Programmation Système : Rapport de projet

Sriguru ELUMALAI, Olha NEMKOVYCH

May 7, 2024

1 Introduction

Notre projet comprend la mise en œuvre de deux types d'ordonnanceurs pour gérer le multithreading dans les systèmes : un ordonnanceur LIFO (Last In, First Out) et un ordonnanceur par Work Stealing. Les deux ordonnanceurs sont conçus pour être utilisés dans des systèmes multicœurs, où une répartition efficace des tâches entre les cœurs est essentielle pour obtenir des performances élevées. L'utilisation de différentes stratégies d'ordonnancement permet de comparer leur efficacité dans diverses conditions et de choisir l'approche la plus appropriée pour des tâches ou des conditions d'exécution spécifiques.

1.1 Ordonnanceur LIFO (sched_lifo.c)

Le fichier sched_lifo.c contient une implémentation d'un ordonnanceur LIFO qui utilise la pile pour gérer les tâches. Dans cette approche, la dernière tâche ajoutée à la pile sera exécutée en premier. Cette structure convient aux tâches pour lesquelles les nouvelles tâches ont une priorité plus élevée, mais elle peut entraîner des retards pour les tâches qui restent longtemps dans la pile. Les opérations de base incluent l'initialisation de la pile, l'ajout de tâches à la pile, leur suppression et leur exécution avec des threads. Toute synchronisation est réalisée au moyen de mutex, que l'on verrouille avec des locks, assurant ainsi la sécurité lors de l'accès aux ressources partagées.

1.1.1 Structures

```
typedef struct TaskNode {
    taskfunc task_function;     // Fonction de la tâche
    void* closure_data;         // Données associées à la tâche
    struct TaskNode* next_task; // Pointeur vers la prochaine tâche
} TaskNode;

typedef struct TaskStack {
    TaskNode* top_task;         // Sommet de la pile
    pthread_mutex_t lock;       // Mutex pour protéger la pile
    int size;                   // Taille actuelle de la pile
```

```
} TaskStack;

typedef struct scheduler {
    int num_threads;            // Nombre de threads dans le scheduler
    int max_task_queue_length;  // Capacité maximale de la pile de tâches
    TaskStack* task_stack;      // Pile de tâches du scheduler
    pthread_mutex_t cond_mutex; // Mutex pour la variable condition
    pthread_cond_t cond;        // Condition pour la synchronisation des threads
    int num_thread_sleep;       // Nombre de threads actuellement endormis
} Scheduler;
```

- **TaskNode** : Une structure représentant une tâche, avec des pointeurs vers une fonction de tâche (`taskfunc`), des données pour cette fonction (`closure_data`) et la tâche suivante sur la pile (`next_task`).
- **TaskStack** : Une structure représentant la pile de tâches, avec un pointeur vers la tâche supérieure (`top_task`), la taille de la pile (`size`), un mutex pour la synchronisation des accès à la pile.
- **Scheduler** : Une structure permettant l'exécution du multithreading et la synchronisation des tâches.

1.1.2 Fonctions de traitement de base des tâches dans la pile

- `lifo_push(TaskStack* stack, TaskNode* task)`: Cette fonction prend une pile LIFO et une tâche à empiler. Elle place la tâche au sommet de la pile, met à jour la taille de la pile.
- `lifo_pop(TaskStack* stack)`: Cette fonction prend une pile LIFO en entrée. Elle récupère la tâche au sommet de la pile, met à jour le sommet de la pile et la taille de la pile avant de retourner la tâche retirée.

1.1.3 Fonction d'exécution de chaque thread

La fonction `thread_function(void* arg)` est responsable de l'exécution des tâches par chaque thread de l'ordonnanceur. À chaque itération de la boucle, le thread vérifie s'il y a une tâche disponible dans la pile de tâches du scheduler. S'il y en a une, le thread récupère la tâche et exécute sa fonction associée avec les données de fermeture correspondantes. Si la pile est vide mais que d'autres threads sont toujours actifs, le thread se met en attente jusqu'à ce qu'une nouvelle tâche soit ajoutée à la pile ou qu'un autre thread le réveille. Cette fonction permet d'assurer une exécution asynchrone des tâches, où chaque thread travaille de manière autonome sur les tâches disponibles.

1.1.4 Initialisation et gestion de l'ordonnanceur

- `sched_init(int nthreads, int qlen, taskfunc f, void *closure)` : Initialise l'ordonnanceur, crée le nombre requis de threads, et ajoute la

tâche initiale à la pile. Elle gère également la création et la terminaison des threads.

- `sched_spawn(taskfunc f, void *closure, struct scheduler *s)` : Elle crée une nouvelle tâche avec la fonction `f` et les données de fermeture `closure`, puis l'ajoute à la pile du planificateur. Ensuite, elle signale aux threads endormis qu'une nouvelle tâche est disponible, ce qui leur permet de se réveiller et de traiter la nouvelle tâche.

1.2 Ordonnanceur par Work Stealing (sched_ws.c)

Le fichier sched_ws.c implémente l'algorithme de l'ordonnanceur par Work Stealing. Cette approche suppose que chaque thread possède sa propre file d'attente de tâches (deque). Lorsqu'un thread termine ses tâches, il peut "voler" une tâche dans la file d'attente d'un autre thread qui a encore des tâches en attente. Cela réduit les temps d'arrêt et améliore l'équilibrage de charge entre les threads. Les files d'attente à double extrémité permettent d'ajouter et de supprimer des éléments aux deux extrémités, ce qui est optimal pour un algorithme par Work Stealing. Comme avec l'ordonnanceur LIFO, toutes les opérations sur les ressources partagées sont synchronisées pour garantir la sécurité du multithreading.

1.2.1 Structures

```
typedef struct Task {
    taskfunc t;                // Pointeur vers la fonction à exécuter
    void *closure;             // Arguments de la fonction
    struct Task *prev;         // Pointeur vers la tâche précédente
    struct Task *next;         // Pointeur vers la prochaine
} Task;

typedef struct Deque {
    int size;                   // Nombre de tâches
    Task *first;                // Pointeur vers la première tâche
    Task *last;                 // Pointeur vers la dernière tâche
    pthread_mutex_t mutex;      // Mutex pour la synchronisation
} Deque;

typedef struct ThreadInfo {
    pthread_t thread;            // Identifiant du thread
    struct Deque *deque;         // File d'attente de tâches associée au thread
    int sleep;                   // Thread en attente (1) ou non (0)
    struct scheduler *scheduler; // Référence vers l'ordonnanceur
    int tasks_executed;          // Nombre de tâches exécutées par le thread
    int successful_work_steals;  // Nombre de vols de travail réussis par le thread
    int failed_work_steals;      // Nombre de vols de travail échoués par le thread
```

```
} ThreadInfo;

typedef struct scheduler {
    int nthreads;                  // Nombre de threads
    int qlen;                      // Longueur minimale de la file d'attente
    ThreadInfo *threads;           // Tableau de threads
    pthread_mutex_t mutex;         // Mutex pour la synchronisation
    int nthreads_sleep;            // Nombre de threads endormis
    int total_tasks_executed;      // Nombre total de tâches exécutées
    int total_successful_work_steals; // Nombre total de vols de travail réussis
    int total_failed_work_steals;     // Nombre total de vols de travail échoués
} scheduler;
```

1.2.2 Fonctions de traitement de base de la deque

- `initDeque()`: Initialise une nouvelle deque, configure les mutex et la mémoire.
- `freeDeque(Deque *deque)`: Libère un deque et ses ressources.
- `initThread(int id, struct scheduler *scheduler)`: Initialise un thread et son deque associé.
- `freeThread(ThreadInfo thread)`: Libère les ressources utilisées par un thread.
- `pushBack(Deque deque, taskfunc f, void closure)` et `pushFront(Deque *deque, taskfunc f, void *closure)`: Fonctions pour ajouter des tâches à l'arrière et à l'avant d'un deque.
- `popBack(Deque *deque)` et `popFront(Deque *deque)`: Fonctions pour supprimer des tâches à l'arrière et à l'avant du deque, prenant en charge les deux sens de suppression.

1.2.3 Fonctions d'exécution de chaque thread

- `workStealing(ThreadInfo *threads, int nthreads, int index)`: Elle permet à un thread de récupérer des tâches à partir des files d'attente d'autres threads lorsque sa propre file est vide. Elle sélectionne aléatoirement un thread disponible, tente de voler une tâche de sa file d'attente et met à jour les statistiques en cas de succès ou d'échec.
- `thread_function(void *arg)`: Elle est exécutée par chaque thread qui vérifie régulièrement sa propre file d'attente (deque). Si elle est vide, le thread tente le Work Stealing. Elle utilise des verrous pour assurer un accès sécurisé à la file d'attente. Si aucune tâche n'est récupérée, le thread peut entrer en mode veille pour économiser les ressources. Enfin, il exécute les tâches disponibles dans sa file d'attente locale, mettant à jour les statistiques après chaque exécution.

1.2.4 Initialisation et gestion de l'ordonnanceur

- `sched_init(int nthreads, int qlen, taskfunc f, void *closure)` : Cette fonction initialise l'ordonnanceur de tâches en s'assurant qu'une seule file d'attente contient la tâche initiale, tandis que les autres sont vides. Chaque thread retire une tâche de sa file après exécution, ou tente de voler une tâche à un autre thread en cas de besoin. Si aucun travail n'est disponible, le thread attend 1ms avant de réessayer. En plus de cela, la fonction alloue la mémoire nécessaire, initialise les paramètres et les verrous, et affiche les statistiques une fois que tous les threads sont inactifs.
- `sched_destroy(struct scheduler *s)` : Nettoie et libère toutes les ressources associées à l'ordonnanceur.
- `sched_spawn(taskfunc f, void *closure, struct scheduler *s)` : Elle ajoute une nouvelle tâche à l'ordonnanceur, spécifiquement au deque du thread qui appelle cette fonction.

2 Comparaison et analyse des Performances

Pour évaluer l'efficacité des stratégies de chaque ordonnanceur implémentées dans notre système : l'ordonnanceur LIFO et l'ordonnanceur par Work Stealing, nous avons systématiquement collecté des données de performance à travers une série de tests. Ces tests ont été conçus pour mesurer plusieurs aspects clés tels que le temps d'exécution total, la capacité de traitement des tâches, et l'efficacité du vol de travail sous diverses charges de travail.

Les graphiques suivants présentent les résultats de ces mesures, offrant une visualisation claire de la performance de chaque stratégie d'ordonnancement sous différents scénarios d'exécution.

Ces visualisations sont essentielles pour comprendre non seulement comment chaque ordonnanceur gère les tâches en temps réel mais aussi pour identifier les opportunités d'optimisation afin d'améliorer la réactivité et l'efficacité globale du système.

Pour chaque variation de débit (de 1 à $2 \times nombredecœurs$), 20 essais ont été réalisés, dont nous avons retenu la valeur moyenne. Après cela, les graphiques suivants ont été dessinés sur deux variantes de cœurs (4 cœurs et 12 cœurs).

Le nombre de tâches en trois variantes (10 000 éléments, 10 485 760 éléments (selon la norme) et 10 737 418 823 éléments (ils fonctionnent très longtemps, mais même sur un petit nombre de cœurs la différence est visible)).

Les deux premiers graphiques sont réalisés sur 4 cores, WSL Linux :

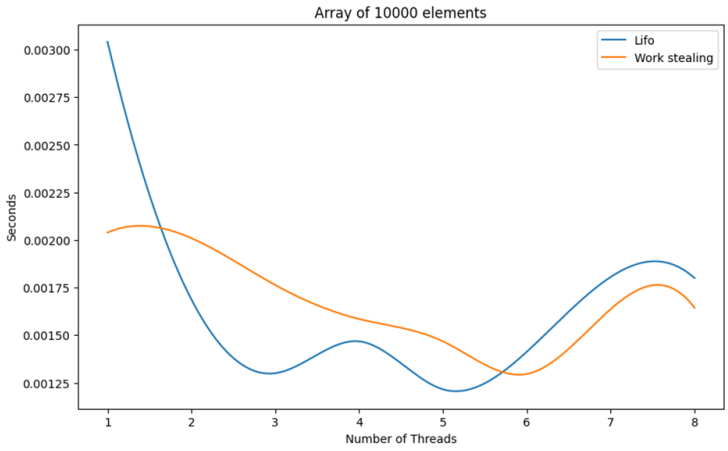

Figure 1: graphique 1

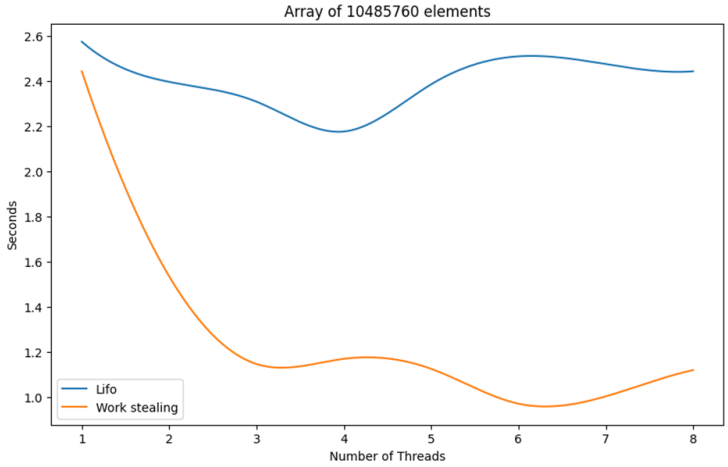

Figure 2: graphique 2

Les trois graphiques suivants sont réalisés sur un MacBook i7 2019, qui possède 12 cœurs (6 physiques et 6 virtuels) :

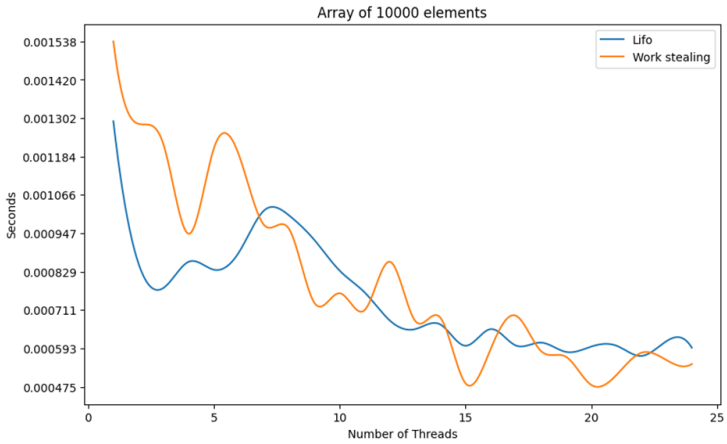

Figure 3: graphique 3

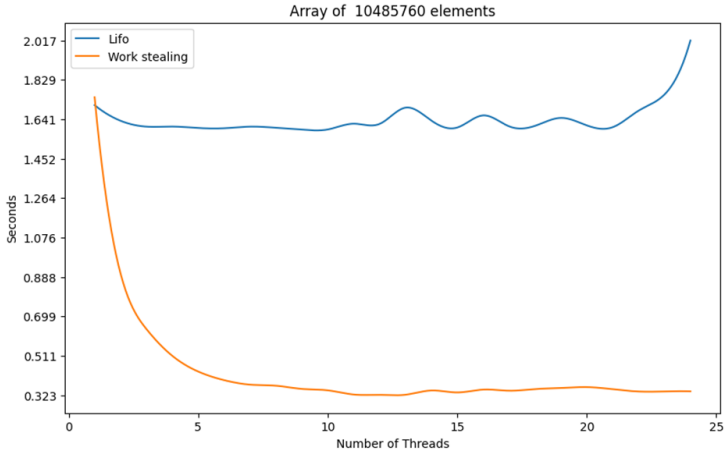

Figure 4: graphique 4

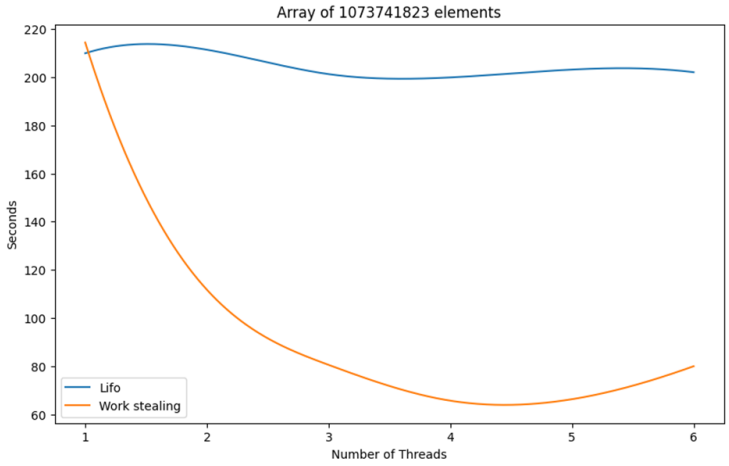

Figure 5: graphique 5

3 Description des graphiques

Les graphiques montrent que l'ordonnanceur par Work Stealing est plus efficace, avec des temps d'exécution plus bas lorsque le nombre de threads augmente. L'ordonnanceur LIFO, en revanche, montre une efficacité moindre, avec des temps d'exécution plus longs et des fluctuations importantes.

4 Comparaison avec MacBook et WSL Linux

Les résultats des tests montrent que l'ordonnanceur LIFO a des temps d'exécution plus longs et des fluctuations significatives lorsque le nombre de threads augmente, en particulier sur le MacBook. À mesure que le nombre de cœurs augmente, les performances du LIFO ont tendance à se stabiliser ou à se détériorer, ce qui indique qu'il n'est pas bien adapté pour exploiter efficacement plusieurs cœurs.

L'ordonnanceur par Work Stealing, quant à lui, permet un équilibrage dynamique de la charge de travail. Il optimise l'utilisation des cœurs disponibles et s'avère plus efficace pour les traitements parallèles. Les tests ont révélé des temps d'exécution réduits et une meilleure adaptation aux variations du nombre de threads sur les deux systèmes, mais particulièrement dans l'environnement du MacBook. Sur le MacBook à 12 cœurs, l'ordonnanceur par Work Stealing a pu effectuer les tâches plus rapidement car il répartissait efficacement les charges de travail entre les cœurs disponibles, bénéficiant ainsi du plus grand nombre de cœurs pour le traitement parallèle.

Les graphiques d'efficacité montrent clairement que l'ordonnanceur par Work Stealing est plus performant et évolutif que l'ordonnanceur LIFO. Cela est évident à la fois dans l'environnement MacBook à 12 cœurs avec 24 threads et dans l'environnement WSL à 4 cœurs avec 8 threads. Le plus grand nombre de cœurs du MacBook offre un avantage distinct au scheduler par Work Stealing, lui permettant de gérer plus efficacement les charges de travail parallèles.

4.1 Analyse de l'ordonnanceur LIFO

4.1.1 Avantages

- Traitement simple et efficace des tâches récemment ajoutées.

4.1.2 Inconvénients

- Inefficace pour le traitement parallèle.
- Retarde les tâches plus anciennes.

4.2 Analyse de l'ordonnanceur par Work Stealing

4.2.1 Avantages

- Équilibrage dynamique de la charge de travail.
- Optimise l'utilisation des cœurs disponibles.

4.2.2 Inconvénients

- Peut présenter des frais généraux plus élevés avec trop de threads.

5 Conclusion

En résumé, l'ordonnanceur par Work Stealing s'avère être une meilleure solution pour les applications multithread, offrant une répartition plus équilibrée des tâches et de meilleures performances globales. L'ordonnanceur LIFO, bien qu'utile dans des scénarios spécifiques, est moins adapté aux environnements multicœurs exigeants. Les résultats obtenus sont cohérents avec les attentes et les comportements caractéristiques de chaque type d'ordonnanceur.

6 Références

1. https://www.geeksforgeeks.org/lifo-last-in-first-out-approach-in-programming/
2. https://stackoverflow.com/questions/2805102/how-is-pushing-and-popping-defined
3. https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/
4. https://blog.molecular-matters.com/2015/09/08/job-system-2-0-lock-free-work-stealing-part-2-a-specialized-allocator/
5. https://blog.molecular-matters.com/2015/09/25/job-system-2-0-lock-free-work-stealing-part-3-going-lock-free/