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

Examen (1h30)

Les documents, calculatrices, téléphones etc. ne sont pas autorisés.
Le sujet comporte 6 pages
Une documentation sur les appels système d'entrée/sortie se trouve en fin de sujet.
Remarque : Vous pouvez négliger les erreurs qu'il n'est pas explicitement demandé de traiter.

Exercice 1. *Conversion de fichier audio*

A. Présentation du format WAV

On s'intéresse dans cet exercice aux fichiers audio au format WAV codés en *modulation par impulsion et codage* ou *Pulse Code Modulation* (PCM), permettant d'encoder un son en enregistrant les valeurs de l'intensité du signal (onde sonore) à intervalles de temps réguliers.

Les fichiers audio peuvent contenir un ou plusieurs canaux qui correspondent au nombre de sources servant à reproduire le son. Chaque canal est constitué d'une suite de valeurs numériques (les *échantillons*) permettant de reconstruire l'onde sonore à laquelle il correspond.

Dans cet exercice on ne considérera que des fichiers *mono* (1 canal) ou *stereo* (2 canaux), et on supposera que tous les échantillons sont représentés sur 16 bits (2 octets).

Les fichiers WAV sont constitués d'un entête donnant des informations sur le codage utilisé (format, nombre de canaux, nombre d'échantillons par seconde, etc.), suivi des octets de données encodant les valeurs des échantillons de tous les canaux.

L'entête d'un fichier WAV commence dès le premier octet (*offset* 0). Il a une taille de 44 octets, et est constitué des champs suivants listés dans l'ordre [1] (il n'est pas nécessaire de comprendre la signification de tous les champs pour le moment) :

```
[Bloc de déclaration d'un fichier au format WAVE]
   FileTypeBlocID  (4 octets) : Constante «RIFF»  (0x52,0x49,0x46,0x46)
   FileSize        (4 octets) : Taille du fichier moins 8 octets
   FileFormatID    (4 octets) : Format = «WAVE»  (0x57,0x41,0x56,0x45)

[Bloc décrivant le format audio]
   FormatBlocID    (4 octets) : Identifiant «fmt »  (0x66,0x6D, 0x74,0x20)
   BlocSize        (4 octets) : Nombre d'octets du bloc - 16  (0x10)
   AudioFormat     (2 octets) : Format du stockage dans le fichier (1: PCM, ...)
   NbrCanaux       (2 octets) : Nombre de canaux (de 1 à 6)
   Frequence       (4 octets) : Fréquence d'échantillonnage (en hertz) [Valeurs
      standardisées : 11 025, 22 050, 44 100 et éventuellement 48 000 et 96 000]
   BytePerSec      (4 octets) : Nombre d'octets à lire par seconde (c.-à-d.,
   Frequence * BytePerBloc).
   BytePerBlock    (2 octets) : Nombre d'octets par bloc d'échantillonnage
      (c.-à-d., tous canaux confondus : NbrCanaux * BitsPerSample/8).
   BitsPerSample   (2 octets) : Nombre de bits utilisés pour le codage de
      chaque échantillon (8, 16, 24)

[Bloc des données]
   DataBlocID      (4 octets) : Constante «data»  (0x64,0x61,0x74,0x61)
   DataSize        (4 octets) : Nombre d'octets des données (c.-à-d.
      taille_du_fichier - taille_de_l'entête  (qui fait 44 octets normalement).
```

Après l'entête, le fichier contient les octets de données représentant les échantillons des différents canaux en alternance (voir figures 1 et 2).

1. source : `https://fr.wikipedia.org/wiki/Waveform_Audio_File_Format`

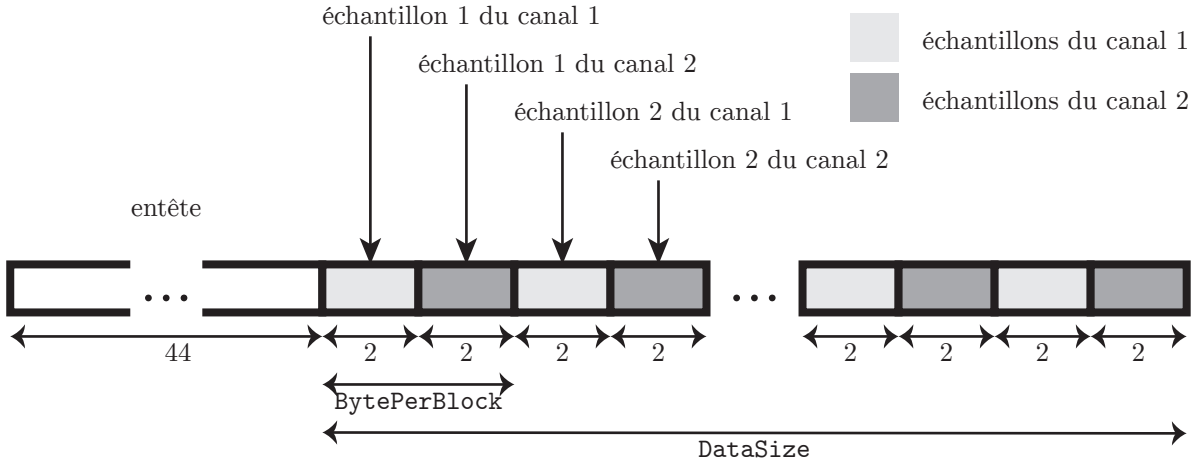

FIGURE 1 – Contenu d'un fichier WAV correspondant à un enregistrement sur 2 canaux (stereo) avec des échantillons sur 16 bits (2 octets).

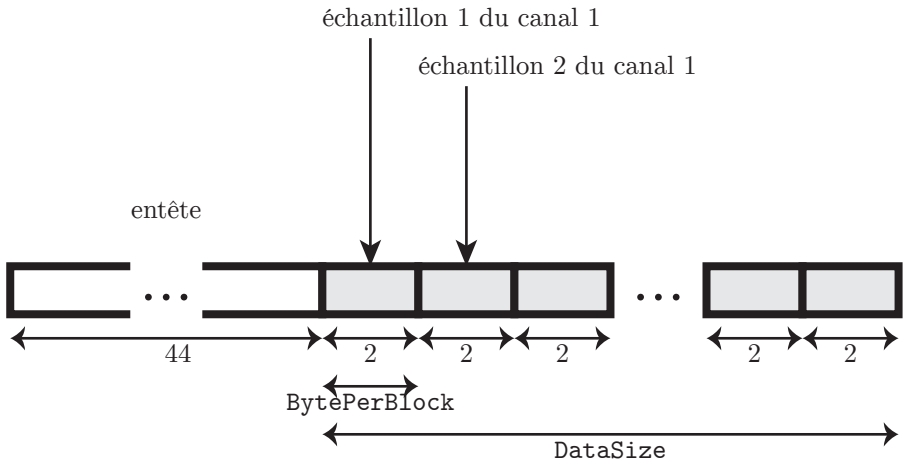

FIGURE 2 – Contenu d'un fichier WAV correspondant à un enregistrement sur 1 canal (mono) avec des échantillons sur 16 bits (2 octets).

B. Organisation du programme

Le but de l'exercice est d'écrire un programme qui prend en entrée un fichier WAV à deux canaux (stereo) et le convertit en un fichier à un seul canal (mono) en ne conservant que le premier canal.

Le programme devra donc exécuter les actions suivantes :

— ouvrir le fichier contenant les données d'entrée (en lecture) et le fichier où écrire la sortie (en écriture) ;
— lire l'entête du fichier en entrée, modifier les valeurs qui changent lorsque le fichier est converti en mono et écrire le nouvel entête dans le fichier en sortie ;
— lire les données du fichier en entrée et ne recopier dans le fichier en sortie que les octets correspondant aux échantillons du premier canal.

On part du squelette de programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fdin, fdout; // descripteurs de fichiers en entrée et en sortie
  char buffer[4096];  // buffer utilisée pour les lectures/écritures
  int nbread; // nombre d'octets lus après chaque appel à read


  // [1] Ouverture des fichiers en entrée et sortie
  // le nom du fichier à lire est passé en premier argument du programme, le
  // nom du fichier dans lequel écrire le résultat est passé en second
  // argument


  // [2] Lecture de l'entête du fichier en entrée, modification des données
  // nécessaires dans le buffer et écriture de l'entête modifié dans le
  // fichier en sortie


  // [3] Lecture des données du fichier en entrée (par blocs de 4096 octets)
  // et écriture dans le fichier en sortie des octets correspondant aux
  // échantillons du premier canal.


  close(fdin);
  close(fdout);
}
```

1 Écrivez les instructions permettant d'ouvrir en lecture seule le fichier dont le nom est passé en premier argument du programme (le descripteur de fichier est placé dans la variable `fdin`), et d'ouvrir en écriture seule le fichier dont le nom est passé en second argument (dans la variable `fdout`). On pourrait par exemple appeler le programme à l'aide de la commande « `./a.out entree.wav sortie.wav` »

— Si le fichier en écriture n'existe pas il faut le créer et s'il existe il faut le réinitialiser (effacer son contenu).
— Si l'ouverture d'un des fichiers ne s'effectue pas correctement, le programme doit afficher un message d'erreur approprié et terminer son exécution.

2 Donnez l'instruction permettant de lire les octets de l'entête du fichier en entrée pour les placer dans le tableau `buffer`. On rappelle que la longueur de l'entête est toujours 44 octets.

Il faut maintenant modifier les informations suivantes dans l'entête du fichier (qui se trouve dans le tableau `buffer`) :

— `FileSize` (octets 4-7) : cet `int` représente la taille du fichier moins 8 octets (c'est-à-dire le nombre d'octets restant dans le fichier après cette valeur). Dans le fichier mono en sortie, l'entête a la même taille (44 octets) mais il y a deux fois moins d'octets de données.

— `NbrCanaux` (octets 22-23) : ce `short int` (entier codé sur deux octets) contient la valeur 2 dans le fichier en entrée mais doit contenir 1 dans le fichier en sortie.
— `BytePerSec` (octets 28-31) : cet `int` doit être divisé par 2 pour obtenir la valeur correspondant à un fichier mono.
— `BytePerBlock` (octets 32-33) : ce `short int` doit également être divisé par 2.
— `DataSize` (octets 40-43) : cet `int` doit être divisé par 2.

3 Écrivez les instructions permettant de modifier le contenu du tableau `buffer` contenant initialement l'entête du fichier en entrée pour qu'il contienne l'entête du fichier en sortie.

Indications : On suppose ici que les `int` sont stockés sur 4 octets et que les `short int` sont stockés sur 2 octets. On suppose également que l'ordre des octets dans la représentation d'un entier (`int` ou `short int`) du système est le même que celui utilisé dans le format WAV (*little-endian*). Avec ces hypothèses, on peut manipuler les valeurs dans le tableau `buffer` de la manière suivante :

```
int *p;
p = (int *) (buffer + 14);
// p est un pointeur vers un entier dont la représentation commence à la
// case 14 du tableau buffer

*p = *p + 42;
// l'entier dont la représentation commence à la case 14 du tableau buffer
// a été incrémenté de 42 (les valeurs dans le tableau ont été modifiées)
```

4 Écrivez l'instruction permettant d'écrire le nouvel entête (qui se trouve dans le tableau `buffer`) dans le fichier en sortie.

5 Écrivez la boucle permettant de copier tous les octets du fichier en entrée correspondant aux échantillons du premier canal dans le fichier en sortie.

Indications : Il faut lire les octets du fichier en entrée dans le tableau `buffer` par blocs de (au plus) 4096 octets, puis déplacer les octets correspondant aux échantillons du premier canal pour les grouper au début du tableau `buffer` (voir figure 3). Il faut enfin recopier le début du tableau `buffer` (correspondant aux octets du premier canal qui ont été déplacés) dans le fichier en sortie.

La boucle doit être interrompue lorsque tous les octets du fichier en entrée ont été lus.

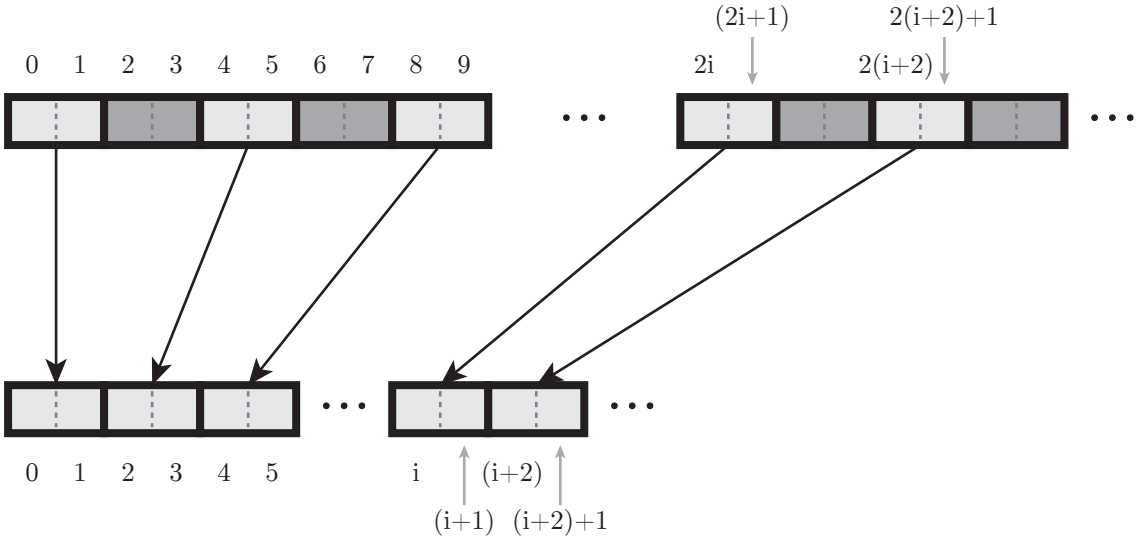

FIGURE 3 – Déplacement des octets correspondant aux échantillons du premier canal d'un fichier stereo (en gris clair en haut) dans le début du tableau pour correspondre à l'encodage d'un fichier mono (en bas).

Exercice 2. *Parallélisme*

On suppose que l'on dispose d'un executable `f` qui prend en argument un entier (ou plus exactement une chaîne de caractères correspondant à l'écriture en base 10 d'un entier), effectue un calcul que l'on suppose très long, et renvoie un entier via le `return` de sa fonction `main`. On peut ainsi par exemple exécuter `f` à l'aide de la commande « `./f 4` » qui exécute `f` avec le paramètre 4.

Le but de cet exercice est d'écrire un programme qui, étant donné deux entiers i et j, calcule la valeur $f(i) + f(j)$. Pour cela, on veut calculer les valeurs $f(i)$ et $f(j)$ dans des processus différents, récupérer les valeurs retournées par ces deux processus, et en faire la somme.

L'intérêt principal d'une exécution parallèle dans deux processus distincts est que, si l'on dispose d'une machine multicœurs, on espère aller deux fois plus vite que si le même processus avait calculé successivement $f(i)$ puis $f(j)$.

On souhaite donc écrire un programme ayant les spécifications suivantes :

— Le programme attend 2 arguments entiers encodant les valeurs i et j (par exemple « `./a.out 4 5` »).
— On suppose qu'il existe (dans le même dossier que le programme) un executable nommé `f` qui prend un entier en paramètre, et retourne un entier.
— Le programme calcule puis affiche la valeur $f(x) + f(y)$. Les 2 appels à `f` sont effectués dans des processus differents pour permettre la parallélisation du calcul.

On supposera que les appels systemes (`fork`, `execlp`) s'exécutent sans erreurs. Les spécifcations de `fork` et `execlp` sont rappelées en annexe.

1 Expliquez pourquoi il est nécessaire d'utiliser au moins 3 processus au total pour effectuer la tâche décrite. En particulier, expliquez pourquoi la structure de programme suivante ne peut pas être réalisée :
— le processus initial (père) crée un nouveau processus (fils) ;
— le père exécute le calcul $f(x)$ à l'aide d'un appel à `execlp` ;
— le fils exécute le calcul $f(y)$ à l'aide d'un appel à `execlp` ;
— le père récupère le résultat du fils et affiche le résultat de la somme $f(x) + f(y)$.

On décide donc d'utiliser 3 processus :

— le processus initial (père) crée successivement deux fils à l'aide de la fonction `fork` ;
— le premier fils exécute le calcul $f(x)$;
— le second fils exécute le calcul $f(y)$;
— le père récupère les deux résultats et affiche la valeur $f(x) + f(y)$.

On considère le code suivant :

```
1  int main(int argc, char **argv) {
2    int pid;
3    pid = fork();
4    if ( ... ) {
5      ...
6    }
7    // suite du programme (complétée plus tard)
8  }
```

2 Complétez le code ci-dessus (lignes 4 et 5) pour que le premier fils exécute le code de l'exécutable `f` (qu'on suppose être dans le même répertoire que le programme) avec pour argument le premier argument passé au programme.

3 En supposant que l'instruction `fork` (ligne 3) et l'instruction de la question précédente (ligne 5) se passent bien, combien de processus vont exécuter les instructions qui seront ajoutées après le bloc `if` (à partir de la ligne 7) ? Justifiez brièvement votre réponse.

4 Donnez les instructions à ajouter au programme (à partir de la ligne 7) pour qu'il crée un second fils qui exécute le contenu de l'exécutable `f` avec pour argument le second argument passé au programme.

5 En utilisant les fonctions `wait(int *status)` et la macro `WEXITSTATUS(status)`, complétez le programme pour que le père affiche à l'écran la somme des deux valeurs renvoyées par les fils (leur code de retour).

Rappels : La fonction `wait(int *status)` est bloquante et attend qu'un des fils du processus courant termine. Lorsqu'un fils termine, un entier contenant des informations sur l'état du processus terminé est écrit à l'adresse passée en argument.
Étant donné un entier `status` contenant les informations d'un processus terminé (comme celui qui a été écrit par la fonction `wait`), la macro `WEXITSTATUS(status)` renvoie le code de retour du processus (résultat de la fonction `main`).

Documentation abrégée

`OPEN` - Ouvrir ou créer éventuellement un fichier ou un périphérique

```
int open(const char *pathname, int flags); int open(const char *pathname, int flags, mode_t mode);
```

Étant donné le chemin `pathname` d'un fichier, `open` renvoie un descripteur de fichier, un petit entier positif ou nul utilisable par des appels système ultérieurs (`read(2)`, `write(2)`, `lseek(2)`, etc.).

Le paramètre `flags` doit inclure l'un des mode d'accès suivants : `O_RDONLY`, `O_WRONLY` ou `O_RDWR`. Ceux-ci demandent respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture-écriture. À cette valeur peut être ajouté zéro ou plusieurs attributs de création de fichier et attributs d'état de fichier avec un OU binaire. Quelques attributs possibles :

— `O_CREAT` Créer le fichier s'il n'existe pas. `mode` indique les permissions à utiliser si un nouveau fichier est créé. Cet argument doit être fourni lorsque `O_CREAT` est spécifié dans flags.

— `O_TRUNC` Si le fichier existe, est un fichier régulier, et est ouvert en écriture (`O_RDWR` ou `O_WRONLY`), il sera tronqué à une longueur nulle.

`open` renvoie le nouveau descripteur de fichier s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

`READ` - Lire depuis un descripteur de fichier

```
ssize_t read(int fd, void *buf, size_t count);
```

`read` lit jusqu'à `count` octets depuis le descripteur de fichier `fd` dans le tampon pointé par `buf`.

`read` renvoie -1 s'il échoue, auquel cas `errno` contient le code d'erreur, et la position de la tête de lecture est indéfinie. Sinon, `read` renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre.

`WRITE` - Écrire dans un descripteur de fichier

```
ssize_t write(int fd, const void *buf, size_t count);
```

`write` écrit jusqu'à `count` octets dans le fichier associé au descripteur `fd` depuis le tampon pointé par `buf`.

`write` renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

`LSEEK` - Positionner la tête de lecture/écriture dans un fichier

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

La fonction `lseek` place la tête de lecture/écriture à la position `offset` dans le fichier ouvert associé au descripteur `fd` en suivant la directive `whence` ainsi :

— `SEEK_SET` La tête est placée à `offset` octets depuis le début du fichier.

— `SEEK_CUR` La tête de lecture/écriture est avancée de `offset` octets.

— `SEEK_END` La tête est placée à la fin du fichier plus `offset` octets.

`lseek`, s'il réussit, renvoie le nouvel emplacement, mesuré en octets depuis le début du fichier.

`CLOSE` - Fermer un descripteur de fichier

```
int close(int fd);
```

`close` ferme le descripteur `fd`, de manière à ce qu'il ne référence plus aucun fichier, et puisse être réutilisé.

S'il réussit, `close` renvoie 0. S'il échoue, il renvoie -1 et `errno` est renseigné en conséquence.

`FORK` - Créer un processus fils

```
pid_t fork(void);
```

`fork` crée un nouveau processus en dupliquant le processus appelant. Le nouveau processus, que l'on appelle *processus fils*, est la copie exacte du processus appelant, que l'on appelle *processus père* ou *parent* à quelques exceptions près, notamment :

— Le fils a son propre identifiant de processus (PID). Ce PID est unique et ne correspond à aucun autre identifiant de groupe de processus existant.

— Le PPID (*Parent Process ID*) du fils est identique au PID du parent.

En cas de succès, le PID du fils est renvoyé au processus parent, et 0 est renvoyé au processus fils.

`EXECLP` - Exécuter un programme

```
int execlp (const char *file, const char *arg, ...);
```

La fonction `execlp` remplace l'image mémoire du processus en cours par un nouveau processus.

L'argument initial est le chemin d'accès du fichier à exécuter.

L'argument `const char *arg` ainsi que les points de suspension peuvent être vus comme une liste `arg0`, `arg1`, ..., `argn` d'un ou plusieurs pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui constituent les arguments disponibles pour le programme à exécuter. Par convention le premier argument doit pointer sur le nom du fichier associé au programme à exécuter. La liste des arguments doit se terminer par un pointeur `NULL`.