# Correction TD Programmation Système: Feuille 3

**Informatique 2ème année. ENSEIRB 2013/2014**

—*Mathieu Faverge* - `mfaverge@enseirb.fr` —

## Gestion de fichiers avec les API POSIX et C standard et création de processus

▸**Exercice 1.** *Appels système et libc :*

*Les fonctions suivantes sont-elles des appels système ou des fonctions de la bibliothèque C standard ?*

*`printf()`, `fopen()`, `fclose()`, `fread()`, `open()`, `close()`, `lseek()`, `rewind()`, `write()`, `exit()`, `_exit()`*

*Aidez-vous des prototypes de ces fonctions pour répondre à cette question.*

✂ ----------

**appels système** `open()`, `close()`, `lseek()`, `write()`, `_exit()`

**fonctions libc** `fopen()`, `fclose()`, `fread()`, `rewind()`, `exit()`

Points :

- les fonctions de la libc sont reconnaissables par leur argument `FILE *` quand elles manipulent des fichiers, les appels système par leur argument `int` de descripteur de fichier
- `exit()` ne peut pas être un appel système parce qu'il doit vider les buffers de la bibliothèque C standard. L'appel système correspondant qui quitte brusquement est `_exit()`.

---------- ✂

▸**Exercice 2.** *libc et buffers :*

*Le but de cet exercice est de mettre en évidence l'effet des* buffers (tampons) *sur les entrées-sorties.*

*Le schéma du programme que vous allez écrire est le suivant :*

1. *afficher une chaîne de caractères de début*
2. *mettre le processus en attente une seconde avec `sleep(3)`*
3. *afficher une chaîne de caractères de fin*
4. *quitter*

*Afin de n'avoir qu'un seul programme source, vous allez utiliser le préprocesseur C pour conditionner le fonctionnement de votre programme.*

- *la chaîne de caractères de début sera contenue dans la variable du préprocesseur (macro) `CHAINE1`*
- *la chaîne de caractères de fin sera contenue dans la variable du préprocesseur `CHAINE2`.*
- *la première écriture se fera avec `fprintf(3)` si `UTILISER_FPRINTF1` est positionnée, et avec `write(2)` sinon*
- *la deuxième écriture sera conditionnée de la même manière par `UTILISER_FPRINTF2`*
- *la première écriture se fera sur la sortie d'erreur standard si `UTILISER_SORTIE_ERREUR1` est positionnée, sur la sortie standard autrement*
- *la deuxième écriture sera conditionnée de la même manière par `UTILISER_SORTIE_ERREUR2`*
- *le programme quittera par `_exit(2)` si `UTILISER__EXIT` est positionnée, et par `exit(3)` sinon.*

*Afin de voir les différents modes de fonctionnement des tampons, le programme est exécuté de deux manières différentes :*

- `./programme` *afin que les sorties soient sur le terminal*
- `./programme 2>&1 | cat -u` *afin que les sorties du programme passent par un tube avant d'être affichées*

*Mettez en évidence les trois modes de fonctionnement des buffers de la bibliothèque C standard, ainsi que le fait que les appels système n'utilisent pas de buffers visibles au niveau utilisateur.*

*Testez notamment les modes de compilations suivants :*

— `-DUTILISER_FPRINTF1`
— `-DUTILISER_FPRINTF2 -DUTILISER_SORTIE_ERREUR2`
— `-DUTILISER_FPRINTF1 -DUTILISER_FPRINTF2 -DUTILISER_SORTIE_ERREUR2` *en mettant un* `\n` *dans la première chaîne.*
— ...

✂ ----------

Au niveau de la couche appels systèmes, il n'y a pas de bufferisation "visible", c'est à dire que lorsqu'on fait un write, au retour de la fonction les données indiquées comme écrites **ont été écrites**.

Le principe de bufferisation consiste donc à différer l'écriture données afin de minimiser le nombre d'appels systèmes (**write** en l'occurrence).

3 modes de bufferisations sont implémentés dans la libc :
— pas de bufferisation : c'est le cas de `stderr` par défaut
— bufferisation par ligne : c'est le retour à la ligne qui déclenche l'envoi du buffer vers la sortie (`\n` ou `\r`)
— bufferisation par bloc : le buffer est vidé lorsqu'il est plein.

on peut modifier le mode de bufferisation avec la fonction `setvbuf`, `setbuf`, ....

La libc adapte le mode de bufferisation en fonction du contexte.

Pour `stderr`, il n'y a pas de bufferisation.

Pour `stdout` (ou tout autre fichier), il y a une bufferisation par ligne si le fichier est associé à un terminal (facile à tester avec la fonction `isatty`). Si le fichier n'est pas un terminal alors le mode de bufferisation est par bloc. Ceci explique le changement de comportement selon que la sortie est associée au terminal ou à autre chose (un tube en l'occurrence).

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

#define CHAINE1 "Debut\n"
#define CHAINE2 "Fin"

int
main(void) {
    int out1 = 1;
    int out2 = 1;
    FILE *fout1 = stdout;
    FILE *fout2 = stdout;

#ifdef UTILISER_SORTIE_ERREUR1
    out1 = 2;
    fout1 = stderr;
#endif

#ifdef UTILISER_SORTIE_ERREUR2
    out2 = 2;
    fout2 = stderr;
#endif

#ifdef UTILISER_FPRINTF1
    fprintf( fout1, "%s", CHAINE1);
#else
    write( out1, CHAINE1, strlen(CHAINE1));
#endif /* UTILISER_FPRINTF1 */

    sleep(1);

#ifdef UTILISER_FPRINTF2
    fprintf( fout2, "%s", CHAINE2);
#else
    write( out2, CHAINE2, strlen(CHAINE2));
#endif /* UTILISER_FPRINTF2 */

#ifdef UTILISER_EXIT
    _exit(EXIT_SUCCESS);
#else
    exit(EXIT_SUCCESS);
#endif /* UTILISER_EXIT */
}
```

---------- ✂

▸**Exercice 3.** *Fork :*

*Écrivez un programme qui affiche son pid sans revenir à la ligne, qui se duplique avec* `fork(2)`*, et dont l'enfant affiche son pid et celui de son parent. Le processus parent ne devra plus rien afficher après l'appel à* `fork(2)`*.*

*Proposez trois solutions à ce problème, et implémentez-en une qui vous paraît la plus pratique et/ou la plus élégante.*

1. si on fait du printf(), le message initial est affiché deux fois (j'ai pas forcement insisté dessus en cours : les tampons de la libc standard sont flushés dans le parent et dans l'enfant)
2. on peut faire un `fflush(stdout);` avant le `fork()`
3. on peut faire un `_exit()` dans le parent
4. on peut ne pas utiliser la libc standard

Chaque solution a ses avantages dépendant du contexte.

▸**Exercice 4.** *Fork et redirection :*
*Le but de cet exercice est d'écrire un programme* `lance` *qui s'exécutera comme suit :*

```
lance sortie commande arg1 arg2 ...
```

1. *Écrivez un programme qui affiche la commande a exécuter sur sortie, puis exécute la commande à l'aide de la fonction* `execvp`.
2. *Étendez ce programme pour que le programme d'origine crée un nouveau processus qui exécutera la commande.*
3. *Dupliquez la sortie pour redigier la sortie de la commande exécutée sur la sortie passée en paramètre.*
4. *Une fois la commande* `commande` *exécutée, votre programme doit afficher des informations sur le statut de terminaison de* `commande` *(c.f.* `wait(2)`*). Écrivez un programme qui génère une faute de bus (segfault) et utilisez le pour tester le programme* `lance`.

Pour le segfault, indiquez leur tout simplement : `*((int*)0) = 42;`

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

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

int main(int argc, char**argv){
  int fd,status;
  if (argc<2) return EXIT_FAILURE;

  switch(fork()){
  case -1:
    perror("fork");
    return EXIT_FAILURE;
    break;
  case 0:
    fd=open("sortie",O_WRONLY|O_CREAT|O_TRUNC,0644);
    if (fd==-1){ perror("sortie"); return EXIT_FAILURE;}
    if (dup2(fd,1)==-1) {perror("dup2"); return EXIT_FAILURE;}
    execvp(argv[1],argv+1);
    perror(argv[1]);
    return EXIT_FAILURE;
    break;
  }
  if (wait(&status)==-1){ perror("wait"); return EXIT_FAILURE;}
  if (WIFEXITED(status))
    printf("fils termine normalement, code de retour: %d\n",WEXITSTATUS(status));
  else{
    if (WIFSIGNALED(status))
      printf("fils termine par signal, numero de signal: %d\n",WTERMSIG(status));
  }
  return EXIT_SUCCESS;

}
```