

Gestion de la mémoire

+ 1. Qu'est-ce que la mémoire ?

- Définitions, exemples

2. Allocation contiguë en mémoire

- Partitions de taille fixe, de taille variable

3. Pagination et mémoire virtuelle

Annexe. La pile : utilisation des arguments passés à *main*.

Gestion de la mémoire

Définitions

- **Mémoire** : juxtaposition de **cellules** ou **mots-mémoire** ou **cases-mémoire** à m bits (Binary digIT). Chaque cellule peut coder 2^m informations différentes.
- **Octet** (byte) : cellule à huit bits. (1 Kilo-octet = 2^{10} octets, 1 Mega-octet = 2^{20} octets).
- **Mot** : cellule à 16, 32, 64 bits.
- Temps d'accès à la mémoire (principale) : 20 à 100 ns.

Généralités :

Sur une machine classique, rien ne distingue, en mémoire, une instruction d'une donnée (architecture Von Neumann).

Exemple de protections logicielles :

- données accessibles en lecture/écriture
- programme accessible en lecture seule

Evolution :

- Sur les machines dotées des processeurs les plus récents, ce type d'architecture est remis en cause. En effet, ces processeurs sont équipés de mémoire appelées caches et ces caches sont souvent distincts pour les données et les instructions. On parle dans ce cas de «*Harvard type cache*».
- Sur certains calculateurs spécialisés (systèmes embarqués, processeurs de traitement du signal, ...) les programmes et les données sont rangés dans des mémoires différentes : programme en ROM (read only memory), données en RAM.

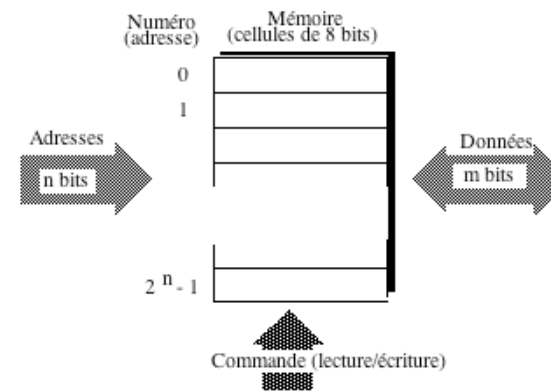
Pourquoi la mémoire ?

Le processeur va chercher les informations dont il a besoin, c'est-à-dire les instructions et les données, dans la mémoire. Il ne va jamais les chercher sur le disque ! Ce dernier sert de support d'archivage aux informations, en effet la mémoire s'efface lorsque on coupe l'alimentation électrique de la machine.

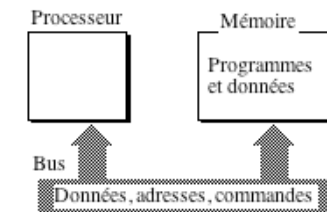
Gestion de la mémoire

La mémoire, ressource du S.E

- La mémoire est assemblage de cellules repérées par leur numéro, ou **adresse**.
- *Gestionnaire de mémoire* : gère l'allocation de l'espace mémoire au système et aux processus utilisateurs.



- n = largeur du bus d'adresses
- m = largeur du bus de données



Gestion de la mémoire

Adresse d'une information :

Chaque information est repérée par son adresse, c'est à dire le numéro de son premier octet. Cet adressage en termes d'octets permet d'adresser, de repérer, en mémoire tous types d'information, quelque soit leur taille. On peut ainsi adresser un simple octet (un char en langage C), un mot sur deux octets (un short en langage C), etc.

Types de base en C

- En langage C, la coutume sur Sun, par exemple, est la suivante :

[unsigned] char 8 bits
[unsigned]short 16 bits
[unsigned]int un mot machine (32 ou 16 bits !)
[unsigned]long 32 bits

Voici la règle :

$1 \leq \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$

et en C ansi :

$\text{sizeof}(\text{short}) \geq 16 \text{ bits}$

Gestion de la mémoire

Adressage de la mémoire

- les cellules mémoire sont repérées par leur adresse en termes d'octets. Ceci permet d'adresser différents types de données, en langage C :

char, int , long

- Exemple d'implantation de variables sur une machine à mots de 4 octets

(les données sont implantées à partir de l'adresse 2) :

```
#include <stdlib.h>
```

```
int j;  
short k;  
char c;
```

```
/*  
j sur 4 octets -> adresse = 2  
k sur 2 octets -> adresse = 6  
c sur 1 octet -> adresse = 8  
*/
```

```
int main(void) {  
    exit(0);  
}
```

- La règle :

$1 \leq \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$

- ... et en C ANSI : **$\text{sizeof}(\text{short}) \geq 16 \text{ bits}$**

Gestion de la mémoire

Utilisation d'une union pour "voir" une zone mémoire de plusieurs façons :

```
int main ( void ){
/*-----
 Visualiser sous forme de quatre octets une adresse
 Internet donnée sur un long :
      0x89c2a032 --> 137.194.160.50 (scapin)
-----*/
...

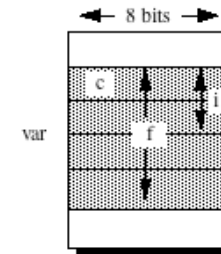
union Adr_IP{
  struct { unsigned char o1, o2, o3, o4;} octets;
  unsigned long entier;
};
...
union Adr_IP Site;
...
/*
Site.entier est initialisé grâce à une fonction du
type gethostbyname ( )
*/
...
printf ("adr. IP -> %d.%d.%d.%d\n",
      Site.octets.o1,
      Site.octets.o2,
      Site.octets.o3,
      Site.octets.o4);
...
}
```

Gestion de la mémoire

Mémoire et langage C : types dérivés

- Une union permet de "voir" la même zone mémoire de façons différentes :

```
union{
  char c;
  short i;
  float f;
}var;
```



- l'union a la taille du plus grand élément qu'elle contient, ici :
sizeof var = sizeof float;

Gestion de la mémoire

- Pointeurs et tableaux en C : `Tab [i]` équivalent à `*(Tab+ i)`

- Arithmétique des pointeurs :
Attention aux différences de pointeurs, elles n'ont de sens que si les pointeurs pointent sur des objets de même type :

```
$ cat essai.c
#include <stdio.h>
#include <stdlib.h>
short int i = 0;
char c = 'a';
short int j = 1;
int main (void){
    char *pc = &c;
    short int *p = &i;
    short int *q = &j;
    int r;
    r = p - q ;
    printf("pc = %x\n", pc);
    printf("p = %x\n", p);
    printf("q = %x\n", q);
    printf("r = %d\n", r);
    exit (0);
}
```

```
$ gcc -Wall essai.c -o essai
$ essai
pc = 804963c
p = 8049644
q = 804963e
r = 3
```

Différence de pointeurs : $r = (p-q)/sizeof\ int$!!!!

Gestion de la mémoire

C et adressage de la mémoire : exemple

• Le tableau `Tab` est implanté en 0, que se passe-t-il en mémoire lors de l'exécution du programme ci-dessous :

```
int main (void){
    short    Tab[4];
    short * Ptr1;
    char * Ptr2;

    Ptr1 = (short *)Tab;
    Ptr2 = (char *)Tab;

    Tab [1] = 0;
    Ptr1 [1] = 0;

    *Ptr2 = 1;
    *Ptr1 = 1;

    *(Tab + 1) = 0;

    /* valeur de Ptr1 après l'instruction suivante ?*/
    *(Ptr1 + 1) = 0;

    /* valeur de Ptr1 après l'instruction suivante ?*/
    *(Ptr1 ++)= 0;

    /* Instruction suivante incorrecte, pourquoi ? */
    *(Tab++) = 0;

}
```

C et adressage de la mémoire : Exemple, commentaires(1)

- Rappels :
 - le nom d'un tableau est un pointeur constant sur l'adresse de début du tableau. Il ne peut donc pas être modifié.
 - Ici la case dont le nom est `Tab` contient donc zéro.
 - L'arithmétique des pointeurs se fait en fonction de la taille de l'objet pointé. Par exemple, si on a : `short * Ptr;`
 - Alors l'instruction :
`Ptr = Ptr + 2;`
signifie :
`Ptr = Ptr + 2*sizeof(short)`

Gestion de la mémoire

Dans l'exemple ci-après, on réécrit la fonction `strcpy` de la bibliothèque C. L'objectif est de montrer les particularités de la gestion des chaînes de caractères en C..., qui sont dues au fait qu'il n'y a **pas de type chaîne de caractères**. Celles-ci sont rangées sous forme d'un **tableau d'octets** terminé par un caractère `NULL` (octet à zéro).

La première version ressemble à ce qu'on écrirait dans n'importe quel autre langage que le C

Ligne 9 de cette version :
si l'utilisateur de la fonction n'a pas mis de caractère `NULL` en fin de tableau, `Copie` va recopier toutes les cases mémoires à partir de l'adresse `source` dans la zone dont l'adresse de début est `dest`.

Deuxième version : on fait l'affectation dans le test.

Troisième version : on manipule des pointeurs plutôt que des index dans un tableau.

Quatrième version : on met en œuvre l'opérateur unaire `++` qui incrémente ici après l'affectation

Cinquième version : on utilise le fait qu'il n'y a pas de type booléen, et que les opérations logiques renvoient un entier :

- (1 signifie vrai, 0 signifie faux)
- de la même façon, tout ce qui n'est pas égal à zéro est vrai.

Le `while` teste donc la valeur de l'affectation, qui sera toujours vraie tant qu'on n'arrive pas sur le `NULL` de fin de chaîne.

Ce style de programmation, qui donne du code difficile à lire et à maintenir **ne produit pas des exécutables beaucoup plus performants. Laissons au compilateur le travail d'optimisation**, dans la plupart des cas, il le fait mieux que les programmeurs.

Gestion de la mémoire

Exemple, commentaires(2)

- Les deux instructions suivantes sont équivalentes :

`Tab [1] = 0 ; et Ptr1 [1] = 0 ;`
Voici leur effet en mémoire : 0

0	
1	
2	0
3	0

Effet de `*Ptr2=1` : 0

0	1
1	
2	0
3	0

Effet de `*Ptr1=1` 0

0	0
1	1
2	0
3	0

Gestion de la mémoire

Exemple : copie de tableaux de caractères en C

- première version :

```
1. void Copie ( char *dest, char *source) {
2.
3.     char *p, *q;
4.     int i;
5.     i=0;
6.     p=dest;
7.     q=source;
8.     for (i=0; q[i] != '\0'; i=i+1)
9.         p[i]=q[i];
10.    p[i]=q[i]='\0';
11. }
```

- remplacement successifs des lignes 8 à 10 :

```
8. while((p[i]=q[i]) != '\0') i=i+1;
9.
10.
```

```
8. while( (*p=*q) != '\0'){
9.     p=p+1;
10.    q=q+1;}
```

```
8. while( (*p++=*q++) != '\0');
9.
10.
```

```
8.     while(*p++=*q++);
9.et 10
```

Le code généré sur SPARC par le compilateur gcc donne 5 instructions assembleur pour la boucle dernière version et 6 pour la boucle première version,

Gestion de la mémoire

Exemple, commentaires(3)

- Après :*(Ptr1 + 1) = 0, Ptr1 vaut 2.

- Si on fait maintenant :*(Ptr1 ++)= 1, Ptr1 passe à 4.

```
/* Instruction suivante incorrecte, pourquoi ? */
*(Tab++) = 0;
```

- On ne peut pas modifier un pointeur constant.

La partie du système d'exploitation qui s'occupe de gérer la mémoire s'attache à atteindre les objectifs suivants :

- *protection* : dans un système multi utilisateurs, plusieurs processus, appartenant à des utilisateurs différents, vont se trouver simultanément en mémoire. Il faut éviter que l'un aille modifier, ou détruire, les informations appartenant à un autre.
- *réallocation* : chaque processus voit son propre espace d'adressage, numéroté à partir de l'adresse 0. Cet espace est physiquement implanté à partir d'une adresse quelconque.
- *partage* : permettre d'écrire dans la même zone de données, d'exécuter le même code.
- *organisation logique et physique* : comme on le verra, à un instant donné, seule une partie du programme et des données peut se trouver en mémoire si la taille de cette dernière est trop faible pour abriter l'ensemble de l'application.

Gestion de la mémoire : objectifs

- *protection* : par défaut, un processus ne doit pas pouvoir accéder à l'espace d'adressage d'un autre,
- *partage* : s'ils le demandent, plusieurs processus peuvent partager une zone mémoire commune,
- *réallocation* : les adresses vues par le processus (adresses relatives ou virtuelles) sont différentes des adresses d'implantation (adresses absolues),
- *organisation logique* : notion de partitions, segmentation (cf. Intel), pagination, mémoire virtuelle,
- *organisation physique* : une hiérarchie de mémoire mettant en œuvre les caches matériels, la mémoire elle-même, le cache disque, le(s) disque(s) (qui est un support permanent, contrairement aux précédents),

Gestion de la mémoire

Un accès à la mémoire ne veut pas forcément dire un accès à la mémoire RAM, comme on pourrait le croire.

Les processeurs étant de plus en plus rapides, les mémoires leur semblent de plus en plus lentes, pour la simple raison que les performances de ces dernières progressent beaucoup plus lentement que les leurs.

Pour pallier ce défaut, les constructeurs implantent de la mémoire sur les "*chips*" c'est à dire sur les processeurs eux-mêmes. Ces mémoires sont appelées "caches". Elles contiennent un sous-ensemble de la mémoire RAM.

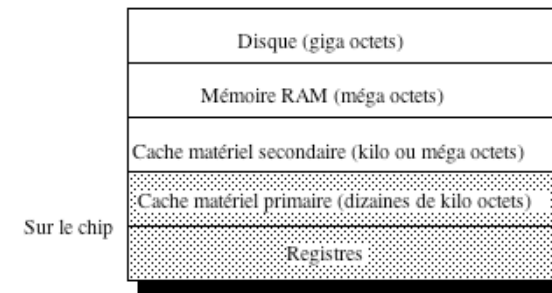
Ces caches s'appellent "on chip caches", *primary* caches ou caches primaires.

Il existe également des caches secondaires, qui ne sont pas sur le processeur lui-même, mais auxquels on fait accès par un bus dédié, plus rapide que celui permettant les accès à la mémoire classique.

Gestion de la mémoire

Hierarchie des mémoires

- Du plus rapide et plus petit (niveaux bas) au plus lent et plus volumineux (niveaux hauts) :



- Quand la capacité décroît, le coût du bit augmente.
- On fait accès aux disques locaux par le bus, l'accès aux disques distants se fait par le réseau.
- Remarque : à un instant donné, une information peut être présente à **plusieurs** niveaux de la hiérarchie.

Gestion de la mémoire

1. Qu'est-ce que la mémoire ?

- Définitions, exemples

+ **2. Allocation contiguë en mémoire**

- **Partitions de taille fixe, de taille variable**

3. Pagination et mémoire virtuelle

Gestion de la mémoire

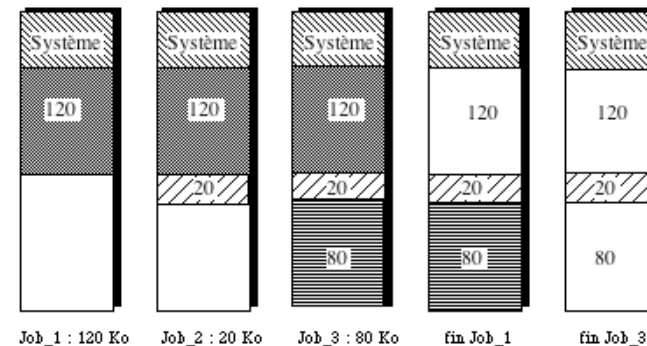
Problèmes liés à la gestion de la mémoire en partitions :

- Fragmentation interne :
-Un processus n'utilise pas la totalité de la partition.
- Fragmentation externe :
-Certaines partitions ne sont pas utilisées, car trop petites pour tous les travaux éligibles.
- Comment choisir a priori la taille des partitions ?

Gestion de la mémoire

Allocation contiguë : les partitions

- L'espace mémoire est alloué dynamiquement, de façon **contiguë**, lors de l'implantation des processus en mémoire : on ne sait pas découper l'espace d'adressage d'un programme en plusieurs parties disjointes :



Job_4 : 125 Ko ? Job_5 : 70 Ko ?

- Sur l'exemple, Job_5 peut être implanté à deux adresses différentes (laquelle choisir ?), mais on ne peut pas charger Job_4 qui demande un espace mémoire de 125 Ko.

- Si plusieurs partitions sont susceptibles de recevoir un processus, laquelle choisir :
 - *First fit* : on range le processus dans la première partition libre et suffisamment grande trouvée,
 - *Best Fit* : on va chercher la partition dont la taille approche au mieux celle du processus à charger en mémoire.
 - *Worst Fit* : on met le processus dans la plus grande partition libre.

Politiques de placement des partitions

- Si plusieurs zones libres sont susceptibles d'accueillir le processus à charger, plusieurs politiques de **placement** sont possibles :
 - *First Fit* : on place le processus dans la première partition dont la taille est suffisante
 - *Best Fit* : on place le processus dans la partition dont la taille est la plus proche de la sienne.
 - *Worst Fit* : on place le processus dans la partition dont la taille est la plus grande.
- *Best Fit* va conduire à la création de nombreux blocs minuscules non-réutilisables, on utilise généralement *Worst Fit* en classant les blocs par ordre de tailles décroissantes, un parcours *First Fit* sur la table ainsi classée donnera un placement *Worst Fit*.
- Dans le cas où il n'existe aucun bloc libre assez grand, on peut faire du *garbage collecting* ou ramasse-miettes (cf. plus loin).

Gestion de la mémoire

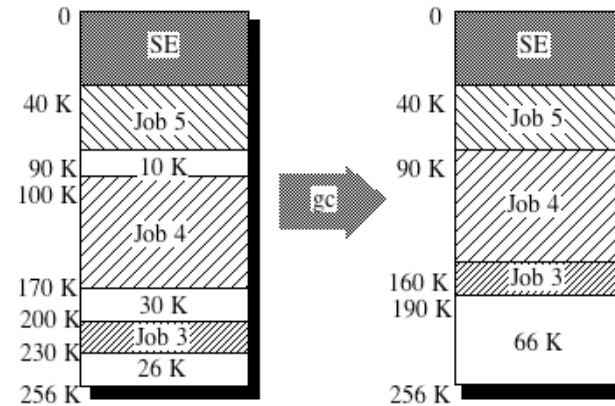
Le garbage collecting est appelé ramasse miettes en français.

Il est utilisé par des logiciels (emacs, écrit en Lisp), par des langages (Java, Lisp), pour gérer leur propre espace mémoire.

La défragmentation d'un disque s'apparente à du garbage collecting..

Gestion de la mémoire

Gestion de la fragmentation : le ramasse-miettes (garbage collecting)



- Coût de l'opération ?
- On peut également faire du compactage (regroupement des seuls trous voisins)

Limites de l'allocation contigüe.

Knuth a démontré que, quelle que soit la stratégie adoptée, on n'arrive jamais à recouvrir parfaitement l'espace libre par des blocs occupés.

Sa démonstration montre que la proportion de blocs libres (trous) sera d'un tiers par rapport au nombre de blocs occupés.

Il répartit les blocs occupés en trois catégories :

- ceux qui ont deux voisins libres,
- ceux qui ont deux voisins occupés,
- ceux qui ont un voisin de chaque nature.

Règle de Knuth (1)

- Soit :
 - N : nombre de blocs occupés,
 - M : nombre de blocs libres,
- Les différents types de blocs occupés :
 - Type a : bloc adjacent à deux blocs libres,
 - Type b : bloc adjacent à un bloc occupé et un libre,
 - Type c : bloc adjacent à deux blocs occupés,
- Si :
 - Na : nombre de blocs de type a
 - Nb : nombre de blocs de type b,
 - Nc : nombre de blocs de type c
- Alors : $N = Na + Nb + Nc$
- et : $M = Na + Nb/2 + \epsilon/2$ ($\epsilon > 0$ ou < 0 selon les bords)

Règle de Knuth (2)

- On note :
 - p : proba (on ne trouve pas de trou de bonne taille),
 - 1-p : proba (on trouve un trou de bonne taille).
- On a :
 - proba (M augmente de 1) = proba (on libère un bloc "c") = N_c/N
 - proba (M baisse de 1) = proba(on trouve un trou de bonne taille) + proba (on libère un bloc "a") = $1-p+N_a/N$
- A l'équilibre : proba (M augmente de 1) = proba (M baisse de 1) donc :
 - $N_c/N = N_a/N+1-p$
 - puisque $N-2M \approx N_c-N_a$ on a $M = N/2 * p$
 - si $p \approx 1$ alors $M \approx N/2$ ou encore $N = 2 * M$
- Conclusion : à l'équilibre, un bloc sur trois est libre, c'est-à-dire inoccupé !

Gestion de la mémoire

1. Qu'est-ce que la mémoire ?

- Définitions, exemples

2. Allocation contigüe en mémoire

- Partitions de taille fixe, de taille variable

+ 3. **Pagination et mémoire virtuelle**

Gestion de la mémoire

Le problème des partitions est la fragmentation car les programmes ont besoin d'espace contigu.

La pagination permet d'utiliser des espaces non contigus.

Comme on le voit sur le schéma, des pages qui se succèdent dans l'espace vu par l'utilisateur peuvent être implantées sur des pages mémoires non contigues. Les pages sont chargées en mémoire là où il y a de la place. La table de pages mémorise le placement de chaque page logique en mémoire physique.

La table de pages (TP) est un tableau tel que $TP[i]$ donne le numéro de page physique (l'adresse en mémoire) où a été chargée la page i .

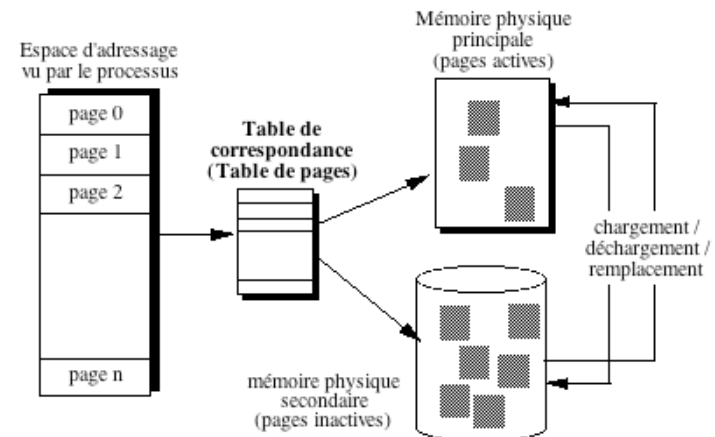
Exemple :

Si les pages 0,1 et 2 d'un processus sont chargées sur les pages mémoire 24, 78 et 100 on aura : $TP [0] = 24$, $TP [1] = 78$ et $TP [3] = 100$.

Gestion de la mémoire

Pagination

- L'espace d'adressage est divisé en *pages*, la mémoire « centrale » est divisée en *blocs*. (taille de la page = taille du bloc).
- *Table des pages* : elle fait correspondre une page logique à un bloc physique.



- Sur l'exemple ci-dessus, seules 3 pages parmi 6 sont chargées en mémoire.

Gestion de la mémoire

Quand l'espace d'adressage de l'application est supérieur à la taille de la mémoire physique, on parle d'adressage virtuel.

On donne l'illusion à l'utilisateur qu'il dispose d'un espace d'adressage illimité (en fait limité à la taille du disque sur lequel est rangée l'application).

Gestion de la mémoire

Mémoire virtuelle

- Soit M la taille de la mémoire disponible sur la machine. Soit T la taille de l'application :
 - Si $T < M$, on parle simplement de pagination,
 - Si $T > M$, on parle de mémoire virtuelle, deux cas sont alors possibles :
 - M est la de la mémoire libre à cet instant,
 - M est la taille de totale de la mémoire (application plus grande que la mémoire physique !)
- La mémoire virtuelle donne l'illusion à l'utilisateur (au processus) qu'il dispose d'un espace d'adressage illimité (en fait, limité par la taille du disque).

Gestion de la mémoire

Passage de l'adresse virtuelle à l'adresse physique :

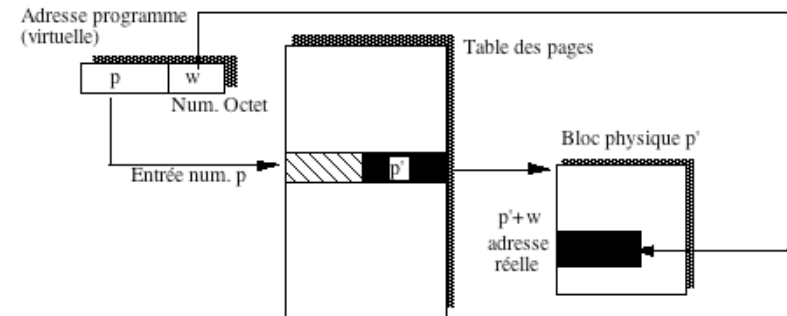
- l'adresse d'une information est divisée en deux champs : numéro de page virtuelle et déplacement dans cette page (ici appelés p et w)
- le contenu de l'entrée p de la table de pages (appelée TP) donne le numéro de page physique p' où est chargée p . Dans cette entrée, c'est à dire dans $TP[p]$ figurent également les droits d'accès à la page en lecture, écriture et destruction, ainsi que des indications nécessaires à la pagination. Ces indications sont données par des bits, citons le bit V (*valid*) qui indique si p' est bien un numéro de page en mémoire, le bit M (*modified* ou *dirty bit*) qui indique si la page a été modifiée.
- pour trouver l'information cherchée on concatène la partie déplacement dans la page au numéro de page physique trouvé.

Lorsqu'on ne trouve pas la page que l'on cherche en mémoire, on parle de défaut de page (*page fault*, en anglais).

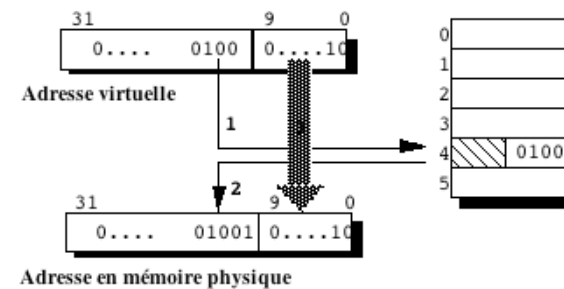
Gestion de la mémoire

Correspondance entre adresse virtuelle et adresse physique

- Principe de traduction d'une adresse virtuelle en adresse physique :



- **Exemple :** les adresses virtuelles et physique sont sur 32 bits, les pages font 1K octets. La page virtuelle 4 est implantée sur la page physique 9. Voici les 3 étapes de passage de l'adresse virtuelle à l'adresse en mémoire :



En allocation par partition, les stratégies sont des stratégies de **placement**, ici on utilise des stratégies de **REPLACEMENT**, les tailles de toutes les partitions (ici des pages) étant égales.

Les principales stratégies sont les suivantes :

- Moins Récemment Utilisée (MRU) / Least Recently Used (LRU) : On suppose que le futur ressemblera au passé. Implique que l'on conserve une trace des **dates** d'accès aux pages.
- Moins Fréquemment Utilisée (MFU) / Least Frequently Used (LFU) : On suppose que le futur ressemblera au passé. Implique que l'on conserve une trace du **nombre** d'accès aux pages. Problème des pages récemment chargées.
- La Plus Ancienne / First In First Out (FIFO) : Implique que l'on conserve une trace de l'**ordre** de chargement.

Stratégies de remplacement

- Les principales stratégies de remplacement (indiquent quelle page doit être remplacée) :
 - Least Recently Used (LRU), la page la moins récemment utilisée: l'algorithme le plus utilisé. Les hiérarchies de mémoires sont gérées LRU.
 - Least Frequently Used (LFU), la page la moins fréquemment utilisée.
 - First In First Out (FIFO), la plus ancienne.

La pile : utilisation des arguments passés à main

- Soit le programme :

```
int main (int argc, char *argv[], char * arge []){
    short i;

    printf("adresse de argc : 0x%x, argc : %d\n",
           &argc, argc );
    printf("adresse de argv : 0x%x, argv : %x\n",
           &argv, argv );
    printf("adresse de arge : 0x%x, arge : %x\n",
           &arge, arge );

    for(i=0 ; argv[i] != 0; i++)
        printf ("argv[%d] (0x%x) : %s\n", i, argv[i], argv[i]);

    for(i=0 ; i < 3 ; i++)
        printf ("arge[%d] (0x%x) : %s\n", i, arge[i], arge[i]);
}
```

- Résultats:

```
adresse de argc : 0xefff6e4, argc : 1
adresse de argv : 0xefff6e8, argv : efff704
adresse de arge : 0xefff6ec, arge : efff70c
argv[0] (0xefff814) : ./a.out
arge[0] (0xefff81c) : ARCH=sparc-sun-solaris2.4
arge[1] (0xefff836) : DISPLAY=:0.0
arge[2] (0xefff843) : EDITOR=emacs
```

La pile : main

- Structure de la pile lors de l'appel à main :
 - nombre de mots sur la ligne de commande : argc,
 - tableau de pointeurs (terminé par un « nul ») contenant les mots de la ligne de commande : argv,
 - tableau de pointeurs (terminé par un nul) contenant les variables d'environnement : arge.

