Module ITC313 - Informatique

Partie C / C++

TD3 Exercice sur le parcours de fichiers, fusion etc

Benoît Darties - benoit.darties@u-bourgogne.fr
Université de Bourgogne

Année universitaire 2016-2017

---

1 Comprendre et manipuler les descripteurs

Exercice 1 : *Gestion des descripteurs et identifiants de descripteurs*

Cet exercice a pour but de vous faire comprendre l'intérêt des descripteurs. Un descripteur est une petite valeur qui référence une ligne dans la table des descripteurs locale d'un processus. Il identifie un fichier dans lequel on peut lire et/ou écrire. On rappelle que sous UNIX tout est vu comme un fichier. Ainsi le flux d'entrée standard `stdin`*, qui est par défaut le clavier, est associé au descripteur 0, tandis que les flux de sortie standard et erreur* `stdout` *et* `stderr`*, qui sont par défaut l'écran, sont associés aux descripteurs 1 et 2. A chaque fois que l'on ouvre correctement un fichier en lecture et/ou écriture, un nouveau descripteur est créé avec la plus petite valeur disponible. Dans cet exercice, nous allons essayer de prédire les différentes valeurs qui vont être affectées aux descripteurs à chaque fois que la fonction* `open` *est appelée. On rappelle que cette fonction ouvre un fichier et renvoie un descripteur sur ce fichier, et que c'est avec ce descripteur que l'on va pouvoir lire ou écrire sur le fichier.*

On considère le fichier `descripteurs.c` suivant :

"Programme descripteurs.c"

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main() {
    int gadget, derrick, kojak, columbo;

    gadget = open("/etc/passwd", O_RDONLY);
    derrick = open("/etc/fichierInexistant", O_RDONLY);
    kojak = open("/etc/group", O_RDONLY);
    close(gadget);
    columbo = open("/etc/profile", O_RDONLY);
    exit(EXIT_SUCCESS);
}
```

Seuls les fichiers `/etc/passwd`, `/etc/group` et `/etc/profile` existent et sont accessibles en

lecture par l'utilisateur exécutant ce programme.

On rappelle que la valeur allouée à chaque nouveau descripteur créé par `open()` est le plus petit entier naturel disponible non référencé dans la table des descripteurs.

1. Quelles devraient alors être les valeurs respectives des descripteurs `gadget`, `derrick`, `kojak` et `columbo` ?
2. Vérifiez votre réponse en affichant les valeurs de ces descripteurs au moyen d'une instruction `printf()` adéquate.
3. Quel serait l'effet de l'instruction `close(1);` sur l'exécution du programme ? Vérifiez votre réponse en ajoutant une suite d'instructions appropriées et en constatant l'effet attendu.

2 Mécanismes d'erreur d'accès aux fichiers

Exercice 2 : *Erreur a l'ouverture d'un fichier, et code retour d'un appel systeme*

*Les objectifs de cet exercice sont doubles : tout d'abord, au travers d'un exemple simple, il vous est montré comment récupérer des paramètres passés en ligne de commande lors de l'appel d'un programme. Ceci nous permet d'avoir plus de flexibilité par la suite pour aborder le second objectif : récupérer le statut d'une demande de création de descripteur avec la fonction `open()`, et déterminer pourquoi cette demande a échoué si le descripteur n'a pas été créé. Nous allons voir que selon les causes d'échec de la fonction `open()`, une variable globale spéciale appelle `errno` est mise à jour, et va permettre d'afficher un message d'erreur correspondant à la cause de l'échec de la fonction. Ce concept de code retour et de message d'erreur pré-enregistré est illustré ici avec la fonction `open()` mais marche avec la quasi-totalité des fonctions et appels systèmes standards en C. Il sera par la suite **très vivement recommandé** d'utiliser des mécanismes similaires pour toutes les fonctions de vos codes, ce*

1. **Partie 1 : passage de paramètres**
   (a) Recopiez ou récupérez le fichier `passageParametres.c` présenté ci-après :

"Programme passageParametres.c"

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      int i;
6      printf("la variable argc vaut %i, soit %i parametres entres\n",
           argc, argc-1);
7
8      for(i=0; i< argc; i++) {
9          printf("le parametre numero %i est argv[%i]=%s\n", i, i,
               argv[i]);
10     }
11 }
```

   (b) Compilez ce programme en un fichier exécutable nommé `passageParam` (en utilisant l'option `-o`), puis lancez-le en utilisant les lignes de commandes suivantes :
   — `./passageParam`
   — `./passageParam jason freddy michael`
   — `./passageParam candyman1 candyman2 candyman3 candyman4 candyman5`

Remarques / questions : benoit.darties@u-bourgogne.fr

(c) En observant le résultat de ces appels et en étudiant le code de `passageParametres.c`, à quoi correspondent resepectivement `argc`, `argv[0]`, `argv[i]` pour `i` quelconque, et d'une manière générale `argv`. Notez que ces variables sont des paramètres de la fonction `main()`. Quels sont les types de chacune de ces variables ?

2. **Partie 2 : détection d'erreurs**

(a) Recopiez ou récupérez le fichier `erreurOuverture.c` présenté ci-après :

"Programme erreurOuverture.c"

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <errno.h>
5  int main(int argc, char **argv) {
6      if (argc !=2)  {
7          printf("usage : %s nom_fichier\n", argv[0]);
8          exit(1);
9      }
10
11     if (open(argv[1], O_RDONLY) ==-1) {
12         perror("erreur fonction open() : ");
13         printf("la valeur errno associee est %i\n", errno);
14     }
15
16     else
17         printf("ouverture en lecture OK\n");
18 }
```

(b) Compilez ce programme en un fichier exécutable nommé `erreurOuverture` (en utilisant l'option `-o`). Exécutez ensuite ce programme en passant en paramètre un nom de fichier à ouvrir. Ce programme va simplement essayer d'ouvrir en lecture le fichier passé en paramètre, et retournera un descripteur valide s'il réussit. S'il échoue, un message d'erreur s'affichera expliquant la raison de l'échec. Exécutez les commandes suivantes en vérifiant pour chaque fichier entré en paramètre si ce dernier existe, et si vous avez les droits de lecture sur ce dernier :
— `./erreurOuverture /etc/passwd`
— `./erreurOuverture /etc/sudoers`
— `./erreurOuverture /etc/blabla`

(c) Quels sont les différentes valeurs de la variable globale `errno` ? notez que vous n'avez pas déclaré cette variable mais que cette dernière est automatiquement gérée par le compilateur pour la gestion des messages d'erreur, à partir du moment ou le fichier en-tête `errno.h` est ajouté à votre programme.

(d) Ouvrez le manuel de la fonction `open()` avec la commande `man 2 open`, descendez à la section ERREURS et notez les différents mots-clés déterminant les erreurs : `EACCES, EAGAIN, EDQUOT`, ... Affichez ensuite la valeur numérique de ces mots-clés au début de votre programme, en ajoutant les instructions adéquates. Un exemple d'instruction est donné ci-après :

```
printf("la valeur de EACCES est %i\n", EACCES);
```

Relancez les commandes précédentes en faisant varier le fichier ouvert. Lorsque l'ouverture échoue, notez la valeur de `errno`, retrouvez le mot-clé correspondant à cette valeur, et en vous aidant du manuel de la fonction `open()`, vérifiez la concordance entre la signification dans le manuel du mot-clé associé à l'erreur générée, et le message d'erreur affiché sur la console.

Remarques / questions : benoit.darties@u-bourgogne.fr

(e) À quoi sert la fonction `perror()` ? Comment fonctionne cette dernière pour déterminer le bon message d'erreur à afficher ? Rechercher si besoin dans le manuel.

(f) Modifiez votre programme en y ajoutant une instruction permettant de lire les 10 premiers octets du fichier ouvert, avec la fonction `read()`, et en ajoutant un test sur la valeur retournée par `read()`. Si la valeur retournée est `-1`, ajoutez les instructions permettant de voir la cause du problème de lecture en utilisant la fonction `perror()` et la valeur du code d'erreur `errno`.

(g) De façon similaire à ce qui a été fait avec `open()`, vérifiez dans le manuel de `read` (commande `man 2 read` que le message affiché correspond au code d'erreur associé dans le manuel.

3 Lecture et écriture de fichiers

Les exercices de cette question reprennent les éléments usuels permettant la manipulation de fichiers. Pour manipuler un fichier, il faut créer un descripteur sur ce fichier avec la fonction `open()`, et effectuer des opérations de lecture et écriture (fonctions `read()` et `write()`) en passant par des mémoires tampon (ou buffer) qui sont généralement des tableaux de caractères de taille fixe. Une fois les opérations de lecture et écriture effectuées, on ferme le descripteur, avec la fonction `close()`.

Exercice 3 : *Lecture du contenu d'un fichier, utilisation d'une memoire tampon*

Lorsque l'on lit un fichier à partir d'un descripteur ouvert sur ce dernier, on utilise une mémoire tampon (en anglais buffer) représentée par un tableau de caractères de taille fixe. Cette zone mémoire étant généralement plus petite que la taille du fichier à lire, il est nécessaire de faire plusieurs lecture. A chaque lecture, le contenu de la mémoire est écrasé par la nouvelle lecture effectuée. La taille du buffer a donc une incidence sur le nombre d'opérations réalisées à chaque lecture.

Dans cet exercice, nous examinons un code simple permettant de copier le contenu du fichier `/etc/passwd` vers un autre fichier `~/copiePasswd` qui sera situé dans votre répertoire courant. Recopiez ou récupérer le code du fichier `copiefichier.c` ci-après :

"Programme copiefichier.c"

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6
7  int main() {
8      int nbLus;
9      int fd, fd2;
10     char buffer[1024];
11     fd = open("/etc/passwd", O_RDONLY);
12     fd2 = open("~/copiePasswd", O_WRONLY | O_CREAT, 0644);
13     do {
14         nbLus = read(fd, buffer, 900);
15         write(fd2, buffer, nbLus);
16     } while (nbLus > 0);
17 }
```

1. Combien d'octets sont lus à chaque opération de lecture sur le fichier `/etc/passwd` ?
2. Où sont stockés les octets lus à chaque itération ?
3. En ne changeant que l'instruction de lecture et pas le reste du code, jusqu'à combien d'octets pourrait-on se permettre de lire sur le fichier ?
4. Décrire quel impact peut alors avoir la taille du tableau buffer ? Penser notamment à ce qui se passe si l'on double ou réduit de moitié la taille de ce tableau.

Exercice 4 : *fusion de fichiers*

Nous étendons les compétences vues dans l'exercice précédent sur la lecture et l'écriture de fichiers au travers d'un petit exercice visant à fusionner et séparer deux fichiers. Cet exemple a priori simple doit mettre en évidence les problèmes liés à la lecture d'un fichier dont la taille n'est pas un multiple de la taille du buffer utilisé pour stocker temporairement les octets lus.

Deux fichiers nommés `sangoku.txt` et `vegeta.txt` sont mis à votre disposition. Il est recommandé de récupérer ces fichiers, mais vous pouvez créer vos propres fichiers en y insérant du texte de votre choix. La seule contrainte est que les deux fichiers doivent être de taille identique.

1. Ecrire un programme `fusion.c` qui lit les deux fichiers `sangoku.txt` et `vegeta.txt` en alternance, et les fusionne en un fichier `gogeta.txt` de la façon suivante : on écrit dans `gogeta.txt` les 10 premiers octets de *sangoku.txt*, puis les 10 premiers de *vegeta.txt*, puis les 10 suivants de *vegeta.txt* et ainsi de suite jusqu'à parcourir entièrement les deux fichiers `sangoku.txt` et `vegeta.txt`. Attention à la dernière lecture ! On rappelle que si les fichiers n'ont pas une taille qui est un multiple de 10, le nombre d'octets réellement lus est renvoyé par la fonction `read()`. Il est donc recommandé d'écrire à chaque fois le nombre d'octets réellement lus. Pour vous aider, un squelette de programme, à compléter, est disponible dans l'archive citée précédemment.
2. On supprime ensuite les fichiers `sangoku.txt` et `vegeta.txt` et on ne garde que le fichier `gogeta.txt`. Ecrire le programme inverse `separation.c` qui, à partir du fichier *gogeta.txt* créé dans la question précédente, permet de recréer les fichiers `sangoku.txt` et `vegeta.txt`. Cette question est beaucoup plus difficile que la précédente : en effet, dans le cas où *gogeta.txt* n'a pas une taille qui est un multiple de 20, il faut pouvoir traiter correctement le dernier bloc qui sera lu. On ne peut donc pas lire le fichier par blocs de 10 et copier tantôt vers `sangoku.txt`, et tantôt vers `vegeta.txt`. Il faut lire par blocs de 20, récupérer le nombre d'octets réellement lus, et copier la première moitié du buffer dans le fichier `sangoku.txt` et la seconde moitié dans le fichier `vegeta.txt`. Pour vous aider à *partager* un buffer en deux, on vous donne le code suivant :

```
1     char buffer1[20];
2     char buffer2[10];
3     char buffer3[10];
4     int nbLus;
5     ...
6
7     // copie des nbLus/2 premiers caracteres dans buffer2
8    memcpy(buffer2, &(buffer1[0]), nbLus/2);
9
10   // copie des nbLus/2 derniers caracteres dans buffer3
11   memcpy(buffer3, &(buffer1[nbLus/2]), nbLus/2);
```

Ce code permet de copier le contenu d'un tableau `buffer` qui peut contenir au plus 20 caractères, dans deux tableaux nommés `buffer1` et `buffer2`. On vous donne également un squelette de programme pour vous aider.

Remarques / questions : benoit.darties@u-bourgogne.fr

4 Pour aller plus loin sur les descripteurs et les redirections

Exercice 5 : *Descripteurs, appels systemes dup() et dup2()*

Cet exercice relativement difficile présente deux appels systèmes nommés `dup()` *et* `dup2()`*. Ces deux fonctions permettent de manipuler la table de descripteurs des fichiers associée à un processus, en dupliquant des entrées de la table des descripteurs, ou en réouvrant des descripteurs existants en modifiant leur valeur.*

Les appels système `dup()` et `dup2()` offrent un moyen simple et efficace de manipuler la table les descripteurs du processus, permettant notamment de dupliquer un descripteur. L'appel système `dup()` renvoie un nouveau descripteur à partir d'un descripteur déjà existant. L'appel système `dup2()` permet de spécifier la valeur du nouveau descripteur à récupérer. Le principal intérêt de `dup2()` est qu'il permet notamment d'écraser des descripteurs existants : si le nouveau descripteur est déjà associé à un fichier, ce dernier est d'abord fermé au moyen de l'appel système `close()` puis réouvert et pointe vers la même entrée que le descripteur à dupliquer.

1. Consultez le manuel de `dup()` et `dup2()` pour comprendre fonctionnement de ces fonctions.
2. Quelle est la particularité du fichier `/dev/null` ?
3. Recopiez ou récupérez le programme `testDupDup2.c` présenté ci-après :

"Programme testDupDup2.c"

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fdr = open("result", O_WRONLY | O_CREAT | O_APPEND, 0666);
    int fdnull = open("/dev/null", O_WRONLY);
    int fdnouveau = dup(2);

    write(1, "Hello World\n", 13);

    dup2(fdr, 1);
    write(1, "Hello World again\n", 18);

    write(fdnouveau, "Un dernier message?\n", 20);
    dup2(fdnull, 1);
    write(1, "Encore un pour la route?\n",25);
}
```

4. Exécutez ensuite ce programme en dressant la table des descripteurs et les fichiers sur lesquels chaque descripteur est censé pointer. Pour vous aider, rajoutez les instructions (fonctions `printf()` permettant d'afficher le contenu des différents descripteurs. Si ces descripteurs venaient à être modifiés après un appel à `dup` ou `dup2`, rajoutez les instructions d'affichage des valeurs avant et après modification. Où sont alors écrits les différents messages de ce programme ? Essayez alors d'en déduire le fonctionnement de `dup` et `dup2`.
5. Même question si vous lancez l'exécution de ce programme avec la ligne de commande `./testdupdup2 > fichierLog`
6. Même question si vous lancez l'exécution de ce programme avec la ligne de commande `./testdupdup2 2> fichierLog2`
7. Même question si vous lancez l'exécution de ce programme avec la ligne de commande `./testdupdup2 > fichierLog 2> fichierLog2`

Remarques / questions : benoit.darties@u-bourgogne.fr