TP 2
Manipulation de buffers et découverte de read/write (1h30)

Téléchargez l'archive `lifse-tp02.tar.gz` sur le site de l'UE, et placez là dans le répertoire que vous utilisez pour l'UE. Dans ce répertoire, décompressez l'archive avec la commande `tar xzvf lifse-tp02.tar.gz`; un répertoire `lifse-tp02` a été créé, dans lequel vous travaillerez pour ce TP.

EXERCICE 1 ▶ Manipulation de buffers et de strings

Cet exercice a pour objectif de vous faire manipuler des buffers et des strings en C++. Cela sera utile pour le reste de l'UE. **Créez** le fichier `buffer.cpp`. La commande `g++ buffer.cpp -o buffer` permettra de le compiler; `./buffer` pour l'exécuter.

1) Dans le programme principal, créez un tableau de caractères en utilisant `char tab[11];`. Remplissez ce tableau pour créer la chaîne de caractères `"abcdefghij"`. Pensez au caractère spécial à mettre à la fin.

2) Faites la même chose en utilisant une objet `str` de la classe `std::string`.

3) Affichez le tableau `tab` et le string `str` (en utilisant le flux `std::cout`). Pour chaque cas, affichez ensuite uniquement leur *troisième* élément.

4) Afin de vous habituer à la manipulation des buffers (qui serviront pour les appels systèmes), créez un tableau de taille plus grand (`char big_tab[21];`) dont vous initialiserez chacun des éléments avec le caractère `'-'` (sauf le dernier qui contiendra le caractère spécial).

5) En utilisant la fonction `memcpy()`, copiez les cinq derniers éléments de `tab` (la chaîne de caractères `fghij`) dans les cinq derniers éléments de `big_tab`. Affichez le contenu de `big_tab` afin de vérifier votre appel à `memcpy()`.

6) Toujours avec la fonction `memcpy()`, copiez les éléments d'*indices* 3 à 7 de `str` dans les éléments d'*indices* 3 à 7 de `big_tab` (documentez vous sur la méthode `c_str` de la classe `std::string`). Affichez à nouveau le contenu de `big_tab` afin de vérifier votre appel.

EXERCICE 2 ▶ Gestion des erreurs

Nous avons vu en CM qu'il était important de vérifier la valeur de retour des appels système afin de ne pas avoir de mauvaise surprise. Dans cet exercice, nous allons illustrer comment traiter cette valeur de retour via la manipulation de la variable `errno`.

La variable `errno` est une **variable entière globale qui est automatiquement définie pour vous dans vos programmes**, à l'inclusion du fichier `errno.h` ou de tout autre fichier déclarant des fonctions POSIX. Comme nous allons le voir, cette variable permet d'être informé de la cause des erreurs des appels systèmes.

1) Nous vous fournissons un programme `errno.cpp`, que vous devez compiler en un exécutable `errno`. Ce programme comporte deux « problèmes » : il tente d'ouvrir en lecture un fichier qui n'existe pas avec un appel à `open()`, puis d'écrire dans un descripteur de fichier qui n'est pas défini. À l'exécution, quel est l'effet produit par ces deux problèmes? Comment pouvez-vous détecter (sans modifier le programme) qu'il y a eu deux erreurs à l'exécution?

2) En tant que programmeur ou programmeuse, quel type d'information souhaiteriez vous avoir si dans votre programme vous tentiez par inadvertance d'**ouvrir** un fichier qui n'existe pas (ou dont l'accès vous est interdit ou impossible)? Consultez la page de manuel de la fonction `open()` (à partir de la section `RETURN VALUE`), et expliquez comment détecter cette situation dans un programme. Quelle valeur prend la variable `errno` dans ce cas?

3) Que devrait-il se passer si on tente de faire une **écriture** dans un fichier qui n'est pas ouvert? Comment détecter cette situation, et quelle valeur prend `errno` dans ce cas?

4) Il n'est pas toujours commode, ni utile, de chercher la signification des valeurs de la variable `errno` dans les pages du manuel... Par contre, la fonction `strerror()` permet de traduire ces valeurs en des messages lisibles. Utilisez la ligne
`std::cerr << "Erreur : " << strerror(errno) << std::endl;`
pour afficher des messages d'erreurs dans le programme fourni : quels sont les messages d'erreur affichés?

Dans les TP suivants (et pour les prochaines UEs « système » que vous suivrez) prenez le réflexe de vérifier le retour de fonction quand vous faites un appel système! Cela vous aidera à déboguer votre code!

EXERCICE 3 ▶ Écriture dans un fichier

Le but est de manipuler les fonctions `open()`, `write()` et `close()` pour écrire dans un fichier. Pour rappel, dans un shell :

- `cat mon_fichier` affiche le contenu d'un fichier la sortie standard,
- `ls -l mon_fichier` affiche les droits d'un fichier (et d'autres informations),

1) Le fichier `writing.cpp` contient le canevas permettant d'ouvrir le fichier `test.txt`. Regardez le sens des paramètres passés à l'appel système `open()` dans la page de manuel. A quoi servent chacun des flags `O_WRONLY`, `O_CREAT` et `O_TRUNC` ? À quoi servent les modes `S_IRUSR` et `S_IWUSR`? Que se passerait-il si on ne donnait pas le mode `S_IRUSR`?

2) Complétez le code de `writing.cpp` pour tenter d'écrire avec un seul appel à la fonction `write()` tous les caractères contenus dans le tableau `tab`. N'oubliez pas de fermer ensuite votre fichier. `g++ -Wall reading.cpp -o reading` permet de compiler votre programme. Exécutez votre programme puis vérifiez le contenu du fichier créé.

3) Sous la commentaire « `deuxième ouverture et écriture dans le fichier` », ajouter le code nécessaire pour ouvrir à nouveau le fichier `test.txt` en écriture, et y ajouter la chaîne de caractères « `Toto fait du vélo !!!\n` ». Attention, cette nouvelle chaîne doit être ajoutée sans écraser la précédente : veillez à utiliser les bons flags lors de l'ouverture du fichier. N'oubliez pas de fermer votre fichier après utilisation.