**Sorbonne Université**

**LIP6**



# Introduction aux threads et à la synchronisation

## 1 Introduction

Le but de ce TP est de regarder les techniques de base utiles pour la synchronisation entre les threads, à travers une application appelée histogramme.

## 2 Premier programme multi-threads

Dans un fichier `exo1.c`, déclarez deux variables globales `char buffer0[128]` et `char buffer1[128]`, puis les variables nécessaires pour implémenter manuellement une barrière à deux threads. Écrivez ensuite deux fonctions :

- Une première fonction, `f0`, qui écrira dans la variable `buffer0`, attendra sur la barrière, puis lira et affichera le contenu de `buffer1`
- Une deuxième fonction, `f1`, qui écrira dans la variable `buffer1`, attendra sur la barrière, puis lira et affichera le contenu de `buffer0`

Enfin, écrivez un main qui effectue une création de thread sur la fonction `f1` (à l'aide de la fonction `pthread_create`), puis exécute la fonction `f0`, avant d'appeler la fonction `pthread_join`.
Compilez et exécutez votre programme, et faites vérifier votre programme avant de passer à la suite.

## 3 Application histogramme

L'application histogramme permet, pour une image donnée, de mesurer la fréquence des valeurs pour chacune des composantes rouge, verte et bleue.
Commencez par récupérer l'archive suivante :
`https://www-soc.lip6.fr/~meunier/cours/hpc_tp_threads.tar.gz`
Le but du tp est d'écrire plusieurs versions parallèles de la fonction principale effectuant le calcul de l'histogramme. Le code séquentiel de l'application vous est donné (fonction `calc_seq`). Cette fonction est appelée avec comme paramètre un pointeur vers une structure `thread_arg_t` décrivant le traitement complet de l'image, et contenant pour les champs `red`, `gre` et `blu` des pointeurs vers les tableaux de résultats de référence. Ces tableaux de référence seront ensuite utilisés pour vérifier que les différentes versions parallèles à écrire (`calc_par_x`) produisent le bon résultat.

**Note :** Vous pouvez visualiser le résultat de l'application en l'exécutant avec l'option `-d`, qui va générer un fichier `data.txt`, puis en entrant les commandes suivantes dans `gnuplot :`

- `plot 'data.txt' using 1:2 lc rgb 'red'`
- `replot 'data.txt' using 1:3 lc rgb 'green'`
- `replot 'data.txt' using 1:4 lc rgb 'blue'`

**Note :** Pour simplifier les choses, tout le code se trouve dans un seul fichier, mais dans le cas d'un développement réel, il faut bien sûr faire un découpage intelligent du contenu.

### 3.1 Fonction calc_par_0

Déclarez une variable globale de type `pthread_mutex_t` et initialisez-la correctement dans le `main()`. La fonction `calc_par_0` devra prendre le lock au début de chaque itération et le relâcher à la fin de chaque itération. Vous veillerez par ailleurs à récupérer le bon argument par thread : l'idée des macros utilisées pour

la création et destruction des threads est qu'elles sont indépendantes de l'application. Le paramètre passé à la fonction est donc un pointeur vers un entier qui est le numéro du thread `thread_id`. À partir de ce numéro, il faut donc récupérer la structure `thread_arg_t` correspondant au thread (`args[thread_id]`). Ces structures ont déjà été initialisées pour chaque thread dans le `main()`.
Compilez et testez votre programme, avec 1 puis 2 threads (option `-n 2`).

### 3.2 Fonction calc_par_1

Dans cette version, déclarez trois locks (un par tableau) et ne lockez que celui nécessaire pour chaque composante.
Compilez et testez votre programme, avec 1 puis 2 threads.

### 3.3 Fonction calc_par_2

Dans cette version, déclarez des tableaux de locks, de manière à ce qu'il y a ait un lock par élément pour chaque tableau. La fonction de calcul ne doit locker que le lock correspondant à l'élément qui est modifié.
Encore une fois, compilez et testez votre programme, avec 1 puis 2 threads.

### 3.4 Fonction calc_par_3

Dans cette version, remplacez les `mutex_locks` de la version précédente par des `pthread_spinlock_t` (regardez si besoin dans le cours).
À nouveau, compilez et testez votre programme avec 1 et 2 threads.

### 3.5 Fonction calc_par_4

Dans cette version, utilisez la macro `cas` définie en haut du fichier (essayez de comprendre sa valeur de retour) ou la fonction builtin de gcc `__atomic_add_fetch()` pour réaliser l'incrémentation de chaque composante.
Compilez et testez votre programme, avec 1 puis 2 threads.

### 3.6 Fonction calc_par_5

Dans cette version, déclarez à l'intérieur de la fonction 3 tableaux locaux dans lesquels vous stockerez les résultats du calcul pour le thread. Dans une deuxième phase, vous effectuerez la mise à jour des tableaux globaux à partir des valeurs locales.
Compilez et testez votre programme.

### 3.7 Fonction calc_par_6

Il est possible de ne pas utiliser de mécanisme de synchronisation, en partageant le travail à faire entre threads selon les données écrites uniquement. Décrivez le principe d'un tel programme, puis implémentez-le. Notez que cela nécessite de modifier la structure décrivant le travail par thread.
Compilez et testez votre programme.

### 3.8 Synthèse des résultats

— Comment interprétez-vous les différents résultats que vous obtenez ?
— Quels facteurs peuvent-ils expliquer ces performances ?
— Quelles sont les limites à la parallélisation ?
— Qu'en concluez-vous sur la synchronisation entre threads ?