Corrigé - Travaux Dirigés
Séance nr. 3 – Adressage virtuel et Cache

Dans cet exercice on se propose de gérer une mémoire physique en utilisant un mécanisme d'adressage virtuel. Par ailleurs, nous simulons de façon logicielle un cache mémoire.

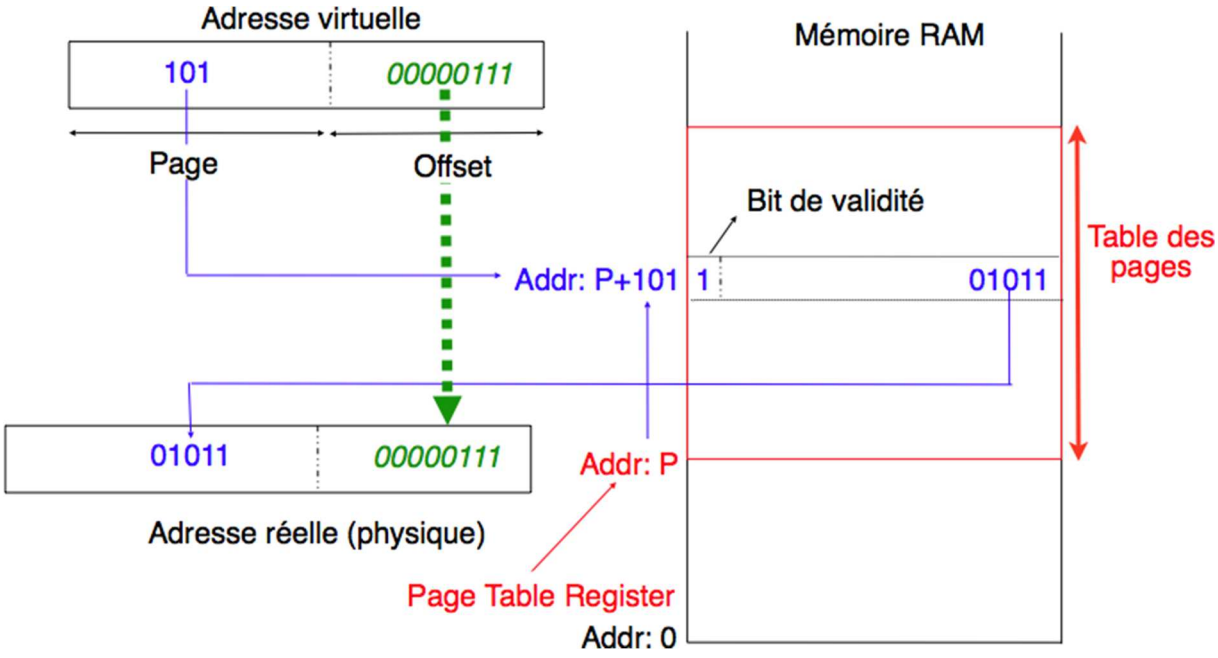
Définitions et cadre du problème

Une table d'adressage virtuel permet d'associer une adresse physique à une adresse virtuelle:

TAV : @virtuelle → @physique

Généralement la mémoire physique est découpée en pages. L'adresse virtuelle est alors composée d'un **numéro de page** et d'un **offset** dans la page. L'adresse physique est obtenue en conservant l'offset et en liant, via une table, le numéro de page à une position de page en mémoire physique. La table est également rangée en mémoire à partir d'une adresse. Cette adresse est donnée par un registre spécifique appelé « table page register ».

Le schéma ci-dessous présente ce mode de gestion *(source et copyright : Univ. du Louvain)*.



1. ***Quelles capacités d'adressage et quelle occupation mémoire ?***

En supposant que notre système utilise des adresses virtuelles codées à l'aide d'entiers non signés de 32 bits (unsigned int) et que les pages font 256 octets (=2^8 octets), combien de pages peut-on encoder au maximum ?

* nous utilisons ici volontairement l' « ancienne » notation ko, Mo, Go, etc. (au lieu de kibi-octet, kio, Mio, Gio, etc.) même si cela s'oppose aux recommandations du système international d'unités (SI), l'usage de l'ancienne notation restant assez largement en vigueur chez les informaticiens, surtout des premières générations...

Corrigé

Pour une page il y a 256=2^8 adresses d'octets. Il reste donc 24 bits pour coder le numéro de page, soit 2^24 pages

La table des pages contient autant de lignes / entrées qu'elle contient de pages. En supposant que chaque entrée de la table nécessite 32 bits de stockage, soit plus que nécessaire pour coder le numéro de page (la table encode également des informations supplémentaires comme un bit de validité, des bits de présence ou non en mémoire ou sur disque, etc.), quelle est la taille occupée par la table elle-même ?

Corrigé

Il y a 2^24 pages et autant de lignes. Chaque ligne occupe 4 octets, donc en tout la table occupe : 2^26 octets, soit 64Mo.

Si l'adresse est codée sur 32 bits, quelle est l'espace d'adressage (la taille maximale de la mémoire virtuelle), sachant qu'une adresse correspond à un mot long de type ***unsigned int*** ?

Corrigé

Il y a 2^32 adresses. La mémoire maximale est donc : 2^34 octets, soit, 8Go.

REM : de nos jours les adresses sont plutôt codées sur 64 bits, bien qu'aucun ordinateur ne possède 2^64 octets de mémoire vive. Un ordinateur disposant de 128 Go de mémoire physique pourrait se contenter d'utiliser des adresses codées sur 37 bits. La mémoire virtuelle offre donc la possibilité d'accéder à plus de mémoire que celle qui est réellement disponible.

2. *Calcul des adresses physiques*

Soient les constantes :

```
#define ADDRESS_WIDTH 32
#define PAGE_WIDTH 24
#define OFFSET_WIDTH 8
#define PAGE_SIZE (1<< OFFSET_WIDTH)
```

La table des pages est stockée en mémoire RAM comme un tableau en C, supposé de type ***unsigned int.*** Nous appelons le tableau correspondant à la mémoire vive :

```
unsigned int  RAM[MAX] ;
```

MAX étant la capacité de la mémoire vive en nombre de mots longs. Sachant par ailleurs que l'on définit une constante externe de type unsigned int appelée ***ptr*** (page table register) indiquant l'adresse de la table dans RAM, proposer un code C permettant de calculer l'adresse physique à partir d'une adresse virtuelle :

```
unsigned int  phys(unsigned int virt);
```

Ecrire une fonction ***load*** permettant de lire une donnée de la RAM à partir d'une adresse virtuelle (les données lues sont de type unsigned int). De même pour l'écriture dans la RAM :

```
unsigned int   load(unsigned int virt);
void           store(unsigned int virt, unsigned int data);
```

Corrigé

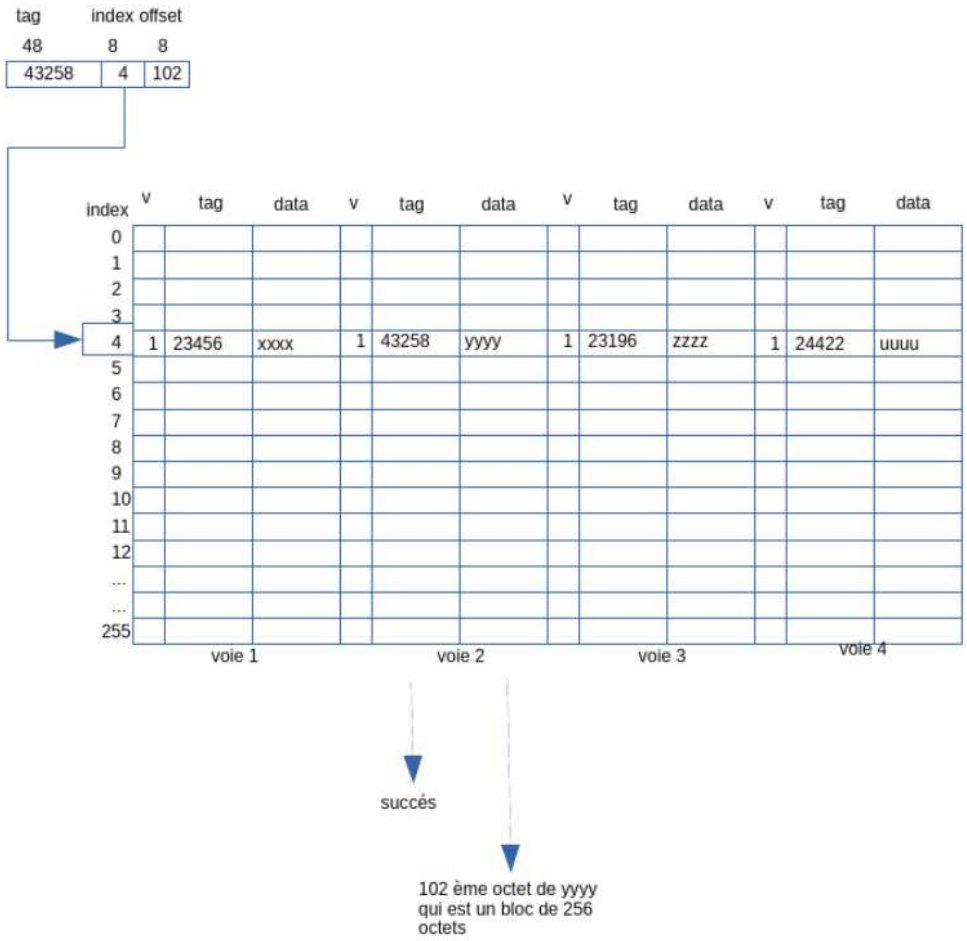
```
extern unsigned int ptr;

unsigned int phys(unsigned int virt)
{
        unsigned int offset = virt & 0xFF;
        unsigned int page = virt >> OFFSET_WIDTH;
        unsigned int phys = ((RAM[ptr + page]&0xFFFFFF) << OFFSET_WIDTH) | offset;
        return phys;
}

unsigned int load(unsigned int virt)
{
        return RAM[phys(virt)];
}

Void store(unsigned int virt, unsigned int data)
{
        RAM[phys(virt)] = data;
}
```

Exercice 2: *Utilisation d'un cache à N-voies*

La fonction ***load*** doit d'abord consulter la table des pages pour traduire l'adresse virtuelle en une adresse physique (appel à la fonction ***phys***). En pratique, l'utilisation d'une table augmente le temps d'accès à une donnée en mémoire : elle nécessite deux accès à RAM (un premier pour accéder à la table et un second pour récupérer ou enregistrer la donnée). L'accès à la RAM est donc un goulet d'étranglement très important.

Pour améliorer les performances, les processeurs mettent en place un mécanisme de mémoire cache. Le principe est de stocker dans une mémoire rapide, généralement située au sein même du processeur, des copies de données qui sont en mémoire centrale RAM. Cela évite les transitions couteuses via le bus.

On se propose de **simuler un mécanisme de cache associatif à 4 voies** comme présenté par la figure ci-dessous. Avant tout accès à RAM, on vérifie si les données ne sont pas présentes dans le cache. Pour cela une adresse @ est divisée en trois parties : **tag, index** et **offset**, offset étant comme précédemment une position dans la page. Le numéro de page est à présent composé d'un tag et d'un index, index étant utilisé pour accéder au cache. Nous supposons tag codé sur 16 bits et index codé sur 8 bits (en général, tag est codé sur 48 bits, mais dans cet exercice un codage sur 16 bits permettra d'utiliser des unsigned int pour simplifier). La mémoire cache dispose d'autant d'entrées qu'il y a d'indexes. Pour un index codé sur 8 bits, il y a donc 256 entrées (lignes) dans le cache.



Chaque ligne du cache est elle-même un tableau de N structures appelées « entry », entry étant composée d'un bit de validité, d'un tag et d'une donnée. N est le nombre de voies. La donnée est simplement une copie de la page correspondante de la mémoire RAM. La taille du champ « donnée » est celle d'une page, donc 256 valeurs dans notre cas. Nous considérons N=4 voies :

```
#define TAG_WIDTH 16
#define ENTRY_WIDTH 8
#define N_WAYS 4

typedef unsigned int address;
typedef struct s_entry {
       unsigned int    v : 1; // un seul bit
       unsigned int    tag : TAG_WIDTH;
       unsigned int   data[PAGE_SIZE];
       } entry;

entry  CACHE[(1<<ENTRY_WIDTH)][N_WAYS] ;
```

En négligeant le bit de validité (v), quelle est la taille de CACHE en octets ?

Si on cherche une valeur de la mémoire RAM à une adresse @, donc la valeur de RAM[@], on cherche d'abord dans CACHE. On calcule l'indice d'entrée donné par les bits « index » de @. On vérifie que le « tag » est dans l'un des 4 emplacements : si oui, c'est un succès et on récupère la valeur dans le bloc « data ». Sinon, c'est un échec il faudra copier la page depuis la RAM dans le cache et donc choisir celle des 4 entrées qui sera victime et remplacée.

2.1 Ecrire les opérations qui permettent de calculer l'offset, le tag et l'index à partir d'une adresse :

```
address a_offset(address pa);
address a_tag(address ba);
address a_index(address ba);
```

Corrigé

```
#define MASK_8 0xFF

address a_offset(address pa)
{
    return pa & MASK_8;
}

address a_tag(address pa)
{
    return pa >> OFFSET_WIDTH >> A_ENTRY_WIDTH;
}

address a_index(address pa)
{
    return (pa >> OFFSET_WIDTH) & MASK_8 ;
}
```

2.2 Initialisation du cache

Ecrire une opération qui initialise le cache :

```
void cache_init();
```

Corrigé

```
void cache_init()
{
    for(int i=0; i<(1<< ENTRY_WIDTH);i++)
        for(int j=0; j<NB_WAYS;j++)
            cache[i][j].valid = false;
}
```

2.3 Vérification du succès

Ecrire une opération qui, à partir d'une adresse, vérifie si elle est dans le cache. Cette fonction renvoie le numéro de voie (0-3) dans laquelle se trouve la donnée, ou -1 en cas d'échec :

```
int cache_isin(address pa);
```

Corrigé

```
int cache_isin(address pa)
{
    address ind = a_index(pa), ta = a_tag(pa);
    address w=0;   // voie
    bool trouve = false;
    for(;w<NB_WAYS && !trouve;w++) trouve = cache[ind][w].tag==ta;
```

```
    return trouve ? w : -1;
}
```

2.4 Lecture et écriture dans le cache

Ecrire une opération qui à partir d'une adresse et d'un numéro de voie (0-3) renvoie (resp. stocke) la donnée correspondante. C'est l'équivalent du *load* et *store* précédents :

```
unsigned int cache_load(address pa, int i_way);
void cache_store(address pa, int i_way, unsigned int data);
```

Corrigé

```
unsigned int cache_load(address pa, int i_way)
{
        address ind = a_index(pa);
        return cache[ind][i_way].data[a_offset(pa)];
}

void cache_store(address pa, int i_way, unsigned int data)
{
        address ind = a_index(pa);
        cache[ind][i_way].data[a_offset(pa)]=data;
}
```

2.5 Complément

La lecture précédente n'est possible qu'en cas de succès. Si la page n'est pas dans le cache, il faut une stratégie qui, si aucune voie n'est libre (ou non valide), libère l'une des 4 voies pour y copier les données depuis la RAM.

De quelle manière peut-on modifier la structure « entry » de façon à mettre en œuvre une stratégie LFU (least frequently used) ?

Que faut-il faire avant de copier la page depuis la RAM vers le cache une fois la voie choisie ? Comment peut on éviter une mise à jour systématique de la RAM ?

Corrigé

Pour la stratégie LFU on ajoute un champ « compteur » dans la structure entry qui compte les accès (incrémente de 1 à chaque accès). Il est ainsi possible de classer chaque voie en fonction du nombre d'accès et de choisir le min.

Lorsque l'on écrit dans le cache avec store, et que l'on sacrifie cette page, il ne faut pas oublier de copier la page sacrifiée du cache vers la RAM, avant de remplacer ses données par une autre page et un autre tag. Pour éviter de faire cette mise à jour systématiquement, on utilise un bit « dirty » qui indique si les données de la page ont été modifiées dans le cache. Seules les pages modifiées sont mises à jour dans la RAM avant d'être sacrifiées.

On obtient la structure entry modifiée suivante :

```
typedef struct s_entry {
       unsigned int    v : 1; // un seul bit
       unsigned int    dirty : 1; // un seul bit
       unsigned int    counter;
```

```
unsigned int    tag : TAG_WIDTH;
unsigned int   data[PAGE_SIZE];
} entry;
```