TP 3
Manipulation de fichiers (1h30)

Trucs et astuces pour accéder à l'historique du shell :

- la flèche du haut ↑
- la commande `history`
- (Ctrl + R) : commencez par taper le début de la commande recherchée, puis utilisez Ctrl + R et Ctrl + S pour parcourir les commandes correspondantes dans l'historique.

Archive pour le TP : récupérez, sur la page web du cours, l'archive contenant les squelettes de codes à remplir. Enregistrez l'archive dans votre répertoire de l'UE (`~/LIFSE/` par exemple). Pour désarchivez, utilisez la commande `tar -xvzf lifse-tp03.tgz` depuis la ligne de commandes.

EXERCICE 1 ▶ Paramètres de la ligne de commande

En C/C++, on peut récupérer les paramètres passés de la ligne de commande, au programme, *via* les paramètres de la fonction `main`, dont le prototype peut être :

```
int main(int argc, char **argv);
```

ou encore

```
int main(int argc, char *argv[]);
```

Dans les deux cas, `argc` donne le nombre d'arguments entrés sur la ligne de commande **plus un** (le nom de l'exécutable compte pour un). `argv` est un tableau de chaînes de caractères :

— `argv[0]` contient le nom de l'exécutable,

— `argv[1]`, ..., `argv[argc - 1]` contiennent chacun des arguments passés à la ligne de commande.

On vous fournit un squelette de code appelé `paraldc.cpp`.

1) Complétez le `Makefile` fourni pour qu'il vous permet de compiler `paraldc` à partir de `paraldc.cpp` quand vous entrez `make paraldc` sur la ligne de commande ; n'oubliez pas de mettre à jour la règle `clean` du `Makefile`.

2) Comment se comporte le programme fourni ? Testez avec différents arguments dans la ligne de commande :

```
$ ./paraldc
$ ./paraldc toto
$ ./paralcd titi toto 42
```

3) Modifiez le programme pour qu'il n'accepte que les lignes de commande avec un seul argument ; si cela n'est pas le cas, le programme affiche un message d'erreur sur la **sortie d'erreur standard** :

```
$ ./paraldc titi 65
Il faut exactement un argument après la commande.
```

EXERCICE 2 ▶ Retour sur read-bin et write-bin

Dans cet exercice, nous revenons sur les programmes `read-bin` et `write-bin` vus lors du TD1 (qui a eu lieu juste avant). Les sources `read-bin.cpp` et `write-bin.cpp` se trouvent dans l'archive fournie. Vous pouvez les compiler à l'aide du `Makefile` fourni avec les commandes `make read-bin` et `make write-bin` respectivement.

1) Exécutez le programme `write-bin`, et essayez de lire le contenu du fichier `test.dat` qui a été généré, en utilisant la commande `cat`. Expliquez le résultat.

2) Exécutez maintenant `read-bin` pour observer les octets un par un. Vous devriez retrouver le résultat du TD.

3) Le programme `read-bin` vous permet d'afficher le contenu de n'importe quel fichier passé sur la ligne de commande (`./read-bin fichier`) et d'afficher le caractère ASCII correspondant à chaque octet si on lui passe un argument de plus (par exemple `./read-bin fichier -`). Essayez les commandes suivantes et expliquez ce que vous voyez :

```
./read-bin test.dat
./read-bin test.dat -
./read-bin read-bin.cpp - | head
./read-bin read-bin.cpp
./read-bin read-bin - | head
```

Pour la dernière ligne, on ne s'intéressera qu'aux 4 premiers octets, et on pourra s'amuser à les retrouver dans la page `https://en.wikipedia.org/wiki/List_of_file_signatures` (en sachant que les fichiers exécutables sous Linux sont au format ELF, Executable and Linkable Format).

4) Dans la vraie vie, on n'a pas besoin d'écrire soi-même le programme `read-bin.cpp`, on peut utiliser la commande `hexdump` (pour afficher le contenu d'un fichier en hexa, par défaut par mots de 16 bits), ou `hexdump -C` (pour afficher les caractères un par un, en hexa et en ASCII). Vous pouvez essayer par exemple :

```
hexdump -C test.dat
hexdump -C read-bin.cpp | head
```

EXERCICE 3 ▶ **Une commande** `wc`

Le but est d'écrire un petit programme qui compte le nombre d'octets que contient un fichier.

1) Complétez pour cela le programme `mywc.cpp` qui est fourni dans l'archive du TP.

```
// ...
int main(int argc, char *argv[]) {
  int fd; // descripteur de fichier
  char c; // servira ici de buffer
  int nbrd, nbbytes;

  if(argc < 2) {
    print_usage(argv[0]);
    return -1;
  }
  // TODO HERE
  // open file in read only mode + ...

  return 0;
}
```

Vous pouvez compiler le programme à l'aide de la cible `mywc` du `Makefile` fourni. Vous comparerez votre résultat avec celui de la commande `wc -c` (utilisez `man wc` si vous ne voyer pas de quoi il s'agit).

2) Que se passe-t-il si vous appelez votre programme avec comme paramètre le nom d'un fichier qui n'existe pas ? Mettez en place une gestion de l'erreur, avec un message adéquat (`man open`, `man errno`, `man strerror` ou `man perror`) :

```
if (fd < 0) {...
  ... strerror(errno) ...
}
```

On peut également tester les cas particuliers d'erreur avec `if (errno == ...)` si nécessaire.

3) Pour l'instant, il est possible que votre programme fonctionne correctement en pratique, tout en contenant des bugs qui pourraient se produire dans des configurations particulières.

- Par exemple, il est rare que `read()` renvoie moins d'octets que demandé sauf dans le cas de la fin de fichier, et les erreurs d'entrée sortie ne sont pas censées arriver. Mais ces cas sont possibles en théorie et devraient être gérés par votre programme.
- Pour tester ces cas, nous vous fournissons un fichier `test-read.h` qui remplace `read()` par une version dégradée qui échoue de manière aléatoire. Décommentez la ligne `test-read.h` dans votre code, recompilez et ré-exécutez votre programme.
- Exécutez-le plusieurs fois pour voir comment il se comporte, et si necessaire corrigez les bugs que vous trouvez dans votre programme avec cet outil.
- On pourra par exemple utiliser cette ligne de commande :

  ```
  head -c 10000000 /dev/zero > zero; while ./mywc zero; do : ; done; rm -f zero
  ```