INF1600 : TP5

Assembleur en ligne et mémoire

Imane Hafnaoui <imane.hafnaoui@polymtl.ca>
Giovanni Beltrame <giovanni.beltrame@polymtl.ca>

École Polytechnique de Montréal

Automne 2014

Introduction

Dans Pour la majorité d'entre vous, cet ultime travail pratique est la dernière occasion de faire un peu d'assembleur, en particulier si vous visez le génie logiciel. Pour les autres, vous retoucherez à ce « langage » seulement si vous y tenez vraiment. Cette connaissance du fonctionnement interne des processeurs cous fera toutefois réfléchir différemment quand viendra le temps de solutionner des problèmes de performance. Profitez-en !

Dans ce TP, vous allez écrire un minimum d'assembleur en ligne. Le reste du TP sera réservé à la compréhension de la mémoire d'un micro-ordinateur.

Remise

Voici les détails concernant la remise de ce travail pratique :

- **méthode** : sur Moodle (une seule remise par équipe) ;
- **échéance** : avant 23h50, le 7 décembre 2014 pour les groupes du Lundi B1, le 14 décembre 2014 pour les groupes du Lundi B2.
- **format** : le rapport en format PDF et le fichier C sous les noms suivant : TP5_matricule1_matricule2.pdf et tp5.c ; le rapport doit comporter une page titre ou figure les noms et matricules des membres du groupe.
- **langue écrite** : français, English ;
- **distribution** : les deux membres de l'équipe recevront la même note.

Barème

Contenu	***Points du cours***
Assembleur en ligne	3
Mémoire – compréhension	3
Mémoire – cache	4
Langage écrit erroné	jusqu'à -1
Format de remise erroné (irrespect des noms de fichiers demandé, fichiers superflus, etc.)	jusqu'à -1
Retard	-1 par jour

Exercice 1 : assembleur en ligne

Nous voulons concevoir un programme qui communique par le réseau via le protocole IP. Pour envoyer des informations binaires sur le réseau, le protocole IP impose l'ordre dans lequel les octets doivent être envoyés sur le réseau (*Network Byte Order*). Cet ordre est selon la convention *big-endian*, c'est-à-dire l'octet de poids le plus fort doit premier.

Malheureusement comme vous le savez, les processeurs x86 fonctionnent en mode *little-endian*. Nous avons donc besoin d'une routine pour convertir des nombres de 32 bits stockés sur notre machine en *big-endian* vers une représentation *little-endian*. Une façon de faire en C serait la suivante :

```
unsigned int little_endian_to_big_endian(unsigned int le) {
      return le >> 24 | (le & 0xff0000) >> 8 | (le & 0xff00) << 8 | (le & 0xff) << 24;
}
```

Malheureusement le compilateur gcc génère trop d 'instructions pour que notre programme ait de bonnes performances. Vous pouvez voir le code généré en tapant la commande :

```
gcc -m32 -S tp5.c
```

Un fichier `tp5.s` sera généré, il contient le code assembleur correspondant au programme.

Votre travail est d 'implémenter la routine `little_endian_to_big_endian` plus efficacement que le compilateur (en utilisant moins d'instructions) via quelques lignes d'assembleur en ligne dans le code C.

Le fichier `tp5.c` qui contient la routine à modifier vous est fourni. Vous pouvez compiler le programme avec la ligne de commande suivante :

```
gcc -Wall -m32 -gdwarf-2 -o tp5 tp5.c
```

et l'exécuter en faisant `./tp5` pour contrôler que votre implémentation est correcte.

Une bonne référence pour l'assembleur en ligne est GCC-Inline-Assembly-HOWTO : http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.

Exercice 2 : mémoire – compréhension

Répondez à chacune des questions suivantes et **justifiez vos réponses**.

1. Comment est-il possible, sous Linux, que deux programmes exécutés en même temps et qui écrivent tout deux à l'écran l'adresse à laquelle ils ont été chargés, affichent la même adresse ?

2. Que se passe-t-il, en ordre, entre le moment où un logiciel déréférence un pointeur invalide et le moment où vous voyez une erreur de segmentation ? On veut ici les étapes menant à cette erreur.
3. Si les registres d'un processeur sont tellement plus performants que les autres types de mémoire volatile, pourquoi ne pas en inclure des centaines, voire des milliers dans chaque processeur ?
4. Quel serait le risque d'avoir un serveur avec un système d'exploitation gérant un processeur sans MMU ?
5. Les bibliothèques partagées (shared library : fichiers .so sur Linux, .dll sur Windows) permettent à plusieurs processus d'utiliser les mêmes routines chargées en mémoire principale. Autrement dit, le même code peut être chargé à un seul endroit en mémoire principale, mais utilisé par des dizaines de processus différents. Comment ceci est-il possible, sachant que chaque processus ne « voit » que son propre espace mémoire ?
6. Une des optimisations très connues réalisée par les compilateurs est le déroulement de boucle. Ceci correspond simplement à copier/coller n fois l'intérieur de la boucle pour une boucle de n itérations plutôt que d'effectivement boucler, évitant ainsi les comparaisons et branchements conditionnels pour économiser des cycles. Considérant ce que vous connaissez sur la mémoire, est-ce que le déroulement de boucle est toujours optimal ? Pourquoi ?
7. Un système d'exploitation a-t-il accès aux adresses physiques de la mémoire (plutôt qu'avoir seulement accès aux adresses virtuelles comme les processus) ? Pourquoi ?
8. Nommez une situation où une politique de remplacement LRU d'une cache serait nécessairement plus efficace qu'une politique de remplacement aléatoire. Pourquoi ?
9. Concernant les caches, est-il plus logique d'utiliser en pair les mécanismes (*write-through* et *write-allocate*) ou (*write-through* et *write-no-allocate*) ? Pourquoi ?
10. Quels sont les avantages d'une cache Harvard par rapport à une cache unifiée ?
11. Sur un ordinateur contenant un processeur Intel Pentium III et 16 Gio de RAM installée, le système d'exploitation indique qu'il n'y a que 4 Gio de mémoire utilisable. Pourquoi ?
12. Pourquoi le code suivant produit-il une erreur d'exécution ?

```
#include <stdlib.h>
int main(void) {
    asm("pushf\norl $ 0x40000, (%esp)\npopf\n");
    *((int*) (((char*) malloc(5)) + 1)) = 23;
    return 0;
}
```

Exercice 3 : mémoire cache

Nous possédons un système embarqué qui comporte une très petite mémoire principale de 32 kio et une mémoire cache de 2048 octets associative par ensemble de deux blocs, avec des lignes de 16 octets. La cache obéit aux politiques *write-back* et *write-allocation*. Son algorithme de remplacement est LRU (*least recently used* : moins récemment utilisé).

a. Vous devez écrire comment l'adresse est divisée dans ce système. Donnez le nombre de bits réservés pour le *tag*, l'ensemble et l'octet.

b. Remplissez ensuite le tableau suivant pour chaque accès mémoire. Les colonnes « tag » et « ensemble » doivent comporter les numéros de *tag* et d'ensemble pour l'accès en question. La colonne « hit » doit être cochée s'il y a un succès d'accès à la cache. La colonne « w-b » doit être cochée lorsque, pour l'accès de la même rangée, il y a un *write-back* vers la mémoire principale d'impliqué.

   N'oubliez pas que, étant donné qu'il s'agit d'une cache associative par ensemble de deux blocs, il y a un *dirty bit* pour chacun des deux blocs.

   La cache est initialement invalide (*dirty bits* tous à 0). Un accès mémoire `rd x` signifie « lire la mémoire à l'adresse `x` » et `wr x` signifie « écrire en mémoire à l'adresse `x` ». Les adresses sont données en base décimale.

accès	tag	ensemble	hit	w-b
wr 22496				
wr 16400				
rd 5552				
wr 16400				
rd 26592				
wr 10672				
rd 21936				
rd 22496				
wr 6112				
rd 2064				
rd 3040				
wr 13744				

c. Donnez l'état de la cache à la fin de ces accès.
   Le format devrait ressembler à (où l'astérisque `*` représente la présence du *dirty bit* pour un bloc) :

```
set     tag0    tag1
#1      27      7*
#5      62*
#9      2       3
```

d. Enfin, en supposant qu'un succès d'accès à la cache prenne 10 ns et qu'un défaut ou un accès à la mémoire principale prenne 160 ns, donnez le temps d'accès effectif de la cache pour la totalité de cette série d'accès.