateurs différents n'utilisent pas nécessairement le même nombre d'octets pour représenter chaque type de données. Ainsi, sur un ordinateur équipé d'un ancien processeur **[IA32]**, les entiers sont représentés sur 32 bits (i.e. 4 bytes) alors que sur les processeurs plus récents ils sont souvent représentés sur 64 bits (i.e. 8 bytes). Cela implique qu'un tableau de 100 entiers en 32 bits sera interprété comme un tableau de 50 entiers en 64 bits.

Le second problème est que les fabricants de processeurs ne se sont pas mis d'accord sur la façon dont il fallait représenter les entiers sur 16 et 32 bits en mémoire. Il y a deux techniques qui sont utilisées : **big endian** et **little endian**.

Pour comprendre ces deux techniques, regardons comment l'entier 16 bits `0b1111111100000000` est stocké en mémoire. En **big endian**, le byte `11111111` sera stocké à l'adresse *x* et le byte `00000000` à l'adresse *x+1*. En **little endian**, c'est le byte `00000000` qui est stocké à l'adresse *x* et le byte `11111111` qui est stocké à l'adresse *x+1*. Il en va de même pour les entiers encodés sur 32 bits comme illustré dans les deux figures ci-dessous **[6]**.



Ecriture d'un entier 32 bits en mémoire en *big endian*



Ecriture d'un entier 32 bits en mémoire en *little endian*

Pour les nombres en virgule flottante, ce problème ne se pose heureusement pas car tous les processeurs actuels utilisent la même norme pour représenter les nombres en virgule flottant en mémoire.

Les processeurs **[IA32]** utilisent la représentation **little endian** tandis que les PowerPC utilisent **big endian**. Certains processeurs sont capables d'utiliser les deux représentations.

Il est également possible en utilisant l'appel système **lseek(2)** de déplacer l'**offset pointer** associé à un **descripteur de fichier**.

```
#include <sys/types.h>
#include <unistd.h>
                                    2è argument (ds la description ci-dessous)
off_t lseek(int fd, off_t offset, int whence);
```

Cet appel système prend trois arguments. Le premier est le **descripteur de fichier** dont l'**offset pointer** doit être modifié. Le second est un entier qui est utilisé pour le calcul de la nouvelle position **offset pointer** et le troisième indique comment l'**offset pointer** doit être calculé. Il y a trois modes de calcul possibles pour l'**offset pointer** :

- `whence==SEEK_SET` : dans ce cas, le deuxième argument de l'appel système indique la valeur exacte du nouvel **offset pointer**
- `whence==SEEK_CUR` : dans ce cas, le nouvel **offset pointer** sera sa position actuelle à laquelle le deuxième argument aura été ajouté
- `whence==SEEK_END` : dans ce cas, le nouvel **offset pointer** sera la fin du fichier à laquelle le deuxième argument aura été ajouté

Note

Fichiers temporaires

Il est parfois nécessaire dans un programme de créer des fichiers temporaires qui sont utilisés pour effectuer des opérations dans le processus sans pour autant être visible dans d'autres processus et sur le système de fichiers. Il est possible d'utiliser **open(2)** pour créer un tel fichier temporaire, mais il faut dans ce cas prévoir tous les cas d'erreur qui peuvent se produire lorsque par exemple plusieurs instances du même programme s'exécutent au même moment. Une meilleure solution est d'utiliser la fonction de la librairie **mkstemp(3)**. Cette fonction prend comme argument un modèle de nom de fichier qui se termine par `xxxxxx` et génère un nom de fichier unique et retourne un descripteur de fichier associé à ce fichier. Elle s'utilise généralement comme suit :

```
char template[]="/tmp/sinf1252PROCXXXXXX";

int fd=mkstemp(template);
if(fd==-1)
  exit_on_error("mkstemp");
// template contient le nom exact du fichier généré
unlink(template);
// le fichier est effacé, mais reste accessible
// via son descripteur jusqu'à close(fd)

// Accès au fichier avec read et write

if(close(fd)==-1)
  exit_on_error("close");
// le fichier n'est plus accessible
```

L'utilisation de **unlink(2)** permet de supprimer le fichier du système de fichiers dès qu'il a été créé. Ce fichier reste cependant accessible au processus tant que celui-ci dispose d'un descripteur de fichier qui y est associé.

Note

Duplication de descripteurs de fichiers

Dans certains cas il est utile de pouvoir dupliquer un descripteur de fichier. C'est possible avec les appels systèmes **dup(2)** et **dup2(2)**. L'appel système **dup(2)** prend comme argument un descripteur de fichier et retourne le plus petit descripteur de fichier libre. Lorsqu'un descripteur de fichier a été dupliqué avec **dup(2)** les deux descripteurs de fichiers partagent le même **offset pointer** et les mêmes modes d'accès au fichier.

Fichiers mappés en mémoire

Lorsqu'un processus Unix veut lire ou écrire des données dans un fichier, il utilise en général les appels systèmes **open(2)**, **read(2)**, **write(2)** et **close(2)** directement ou à travers une librairie de plus haut niveau comme la libraire d'entrées/sorties standard. Ce n'est pas la seule façon pour accéder à des données sur un dispositif de stockage. Grâce à la mémoire virtuelle, il est possible de placer le contenu d'un fichier ou d'une partie de fichier dans une zone de la mémoire du processus. Cette opération peut être effectuée en utilisant l'appel système **mmap(2)**. Cet appel système permet de rendre un fichier accessibles directement dans la mémoire du processus.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

L'appel système **mmap(2)** prend six arguments, c'est un des appels systèmes qui utilise le plus d'arguments. Il permet de rendre accessible une portion d'un fichier via la mémoire d'un processus. Le cinquième argument est le descripteur du fichier qui doit être mappé. Celui-ci doit avoir été préalablement ouvert avec l'appel système **open(2)**. Le sixième argument spécifie l'offset à partir duquel le fichier doit être mappé, `0` correspondant au début du fichier. Le premier argument est l'adresse à laquelle la première page du fichier doit être mappée. Généralement, cet argument est mis à `NULL` de façon à laisser le noyau choisir l'adresse la plus

appropriée. Le deuxième argument est la longueur de la zone du fichier qui doit être mappée en mémoire. Le troisième argument contient des drapeaux qui spécifient les permissions d'accès aux données mappées. Cet argument peut soit être `PROT_NONE`, ce qui indique que la page est inaccessible soit une permission classique :

- `PROT_EXEC`, les pages mappées contiennent des instructions qui peuvent être exécutées
- `PROT_READ`, les pages mappées contiennent des données qui peuvent être lues
- `PROT_WRITE`, les pages mappées contiennent des données qui peuvent être modifiées

Ces drapeaux peuvent être combinés avec une disjonction logique. Le quatrième argument est un drapeau qui indique comment les pages doivent être mappées en mémoire. Ce drapeau spécifie comment un fichier qui est mappé par deux ou plusieurs processus doit être traité. Deux drapeaux sont possibles :

- `MAP_PRIVATE`. Dans ce cas, le fichier est mappé dans chaque processus, mais si un processus modifie une page, cette modification n'est pas répercutée aux autres processus qui ont mappé le même fichier.
- `MAP_SHARED`. Dans ce cas, plusieurs processus peuvent accéder et modifier la page qui est mappée en mémoire. Lorsqu'un processus modifie le contenu d'une page, la modification est visible aux autres processus. Par contre, le fichier qui est mappé en mémoire n'est modifié que lorsque le noyau du système d'exploitation décide d'écrire les données modifiées sur le dispositif de stockage. Ces écritures dépendent de nombreux facteurs, dont la charge du système. Si un processus veut être sûr des écritures sur disque des modifications qu'il a fait à un fichier mappé un mémoire, il doit exécuter l'appel système **msync(2)** ou supprimer le mapping via **munmap(2)**.

Ces deux drapeaux peuvent dans certains cas particuliers être combinés avec d'autres drapeaux définis dans la page de manuel de **mmap(2)**.

Lorsque **mmap(2)** réussit, il retourne l'adresse du début de la zone mappée en mémoire. En cas d'erreur, la constante `MAP_FAILED` est retournée et `errno` est mis à jour en conséquence.

L'appel système **msync(2)** permet de forcer l'écriture sur disque d'une zone mappée en mémoire. Le premier argument est l'adresse du début de la zone qui doit être écrite sur disque. Le deuxième argument est la longueur de la zone qui doit être écrite sur le disque. Enfin, le dernier contient un drapeau qui spécifie comment les pages correspondantes doivent être écrites sur le disque. Le drapeau `MS_SYNC` indique que l'appel **msync(2)** doit bloquer tant que les données n'ont pas été écrites. Le drapeau `MS_ASYNC` indique au noyau que l'écriture doit être démarrée, mais l'appel système peut se terminer avant que toutes les pages modifiées aient été écrites sur disque.

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

Lorsqu'un processus a fini d'utiliser un fichier mappé en mémoire, il doit d'abord supprimer le mapping en utilisant l'appel système **munmap(2)**. Cet appel système prend deux arguments. Le premier doit être un multiple de la taille d'une page [7]. Le second est la taille de la zone pour laquelle le mapping doit être retiré.

```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

A titre d'exemple d'utilisation de **mmap(2)** et **munmap(2)**, le programme ci-dessous implémente l'équivalent de la commande **cp(1)**. Il prend comme arguments deux noms de fichiers et copie le contenu du premier dans le second. La copie se fait en mappant le premier fichier entièrement en mémoire et en utilisant la fonction **memcpy(3)** pour réaliser la copie. Cette solution fonctionne avec de petits fichiers. Avec de gros fichiers, elle n'est pas très efficace car tout le fichier doit être mappé en mémoire.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main (int argc, char *argv[]) {
 int file1, file2;
 void *src, *dst;
 struct stat file_stat;
 char dummy=0;

 if (argc != 3) {
   fprintf(stderr,"Usage : cp2 source dest\n");
   exit(EXIT_FAILURE);
 }
 // ouverture fichier source
 if ((file1 = open (argv[1], O_RDONLY)) < 0) {
   perror("open(source)");
   exit(EXIT_FAILURE);
 }

 if (fstat (file1,&file_stat) < 0) {
   perror("fstat");
   exit(EXIT_FAILURE);
 }
 // ouverture fichier destination
```

```
  if ((file2 = open (argv[2], O_RDWR | O_CREAT | O_TRUNC, file_stat.st_mode)) < 0) {
    perror("open(dest)");
    exit(EXIT_FAILURE);
  }

  // le fichier destination doit avoir la même taille que le source
  if (lseek (file2, file_stat.st_size - 1, SEEK_SET) == -1) {
    perror("lseek");
    exit(EXIT_FAILURE);
  }

  // écriture en fin de fichier
  if (write (file2, &dummy, sizeof(char)) != 1) {
    perror("write");
    exit(EXIT_FAILURE);
  }

  // mmap fichier source
  if ((src = mmap (NULL, file_stat.st_size, PROT_READ, MAP_SHARED, file1, 0))  == (void *)(-1)) {
    perror("mmap(src)");
    exit(EXIT_FAILURE);
  }

  // mmap fichier destination
  if ((dst = mmap (NULL, file_stat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, file2, 0)) == (void *)(-1)) {
    perror("mmap(src)");
    exit(EXIT_FAILURE);
  }

  // copie complète
  memcpy (dst, src, file_stat.st_size);

  // libération mémoire
  if(munmap(src,file_stat.st_size)<0) {
    perror("munmap(src)");
    exit(EXIT_FAILURE);
  }

  if(munmap(dst,file_stat.st_size)<0) {
    perror("munmap(dst)");
    exit(EXIT_FAILURE);
  }
  // fermeture fichiers
  if(close(file1)<0) {
    perror("close(file1)");
    exit(EXIT_FAILURE);
  }

  if(close(file2)<0) {
    perror("close(file2)");
    exit(EXIT_FAILURE);
  }
  return(EXIT_SUCCESS);
}
```

Footnotes

[1] Cette structure est partiellement inspirée du format des inodes du système de fichiers Minix, voir **linux/minix_fs.h**.

[2] Le champ `zone[10]` permet de stocker dans l'**inode** les références vers les premiers secteurs du fichier et des références vers d'autres blocs qui contiennent eux-aussi des références vers des blocs. Cela permet de stocker une liste de secteurs qui est de taille variable à partir d'un **inode** qui a lui une taille fixe. Une description détaillée des inodes peut se trouver dans une référence sur les systèmes d'exploitation telle que **[Tanenbaum+2009]**.

[3] Source : **linux/ext2_fs.h**

[4] Dans un système de fichiers Unix, un lien ne peut être créé avec **ln(1)** ou **link(2)** que lorsque les deux répertoires concernés sont situés sur le même système de fichiers. Si ce n'est pas le cas, il faut utiliser un **lien symbolique**. Ceux-ci peuvent être créés en utilisant l'appel système **symlink(2)** ou via la commande **ln(1)** avec l'argument `-s`.

[5] Il y a une limite maximale au nombre de fichiers qui peuvent être ouverts par un processus. Cette limite peut être récupérée avec l'appel système **getdtablesize(2)**.

[6] Source : **http://en.wikipedia.org/wiki/Endianness**

[7] Il est possible d'obtenir la taille des pages utilisée sur un système via les appels **sysconf(3)** ou **getpagesize(2)**