TP 6 – *Threads*, *mutex*

Ce sujet de TP est optionnel.

Il est toutefois fortement recommandé de faire au moins l'exercice 1 si vous envisagez de suivre le parcours *Calcul Intensif et Données Massives* (CIDM) ou le parcours *Génie Logiciel* (GL) l'an prochain.

Exercice 1 - pthreads

Le but de cet exercice est de reprendre ce qui a été fait dans l'exercice 5 du TP 4, et de proposer une nouvelle variante reposant cette fois sur des *pthreads* (à la place des processus lourds créés par `fork`).

1. Ecrivez la variante `wc-par-v4.c`, dans laquelle le père crée les 4 threads. Chaque thread va compter et renvoyer le nombre de sauts de lignes dans la partie du fichier qu'il doit traiter.

   Le père attend la terminaison des threads, récupère les différentes valeurs renvoyées, et affiche leur somme sur le flux de sortie standard.

aide :

Le troisième argument attendu par `pthread_create` est une fonction de prototype

```
void* start_routine(void*)
```

c'est-à-dire une fonction qui :

- prend en entrée un pointeur vers une zone mémoire contenant les arguments,
- renvoie un pointeur vers la zone mémoire où le résultat a été stocké.

Le type `void*` est utilisé ici comme *joker*, puisqu'il n'est pas possible de savoir à l'avance quels vont être les besoins en terme de types pour la routine choisie par l'utilisateur.

Pour gérer les arguments des différents threads, une approche simple est de définir une structure (type record) genre

```
struct arg {
    char* file;
    int begin;
    int end;
};
```

On peut alors lancer les quatre threads de la façon suivante :

```
pthread_t thids[4];
struct arg args[4];

for (int i = 0; i < 4; ++i) {
    args[i].file  = argv[1];
    args[i].begin = ... ;
    args[i].end   = ... ;
    pthread_create(&thids[i], NULL, start_routine, &args[i]);
}
```

Enfin, pour la valeur de retour, on veillera à allouer la mémoire nécessaire via un appel à `malloc` dans le code de `start_routine`. Une fois que le père aura récupéré cette valeur de retour, il devra libérer la mémoire via un appel à `free`.

2. Ecrivez la variante `wc-par-v5.c`, dans laquelle le père crée les 4 threads. Chaque thread va :
   — compter le nombre de sauts de lignes dans la partie du fichier qu'il doit traiter,
   — mettre à jour un compteur global.

   Le père attendra la terminaison des threads qu'il a lancé, puis affichera la valeur finale du compteur.

   On utilisera un sémaphore (type `pthread_mutex_t`) pour s'assurer que le compteur global n'est modifié que par un seul thread à la fois.

Exercice 2 - processus, threads, `time`

Cet exercice porte sur les programmes `runcmd-fork` et `runcmd-thread`, dont les sources sont disponibles dans `/pub/FISE_OSSE11/syscall/runcmd-fork.c` et `/pub/FISE_OSSE11/syscall/runcmd-thread.c`.

Ils sont fonctionnellement semblables :

1. Ils allouent n mégaoctets de mémoire (n étant la valeur du 1er argument passé au programme) ;
2. Ils répètent 1000 fois la suite d'actions suivante :
   — création d'un fils qui affiche `hello` sur la sortie standard
   — attente de la fin du fils.

Comme les noms le suggèrent, ces deux programmes diffèrent par leur implémentation :

— runcmd-fork crée ses fils (processus lourds) avec `fork`,

— runcmd-thread crée ses fils (processus légers) avec `pthread_create`.

1. Compilez les deux programmes.

```
sh$ gcc -Wall -o runcmd-fork /pub/FISE_OSSE11/syscall/runcmd-fork.c
sh$ gcc -Wall -pthread -o runcmd-thread /pub/FISE_OSSE11/syscall/runcmd-thread.c
```

2. Mesurez les temps d'exécution des deux programmes avec une mémoire allouée de 1 Mo.

```
sh$ time ./runcmd-fork 1 >/dev/null
sh$ time ./runcmd-thread 1 >/dev/null
```

   On redirige ici le flux standard de sortie dans `/dev/null` afin de ne pas mesurer le temps lié à l'affichage (qui n'est pas du tout négligeable, et qui n'apporte rien pour la suite).

3. Recommencez ces mesures pour une mémoire allouée de 10, 100, 200, 500 et 1000 Mo.

4. Expliquez les mesures obtenues.

Exercice 3 - pthreads, mutex

Le programme `hello` a pour but d'écrire le message `"hello"` sur le flux standard de sortie. Pour cela, il crée deux threads :

- Le premier thread écrit `'h'` et `'o'`.
- Le deuxième thread écrit `'ell'`.
- Le programme principal écrit le `'\n'` final.

1. Écrivez le code `hello_sleep.c`, dans lequel la synchronisation des écritures est réalisée grâce à des appels à `sleep`.

2. Écrivez le code `hello_mutex.c`, dans lequel la synchronisation des écritures est réalisée grâce à des sémaphores (type `pthread_mutex_t`).