TP nº 7: mutex, files, et pointeurs sur tout ou rien

L'objectif de ce TP est de vous familiariser avec les *mutex*, c'est-à-dire des verrous.

Exercice 1. *Vous avez vu ce qu'est un fils et un fil. Voici maintenant une file.*

Le but de cet exercice est de concevoir une file (aussi appelée FIFO [1] ou « queue »), qui support l'insertion d'un côté et la récupération de l'autre.
Voici le fichier de headers `queue.h` à utiliser :

```
struct queue; /* Structure opaque, à définir dans queue.c */

struct queue* queue_new(void);
void queue_free(struct queue *queue);
void queue_push(struct queue *q, void *value);
void* queue_pop(struct queue *q);
void* queue_pop_blocking(struct queue *q);
```

Les questions de code sont incrémentales. Pensez à sauvegarder votre réponse à une question avant de passer à la suivante et de risquer de casser complètement votre code. Utiliser Git est une bonne idée, par exemple.

1. Commencez par implémenter une structure de file mono-thread, qui soit utilisable sans accéder aux champs de la structure (ie. on doit pouvoir s'en servir en appelant seulement vos fonctions). La fonction de récupération du dernière élément doit renvoyer `NULL` si la file est vide.
Écrivez un petit programme qui la teste.
2. Quels sont les problèmes que vous allez avoir si nous avons plusieurs threads ? Détaillez les différents cas possibles.
3. Corrigez ces problèmes en utilisant un mutex. Écrivez un petit programme qui teste votre nouvelle implémentation.
4. Quel est l'inconvénient de n'utiliser qu'un seule mutex pour toute la file ? Corrigez cela, en utilisant plusieurs mutex (deux devraient suffire).
5. Actuellement, la fonction de récupération d'un élément renvoie `NULL` si la file est vide – on dit qu'elle est non-bloquante. Ajoutez une fonction de récupération bloquante, qui attend qu'un élément soir inséré dans la file s'il n'y en a pas. Bien entendu, nous attendons une approche intelligente, qui ne fait pas qu'appeler la précédente en boucle tant qu'elle renvoie `NULL`.
Écrivez un petit programme qui teste votre nouvelle implémentation, et exécutez-le plusieurs fois pour vous assurer que ça marche et que vous n'avez pas seulement eu un coup de chance la première fois.

Exercice 2. *Map-Reduce*

Dans cet exercice, nous allons utiliser la file précédemment créée pour implémenter un système de MapReduce (le principe de fonctionnement de Hadoop).

1. First In First Out [2]
2. Et non pas « les derniers seront les premiers ».

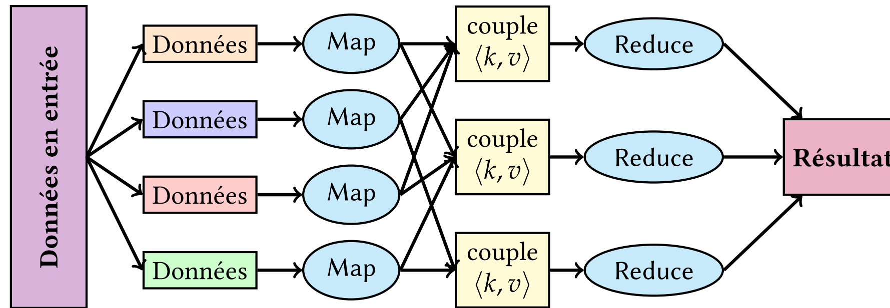

FIGURE 1 – Schéma du principe de MapReduce

Voici le fichier de headers `map_reduce.h` à utiliser :

```
#include "queue.h"
struct key_value_pair { unsigned int key; void *value; };
typedef void (*mapper_t)(void *argument, struct queue *out_q);
/* Quand in_q contient un NULL, le reducer met une valeur dans out_q et s'arrête. */
typedef void (*reducer_t)(unsigned int key, struct queue *in_q, struct queue *out_q);

struct map_worker_args { mapper_t mapper; struct queue *in_q; struct queue *out_q; };
void map_worker(struct map_worker_args *args);

struct reduce_worker_args { reducer_t reducer; unsigned int key;
                            struct queue *in_q; struct queue *out_q; };
void reduce_worker(struct reduce_worker_args *args);

void dispatcher(struct queue *in_q, struct queue **out_qs, unsigned int nb_out);

void** map_reduce(mapper_t mapper, reducer_t reducer,
                  unsigned int nb_keys, void *in[]);
```

Les deux `typedef` sont des types de pointeurs sur fonctions (aussi appelés foncteurs, bien que ça n'ait rien à voir avec les foncteurs de la théorie des catégories ou ceux d'OCaml).

1. Implémentez la fonction `dispatcher` qui prend des paires dans son argument `in_`, lit leur clé `key`, et place leur valeur dans la `key`-ème file de sortie (`out_q`). Elle doit s'arrêter quand elle a dispatché un `NULL` dans la file d'entrée de chacun des map-workers, et ajouter un `NULL` dans chacune des files de sortie. (environ 15 lignes)
2. Implémentez la fonction `map_worker`. Elle appelle la fonction `mapper` pour chacun des éléments de la file d'entrée (il n'ajoute rien dans la file de sortie, `mapper` le fait lui-même). Elle doit s'arrêter quand elle rencontre un pointeur `NULL` dans `in_q`, et pusher un `NULL` sur sa file de sortie. (environ 5 lignes)
3. Implémentez la fonction `reduce_worker`, qui se contente d'appeler `reducer` avec les bons arguments. (environ 2 lignes)
4. Implémentez la fonction `map_reduce`. Elle s'occupe de créer les files nécessaires, créer les threads en leur donnant les bons paramètres, et attendre tous les threads aient fini de s'exécuter, puis récupérer tous les résultats de réduction pour les mettre dans un tableau de sortie de taille `nb_keys` (en l'allouant au passage).
   N'oubliez pas qu'elle doit aussi s'occuper d'ajouter des pointeurs `NULL` dans certaines files pour que tout se termine bien. (environ 60 lignes)
5. Écrivez un programme qui se sert de `map_reduce` pour la tester. Il devra compter le nombre d'occurrences de chaque lettre dans un gros texte donné sur stdin (une variante du *Word Count*[3]). (environ 70 lignes).

3. Il y a un joli schéma ici : https://dzone.com/articles/word-count-hello-word-program-in-mapreduce