TP 3 – Flux

Vous devez faire au moins les exercices 1 à 4.
Un exemple de sujet d'examen concernant cette partie est dans l'exercices 5.
Les exercices 6 et 7 sont des compléments.

Exercice 1 - Flux noyau

On utilisera uniquement des **flux noyau** pour cet exercice. On rappelle que la documentation des fonctions pour manipuler les flux noyau sont dans la section 2 de `man`. Ainsi, pour consulter la documentation de la fonction `write` par exemple (et pas celle de la commande shell `write`), il faudra utiliser la commande suivante :

```
sh$ man 2 write
```

Le but de cet exercice et de créer le programme `delLastByte`, dont le comportement est le suivant :

- Il prend un seul argument, qu'on appelera f ici.
- f doit être un chemin vers un fichier régulier accessible en lecture et en écriture (on renverra un message d'erreur précis dès qu'un de ces points fait défaut).
- Le programme modifie f en lui enlevant son dernier caractère.

Il n'existe pas de fonction permettant d'enlever directement un caractère à un fichier. Nous pouvous seulement lire le contenu d'un fichier, écrire dans un fichier, ou le vider.

Pour répondre au besoin, nous allons donc utiliser l'approche suivante :

a. ouvrir le fichier f en mode *lecture seule*,
b. calculer la taille t du fichier f avec `lseek`,
c. allouer un tableau de taille $t - 1$
d. lire d'un seul bloc les $t - 1$ premiers octets de f et stocker le résultat dans notre tableau,
e. fermer f et le réouvrir en mode *écriture seule* plus *troncature*.
f. écrire le contenu du tableau dans f,
g. fermer le fichier et libérer la mémoire allouée.

1. Écrivez le fichier `delLastByte.c`.

2. Compilez et testez votre programme.

3. Écrivez une variante `delLastByte-v2.c` en utilisant cette fois la fonction `fstat` pour récupérer la taille du fichier.

   note : Cette fonction a un comportement similaire à celui de la fonction `scanf`. Pour les détails, consultez le man de `fstat` et en particulier la partie `EXAMPLES`.

Exercice 2 - Flux `libc`

1. Écrivez le programme `delLastByte-libc` qui est fonctionnellement identique à `delLastByte` (exercice précédent), mais qui est implémenté en utilisant uniquement les **flux libc**.

   note : On rappelle que la documentation des fonctions de la `libc` est située dans la section 3 du man. En particulier, on pourra consulter la documentation de `fseek` et `ftell`, qui serviront à calculer la taille du fichier.

Exercice 3 - `execl`, `dup`, `dup2`

1. Que fait la fonction `lire` du fichier `/pub/FISE_OSSE11/syscall/lire.c` ?
   Précisez en particulier à quoi correspondent les 0 et 1 utilisés dans les appels à `read` et `write`.

2. On souhaite créer un programme `lire` qui écrit `"Je suis le programme lire."` sur la sortie standard d'erreur, puis qui appelle la fonction `lire`.

   a. Écrivez dans un nouveau fichier `lire.c` (que vous placerez où vous voulez) le code suivant.

```c
#include <stdio.h>
#include "/pub/FISE_OSSE11/syscall/lire.h"

int main() {
    fprintf(stderr, "Je suis le programme lire.\n");
    lire();
    return 0;
}
```

   b. Générez l'exécutable `lire`.

```
sh$ gcc -Wall -Wextra -o lire /pub/FISE_OSSE11/syscall/lire.c lire.c
```

   c. Testez votre nouveau programme.

```
sh$ ./lire < /pub/FISE_OSSE11/syscall/data.in
Je suis le programme lire.
lire(): nbCarLus = 10
aaaaaaaaa
```

3. On va maintenant créer le programme `lire3exec`, qui appelle la fonction `lire` deux fois, puis qui exécute le programme `lire` précédent grâce à un appel à `execl`.

   a. Écrivez le code de ce programme dans `lire3exec.c`.

   b. Générez l'exécutable `lire3exec`, et testez le en redirigeant le fichier standard d'entrée sur `/pub/FISE_OSSE11/syscall/data.in` et le fichier standard de sortie sur `data.out` :

```
sh$ ./lire3exec < /pub/FISE_OSSE11/syscall/data.in > data.out
lire(): nbCarLus = 10
lire(): nbCarLus = 10
Je suis le programme lire.
lire(): nbCarLus = 10
sh$ cat data.out
aaaaaaaaa
bbbbbbbbb
ccccccccc
```

   c. Comment sont transmis les flux lors de l'appel à `execl` ?

4. a. Écrivez le programme `lire3a`, similaire à `lire3exec` mais qui :
   - prend en argument deux chemins vers des fichiers,
   - redirige le `read(0,...)` de la fonction `lire` sur le fichier donné par `argv[1]`,
   - redirige le `write(1,...)` de la fonction `lire` sur le fichier donné par `argv[2]`.

aide :

— Copiez `lire3exec.c` dans `lire3a.c`.

— Avant le premier `lire`, ouvrez le fichier `argv[1]` en lecture et le fichier `argv[2]` en écriture + troncature + création si besoin.

```
open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

— Utilisez `dup2` pour que le flux 0 correponde au fichier `argv[1]`.

— Utilisez `dup2` pour que le flux 1 correponde au fichier `argv[2]`.

b. Compilez le, puis faites les tests ci-dessous.

```
sh$ ./lire3a
./lire3a: usage: ./lire3a infile outfile
sh$ ./lire3a /pub/FISE_OSSE11/syscall/data.in /ho/la/la
./lire3a: cannot open '/ho/la/la' for writing: No such file or directory
sh$ ./lire3a /ho/la/la data.out
./lire3a: cannot open '/ho/la/la' for reading: No such file or directory
sh$ ./lire3a /pub/FISE_OSSE11/syscall/data.in data.out
lire(): nbCarLus = 10
lire(): nbCarLus = 10
Je suis le programme lire.
lire(): nbCarLus = 10
sh$ cat data.out
aaaaaaaaa
bbbbbbbbb
ccccccccc
```

5. Écrivez le programme `lire3b` similaire à `lire3a`, mais qui effectue :
   — le `read` et le `write` du premier appel à `lire` avec les flux originaux,
   — les `read` et `write` suivants sur les fichiers `argv[1]` et `argv[2]`.

aide :

— Copiez `lire3a.c` dans `lire3b.c`.

— Déplacez les `dup2` après le premier appel à `lire()`.

Faites le test ci-dessous.

```
sh$ echo 111111111 | ./lire3b /pub/FISE_OSSE11/syscall/data.in data.out
lire(): nbCarLus = 10
111111111
lire(): nbCarLus = 10
Je suis le programme lire.
lire(): nbCarLus = 10
sh$ cat data.out
aaaaaaaaa
bbbbbbbbb
```

6. Écrivez le programme `lire3c` similaire à `lire3b`, mais qui effectue :
   — le `read` et le `write` du premier appel à `lire` sur les flux originaux,
   — le `read` et le `write` du deuxième appel à `lire` sur les fichiers `argv[1]` et `argv[2]`,
   — le `read` et `write` du troisième appel à `lire` sur les flux originaux.

aide :

- Copiez `lire3b.c` dans `lire3c.c`.
- Avant les `dup2`, sauvegardez les flux 0 et 1 (utilisez `dup`).
- Avant l'appel à `execl`, restaurez les flux 0 et 1 avec `dup2`.

Faites le test ci-dessous.

```
sh$ { echo 111111111 ; echo 222222222 ; } \
    | ./lire3c /pub/FISE_OSSE11/syscall/data.in data.out
lire(): nbCarLus = 10
111111111
lire(): nbCarLus = 10
Je suis le programme lire.
lire(): nbCarLus = 10
222222222
sh$ cat data.out
aaaaaaaaa
```

Exercice 4 - Mise en place d'un wrapper (version C)

L'exécutable `/pub/FISE_OSSE11/shell/iacmp/iacmp` compare les chaînes de caractères passées en argument et les écrits sur le flux de sortie standard, triées par ordre alphabétique.

Pour fonctionner, il a besoin des variables d'environnement :

`IacmpDir` qui doit contenir la valeur `/pub/FISE_OSSE11/shell/iacmp`,

`LD_LIBRARY_PATH` qui doit contenir la valeur `/pub/FISE_OSSE11/shell/iacmp/lib`.

1. Complétez le code `/pub/FISE_OSSE11/syscall/iacmp.c`.
   Ce code jouera le rôle de *wrapper* pour `/pub/FISE_OSSE11/shell/iacmp/iacmp` en configurant son environnement, puis en le lançant tout en lui transmettant les arguments grâce à `execv`.

2. Une fois compilé, testez le avec la commande ci-dessous. Elle doit fonctionner quelque soit le nombre d'arguments.

```
sh$ ./iacmp 'chat roux' 'chat blanc' 'chat noir'
chat blanc
chat noir
chat roux
```

Exercice 5 - Numérotation des lignes **(examen 2023-2024)**

L'objectif de cet exercice est d'écrire un programme `numbering`, qui va lire un flux d'entrées ligne par ligne, et va restituer ces lignes précédées de leurs numéros de ligne et du symbole `'\t'` (tabulation).

On utilisera des **flux *libc*** pour répondre aux questions.

1. Dans un premier temps, on va utiliser les flux standards d'entrée et de sortie.

   Donner le code de `numbering.c`.

   notes :

   — Il est recommandé de lire les caractères un par un grâce à `fread`.

   — Un exemple d'exécution est donné par :

   ```
   sh$ { echo 'first line' ; echo 'second line' ; } | ./numbering
   1       first line
   2       second line
   ```

2. Expliquer comment adapter le code `C` pour que l'entrée soit lue à partir d'un fichier dont le nom est passé en argument du programme `numbering`.

Exercice 6 - Lecture d'un mot de passe

1. Dans le fichier `readpasswd.c`, écrivez une fonction

```
int readpassword(char* pass) { ... }
```

qui :
- lit un mot de passe d'au plus 15 caractères sur l'entrée standard ;
- fait cette lecture caractère par caractère jusqu'à un saut de ligne (`'\n'` ou `'\r'`) ;
- compte le nombre n de caractères lus (sans compter le saut de ligne) ;
- stocke le résultat de cette lecture dans la variable `pass` passée en argument (on supposera qu'il s'agit d'un pointeur vers une zone mémoire valide de 16 octets et seuls les 15 premiers caractères seront écrits) ;
- met la valeur `'\0'` dans `pass[15]` si $n \geq 16$, et dans `pass[n]` sinon ;
- renvoie le nombre n.

2. Écrivez aussi une fonction `main`. Compilez et testez votre programme.

3. Le problème du code précédent est que le mot de passe saisi est affiché en clair dans le terminal. Pour y remédier, procédez de la manière suivante :
   a. rechercher le nom du fichier associé à l'entrée standard du terminal grâce à la fonction `ttyname`,

```
char* ttypath = ttyname(0);
if (ttypath == NULL) { fprintf(...); exit(1); }
```

   b. ouvrir le fichier `ttypath` en lecture/écriture,
   c. mettre le mode de `ttypath` à *raw* et *sans écho*,

```
system("/bin/stty raw -echo");
```

   Le mode *sans écho* corrige notre problème d'affichage du mot de passe.

   Le mode *raw* sera quant à lui très utile pour la suite. Il permet de recevoir les caractères dès leur saisie (sinon, les caractères sont envoyés seulement lorsque l'utilisateur appuie sur <ENTER>, produisant ainsi un saut de ligne). De plus, les caractères spéciaux comme <CTRL-c> ne sont plus interprétés.
   d. écrire la chaîne `"password: "` et appeler la fonction `readpasswd`,
   e. remettre `ttypath` dans un mode normal en lançant la commande

```
/bin/stty -raw echo
```

   via un appel à la fonction `system`.

note : Pour des raisons de sécurité, l'utilisation de la fonction `system` est fortement déconseillé. L'usage qui en est fait dans ce TP reste cependant raisonnable car :

1. On s'efforce d'appeler `stty` via son chemin absolu. On ne risque donc pas de lancer par mégarde un autre `stty` qui serait situé ailleurs dans le PATH de l'utilisateur courant ;
2. Les arguments de la commande `stty` sont donnés *en dur* dans le code. Si ces arguments étaient issus de données saisies par l'utilisateur, une personne malveillante pourrait en profiter pour essayer d'injecter son propre code shell (ce qui serait dramatique en cas du succès).

Il vaudrait mieux utiliser `execv` ou `execl` comme nous l'avons fait dans les autres exercices. Cela n'est cependant pas applicable directement ici (pourquoi ?), mais une solution sera apportée dans le cours suivant.

4. À cause du mode *sans echo*, l'utilisateur n'a plus aucune information sur ce qu'il saisit. Modifiez la fonction `readpasswd` pour qu'elle affiche un caractère `'*'` à chaque nouveau caractère reçu (sauf le saut de ligne).

5. La commande

   sh$ echo mon-pass | ./a.out

   permet-elle d'entrer un mot de passe ?

6. Modifiez votre code pour gérer le caractère *backspace*.

   aide :
   En mode `raw`, l'appui sur la touche *backspace* se traduit par l'émission du caractère numéro 127.

   S'il n'y a rien à effacer, on ignore ce caractère. Sinon, on peut écrire la chaîne `"\b \b"`, qui aura pour effet visuel d'effacer un caractère `*`.

Exercice 7 - factorielle

1. Complétez le programme `/pub/FISE_OSSE11/syscall/factorial.c` qui, si il est lancé avec comme nom d'exécutable n, calcule et affiche la valeur $n!$.

   Son fonctionnement est le suivant :

   lancé avec 1 argument non vide (le nom du programme)

   - Il vérifie que le nom de base de cet argument correspond à un nombre entier positif.
   - Il initialise la première variable d'environnement à ce nombre.
   - Il initialise la deuxième variable d'environnement au nombre 1.
   - Il initialise la troisième variable d'environnement au nom du programme.
   - Il exécute le programme avec un seul argument vide et un environnement contenant les 3 variables initialisées ci-dessus.

   lancé avec 1 argument vide

   - Il extrait `n`, `fac` et `cmd` à partir des trois premières variables d'environnement (on utilisera `strtol` pour passer d'une chaîne de caractère à un entier).
   - Si `n` vaut 1, il écrit `fac` sur la sortie standard et s'arrête.
   - Sinon, il exécute le programme avec un seul argument vide et un environnement composé de trois variables ayant pour valeurs respectives `n-1`, `n*fac` et `cmd`.

   lancé avec plus que un argument

   - Il affiche un message d'erreur, puis s'arrête.

2. Une fois compilé, testez le en calculant les factorielles de 5 et de 6.

3. À quoi sert la variable `tmp` ?

4. Ce programme respecte-t-il les conventions d'Unix pour les paramètres ?

5. Ce programme respecte-t-il les conventions d'Unix pour les variables d'environnement ?

6. Lors du calcul de la factorielle de 6, combien de processus ont été créés ?

7. Lors du calcul de la factorielle de 6, combien de programmes ont été exécutés ?