Structures de données concurrentes

Orestis Malaspinas, Steven Liatti

1 Les structures de données basées sur les verrous

Le contenu de ce chapitre est basé sur l'excellent livre de R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau [1].

Dans ce chapitre, nous allons brièvement voir comment utiliser des verrous dans des structures de données standards de façon à ce qu'elles soient sûres à l'utilisation dans des applications multi-threadées. La difficulté principale est de garder une bonne performance tout en garantissant un fonctionnement correct.

1.1 Les compteurs concurrents

La structure de données la plus simple qu'on puisse imaginer est un simple compteur. Nous avons déjà vu dans les exercices comment incrémenter de façon fausse un compteur de façon concurrente, puis une façon juste mais peu efficace. Histoire de formaliser tout ceci un peu ici, écrivons un compteur atomique dans la structure `counter_t` ainsi que les opérations d'incrémentation et de décrémentation.

```
typedef struct __counter_t {
    int value;            // la valeur du compteur
    pthread_mutex_t lock; // le mutex protégeant le compteur
} counter_t;

void init(counter_t *cnt) {
    cnt->value = 0;
    pthread_mutex_init(&cnt->lock, NULL);
}

void increment(counter_t *cnt) {
    pthread_mutex_lock(&cnt->lock);
    cnt->value += 1;
    pthread_mutex_unlock(&cnt->lock);
}

void decrement(counter_t *cnt) {
    pthread_mutex_lock(&cnt->lock);
```

1. R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ed. 0.91, (2015).

```
    cnt->value -= 1;
    pthread_mutex_unlock(&cnt->lock);
}

int get_value(counter_t *cnt) {
    pthread_mutex_lock(&cnt->lock);
    int value = cnt->value;
    pthread_mutex_unlock(&cnt->lock);

    return value;
}
```

Ici, nous avons rendues atomiques les opérations d'incrémentation, de décrémentation et de retour de la valeur du compteur en protégeant les sections critiques (d'incrémentation, de décrémentation et du compteur) à l'aide de verrous. Notre structure de données est donc parfaitement concurrente et fonctionne correctement. Néanmoins, on peut assez facilement se rendre compte que cette méthode peut souffrir d'un problème de performances. Néanmoins, on a ici une méthode très très simple de rendre une structure de données concurrente: ajouter un verrou! Pour rendre ces structures plus efficaces, il faut utiliser la ruse.

Pour illustrer le problème de performances d'un code n'utilisant qu'un simple verrou, on peut s'intéresser à la performance du code de l'incrémentation de notre compteur un million de fois. Si on mesure le temps d'exécution du code sur un, deux ou quatre fils d'exécution, on obtient le tableau ci-dessous :

	1 thread	2 threads	4 threads
Temps $[ms]$	0.031	0.149	0.447

On voit un ralentissement assez spectaculaire de l'exécution (plus d'un facteur 10) entre le cas à un seul thread et le cas à 4 threads. Ici, la quantité de travail à effectuer étant constante par processeur (chaque CPU devait incrémenter un million de fois la variable) on aimerait que le temps total reste constant (ce qui serait une **mise à l'échelle parfaite**, ou **perfect scaling**).

Différentes approches pour résoudre ce problème ont été proposées. Ici, nous allons étudier celle du **approximate counter**. L'idée générale est d'avoir **plusieurs** compteurs logiques locaux (un par CPU) et un compteur **global**. On aura en plus un verrou **local** par CPU, lié aux compteurs locaux (celui-ci n'est nécessaire qu'en cas d'exécution de plusieurs threads par CPU), un verrou **global** pour le compteur global. On va incrémenter "localement" tous les compteurs des CPUs qui se synchroniseront avec leurs verrous locaux respectifs. Comme les threads dans leurs CPUs respectifs peuvent incrémenter leurs compteurs sans contention, la mise à l'échelle sera bonne. Néanmoins, il est nécessaire de synchroniser les compteurs entre les CPUs afin de pouvoir lire la valeur du compteur si nécessaire. On devra donc périodiquement transférer les valeurs des compteurs locaux au compteur global de temps en temps, en additionnant tous les compteurs locaux au compteur global. A ce moment là les compteurs locaux seront remis à zéro.

Le moment où les compteurs locaux seront synchronisés dépendra d'une variable, s. Plus s sera grand, moins la valeur stockée dans s sera précise (plus il y aura de chances qu'elle ne contienne pas la bonne valeur), mais plus le calcul aura une bonne performance sur plusieurs threads. A l'inverse un petit s aura de moins bonnes performances, mais donnera une valeur plus correcte du compteur. Considérons un exemple où on a 4 threads T_1-T_4, avec quatre compteurs locaux L_1 à L_4, et où $s = 4$. Un exemple d'exécution est résumé dans la table ci-dessous

Temps	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	1	0	1	1	0
2	2	0	1	2	0
3	3	1	1	3	0
4	4	1	2	$4 \rightarrow 0$	4 (de L_4)
5	$4 \rightarrow 0$	2	3	0	8 (de L_1)

Les compteurs locaux s'incrémentent chacun à leur vitesse, mais lorsqu'un d'entre eux atteint $s = 4$, sa valeur est ajoutée au compteur global G, et il est remis à zéro. On voit donc pourquoi la valeur est approximée. Au temps 5, le compteur a été incrémenté 13 fois, hors la valeur que nous pouvons lire dans le compteur global est de huit uniquement.

On peut à présent écrire le code suivant pour le compteur approximé :

```
typedef struct __approx_counter_t {
    int glob_counter;           // global counter
    pthread_mutex_t glob_mutex; // global mutex

    int *local_counter;           // local counter array
    int threshold;   // threshold where we communicate
} approx_counter_t;

void init(approx_counter_t *ac, int threshold) {
    ac->threshold = threshold;

    ac->glob_counter = 0;
    pthread_mutex_init(&ac->glob_mutex, NULL);

    ac->local_counter = malloc(num_threads * sizeof(int));
    ac->local_mutex = malloc(num_threads * sizeof(pthread_mutex_t));
    for (int i = 0; i < num_threads; ++i) {
        ac->local_counter[i] = 0;
        pthread_mutex_init(&ac->local_mutex[i], NULL);
    }
}

void increment_by(approx_counter_t *ac, int tid, int amount) {
    ac->local_counter[tid] += amount;
    if (ac->local_counter[tid] >= ac->threshold) {
```

```
        pthread_mutex_lock(&ac->glob_mutex); // verrou global
        ac->glob_counter += ac->local_counter[tid];
        pthread_mutex_unlock(&ac->glob_mutex); // fin verrou global
        ac->local_counter[tid] = 0;
    }
}

int get_counter(approx_counter_t *ac) {
    pthread_mutex_lock(&ac->glob_mutex);
    int value = ac->glob_counter;
    pthread_mutex_unlock(&ac->glob_mutex);
    return value;
}
```

Il faut noter que dans notre structure `approx_counter_t` nous n'avons pas utilisé de verrou local. On suppose ici par simplicité qu'il n'y a qu'un thread par cœur.

On peut à présent mesurer la performance de ce compteur pour un s donné si on incrémente un million de fois un compteur sur un à quatre threads.

	1 thread	2 thread	4 thread
Temps, $s = 1$, $[ms]$	0.053	0.36	0.69
Temps, $s = 10$, $[ms]$	0.012	0.044	0.086
Temps, $s = 100$, $[ms]$	0.008	0.018	0.032
Temps, $s = 1000$, $[ms]$	0.009	0.01	0.012

1.2 La liste chaînée concurrente

Une liste chaînée est une structure de données où un élément de notre liste contient une valeur, ainsi qu'un pointeur vers le prochain élément de la liste (vous avez déjà vu ces choses là en première). Un petit code tout simple implémentant l'insertion en tête de la liste d'un élément entier peut s'écrire de la façon suivante, ainsi qu'une fonction permettant de déterminer si un élément est bien présent dans votre liste peut se trouver ci-dessous :

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;

void init(list_t *l) {
    l->head = NULL;
}

int insert(list_t *l, int key) {
```

```
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        printf("Malloc failed.\n");
        return -1;
    }
    new->key = key;
    l->head = new;
    return 0;
}

int lookup(list_t *l, int key) {
    node_t *current = l->head;
    while (current) {
        if (current->key) {
            return 0;
        }
        current = current->next;
    }
    return -1;
}
```

Il est évident qu'avec ce petit bout de code il est impossible d'insérer et de vérifier la présence ou non d'un élément dans notre liste de façon concurrente.

On veut à présent réécrire ce code pour qu'il soit utilisable dans une application multi-threadée. La première approche serait de protéger le moment de l'insertion, ainsi que la lecture par un verrou qui serait intégré dans notre structure `list_t`. Le code deviendrait

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
    pthread_mutex_t mutex;
} list_t;

void init(list_t *l) {
    l->head = NULL;
    pthread_mutex_init(&l->mutex, NULL);
}

int insert(list_t *l, int key) { // on retourne vrai si tout s'est bien passé
    pthread_mutex_lock(&l->mutex); // début de la section critique
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        printf("Malloc failed.\n");
        pthread_mutex_unlock(&l->mutex); // fin de la section critique
```

```
        return -1; // error
    }
    new->key = key;
    new->next = l->head;
    l->head = new;
    pthread_mutex_unlock(&l->mutex); // fin de la section critique

    return 0;  // success
}

int lookup(list_t *l, int key) {
    pthread_mutex_lock(&l->mutex); // début de la section critique
    node_t *current = l->head;
    while (current) {
        if (current->key == key) {
            pthread_mutex_unlock(&l->mutex); // autre fin de la section critique
            return 0;
        }
        current = current->next;
    }
    pthread_mutex_unlock(&l->mutex); // autre fin de la section critique

    return -1;
}
```

Regardons d'abord la fonction `lookup()`. On constate qu'au tout début on `lock()` notre verrou, afin de protéger la partie où on va vérifier si un élément est présent ou non dans notre liste. Il faut en effet être certain · e · s qu'aucun nouvel élément n'est ajouté pendant que nous lisons, car cela pourrait avoir des effets dramatiques [2]. Le verrou est libéré non seulement lorsqu'on a trouvé l'élément `key` mais également lorsqu'il est absent: il y a deux endroits distincts où il faut penser à faire un `unlock()`. En effet, il faut libérer le verrou **avant** de sortir de la fonction sinon le verrou ne sera jamais déverrouillé...

La fonction `insert()` verrouille le `mutex`, crée l'élément suivant dans la liste et l'insère en tête avant de déverrouiller le `mutex`. Rien de très fou. Néanmoins, il faut remarquer que le verrou doit être libéré dans le cas où `malloc()` retourne une erreur. Bien que cela ne se produise pas souvent, ce cas peut conduire à des threads qui attendent indéfiniment après une erreur d'allocation. De façon générale, il faut être prudent · e lorsqu'on a des branchement conditionnels qui modifient l'exécution d'un programme. Souvent c'est dans ce genre de branchement que se trouvent les erreurs les plus difficiles à détecter.

Cette façon d'écrire une liste chaînée concurrente est simple, mais nous utilisons trop de branchements conditionnels ce qui augmente les chances d'introduire des erreurs. Il est possible de modifier légèrement le code afin d'avoir un plus petit nombre de chemins possibles pour le code, réduisant le nombre de libérations de verrous à effectuer :

2. Plusieurs bébés chats sont morts à la suite de lectures non protégées de listes concurrentes.

```c
void init(list_t *l) {
    l->head = NULL;
    pthread_mutex_init(&l->mutex, NULL);
}

void insert(list_t *l, int key) { // on retourne vrai si tout s'est bien passé
    node_t *new = malloc(sizeof(node_t)); // malloc est thread safe
    if (new == NULL) {
        printf("Malloc failed.\n");
        return; // error
    }
    new->key = key;

    pthread_mutex_lock(&l->mutex); // début de la section critique
    new->next = l->head;
    l->head = new;
    pthread_mutex_unlock(&l->mutex); // fin de la section critique
}

int lookup(list_t *l, int key) {
    int ret_val = -1;
    pthread_mutex_lock(&l->mutex); // début de la section critique
    node_t *current = l->head;
    while (current) {
        if (current->key == key) {
            ret_val = 0;
            break;
        }
        current = current->next;
    }
    pthread_mutex_unlock(&l->mutex); // fin de la section critique

    return ret_val;
}
```

Ici, nous constatons que nous avons simplement enlevé l'acquisition du verrou devant `malloc()`. Cela est possible, car `malloc()` est *thread-safe*. Nous n'avons besoin de verrouiller que lorsqu'on écrit ou qu'on lit dans la liste. Par ailleurs, en ne retournant qu'une seule fois depuis `lookup()`, on s'affranchit de déverrouiller une fois supplémentaire.

Ce genre de liste chaînée n'a pas une grande efficacité lorsqu'on augmente le nombre de threads. Néanmoins, nous n'avons pas réussi à faire beaucoup mieux. Une technique explorée par les chercheurs a été le "hand-over-hand locking". L'idée est la suivante. Au lieu d'avoir un seul verrou pour toute la liste chaînée, on a un verrou par noeud. Lorsqu'on parcourt la liste, on acquière d'abord le verrou du nœud suivant et libère le verrou du nœud courant. Ainsi, la liste peut être parcourue de façon concurrente. Néanmoins, le coût de verrouillage/déverrouillage rend cette façon de faire moins efficace en pratique.

1.3 La file concurrente

De façon similaire à ce que nous avons fait jusque là, nous allons écrire une file concurrente. Faites si vous voulez la partie simple consistant à écrire une file séquentielle, puis à la rendre concurrente à l'aide d'un verrou de la façon la plus triviale possible. Ici, nous allons étudier un algorithme proposé par Michael et Scott [3]. Un code reproduisant leur idée se trouve ci-dessous :

```
typedef struct __node_t {
    int value;
    struct __node_t *next;
} node_t;

typedef struct __queue_t {
    node_t *head;
    node_t *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} queue_t;

void init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}

void enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);

    tmp->value = value;
    tmp->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tail_lock);
}

int dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->head_lock); // attention branchement
```

3. M. Michael, M. Scott, *Nonblocking Algorithms and Preemption-safe Locking on by Multiprogrammed Shared-memory Multiprocessors Journal of Parallel and Distributed Computing*, **51**, No. 1, (1998).

```
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->head_lock); // attention branchement

    free(tmp);
    return 0;
}
```

Dans ce code, on constate qu'il y a deux verrous: un pour la queue et un pour la tête. On peut ainsi avoir une approche suffisamment fine pour enfiler ou défiler de façon concurrente (les deux opérations ne sont pas mutuellement exclusives). Cela est rendu possible par la création d'un nœud fictif lors de la création de la file. Sans lui les fonction `enqueue()` et `dequeue()` ne pourraient avoir lieu de façon concurrente.

1.4 La table de hachage concurrente

A l'aide de la liste concurrente que nous avons implémenté tout à l'heure, il est très simple de créer une table de hachage concurrente (très simple) concurrente: elle sera statique.

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void init(hash_t *h) {
    for (int i = 0; i < BUCKETS; i++) {
        list_init(&h->lists[i]);
    }
}

int insert(hash_t *h, int key) {
    int bucket = key % BUCKETS;
    return list_insert(&h->lists[bucket], key);
}

int lookup(hash_t *h, int key) {
    int bucket = key % BUCKETS;
    return list_lookup(&H->lists[bucket], key);
}
```

On constate que cette table de hachage ne nécessite aucun nouveau verrou. Toutes les sections critiques sont cachées dans les fonctions de la liste chaînée. En effet, la table de hachage n'est rien d'autre qu'un tableau de listes chaînées dans le cas simple où le nombre d'alvéoles est statique.

Exercice 1

Implémenter la table de hachage dynamique.
