Rapport - Projet systèmes d'exploitation

Sonny Klotz - Younes Ben Yamna - Malek Zemni

29 avril 2016

1 Introduction

Le but de ce programme est d'effectuer une opération donnée *(minimum, maximum, moyenne, somme ou impairs)* sur un ensemble de valeurs répartis dans plusieurs fichiers.
Dans un premier lieu, cette opération sera effectuée localement sur chaque fichier. Dans un second lieu, cette opération sera effectuée sur les résultats des opérations locales aux fichiers pour obtenir un résultat final.
Cette manipulation est donc une réduction à deux niveaux : le travail sera d'abord divisé entre N processus, puis entre M threads pas processus.
La réalisation de cette opération de réduction va nécessiter l'implémentation de deux fonctions principales :

- La fonction **directeur** : effectue la première réduction
- La fonction **chef** : effectue la seconde réduction

La fonction **directeur** va divise le travail entre N processus fils. Chaque processus fils quant à lui va faire appel à la fonction **chef** qui divise le travail entre M threads.
Ces deux opérations de rédution vont etre détaillées dans les parties suivantes.

2 Directeur

La fonction principale **directeur** et toutes ses fonctionnalités sont codées dans le fichier **directeur.c**.

D'abord, la fonction **directeur** crée un fichier *resultats.txt* et y insère le nombre total de fichiers fournis. Ce fichier permet de stocker, ligne par ligne, les résultats locaux des chefs d'équipes (processus). Ce fichier sera traité à la fin pour obtenir un résultat final.
Ensuite, cette fonction crée les processus chef d'équipe. Une première boucle permet de créer autant de processus fils que de fichiers fournis en utilisant l'appel système **fork()**. Ces processus fils lançeront à leur tour la fonction **chef** pour répartir le travail entre les employés (threads).
Enfin, une dernière boucle permet au processus père de se synchroniser et d'attendre la fin de tous ses processus fils en utilisant l'appel système **wait()**.

Remarque : utilisation du fichier resultats.txt au détriment des tubes
Nous aurions pu utiliser des pipes pour communiquer les résultats entres les différents processus fils, mais cela aura nécéssité la reprogrammation de fonctions *minimum, maximum, moyenne, somme ou impairs* simples afin de générer un résultat final. Nous avons préféré stocker le résultat de chaque processus en un seul et unique fichier afin de pouvoir réutiliser la fonction **chef** et ainsi réappliquer l'opération une dernière fois sur ce fichier et obtenir directement un résultat global.

3 Chef

Le deuxième niveau de réduction, c'est-à-dire le partage du travail entre plusieurs employés (threads), va être réalisé à l'aide des fonctions du fichier **chef.c**.

On peut représenter schématiquement son flux d'exécution en 3 étapes :

1. L'initialisation de la structure (argument nécessaire pour appeler l'opération demandée par l'utilisateur)
2. La création des threads, ainsi que leur attente : (fonction *creaEmployes)*
3. Ecriture du résultat local de l'opération dans le fichier *resultats.txt*

La difficulté qu'il va falloir prendre en compte est la gestion de la concurrence. En effet, nous avons deux variables critiques, car nos threads vont vouloir modifier et lire des variables partagées :

— La valeur de retour de la structure, qui est modifiée au fur et à mesure qu'un thread effectue sa tâche
— Le fichier, dont le pointeur de position va être modifié après les multiples lectures

Nous initialisons donc deux mutex dans notre structure pour protéger ces deux variables. Voici la structure utilisée pour stocker les informations concernant un thread :

```
typedef struct
{
  int fd;           //descripteur du fichier sur lequel on
    travaille
  int nb_val;         //nombre de valeurs du fichier
  double *retour;       //stocker le resultat du calcul
  pthread_mutex_t *mut_fic; //pour protéger la lecture du fichier
  pthread_mutex_t *mut_ret; //pour protéger la valeur de retour
} inf;
```

donnees.h

4 Fonctions

Le fichier **fonctions.c** contient d'une part les opérations que va effectuer un thread employé *(minimum, maximum, moyenne, somme ou impairs)*, et d'autre

part, des fonctions, utilisées pour la gestion des fichiers, qui n'ont recourt qu'à des appels systèmes.
Les fonctions représentant les opérations ont toutes le meme schéma : on commence par caster l'argument, puis lire les éléments du fichier (100 au maximum par thread) tout en veillant à le protéger par le mutex *mut_fic**, et finalement, on modifie le résultat du thread qui est lui aussi protégé par le mutex *mut_ret*.

5 Remarques et Conclusion

Gestion d'erreurs :

Pour la plus part des appels systèmes, la gestion d'erreurs a été assurée par la variable globale *errno*, fournie par *errno.h*, dont le contenu a été affiché à l'aide de la fonction *perror*.
Pour les autres fonctions, les messages d'erreurs ont été affichés sur la sortie d'erreur standard *stderr*.

Une erreur étrange à cause d'un mutex* :

Nous avons décidé dans tout le projet de mettre en commentaire toutes les occurences du mutex lié au fichier. Cela est dû à un problème dont nous ignorons la cause. Les deux mutex sont utilisés strictement de la même façon, pourtant, le mutex lié au résultat va toujours fonctionner, alors que le mutex lié au fichier ne fonctionnera que sur certains ordinateurs. Mais le retrait du mutex au fichier ne pose aucun problème, peu importe la machine utilisée. L'erreur produite fait qu'à l'arrivée au mutex du fichier, le programme semble s'arrêter sans quitter. La documentation des mutex nous laisse penser que ce problème est un *deadlock*, mais la résolution de ce type de problème nous est inconnue. Le projet fonctionnant correctement ainsi, nous avons décidé de ne pas approfondir le cas.