Rapport de projet - IT202

Département Informatique
S8 - Année 2022/2023

1 Introduction

Ce rapport a pour but de montrer les fonctionnalités mises en place pour réaliser une bibliothèque de gestion des threads. Le sujet est projet est disponible ici.

Nous verrons comment elles ont été implémentées ainsi que les choix d'implémentation. Nous vérifierons que ces fonctionnalités sont bien conformes aux attentes du sujet et nous testerons les performances de la bibliothèque.

2 Fonctionnalités implémentées

Voici les différentes fonctionnalités que supporte la bibliothèque développée. Les choix de réalisation et principes de fonctionnement y sont expliqués.

2.1 Gestion des threads

2.1.1 Structure d'un thread

Le cœur de notre bibliothèque de threads est la structure `thread_c` (figure 1). Cette structure est conçue pour encapsuler toutes les informations nécessaires pour la gestion d'un thread au sein de notre bibliothèque. Chaque instance de cette structure représente un thread individuel. (cf. figure 1)
L'ensemble de ces champs permet de définir un thread de manière unique, et chaque

```
typedef struct thread_c {
    thread_t id;
    ucontext_t context;
    TAILQ_ENTRY(thread_c) entry;
    void *ret;
    enum status state;
    int valgrind_stackid;
    struct join_chain jc;
    struct thread_c* waiting_last_mutex_locked;
    #ifdef STATIC
    char stack[STACK_SIZE];
    #endif
} thread_c;
```

FIGURE 1

champ à sa propre signification :

id : Un identifiant unique pour le thread.

context : Un champ de type `ucontext_t` qui stocke le contexte d'exécution du thread.

entry : Un champ utilisé pour que cette structure puisse être insérée dans une queue de type `TAILQ_ENTRY`.

ret : Un pointeur vers la valeur de retour du thread.

status : Un champ pour stocker l'état actuel du thread.

stack_id : Un champ pour stocker l'identifiant de la pile Valgrind pour le thread. Ceci est utilisé pour éviter les faux positifs de fuites de mémoire lors de l'utilisation de Valgrind

waiting_thread : Un pointeur vers un autre `thread_c` sur lequel le thread actuel attend

waiting_last_mutex_locked : Un pointeur vers le dernier verrou de mutex sur lequel le thread actuel a attendu.

stack : Un tableau de caractères utilisé comme pile pour le thread. Ce champ n'est présent que si le macro `STATIC` est défini, indiquant que la bibliothèque est configurée pour utiliser une pile statique plutôt qu'une pile dynamique. La taille de cette pile est déterminée par la constante `STACK_SIZE`.

2.1.2 Création d'un thread

Pour la création d'un thread (avec `thread_create`), nous avons choisi d'utiliser l'allocation dynamique grâce à la fonction `malloc`. La structure allouée peut être ensuite manipulée et stockée.

Dans notre cas, la structure de thread est liée (par des pointeurs) à d'autres threads, par l'intermédiaire d'une (ou plusieurs) structures chaînées de file (FIFO). Ces structures de files, permettent d'ordonnancer et stocker (par référence) les threads.

De ce fait, un premier choix dans la politique d'ordonnancement consiste à décider s'il faut changer de thread lors de la création d'un nouveau.
Nous avons fait le choix de directement changer de contexte (ou de thread courant) vers le thread créé.
On verra plus tard l'impact que ce choix peut engendrer sur les performances.

2.1.3 Terminaison d'un thread

La fonction `thread_exit` est chargée de la terminaison du thread courant. Lorsqu'un thread a terminé toutes les taches qui lui ont assignés, cette fonction est appelée pour signaler sa fin.

Après avoir signalé la fin du `thread`, nous devons décider quel `thread` exécuter ensuite. Plusieurs critères joue un role dans la prise de cette décision, que ça soit le fait que ce `thread` possède un `thread` en attente, ou dans le cas contraire, nous cherchons un autre thread à exécuter parmi ceux disponibles. Dans le pire des cas, le main `thread` prendra toujours la main.

2.1.4 Jointure de deux threads

La fonction `thread_join` est conçue pour permettre à un thread d'attendre la fin d'un autre thread. Pendant cette attente, l'ordonnanceur change de thread courant.

Un point crucial à noter est l'importance de s'assurer que le thread qui est mis en attente ne soit pas réactivé tant que le thread qu'il attend n'a pas achevé son exécution. Il s'agit d'une étape fondamentale pour garantir le bon fonctionnement

du processus d'exécution concurrente.

Cependant, la manière dont cette précaution est prise en charge sera présenté ultérieurement dans le cadre des optimisations utilisées pour améliorer l'efficacité de la gestion des threads.

2.1.5 Changement de thread courant

2.2 Gestion des mutex

Un mutex permet de garantir que seul un thread à la fois peut accéder à une ressource partagée. Dans votre implémentation, chaque mutex est représenté par une structure `mutex_c`, qui comprend un identifiant, le thread qui détient actuellement le mutex, le dernier thread qui a verrouillé le mutex. (cf. figure 2)

```
typedef struct mutex_c {
    int id;
    thread_t user;
    thread_t last_locked;
} mutex_c;
```

FIGURE 2

Principalement, on utilise `thead_mutex_init` afin d'initialiser un mutex, en allouant de la mémoire et l'ajouter à la table des mutex. Réciproquement, on peut le détruire en libérant la mémoire associéé.
Le fonctionnement de `thread_mutex_lock` et `thread_mutex_unlock` est un processus assez simple. Si le mutex est déjà verrouillé par un autre thread, le thread courant est mis en état de sommeil en attendant que le mutex soit déverouillé en cédant sa place au thread verrouillant. Dans le cas du déverrouillage, on libère le mutex, on cède la main soit au thread courant, soit à celui qu'il la verrouillait

Ces deux fonctions sont la clé pour garantir l'exclusion mutuelle, en assurant qu'une seule thread à la fois peut accéder à une ressource critique. Leur bon usage permet d'éviter les conditions de course et d'autres problèmes liés à la concurrence.

2.3 Gestion de la préemption

La préemption nécessite généralement des interruptions de temps qui déclenchent un signal à des intervalles réguliers pour interrompre le thread actuel et donner le contrôle à un autre thread. C'est l'intérêt de l'utilisation de `ualarm`.

En effet, quand l'alarme est déclenchée, le gestionnaire de signaux thread_yield_handler est invoqué, et effectue un yield au thread suivant dans la queue.
Bien évidement, on aura besoin de protéger le section critque, par la désactivation de la préemption en utilisant `alarm(0)` avant de l'exécuter. D'autres sections critiques sont à noter comme les dernières opérations pendant la terminaison d'un thread, typiquement l'appel à threads_exit. La présence de préemption avec les mutex doit

elle aussi être gérer correctement sinon cela peut entraîner des problèmes de synchronisation dans les programmes concurrents si un processus qui détient un mutex se fait préempter.

2.4 Détection de cycle de jointures

La détection de cycles de jointures est une préoccupation importante lors de la gestion des threads, car cela peut conduire à des situations de deadlock (interblocage) où un ou plusieurs threads se bloquent indéfiniment.

Un cycle de jointure se produit lorsque deux threads ou plus sont bloqués en attendant que l'autre termine, créant un cycle sans fin. Cette détection sera fait principalement dans `thread_join`.

Pour réaliser la détection de cycle, il faut poser des hypothèses/conditions sur l'utilisation de la bibliothèque.
D'abord, on admet qu'un thread ne peut pas réaliser une jointure sur plusieurs threads en même temps..
Ensuite, deux threads (distincts) ne peuvent pas réaliser une jointure sur un même thread (le test `82-join-same` s'assurera de vérifier que la bibliothèque respecte cette condition).

Avec ces hypothèses on se rend compte que les jointures entre les threads forment uniquement des listes chaînés (à sens unique) sans bifurcations.
Par exemple pourrait avoir à un moment donné durant une exécution une configuration : `1 -> 2 -> 3` et `4 -> 5`. Ici 1 et 4 sont des têtes de chaînes alors que 3 et 5 sont des queues.

On remarque qu'un "deadlock" (cycle de jointures) apparaît si et seulement si, le thread en tête d'une chaîne réalise une jointure avec le thread en queue de la même chaîne.

On peut ainsi définir une structure de "jointures chaînées", possédant une tête une queue ainsi que des threads liés par une chaîne.

```
struct join_chain{
    struct thread_c* next;
    struct thread_c* head;
    struct thread_c* tail;
};
```

FIGURE 3

L'implémentation de cette structure est définie en figure 3.
Chaque thread va posséder un champ (de type `struct join_chain`) qui correspond à son maillon dans la chaîne des jointures.

Ensuite il suffira d'actualiser intelligemment la structure de chaînes de jointure pour les threads concernés. Et on remarque, que l'on est pas obligé d'actualiser tous les threads/maillons de la chaîne, mais seulement ceux des têtes et des queues.

Ainsi, un "deadlock" de jointures est détecté en temps constant ($O(1)$). Et l'actualisation intelligente conserve la cohérence de la chaîne de jointures aussi en temps constant.

3 Optimisations

Dans le but d'obtenir une bibliothèque rapide et peu coûteuse en mémoire (vive), une série d'optimisations du code et méthodes ont étés implémentées.

3.1 Tables de threads et mutex

Actuellement les structures de thread/mutex alloués, sont récupérables à partir d'une structure chaînée (notamment une file FIFO).
Une première optimisation consiste donc à utiliser un tableau de pointeurs vers une structure de thread/mutex. Ainsi on obtiendrait un accès direct à un thread/mutex.

Pour cela il faut associer un thread/mutex à un identifiant qui représentera sa position dans la table. Il suffit d'incrémenter de 1 une variable donnant un identifiant à chaque nouvelle création.

Théoriquement, l'obtention d'un thread/mutex passe d'une complexité temporelle dans le pire des cas en $O(n)$ à $O(1)$ avec n le nombre de threads/mutex alloués durant l'exécution.

Dans notre code, la table est simplement un tableau de taille fixe alloué statiquement : `thread_c* thread_table[MAX_THREAD];`
Cela pose un problème dans la limitation du nombre de threads/mutex utilisables. En effet pour conserver le bon fonctionnement de la bibliothèque, le nombre de thread/mutex ne pourra pas dépasser `MAX_THREAD`.

3.2 Statut et files (FIFO) de threads

Il est intéressant de donner un statut aux threads, en effet sans connaissance du statut il est plus complexe de savoir quel thread est actif, en attente ou terminé.
Ainsi, nous avons mis en place un champ dans la structure de thread permettant de retenir le statut.

Ce statut permet notamment de ne pas revenir sur un thread lors de l'ordonnancement. On économise alors grandement le temps de calcul en évitant les changements de contextes inutiles.

Cependant ce n'est pas suffisant. L'utilisation d'une seule file (FIFO) contenant tous les threads n'est pas efficace : lors du changement de thread courant, le choix du prochain thread se fait dans le pire des cas en $O(n)$ (n le nombre de threads dans la file).

Ainsi, on remarque qu'il est préférable d'utiliser plusieurs files (FIFO), une pour chaque statut (actif, en attente, terminé).
Avec les différentes files, le changement de thread courant est direct, soit une complexité temporelle en $O(1)$.

3.3 Changement intelligent de thread courant

Jusqu'à présent, le changement de thread courant se fait selon une politique très simple : **Changer de contexte (thread courant) avec le prochain thread dans la file des thread actifs**.

Mais il peut être plus intéressant de choisir quel sera le nouveau thread courant, à partir de règles simples (qui logiquement devraient faire gagner du temps).
C'est pourquoi la fonction `thread_yield_to` à été implémentée pour réaliser un changement ciblé de thread courant.

Les situations où, nous avons jugé susceptible de choisir intelligemment un nouveau thread courant sont : la terminaison du thread courant, la jointure du thread courant et la libération d'un mutex par le thread courant.

Voici les situations où, nous avons jugé susceptible de choisir intelligemment un nouveau thread courant, ainsi que la règle d'ordonnancement à appliquer :

- Terminaision du thread courant -> Le nouveau thread est celui qui à effectué la jointure avec le thread courant.
- Jointure du thread courant -> Le nouveau thread est celui en fin de la chaîne des jointures. (pour rappel c'est le seul thread de la chaîne à être potentiellement actif).
- Libération d'un mutex par le thread courant -> Le nouveau thread est celui le plus haut dans la pile des thread en attente du mutex.

3.4 Allocation statique

Plutôt qu'allouer dynamiquement les nouveaux threads et mutex, on pourrait utiliser les variables statiques (globales) en fixant une taille limite à la compilation.
Par exemple, la table des threads dans le code deviendrait :
`thread_c thread_table[MAX_THREAD];` (on retire l'étoile `*` pour ne plus manipuler des pointeurs, mais les structures directement).

L'avantage de ce type d'allocation est qu'il permet d'éviter un appel système potentiellement coûteux (ici `malloc`). Par contre, l'espace mémoire de la section `bss` lors de l'exécution risque d'être surchargé et s'il est trop important, le programme ne se lancera pas.

L'utilisateur a la possibilité d'utiliser l'allocation statique en compilant la bibliothèque avec l'option `STATIC`. Nous verrons dans la partie suivante l'impact de l'allocation statique sur performances.

4 Performances

4.1 Vérification de la correction

Dans ce projet, nous avons suivi une approche "test-driven development" grâce aux tests qui nous ont été fournis au début du projet. Ceci nous a permis d'adapter notre code progressivement pour valider l'ensemble des tests :

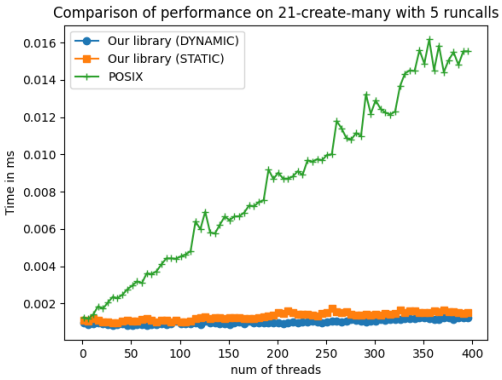
— **01-main :** ce test nous a permis de s'assurer de la validité de notre implémentation initiale en testant les fonctions `thread_yield()` et `thread_self()`
— **02-switch :** grâce à ce test nous avons pu tester si notre bibliothèque est capable de permettre des switchs entre les threads sans erreurs
— **03-equity :** ce test nous a permis de vérifier si notre bibliothèque permet un ordonnancement équitable entre les threads
— **11-join :** grâce à ce test, nous avons tester la fonction `thread_join()`
— **12-join-main :** nous avons testé que nous pouvons faire un join main par un thread fils.
— **51-fibonacci :** ce test nous a permis de vérifier notre implémentation par un calcul distribué de la fonction fibonacci
— **52-sum-list :** dans ce test nous réalisons la somme des éléments d'une liste par plusieurs threads
— **63-mutex-quality :** à travers ce test, nous vérifions que nos mutex ne mettent pas l'équité des threads en défaut
— **71-preemption :** comme son nom l'indique, ce test permet de vérifier la préemption des threads en calculant le temps d'exécution de chaque thread existant.
— **81-deadlock & 83-deadlock :** ces tests permettent de vérifier la détection d'un potentiel deadlock lors d'une suite de join effectué par un cycle de threads
— **82-join-same :** ce test permet de tester le comportement dans le cas ou deux threads ont effectué un join sur un même thread.

À la fin du projet, notre bibliothèque passe l'ensemble des tests avec succès, ceci nous permet de conclure la correction de notre implémentation.

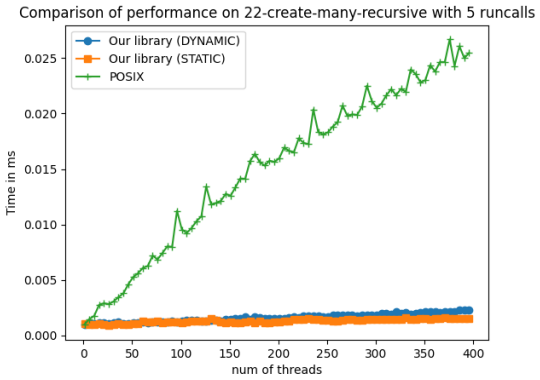
4.2 Performances expérimentales

Afin d'évaluer les performances de notre bibliothèque de threads, nous avons réalisé un ensemble de graphes représentant le temps d'exécution des différents tests pour chaque implémentation.
Pour réduire les variations aléatoires (le bruit par exemple) et afin d'avoir des valeur de temps d'exécution représentatives, nous avons calculé les valeurs en moyennant sur plusieurs runcalls au niveau de chaque tests.

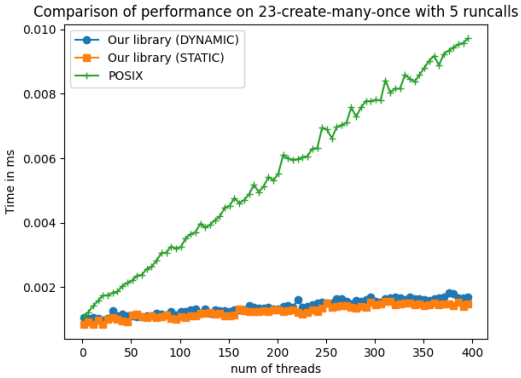
Voici figures illustrant les courbes :



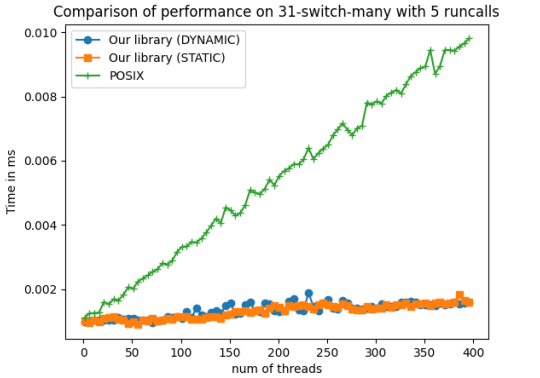
(a) Test : 21-create-many



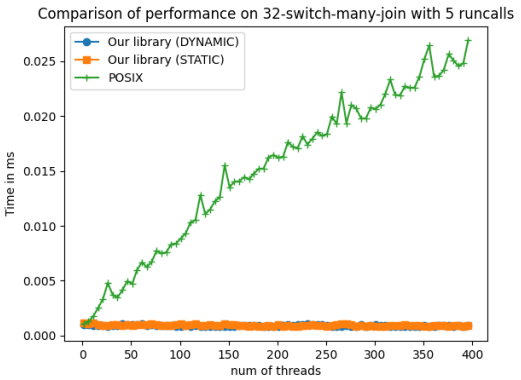
(b) Test : 22-creat-many-recursive



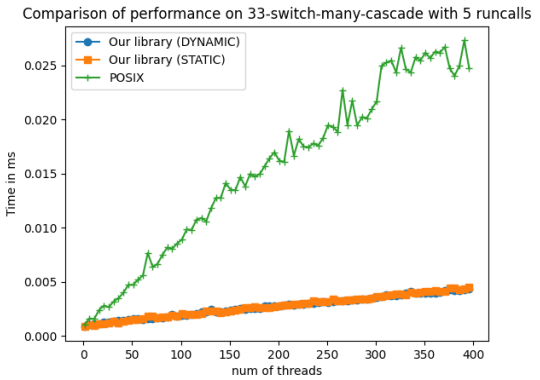
(c) Test : 23-create-many-once



(d) Test : 31-switch-many

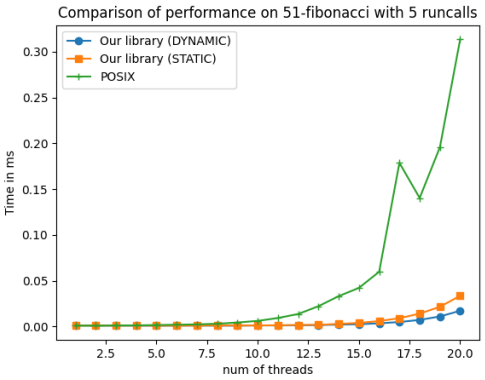


(e) Test : 32-switch-many-join

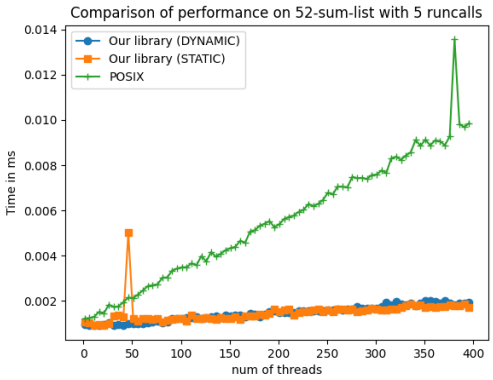


(f) Test : 33-switch-many-cascade

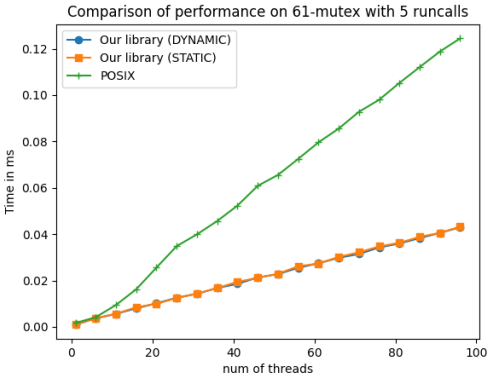
FIGURE 4



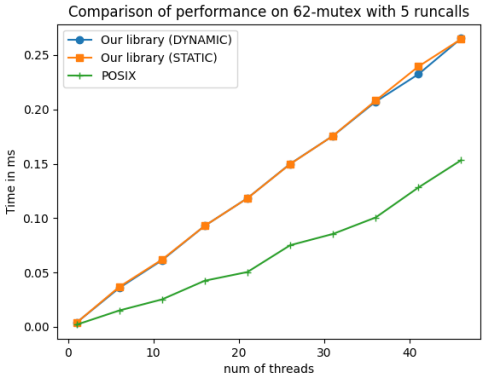
(a) Test : 51-fibonacci



(b) Test : 52-num-list



(c) Test : 61-mutex



(d) Test : 62-mutex

FIGURE 5

D'après les figures, nous constatons que la bibliothèque que nous avons réalisé est plus performante que l'implémentation pthread de POSIX avec un écart considérable au niveau du temps d'exécution. Néanmoins, au niveau du test 62-mutex en particulier, la complexité de notre implémentation de threads est quadratique, et surpasse considérablement celle de pthread.
En résumé, la complexité du coté de pthread a une allure quadratique sur la plupart des tests tandis que la complexité de notre bibliothèque reste constante sur la plupart des tests.
Par ailleurs, nous constatons que les versions statiques et dynamique de notre implémentation performe de manière égale sur les tests.

On peut voir, que toutes les versions de bibliothèques utilisées pour les tests de performances donnent un résultat en **complexité temporelle linéaire** selon le nombre de threads/mutex utilisés (sauf pour fibonacci, dont le paramètre n'est pas le nombre de threads, mais la valeur de calcul).
Ce qui est très satisfaisant, avec nos objectifs.

On remarque aussi que l'allocation statique ne change pas significativement les performances de la bibliothèque (par rapport à l'allocation dynamique).

5 Conclusion

La bibliothèque réalisée, valide les objectifs principaux du projet ainsi que quelques objectifs avancés.
De plus notre priorité d'avoir une complexité constante (pour chaque appel de fonction) quelque soit les conditions d'utilisations, a bien été respectée.
Les performances observées sur le leaderboard sont très correctes dans l'ensemble, mais différentes selon les tests (cela est dû à nos politiques d'ordonnancement).

Finalement, les fonctionnalités et résultats de la bibliothèque produite sont satisfaisants.