TD de Système n° 9

Exercice 1 : Arborescence fantôme

Le but de cet exercice est de copier un répertoire et récursivement tout ce qu'il contient, mais au lieu de créer des copies des fichiers sources, on créera des liens symboliques vers eux. On obtient ce qu'on appelle une «arborescence fantôme» puisqu'elle ne contient que son «squelette», pas sa «chair» (le contenu des fichiers). Cette technique est utilisée pour ne pas polluer l'arborescence source tout en ayant accès à ses fichiers, par exemple quand on compile un gros projet.

1. Ecrire une fonction récursive

   ```
   void parcours_dir(char *path);
   ```

   qui parcourt le répertoire `path` a□che le nom de son contenu et parcourt récursivement ses sous-répertoires. Vous aurez besoin des appels système

   ```
   DIR *opendir(const char *name);
   struct dirent *readdir(DIR *dirp);
   int stat(const char *path, struct stat *buf);
   int closedir(DIR *dirp);
   ```

2. Modifier la fonction précédente pour qu'elle crée une arborescence fantôme. Pour chaque membre du répertoire courant :
   – Si c'est un répertoire, elle créera un sous-répertoire dans l'arborescence fantôme,
   – Si c'est un fichier, elle créera un lien symbolique vers ce fichier.
   Vous aurez besoin de :

   ```
   int mkdir(const char *pathname, mode_t mode);
   int symlink(const char *oldpath, const char *newpath);
   ```

Exercice 2 : « du »

La commande `du` utilisée avec un nom de fichier (au sens large) en argument a□che la taille en kilo octets occupée par l'arborescence A située sous ce fichier (sans suivre les liens symboliques) ainsi que les tailles des sous-arborescences des répertoires apparaissant dans A. Avec l'option `-L` on va suivre les liens symboliques.

Programmer cette commande avec l'option -L.

```
int lstat(const char *path, struct stat *buf);
struct stat {
    ...
    mode_t   st_mode;   /* protection */
    ...
    off_t    st_size;   /* total size, in bytes */
    ...
};
S_ISLNK(st_mode)
int readlink(const char *restrict path, char *restrict buf, size_t bufsize);
```

Exercice 3 : Deadlock

Voici la description de la fonction `fcntl` : `int fcntl(in fd, int op, struct flock *verrou);` où la structure `struct flock` est définie comme suit :

```
struct flock
{
    short int l_type; /* F_RDLCK, F_WRLCK ou F_UNLCK */
    short int l_whence; /* SEEK_SET, SEEK_CUR ou SEEK_END */
    off_t l_start;    /* position relative \‘a l_whence */
    off_t l_len;      /* longueur de l’intervalle */
    pid_t l_pid;      /* pid du processus auquel appartient le verrou */
};
```

et `op` est parmi :

- `F_SETLK` : Demande de pose non bloquante de verrou.
- `F_SETLKW` : Demande de pose bloquante de verrou. Si il existe déjà un verrou bloquant, le processus est endormi jusqu'à ce qu'il n'y ait plus de verrou incompatible ou que l'appel soit interrompu.
- `F_GETLK` : Test d'existence d'un verrou incompatible avec le verrou donné. S'il n'existe pas, le champ `l_type` de `verrou` vaut `F_UNLCK` sinon, `verrou` est rempli avec les informations sur ce verrou incompatible.

Le type du verrou décrit les possibilités de cohabitation des di□érents verrous :

- `F_RDLCK` (verrou partagé) : plusieurs verrous de ce type peuvent cohabiter, c'est-à-dire avoir des portées non disjointes, par exemple les verrous [80,150] et [100,123] ;
- `F_WRLCK` (verrou exclusif) : pas de cohabitation possible avec un autre verrou quel que soit son type.

L'avantage d'utiliser des verrous est de pouvoir protéger une ressource, mais ce n'est pas sans risque.

1. Écrire un programme `verrou` qui va prendre un verrou sur un fichier1, attendre durant un temps aléatoire, puis poser un verrou sur un fichier2, attendre durant un temps aléatoire, rendre les verrous et recommencer un peu plus tard et ainsi de suite. Les noms des fichiers sont passés en argument de la commande. Vous aurez besoin des fonctions :

   ```
   unsigned int sleep (unsigned int nb_sec);
   int rand (void);
   ```

2. Quel problème risque de subvenir si on lance la commande « `verrour fichier1 fichier2 & verrou fichier2 fichier1` » ?