IUT de Montpellier, Dép. Informatique
M2101 - Archi 2 V. Poupet

TD n° 3 - Manipulation de fichiers

Dans ce TD, nous allons utiliser les fonctions `open`, `close`, `read`, `write`, et `lseek` pour manipuler les fichiers directement en utilisant les appels système. Il ne faut donc pas utiliser les fonctions plus haut niveau telles que `fopen`, `fgetc`, `fputc`, `printf`, etc.

Si nécessaire, vous pouvez consulter les transparents du second cours sur l'ENT.

1 Lecture et copie

1 Cherchez à l'aide de la commande `man` les bibliothèques à inclure pour utiliser les appels système `open`, `close`, `read`, `write` et `lseek`.

Remarque : Pour indiquer que l'on veut la documentation des appels système, il faut utiliser l'option `2` dans `man` (par exemple « `man 2 read` »).

Solution : La section *Synopsis* dans le manuel indique les bibliothèques à inclure :

```
...
SYNOPSIS
     #include <sys/types.h>
     #include <sys/uio.h>
     #include <unistd.h>

     ssize_t
     pread(int d, void *buf, size_t nbyte, off_t offset);

     ssize_t
     read(int fildes, void *buf, size_t nbyte);
...
```

2 Écrivez un programme qui ouvre le fichier `message.txt` du répertoire courant et affiche son contenu à l'écran (pensez à créer un fichier `message.txt` contenant du texte avant de tester votre programme).

Indication : Pour détecter que l'on arrive à la fin du fichier il faut regarder le résultat renvoyé par `read`, qui correspond au nombre de caractères lus.

Rappel : les descripteurs de fichier 0, 1 et 2 correspondent respectivement à l'entrée standard, la sortie standard et la sortie d'erreur. Vous pouvez donc par exemple utiliser la fonction `write` en lui passant le descripteur 1 pour écrire du texte à l'écran.

Solution :

```
int main(int argc, char **argv) {
  int fdin = open("message.txt", O_RDONLY);

  char buffer[256];
  ssize_t nbread;
  do {
```

```
    nbread = read(fdin, buffer, 256);
    write(1, buffer, nbread);
  }
  while (nbread == 256);
}
```

3 Écrivez un programme `copy.c` qui copie le contenu du fichier `message.txt` dans un fichier `copie.txt`.

Indication : Il faut ouvrir le premier fichier en lecture et le second en écriture (en le créant si nécessaire) et copier le contenu du premier dans le second.

Attention : Si le fichier de destination n'existe pas, il faut le créer en lui donnant des permissions. Ces permissions sont données à la fonction `open` en quatrième argument. Vous pouvez utiliser la notation en octal comme pour la fonction `chmod` (par exemple 0644) ou utiliser des *flags*, dont vous trouverez le détail dans la page de manuel de `open`.

Solution :

```
int main(int argc, char **argv) {
  int fdin, fdout;
  ssize_t nbread;
  char buffer[4096];

  fdin = open("message.txt", O_RDONLY);
  if (fdin == -1) {
    perror(argv[1]);
    exit(1);
  }
  fdout = open("copie.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fdout == -1) {
    perror(argv[2]);
    exit(1);
  }

  do {
    nbread = read(fdin, buffer, 4096);
    write(fdout, buffer, nbread);
  } while (nbread == 4096);
}
```

2 Lecture d'une ligne

On veut maintenant écrire une fonction `int lireligne(int fd, char *s, int size)` permettant de lire une ligne d'un fichier :

- la fonction lit des caractères dans le fichier correspondant au descripteur `fd` jusqu'à lire un retour à la ligne (`\n`) ou avoir lu `size` caractères ;
- elle place les octets lus dans le tableau `s` passé en argument (on suppose que le tableau `s` peut contenir `size` caractères) ;
- le résultat renvoyé par la fonction est le nombre de caractères effectivement lus dans le fichier.

4 En utilisant les appels système, écrivez la fonction `lireligne` en lui faisant lire les caractères un par un afin de pouvoir s'arrêter facilement si elle rencontre un retour à la ligne.

Indication : Pour lire les caractères un par un, vous pouvez déclarer un tableau d'un caractère `char c[1]` ou bien déclarer un caractère `char c` et passer l'adresse du caractère `&c` à la fonction `read`.

Solution :

```
int lireligne(int fd, char *buffer, int size) {
  char c; // caractère lu
  int i; // indice dans la chaîne buffer
  ssize_t nbread; // octets lus par read

  // la boucle s'arrête lorsque i = size
  for (i = 0; i < size; i++) {
    nbread = read(fd, &c, 1);
    if (nbread == -1) {
      // erreur de lecture
      return -1;
    } else if (nbread == 0) {
      // fin du fichier
      break;
    } else {
      // un nouveau caractère a été lu
      buffer[i] = c;
    }

    if (c == '\n') {
      // fin de ligne : on passe à l'indice
      // suivant et on arrête la lecture
      i++;
      break;
    }
  }
  return i;
}
```

5 Vérifiez le bon fonctionnement de `lireligne` en lui faisant lire des lignes du fichier `message.txt`. Essayez de lire à la fois des lignes plus courtes et des lignes plus longues que la valeur indiquée.

Chaque appel à l'une des commandes `open`, `close`, `read`, `write` et `lseek` provoque un appel système, qui est très coûteux en nombre d'opérations effectuées puisqu'il faut sauvegarder l'état du processus, donner le contrôle au système pour effectuer l'action puis recharger l'état du processus pour reprendre le fonctionnement.

Il est donc préférable de réduire autant que possible le nombre d'appels systèmes effectués. En particulier, il est très inefficace de lire le contenu d'un fichier en faisant un appel à `read` pour chaque caractère. Pour lire un fichier, on créé donc un tableau pouvant contenir des caractères (un tampon ou *buffer*) et on lit les octets en remplissant le tampon à chaque lecture (tant qu'on n'est pas au bout du fichier).

Si la taille du tampon est trop petite, on va effectuer beaucoup d'appels système. Si elle est trop grande, on va utiliser de la mémoire inutilement. La taille optimale est difficile à déterminer. Les documentations *Unix* conseillent d'utiliser un tampon de 4096 octets.

Dans le cas de la fonction `lireligne`, on peut utiliser le tableau passé en argument comme tampon, et donc lire `size` caractères à la fois.

Si les caractères lus contiennent un retour à la ligne, la fonction renvoie le nombre de caractères lus jusqu'au retour à la ligne. Le problème est que pour remplir le tampon, la fonction `read` a avancé le curseur de lecture dans le fichier. Si l'on trouve un retour à la ligne dans les octets lus, il faut donc replacer le curseur juste après ce retour à la ligne pour ne pas perturber les lectures suivantes dans le fichier (qui reprendront donc au début de la ligne suivante). On utilise pour cela la fonction `lseek`.

6 Modifiez la fonction `lireligne` pour qu'elle lise les caractères du fichier par blocs de `size` caractères à la fois.

Indication : La fonction doit donc utiliser `read` pour remplir le tableau `s`, puis parcourir le tableau afin de localiser le premier retour à la ligne. Quand un retour à la ligne est trouvé, il faut remettre le curseur de lecture du fichier au bon emplacement. La fonction doit également renvoyer le nombre d'octets lus jusqu'au retour à la ligne.

Solution :

```c
int lireligne(int fd, char *buffer, int size) {
  // on essaie de lire size caractères dans fd
  // on place les caractères dans buffer
  // nbread est le nombre de caractères lus
  ssize_t nbread = read(fd, buffer, size);
  if (nbread == -1) {
    // si la lecture a échoué
    return -1;
  }

  int i; // indice de parcours de buffer
  // on parcourt les nbread premières cases de buffer
  // pour trouver un éventuel retour à la ligne
  for (i = 0; i < nbread; i++) {
    if (buffer[i] == '\n') {
      // si on trouve un retour à la ligne
      // on incrémente i et on arrête le parcours
      i++;
      break;
    }
  }
  // ici, i correspond au nombre de caractères retenus

  // il faut maintenant reculer le curseur de lecture
  // nb caractères lus = nbread
  // nb caractères conservés = i
  // -> il faut reculer de (nbread-i)
  lseek(fd, i - nbread, SEEK_CUR);
  return i;
}
```

3 Passage de paramètres

Jusqu'ici, tous les programmes que nous avons écrits avaient une fonction `main` ne prenant aucun argument : `int main()`. Il est toutefois possible de lui passer deux arguments, un entier et un tableau de chaînes de caractères : `int main(int argc, char **argv)` [1]. Dans ce cas, si l'exécutable est appelé depuis un terminal avec des arguments, l'entier `argc` correspond au nombre d'arguments, et le tableau `argv` contient tous les arguments sous forme de chaînes de caractères.

Le nom de l'exécutable qui a été appelé est toujours considéré comme le premier argument (donc il y a toujours au moins un argument). Si par exemple on a créé un exécutable `prog` et qu'on l'exécute avec la commande

```
$ ./prog -l toto
```

alors `argc` vaut 3 et le tableau `argv` contient les chaînes de caractères « `prog` », « `-l` » et « `toto` » aux indices 0, 1 et 2 respectivement.

7 Modifiez les programmes des questions 2 et 3 pour qu'ils prennent en paramètres les noms des fichiers sur lesquels ils travaillent (le premier attend un nom de fichier, le second en attend deux). Faites en sorte que le programme s'interrompe en affichant un message d'erreur si le nombre d'arguments reçus ne correspond pas au nombre attendu.

Indication : Pour interrompre l'exécution du programme, vous pouvez utiliser la commande « `exit(1);` » de la bibliothèque `<stdlib.h>`.

4 Gestion des erreurs (`errno` et `perror`)

Lorsqu'une erreur se produit pendant l'exécution d'un appel système, la fonction appelée renvoie en général le résultat -1. Pour connaître la nature de l'erreur qui s'est produite, il faut regarder la valeur de la variable `errno`.

8 Regardez rapidement la page de documentation correspondant à `errno`. Quelle bibliothèque faut-il ajouter à votre programme si vous voulez utiliser cette variable ?

Solution :

```
$ man errno

ERRNO(3)    Linux Programmer's Manual    ERRNO(3)
NAME
    errno - number of last error
SYNOPSIS
    #include <errno.h>
...
```

Il faut donc inclure la bibliothèque `<errno.h>`

Lorsqu'une erreur se produit, la fonction qui a généré l'erreur donne une valeur à la variable `errno` puis renvoie un code d'erreur (dans le cas des appels systèmes c'est -1). La documentation des fonctions utilisant ce mécanisme donne en général la liste des erreurs pouvant se produire.

9 Regardez la documentation de l'appel système `open` et cherchez la section donnant la liste des erreurs.

1. On peut également utiliser les déclarations `int main(int argc, char argv[][])` ou même `int main(int argc, char *argv[])` qui sont équivalentes.

Chaque erreur correspond à un nom, qui est en réalité un mot clé associé à un entier (les valeurs associées à chaque mot-clé sont définies dans les bibliothèques mais il n'est pas nécessaire de les connaître puisqu'on doit justement utiliser les mots-clés).

On trouve par exemple dans la documentation de `open` (éventuellement en français sur votre machine) :

```
ENOENT O_CREAT is not set and the named file does not
       exist.  Or, a directory component in pathname does
       not exist or is a dangling symbolic link.
```

qui indique que lorsqu'on essaie d'ouvrir un fichier qui n'existe pas sans donner l'option `O_CREAT` (qui force la création des fichiers inexistants), l'erreur `ENOENT` est placée dans la variable `errno` et la fonction `open` renvoie -1.

Pour connaître l'erreur qui s'est produite, on peut alors tester l'égalité de `errno` et de chacune des erreurs possibles : « `if (errno == ENOENT)` ».

10 Modifiez le programme de la question 2 pour qu'il affiche un message d'erreur spécifique lorsque le fichier à ouvrir n'existe pas.

Indication : Il faut tester si le résultat de `open` est -1, et si c'est le cas, regarder si `errno` est égal à `ENOENT`.

Solution :

```
...
  int fdin = open(argv[1], O_RDONLY);
  if (fdin == -1) {
    if (errno == ENOENT) {
      write(1, "Le fichier n'existe pas\n", 24);
    }
    exit(1);
  }
...
```

Cette façon de faire est cependant très pénible à réaliser (dans la question précédente, on ne tient compte que d'une seule erreur possible, mais il faudrait toutes les faire). Fort heureusement, il existe une fonction permettant de décrire automatiquement l'erreur qui s'est produite : `perror`.

La fonction `perror` prend un argument qui est une chaîne de caractères et affiche cette chaîne de caractères suivie d'une description en langage courant de l'erreur qui se trouve dans la variable `errno`. Ainsi, si l'on appelle « `perror(argv[1]);` », le programme affichera

```
$ ./a.out toto.txt
toto.txt: No such file or directory
```

si le fichier `toto.txt` n'existe pas.

11 Modifiez votre programme pour qu'en cas d'échec d'ouverture du fichier il affiche une description de l'erreur produite et s'interrompe.

Solution :

```
...
  int fdin = open(argv[1], O_RDONLY);
  if (fdin == -1) {
```

```
    perror(argv[1]);
    exit(1);
  }
...
```

12 Modifiez tous les programmes écrits dans ce TD pour gérer les éventuelles erreurs après les appels systèmes.

De manière général, il est très important de toujours vérifier qu'il n'y a pas eu d'erreur après un appel à `open`. Pour les autres il est toujours conseillé de surveiller les erreurs éventuelles mais elles se produisent moins fréquemment.

5 Informations sur les fichiers

La fonction `stat` correspond à un appel système permettant d'obtenir des informations concernant l'inode d'un fichier dont le nom est passé en paramètre. Son prototype est :

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

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

Les informations sur le fichier sont placées dans la structure `buf` passée en argument. Les champs de la structure sont décrits dans la page de manuel.

Par exemple, pour obtenir le numéro du propriétaire du fichier `blop.txt`, on peut utiliser les instructions suivantes :

```
struct stat buf;
uid_t proprietaire;
if (stat("blop.txt", &buf) == 0) {
  proprietaire = buf.st_uid;
} else {
  perror("blop.txt");
}
```

13 Écrivez un programme `bigfich.c` qui prend en argument une liste de noms de fichiers et affiche le nom du plus grand d'entre eux.

Le nombre de noms de fichiers passé en argument doit être au moins 1 mais peut être arbitrairement grand. Affichez un message d'erreur si aucun fichier n'est passé en argument, ainsi qu'un message d'erreur pour chaque nom de fichier pour lequel on ne peut pas obtenir d'informations à l'aide de la fonction `stat`.

Solution :

```
int main(int argc, char **argv) {
  struct stat buf;
  off_t max_size = -1;
  char *nom_fichier;
```

```
  for (int i = 1; i < argc; i++) {
    if (stat(argv[i], &buf) == 0) {
      if (buf.st_size > max_size) {
        max_size = buf.st_size;
        nom_fichier = argv[i];
      }
    } else {
      perror(argv[i]);
    }
  }

  if (max_size >= 0) {
    printf("Le plus grand fichier est %s (%lld)\n", nom_fichier, max_size);
  } else {
    printf("Erreur: aucun fichier valide\n");
  }
}
```