Systèmes d'exploitation

TP 4 - Mémoire

2023-04-05

1 Exercice : mémoire virtuelle

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

Dans cet exercice, nous allons simplement découvrir quelques outils utiles pour examiner l'utilisation de la mémoire virtuelle sur les systèmes basés sur Linux. Il ne s'agit que d'un bref aperçu de ce qui est possible ; vous devrez approfondir par vous-même pour devenir un véritable devenir un expert (comme toujours !).

1. Le premier outil Linux que vous devez découvrir est l'outil `free`. Tout d'abord, tapez `man free` et lisez la page du manuel, elle est courte, ne vous inquiétez pas !

   Vous devriez y lire des références à la *"swap memory"*. Celle-ci se définit de la façon suivante :

   > L'espace d'échange, aussi appelé par son terme anglais swap space ou simplement swap, est une zone d'un disque dur faisant partie de la mémoire virtuelle de votre ordinateur. Il est utilisé pour décharger la mémoire vive physique (RAM) de votre ordinateur lorsque celle-ci arrive à saturation. L'espace d'échange, dans Ubuntu, se trouve généralement sous une forme de partition de disque dur – on parle alors de partition d'échange. Il peut aussi se présenter sous forme de fichier – on parle alors de fichier d'échange.
   >
   > https://doc.ubuntu-fr.org/swap

2. Maintenant, lancez `free`, en utilisant éventuellement certains arguments qui pourraient être utiles (par exemple, `--mega`, pour afficher les totaux de mémoire en mégaoctets). Quelle est la quantité de mémoire de votre système ? Quelle est la quantité de mémoire libre ?

   Notez bien la différence entre `free` et `available`. La mémoire `free` n'est pas utilisée et reste là à ne rien faire. Tandis que `available` est la mémoire utilisée ou non (comprend notamment les caches) qui peut être libérée sans que les performances soient pénalisées par l'utilisation de l'espace d'échange (*swap memory*). Pour en savoir plus, consultez https://www.linuxatemyram.com/.

3. Ensuite, créons un petit programme qui utilise une certaine quantité de mémoire, appelé `memory-user.c`. Ce programme doit prendre un argument en ligne de commande : le nombre de mégaoctets de mémoire qu'il va utiliser. Lorsqu'il est exécuté, il doit allouer un tableau, et parcourir constamment le tableau, en modifiant chaque entrée. Le programme doit faire cela indéfiniment pendant un certain temps également spécifié sur la ligne de commande.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/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 numBytes = atoi(argv[1]) * 1024 * 1024;
    int duration = atoi(argv[2]);

    // char has a size of 1 byte
    char *arr = (char *) malloc(numBytes * sizeof(char));

    struct timeval start;
    gettimeofday(&start, NULL);

    struct timeval current = start;

    while (current.tv_sec - start.tv_sec < duration) {
        for (int i = 0; i < numBytes; i++) {
            arr[i] += 1;
        }

        gettimeofday(&current, NULL);
        printf("running...\n");
    }

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

4. Maintenant, tout en exécutant votre programme utilisateur de mémoire, lancez également (dans une autre fenêtre de terminal, mais sur la même machine) l'outil `free`. Comment les totaux d'utilisation de la mémoire changent-ils lorsque votre programme est en cours d'exécution ? Qu'en est-il lorsque vous arrêtez le programme utilisateur de mémoire ? Les chiffres correspondent-ils à vos attentes ? Essayez ceci pour différentes quantités de mémoire utilisées. Que se passe-t-il lorsque vous utilisez grandes quantités de mémoire ?

5. Essayons un autre outil, connu sous le nom de `pmap`. Prenez le temps de lire en détail la page de manuel de `pmap`.
6. Pour utiliser pmap, vous devez connaître l'ID du processus qui vous intéresse. Pour cela, lancez d'abord `ps auxw` pour obtenir la liste de tous les processus ; puis choisissez un processus intéressant, tel qu'un navigateur. Vous pouvez également utiliser votre programme d'utilisateur de la mémoire dans ce cas (en effet, vous pouvez même demander à ce programme d'appeler `getpid()` et d'imprimer son PID pour vous pour vous faciliter la tâche).
7. Exécutez maintenant pmap sur certains de ces processus, en utilisant différentes options (comme `-X`) pour révéler de nombreux détails sur le processus. Que voyez-vous ? Combien d'entités différentes composent un espace d'adressage moderne, contrairement à notre conception simple du code, de la pile et du tas ?
8. Enfin, lançons pmap sur votre programme `memory-user.c`, avec différentes quantités de mémoire utilisée. Que voyez-vous ici ?

2 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.

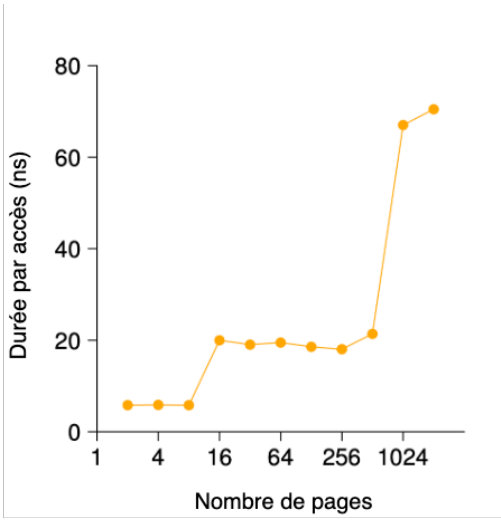
Le code de base pour parcourir un tableau en boucle une fois devrait ressembler à ceci :

```
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. Ecrivons 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`) ?

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 ?

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 ?

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.
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`).
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.

3 Références

[SB92] "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking" by Rafael H. Saavedra-Barrera. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-92-684, February 1992.

À retenir

- L'espace d'échange, aussi appelé *swap space* ou simplement *swap*, est une zone du disque dur faisant partie de la mémoire virtuelle. Il est utilisé pour décharger la mémoire vive physique (RAM) lorsque celle-ci arrive à saturation.
- La commande `free` affiche la quantité totale de mémoire physique et *swap* libre et utilisée dans le système.
- La commande `pmap` renseigne sur la composition de l'espace d'adressage d'un processus.
- Le ***translation lookaside buffer (TLB)*** est un cache matériel qui contient les fréquentes traductions d'adresses virtuelles en adresses physiques.
- Les systèmes récents possèdent souvent une **hiérarchie de TLBs**.