Les opérations E/S sur les fichiers

Introduction

L'objectif du TD est de rappeler les opérations d'E/S en C pour manipuler des fichiers. En particulier, nous allons utiliser les fonctions bas niveau : `open`, `read`, `write`, `close` et `lseek`.

Comme application, vous allez développer un système de cryptage de message texte. Ainsi, il sera possible d'envoyer un message texte crypté à un destinataire avec le risque qu'il soit intercepté sur le canal de transmission. Cette application devra rendre la tâche de décodage du message facile pour le récepteur légitime et 'assez' complexe l'intercepteur illégitime.

Rappels sur les fonctions E/S

Voici pour rappel les fonctions élémentaires de manipulation des fichiers:

- open : permet l'ouverture d'un fichier à partir d'un chemin; cette fonction renvoie un descripteur qui identifie le fichier ouvert;
- read : permet de lire des données avec déplacement du curseur de lecture;
- write : permet d'écrire des données dans le fichier avec avancement du curseur;
- lseek: permet de mettre le curseur de lecture/écriture à une position déterminée:
- close: ferme le descripteur de ficher;

TODO

- Consultez les pages `man` des fonctions citées

Rappels sur la compilation séparée en C

Le code `C` d'une application peut être écrit dans plusieurs fichiers sources (extension .c) et en-têtes (extension .h)

Vous pouvez inclure le contenu d'un fichier dans un autre fichier en utilisation la macro `#include`.

Généralement, vous allez écrire les déclarations des fonctions dans un fichier en-tête (.h) et inclure ces déclarations dans plusieurs fichiers sources (.c)

`#include` est une macro qui ne fait que copier le contenu d'un ficher dans un autre. Vous pouvez très bien inclure un fichier source (.c) dans un autre fichier source (.c) ; mais cet usage est déconseillé pour l'instant.

Si une variable globale est déclarée dans fichier source et qu'elle est utilisée dans un autre ficher, vous devez la déclarer comme *externe* dans le deuxième fichier. En effet, si nous déclarons la variable globale dans les deux fichiers alors le compilateur va réserver *deux fois* l'espace mémoire pour la même variable et cela provoquera une erreur lors de l'édition des liens.

Pour éviter cela, vous devez informer le compilateur, que votre fichier va utiliser une variable mais que l'espace mémoire est déjà réservé. Il faut dans ce cas, précéder la déclaration de cette variable par le mot clé `extern`

Par example, voici un premier fichier source qui déclare une variable globale:

```
//module1.c
int seed = 0;
void init_seed(int i){
    seed = i;
}
```

Et maintenant, le code d'un autre fichier source qui utilise la variable globale:

```
//module2.c
extern int seed ; // Notez le mot cle extern
int nextInt(){
    seed = 8253729 * seed + 2396403;
    return (seed  % 32767);
}
```

Les deux fichiers `module1.c` et `module2.c` peuvent être compilés séparement. De plus, lors de l'édition des liens les deux fonctions vont utiliser *la même variable* `seed`

Cryptographie par substitution

Nous allons développer un système de cryptographie par substitution. Le plus célèbre de ces systèmes, fut celui utilisé par Jules César pour envoyer ces instructions aux différentes provinces.

Le principe est simple: nous disposons d'une palette, c'est un tableau qui contient toutes les lettres utilisables. Pour écrire un message nous allons introduire un décalage dans la palette et pour lire un message nous utiliserons un décalage inverse.

Voici un exemple d'une palette sur laquelle nous avons réalisé un décalage de `+2`:

a	b	c	d	e	f	g	h	i	j	k	l
x	z	a	b	c	d	e	f	g	h	i	j

Pour transmettre le mot `cafe` nous devons écrire le mot `echg` (lire le caractère sur la ligne du bas et écrire celui de la ligne du haut). Et inversement, à la réception de `echg`, nous allons écrire `cafe` (lire le caractère sur la ligne du haut et écrire celui de la ligne du bas)

Ce système est trop simple et ne résiste pas à une cryptanalyse par analyse des fréquences. En effet, la fréquence d'une lettre dans un texte va nous donner une idée sur le vrai caractère utilisé...

Nous proposons pour ce TP, d'améliorer ce système élémentaire en deux étapes:

1. Premièrement nous allons diviser le texte originel en plusieurs messages. Chaque nouveau message va représenter une lettre de la palette et donnera toutes les coordonnées de cette lettre dans le texte.
2. Dans un deuxième temps, nous allons procéder à plusieurs décalages générés par une séquence aléatoire connue par l'émetteur et le récepteur du message.

Nous pouvons ainsi améliorer le système de base car:

- Si un message est intercepté alors celui-ci ne représente qu'une lettre du texte; il sera alors difficile de retrouver le texte originel uniquement à partir des positions de cette lettre.
- Avec les décalages multiples nous allons introduire du bruit qui va rendre la cryptanalyse par fréquence plus difficile.

Première Etape: Projection d'un fichier texte sur une palette

Dans le fichier `etape1/palette.c` nous vous proposons une palette pour écrire les messages. Elle se compose des 26 lettres de l'alphabet; des signes de ponctuations; et de l'espace.

Votre objectif dans cette premiere partie, est de lire le contenu d'un fichier texte et de créer pour chaque élément de la palette un fichier spécifique qui contenient toutes les coordonnées de ce caractère dans le message.

Exemple:

Si le message est:

```
Hello !
```

Votre programme devra produire les fichiers binaires suivants en sortie:

- nom de fichier : h.in

`00000000`

- nom de fichier : e.in

`00000001`

- nom de fichier : l.in

  `0000000200000003`

- nom de fichier : o.in

`00000004`

- nom de fichier :space.in

`00000005`

- nom de fichier : exclamation.in

`00000006`

Nous allons envoyer ces différents fichiers à notre déstinataire par des messages séparés. Il devra reconstuire le texte originel en placant les caractères dans les positions indiquées.

A faire:

- Consultez les fichiers suivants: `commun/td3.h`, `commun/palette.c`, `commun/handlers.c`
- Que contient la variable `HANDLER_TABLE` ?
- Comment pouvons nous utiliser un handler de cette table ?

A faire:

Implémentez la fonction `void codage(int in)` qui permet de coder un message sur plusieurs fichiers de positions des lettres.

Voici pour indication, un algorithme de principe:

```
Procedure codage ( descripteur )
Debut
    position <- 0
    Tantque ( c <- Lecture_Caractere( descripteur ) ) Faire
        c <- miniscule(c)
        index_caracter <- avoir_index_caractere_dans_palette(c)
```

```
        nomfichier <- nom_fichiers[index_caracter]
        out <- ouvrir_fichier(nomfichier)
        ecrire_position( out,  position )
        fermer_fichier(out)
        position++
    Fin Tanque
Fin
```

A faire:

Implémentez la fonction `void decodage(int out)` qui permet de décoder le message en utilisant les fichiers de positions des lettres. Tous les fichiers se trouvent dans le répertoire courant du programme (la fonction `lseek` sera utile ici)

Voici pour indication, un algorithme de principe de la procédure de décodage:

```
Procedure Decodage( descripteur )
Debut
    int i;
    //Boucle sur tous les caracteres de la palette
    Pour i de 0 ... PALETTE_SIZE Faire
        fd <- ouvrir_fichier(NOM_FICHIERS[i])
        Tantque ( position <- Lire_Entier( fd ) ) Faire
            mettre_curseur(descripteur,position);
            HANDLER_TABLE[i](descripteur); // commande d ecriture
        Fin Tantque
        fermer(fd)
    Fin Pour
Fin Procedure
```

Deuxième Etape

Dans la première partie, nous avons realisé un premier programme qui décompose un texte en plusieurs messages où chaque message ne contient que les positions du caractère qu'il représente.

Maintenant, nous allons effectuer des décalages pour ne pas écrire le vrai caractère, mais un autre.

Contrairement au code de César, nous ne souhaitons pas que le décalage soit fixe mais variable. Pour cela, nous allons utiliser un générateur de séquences pseudo-aléatoires dont voici le code:

```
int _seed;
void init_seed(int seed){
```

```
    _seed = seed;
}
int get_random(){
    _seed = 8253729 * _seed + 2396403;
    return (_seed%32767);
}
```

Un générateur peudo-aléatoire est un algorithme qui calcule une suite qui possède certaines propriétés comme le fait que les valeurs ne soient pas toutes concentrées dans un intervalle réduit.

Le code précédent nous donnera donc une suite de nombres et il faut remarquer que pour la même valeur de la graine (seed) nous allons avoir toujours la même séquence.

Donc si l'émetteur et le récepteur partagent ce secret (la graine), alors ils vont avoir la même séquence.

TODO: Le codage

Donnez le nouveau code pour la fonction de codage en modifiant la fonction `codage` de la partie 1.

Vous pouvez utiliser le pseudo code suivant comme guide:

```
Procedure codage ( descripteur )
Debut
    Initialiser_graine(SECRET)
    position <- 0
    Tantque ( c <- Lecture_Caractere( descripteur ) ) Faire
        n <- tirage_aleatoire()
        n <- n % TAILLE_PALETTE

        index_caracter <- avoir_index_caractere_dans_palette(c)

        nomfichier <- avoir_nom_fichier_avec_decalage(NOM_FICHIERS, n)

        ecrire_position( nomfichier , position )

        position++
    Fin Tanque
Fin
```

Le décodage

Donnez le nouveau code pour la fonction de décodage pour un texte qui a subi des décalages successifs.

Indications:

- Vous pouvez décoder le message en utilisant la fonction de décodage de la partie 1 dans un fichier temporaire
- Ensuite, vous allez remplacer les caractères de ce fichier en appliquant des décalages inverses; pour retrouver les décalages de l'émeteur, il suffit de jouer la même séquence aléatoire en utilisant le secret partagé comme graine.

Voici un pseudocode de principe:

```
Procedure Decodage(sortie)
Debut
    decodage_etape1(fichier_temporaire)
    input <- ouverture_fichier(fichier_temporaire)
    Initialiser_graine(SECRET)

    Tantque ( car <-  Lecture_Caractere(input) ) Faire
        index <- index_du_caractere_dans_palette(car)
        decalage <- prochain_entier_aleatoire() % PALETTE_SIZE
        index <-   index - decalage
        Tantque (index < 0) Alors
            index = index + PALETTE_SIZE
        FinTantque
        HANDLER_TABLE[index]( sortie );
    Fin Tantque
Fin Procedure
```