

Ressource R3.05 - TD 1

Mémoire

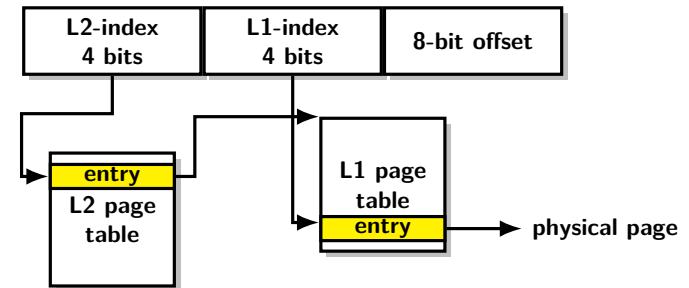
Table des pages

- Une table (simplifiée) des pages possède les entrées suivantes (U : User , P : Present, W : Writeable)

index	numéro physique de la page	permissions
0	0x00	PTE_U
1	0x01	PTE_P
2	0x02	PTE_P PTE_W
3	0x03	PTE_P PTE_W PTE_U
4	0xFF	PTE_W PTE_U
5	0xFE	PTE_U
6	0x80	PTE_W
7	0x92	PTE_P PTE_W PTE_U
8	0xAB	PTE_P PTE_W PTE_U
9	0x09	PTE_P PTE_U
10	0xFE	PTE_P PTE_U
11	0x00	PTE_W
12	0x11	PTE_U

Pour chaque action ci-dessous, donnez l'adresse physique correspondante, ou identifiez la faute qui sera produite (défaut de page, violation de privilège, violation de permission). La taille des pages est 4ko.

- le noyau déréférence un pointeur null.
 - le noyau écrit à l'adresse 0x8432.
 - un processus utilisateur écrit à l'adresse 0xB123.
 - le noyau lit à l'adresse 0x9876.
 - un processus utilisateur lit à l'adresse 0x7654.
 - un processus utilisateur écrit à l'adresse 0xABCD
 - un processus utilisateur écrit à l'adresse 0x2321.
- Une architecture utilise des adresses virtuelles sur 16 bits, et des pages de 256 octets. La traduction des adresses est assurée par la table des pages sur 2 niveaux de la figure suivante :



- Combien y-at'il d'entrées dans une table L1 et L2 ?
- Combien de table de pages L1 et L2 un processus peut avoir ?
- Les PTE occupent 16 bits (12 bits pour le numéro de page physique, trois flags P,W,U, et le dernier est inutilisé). Combien de mémoire physique cette architecture peut-elle supporter ?
- Complétez le code source de la fonction suivante :

```

#define PTE_P 8
typedef uint16_t pageentry;

unsigned virtual_to_physical(pageentry* pagetable, uint16_t vaddr)
{
    // pagetable : L2 page table
    // vaddr : virtual address
    pageentry pte = 0;
    uint16_t* l1pt = NULL;

    // grab the PTE for the pagetable (L2 page table)
    pageentry pte = ??? ;
    if ((pte & PTE_P) == 0) {
        return INVALID_PHYSADDR;
    }

    // Calculate L1 page table
    l1pt = ??? ;
    pte = ???;
    if ((pte & PTE_P) == 0) {
        return INVALID_PHYSADDR;
    }
    // return physical adress
    return ???;
}

```

Caches

1. On considère la mémoire conventionnelle dont les paramètres sont :

- taille adresse : 16 bits;
- unité d'adressage : 8 bits;
- mot : 16 bits.

et la mémoire cache telle que

- une ligne contient un mot;
- nombre de ligne dans le cache : 256;
- le cache est direct

Dans quelle ligne vont les données d'adresse 0xabcd ? Quelle le contenu de cette ligne ?

2. (a) Un cache fait 256 Ko, 4-associatif. Ses lignes font 32 octets. l'adresse est 32 bits, et l'unité d'adressage l'octet. Chaque entrée du cache, en plus de l'étiquette, stocke 2 bits de validité, 1 bit de modification et 1 bit de remplacement.
- i. Quelle est la taille de l'étiquette (tag) ?
 - ii. Quelle est la taille totale des entrées du cache en bit ?
- (b) Un cache direct de taille 8 Ko est organisé en ligne de 32 octets. Les adresses sont sur 32 bits, l'unité d'adressage est l'octet. Chaque entrée, en plus du tag, stocke 2 bits (validité et modification). Quelle est la taille totale en bit des entrées ?

3. Soit la liste de pages virtuelles référencées au cours du temps

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

La mémoire centrale est composée de 3 cases initialement vides. Représentez l'évolution de la mémoire centrale au fur et à mesure des accès pour les deux politiques de remplacement FIFO et LRU. Comparez avec la solution optimale.

Organisation mémoire, alignement

1. Soit le programme suivant (on suppose que les adresses et le type `long int`) sont sur 64 bits) :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  struct ptrs {
5      long int** x;
6      long int* y;
7  };
8  struct ptrs global;
9
10 void init(struct ptrs* p) {
11     long int* a = malloc(sizeof(long int));
12     long int* b = malloc(sizeof(long int));
13     long int* c = malloc(sizeof(long int));
14     long int* d = malloc(sizeof(long int));
15     long int* e = malloc(sizeof(long int) * 2);
16     long int** f = malloc(4 * sizeof(long int*));
17     long int** g = malloc(sizeof(long int*));
18
19     *a = 0;
20     *b = 0;
21     *c = (long int) a;
22     *d = *b;
23     e[0] = 29;
24     e[1] = (long int) &d[100000];
25
26     f[0] = b;
27     f[1] = c;
28     f[2] = 0;
29     f[3] = 0;
30
31     *g = c;
32
33     global.x = f;
34     global.y = e;
35
36     p->x = g;
37     p->y = &e[1];
38 }
39
40 int main(int argc, char** argv) {
41     struct ptrs p;
42     init(&p);
```

En supposant que `a` contient la valeur `0x5573de4342a0`, et que `malloc` a retourné des adresses croissantes, associez correctement les expressions et leurs valeurs (contexte du `main`) :

	Valeur		Expression
1	0x7ffc159ab650	A	<code>global.y</code>
2	0x5573de434320	B	<code>&global.y</code>
3	0x5573de4342a0	C	<code>(long int*)p.y</code>
4	0x5573de4f7800	D	<code>&p</code>
5	0x5573dcdac058	E	<code>(long int*)p.x[0]</code>

Rappel sur l'ordre de priorité des opérateurs du C :

priorité	opérateur	associativité
16	<code>[] .</code>	gauche
15	<code>* &</code>	droite
14	conversion	droite

Rappel : les adresses sont rangées dans l'ordre croissant suivant :

`code < data < heap < mapping < stack < kernel`

2. Donnez les types par ordre croissant de taille :

```

1 char
2 struct minipoint { uint8_t x; uint8_t y; uint8_t z; }
3 int
4 char **
5 double[0]
```

3. Pour chaque déclaration, donnez la taille, l'alignement correspondant :

struct s1	struct s2	struct s3	struct s4
{ float x; char n[1]; };	{ short s; char n[3]; };	{ char Data1; short Data2; int Data3; char Data4; };	{ char Data1; char Data4; short Data2; int Data3; };

4. En utilisant une seule fois les membres `char a`; `unsigned char b`; `short c`; `int d`; et en plaçant `a` en premier, peut on construire une structure de taille 16, 12, 8 ?