Systèmes d'exploitation

TP 4 - Mémoire - Correction

2023-04-05

Exercice : déterminer la taille du TLB

Cet exercice est adapté d'un exercice d'Operating Systems: Three Easy Pieces.

Dans ce devoir, vous devez mesurer la taille et le coût de l'accès du TLB. L'idée est basée sur les travaux de Saavedra-Barrera [SB92], qui a développé une méthode simple mais efficace pour mesurer de nombreux aspects des hiérarchies de caches, le tout avec un programme utilisateur très simple.

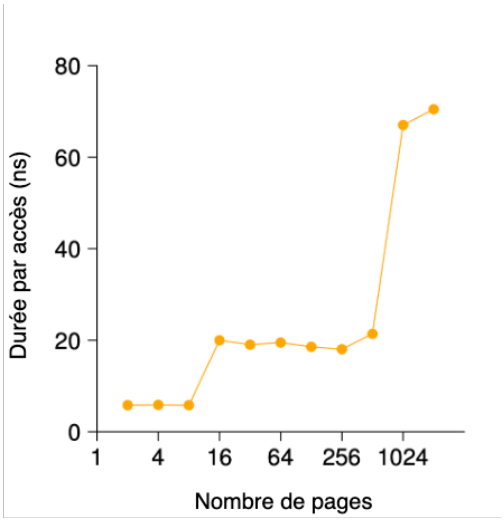
L'idée de base est d'accéder à un certain nombre de pages à l'intérieur d'une grande structure de données (par exemple, un tableau) et de chronométrer ces accès. Par exemple, disons que la taille du TLB d'une machine soit de 4 (ce qui serait très petit, mais utile dans le cadre de cette discussion). Si vous écrivez un programme qui touche 4 pages ou moins, chaque accès devrait être un hit TLB, et donc relativement rapide. Cependant, dès que vous touchez 5 pages ou plus, de manière répétée dans une boucle, le coût de chaque accès augmentera soudainement pour devenir celui d'un TLB miss.

Le code de base pour parcourir un tableau en boucle une fois devrait ressembler à ceci :

```
int jump = PAGESIZE / sizeof(int);

for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

Dans cette boucle, un entier par page du tableau `a` est mis à jour, jusqu'au nombre de pages spécifié par `NUMPAGES`. En chronométrant une telle boucle de manière répétée (par exemple, quelques centaines de millions de fois dans une autre boucle autour de celle-ci, ou le nombre de boucles nécessaires pour fonctionner pendant quelques secondes), vous pouvez mesurer le temps que prend chaque accès (en moyenne). En observant les augmentations de coût au fur et à mesure que `NUMPAGES` augmente, vous pouvez déterminer approximativement la taille de la TLB de premier niveau, déterminer s'il existe une TLB de second niveau (et quelle est sa taille si c'est le cas) et, en général, avoir une bonne idée de la façon dont les hits et les misses de la TLB peuvent affecter les performances.

Le graphique montre le temps moyen par accès lorsque le nombre de pages accédées dans la boucle augmente. Comme vous pouvez le voir sur le graphique, lorsque quelques pages seulement sont accédées (8 ou moins), le temps d'accès moyen est d'environ 5 nanosecondes. Lorsque l'on accède



à 16 pages ou plus, le temps d'accès passe brusquement à environ 20 nanosecondes par accès. Une dernière augmentation du coût se produit aux alentours de 1024 pages, où chaque accès prend environ 70 nanosecondes. Ces données nous permettent de conclure qu'il existe une hiérarchie TLB à deux niveaux ; le premier est assez petit (il contient probablement entre 8 et 16 entrées) ; le second est plus grand mais plus lent (il contient environ 512 entrées). La différence globale entre les *hits* dans le TLB de premier niveau et les *misses* est est assez importante, environ un facteur de quatorze. Les performances du TLB sont importantes !

1. Écrivons un programme, appelé `tlb.c` pour mesurer approximativement le coût d'accès à chaque page. Les données d'entrée du programme sont : le nombre de pages à toucher et le nombre d'essais.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "invalid number of arguments\n");
        return EXIT_FAILURE;
    }

    int pages = atoi(argv[1]);
    int trials = atoi(argv[2]);
    if (pages <= 0 || trials <= 0) {
        fprintf(stderr, "invalid input\n");
        return EXIT_FAILURE;
    }

    long PAGESIZE = sysconf(_SC_PAGESIZE);
```

```
    int *a = (int *) malloc(pages * PAGESIZE);

    long jump = PAGESIZE / sizeof(int);

    struct timespec start, end;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);

    for (int j = 0; j < trials; j++) {
        for (int i = 0; i < pages * jump; i += jump) {
            a[i] += 1;
        }
    }

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);

    // in nanoseconds
    float totalDuration = (end.tv_sec - start.tv_sec) * 1E9 + end.tv_nsec -
     ↪  start.tv_nsec;
    float meanDuration = totalDuration / (trials * pages);

    printf("%f\n", meanDuration);

    free(a);
    return EXIT_SUCCESS;
}
```

2. Pourquoi utiliser `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)` pour chronométrer le temps d'exécution de la boucle ? Quelle est la résolution de `CLOCK_PROCESS_CPUTIME_ID` sur votre système (pour le savoir, utilisez la fonction `clock_getres`) ?

`CLOCK_PROCESS_CPUTIME_ID` est une horloge qui mesure le temps CPU d'un processus donné. Ainsi, seul sera pris en compte la durée pendant laquelle le processus est en train de s'exécuter et pas celle durant laquelle le processus est en attente.

`CLOCK_PROCESS_CPUTIME_ID` a une résolution de 1000 nanosecondes sur macOS et de 1 nanoseconde sur Linux. On peut déterminer la résolution de l'horloge grâce au programme ci-dessous.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void ) {
    struct timespec res;

    if (clock_getres( CLOCK_REALTIME, &res) == -1) {
      return EXIT_FAILURE;
```

```
    }

    printf( "Resolution is %ld nano seconds.\n", res.tv_nsec );
    return EXIT_SUCCESS;
}
```

3. Nous devons nous assurer que les optimisations du compilateur ne faussent pas nos mesures. Les compilateurs font toutes sortes de choses intelligentes, y compris supprimer les boucles qui incrémentent des valeurs qu'aucune autre partie du programme n'utilise par la suite. Quelles options de compilations utiliser pour que le compilateur ne supprime pas la boucle principale ci-dessus de l'estimateur de la taille du TLB ?

On peut utiliser l'option `-O0` pour désactiver l'optimisation avec GCC. Il s'agit du paramètre par défaut.

4. Il faut également tenir compte du fait que la plupart des systèmes actuels possèdent plusieurs processeurs, et que chaque processeur possède sa propre hiérarchie TLB. Pour obtenir de bonnes mesures, vous devez exécuter votre code sur un seul processeur, au lieu de laisser l'ordonnanceur le faire passer d'un processeur à l'autre. Comment faire ? (hint: cherchez *"pinning a thread"* sur Google). Que se passera-t-il si vous ne le faites pas et que le code passe d'un processeur à l'autre ?

Il y a de multiples façons de réaliser cela sur Linux. On peut utiliser les outils suivants `sched_setaffinity(2)`, `pthread_setaffinity_np(3)`, `taskset(1)` ou `sudo systemd-run -p AllowedCPUs=0 ...`. Sur macOS, ce n'est vraiment possible de faire cela.

5. Un autre problème qui peut se poser concerne l'initialisation. Si vous n'initialisez pas le tableau `a` avant d'y accéder, la première fois que vous y accéderez sera très coûteuse, en raison des coûts d'accès initiaux. En effet, avec la *demand zeroing* les pages ne sont chargées dans la mémoire que lorsque le processus en cours d'exécution cherche à y accéder (pour en savoir plus voir Operating Systems: Three Easy Pieces - Chapitre 23). Une solution pour régler ce problème est d'allouer la mémoire avec la fonction calloc au lieu de malloc.

```
int *a = (int *) calloc(pages, PAGESIZE);
```

6. Vous devriez maintenant avoir mis à jour le programme pour régler les problèmes indiqués ci-dessus. Écrivez alors un script dans votre langage de script favori (Python ?) pour exécuter ce programme, tout en faisant varier le nombre de pages accédées de 1 à environ 9000, en doublant le nombre de pages à chaque itération. Pour obtenir des mesures fiables vous devrez sans doute fixer le paramètre trials à une valeur élevée (`~100000`).

```
import argparse
import platform
import subprocess

import matplotlib.pyplot as plt
```

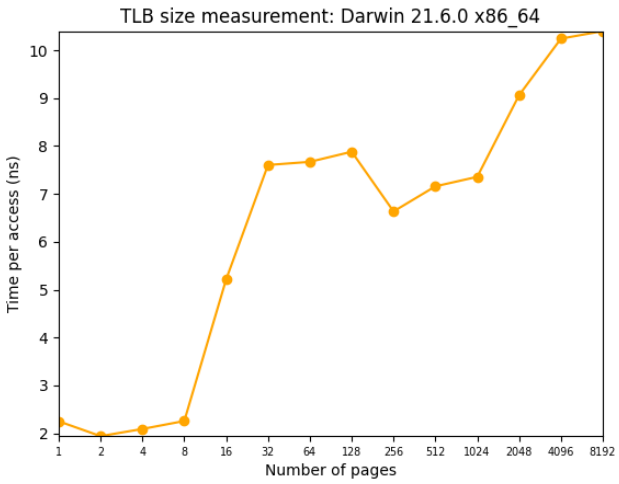
```
parser = argparse.ArgumentParser()
parser.add_argument("max_pages", type=int)
parser.add_argument("trials")
args = parser.parse_args()

results = []
pages = []
x_axis = []

idx = 0
page = 1
while page <= args.max_pages:
    r = subprocess.run(
        ["./tlb.out", str(page), args.trials],
        capture_output=True,
        check=True,
        text=True,
    )
    res = float(r.stdout)
    print("page: ", page, "time:", res)

    pages.append(page)
    results.append(res)
    x_axis.append(idx)

    idx += 1
    page *= 2

plt.plot(x_axis, results, marker="o", color="orange")
plt.margins(0)
plt.xticks(x_axis, pages, fontsize="x-small")
plt.xlabel("Number of pages")
plt.ylabel("Time per access (ns)")
title = "TLB size measurement: "
title += f"{platform.system()} {platform.release()} {platform.machine()}"
plt.title(title)
plt.show()
```

7. Une fois que vous avez collecté des données, représentez les résultats sous forme de graphique, en créant un graphique similaire à celui ci-dessus. Utilisez un outil comme matplotlib ou ploticus. La visualisation rend généralement les données plus faciles à assimiler. Avec une inspection visuelle, vous devriez alors pouvoir déterminer, la taille, le nombre et la performance des TLBs présents sur votre machine.

On lance le script Python présenté ci-dessus de la manière suivante :

```
python plot.py 8192 100000
```

On obtient alors le résultat suivant :



Ce graphique nous permet d'identifier une hiérarchie TLB à deux niveaux ; le premier TLB est assez petit (il contient probablement entre 8 et 16 entrées) ; le second est plus grand mais plus lent (il contient environ 1024 entrées).