Manipulations de fichiers

Exercice 1. Questions de cours Les questions de cours sont à destinées à vous permettre de vérifier votre compréhension du cours. Elles sont à travailler à l'avance et ne seront pas traitées en TD ou TP.

1. Pourquoi faut-il ouvrir les fichiers ?
2. Donnez trois modes de contrôle d'accès différents.
3. Qu'est-ce qu'un File Control Bloc ?
4. Quels sont les avantages et inconvénients des allocations contiguë, chaînée et indexée ?

Exercice 2. Copie de fichiers L'objectif de cet exercice est de programmer un équivalent simplifié de la commande de copie de fichier `cp`. Chaque partie demande d'étendre le programme précédent afin d'ajouter des fonctionnalités. Assurez-vous de bien tester votre code à chaque étape.

La documentation de toutes les fonctions à utiliser est disponible en section 2 des pages de manuels. Par exemple pour obtenir la documentation de la fonction `stat` il vous suffit d'utiliser la commande `man 2 stat`. Attention, les pages de manuel vous indiquent les fichier d'en-tête à inclure afin de pouvoir utiliser les fonctions qu'elles décrivent. N'oubliez pas d'inclure ces fichiers au début de votre programme.

Correction: *Pour tous les codes suivants, on va avoir besoin des includes suivants :*

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

Les correction des différentes question sont sous la forme de code simplifié sans gestion des erreurs afin de le rendre plus simple à suivre. Le code complet du programme avec la gestion des erreurs est donnée en fin de TD.

1. Étape 1 : Vous commencerez par écrire un premier programme de copie de fichier simple qui copie un seul fichier. Ce programme prend en paramètre un fichier source et un fichier cible et réalise la copie.

Vous aurez besoin des fonctions `open`, `close`, `read` et `write`. Il vous est conseillé d'implémenter la copie dans une fonction séparée afin de faciliter les étapes suivantes.

La copie doit être réalisée par blocs. Il s'aggit de lire un bloc de donnée depuis le fichier source et de l'écrire dans le fichier cible et de répéter jusqu'à ce que la fin du fichier source soit atteinte.

Correction:

```
int copyfile(const char *iname, const char *oname) {
    int idesc = open(iname, O_RDONLY);
    int odesc = open(oname, O_WRONLY | O_CREAT | O_EXCL, 0666);
    while (1) {
        char buffer[4096];
        int rcnt = read(idesc, buffer, sizeof(buffer));
        if (rcnt == 0)
            break; // End of file
        int pos = 0;
        while (rcnt != 0) {
            int wcnt = write(odesc, buffer + pos, rcnt);
            rcnt -= wcnt;
            pos  += wcnt;
        }
```

```
15      }
16      close(idesc);
17      close(odesc);
18      return 0;
19  }
```

2. Étape 2 : Votre premier programme ne préserve pas les droits d'accès aux fichiers qu'il copie. Vous allez maintenant le modifier pour que lors de la copie il récupère les droits du fichier source et les appliques au fichier cible.

Vous aurez besoin des fonctions `stat` et `chmod`, ou `fstat` et `fchmod`. Vous devez maintenant avoir une fonction qui réalise correctement la copie d'un fichier de manière fiable.

Correction:

```
1  int copyfile(const char *iname, const char *oname) {
2      int idesc = open(iname, O_RDONLY);
3      int odesc = open(oname, O_WRONLY | O_CREAT | O_EXCL, 0666);
4      struct stat istat;
5      fstat(idesc, &istat);
6      fchmod(odesc, istat.st_mode);
7      while (1) {
8          char buffer[4096];
9          int rcnt = read(idesc, buffer, sizeof(buffer));
10         if (rcnt == 0)
11             break; // End of file
12         int pos = 0;
13         while (rcnt != 0) {
14             int wcnt = write(odesc, buffer + pos, rcnt);
15             rcnt -= wcnt;
16         pos  += wcnt;
17       }
18     }
19     close(idesc);
20     close(odesc);
21     return 0;
22 }
```

3. Étape 3 : Nous allons maintenant ajouter la gestion de la copie de répertoires. Dans un premier temps nous considérerons le cas simple de la copie d'un répertoire ne contenant que des fichiers standards, pas de sous répertoire. Modifiez votre programme de manière à ce qu'il copie tous les fichiers contenus dans un répertoire source vers un répertoire cible déjà existant.

Vous aurez besoin des fonctions `opendir`, `closedir` et `readdir`. (Attention, il n'est pas garanti que les noms des répertoires source et destination finissent par le caractère '/'.

Correction:

```
1  char *makepath(const char *path, const char *file) {
2      int   lpath = strlen(path);
3      int   lfile = strlen(file);
4      char *res   = malloc(lpath + lfile + 2);
5      strcpy(res, path);
6      if (lpath != 0 && path[lpath - 1] != '/')
7        strcat(res, "/");
8      strcat(res, file);
9      return res;
10 }
11
12 void copydir(const char *idir, const char *odir) {
13     DIR *dir = opendir(idir);
14     struct stat istat;
15     stat(idir, &istat);
16     mkdir(odir, 0777);
17     chmod(odir, istat.st_mode);
18     while (1) {
```

```
19        struct dirent *ent = readdir(dir);
20        if (ent == NULL)
21          break;
22        if (ent->d_name[0] == '.')
23          continue;
24        char *iname = makepath(idir, ent->d_name);
25        char *oname = makepath(odir, ent->d_name);
26        copyfile(iname, oname);
27        free(iname);
28        free(oname);
29      }
30      closedir(dir);
31  }
32
33  int main(int argc, char *argv[]) {
34      if (argc != 3) {
35        fprintf(stderr, "error: missing arguments\n");
36        return EXIT_FAILURE;
37      }
38      copydir(argv[1], argv[2]);
39      return EXIT_SUCCESS;
40  }
```

4. Étape 4 : La dernière étape de la réalisation de ce programme de copie consiste à rendre la copie récursive. Votre programme final doit être capable de copier aussi bien des fichiers que des répertoires, et pour ces derniers la copie doit être récursive.

Les fonctions `stat` et `fstat` permettent de différentier les fichier des répertoires. Vous aurez besoin de la fonction `mkdir`.

Correction:

```
1  void copydir(const char *idir, const char *odir) {
2      DIR *dir = opendir(idir);
3      struct stat istat;
4      stat(idir, &istat);
5      mkdir(odir, 0777);
6      chmod(odir, istat.st_mode);
7      while (1) {
8        struct dirent *ent = readdir(dir);
9        if (ent == NULL)
10         break;
11       if (ent->d_name[0] == '.')
12         continue;
13       char *iname = makepath(idir, ent->d_name);
14       char *oname = makepath(odir, ent->d_name);
15       struct stat istat;
16       stat(iname, &istat);
17       if (S_ISDIR(istat.st_mode))
18         copydir(iname, oname);
19       else
20         copyfile(iname, oname);
21       free(iname);
22       free(oname);
23     }
24     closedir(dir);
25 }
26
27 int main(int argc, char *argv[]) {
28     if (argc != 3) {
29       fprintf(stderr, "error: missing arguments\n");
30       return EXIT_FAILURE;
```

```
31      }
32      struct stat istat;
33      stat(argv[1], &istat);
34      if (S_ISDIR(istat.st_mode))
35        copydir(argv[1], argv[2]);
36      else
37        copyfile(argv[1], argv[2]);
38      return EXIT_SUCCESS;
39  }
```

Exercice 3. Performances Sur un petit nombre de petits fichiers, les performances de votre programme sont à la fois peu importantes et difficiles à mesurer. Un cas où les performances peuvent être mauvaises est la copie d'un grand nombre de petits fichiers. Dans ce cas il est difficile de régler le problème car la copie d'un grand nombre de fichiers nécessitera toujours un grand nombre d'appels système.

Le cas de la copie de gros fichier est plus intéressant car la manière de copier à une grande importance. Afin de tester les performance de votre programme, il va falloir que vous mesuriez sont temps d'exécution lors de la copie de gros fichiers. Si vous ne disposez pas de tels fichier vous pouvez en créer avec un contenu aléatoire à l'aide de la commande suivante :

```
dd if=/dev/random of=toto count=2048
```

Le paramètre `of` permet de spécifier le nom du fichier à créer et le paramètre `count` indique sa taille en multiple de 512o. La commande précédente va donc créer un fichier de 1Mo.

1. Étudier l'impact de la taille du bloc utilisé pour la copie sur les performance de la copie.
Correction: *Le résultat est très dépendant d'un grand nombre de facteurs notament de la machine utilisée, de la taille du fichier copié et de sa présence dans les cache de l'OS. On peut toutefois remarquer que les petites tailles sont particulièrement lentes car elle impliquent un très grand nombre d'appels systèmes. De même, les très grande taille ne sont pas optimales car elle implique l'utilisation d'un grand nombre de pages mémoire ce qui n'est pas optimal. De manière générale, les meilleur performances sont obtenues lorsque la taille de bloc correspond à un petit nombre de pages mémoire.*

Choisir une bonne taille de bloc permet de réaliser la copie de manière relativement rapide et portable. Toutefois, un grand nombre d'appels système sont quand même nécessaires et les performances restent en dessous de ce que le materiel peut fournir. La plupart des systèmes ont maintenant ajouté des appels système spéciquement destinés à réduire ce problème. Sous Linux, c'est le rôle de l'appel système `copy_file_range` qui est destiné à remplacer la boucle de copie par blocs. L'utilisateur est toujours responsable d'ouvrir et fermer les différents fichiers mais la copie des données d'un fichier à l'autre est directement réalisée par cet appel. Cet appel n'étant pas standard, il est nécessaire d'ajouter la ligne :

```
#define _GNU_SOURCE
```

avant l'inclusion des fichiers d'en-tête dans votre programme pour qu'il soit disponible.

2. Modifiez votre fonction de copie de manière à ce qu'elle effectue la copie à l'aide de ce nouvel appel système et évaluez ses performances.
Correction: *Il suffit de remplacer la boucle de copie par bloc de la première question par la boucle suivante :*

```
1       size_t len = istat.st_size;
2       do {
3         ssize_t cnt = copy_file_range(idesc, NULL, odesc, NULL, len, 0);
4         if (cnt < 0) {
5           perror("cannot copy file");
6           exit(EXIT_FAILURE);
7         }
8         len -= cnt;
9       } while (len > 0);
```

Sur de gros fichiers on peut voir un gain de performance très notable. Le gain exact est très dépendant de la taille du fichier ainsi que du système de fichier utilisé.

Correction: Code complet

```
1  #include <stdlib.h>
2  #include <stdio.h>
```

```
3  #include <errno.h>
4  #include <string.h>
5
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <dirent.h>
10 #include <unistd.h>
11
12 char *makepath(const char *path, const char *file) {
13     int   lpath = strlen(path);
14     int   lfile = strlen(file);
15     char *res   = malloc(lpath + lfile + 2);
16     if (res == NULL) {
17       fprintf(stderr, "error: out of memory\n");
18       exit(EXIT_FAILURE);
19     }
20     strcpy(res, path);
21     if (lpath != 0 && path[lpath - 1] != '/')
22       strcat(res, "/");
23     strcat(res, file);
24     return res;
25 }
26
27 int copyfile(const char *iname, const char *oname) {
28     // Open input and output file. For the output file the O_EXCL mode is
29     // used to raise an error if the file already exist.
30     int idesc = open(iname, O_RDONLY);
31     if (idesc == -1) {
32       perror("cannot open input file");
33       exit(EXIT_FAILURE);
34     }
35     int odesc = open(oname, O_WRONLY | O_CREAT | O_EXCL, 0666);
36     if (odesc == -1) {
37       perror("cannot open output file");
38       exit(EXIT_FAILURE);
39     }
40     // Transfer access rights from input file to ourput file. If this was
41     // done at open time, the UMASK would have been applyed.
42     struct stat istat;
43     if (fstat(idesc, &istat) < 0) {
44       perror("cannot stat input file");
45       exit(EXIT_FAILURE);
46     }
47     if (fchmod(odesc, istat.st_mode) < 0) {
48       perror("cannot set output file mode");
49       exit(EXIT_FAILURE);
50     }
51     // Copy the file block by block instead of one character at a time for
52     // efficiency.
53     while (1) {
54       char buffer[4096];
55       // Read an input block. EAGAIN and EINTR are not fatal but mean
56       // that the read call sould be restarted.
57       int rcnt = read(idesc, buffer, sizeof(buffer));
58       if (rcnt == 0)
59         break; // End of file
60       if (rcnt < 0) {
```

```
61          if (errno == EAGAIN || errno == EINTR)
62            continue;
63          perror("cannot read from input file");
64          exit(EXIT_FAILURE);
65        }
66        // Write the output block. We have to repeat until the block is
67        // fully written.
68        int pos = 0;
69        while (rcnt != 0) {
70          int wcnt = write(odesc, buffer + pos, rcnt);
71          if (wcnt < 0) {
72            if (errno == EAGAIN || errno == EINTR)
73              continue;
74            perror("cannot write to output file");
75            exit(EXIT_FAILURE);
76          }
77          rcnt -= wcnt;
78          pos  += wcnt;
79        }
80      }
81      // And cleanup...
82      close(idesc);
83      close(odesc);
84      return 0;
85  }
86
87  void copydir(const char *idir, const char *odir) {
88      DIR *dir = opendir(idir);
89      if (dir == NULL) {
90        perror("cannot open input directory");
91        exit(EXIT_FAILURE);
92      }
93      // Make the output directory and ensure it have the same access right
94      // than the input one.
95      struct stat istat;
96      if (stat(idir, &istat) < 0) {
97        perror("cannot stat input file");
98        exit(EXIT_FAILURE);
99      }
100     if (mkdir(odir, 0777) < 0) {
101       perror("cannot create ouput directory");
102       exit(EXIT_FAILURE);
103     }
104     if (chmod(odir, istat.st_mode) < 0) {
105       perror("cannot set output dir mode");
106       exit(EXIT_FAILURE);
107     }
108     // Copy all contents of the input directory one by one either using
109     // copyfile or copy dir recursively.
110     while (1) {
111       // Read next entry from the directory. End of dir should be
112       // differentiated from errors using errno.
113       errno = 0;
114       struct dirent *ent = readdir(dir);
115       if (ent == NULL) {
116         if (errno == 0)
117           break;
118         perror("cannot read directory contents");
```

```
119         exit(EXIT_FAILURE);
120       }
121       // As the usual cp command, we don't copy the dotfiles. This
122       // also eliminate the '.' and '..' entry.
123       if (ent->d_name[0] == '.')
124         continue;
125       // Make the input and ouput file name. Some care is needed here
126       // to ensure that the directory separator is added if not
127       // already present.
128       char *iname = makepath(idir, ent->d_name);
129       char *oname = makepath(odir, ent->d_name);
130       // Check if the input is directory. In this case we should
131       // create the output directory and recursively call this
132       // function, else we call the copyfile.
133       struct stat istat;
134       if (stat(iname, &istat) < 0) {
135         perror("cannot stat input file");
136         exit(EXIT_FAILURE);
137       }
138       if (S_ISDIR(istat.st_mode))
139         copydir(iname, oname);
140       else
141         copyfile(iname, oname);
142       free(iname);
143       free(oname);
144     }
145     closedir(dir);
146 }
147
148 int main(int argc, char *argv[]) {
149     if (argc != 3) {
150       fprintf(stderr, "error: missing arguments\n");
151       return EXIT_FAILURE;
152     }
153     // Check if the input is a directory or a file to dispatch to the good
154     // function.
155     struct stat istat;
156     if (stat(argv[1], &istat) < 0) {
157       perror("cannot stat input file");
158       exit(EXIT_FAILURE);
159     }
160     if (S_ISDIR(istat.st_mode))
161       copydir(argv[1], argv[2]);
162     else
163       copyfile(argv[1], argv[2]);
164     return EXIT_SUCCESS;
165 }
```