# CATTELIN Céline – SEGUENI Julie

# Rapport de projet – Système d’exploitation

## Choix de conception et découpage du programme

Nous avons choisi de faire plusieurs fichiers pour implémenter le programme. Un fichier fonction.c contenant toutes les fonctions nécessaires au fonctionnement du programme ainsi que son fichier correspondant fonction.h. Le fichier main.c contenant uniquement la fonction main du programme et donc n’appelant que les principales fonctions du programme, et le Makefile contenant les directives de compilation du programme.

Nous avons voulu faire des structures qui permettaient de stocker tous les paramètres de traitement et le contenu des fichiers messages. Elles ont été mises dans le fichier fonction.h.

Le principe de nos structures est l’imbrication entre elles afin de simplifier la transmission des données entre les processus : on inclut un tableau de caractères dans une structure mot, puis on inclut un tableau de structures mots dans une structure message et enfin on inclut un tableau de structures messages dans la structure traitement.

On ne se sert donc ainsi que d’une seule structure traitement contenant alors toutes les données nécessaires au fonctionnement du programme et ce, de façon structurée.

A chaque processus fils créé, on assignera un message à traiter. Et à chaque thread créé on associera un mot à traiter.

Cette méthode permet de simplifier les accès aux données et leurs transmissions même si les structures sont assez lourdes. Notons qu’on ne modifie jamais les fichiers d’origine car ils nous sont utiles dans leur intégrité lors de la phase de retour des messages après leur traitement par les threads.

Les tâches ont été réalisées équitablement :

* Nous avons créé les structures de données ensemble.
* Julie a fait l’extraction des données (fonctions d’affectation aux structures) et la création des processus chefs d’équipe et des threads.
* Nous avons fait la fonction de thread ensemble.
* Céline a fait la partie de traitement des données (fonctions de cryptage et de décryptage des mots et des caractères appelées par chaque thread) et le début du retour des données.
* Julie a fait le reste de la partie du retour des données en fonction de mode de traitement des données.
* Céline a fait la relecture du programme et les premiers tests généraux du programme, le principal de la documentation doxygen de chaque fichier du programme et le rapport du projet.
* La vérification du doxygen a été faite ensemble tout comme les tests précis sur les différents fichiers afin de vérifier tous les cas possibles.
* Julie s’est occupée de la libération de la mémoire allouée.
* Nous avons effectué ensemble les tests pour détecter les erreurs dans le fichier principal.
* Nous avons tout revérifié et terminé de commenter ensemble lorsque tout nous semblait fini et fonctionnel.

Notre programme est capable de détecter les erreurs sur le fichier qui contient les paths et les données de cryptage. Une seule erreur sur ce fichier est bloquante et fait terminer le programme sans exécution de la suite du programme car les erreurs sont détectées lors de l’extraction des données, donc avant même les créations des processus chefs d’équipes.

Les erreurs reconnues sont :

* Chemin du fichier non valide
* Clé trop grande si supérieure à 99. Cependant on reconnait une clé valide jusqu’à 99 car on fait un modulo 26 si la clé est supérieure à 10. Clé non composée de chiffres.
* Caractère de mode de traitement non valide ou supérieur à 1 caractère.

Nous avons implémenté plusieurs niveau de contrôle dans le programme : contrôle du nombre d’arguments donnés lors de l’exécution du programme, contrôle de la bonne ouverture et de la bonne lecture du fichier principal (fichier donné lors de l’appel du programme par ./cesar fichier), contrôle de la validité des données du fichier principal et contrôle de la bonne ouverture et de la bonne lecture des fichiers messages dont le path est donné dans le fichier principal. Lorsque l’un de ces contrôles détecte une erreur, la mémoire allouée jusque là est libérée et le programme termine sur un affichage dépendant de l’erreur rencontrée.

Pour finir, précisons que les fichiers messages doivent toujours se terminer pas un \n par convention Unix, sinon notre programme compte mal le nombre de mots et de caractères et ne donne pas le bon résultat pour le dernier mot du message.

## Problèmes rencontrés

Nous avons eu beaucoup de problèmes principalement lors de la partie d’exécution des threads avec des soucis liés au mutex et au buffer, et lors de la partie de retour des données après le traitement car les tubes que nous faisions ne fonctionnaient pas. Nous avons alors décidé de trouver une autre méthode pour le retour au processus directeur et de ne pas utiliser de tube.

Nous avons essayé de gérer le buffer et l’accès au buffer comme une section critique mais nous avons eu une boucle infinie. Pour régler ce problème nous avons fini par utiliser toutes les capacités de nos structures et nous avons ainsi calculé l’emplacement d’écriture de chaque thread dans le buffer puisque nous connaissons la taille de chaque mot à traiter.

Pour ce qui est des tubes, comme le processus père attend pendant que le processus fils écrit dans le tube, ils ne s’exécutent jamais en même temps. Ainsi si on utilise un tube nommé on obtient une boucle infinie.

Lors de l’utilisation de tubes anonymes, le processus fils écrit dans le tube anonyme caractère par caractère mais seul le dernier caractère était récupéré par le processus père. Nous aurions pu utiliser un tube par caractère mais cette solution nous semblait aberrante. C’est pourquoi nous nous servons d’un fichier \_decypher qui contient le contenu du buffer après le traitement par les threads. Le fichier\_ decypher est créé et rempli par le processus fils pour chaque message à déchiffrer. Le processus directeur va lire tous les fichiers \_decypher pour tous les fichiers à déchiffrer, afficher leur contenu en respectant le nombre d’espace blancs du fichier original et supprimer le fichier \_decypher après l’affichage.

Quelques petits problèmes lors de chiffrage ou du déchiffrage d’un caractère sont également apparus. Certains caractères ne se codaient pas comme il le fallait après l’addition ou la soustraction de la clé car lorsqu’on ajoutait la clé au code ASCII du caractère, nous avions parfois des résultats sortant de la table ASCII. Nous avons alors géré ce problème par un contrôle systématique pour les minuscules de la valeur du caractère à qui on rajoute la clé. Si cette somme est supérieure à 128 (la dernière valeur de la table ASCII) alors on fait obligatoirement une soustraction par 26 pour obtenir une valeur exploitable. Si la somme est inférieure, alors on fait une soustraction par 26 seulement si on sort de la plage des lettres (majuscules ou minuscules). Pour ce qui est des majuscules, cela n’est pas nécessaire car même en ajoutant une clé de 25, on ne peut pas dépasser 128. Lors du déchiffrage, il n’est pas nécessaire de vérifier qu’on reste bien supérieure à 0 puisque même en retirant une clé de 25 il est impossible de sortir de la table.

## Comment bien exécuter le programme

Il faut commencer par compiler le programme  avec la commande « make cesar ».

Ensuite il faut l’exécuter avec la commande « ./cesar fichier » , fichier étant le fichier principal. Les fichiers \_cypher seront créés dans le même dossier que le fichier message d’origine.

Ainsi, si les fichiers messages sont dans le même dossier que le reste du projet, les fichiers \_cypher qui seront logiquement eux aussi dans ce dossier peuvent être effacés par la commande « make clean » qui supprime également les fichiers .o créés et l’exécutable cesar dans le dossier du programme.