Programming Lab - Advanced IO

1- File Descriptors

Premièrement on crée un fichier text classique où l'on écrit « Blabla »

On execute la commande suivante : cat text1 > text2.

```
adrito@ubuntu:~/Documents/ECE/Lab4$ cat text1 > text2
adrito@ubuntu:~/Documents/ECE/Lab4$ ls -l
total 12
-rw-rw-r-- 1 adrito adrito  0 Oct 18 02:07 lab4.c
-rw-rw-r-- 1 adrito adrito  7 Oct 18 02:12 text1
-rw-rw-r-- 1 adrito adrito 85 Oct 18 02:10 text1.c
-rw-r--r-- 1 adrito adrito  7 Oct 18 02:12 text2
adrito@ubuntu:~/Documents/ECE/Lab4$
```

On remarque qu'un fichier text2 est apparu. Il contient le même contenu que text1.

2. Pipe

La commande ps affiche les processus actifs du système. Le paramètre more affiche le résultat sur plusieurs pages, c'est donc plus lisible.

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

int main(int argc, char **argv)
{
    int     pipefd[2];
    pid_t   cpid;

    if(pipe(pipefd) == -1) {
        perror("pipe : ");
        exit(EXIT_FAILURE);
    }

    cpid = fork(); //Creation du process enfant
    if(cpid == -1) {
        perror("fork : ");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) { //Process enfant, execute la commande more
        close(pipefd[1]); //Fermeture de write
        dup2(pipefd[0], STDIN_FILENO);  //fonction dup2 , pour l'input de base, appelée grace à pipfd[0]
        system("more");  //Equivaut à une ligne de commande dans le SHELL
        close(pipefd[0]); //Fermeture de pipfd et du fichier
        close(STDIN_FILENO);
        exit(EXIT_SUCCESS);
    }
    else { //Process parent, execute la commande ps aux
        close(pipefd[0]); //Fermeture de read
        dup2(pipefd[1], STDOUT_FILENO);
        system("ps aux"); //Equivaut à une ligne de commande dans le SHELL
        close(pipefd[1]); //fermeture
        close(STDOUT_FILENO);
        wait(NULL); //Attente de la fin du child
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

3. Non-Blocking Calls

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

int main() {
  int i;
  char buf[100];
// ouvrir un le stdin en lecture non bloquante
//  fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
    for (i = 0; i < 10; i++) {
        int nb;
        nb = read(STDIN_FILENO, buf, 100);
        //
        printf("nwrites = %d\terror = %d\n", nb, errno);
        //On affiche le nb d'octet lu puis la valeur de errno.
    }
}
```

Ce qui donne en sortie :

```
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
adrito@ubuntu:~/Documents/ECE/OS/Lab4$
```

La fonction fcntl prend en argument la sortie standard, F_SETFL (qui set les flags), et O_NONBLOCK qui bloque la lecture de la sortie standard.
Ce qui fait que par la suite, notre programme essaye de lire (read()) une sortie bloquée donc renvoie -1.