TD3 - Exploiter le multi-coeur

D'Andréa William

Exercice 2 - The silent killer

Générateur de nombre aléatoire

```
static unsigned long x, y=362436069, z=521288629;

unsigned long xorshf96(void) {
    unsigned long t;

    x ^= x << 16;
    x ^= x >> 5;
    x ^= x << 1;

    t = x;
    x = y;
    y = z;
    z = t ^ x ^ y;

    return z;
}
```

Question

Pour que le générateur de nombre aléatoire fonctionne correctement, il faut que la graine 'x' (et les variables intermédiaires soient propre à chaque thread). Ce n'est malheureusement pas le cas par défaut s'il est appelé par plusieurs threads en parallèle. On peut cependant le faire fonctionner en ajoutant un seul mot-clé. Lequel ? Indice : regardez ici.

Réponse

Afin que ce générateur de nombre aléatoire fonctionne correctement, nous pouvons utiliser le mot-clé `__thread`, comme ceci :

```
static __thread unsigned long x, y=362436069, z=521288629;
```

Il est à noter que nous aurions également pu utiliser le mot-clé `volatile`, cependant, cela n'aurait pas marché dans notre cas, car `volatile` permet à plusieurs threads d'utiliser la même variable tandis-que le mot-clé `__thread` fait une copie de la variable dans chaque thread.

Question

Créez un fichier exercice1.c contenant cette fonction et ses variables d'initialisation modifiées. Ajoutez un tableau global (results) qui stockera les résultats pour chaque thread.

```
uint64_t results[4];
```

Créez une fonction exécutée par chacune des threads. Celle-ci reçoit en paramètre un thread_id (entier compris entre 0 et 3 pour quatre threads). Le code de celle-ci est le suivant :

- Chaque thread se place sur un coeur différent. Attention : selon les systèmes, il faut soit le programmer (voir pthread_setaffinity_np()), soit cela est pris en charge par le scheduler
- Chaque thread initialise la racine (variable x) de son générateur avec un calcul faisant intervenir son identifiant et le compteur de microsecondes de son timestamp courant (voir gettimeofday()). Inutile de faire un calcul compliqué : addition par exemple.
- Chaque thread appelle la fonction xorshf96() un très grand nombre de fois (par exemple 10^8) et ajoute son résultat modulo 2 à results[thread_id]. Si le résultat est impair (le module 2 est à 1 et la variable results[thread_id] est incrémentée de 1 et si le module 2 est à 0, la variable results[thread_id] est inchangée).

Créez une fonction main() qui crée quatre threads, leur transmet leur identité (i.e. la valeur de thread_id), attend leur terminaison, puis affiche les quatre cases du tableau results.

Exécuter le programme ci-dessus et mesurer son temps d'exécution (voir la commande time). Si tout s'est bien passé, l'exécution du programme devrait prendre entre 5 et 10 secondes, et chaque thread devrait produire environ 10^8/2 nombres aléatoires impairs.

Réponse

```
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>
#include <sys/time.h>

#define THREADS 4
#define ITERATIONS 100000000

static __thread unsigned long x, y=362436069, z=521288629;

uint64_t results[THREADS];

unsigned long xorshf96(void) {
    unsigned long t;

    x ^= x << 16;
    x ^= x >> 5;
    x ^= x << 1;

    t = x;
    x = y;
    y = z;
    z = t ^ x ^ y;

    return z;
}

void* execution(void* thread_id) {
    int typed_thread_id = (int)(intptr_t)thread_id;
```

```
    pthread_t current_thread = pthread_self();
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &typed_thread_id);

    struct timeval now;
    gettimeofday(&now, NULL);
    x = now.tv_usec + typed_thread_id;

    for (int i = 0; i < ITERATIONS; i++) {
        results[typed_thread_id] += xorshf96() % 2;
    }

    return NULL;
}

struct timeval start, end;

int main(int argc, char *argv[]) {

    pthread_t threads[THREADS];
    gettimeofday(&start, NULL);

    for (int i = 0; i < THREADS; i++) {
        pthread_create(&threads[i], NULL, execution, (void*)(intptr_t)i);
    }

    for (int i = 0; i < THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    for (int i = 0; i < THREADS; i++) {
        printf("results[%d] = %llu \n", i, results[i]);
    }
    gettimeofday(&end, NULL);

    printf ("Execution time : %f seconds\n", (double) (end.tv_usec - start.tv_usec) / 1000

    return 0;
}
```

Résultat :

```
results[0] = 50002905
results[1] = 50006672
results[2] = 49998873
results[3] = 50005107
Execution time : 6.745319 seconds
```

Grâce à `pthread_setaffinity_np()`, nous plaçons bien chaque thread sur un coeur différent (nous ne laissons pas la place a la configuration de l'ordonnanceur). Il est important de noter que la méthode `pthread_setaffinity_np()` est disponible sur Linux mais pas forcément sur un autre OS (notamment MacOS). Notre fonction principale va donc créer 4 threads (qui s'exécuteront sur 4 cœurs), ceux-ci vont ensuite générer des nombres aléatoires, grâce à la fonction `xorshf96()`, et si ce nombre est pair, nous

incrémentons sa valeur `results[index_thread]` de 1. C'est donc pour cela que le résultat est proche de `50000000` car la probabilité qu'un nombre aléatoire est de `0.5` , donc `10^8 * 0.5 = 50000000`

Nous obtenons sur Linux un temps d'exécution de 6.74 secondes, ce qui est dans l'intervalle donné.

Question

Le programme que vous avez écrit à un important goulot d'étranglement (bottleneck), lequel ?

Corrigez-le de manière simple. Sur certaines architectures, l'amélioration des performances est drastique. Sur d'autre cela n'a aucun effet. Quelle accélération obtenez-vous par rapport à la version précédente ?

Jouez sur l'optimiseur de compilation : -O1, -O2 ou -O3. Qu'observez-vous ?

Réponse

Nous pouvons remarquer dans l'implémentation précédente que nous cherchons toujours à modifier le tableau results. En effet, il se est modifié par plusieurs threads de manière concurente, cependant, cela n'est pas nécessaire car les tâches de chaques threads sont indépendantes. Nous pouvons donc limité l'utilisation du tableau `results` comme ceci :

```
void* execution(void* thread_id) {
    int typed_thread_id = (int)(intptr_t)thread_id;


    pthread_t current_thread = pthread_self();
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &typed_thread_id);

    struct timeval now;
    gettimeofday(&now, NULL);
    x = now.tv_usec + typed_thread_id;

    uint64_t result = 0;


    for (int i = 0; i < ITERATIONS; i++) {
        result+= xorshf96() % 2;

    }

    results[typed_thread_id] = result;

    return NULL;
}
```

Nous gagnons bel et bien du temps, car, sur 10 tests, nous avons un temps moyen d'éxécution de `1.725297 secondes` , ce qui est 4 fois plus rapide que la précédente implémentation, et nous n'avons quasiment rien changé à notre implémentation!

Test	Normal	-O1	-O2	-O3
1	1.642436	0.650538	0.566137	0.799144
2	1.715748	0.954277	0.606576	0.656794

Test	Normal	-O1	-O2	-O3
3	1.909455	0.925313	0.451366	0.674282
4	1.662293	0.911298	0.601094	0.635380
5	1.802829	0.879658	0.618264	0.613940
6	1.648595	0.729982	0.601170	0.563278
7	1.804651	0.896955	0.744746	0.612524
8	1.702469	0.783709	0.616231	0.689309
9	1.715427	0.801933	0.593642	0.627433
10	1.649063	0.763962	0.652926	0.629075
Total	1.725297	0.829762	0.625215	0.650116

Nous remarquons que grâce à l'optimisation du compilateur, nous obtenons des performances 2 fois supérieur en utilisant l'optimiseur `-O1` , l'optimiseur `-O2` est meilleur que l'optimiseur `-O1` , cependant, l'optimiseur `-O3` est surement trop optimisé (je ne sais pas si l'on peut parler d'over fitting dans cette situation), car il a des performances inférieures à l'optimiseur `-O2`

Question

Dans le but d'estimer la taille des lignes de cache de votre machine, on va modifier le programme précédent pour faire en sorte que la variable résultat de chaque thread soit suffisamment éloignée les unes des autres. Nous allons utiliser pour cela une structure :

```
struct results {
    uint64_t result0;
    char padding1[PADDING];
    uint64_t result1;
    char padding2[PADDING];
    uint64_t result2;
    char padding3[PADDING];
    uint64_t result3;
} __attribute__ ((packed)) results __attribute__ ((aligned (64)));
```

La constante PADDING sera passé à chaque programme lors de la compilation.

Construisez ensuite un script qui fait varier le paramètre PADDING de 1 à 100. Rappel : pour passer un paramètre lors de la compilation on utilise la directive -DPADDING=valeur. On compilera sans aucune optimisation (pas d'option -O1, -O3).

Recueillez les résultats et dans un document Excel (ou autres outils à votre choix), tracez les performances correspondantes. Insérez le graphe dans votre compte-rendu et expliquez-le mieux possible ce que vous y voyez

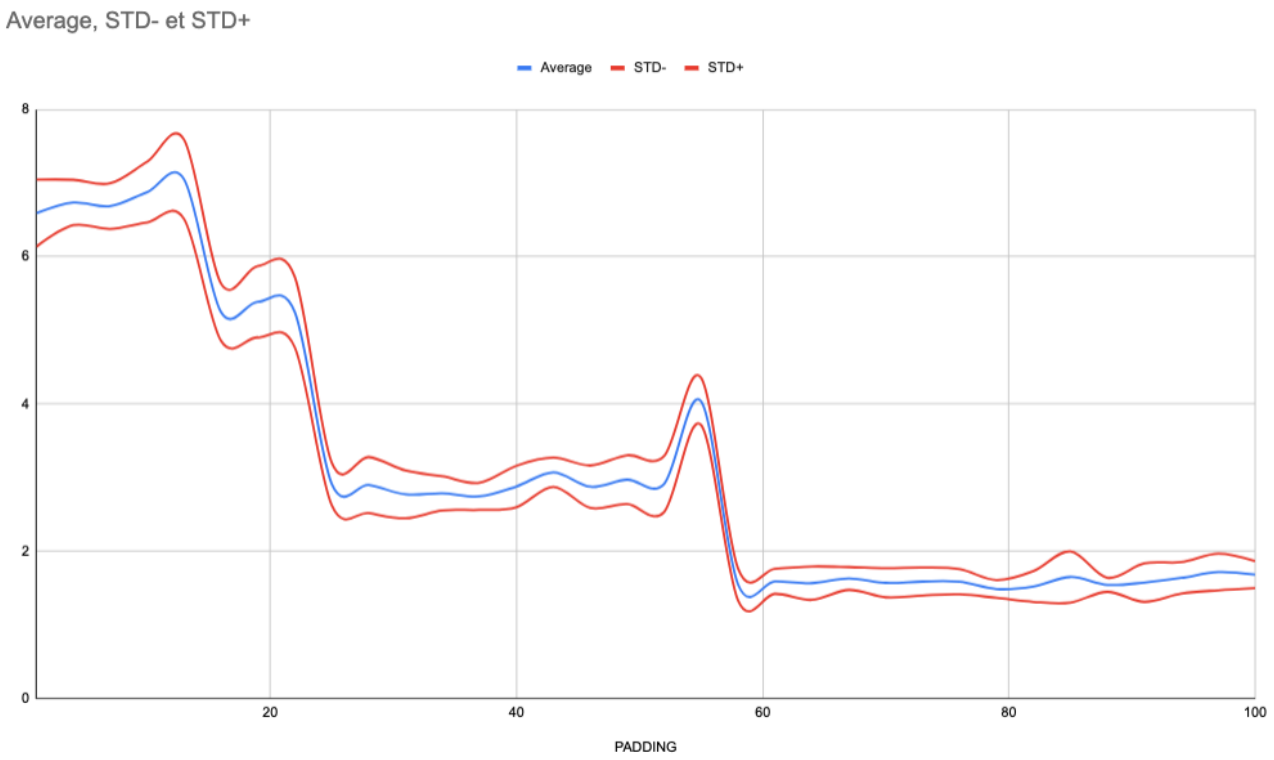
Réponse

```
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>
#include <sys/time.h>

#define THREADS 4
#define ITERATIONS 100000000

// #define PADDING 1

static __thread unsigned long x, y=362436069, z=521288629;

// uint64_t results[THREADS];
typedef struct results {
    uint64_t result0;
    char padding1[PADDING];
    uint64_t result1;
    char padding2[PADDING];
    uint64_t result2;
    char padding3[PADDING];
    uint64_t result3;
} __attribute__ ((packed)) Results __attribute__ ((aligned (64)));

Results results;


unsigned long xorshf96(void) {
    unsigned long t;

    x ^= x << 16;
    x ^= x >> 5;
    x ^= x << 1;

    t = x;
    x = y;
    y = z;
    z = t ^ x ^ y;

    return z;
}

void* execution(void* thread_id) {
    int typed_thread_id = (int)(intptr_t)thread_id;


    pthread_t current_thread = pthread_self();
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &typed_thread_id);


    struct timeval now;
    gettimeofday(&now, NULL);
    x = now.tv_usec + typed_thread_id;



    for (int i = 0; i < ITERATIONS; i++) {
        // result += xorshf96() % 2;
        switch (typed_thread_id) {
            case 0: results.result0 += xorshf96() % 2; break;
```

```
            case 1: results.result1 += xorshf96() % 2; break;
            case 2: results.result2 += xorshf96() % 2; break;
            case 3: results.result3 += xorshf96() % 2; break;
        }
    }

    return NULL;
}

struct timeval start, end;

int main(int argc, char *argv[]) {

    // printf("PADDING : %d \n", PADDING);


    pthread_t threads[THREADS];
    gettimeofday(&start, NULL);

    for (int i = 0; i < THREADS; i++) {
        pthread_create(&threads[i], NULL, execution, (void*)(intptr_t)i);
    }

    for (int i = 0; i < THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    gettimeofday(&end, NULL);

    // printf("results[%d] = %lu \n", 0, results.result0);
    // printf("results[%d] = %lu \n", 1, results.result1);
    // printf("results[%d] = %lu \n", 2, results.result2);
    // printf("results[%d] = %lu \n", 3, results.result3);

    printf ("PADDING : %d, Execution time : %f seconds\n", PADDING, (double) (end.tv_usec -

    return 0;
}
```

Nous avons ensuite executé ce programme avec plusieurs valeurs de PADDING, nous avons d'abord itérer à travers toutes les valeurs entre 1 et 100 avec un pas de 1, nous nous sommes rendus compte qu'il n'était pas nécessaire d'avoir un pas si petit, et nous avons donc préféré itérer à travers un pas de 3, ce qui permet d'avoir une visualisation compléte tout en ne perdant pas trop de temps et pouvoir réaliser plusieurs fois le test afin d'avoir un graphique fiable.

PADDING	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Mediane	Average	STD-	STD+
1	6,692876	6,356996	7,476342	6,40753	6,353649	6,695974	6,140165	6,297503	6,202753	7,282671	6,382263	6,5906459	6,134999928	7,046291872
4	7,022721	6,454957	6,679156	6,949177	6,658081	6,909944	7,160919	6,137941	6,504912	6,878068	6,778612	6,7355876	6,426824537	7,044350663
7	6,579003	6,23053	6,690046	7,279926	6,572924	6,329363	6,738802	6,707495	7,070569	6,67367	6,681858	6,6872328	6,377587464	6,996878136
10	6,836446	7,728749	6,602905	6,317532	6,546859	6,832147	7,118423	6,528252	7,170921	7,081597	6,8342965	6,8763831	6,463478176	7,289288024
13	6,522782	6,598166	6,399701	7,533107	7,714273	6,744549	6,535657	7,319053	7,395551	7,747712	7,031801	7,0510551	6,511910972	7,590199228
16	4,91539	5,441948	4,741488	5,118567	5,632503	5,110634	5,212921	5,627234	5,890379	4,783499	5,165744	5,2474563	4,858480082	5,636432518
19	4,957743	5,736973	4,61753	5,839008	5,863854	5,863889	4,958617	4,903277	5,761963	5,371387	5,55418	5,3874241	4,9025083	5,8723399
22	4,498679	4,794827	5,488552	5,484279	5,199177	5,016198	5,320327	5,64371	6,145357	4,856865	5,259752	5,2447971	4,766767836	5,722826364
25	3,163868	2,694567	3,156056	2,753838	2,843236	2,821468	3,530379	2,764134	2,537943	2,843825	2,832352	2,9109314	2,62090656	3,20095624
28	3,214373	2,631327	3,500553	2,412682	3,332121	2,981785	3,125233	2,569317	2,604364	2,599286	2,806556	2,8971041	2,516986795	3,277221405
31	3,022226	2,545889	3,463857	2,640333	3,075568	2,742022	2,498926	2,489598	2,756484	2,474088	2,6911775	2,7708991	2,446674514	3,095123686
34	3,352095	2,766769	2,729478	2,567827	2,550477	2,723339	2,818143	2,636662	2,756597	2,941794	2,7430375	2,7843181	2,553505986	3,015130214
37	3,00363	2,79685	2,691172	2,945799	2,473789	2,828725	2,531919	2,754032	2,527993	2,883325	2,775441	2,7437234	2,559617298	2,927829502
40	3,080858	2,721256	2,75383	2,287928	2,949847	3,272667	2,831313	2,870499	2,799375	3,212348	2,850906	2,8779921	2,597030724	3,158953476
43	3,315038	3,035426	3,054984	2,787323	3,031632	3,378954	3,29412	2,855654	3,010979	2,946776	3,033529	3,0710886	2,873007474	3,269169726
46	2,98477	2,744672	2,692923	2,840431	2,423213	2,776692	3,471515	2,678365	3,11073	3,026104	2,8085615	2,8749415	2,586255909	3,163627091
49	2,850672	2,880242	3,156849	3,60956	3,068526	2,460981	3,329358	2,693574	2,901823	2,755636	2,8910325	2,9707221	2,638859135	3,302585065
52	3,311185	2,574708	2,828828	3,700083	2,847727	2,992615	3,136322	2,670174	2,531086	2,538628	2,8382775	2,9131356	2,532343129	3,293928071
55	4,23128	3,655542	4,312682	4,120198	4,540822	4,207952	4,119805	3,824387	3,517316	3,810681	4,1200015	4,0340665	3,713909379	4,354223621
58	1,768592	1,371113	1,371495	1,986473	1,583176	1,376397	1,729173	1,533778	1,381245	1,373577	1,4575115	1,5475019	1,330118107	1,764885693
61	1,767356	1,376185	1,613338	1,865761	1,504467	1,522163	1,790131	1,499525	1,583339	1,378465	1,552751	1,590073	1,420528491	1,759617509
64	2,077924	1,486659	1,372953	1,809987	1,497213	1,365593	1,428145	1,648333	1,399829	1,560923	1,491936	1,5647559	1,337829966	1,791681834
67	1,821392	1,383704	1,507182	1,851866	1,683892	1,546772	1,570005	1,676002	1,479415	1,763817	1,6230035	1,6284047	1,472874919	1,783934481
70	1,879643	1,493609	1,49754	1,975246	1,585187	1,447712	1,420922	1,435681	1,541514	1,416842	1,4955745	1,5693896	1,371932126	1,766847074
73	1,990816	1,365498	1,50155	1,628171	1,459154	1,439027	1,551338	1,802967	1,479921	1,647748	1,526444	1,586619	1,397340791	1,775897209
76	1,906045	1,576886	1,374748	1,598612	1,71256	1,42618	1,63655	1,736479	1,416793	1,445088	1,587749	1,5829941	1,41222646	1,75376174
79	1,765694	1,376909	1,517265	1,425451	1,453021	1,451236	1,619407	1,43259	1,425261	1,382534	1,441913	1,4849368	1,363783728	1,606089872
82	1,664397	1,297273	1,3216	1,646872	1,40439	1,361196	1,667176	1,438992	1,426464	1,958602	1,432728	1,5186962	1,30951047	1,72788193
85	2,491978	1,297752	1,376292	1,293637	1,520468	1,766193	1,627746	1,61278	1,786629	1,710546	1,620263	1,6484021	1,300490241	1,996313959
88	1,783495	1,474962	1,535414	1,562332	1,473564	1,454481	1,478819	1,573207	1,556805	1,525315	1,5303645	1,5418394	1,446926112	1,636752688
91	1,996597	1,430076	1,40503	1,345921	1,551491	1,465582	1,427675	1,529146	2,111428	1,464037	1,4648095	1,5726983	1,31098153	1,83441507
94	1,890859	1,311494	1,592284	1,493213	1,69796	1,973206	1,571885	1,473503	1,862112	1,491758	1,5820845	1,6358274	1,421726827	1,849927973
97	1,906703	2,059554	1,797057	1,599786	1,528668	1,761627	1,351417	1,573621	2,089785	1,49052	1,6807065	1,7158738	1,467457291	1,964290309
100	1,96249	1,645966	1,751287	1,42196	1,524293	1,621033	1,555552	1,686288	2,001371	1,637217	1,6415915	1,6807457	1,497874113	1,863617287



Nous remarquons que le temps d'exécution baisse par palier, en effet, jusqu'à environ 15 de PADDING nous avons un temps moyen constant de 6.5 / 7 secondes, ensuite un deuxième pallier jusqu'à environ 30 de PADDING, et 2 derniers palliers dont le plus grand arrive à environ 60 de PADDING, ensuite, nous avons un temps d'exécution constant d'environ 1.5 seconde.

L'amélioration de la performance en fonction de l'augmentation du `PADDING` est dû au fait que l'on augmente la taille des espaces vides entre les différentes valeurs `results.result0` , `results.result1` ...

En effet, afin de réduire le temps d'execution, le processeur va stocker en cache une copie des données lues dans la mémoire principale, et, ce cache est organisé en lignes de mémoires qui sont de taille fixe. Et, quand on augmente la value PADDING, on répartit les données sur plusieurs lignes de cache, ce qui réduit

le nombre de collision de cache. (à l'inverse du tableau results[] que l'on avait précédemment, car plus la variable PADDING est grande, plus on peut considérer que chaque variable de results sont indépendantes, donc facilement editable par chaque thread).

Exercice 3 - Du faux partage en Java ?

Question

Créez une classe Exercice3. Dans celle-ci, créez une classe interne statique ThreadInfo qui contiendra un volatile long progress qui stockera l'avancement d'un thread, et un long result qui stockera le résultat d'un thread.

Créez ensuite une variable infos, attribut de la classe Exercice3, de type 'tableau ThreadInfos' qui sera de taille 4 (static ThreadInfo[] infos = new ThreadInfo[4])

Le reste de l'implémentation est similaire à celle de l'exercice précédent. Implémentez la fonction xorshf96() en Java, puis créez une fonction main() qui crée quatre threads. Ces threads effectueront chacun 10^8 itérations, dans lesquelles ils appelleront xorshf96(), ajouteront le résultat modulo 2 à result, et incrémenteront progress.

Une fois le programme décrit ci-dessus implémenté, lancez-le. Quel est le temps d'exécution ? Où se trouve le false sharing potentiel ?

Réponse

```
public class Exercice3 {

    public static int THREADS = 4;
    public static int ITERATIONS = 100000000;
    public static ThreadInfo[] infos = new ThreadInfo[4];

    static class ThreadInfo {
        volatile public long progress;      // avancement d'un thread
        public long result;                 // résultat d'un thread
    }


    public static Map<String, Long> xorshf96(long x, long y, long z) {
        long t;

        x ^= x << 16;
        x ^= x >>> 5;
        x ^= x << 1;

        t = x;
        x = y;
        y = z;
        z = t ^ x ^ y;

        Map<String, Long> map = new HashMap<String, Long>();
        map.put("x", x);
        map.put("y", y);
        map.put("z", Math.abs(z));
```

```
        return map;
    }

    public static void main(String[] args) throws InterruptedException {

        Thread[] threads = new Thread[THREADS];

        long startTime = System.currentTimeMillis();

        for (int threadId = 0; threadId < THREADS; threadId++) {
            int finalThreadId = threadId;

            infos[finalThreadId] = new ThreadInfo();

            threads[finalThreadId] = new Thread(() -> {

                long x = 0L;
                long y = 362436069L;
                long z = 521288629L;

                for (int j = 0; j < ITERATIONS; j++) {
                    Map<String, Long> result = xorshf96(x, y, z);
                    x = result.get("x");
                    y = result.get("y");
                    z = result.get("z");

                    long xorValue = result.get("z");
                    long modulo = xorValue % 2;
                    infos[finalThreadId].result += modulo;
                    infos[finalThreadId].progress += 1;
                }
            });
            threads[finalThreadId].start();
        }

        for (int i = 0; i < THREADS; i++) {
            threads[i].join();
        }

        for (int i = 0; i < THREADS; i++) {
            System.out.println("result[" + i + "] = " + infos[i].result + " | Progress : "
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Execution time: " + (endTime - startTime) / 1000.0 + " seconds'
    }

}
```

NB : Entre la version C et la version Java, j'ai changé de machine, je suis passé d'une machine virtuelle sous Linux (Ubuntu) à une machine MacOS.

Dans la méthode `xorshf96()` il a été nécessaire de rajouter `Math.abs` à la valeur retournée car il arrivait que des valeurs soient négatives, ce qui trompait le résultat.

Voici le résultat de l'exécution :

```
result[0] = 50005303 | Progress : 100000000
result[1] = 50005303 | Progress : 100000000
result[2] = 50005303 | Progress : 100000000
result[3] = 50005303 | Progress : 100000000
Execution time: 26.506 seconds
```

Nous pouvons directement voir que le temps d'éxecution est bien plus important que lors de l'execution d'un code C.

Le false-sharing se situe probablement dans la variable `public static ThreadInfo[] infos` , en effet, celle-ci va faire un ping-pong entre les différents threads. La ligne de cache associé va naviguer entre les différents threads ce qui peut amener à un ralentissement.

Question

Reprenez la classe Exercice3 et ajoutez le mot clé volatile devant la variable en faux partage. Dites-en quoi, sa signification en Java est un peu différente de celle en C.

Une fois le programme décrit ci-dessus implémenté, lancez-le. Qu'observez-vous ?

Réponse

```
volatile public static ThreadInfo[] infos = new ThreadInfo[4];
```

Nous avons juste à changer l'attribue `infos` en y mettant `volatile` devant. En java, l'utilisation de `volatile` est utile lorsque plusieurs threads cherchent à accéder à une même variable. En mettant volatile devant, on est assuré que chaque thread aura la dernière valeur connue (la plus a jour). Cependant, en C, l'utilisation de `volatile` indique au compilateur que celui-ci ne doit pas faire d'optimisation de cache autour de la variable déclaré `volatile` , car en général nous utilisons le mot-clé volatile pour permettre à des processus externe de modifier la variable. En Java, `volatile` permet plus de dire "a chaque accès de la variable volatile, j'ai la bonne valeur", tandis que en C, c'est plutot "le compilateur ne doit pas optimiser le cache de la variable volatile".

Quand on lance le programme, il n'y a pas de différence notoire de performance, nous restons autour de 21 secondes sur notre machine. Cela veut potentiellement dire que, à la base, le transfert de cache entre chaque thread est déjà implicitement fait, sans la nécessité de mettre volatile.

Question

Ajoutez du padding pour éviter le false sharing potentiel. Observez-vous une amélioration des performances ? Si vous n'en observez pas, faites en sorte que les variables de padding soient utilisées artificiellement quelque part pour vous assurer que Java ne les supprime pas. Les performances sont-elles maintenant améliorées ?

Réponse

```
public class Exercice3 {

    public static int THREADS = 4;
```

```
public static int ITERATIONS = 100000000;
volatile public static ThreadInfo[] infos = new ThreadInfo[4];

static class ThreadInfo {
    volatile public long progress;      // avancement d'un thread
    public long result;                 // résultat d'un thread
    public int[] padding = new int[64];

    public void usePadding() {this.padding[0] = 0;}
}


public static Map<String, Long> xorshf96(long x, long y, long z) {
    long t;

    x ^= x << 16;
    x ^= x >>> 5;
    x ^= x << 1;

    t = x;
    x = y;
    y = z;
    z = t ^ x ^ y;

    Map<String, Long> map = new HashMap<String, Long>();
    map.put("x", x);
    map.put("y", y);
    map.put("z", Math.abs(z));

    return map;
}

public static void main(String[] args) throws InterruptedException {

    Thread[] threads = new Thread[THREADS];

    long startTime = System.currentTimeMillis();

    for (int threadId = 0; threadId < THREADS; threadId++) {
        int finalThreadId = threadId;

        infos[finalThreadId] = new ThreadInfo();

        threads[finalThreadId] = new Thread(() -> {

            long x = 0L;
            long y = 362436069L;
            long z = 521288629L;

            for (int j = 0; j < ITERATIONS; j++) {
                Map<String, Long> result = xorshf96(x, y, z);
                x = result.get("x");
                y = result.get("y");
                z = result.get("z");

                long xorValue = result.get("z");
                long modulo = xorValue % 2;
                infos[finalThreadId].result += modulo;
                infos[finalThreadId].progress += 1;
                infos[finalThreadId].usePadding();
```

```
                }
            });
            threads[finalThreadId].start();
        }

        for (int i = 0; i < THREADS; i++) {
            threads[i].join();
        }

        for (int i = 0; i < THREADS; i++) {
            System.out.println("result[" + i + "] = " + infos[i].result + " | Progress : "
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Execution time: " + (endTime - startTime) / 1000.0 + " seconds'
    }

}
```

```
result[0] = 50005303 | Progress : 100000000
result[1] = 50005303 | Progress : 100000000
result[2] = 50005303 | Progress : 100000000
result[3] = 50005303 | Progress : 100000000
Execution time: 18.699 seconds
```

Afin d'être sûr que le padding ne se supprime, pas, dans la fonction `main`, nous avons appelé la méthode `usePadding()`

En ajoutant un padding, nous obtenons une amélioration du temps execution, nous passons d'environ 27/28 secondes à 18/19 secondes. Le padding joue donc un rôle dans la performance. Il faut ainsi y faire attention car, en Java, nous pourrions être tentés de ne pas penser à ce genre de problèmes dans la mesure où nous n'interagissons pas avec les emplacements mémoires.

J'aimerais juste finalement revenir sur le mot-clé `volatile`. En effet, en analysant notre code, nous nous rendons compte qu'il n'est pas forcément nécessaire d'instancier la variable `infos` en volatile, car l'utilisation de volatile sur celle-ci n'apporte pas de gain de performance significatif, parce que chaque thread ne modifie qu'un seul élément de l'array `infos`. Les threads ne modifient pas simultanément la même variable, donc on n'a pas de besoin de synchronisation pour garantir une mise à jour correcte.

Je pense également qu'il n'est pas nécessaire de mettre `progress` en volatile, car chaque thread incrémente seulement sa propre variable `progress`, et pas celle d'un autre thread.