INF1600 - TP5
Assembleur en ligne et mémoire

Remis le
14 décembre 2014
par

Samuel Rondeau (1723869)
&
Pacôme Bondet de la Bernardie (1617489)

Exercice 1 : assembleur en ligne

Le code se trouve dans tp5.c et inclut deux approches différentes.

Exercice 2 : mémoire – compréhension

1. Ils utilisent la mémoire virtuelle, c'est-à-dire une adresse relative dans le bloc de mémoire qui leurs sont réservés. Si A affiche l'adresse 0x1000 (virtuelle) et que A commence à 0x50 (physique), et que B affiche l'adresse 0x1000 (virtuelle) et que B commence à 0x1200 (physique), les deux adresses virtuelles sont identiques, mais les adresses physiques sont différentes.

2. D'abord, le programme s'exécute à son adresse virtuelle, disons 0x9, qui correspond à l'adresse physique 0x4000 (par exemple). Puisque le pointeur nul pointe vers l'adresse 0x0 et qu'on tente de le déréférencer, c'est-à-dire d'accéder à son contenu, mais que l'adresse 0x0 ne contient aucun donnée accessible, on n'accède à rien. De plus, on tente d'accéder à une adresse plus basse que le *segment base register*, associé au programme actuel, ce qui cause une erreur de segmentation, c'est-à-dire accéder à un segment qui n'est pas accessible par le segment actuel. C'est le système d'exploitation qui gère ces droits d'accès et donc les erreurs de segmentations.

3. Parce qu'ils sont bien plus coûteux. Avec autant de registres, personne n'aurait les moyens de s'acheter un tel ordinateur.

4. Le MMU est ce qui protège le système des accès en mémoire illégaux, i.e. erreurs de segmentation. Il s'assure de convertir les adresses physiques en adresses virtuelles et vice-versa, et est utilisé pour protéger la mémoire du segment A des accès non permis d'autres segments. Ainsi, sans MMU, les risques sont qu'un système soit vulnérable aux accès illégaux et aux *trojan horses*.

5. Le processus s'appelle du *Memory Mapping*, ceci est géré par le système d'exploitation, les pages des programmes ont le droit de lire les données dans les librairies partagés mais pas le droit d'y écrire. L'adresse de ces librairies sont chargés dans le processus courant, dans la memoire virtuelle.
   Ainsi, plusieurs processus peuvent partager les memes librairies.

6. Pour certaines boucles de petite taille, executer le code n fois est relativement optimal par rapport a faire n comparaisons avec des registres, rechercher l'instruction, etc.. .
   Par contre, si par malheur il faudrait executer des boucles imbriquées il faudrait copier le code n*m fois, ceci ne serait pas optimal parce que le programme très gros, et occuperait trop d'espace mémoire à l'exécution.

7. La MMU est gérée par le systeme d'exploitation lui même, c'est lui qui gère la traduction des adresses virtuelles vers les adresses physiques et vice-versa, donc le systéme d'exploitation est obligé d'avoir accès aux adresse physiques pour pouvoir gérer les traductions.

8. Dans le cas où on aurait besoin de plus de pages récemment ajoutées dans la cache, on sait que les tags seront là si on en aura besoins dans l'execution peu apès le placement de cette page.
   Par contre si les blocs sont remplacés aléatoirement dans une cache, si on a besoin de cette instruction/donnée on n'est pas certain que le tag y est toujours après un temps d'execution court.

9. Le premier cas, (*write-through* et *write-allocate*) est optimal. Dans le second cas, suite à un défaut d'accès, la mémoire principale et la mémoire cache ne sont pas à jour, ce qui peut poser des problèmes au niveau du programme. Dans le premier cas, les mémoires sont toujours à jour.

10. Dans la cache Harvard, le temps d'accès est plus rapide. Puisque les instructions et les données sont séparées, sachant à quoi on veut accéder, on a besoin de chercher dans moins de mémoire pour obtenir l'instruction ou la donnée désirée.

11. Dans une architechture d'un processeur a 32-bit, on peut seulement adresser jusqu'à 4GB de données, car 2^{32} bits = 4GB. Ajouter plus de RAM ne sert donc à rien.

12. Plusieurs raisons :
    1. Chaque ligne d'instruction devrait se retrouver dans une chaîne de caractères séparée;
    2. Lorsqu'on utilise un registre, on doit utiliser deux fois le caractère '%';
    3. On ne devrait pas utiliser "(%esp)".

Exercice 3 : mémoire cache

a. **ensemble** = 2^11 octets / 2^4 octets/ligne / 2^1 lignes/ensemble = 2^6 ensembles, donc 6 bits pour l'ensemble;
   octet = 2^4, donc 4 bits pour l'octet;
   tag = 15 - 6 - 4 = 5 bits pour le tag, car 2^15 = 32 ko.

   La structure est donc 5 | 6 | 4

b.

accès		**tag**	**ensemble**		**hit**	**w-b**
wr 0x57E0		0x15	0x3E	1	dirty bit	
wr 0x4010		0x10	0x01	1	dirty bit	
rd 0x15B0		0x05	0x1B	1		
wr 0x4010		0x10	0x01	1	succès, (dirty bit)	
rd 0x67E0		0x19	0x3E	2		
wr 0x29B0		0x0A	0x1B	2	dirty bit	
rd 0x55B0		0x15	0x1B	1		
rd 0x57E0		0x15	0x3E	1	succès, (dirty bit)	
wr 0x17E0		0x05	0x3E	2	dirty bit	
rd 0x0810		0x02	0x01	2		
rd 0x0BE0		0x02	0x3E	1		x
wr 0x35B0		0x0D	0x1B	2		x

c.

Set	1	2	3	4	5	6	7	8	9	10	11	12	fin
Succ				O				O			rb	rb	
Mem	R	R	R		R	R	R		R	R	WR	WR	
0x01		10x		x						01			10x 01
0x1B			05			0Ax	15					0D	15 0D
0x3E	15x				19			x	05x		02		02 05x

set	**tag0**	**tag1**
0x01	0x10*	0x01
0x1B	0x15	0x0D
0x3E	0x02	0x05*

d. h*tp+(1-h)*ts = (2/12)*10ns+(1-2/12)*160ns = 135 ns/accès