Langage C et Programmation Système
TP n° 3 : Manipulation de fichiers en C

Exercice 1 : Création d'un fichier `Makefile`

Dans cet exercice, nous allons créer un fichier `Makefile` pour compiler le code de l'exercice 1 du TP de la semaine précédente.

1. Mettez dans un nouveau répertoire les fichiers `strlib.h`, `strlib.c` et `myecho.c` et créez un fichier intitulé `Makefile` ayant le contenu suivant (attention à respecter correctement les tabulations) :

```
all: myecho

myecho: strlib.o myecho.o
	gcc -o myecho strlib.o myecho.o

strlib.o: strlib.c
	gcc -o strlib.o -c strlib.c -Wall

myecho.o: myecho.c strlib.h
	gcc -o myecho.o -c myecho.c -Wall

clean:
	rm -rf *.o
```

Dans ce fichier les noms à gauche des deux points (`:`) correspondent aux noms des règles, ce qui se trouve à droite des deux points sont les dépendances nécessaires à la règle et la ligne en-dessous correspond à ce qui est exécuté quand la règle est appelée.

2. Exécutez `make myecho.o`, qu'observez-vous dans votre répertoire ? Exécutez ensuite `make clean`, qu'observez-vous maintenant ? La règle par défaut qu'on appelle en faisant simplement `make` est la règle intitulée `all`. Exécutez-la. Que fait-elle ? Que se passe-t-il si on fait `make` une seconde fois ensuite ? Modifiez le fichier `strlib.c` et réexécutez `make`. Qu'en déduisez-vous ?

3. Le fichier `Makefile` de la question précédente est relativement simple et peut être amélioré pour être plus modulable. En voilà une version améliorée :

```
CC=gcc
CFLAGS=-Wall
EXEC=myecho

all: $(EXEC)

myecho: strlib.o myecho.o
	$(CC) -o $@ $^

strlib.o: strlib.c
```

```
	$(CC) -o $@ -c $< $(CFLAGS)

myecho.o: myecho.c strlib.h
	$(CC) -o $@ -c $< $(CFLAGS)

clean:
	rm -rf *.o

mrproper: clean
	rm -rf $(EXEC)
```

En vous servant des informations que vous pourrez trouver par exemple à l'url suivante `http://gl.developpez.com/tutoriel/outil/makefile/`, expliquez la signification des variables `$@` `$<` et `$^`. Comprenez comment est construite la règle `strlib.o`.

4. On remarque que dans la version précédente les règles `strlib.o` et `myecho.o` sont similaires. Il est possible de faire une règle générique pour générer les fichiers `.o`. En vous aidant du tutorial dont l'url a été proposée auparavant, modifiez le fichier `Makefile` pour inclure une règle générique. Faîtes attention aux dépendances de `myecho.o`.
5. Par la suite, il vous sera toujours demandé de compiler votre code via un tel fichier `Makefile`.

Exercice 2 : Manipulation de fichiers

1. Pour la manipulation de fichiers, on va se servir des fonctions `fopen`, `fclose`, `fputs` et `fgets` de la librairie `stdio.h`. Créez un fichier `test.txt` en y mettant un texte quelconque et retirez les droits en lecture pour tout le monde. Essayez d'ouvrir ce fichier dans un programme en utilisant `fopen` avec l'option `"r"` et testez-si la valeur retournée est `NULL`. Que concluez-vous ?
2. Dans le programme précédent, aucun message d'erreur n'a été affiché par le système, cependant en lisant la page du manuel de `fopen`, on voit qu'en cas de problème, la variable globale `errno` reçoit une valeur. En utilisant, la fonction `perror` affichez le message d'erreur correspondant.
3. Écrivez un programme `fill.c` qui attend en argument un nom de fichier et un chiffre entier `n`, crée le fichier correspondant et le remplit de `n` lignes avec un message choisi par vous même sur chaque ligne. (Allez voir la page de manuel de `atoi`). Que se passe-t-il si vous appelez deux fois votre programme avec le même nom de fichier ?
4. Comment modifier votre programme pour que si le fichier donné en argument existe déjà, les lignes soient ajoutées à la fin ?
5. Dans un ficher `fillib.c`, écrivez une fonction `int copy(FILE *fsrc, FILE *fdst)` qui prend deux descripteurs de fichiers et copie le contenu du premier dans le deuxième. (N'oubliez pas de créer le fichier `fillib.h` correspondant.)
6. Créez un programme `mycp.c` qui prend en argument deux noms de fichiers et copie le premier dans le deuxième (il faut bien entendu utiliser la fonction précédente). Vérifiez que `./mycp` se comporte comme `cp` (sans options, avec deux noms de fichiers en argument).

7. Créez un programme `mycat.c` qui prend en argument un nombre arbitraire de noms de fichier et affiche le contenu des fichiers correspondants sur la sortie standard (l'un après l'autre, sans séparateur). Si aucun argument n'est donné, le programme affiche le contenu de `stdin`. (Pensez à utiliser votre fonction `copy`.) Vérifiez que `./mycat` se comporte comme `cat` (sans options).

8. Dans le fichier `fillib.c`, écrivez une fonction `int nblines(FILE *f)` qui renvoie le nombre de lignes dans le fichier donné en argument (sachant qu'une ligne finit par le caractère `'\n'`). Testez votre fonction avec un fichier ayant de "grandes" lignes.

9. Comment se comporte la fonction précédente si on lui donne comme argument le descripteur de fichiers de l'entrée standard ?

10. Dans `fillib.c`, écrivez une fonction `int copylines(FILE *fsrc, FILE *fdst)` qui prend deux descripteurs de fichiers et copie le contenu du premier dans le deuxième en ajoutant le numéro de la ligne à chaque fois, mais ce uniquement pour les lignes non-blanches.

11. Modifiez votre programme `mycat.c` pour que si on lui donne l'option `-b`, il se comporte comme la commande `cat -b`.

Exercice 3 : Miroir

Le but de cet exercice est d'écrire un programme qui prend en argument un nom de fichier (par exemple `fichier.txt`) et crée un fichier où tout le texte est inversé (le dernier caractère devient le premier, l'avant dernier le second, etc). On voudra de plus que le fichier en entrée ait l'extension `.txt` et que le nom du fichier en sortie soit l'inverse du nom du fichier en entrée (sans prendre en compte l'extension `.txt`).

1. Écrire une fonction `int reverse(char *s)` qui prend en argument une chaîne de caractères et l'inverse.

2. En utilisant les pages manuel du système UNIX, décrivez ce que fait le programme suivant :

```
#include <stdio.h>

int main(){
  FILE *f=fopen("fic.txt","r");
  fseek(f,0,SEEK_END);
  long t=ftell(f);
  printf("%ld \n",t);
  fseek(f,-5,SEEK_END);
  char s[4];
  fgets(s,4,f);
  printf("%s\n",s);
  return 0;
}
```

3. Testez le programme précédent avec un fichier écrit avec une ligne contenant `0123456789\n`. Le programme a-t-il bien le comportement que vous attendiez ?

4. Écrivez le programme qui réalise le code décrit au début de l'exercice. Il est déconseillé de lire (ou d'écrire) caractère par caractère dans les fichiers.

Exercice 4 : Cadavre exquis

1. Créez un programme qui prend en argument deux noms de fichier avec l'extension `.txt` (par exemple `fichier1.txt` et `file2.txt`) et qui écrit dans un troisième fichier (dans notre cas `ffiiclheier12.txt`), mélangera le contenu en alternant entre les mots du fichier 1 et les mots du fichier 2 (on supposera ici que ce qui sépare les mots sont les espaces et les caractères de retour à ligne ; ainsi un point encadré par deux espaces sera considéré comme un mot). Le nom du fichier sera généré d'une façon similaire à l'exemple. On supposera que les noms de fichiers ne font pas plus de 10 caractères et que les fichiers ne contiennent pas de mots de plus de 20 caractères. Si l'un des deux fichiers a moins de mots que l'autre, alors on finira de remplir avec uniquement les mots du fichier le plus long.
2. Testez votre programme avec les deux fichiers `bao.txt` et `rv.txt` qui se situent dans le répertoire `/ens/sangnier/sysc/tp3/exemples/` et que vous copierez dans votre répertoire de travail.
3. Étendez votre programme pour qu'il puisse prendre entre deux et dix noms de fichier en entrée.