Projet de Programmation Système
Un ordonnanceur par *work stealing*

Juliusz Chroboczek

6 mars 2024

1 Introduction

Le processeur de votre machine dispose de plusieurs *cœurs* dont chacun peut exécuter, à un moment donné, un *thread* système ou processus. Un programme purement séquentiel n'exploite qu'une fraction de la puissance de calcul du processeur ; pour l'exploiter davantage, il faut écrire des programmes parallèles.

Il est parfois possible de paralléliser un programme manuellement, mais il est généralement difficile de distribuer les tâches à effectuer sur le nombre limité de cœurs disponibles. Considérez par exemple le *quicksort* parallèle suivant, écrit dans un C étendu où le mot-clé `spawn` crée un nouveau *thread* :

```
void quicksort(int *a, int lo, int hi) {
    int p;
    if(lo >= hi)
        return;
    p = partition(a, lo, hi);
    if(???) {
        spawn quicksort(a, lo, p - 1);
        spawn quicksort(a, p + 1, hi);
    } else {
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

À chaque étape de la récursion, il faut décider s'il faut exécuter les sous-tâches en parallèle ou séquentiellement. Évidemment, les premières étapes récursives doivent exécuter les sous-tâches en parallèle ; mais à quel moment faut-il passer à une exécution séquentielle ? Si on le fait trop tôt, tous les cœurs ne seront pas exploités ; si on le fait trop tard, il y aura beaucoup trop de *threads*, dont chacun consommera de la mémoire et causera des changements de contexte inutiles.

Une *tâche (task)* est un morceau atomique de travail qui sera entièrement exécuté sur un *thread*. Un ordonnanceur *(scheduler)* s'occuppe de distribuer un nombre potentiellement très grand de tâches sur un nombre limité de *threads* système. Il existe deux techniques principales d'ordonnancement :

- le *work sharing* où les tâches sont distribuées parmi les *threads* (statiquement ou dynamiquement) ;
- le *work stealing*, où les *threads* oisifs (qui n'ont rien à faire) « volent » des tâches aux autres *threads*.

Le but de ce projet est d'implémenter deux ordonnanceurs, dont un qui utilise une simple file partagée et un qui fait du *work stealing*, et d'en comparer les performances.

2 Interface de l'ordonnanceur

Nous vous fournissons un fichier `sched.h` qui contient l'interface que votre ordonnanceur devra *obligatoirement* implémenter.

Représentation des tâches Une tâche est représentée par une paire d'un pointeur de fonction de type `taskfunc` et d'un pointeur non-typé `void*`.

```
typedef void (*taskfunc)(void*, struct scheduler *);
```

Effectuer une tâche (`f`, `p`) consiste à évaluer `f(p, s)`, où `s` est un pointeur sur une structure de données représentant l'ordonnanceur.

`sched_init` La fonction `sched_init` lance l'ordonnanceur.

```
int sched_init(int nthreads, int qlen,
               taskfunc f, void *closure);
```

Elle prend en paramètre :

- le nombre `nthreads` de *threads* que va créer l'ordonnanceur ; si ce paramètre vaut zéro, le nombre de *threads* sera égal au nombre de cœurs de votre machine (que vous pouvez déterminer à l'aide d'un appel à `sysconf(_SC_NPROCESSORS_ONLN)`) ;
- le nombre minimum `qlen` de tâches simultanées que l'ordonnanceur devra supporter ; si l'utilisateur essaie d'enfiler plus de `qlen` tâches, l'ordonnanceur pourra retourner une erreur (comme décrit ci-dessous) ;
- la tâche initiale (`f`, `closure`).

La fonction `sched_init` retourne lorsqu'il n'y a plus de tâches à effectuer, et alors son résultat vaut 1. Elle retourne -1 si l'ordonnanceur n'a pu être initialisé.

`sched_spawn` La fonction `sched_spawn` sert à enfiler une nouvelle tâche.

```
int sched_spawn(taskfunc f, void *closure, struct scheduler *s);
```

Elle prend en paramètre une tâche (`f`, `closure`) et un ordonnanceur `s`. Elle a pour effet d'insérer la tâche dans l'ensemble des tâches à exécuter par l'ordonnanceur `s`. Cette fonction ne peut être exécutée qu'à partir d'une tâche qui s'exécute sur `s`.

Si le nombre de tâches en file est déjà supérieur ou égal à la capacité de l'ordonnanceur (la valeur du paramètre `qlen` passé à `sched_init`), cette fonction peut soit enfiler la tâche quand même, soit retourner -1 avec `errno` valant `EAGAIN`. En d'autres termes, l'ordonnanceur n'est pas obligé d'enfiler plus de tâches que prévu, mais il n'est pas non plus obligé de détecter ce cas.

La fonction `sched_spawn` retourne immédiatement (dès qu'elle s'est arrangée pour que la tâche soit effectuée).

3 Un ordonnanceur LIFO

Dans cette partie, je décris un ordonnanceur facile à implémenter qui servira de point de comparaison pour évaluer les performances de votre ordonnanceur par *work stealing*.

L'ordonnanceur LIFO contient une seule pile de tâches dont la taille maximale est donnée par le paramètre passé à `sched_init`. Initialement, la pile ne contient que la tâche initiale. Lorsqu'un *thread* n'a rien à faire, il dépile une tâche de la pile et l'effectue ; s'il n'y a aucune tâche prête, le *thread* s'endort en attendant qu'il y en ait une (par exemple, la fonction `sched_spawn` peut essayer de réveiller un *thread*). Lorsqu'une nouvelle tâche est créée, elle est empilée sur la pile. L'ordonnanceur termine lorsque la pile est vide et tous les *threads* sont endormis.

Vous remarquerez que la pile est une structure partagée, il faudra donc la protéger par des primitives de synchronisation. Il faudra aussi utiliser des primitives de synchronisation pour réveiller les *threads* endormis, et pour qu'ils terminent lorsque l'ordonnanceur devient oisif.

4 Un ordonnanceur par *work stealing*

La pile de tâches de l'ordonnanceur LIFO constitue un point de contention, qui limite le passage à l'échelle. De plus, l'ordonnanceur LIFO ne fait aucun effort pour optimiser la localité : si une tâche T crée une tâche U, aucun effort n'est fait pour que T et U s'exécutent sur le même *thread*, ce qui cause une mauvaise utilisation du cache.

L'ordonnanceur par *work stealing* évite ces problèmes en utilisant une file à double bout (une *deque*) par *thread*, et donc, en pratique, une *deque* par cœur.

Deques Une *deque* est une structure de données qui généralise les files et les piles : elle permet l'enfilage et le défilage des deux côtés. Une *deque* a deux bouts, le haut et le bas, et quatre opérations : enfiler en bas, enfiler en haut, défiler en bas et défiler en haut. On peut implémenter une *deque* comme une file circulaire ou comme une liste doublement chaînée.

Ordonnanceur L'ordonnanceur contient *n threads* et *n deques*, une par *thread*. Initialement, toutes les *deques* sont vides sauf une, qui contient la tâche initiale. Lorsqu'un *thread* a fini d'exécuter une tâche, il défile la tâche qui est en bas de la *deque* qui lui est associée ; si cette *deque* est vide, il effectue une étape de *work stealing* (décrite ci-dessous). S'il n'a toujours pas réussi à trouver du travail, il s'endort pendant 1 ms, puis tente une étape de *work stealing* de nouveau.

Un processus qui est dans la phase de *work stealing* se comporte de la façon suivante. Il commence par tirer au hasard un *thread k*, et essaie de défiler une tâche du haut de la *deque* associée à ce *thread*. Si cette opération réussit (la *deque* n'était pas vide), il effectue la tâche ainsi obtenue puis continue à exécuter l'algorithme normal. Sinon, il itère à travers tous les autres *threads* en commençant à $k + 1$ (*modulo* le nombre de *threads*) et essaie de voler du travail à l'un d'eux. Si toutes les *deques* étaient vides, le *work stealing* a échoué, et le *thread* s'endort.

Lorsqu'une nouvelle tâche est créée, elle est enfilée en bas de la *deque* associée au *thread* qui a fait l'appel à `sched_spawn` (la création de tâches est locale). L'ordonnanceur termine lorsque tous les *threads* sont oisifs : ils sont tous en train de dormir après avoir échoué à voler du travail.

Quelques commentaires Dans le cas où le nombre de *threads* est égal à 1, les tâches sont enfilées et défilées en bas de l'unique *deque*, qui se comporte donc comme une pile. L'ordonnanceur dégénère donc dans ce cas en un ordonnanceur LIFO.

L'efficacité de cet algorithme est due à deux facteurs. Tout d'abord, chaque *deque* est dédiée à un *thread*, ce qui évite qu'elle constitue un point de contention. Cependant, lors de l'étape de *work stealing* un *thread* accède aux *deques* des autres *threads*, ce qui implique qu'il faut quand même protéger ces dernières par des primitives de synchronisation.

Ensuite, une nouvelle tâche est toujours enfilée sur la *deque* associée au *thread* qui l'a créée, et une tâche ne migre que durant l'étape de *work stealing*, qui est relativement rare. De ce fait, l'algorithme maximise la localité, ce qui entraîne une bonne utilisation des caches du processeur.

5 Sujet minimal

Vous devrez, au minimum, nous fournir :

- une implémentation d'un ordonnanceur LIFO implémentant *obligatoirement* l'interface définie dans le fichier `sched.h` ;
- une implémentation d'un ordonnanceur par *work stealing* implémentant la même interface ;
- un rapport au format PDF qui décrit votre travail et qui contient entre autres les résultats de *benchmarks* qui comparent les performances de vos deux ordonnanceurs en fonction du nombre de cœurs.

Votre code devra être écrit en C, les solutions dans d'autres langages ne seront pas acceptées. Votre code doit être basé sur la bibliothèque *pthread*, et vous n'êtes pas autorisés à utiliser des bibliothèques autres que la bibliothèque standard du C. Votre code devra compiler et s'exécuter sur un Linux récent (mais pourra être portable sous d'autres systèmes). En dehors de ces conditions, vous êtes libres des détails d'implémentation. Par exemple, vous pourrez implémenter les *deques* comme des listes chaînées ou comme des files circulaires stockées dans des tableaux, et

nous vous laissons libres des primitives de synchronisation à utiliser (du moment que vous les utilisez correctement).

6 Extensions

Comme toujours, toutes les extensions seront les bienvenues. Voici quelques idées.

Statistiques et *benchmarks* Une bonne façon de s'assurer du bon comportement de votre algorithme est de l'instrumenter pour obtenir des statistiques. Il est notamment intéressant de connaître le nombre de tâches effectuées, le nombre d'étapes de *work stealing* qui réussissent, et le nombre d'étapes de *work stealing* qui échouent. Attention cependant à ne pas perturber le système que vous observez : si vous incrémentez un compteur global protégé par un *mutex* ou une variable atomique globale, ce compteur global constituera un point de contention. (Une bonne solution consiste à utiliser un compteur local à chaque *thread*.)

Fournissez-nous aussi des *benchmarks* et leurs résultats [1]. Il y a plusieurs types de résultats qui peuvent être intéressants. Tout d'abord, il faudra déterminer comment se comporte votre code par rapport au *scheduler* LIFO, nous nous attandons donc à ce que cette comparaison sont présente dans votre rapport. Ensuite, il est important de comprendre comment se comporte votre code lorsqu'on varie le nombre de cœurs et le nombre de *threads* : un code optimisé pour le cas de deux cœurs peut mal se comporter sur huit cœurs, donc si vous possédez une machine de *gamer*, profitez-en pour me fournir les résultats que je ne pourrais pas obtenir moi-même.

Ensuite, je trouve intéressantes les comparaisons entre différentes primitives de synchronisation. Par exemple, les files de votre ordonnanceur peuvent être protégées par des *mutex pthreads*, des *spinlocks* codés à la main, des sémaphores POSIX, ou d'autres primitives de synchronisation. Quelles sont les conséquences du choix de la primitive sur les performances de votre code ?

Enfin, j'aimerais bien voir des comparaisons entre différents systèmes exploitation, de préférence sur le même matériel. Les *mutex* Linux sont très rapides, mais n'offrent aucune garantie d'équité (ce qui est probablement le bon choix pour le *work stealing*). Je me suis laissé dire que Mac OS X utilisait des *mutex* équitables par défaut, ce qui cause des changements de contexte supplémentaires et est susceptible au problème du *Lock Convoy* [2], mais évite certains cas pathologiques en présence de contention. Enfin, *Windows* fournit plusieurs types de primitives d'exclusion mutuelle ayant des performances très différentes, et dont certaines interagissent avec les priorités des *threads*. Quelles sont les conséquences de ces choix d'implémentation pour votre ordonnanceur ?

Démos et programmes d'exemple Une démo réussie lors de la soutenance fait toujours bonne impression, et même une démo ratée ne devrait pas vous pénaliser. Fournissez-nous des programmes d'exemple d'utilisation de votre *scheduler*. Les programmes qui font des graphiques sont une bonne idée, car ils vous donnent une chance d'impressionner l'examinateur.

1. On adore les figures, mais on déteste les figures en 3D.
2. *Wikipedia* est votre amie.

Optimisations Votre ordonnanceur peut probablement être optimisé. En particulier, la procédure qui consiste à s'endormir pendant 1 ms en cas de famine est douteuse. Vous pourriez penser à déterminer dynamiquement la bonne valeur du temps d'attente (par exemple en augmentant sa valeur à chaque fois que le *work stealing* échoue, et en la diminuant à chaque fois qu'il réussit). Vous pourriez aussi penser à utiliser une primitive de synchronisation pour réveiller un *thread* oisif lorsque du travail devient disponible, mais attention, les opérations de synchronisation ne sont pas gratuites : vérifiez que ça améliore effectivement vos *benchmarks*.

Une inefficacité évidente est le fait qu'un *thread* prend un *lock* sur sa propre file, même lorsqu'aucun autre *thread* ne souffre de famine. Comme le *work stealing* est relativement rare, il vaudrait mieux éviter cette synchronisation, même si cela rend le *work stealing* moins efficace. Peut-on éviter une synchronisation dans le cas commun en utilisant des opérations atomiques ?

Opérations bloquantes Si une tâche effectue une opération d'entrée-sortie bloquante, elle occupe un *thread* qui ne sert donc plus à exécuter d'autres tâches. Proposez une interface qui évite ce problème et implémentez-la. Vous pourrez par exemple fournir une interface qui permet de temporairement créer un *thread* supplémentaire durant l'exécution d'une opération potentiellement bloquante, et de tuer un *thread* (pas forcément le même) après que l'opération a terminé.

7 Modalités de soumission

Le projet sera fait par groupes de deux étudiants [3]. Le travail devra être fait par les membres du groupe, sans s'inspirer du travail des autres groupes. Si, exceptionnellement, vous vous faites aider par les membres d'un autre groupe, il sera *obligatoire* de le mentionner dans le rapport (en créditant nominalement votre source).

Si vous vous inspirez de code trouvé dans des livres, des articles, ou des sources sur l'Internet, il sera *obligatoire* de citer votre source. En particulier, il est strictement interdit d'utiliser des outils tels que *ChatGPT*, qui ne fournissent pas la source des informations qu'ils vous donnent.

Vous nous remettrez avant la soutenance une archive *nom1-nom2*`.tar.gz` contenant notamment :

- le source complet de votre programme, accompagné d'un fichier `README` indiquant sommairement comment le compiler et l'exécuter ;
- un rapport sous format PDF, contenant une description sommaire de votre programme et les statistiques et résultats de *benchmarks* que vous aurez obtenus.

Cette archive devra s'extraire dans un sous-répertoire *nom1-nom2* du répertoire courant. Par exemple, si vous vous appelez Stefan Banach et Hugo Steinhaus, votre archive devra porter le nom `banach-steinhaus.tar.gz` et son extraction devra créer un répertoire nommé `banach-steinhaus` contenant tous les fichiers que vous nous soumettez. Vous enverrez votre archive par courrier électronique à l'adresse `<jch@irif.fr>`. Votre mail devra *obligatoirement* avoir un entête `Subject` ayant la valeur « `Projet système` ».

3. Pas trois. Les singletons seront tolérés, mais traités sans indulgence particulière.