Chapitre 4

La bibliothèque standard

4.1 Les descripteurs de fichiers.

Le fichier d'inclusion `<stdio.h>` contient la définition du type `FILE`. Ce type est une structure contenant les informations nécessaires au système pour la manipulation d'un fichier ouvert. Le contenu exact de cette structure peut varier d'un système à l'autre (UNIX, VMS, autre).

Toutes les fonctions d'E/S utilisent en premier argument un pointeur sur une telle structure : `FILE *`. Le rôle de cet argument est d'indiquer le flux sur lequel on doit effectuer l'opération d'écriture ou de lecture.

Pour pouvoir utiliser une fonction d'entrée-sortie il faut donc avoir une valeur pour ce premier argument, c'est le rôle de la fonction `fopen` de nous fournir ce pointeur en "ouvrant" le fichier. Les deux fonctions `printf` et `scanf` sont des synonymes de

```
fprintf(stdout, format, ...)
```

et

```
fscanf(stdin, format, ...)
```

où `stdout` et `stdin` sont des expressions de type `FILE *` définies sous forme de macro-définitions dans le fichier `<stdio.h>` . Avec POSIX ce sont effectivement des fonctions.

Sur les système de la famille UNIX les fichiers ouverts par un processus le restent dans ses fils. Par exemple le shell a en général trois flux standards :

`stdin` le terminal ouvert en lecture.

`stdout` le terminal ouvert en écriture.

`stderr` le terminal ouvert en écriture, et en mode non bufferisé.

ainsi si l'exécution d'un programme C est réalisée à partir du shell le programme C a déjà ces trois descripteurs de fichiers utilisables. C'est pourquoi il est en général possible d'utiliser `printf` et `scanf` sans ouvrir préalablement de fichiers. Mais si l'entrée standard n'est pas ouverte, `scanf` échoue :

```
#include <stdio.h>
main()
{
    int i;

    if (scanf("%d", &i) == EOF)
    {
        printf("l\'entree standard est fermee\n");
    }
    else
    {
        printf("l\'entree standard est ouverte\n");
```

```
    }
}
```

Compilé,(`a.out`), cela donne les deux sorties suivantes :

```
$ a.out
l'entree standard est ouverte
$ a.out <&- # fermeture de l'entree standard en ksh
l'entree standard est fermee
```

De même `printf` échoue si la sortie standard est fermée.

4.1.1 Ouverture d'un fichier

La fonction de la bibliothèque standard `fopen` permet d'ouvrir un fichier ou de le créer.

```
#include <stdio.h>
FILE  *fopen(const char *filename,
             const char *type);
```

filename est une référence absolue ou relative du fichier à ouvrir ; si le fichier n'existe pas alors il est créé *si et seulement si* l'utilisateur du processus a l'autorisation d'écrire dans le répertoire.

type est une des chaînes suivantes :

"r" ouverture en lecture au début du fichier

"w" ouverture en écriture au début du fichier avec écrasement du fichier si il existe (le fichier est vidé de son contenu à l'ouverture).

"a" ouverture en écriture à la fin du fichier (mode append).

"r+","w+","a+" ouverture en lecture écriture respectivement au début du fichier, au début du fichier avec écrasement, à la fin du fichier.

```
FILE *f;
...
if ((f = fopen("toto", "r")) == NULL)
{
    fprintf(stderr, "impossible d'ouvrir toto\n");
    exit(1);
}
...
```

La fonction retourne un pointeur sur un descripteur du fichier ouvert ou `NULL` en cas d'échec, (accès interdit, création impossible, etc).

4.1.2 Redirection d'un descripteur : `freopen`

Permet d'associer un descripteur déjà utilisé à une autre ouverture de fichier. Ceci permet de réaliser facilement les redirections du shell.

```
FILE *freopen(const char *ref,
              const char *mode,
              FILE *f)
```

Par exemple les redirections de la ligne shell :

```
com <ref1 >>ref2
```

peuvent être réalisées avec

```
if (!freopen("ref1", "r", stdin) || !freopen("ref2", "a", stdout))
{
    fprintf(stderr, "erreur sur une redirection\n");
    exit(1);
}
execl("./com", "com", NULL);
```

4.1.3 Création de fichiers temporaires

La fonction

```
#include <stdio.h>
FILE *tmpfile(void);
```

crée et ouvre en écriture un nouveau fichier temporaire, qui sera détruit (un `unlink` est réalisé immédiatement) à la fin de l'exécution du processus, attention le descripteur est la aussi hérité par les fils, et il faut en gérer le partage. Cette fonction utilise la fonction

```
int mkstemp(char *patron);
```

Les 6 dernier caractère du chemin patron doivent être "XXXXXX" il seront remplacé par une chaine rendant le nom unique, ce chemin sera utilisé pour ouvrir un fichier temporaire avec l'option création et echec sur création avec les droit 0600 ce qui permet d'éviter des troux de sécurité. La fonction retourne le descripteur. Attention `mkstemp` n'assure pas que le fichier sera détruit après utilisation comme c'etait le cas avec `tmpfile` , par contre il devient très difficile de réaliser une attaque sur les fichiers temporaire créer par mkstemp.

4.1.4 Ecriture non formatée

Les deux fonctions suivantes permettent d'écrire et de lire des zones mémoire, le contenu de la mémoire est directement écrit sur disque sans transformation, et réciproquement le contenu du disque est placé tel quel en mémoire. L'intérêt de ces fonctions est d'obtenir des entrées sorties plus rapides et des sauvegardes disque plus compactes mais malheureusement illisibles (binaire). D'autre part les fonction de lecture et d'ecriture sont exactement symétrique ce qui n'est pas le cas de `scanf` et `printf`

```
#include <stdio.h>
int fwrite(void *add, size_t ta, size_t nbobjets, FILE *f);
```

Ecrit `nbobjets` de taille `ta` qui se trouvent à l'adresse `add` dans le fichier de descripteur `f`.

```
#include <stdio.h>
int fread(void *add, size_t ta, size_t nbobjets, FILE *f);
```

Lit `nbobjets` de taille `ta` dans le fichier de descripteur `f` et les place à partir de l'adresse `add` en mémoire.

Attention : La fonction `fread` retourne 0 si l'on essaye de lire au delà du fichier. Pour écrire une boucle de lecture propre on utilise la fonction `feof(FILE *)` :

```
int n[2];
while (fread(n, sizeof(int), 2, f), !feof(f))
      printf("%d %d \n", n[0], n[1]);
```

4.1.5 Accès séquentiel

On distingue deux techniques d'accès aux supports magnétiques :
- L'accès séquentiel qui consiste à traiter les informations dans l'ordre où elle apparaissent sur le support (bandes). Le lecteur physique avance avec la lecture, et se positionne sur le début de l'enregistrement suivant.
- L'accès direct qui consiste à se placer directement sur l'information sans parcourir celles qui la précèdent (disques). Le lecteur physique reste sur le même enregistrement après une lecture.

En langage C l'accès est séquentiel mais il est possible de déplacer le "pointeur de fichier" c'est à dire sélectionner l'indice du prochain octet à lire ou écrire.

Comme nous venons de le voir dans les modes d'ouverture, le pointeur de fichier peut être initialement placé en début ou fin de fichier.

Les quatre fonctions d'entrée-sortie (`fgetc, fputc, fscanf, fprintf`) travaillent séquentiellement à partir de cette origine fixée par `fopen`, et modifiable par `fseek`.

4.1.6 Manipulation du pointeur de fichier

Le pointeur de fichier est un entier `long` qui indique à partir de quel octet du fichier la prochaine fonction d'entrée-sortie doit s'effectuer.

En début de fichier cet entier est nul.

```
#include <stdio.h>
int fseek(FILE *f, long pos, int direction);
```

`f` le descripteur du fichier dans lequel ont déplace le pointeur.

direction est une des trois constantes entières suivantes :

SEEK_SET positionnement sur l'octet pos du fichier

SEEK_CUR positionnement sur le `pos`-ième octet après la position courante du pointeur de fichier. (équivalent à SEEK_SET courant+pos).

SEEK_END positionnement sur le `pos`-ième octet après la fin du fichier.

Remarquer que `pos` est un entier signé : il est possible se placer sur le 4ième octet avant la fin du fichier :

```
fseek(f, -4L, SEEK_END);
```

4.1.7 Un exemple d'accès direct sur un fichier d'entiers.

La fonction suivante lit le n-ième entier d'un fichier d'entiers préalablement écrit grâce à `fwrite` :

```
int lirenieme(int n, FILE *f)
{
    int buf;

    fseek(f, sizeof(int)*(n-1), SEEK_SET);
    fread(&buf, sizeof(int), 1, f);
    return buf;
}  \istd{fseek}\istd{fread}
```

4.1.8 Les autres fonctions de déplacement du pointeur de fichier.

La fonction `ftell`

```
long int ftell(FILE *);
```

retourne la position courante du pointeur.
La fonction `rewind`

```
void rewind(FILE *f);
```

équivalent à : `(void) fseek (f, 0L, 0)`

4.2 Les tampons de fichiers de stdlib.

La bibliothèque standard utilise des tampons pour minimiser le nombre d'appels système. Il est possible de tester l'efficacité de cette bufferisation en comparant la vitesse de recopie d'un même fichier avec un tampon de taille 1 octet et un tampon adapté à la machine, la différence devient vite très importante. Une façon simple de le percevoir est d'écrire un programme `com` qui réalise des écritures sur la sortie standard ligne par ligne, de regarder sa vitesse puis de comparer avec la commande suivantes `:com | cat` la bibliothèque standard utilisant des buffer différents dans les deux cas une différence de vitese d'exécution est perceptible (sur une machine lente la différence de vitesse est évidente, mais elle existe aussi sur une rapide...).

4.2.1 Les modes de bufferisation par défaut.

Le mode de bufferisation des fichiers ouverts par la bibliothèque standard dépend du type de périphérique.

- Si le fichier est un **terminal** la bufferisation est faite ligne à ligne.
  En *écriture* le tampon est vidé à chaque écriture d'un `'\n'` , ou quand il est plein (première des deux occurences).
  En *lecture* le tampon est rempli après chaque validation (RC), si l'on tape trop de caractères le terminal proteste (beep) le buffer clavier étant plein.
- Si le fichier est sur un **disque magnétique**
  En *écriture* le tampon est vidé avant de déborder.
  En *lecture* le tampon est rempli quand il est vide.

Le shell de login change le mode de bufferisation de **stderr** qui est un fichier terminal à non bufferisé.

Nous avons donc à notre disposition trois modes de bufferisation standards :
- Non bufferisé (sortie erreur standard),
- Bufferisé par ligne (lecture/écriture sur terminal),
- Bufferisé par blocs (taille des tampons du buffer cache).

Un exemple de réouverture de la sortie standard, avec perte du mode de bufferisation :

```
#include <stdio.h>
main()
{
    freopen("/dev/tty", "w", stderr);
    fprintf(stderr, "texte non termine par un newline ");
    sleep(12);
    exit(0); /* realise fclose(stderr) qui realise fflush(stderr) */
}
```

Il faut attendre 12 secondes l'affichage.

4.2.2 Manipulation des tampons de la bibliothèque standard.

Un tampon alloué automatiquement (malloc) est associé à chaque ouverture de fichier par `fopen` au moment de la première entrée-sortie sur le fichier.

La manipulation des tampons de la bibliothèque standard comporte deux aspects :

1. Manipulation de la bufferisation de façon ponctuelle (vidange).
2. Positionnement du mode de bufferisation.

Manipulations ponctuelles

La fonction suivante permet de vider le tampon associé au FILE * f :

```
#include <stdio.h>
fflush(FILE *f);
```

En écriture force la copie du tampon associé à la structure f dans le tampon système (ne garantit pas l'écriture en cas d'interruption du système !).
En lecture détruit le contenu du tampon, si l'on est en mode ligne uniquement jusqu'au premier caractère `'\n'`.

La fonction fclose() réalise un fflush() avant de fermer le fichier.
La fonction exit() appel fclose() sur tous les fichiers ouvert par fopen (freopen,tmpfile,...) avant de terminer le processus.

Manipulations du mode de bufferisation et de la taille du tampon.

La primitive

```
int setvbuf(FILE *f,
      char *adresse,
      int mode,
      size_t taille);
```

permet un changement du mode de bufferisation du fichier *f* avec un tampon de taille *taille* fourni par l'utilisateur à l'adresse *adresse* si elle est non nulle, avec le *mode* défini par les macro-définitions suivantes (`<stdio.h>`) :

```
_IOFBF        bufferise
_IONBF        Non bufferise
_IOMYBUF      Mon buffer
_IOLBF        bufferise par ligne (ex: les terminaux)
```

Attention : Il ne faut pas appeler cette fonction après l'allocation automatique réalisée par la bibliothèque standard après le premier appel à une fonction d'entrée-sortie sur le fichier.
Il est fortement conseillé que la zone mémoire pointée par *adresse* soit au moins d'une taille égale à *taille*.

Seul un passage au mode bufferisé en ligne ou non bufferisé peut être réalisé après l'allocation automatique du tampon, au risque de perdre ce tampon (absence d 'appel de `free`). Ce qui permet par exemple de changer le mode de bufferisation de la sortie standard après un `fork`. Attention ce peut être dangereux, pour le contenu courant du tampon comme le montre l'exemple suivant.

Avant cette fonction de norme POSIX on utilisait trois fonctions :

```
void setbuf(FILE *f, char *buf);
void setbuffer(FILE *f,char *adresse,size_t t);
void setlignebuf(FILE *f);
```

```
#include <stdio.h>
main()
{
  printf("BonJour ");
  switch(fork())
  {
    case -1 :
       exit(1);
    case  0 :
       printf("je suis le fils");
/* version 1 sans la ligne suivante version 2 avec  */
       setbuffer(stdout, NULL, 0);
       sleep(1);
       printf("Encore le fils");
       break;
    default :
       printf("je suis le pere");
       sleep(2);
  }
  printf("\n");
}
version 1
fork_stdlib
BonJour je suis le fils Encore le fils
BonJour je suis le pere
version 2
Encore le fils
BonJour je suis le pere
```

4.3 Manipulation des liens d'un fichier

Changer le nom d'un fichier :

```
int rename(const char *de,const char *vers);
```

permet de renommer un fichier (ou un répertoire). Il faut que les deux références soient de même type (fichier ou répertoire) dans le même système de fichiers.
Rappel : ceci n'a d'effet que sur l'arborescence de fichiers.

Détruire une référence :

```
int remove(const char *filename);
```

Détruit le lien donné en argument, le système récupère l'inode et les blocs associés au fichier si c'était le dernier lien.

4.4 Lancement d'une commande shell

```
#include <stdlib.h>
int system(const char *chaine_de_commande);
```

Crée un processus "/bin/posix/sh" qui exécute la commande ; il y a attente de la fin du shell, (la commande peut elle être lancée en mode détaché ce qui fait que le shell retourne immédiatement

sans faire un `wait`). Ce mécanisme est très coûteux. Attention la commande `system` bloque les signaux SIGINT et SIGQUIT, il faut analyser la valeur de retour de `system` de la même façons que celle de `wait`. Il est conseillé de bloquer ces deux signaux avant l'appel de `system` .

4.5 Terminaison d'un processus

_exit La primitive de terminaison de processus de bas niveau :

```
#include <stdlib.h>
void _exit(int valeur);
```

La primitive `_exit` est la fonction de terminaison "bas niveau"
- elle ferme les descripteurs ouverts par *open, opendir* ou hérités du processus père.
- la `valeur` est fournie au processus père qui la récupère par l'appel système *wait*. Cette valeur est le code de retour de processus en shell.

Cette primitive est automatiquement appelée à la fin de la fonction `main` (sauf en cas d'appels récursifs de `main`).

exit La fonction de terminaison de processus de stdlib :

```
#include <stdlib.h>
void exit(int valeur);
```

la fonction *exit* :
- lance les fonctions définies par *atexit*.
- ferme l'ensemble des descripteurs ouverts grâce à la bibliothèque standard (*fopen*).
- détruit les fichiers fabriqués par la primitive *tmpfile*
- appelle *_exit* avec `valeur`.

atexit La primitive `atexit` permet de spécifier des fonctions à appeler en fin d'exécution, elle sont lancées par `exit` dans l'ordre inverse de leur positionnement par `atexit`.

```
#include <stdlib.h>
int atexit(void (*fonction) (void ));
```

Exemple :

```
void bob(void) {printf("coucou\n");}
void bib(void) {printf("cuicui ");}

main(int argc)
{
  atexit(bob);
  atexit(bib);
  if (argc - 1)
    exit(0);
  else
    _exit(0);
}
$ make atexit
cc   atexit.c  -o atexit
$ atexit
$ atexit unargument
cuicui coucou
$
```

4.6 Gestion des erreurs

Les fonctions de la bibliothèque standard positionnent deux indicateurs d'erreur, la fonction suivante les repositionne :

```
void clearerr(FILE *);
```

La fonction `int feof(FILE *)` est vraie si la fin de fichier est atteinte sur ce canal, `int ferror(FILE *)` est vraie si une erreur a eu lieu pendant la dernière tentative de lecture ou d'écriture sur ce canal.
Une description en langue naturelle de la dernière erreur peut être obtenue grace à

```
void perror(const char *message);
```

l'affichage se fait sur la sortie erreur standard (`stderr`).

4.7 Création et destruction de répertoires

Création d'un répertoire vide (même syntaxe que `creat`) :

```
#include <unistd.h>
int mkdir(char *ref, mode_t  mode);
```

Destruction :

```
int rmdir(char *ref);
```

avec les mêmes restrictions que pour les shells sur le contenu du répertoire (impossible de détruire un répertoire non vide).