Pour le TP on téléchargera l'archive `TP15.zip`. Cette archive contient 4 fichiers :

- un fichier `TP15.c` qui sera le fichier principal à modifier et celui qui contient la fonction `main` ;
- un fichier `utils.c` qui contient des fonctions utilitaires. Certaines fonctions seront à modifier dans la troisième partie ;
- un fichier d'entête `utils.h` qui contient les définitions des types structurés utilisés et les déclarations des fonctions utilitaires ;
- un fichier `makefile` qui contient les commandes de compilation. On tapera `make build`, `make run` ou `make all` dans une console (en s'étant placé dans le bon répertoire) pour exécuter l'une des commandes. Elles correspondent à une commande pour compiler, une commande pour exécuter et une commande pour faire les deux à la fois.

On rappelle les fonctions pour manipuler les fils d'exécution et les mutex en C :

- `pthread_t fil;` pour créer un fil d'exécution ;
- `pthread_create(&fil, NULL, f, arg);` pour lancer un fil d'exécution qui calcule `f(arg)`. `f` doit être une fonction qui prend en argument un pointeur et renvoie un pointeur (généralement `NULL`), `arg` doit être un pointeur. Dans la fonction `f`, on doit convertir ce pointeur en lui redonnant le bon type si nécessaire ;
- `pthread_join(fil, NULL);` attend la fin de l'exécution d'un fil ;
- `pthread_mutex_t verrou;` pour créer un mutex ;
- `pthread_mutex_init(&verrou, NULL);` pour initialiser un mutex ;
- `pthread_mutex_destroy(&verrou);` pour détruire (libérer la mémoire allouée) un mutex ;
- `pthread_mutex_lock(&verrou);` pour verrouiller un mutex ;
- `pthread_mutex_unlock(&verrou);` pour déverrouiller un mutex.

1 Prise en main

On souhaite réécrire les exemples du cours, avec deux fils qui exécutent de manière concurrente l'algorithme suivant, où N et *cpt* (pour compteur) sont des variables globales :

Fonction `Incrémenter(`i`)`
  Pour k de 1 à N **Faire**
    $cpt \leftarrow cpt + 1$.

Ici, l'argument i est le numéro du fil qui fait l'incrémentation. Cet argument sera utile plus tard.

1. Écrire une fonction `void* incrementer1(void* arg)` qui réalise cet algorithme et renvoie un pointeur `NULL`.
2. Dans le `main`, créer deux fils d'exécution qui exécutent cette fonction, puis afficher la valeur du compteur `cpt`.
3. Exécuter le code, en modifiant la variable globale `N`. Que constate-t-on ?
4. Écrire une fonction `void* incrementer2(void* arg)` similaire à la précédente, mais qui effectue l'incrémentation de manière protégée par le mutex global `verrou`.
5. Tester l'exécution avec deux fils d'exécutions et afficher le résultat du compteur.

2 Algorithme de Peterson

On souhaite implémenter l'algorithme de Peterson pour un mutex à deux fils. On utilise pour cela le type défini par :

```c
struct Mutex {
    bool attente[2];
    int tour;
};

typedef struct Mutex mutex;
```

Les deux premières fonctions sont à modifier dans le fichier `utils.c`.

6. Écrire une fonction `void deverrouiller(mutex* m, int i)` qui prend en argument un pointeur de mutex et un numéro de fil valant 0 ou 1 et déverrouille le mutex pour ce fil selon l'algorithme de Peterson.
7. Écrire une fonction `void verrouiller(mutex* m, int i)` qui prend en argument un pointeur de mutex et un numéro de fil valant 0 ou 1 et verrouille le mutex pour ce fil selon l'algorithme de Peterson.
8. Écrire une fonction `void* incrementer3(void* arg)` qui prend en argument un pointeur vers un entier valant 0 ou 1 et fait l'incrémentation décrite dans la partie précédente avec une protection par le mutex de Peterson global `m` (déjà créé dans le fichier).
9. Tester l'exécution avec deux fils d'exécutions avec différentes valeurs de `N`. Que constate-t-on ?
10. Dans le fichier `makefile`, modifier l'option `-O0` en `-O1`, puis `-O2` et reprendre la question précédente à chaque fois. Que constate-t-on ?

Remarque

Cette option permet de modifier l'optimisation du fichier obtenu après compilation : plus le numéro d'optimisation est élevé, plus le programme en résultant sera rapide, mais plus le compilateur s'autorise des raccourcis pour y arriver (et donc s'éloigne du modèle de calcul théorique auquel on est habitué).

L'algorithme de Peterson, qu'on a pourtant prouvé comme correct, ne fonctionne pas comme il devrait. C'est dû au fait que nous avons fait des hypothèses sur les objets manipulés qui ne sont pas vérifiées en pratique. Pour corriger cela, on peut rendre atomiques toutes les lectures et écritures des entiers et booléens.

11. Dans le fichier `utils.h`, modifier les types des champs de la structure `Mutex` en `atomic_bool` et `atomic_int` et reprendre les deux questions précédentes. Que constate-t-on ?

Remarque

On a chargé `stdatomic.h` pour manipuler ces types.

3 Tri rapide concurrent

Dans cette partie, on souhaite paralléliser le tri d'un tableau d'entiers.

3.1 Tri séquentiel

On représente une tranche de tableau par un pointeur vers le premier élément de la tranche et un entier désignant la taille de la tranche :

```c
struct Tranche {
    int* debut;
    int taille;
};

typedef struct Tranche tranche;
```

12. Lire la description des quatre premières fonctions utilitaires dans `utils.c`. On pourra utiliser librement ces fonctions.

13. Justifier pourquoi il ne sera généralement pas nécessaire d'allouer dynamiquement une tranche sur le tas.

    On parle ici de la tranche elle-même, pas du tableau de données auquel elle fait référence.
14. Écrire une fonction `void* tri_insertion(void* arg)` qui prend en argument un pointeur de tranche (qu'il faudra transtyper) et trie **en place** les éléments de la tranche par insertion.
15. Écrire une fonction `int partition(tranche tr)` qui prend en argument une tranche et effectue une partition des éléments qui s'y trouvent selon un pivot (qu'on choisira comme le premier élément), puis renvoie l'indice dans la tranche où le pivot a été placé.
16. En déduire une fonction `void* tri_rapide1(void* arg)` qui prend en argument un pointeur de tranche et la trie par un algorithme de tri rapide séquentiel. On arrêtera les appels récursifs dès qu'on tombera sur une tranche de taille plus petite que la constante littérale `SEUIL_TRI` définie dans le fichier, et on triera ces tranches par insertion.
17. Créer des tableaux aléatoires de tailles 1000 et 10000 et comparer le temps d'exécution pour les trier selon le tri par insertion et le tri rapide. On pensera à libérer et créer de nouveaux tableaux pour éviter de trier deux fois de suite le même tableau.

    On rappelle que l'option de formattage pour l'affichage des flottants est `%lf`.

3.2 Une file de tâches

Pour mettre en place un tri concurrent, on souhaite réaliser une file de tâches (ou de tranches). Le principe est alors le suivant : lorsqu'un fil est disponible, il essaie d'extraire une tâche de la file pour la traiter, et enfile éventuellement de nouvelles tâches.

Une file peut être implémentée efficacement avec une liste simplement chaînée si on dispose de pointeurs vers le début et la fin de la liste. L'idée est alors d'enfiler à la fin et de défiler au début de la liste.

On implémente une file de tâches par :

```
struct Maillon {
    tranche tr;
    struct Maillon* suivant;
};

typedef struct Maillon maillon;

struct File {
    maillon* premier;
    maillon* dernier;
    atomic_int taille_max;
    pthread_mutex_t verrou;
};

typedef struct File file;
```

où si `f` est un pointeur de `file`, alors `f->premier` désigne le premier maillon de la file, `f->dernier` le dernier maillon, `f->taille_max` désigne un **majorant** du nombre de tâches à traiter (en particulier, si `f->taille_max` vaut 0, il n'y a plus de tâches à traiter, mais la file peut être temporairement vide et de nouvelles tâches peuvent être insérées plus tard si `f->taille_max` ne vaut pas 0) et `f->verrou` un mutex protégeant toutes les modifications de la file. Chaque maillon contient une tranche et un pointeur vers le maillon suivant.

Toutes les fonctions de cette sous-partie sont à écrire dans le fichier `utils.c`. Toute modification autre que `f->taille_max` (qui est atomique) se fera en verrouillant le mutex.

18. Écrire une fonction `file* creer_file(void)` qui crée une file vide (donc sans maillon) et renvoie un pointeur vers cette file. On initialisera le mutex et on posera une taille maximale à 0.

19. Écrire une fonction `void liberer_file(file* f)` qui libère une file. On pensera à détruire le mutex.
20. Écrire une fonction `void enfiler(file* f, tranche tr)` qui enfile une tranche dans une file.
21. Écrire une fonction `bool defiler(file* f, tranche* tr)` qui prend en argument un pointeur de file et un pointeur de tranche et :
    - essaie de défiler une tranche de `f` et de recopier son contenu dans la tranche donnée en argument ;
    - si le défilage s'est déroulé avec succès, la fonction renvoie `true`, sinon (la file était vide au moment du défilage), elle renvoie `false`.

3.3 Tri concurrent

22. Écrire une fonction `void traiter_tranche(file* f, tranche tr)` qui effectue le traitement d'une tranche selon le principe suivant :
    - si la tranche est de taille inférieure à `SEUIL_TRI`, on effectue le tri de la tranche par insertion ;
    - sinon, on partitionne la tranche, on enfile dans `f` une tranche correspond à une des deux portions obtenues, et on traite récursivement la deuxième tranche.

    La fonction devra incrémenter ou décrémenter `f->taille_max` selon le cas rencontré.
23. Écrire une fonction `void* traitement(void* arg)` qui prend en argument un pointeur de file (qu'on transtypera avant manipulation) et essaie de défiler et de traiter des tâches de la file tant que le nombre de tâches n'est pas nul.
24. En déduire une fonction `void* tri_rapide2(void* arg)` qui prend en argument un pointeur de tranche et la trie en lançant `NB_FILS` fils d'exécution en parallèle.
25. Comparer les temps d'exécution entre `tri_rapide1` et `tri_rapide2` pour des tableaux de 1 million d'éléments. Est-ce raisonnable ? Est-ce qu'augmenter le nombre de fils améliore le temps de calcul ? Pourquoi ?

3.4 Pour aller plus loin

Dans la méthode précédente, on peut perdre du temps dans l'accès concurrent à la file. On peut faire en sorte que chaque fil possède sa propre liste de tâches. Cependant, si un fil termine plus tôt l'ensemble de ses tâches, on ne met pas à profit ses capacités de calcul.

Pour améliorer cela en faisant en sorte qu'un file « vole » du travail à un autre. Pour mieux répartir le travail, un fil en cours de traitement :

- commence par essayer de traiter une de ses tâches ; s'il y en a, il traite en priorité une des tâches les **plus récentes** (qui sont les plus courtes) ;
- sinon, il essaie d'en voler une à un autre fil, en choisissant en priorité une des tâches les **plus anciennes** (qui sont les plus grosses).

26. Proposer une implémentation qui tri un tableau selon ce principe. On créera un nouveau type de liste, cette fois-ci **doublement chaînée**, avec des ajouts et retraits possibles au début ou à la fin.

Remarque

> Une telle structure est appelée *double-ended queue* ou *deque*.