EPITA - Spé:
C: Entrée - Sortie.

Marwan BURELLE*

1 Manipulations élémentaires

Nous commencerons ce TP avec des manipulations simples sur les descripteurs de fichiers à l'aide des fonctions `read(2)`, `write(2)`, `open(2)` et `close(2)`.

Remarque 1.1 (Compilations):
Comme vous l'avez vu lors des premiers TP de C, il est possible de compiler avec `make(1)` sans faire de fichier `Makefile`. Cette compilation se fait avec la configuration par défaut de `make(1)` :

```
> ls
fic.c
> make fic
cc -O -pipe fic.c -o fic
> ls
fic   fic.c
```

Si vous voulez forcez `make(1)` à utiliser certains options de compilation et le compilateur `gcc(1)` (un peu inutile, `cc(1)` et `gcc(1)` correspondent au même programme sur FreeBSD) vous pouvez utiliser les variables d'environnement `CFLAGS` et `CC`. Voici un exemple en `tcsh(1)` (votre shell par défaut) :

```
> setenv CC gcc
> setenv CFLAGS ' -W -Wall -Werror -ansi -pedantic'
> ls
fic.c
> make fic
gcc  -W -Wall -Werror -ansi -pedantic fic.c -o fic
>
```

Pour que ces variables soient définies définitivement, vous pouvez les fixer dans votre `.mytcshrc` (ou tout autre fichier de configuration.)

Remarque 1.2 (Documentations):
Les pages de manuel des appels systèmes tels que `read`, `write`, `open` et `close` se trouvent dans la section 2. Pour les lire avec la commande `man(1)` il faut d'abord préciser le numéro de section, puis le nom de l'appel :

```
> man 2 read
READ(2)                    FreeBSD System Calls Manual                    READ(2)
```

*marwan.burelle@lse.epita.fr

```
NAME
     read, readv, pread, preadv -- read input

LIBRARY
     Standard C Library (libc, -lc)

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

     ssize_t
     read(int d, void *buf, size_t nbytes);
...
```

De manière générale, lorsqu'une commande ou une fonction dispose d'une page de manuel, elle sera présentée sous la forme : `cmd(n)` *ou* `cmd` *est le nom de la commande et* `n` *le numéro de section.*

Exercice 1.1:
On se propose d'écrire un message à l'aide de la commande `write(2)` *sur la sortie standard. On rappelle que le descripteur de fichier de la sortie standard est désigné par la macro* `STDOUT_FILENO`.

Compléter le programme C du listing 1 pour afficher un message sur la sortie standard.

Listing 1 – `exo_1_1.c`

```c
/* exo_1_1.c Exercice 1.1 */
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
  int w;
  w = /* faire l'affichage avec write(2) ici */;
  /*
   * On fait un peu de controle d'erreur, si le retour de write(2) est
   * negatif, on affiche le message d'erreur a l'aide de perror(3).
   */
  if (w < 0)
    perror("Erreur lors de l'ecriture: ");
  return(0);
}
```

Exercice 1.2:
On va maintenant réécrire la commande `cat(1)` *à l'aide de* `write(2)`, `read(2)` *et* `open(2)`.

Dans un premier temps, on va utiliser un buffer de 1 octets, puis on étendra ce buffer pour comparer l'impacte sur la vitesse de la commande.

Le listing 2 présente une solution complète. On notera au passage la définition de la macro `STEP` *définissant le nombre d'octet lu à chaque passe.*

Remplacer la valeur actuelle (1) par une valeur plus importante (comme 512 ou 4096) et comparer le résultat sur des fichiers de grande taille.

Le programme `cat(1)` *permet également d'afficher ce qui est entré sur l'entrée standard lorsque le nom du fichier fourni est* `-`*, la gestion de ce cas particulier est* prévu *mais pas implanté dans le code de l'exercice, à vous de l'ajouter.*

Remarque 1.3:
On notera dans le listing 2 que l'on vérifie l'interruption de la lecture : si `read(2)` *renvoie une valeur négative et que la variable* `errno(3)` *vaut* `EINTR`*. Dans ce cas, on continue la lecture. Ce cas peut se présenter lorsque des signaux (rattrapés, pas ignorés) interrompent l'exécution du processus sans le tuer.*

Listing 2 – exo_1_2.c

```c
/* exo_1_2.c: mycat */

#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

/* On definit une constante */
#define STEP 1

void mycat(int fd)
{
  char                  buf[STEP];
  int                   r;
  while ( (r = read(fd, buf, STEP)) )
    {
      if (r == -1)
        {
          if (errno == EINTR)
            continue;
          perror("mycat, read error");
          exit(2);
        }
      write(STDOUT_FILENO, buf, r);
    }
}

int main(int argc, char *argv[])
{
  int                   fd;
  if (argc < 2)
    {
      write(STDERR_FILENO, "mycat: parametre manquant.\n",27);
      exit(1);
    }

  /* gestion du paramètre "-" */
  /* FIXME                    */

  if ((fd = open(argv[1],O_RDONLY)) == -1)
    {
      perror("mycat, open");
      exit(2);
    }
  mycat(fd);
  close(fd);
  return 0;
}
```

2 Déplacement dans un fichier

On va maintenant étudier les déplacements dans les fichiers : ceux induits par l'usage de `read(2)` et `write(2)` et ceux provoqués par l'usage de `lseek(2)`.

On commence par créer un nouveau fichier contenant l'alphabet complet en majuscule puis en minuscule, ainsi on pourra repérer facilement notre position dans le fichier : l'octet n correspondra à la *n-ième* lettre de l'alphabet majuscule pour $n < 27$ et l'octet $n + 27$ correspondra à la *n-ième* lettre de l'alphabet minuscule. Par exemple la lettre `b` correspondra à l'octet 28.

Le listing 3 présente le programme permettant de créer un tel fichier. Dans ce programme on utilise l'appel système `open(2)` pour ouvrir le fichier en écriture (`O_WRONLY`) avec création si le fichier n'existe pas (`O_CREAT`). On notera au passage l'usage du drapeau `O_TRUNC` indiquant que l'on doit écraser complètement le fichier si celui-ci existe. On indique explicitement les permissions pour la création (valeur octale `0666`.) Concernant les permissions on notera que la valeur `0666` n'est pas utilisée directement par le système, elle est *masquée* par le masque par défaut (normalement `0022`, voir `umask(2)`) donnant au final une valeur de `0644`.

Listing 3 – `makefic.c`

```
1  /* makefic.c */
2
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  int main()
9  {
10   int fd,i;
11   char buf[52];
12   /* ouverture du fichier */
13   fd = open("ficref",O_WRONLY|O_CREAT|O_TRUNC,0666);
14   /* Si l'ouverture echoue */
15   if (fd<0)
16     {
17       perror("open failed ");
18       return(2);
19     }
20   /* Remplissage de buf */
21   for (i=0;i<26;i++)
22       buf[i] = 'A' + i;
23   for (i=0;i<26;i++)
24       buf[i+26] = 'a' + i;
25   /* Ecriture dans le fichier */
26   write(fd,buf,52);
27   close(fd);
28   return(0);
29 }
```

Exercice 2.1:
Réécrie le programme du listing 3 en utilisant un seul caractère à la place du buffer et en effectuant une écriture par caractère. Comparer les deux versions.

Exercice 2.2:
Modifier le programme du listing 3 pour qu'il prenne le nom du fichier en paramètre sur sa ligne de commande. Votre programme devra utiliser un nom par défaut si aucun nom n'est fournit.

Exercice 2.3:
On va maintenant observer le déplacement implicite dans un fichier. Vous devez écrire un programme ouvrant le fichier créé par le programme précédant en lecture et en écriture (mais pas en création et surtout pas en écrasement.)

Votre programme devra effectuer une lecture de 5 octets (en les affichant), puis une écriture de 1 octet (écrire le caractère `'_'`) et enfin une autre lecture de 5 octets.

Où se trouve le caractère `'_'` dans le fichier après terminaison de notre programme ?

Remarque 2.1:
Pensez à régénérer le fichie de référence (celui avec l'alphabet) à chaque fois, vu que celui-ci va être modifié.

On va maintenant effectuer des déplacements explicites à l'aide de l'appel système `lseek(2)`. La page de manuel de `lseek(2)` nous donne le prototype suivant :

```
off_t lseek(int fildes, off_t offset, int whence);
```

Un peu de recherche nous permet de découvrir que le type `off_t` correspond aux entiers.

Enfin le paramètre `whence` indique comme l'offset doit être interpréter :

`SEEK_SET` : le décalage s'effectue depuis le début du fichier.

`SEEK_CUR` : le décalage s'effectue depuis la position courante dans le fichier.

`SEEK_END` : le décalage s'effectue depuis la fin du fichier.

Exercice 2.4:
Soit le programme du listing 4. Quels sont les caractères affichés par ce programme ? Comparer avec le résultat du programme.

Listing 4 – exo_2_4.c

```c
/* exo_2_4.c */
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

void checkread(int r){
  if (r<0) {
    perror("read ");
    exit(2);
  }
}

int main()
{
  int fd,i;
  char buf[10];
  fd = open("ficref",O_RDONLY);
  /* Si l'ouverture echoue */
  if (fd<0)
    {
      perror("open failed ");
      return(2);
    }
  i = read(fd,buf,10);
  checkread(i);
  write(STDOUT_FILENO,"READ de 10\n",11);
  lseek(fd,5,SEEK_SET);
  i = read(fd,buf,1);
  checkread(i);
  write(STDOUT_FILENO,"LSEEK de 5 avec SEEK_SET: ",25);
  write(STDOUT_FILENO,buf,1);
  write(STDOUT_FILENO,"\n",1);
  i = read(fd,buf,10);
  checkread(i);
  write(STDOUT_FILENO,"READ de 10\n",11);
  lseek(fd,5,SEEK_CUR);
  i = read(fd,buf,1);
  checkread(i);
  write(STDOUT_FILENO,"LSEEK de 5 avec SEEK_CUR: ",25);
  write(STDOUT_FILENO,buf,1);
  write(STDOUT_FILENO,"\n",1);
  lseek(fd,0,SEEK_SET);
  i = read(fd,buf,1);
  checkread(i);
  write(STDOUT_FILENO,"LSEEK de 0 avec SEEK_SET: ",25);
  write(STDOUT_FILENO,buf,1);
  write(STDOUT_FILENO,"\n",1);
  lseek(fd,-1,SEEK_END);
  i = read(fd,buf,1);
  checkread(i);
  write(STDOUT_FILENO,"LSEEK de 0 avec SEEK_END: ",25);
  write(STDOUT_FILENO,buf,1);
  write(STDOUT_FILENO,"\n",1);
  close(fd);
  return(0);
}
```

3 Accès concurrents

On va maintenant observer le comportement des descripteurs de fichier partagés entre deux processus. Pour ça, il nous faut utiliser `fork(2)` pour engendrer un nouveau processus possédant les mêmes ressources.

Listing 5 – `exo_3_1.c`

```c
/* exo_3_1.c */

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

int main()
{
  int i=0;

  if (fork())
    {
      /* dans le pere */
      printf("pere: %i\n",i);
    }
  else
    {
      /* dans le fils */
      i++;
      printf("fils: %i\n",i);
    }
  /* Dans les 2 */
  printf("tous: %i\n",i);
  return(0);
}
```

Exercice 3.1 (`fork(2)`):
Ce premier programme a pour objectif de tester l'appel système `fork(2)`. Que donne la sortie du programme du listing 5 ?

Exercice 3.2:
Ce programme a pour objectif de tester les valeurs de retour de l'appel système `fork(2)`. Que donne la sortie du programme du listing 6 ?

Exercice 3.3:
On va maintenant tester les interférences entre 2 programmes partageant un même descripteur de fichier.. Que donne la sortie du programme du listing 7 ?

Listing 6 – exo_3_2.c

```
1  /* exo_3_2.c */
2
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  int main()
9  {
10   int i;
11
12   if ((i=fork()))
13     {
14       /* dans le pere */
15       printf("pere: %i\n",i);
16     }
17   else
18     {
19       /* dans le fils */
20       printf("fils: %i\n",i);
21     }
22   /* Dans les 2 */
23   return(0);
24 }
```

Remarque 3.1:
Dans le listing 7, on a utilisé quelques petits trucs *:*

- *Ce programme utilise les fonctions de la bibliothèque standard pour manipuler les chaînes de caractères (`strncpy(3)` et `strlen(3)`.)*
- *Le nom du fichier a ouvrir est soit obtenu par la ligne de commande, soit un nom par défaut (lignes 15 à 19).*
- *Un on a construit un message d'erreur plus explicite pour le cas d'un echec d'`open(2)`. Le message est créé avant l'appel de `open(2)`, dans le cas où l'une des fonctions utilisée affecterait la variable `errno`.*
- *On mélange explicitement l'utilisation de pointeur et de tableau : à la ligne 41 on utilise le pointeur `buf` décalé de 7 octets pour laisser la chaîne copiée à la ligne 40 (idem pour la ligne 49.)*

Listing 7 – exo_3_3.c

```c
/* exo_3_3.c */

#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  int fd,i;
  char *nom, *msg;
  char buf[32];

  /* Choix du nom du fichier */
  if (argc<2)
    nom = "ficref";
  else
    nom = argv[1];

  /* message d'erreur */
  i = strlen(nom);
  msg = malloc(i + 6);
  strncpy(msg,"open ",5);
  strncpy(msg+5,nom,i);
  msg[5 + i] = 0;
  /* ouverture */
  fd = open(nom,O_RDONLY);
  if (fd<0)
    {
      perror(msg);
      exit(2);
    }
  /* msg inutil */
  free(msg);

  if (fork())
    {
      /* dans le pere */
      strncpy(buf,"pere : ",7);
      i = read(fd,buf+7,5);
      buf[12] = '\n';
      write(STDOUT_FILENO,buf,13);
    }
  else
    {
      /* dans le fils */
      strncpy(buf,"fils : ",7);
      i = read(fd,buf+7,5);
      buf[12] = '\n';
      write(STDOUT_FILENO,buf,13);
    }
  /* Dans les 2 */
  return(0);
}
```

4 Fichiers structurés et gros fichiers

Jusqu'à présent, nous n'avons manipulé que des fichiers contenant du texte brut. On va maintenant s'intéresser à l'accès aux fichiers structurés, et plus particulièrement aux en-têtes de ces fichiers. Enfin, nous verrons comment lire des fichiers de grande taille *directement*.

4.1 Le format BMP

Nous allons prendre le format BMP comme référence pour cette exemple. Un fichier BMP est composé de deux en-tête suivis des données de l'image. Nous simplifierons notre approche pour nous intéresser à des fichiers sans compression codés en 24bits.
Le premier en-tête se décompose sous la forme suivante :

Offset	Taille	Usage
0	2	Le *magic number* BMP : 0x42 0x4D (code ASCII pour B et M.)
2	4	La taille du fichier en octet.
6	2	réservé ; souvent utilisé par l'application créatrice de l'image.
8	2	réservé ; souvent utilisé par l'application créatrice de l'image.
10	4	offset : l'adresse de départ des données de l'image.

Le second en-tête détaile les informations de l'image. Il existe plusieurs version de l'en-tête, nous nous concentrerons sur la version 3 qui semble la plus répandue :

Offset	Taille	Usage
14	4	Taille de l'en-tête (40 octets)
18	4	Largeur de l'image en pixels (signed integer).
22	4	Hauteur de l'image en pixels (signed integer).
26	2	Le nombre de plan de couleur utilisé. Doit être à 1.
28	2	Le nombre de bits par pixel. En général : 1, 4, 8, 16, 24 ou 32.
30	4	La méthode de compression utilisée. Toujours 0 dans notre cas.
34	4	La taille des données de l'image (attention ne pas confondre avec la taille du fichier.)
38	4	La résoltion horizontal. (en mètre par pixel, signed integer)
42	4	La résoltion vertical. (en mètre par pixel, signed integer)
46	4	Le nombre de couleur dans la palette, ou 0 pour la valeur par défaut (2^n).
50	4	Le nombre de couleur importante, ou 0 si elles le sont toutes (généralement ignoré.)

On va sauver ces deux en-têtes dans deux structures dont les champs correspondront exactement (en taille et en position) avec ceux des en-têtes. Le premier en-tête n'étant pas aligné (les structures C ont une taille aligné sur la taille des entiers, 4 octets dans notre cas), nous allons lui rajouter un champ de 2 octets au début.

Nous allons sauver ces deux structures dans un fichier `bmpheader.h` qui sera inclus avec par reste de notre code. Le listing 8 présente ce fichier.

Exercice 4.1:
Dans un premier temps nous allons lire les en-têtes et remplir les structures champs par champs. Écrire un programme qui prend en argument le nom d'un fichier au format BMP (seul le format compte, l'extension n'a pas d'incidence sur le chargement) et remplit les deux structures. Votre programme devra inclure le fichier `bmpheader.h`.

Listing 8 – bmpheader.h

```c
/* BMP header */

struct s_bmp_image_header {
  unsigned int                header_size;
  unsigned int                width;
  unsigned int                height;
  short int                   planes;
  short int                   bit;
  unsigned int                comp_method;
  unsigned int                image_size;
  int                         hres;
  int                         vres;
  unsigned int                color_number;
  unsigned int                important_color;
};

struct s_bmp_file_header {
  short                       padding;
  char                        magic[2];
  unsigned int                file_size;
  unsigned int                reserved;
  unsigned int                offset;
  struct s_bmp_image_header   dib;
};

void print_bmp_info(struct s_bmp_file_header *h);
```

Remarque 4.1:
Le fichier `bmpheader.h` *contient la déclaration d'une fonction* `print_bmp_info` *chargée d'afficher les informations des en-têtes. Vous devez fournir l'implantation de cette fonction.*

Exercice 4.2:
Comme nos structures sont bien alignées et suivent l'organisation des en-têtes, il serait plus simple de les remplir en une seule passe. La lecture des en-tête devra se faire avec un seul `read(2)`*. Attention, le premier champ de la première structure ne fait pas partie de l'en-tête.*

Écrire un programme qui prend en argument le nom d'un fichier au format BMP (seul le format compte, l'extension n'a pas d'incidence sur le chargement) et remplit les deux structures. Votre programme devra inclure le fichier `bmpheader.h`*.*

Remarque 4.2:
Pour remplir la structure depuis le second champ il va falloir faire une addition sur le pointeur, malheureusement les règles de l'arithmétique des pointeurs ne nous permettent pas de faire une addition de 2 octets sur un type autre que `char` *(ou assimilé). Nous allons pallier à ce problème en utilisant une* ***union****.*

Une ***union*** *est la construction duale des structures. Une union ne possède qu'une seule valeur, mais dispose de plusieurs façon d'y accéder (plusieurs types.) On peut assimiler les unions de C au types sommes d'OCaml (le contrôle de type en moins.)*

Grâce à cette union, nous pouvons manipuler notre pointeur à la fois comme un pointeur sur structure et un pointeur sur caractères et du coup pouvoir faire nos opération à l'octet prêt.

Le listing 9 présente notre ***union****. On déclarera une variable de ce type comme indiqué à la ligne 6. Enfin, l'expression* `h.header` *donne accès au pointeur avec le type* `struct s_bmp_file_header*` *et l'expression* `h.c` *au même pointeur avec le type* `char*`*.*

Listing 9 – Une union

```
1 union u_header {
2   struct s_bmp_file_header *header;
3   char                     *c;
4 };
5
6 union u_header h;
```

4.2 Fichiers de grande taille

Lorsqu'un fichier est de taille importante, il devient coûteux de lire son contenu bloc par bloc (nombreux `read(2)` et nombreux tours de boucles.) Le système offre un moyen simple d'associer en mémoire le contenu d'un fichier en seule fois : `mmap(2)`.

L'appel système `mmap(2)` permet d'associer un descripteur de fichier à un certain nombre de pages (en fonction de la demande.) Il propose différentes options (lecture, écriture, exécution, partage avec d'autre processus ...) Nous allons l'utiliser simplement pour charger en mémoire notre fichier BMP, convertit celui-ci en niveau de gris (avec la méthode la plus simple) et sauver le résultat sur disque (attention, en écrasant directement l'original.)

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

`addr` : `mmap(2)` va essayer d'allouer la page *aux allentours* de cette adresse.

`len` : la taille dont on a besoin.

`prot` : les protections

- **`PROT_NONE`** : aucun accès ;
- **`PROT_READ`** : accès en lecture ;
- **`PROT_WRITE`** : accès en écriture ;
- **`PROT_EXEC`** : accès en exécution.

`flags` : options de `mmap(2)`

- **`MAP_ANON`** : `mmap(2)` anonyme (sans fichier associé) ;
- **`MAP_FIXED`** : utilisation d'`addr` ou échec ;
- **`MAP_PRIVATE`** : les pages allouées sont privées (pas de partage) ;
- **`MAP_SHARED`** : les pages allouées peuvent être partagées avec d'autre processus.

`fd` : le descripteur de fichier associé.

`offset` : décalage dans le fichier (attention, aligné sur la taille d'une page.)

Nous allons suivre le cheminement suivant (on suppose toujours que l'on travaille avec un BMP 24bits au format décrit plus haut) :

1. Ouverture du fichier ;
2. Récupération de la taille du fichier dans l'en-tête ;

3. Chargement du fichier via `mmap(2)` ;
4. Conversion en niveau de gris pixel par pixel ;
5. Libération des pages et fermeture du fichier.

Comme le fichier est *mappé* en mémoire, il n'est pas nécessaire *d'écrire* dans le fichier, la synchronisation entre la mémoire et le fichier se fait d'elle même.

Nous allons commencer par modifier la structure de l'en-tête de fichier pour disposer d'un point d'accès sur les données. Comme les donnés se trouvent directement en mémoire à la suite de l'en-tête (comme dans le fichier) le dernier champ (un tableau) fournit un pointeur sur caractères pour la zone de données. La taille de ce champ importe peu (la taille des tableaux dans une structure n'est là que pour `sizeof`.) Le listing 10 présente la nouvelle version de ces structures.

Listing 10 – `bmpheader-v2.h`

```
1  /* BMP header v2 */
2
3  struct s_bmp_image_header {
4    unsigned int                header_size;
5    unsigned int                width;
6    unsigned int                height;
7    short int                   planes;
8    short int                   bit;
9    unsigned int                comp_method;
10   unsigned int                image_size;
11   int                         hres;
12   int                         vres;
13   unsigned int                color_number;
14   unsigned int                important_color;
15 };
16
17 struct s_bmp_file_header {
18   short                       padding;
19   char                        magic[2];
20   unsigned int                file_size;
21   unsigned int                reserved;
22   unsigned int                offset;
23   struct s_bmp_image_header   dib;
24   unsigned char               data[4];
25 };
```

Le reste est relativement direct, on s'assurera juste que le pointeur sur la structure est correctement décalé par rapport au champ d'alignement. Le listing 11 présente le code à compléter. Après l'ouverture du fichier, vous devez lire sa taille dans l'en-tête (on pourrait aussi l'obtenir avec un appel à `lstat(2)` puis on *map* le fichier (en ayant ramener le pointeur au début de fichier.) Le fichier a été ouvert en lecture écriture (`O_RDWR`) et les pages obtenues ont les protections correspondantes : `PROT_READ|PROT_WRITE`. Enfin, vu que l'on veut modifier le fichier en place on le positionne en mode partage avec l'option `MAP_SHARED`.

Vous devez compléter la partie indiquée par un `FIX ME`.

La conversion en niveau de gris est un grand classique (moyenne pondérée et double boucle sur l'image.) Enfin, vu que l'écriture se fait directement sur la page, il suffit de fermer le fichier et de libérer les pages pour que nos modifications soient reportées dans le fichier d'origine.

Vous devez également implanter la fonction `to_grey` qui modifie en place l'image en effectuant une moyenne pondérée des composantes de chaque pixel suivant la formule : $0.11 \times r + 0.59 \times g + 0.3 \times b$**.**

Listing 11 – bmploader.c

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include "bmpheader-v2.h"
#define  align4(i) (((((i)-1)>>2)<<2) + 4) /* Alignement sur 4 */

union u_header {
  struct s_bmp_file_header *header;
  char                     *c;
};

/* passage en niveau de gris */
void to_grey(struct s_bmp_file_header *bmp)
{
  unsigned char        *data;
  unsigned int          i,j,k,pad;
  unsigned char         grey;

  /* FIX ME */

}

int main(int argc, char **argv)
{
  union u_header        h;
  int                   fd;
  unsigned int          fsize;
  if (argc<2)
    {
      write(STDERR_FILENO,"usage: exo_4_1 <bmp file>\n",26);
      return(1);
    }
  fd = open(argv[1],O_RDWR);
  if (fd<0)
    {
      perror("open ");
      return(2);
    }

  /* FIX ME: lecture de la taille et mapping du fichier */
  /* en memoire.                                        */

  to_grey(h.header);
  close(fd);

  munmap(h.c + 2, fsize);
  return(0);
}
```

5 Processus

Pour finir ce tour, nous allons revenir sur la manipulation de processus (nous avons déjà vu rapidement des exemples de `fork(2)`.)

5.1 Duplication

On va déjà vu des exemples de base sur `fork(2)`, on va maintenant voir le comportement du père vis à vis de la mort de son fils. Pour ce faire on va utiliser `wait(2)`.

Exercice 5.1:
Reprendre l'exercice 3.1, et modifier le code du père (dans le `if`) pour que le processus père attende la mort de son fils avant de faire ses affichages.

En lisant la page de manuel de `wait(2)`, on peut trouver les noms des macros permettant de manipuler les informations sur la mort du fils.

Exercice 5.2:
Le listing 12 présente un utilisation de base de ces macros. Comment peut-on obtenir l'affichage de la ligne 22 ?

Exercice 5.3:
Modifier le listing 12 pour que l'on puisse simplement envoyer un signal (à l'aide de la commande `kill(1)`) au fils et voir s'afficher le message concernant la mort par signaux.

5.2 Remplacement/Recouvrement

Nous allons maintenant nous intéresser au remplacement d'un processus par un autre programme. Nous nous concentrerons sur la fonction `execvp(3)`.

Exercice 5.4:
Écrire un programme qui se remplace par le programme `ls(1)` avec l'option `-l`.

Exercice 5.5:
Généraliser l'exercice précédant en récupérant le nom du programme à exécuter et ses arguments sur la ligne de commande de votre programme. Vous devez (pour cet exercice) reconstruire complètement le tableau d'arguments que vous passez à `execvp(3)`.

Listing 12 – exo_5_2.c

```c
/* exo_5_2.c */

#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int main()
{
  int status, fpid;

  if ((fpid = fork()))
    {
      /* dans le pere */
      wait(&status);
      printf("pere:\n");
      if (WIFEXITED(status))
        printf("%i est sortie avec %i\n",fpid,WEXITSTATUS(status));
      else
        if (WIFSIGNALED(status))
          printf("%i a recu le signal %i\n",fpid,WTERMSIG(status));
    }
  else
    {
      /* dans le fils */
      printf("fils\n");
    }
  return(0);
}
```

Exercice 5.6 (Exécution *à la* shell):
On va maintenant combiner `fork(2)` et `execvp(3)`. Reprendre le programme de l'exercice précédant, sauf que vous effectuerez le remplacement, après une duplication, dans le fils. Le père devra attendre la fin de l'exécution du fils et afficher les informations de terminaison (valeur de sortie ou signal de fin) du fils.

5.3 Redirections

On va maintenant simuler les redirections du shell. On se contentera des redirections depuis ou vers un fichier. On rappelle l'usage de la fonction `dup2(2)` :

```
int dup2(int oldd, int newd);
```

`dup2(f1,f2)` fait pointer le descripteur `f2` sur la même ressource que `f1`, après avoir préalablement fermé `f2` s'il y a lieu.

Exercice 5.7:
On va commencer par rediriger la sortie standard sur un fichier dont le nom est fournit en argument. Le listing 13 propose un exemple de programme dont la sortie est redirigée (notez l'utilisation de `getpid(2)`, `getuid(2)` et `geteuid(2)`.)

Listing 13 – `exo_5_7.c`

```c
/* exo_5_7.c */

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

int main(int argc, char **argv)
{
  int fd,i;

  if (argc<2)
    {
      write(STDERR_FILENO, "usage: exo_5_7 <fichier>\n",25);
      return(1);
    }
  fd = open(argv[1],O_WRONLY|O_CREAT|O_TRUNC,0666);
  if (fd<0)
    {
      perror("open failed ");
      return(2);
    }

  printf("Le fichier %s a ete ouvert normalement\n",argv[1]);

  i = dup2(fd,STDOUT_FILENO);
  if (i<0)
    {
      perror("dup2");
      return(2);
    }

  /* les ecritures auront maintenant lieu dans le fichier ouvert */
  printf("Les ecritures ont lieu dans le fichier %s.\n", argv[1]);
  printf("%s: pid=%i uid=%i euid=%i\n",
         argv[0], getpid(), getuid(), geteuid());
  printf("STDOUT_FILENO: %i, fd: %i\n",STDOUT_FILENO,fd);
  close(fd);
  printf("Le fichier est toujours ouvert\n");

  return(0);
}
```

Exercice 5.8:
Reprendre le code de l'exercice 5.6 et rediriger la sortie de la commande appelée dans un fichier que vous nomerez `output` *et qui sera écrasé (s'il existe) à chaque appel. La redirection ne doit concerner que le fils (et le programme appelé.)*