# Module Architecture et Système 2 L3/S6 – TD/TP 5
# Répertoire, permissions, i–nœuds, parcours

## 1 Permissions associées aux répertoires (TD)

Sous UNIX, l'enregistrement des fichiers est hiérarchisé grâce à la notion de répertoires. Mais, en réalité, les répertoires sont eux-mêmes des fichiers, dont le contenu définit l'ensemble des fichiers et sous-répertoires qui leurs sont affectés.

La signification des indicateurs de permissions associées aux fichiers répertoires est la suivante :

- **r** Permission de lecture du fichier répertoire ; permet le listage des fichiers affectés au répertoire.
- **w** Permission d'écriture du fichier répertoire ; permet de créer ou de détruire des fichiers dans le répertoire.
- **x** permission d'accès ; permet d'accéder aux fichiers affectés au répertoire, et/ou permet de définir le répertoire comme répertoire courant.

**Questions** Soit l'arborescence de fichiers suivantes :

```
. ______REP1 (dr-x) _____ REP2 (d-wx)
                        |
                        |__ REP3 (drw-) __ REP5 (drwx)
                        |
                        \__ REP4 (dr-x) __ toto (-rw-)
```

Les commandes suivantes sont-elles possibles à partir du répertoire courant `.` ?

1. `ls REP1`
2. `cd REP1`
3. `ls REP1/REP2`
4. `cd REP1/REP2`
5. `touch REP1/REP2/toto`
6. `ls REP1/REP3`
7. `touch REP1/toto`
8. `cat >> REP1/REP4/toto`
9. `rm REP1/REP4/toto`
10. `ls REP1/REP3/REP5`

# 2 État d'un fichier (TD)

L'appel système `stat` permet d'extraire certaines informations de l'i-nœud d'un fichier.

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

int stat(const char *file_name, struct stat *buf);
```

L'argument `file_name` est le nom du fichier concerné. La fonction "remplit" une structure `struct stat` dont l'adresse est fournie en argument, retourne 0 en cas de succès. Une structure de type `struct stat` est de la forme :

```
struct stat {
            dev_t         st_dev;      /* Peripherique                */
            ino_t         st_ino;      /* Numero i-noeud              */
            mode_t        st_mode;     /* Protection                  */
            nlink_t       st_nlink;    /* Nb liens materiels          */
            uid_t         st_uid;      /* UID proprietaire            */
            gid_t         st_gid;      /* GID proprietaire            */
            dev_t         st_rdev;     /* Type peripherique           */
            off_t         st_size;     /* Taille totale en octets     */
            blksize_t     st_blksize;  /* Taille de bloc pour E/S     */
            blkcnt_t      st_blocks;   /* Nombre de blocs alloues     */
            time_t        st_atime;    /* Heure dernier acces         */
            time_t        st_mtime;    /* Heure derniere modification */
            time_t        st_ctime;    /* Heure dernier changement    */
};
```

Afin de facilement exploiter l'information présente dans le champ `st_mode` de type `mode_t`, des macros sont définies :

```
#define S_IFMT    0xF000    /* masque du type de fichier */
#define S_IAMB    0x01FF    /* masque des droits d'acces */

#define S_IFSOCK  0xC000    /* socket                    */
#define S_IFLNK   0xA000    /* lien symbolique           */
#define S_IFREG   0x8000    /* fichier regulier          */
#define S_IFBLK   0x6000    /* peripherique blocs        */
#define S_IFDIR   0x4000    /* repertoire                */
#define S_IFCHR   0x2000    /* peripherique caracteres   */
#define S_IFIFO   0x1000    /* fifo                      */

#define S_ISUID   0004000   /* bit Set-UID   (0x800)     */
#define S_ISGID   0002000   /* bit Set-Gid   (0x400)     */
#define S_ISVTX   0001000   /* bit "sticky"  (0x200)     */
```

```
#define S_IRWXU   00700     /* lecture/ecriture/execution du proprietaire */
#define S_IRUSR   00400     /* le proprietaire a le droit de lecture      */
#define S_IWUSR   00200     /* le proprietaire a le droit d'ecriture      */
#define S_IXUSR   00100     /* le proprietaire a le droit d'execution     */
#define S_IRWXG   00070     /* lecture/ecriture/execution du groupe       */
#define S_IRGRP   00040     /* le groupe a le droit de lecture            */
#define S_IWGRP   00020     /* le groupe a le droit d'ecriture            */
#define S_IXGRP   00010     /* le groupe a le droit d'execution           */
#define S_IRWXO   00007     /* lecture/ecriture/execution des autres      */
#define S_IROTH   00004     /* les autres ont le droit de lecture         */
#define S_IWOTH   00002     /* les autres ont le droit d'ecriture         */
#define S_IXOTH   00001     /* les autres ont le droit d'execution        */

#define S_ISSOCK(mode) (((mode) & S_IFMT) == S_IFSOCK)
#define S_ISLNK(mode)  (((mode) & S_IFMT) == S_IFLNK)
#define S_ISREG(mode)  (((mode) & S_IFMT) == S_IFREG)
#define S_ISBLK(mode)  (((mode) & S_IFMT) == S_IFBLK)
#define S_ISDIR(mode)  (((mode) & S_IFMT) == S_IFDIR)
#define S_ISCHR(mode)  (((mode) & S_IFMT) == S_IFCHR)
#define S_ISFIFO(mode) (((mode) & S_IFMT) == S_IFIFO)
```

**Questions**

1. Donner la valeur en octal, puis en hexadecimal, du champ `st_mode` d'un fichier répertoire accessible en lecture et en écriture pour tout le monde, et qui peut être défini comme répertoire courant uniquement par le propriétaire.

2. Écrire un extrait de code en C permettant de vérifier que le fichier `/etc/shadow` est accessible uniquement en lecture et écriture pour l'utilisateur `root`.

# 3 Lecture d'un répertoire (TD)

Les répertoires sont des fichiers qui définissent la hiérarchie logique d'un système de fichiers ; ils contiennent une liste de noms de fichiers (ou de répertoires) et leur i-nœud associé.

En tant que fichier, il est possible sous UNIX de lire leur contenu par l'intermédiaire des fonctions habituelles (`open`, `read`, ...). La structuration des enregistrements correspond à la structure C `struct dirent` définie dans le fichier `dirent.h`.

Cependant, afin d'améliorer la lisibilité et la portabilité des programmes, il est conseillé d'utiliser un ensemble de fonctions dédiées à la lecture des répertoires.

### Ouverture d'un fichier répertoire

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (const char *dirname);
```

`dirname` est le nom du fichier répertoire à ouvrir. La paramètre de retour est un pointeur sur une structure `DIR` qui est un "descripteur de répertoire [1]".

### Fermeture d'un fichier répertoire

```
#include <sys/types.h>
#include <dirent.h>

int closedir (DIR *dir);
```

`dirp` est un pointeur de descripteur de répertoire. La fonction retourne 0 en cas de succès, et -1 en cas d'erreur.

### Lecture d'une entrée du répertoire [2]

```
#include <sys/types.h>
#include <dirent.h>

struct dirent * readdir (DIR * dir);
```

`dir` est un pointeur de descripteur de répertoire. La fonction retourne un pointeur sur une structure C `struct dirent`, qui contient, entre autre, le nom d'un fichier appartenant au répertoire, et son i-nœud.

Des appels successifs à `readdir` permettent d'obtenir les noms et les i-nœuds de tous les fichiers du répertoire. Lorsque `readdir` retourne `NULL`, c'est fini !

Quelques informations sur la structure C `struct dirent` :

```
struct dirent {
               long d_ino;               /* inode number               */
               off_t d_off;              /* offset to this dirent      */
               unsigned short d_reclen;  /* length of this d_name      */
               char d_name[NAME_MAX+1];  /* file name (null-terminated) */
};
```

1. en pratique `DIR` est une structure qui contient un descripteur de fichier, et des informations sur le flot de lecture du répertoire.

2. la fonction `readdir` présentée ici n'est pas POSIX, et n'est pas sûr pour un programme multi-activités (utiliser `readdir_r` à la place).

**Questions**

1. Écrire un programme qui vérifie si un bit "set user id" est positionné sur l'un des fichiers exécutables du répertoire courant.
2. Modifier le programme précédant pour effectuer la même vérification sur tous les fichiers d'une hiérarchie de répertoires. Il faut utiliser pour cela :

```
#include <unistd.h>

int chdir(const char *path); /* retourne 0 si ok, -1 si ko */
```

# 4 mémoire virtuelle

La fonction `mmap` établit une correspondance entre une zone de l'espace d'adressage d'un processus et un fichier. Les accès au fichier sont alors gérés par le système de gestion de la mémoire virtuelle. Les lectures et les écritures dans le fichier ne sont plus effectuées par l'intermédiaire des appels systèmes `read` et `write`, mais simplement par des accès mémoire dans la zone mémoire correspondant au fichier.

1. Eecrire un programme permettant de créer un fichier de 10000 octets (10000 'A' par exemple).
2. Utiliser la fonction `mmap` pour accéder à ce fichier dans un autre programme C.
3. Puis, protéger en écriture la première page mémoire qui correspond au fichier. Vérifier l'efficacité de la protection.
   Voir l'utilisation de la fonction `mprotect`.
   Signalons que, la taille d'une page mémoire s'obtient de la façon suivante :

```
long pagesize;

pagesize=sysconf(_SC_PAGESIZE);
```

4. Utiliser maintenant `mmap` pour créer une zone de mémoire partagée entre 2 processus.

# 5 Recherche récursive

Nous cherchons a réaliser un programme qui retrouve un fichier dans une arborescence (i.e. la première occurence du nom).

Réaliser pour cela une fonction récursive `lookfor`, ayant le prototype suivant :

```
int lookfor(char *fileName, char *path, size_t size);
```

Cette fonction recherche le fichier `fileName` à partir du répertoire courant.
L'argument `size` indique la taille qui a été allouée pour la chaîne `path`. En effet, cette fonction lookfor doit remplir `path` de telle sorte qu'il désigne le chemin menant à `fileName`.

Cette fonction retourne trois valeurs possibles :
- `0` → `fileName` a été trouvé, `path` le désigne ;
- `-1` → `fileName` n'a pas été trouvé, `path` est indéterminé ;
- `1` → la capacité `size` est insuffisante, `path` est indéterminé.

Dans un premier temps nous supposerons que la taille de la zone mémoire allouée pour `path` est suffisante et nous ignorerons l'argument `size`. En fin de scéance, s'il reste du temps, les contrôles nécessaires à une gestion rigoureuse de la mémoire pourront être ajoutés.

**Remarque** :

```
#include <unistd.h>
char * getcwd (char *buf, size_t size);
```

La fonction `getcwd` copie le chemin d'accès absolu du répertoire de travail courant dans la zone pointée par `buf` qui est de longueur `size` (le caractère de fin de chîne '\0' est positionné. Si le chemin du répertoire courant nécessite une zone plus grande que `size` octets, la fonction retourne `NULL`.