Langage C et Programmation Système
TP n° 7 : Manipulations de fichiers et de répertoires

Exercice 1 : Manipulations basiques

Dans cet exercice, on va se familiariser avec quelques fonctions pour manipuler le système de gestion de fichiers.

1. Écrivez un programme `mypwd` qui affiche le chemin absolu du répertoire de travail. On pourra supposer que la taille d'un chemin ne dépasse 1024 caractères. (*Indice :* `man 3 getcwd`)
2. Écrivez un programme `mypwd_above` qui attend un entier `k` en argument et qui affiche une chaîne de caractères contenant le chemin absolu du répertoire situé `k` niveaux au-dessus. Si on arrive à la racine avant, on affichera la racine `/`. (*Indice :* `man 2 chdir`)
3. Écrivez un programme `basewd` qui affiche le nom du répertoire de travail (sans le chemin d'accès). Pour cela, il suffit de récupérer l'i-noeud du répertoire courant et d'aller voir dans le répertoire parent quel est le nom du répertoire correspondant à cet i-noeud. Attention cette technique ne marche pas forcément si le répertoire est un point de montage, par exemple d'un disque ou d'un dossier NFS. (*Indice :* `stat(2)`, `opendir(3)`, `readdir(3)` et `closedir(3)`)
4. Écrivez un programme `myls` qui affiche le nom des fichiers du répertoire de travail avec un nom par ligne. (*Indice :* `opendir(3)`, `readdir(3)` et `closedir(3)`)
5. Modifiez le programme précédent en ajoutant la possibilité de mettre une option `-l` afin que, pour chaque nom, on précise si il s'agit d'un répertoire (en écrivant le préfixe `d`) ou d'un fichier (avec le préfixe `-`) et on affiche également les droits. Par exemple, une ligne devra ressembler à : `drwxr-xr-x TP7/`, pour un répertoire, et `-rw-r--r-- tp7.pdf`, pour un fichier. (*Indice :* `stat(2)`)
6. *Optionnel :* Modifiez le programme précédent pour qu'il puisse accepter comme autre option `-R` pour faire une recherche récursive dans tous les sous-répertoires. Vous pouvez vous inspirer du résultat de la commande `ls -R` pour cette question.

Exercice 2 : Représentation du contenu d'un répertoire sous forme d'arbre

Dans cet exercice, on va construire un arbre représentant la structure de fichiers du répertoire courant. Dans un fichier `treedir.h` on mettra les types utilisés par notre structure. Le type `node` représente un noeud de l'arbre avec trois champs : `name` est un pointeur sur le nom du noeud, `nb_children` est le nombre de fils du noeud dans l'arbre, et `children` est un pointeur sur une liste chaînée de `node` qui correspond à la liste des fils.

```
typedef struct _node node;
typedef struct _list_node list_node;
struct _node {
  char *name;
  int nb_children;
  list_node *children;
```

```
};
struct _list_node {
  node n;
  list_node *next;
};
```

1. Écrivez une fonction `node *new_node(const char *name)` qui prend en argument une chaîne de caractères et crée un noeud en mettant à `NULL` la liste des fils et à `0` le nombre de fils. Il faudra utiliser `malloc` pour allouer la mémoire pour le noeud et une copie de la chaîne de caractères. La signature de cette fonction sera à mettre dans `treedir.h`, et le code correspondant dans `treedir.c`. De manière analogue, écrivez une fonction `void delete_node(node *nd)` qui libère (toute) la mémoire allouée par `new_node` lors de la création du noeud `*nd`.
2. Écrivez une fonction `int new_child(node *current, const char *name)` qui rajoute un noeud fils dans la liste des noeuds fils de `*current`. Le noeud ajouté aura une liste de noeuds égale à `NULL` et son nombre de fils sera égal à `0`. Cette fonction renverra `0` si tout se passe bien et `-1` en cas de problème. Comme précédemment, la signature et le code sont à mettre dans `treedir.h` et `treedir.c`, respectivement.
3. Écrivez une fonction `int print_tree(node *current)` qui affiche à l'écran l'arbre avec `*current` comme noeud racine, en affichant une ligne *par noeud*. Cette ligne contiendra les chaînes de caractères des noeuds traversés depuis le noeud `*current` et séparées par le caractère `:`. Par exemple, pour l'arbre avec trois noeuds étiquetés `un`, `deux` et `trois`, tel que les deux derniers noeuds sont fils du premier, cette fonction affichera :
   ```
   un
   un:deux
   un:trois
   ```
   Comme avant, la fonction sera à mettre dans `treedir.h` et `treedir.c`.
4. Écrivez une fonction `node *build_from_current_file()` qui construit l'arbre correspondant au répertoire courant en mettant dans la liste des fils les noms des fichiers et des sous-répertoires, et dans chaque noeud correspondant à un répertoire la liste de ces sous-répertoires, etc. Pour le noeud racine de l'arbre, on mettra le nom du répertoire courant (on peut se servir d'une des solutions du premier exercice pour cela). Cette fonction sera à mettre dans `treedir.h` et `treedir.c`.
5. Écrivez une fonction récursive `void delete_tree(node *current)` qui libère toute la mémoire occupée par l'arbre de racine `*current`.
6. Pour tester vos fonctions, créez un fichier `main.c`, et écrivez une fonction de signature `int fill_dir(int k)`, qui prend en argument un entier `k` et crée `k/2` sous-répertoires, et dans chaque sous-répertoire met `(k/2)/2` sous-répertoires, et ainsi de suite, jusqu'à ce que `(k/2)/2/.../2` vaille 0 ou 1. Dans les derniers sous-répertoires, votre fonction mettra un ficher `tmp.txt`. Trouvez également une façon de nommer vos sous-répertoires. (*Indice :* `man 2 mkdir`). Chaque répertoire créé aura les droits `-rwxr-xr-x`.
7. Dans le fichier `main.c`, testez votre fonction `build_from_current_file` en commençant par appeler `fill_dir` avec une certaine valeur, puis construisez l'arbre, et finalement, affichez-le grâce à `print_tree`. Pensez aussi à libérer la mémoire occupée par l'arbre avec `delete_tree` (seulement après l'avoir affiché, évidemment).