Gestion de la mémoire

Plan

+ **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 :

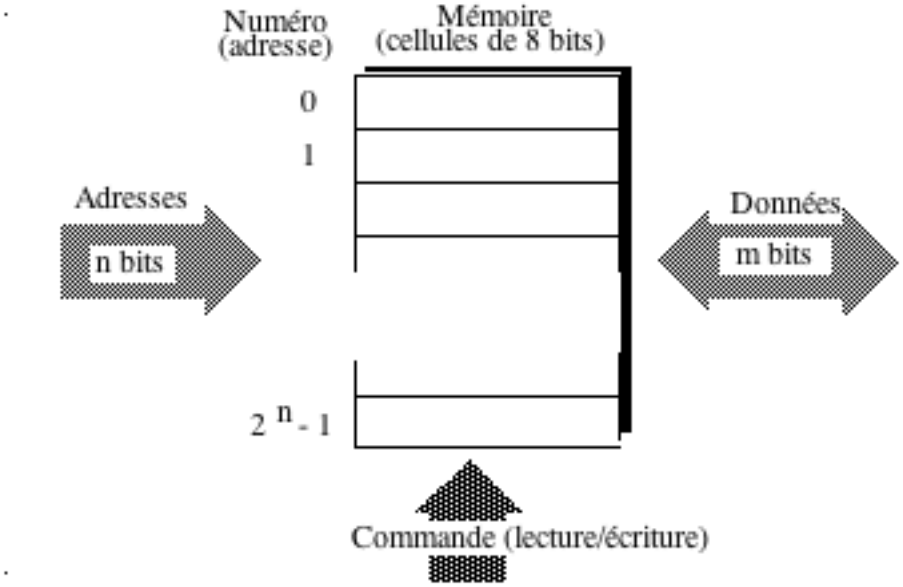
- Sur les machines dotées des processeurs les plus récents, ce type d'architecture est remis en cause. En effet, ces processeurs sont équipés de mémoire appelées caches et ces caches sont souvent distincts pour les données et les instructions. On parle dans ce cas de «*Harvard type cache* ».
- Sur certains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

Pourquoi la mémoire ?

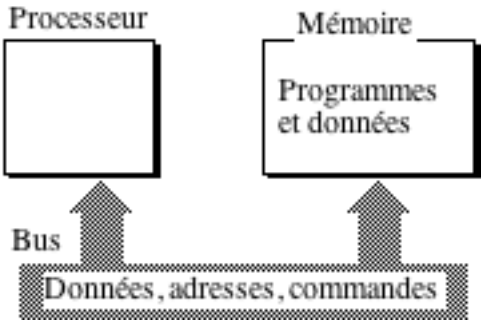
Le processeur va chercher les informations dont il a besoin, c'est-à-dire les instructions et les données, dans la mémoire. Il ne va jamais les chercher sur le disque ! Ce dernier sert de support d'archivage aux informations, en effet la mémoire s'efface lorsque on coupe l'alimentation électrique de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

La mémoire, ressource du S.E

- La mémoire est un 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



```
struct noalign
{
   char c;
   double d;
   int i;
   char c2[3];
}
```

le sizeof(noalign) va donner une valeur en octets qui dépend fortement de l'arachitecture et du compilateur.
On pourrait croire que sizeof(noalign) = 1 + 8 + 4 + 3*1 = 16 .
Or la taille observée est sizeof(noalign) = 24 .
Ceci met en évidence un phénomène d'alignement mémoire. En effet, le compilateur rajoute des bits dits de « padding » pour aligner les données avec des multiples de 2,4,8… selon les caractéristiques du processeur cible, de la largeur du bus d'adresse et de la largeur du bus de données.
Par exemple dans ce cas la structure noalign ressemble en réalité à ceci :

```
struct noalign
{
   char c;
   char pad1[7];
   double d;
   int i;
   char c2[3];
   char pad2;
}
```

On remarque que pad1 permet à d de débuter à une adresse multiple de 8 et pad2 complète la structure pour atteindre 24 (multiple de 8)
Par contre la structure suivante qui contient les mêmes membres mais dans un ordre différent est bien alignée avec des multiples de 8 :

```
struct align
{
   double d;
   int i;
   char c2[3];
   char c;
}
```

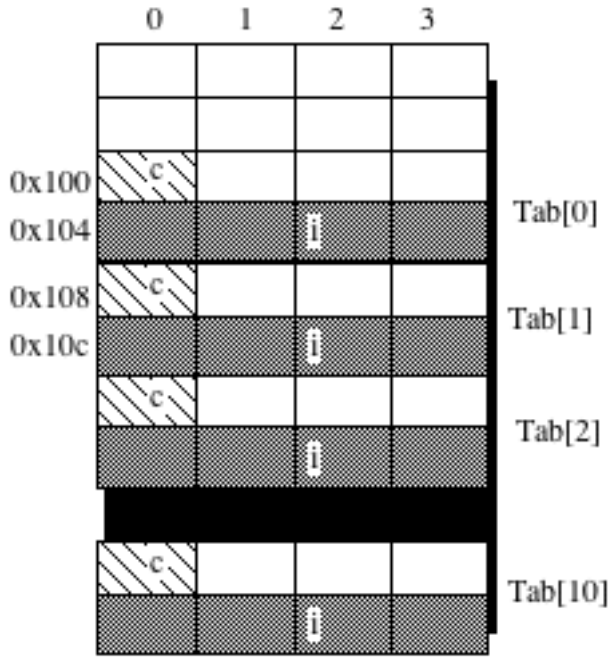
Et donc le sizeof(align) = 16

Mémoire : alignement

- Conséquence de l'accès 32 bits : allocation mémoire d'un tableau de structures sur une machine alignant sur 32 bits (le tableau est implanté à partir de l'adresse `0x100` :

```
struct S{
        char c;
        int i;
}


struct S Tab[10];
```



- Ici, `sizeof(S)` renvoie la valeur 8 et non pas 5 !

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).

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.

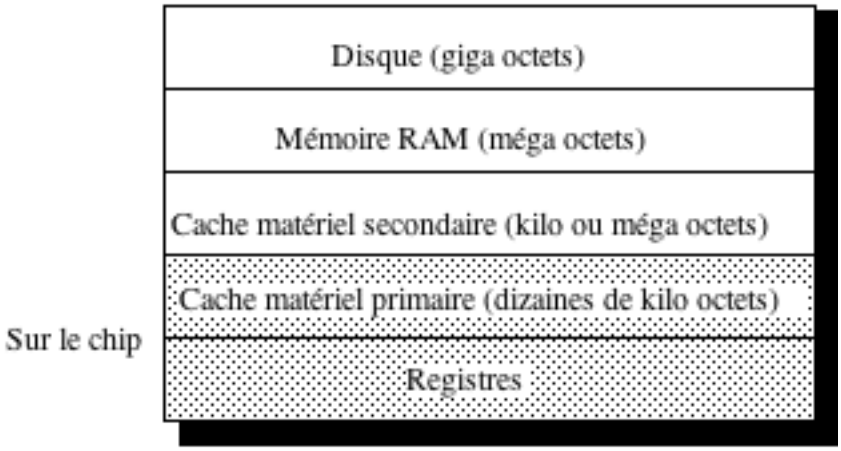
Pour pallier ce défaut, les constructeurs implantent de la mémoire sur les "*chips*" c'est à dire sur les processeurs eux-mêmes. Ces mémoires sont appelées "caches". Elles contiennent un sous-ensemble de la mémoire RAM.

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.

Hiérarchie 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

Problèmes liés à la gestion de la mémoire en partitions :

• Fragmentation interne :
-Un processus n'utilise pas la totalité de la partition.

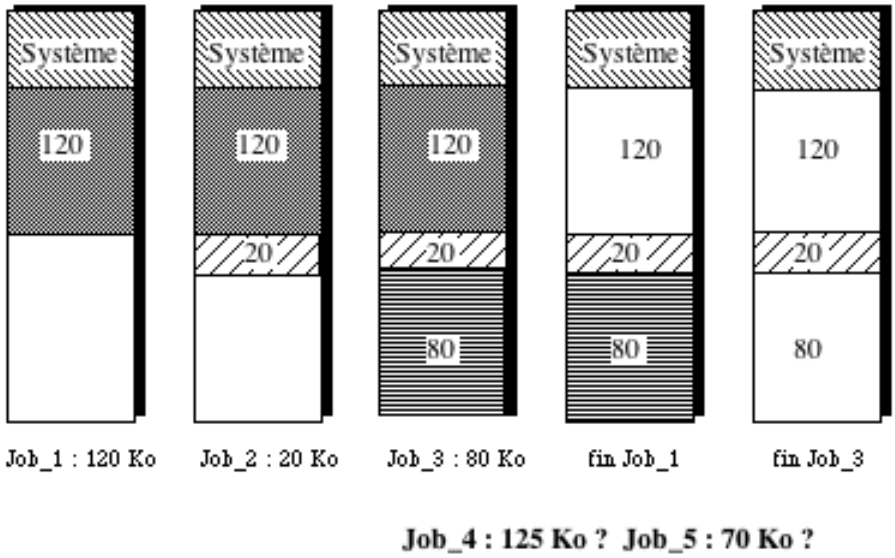
• Fragmentation externe :
-Certaines partitions ne sont pas utilisées, car trop petites pour tous les travaux éligibles.

• Comment choisir a priori la taille des partitions ?

Allocation contiguë : les partitions

• Comment partager la mémoire entre les différents processus ?

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 :



- 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.
- Remarque : l'isolation spatiale induite par le partitionnement permet la protection des accès mémoire.

Politiques de placement des partitions

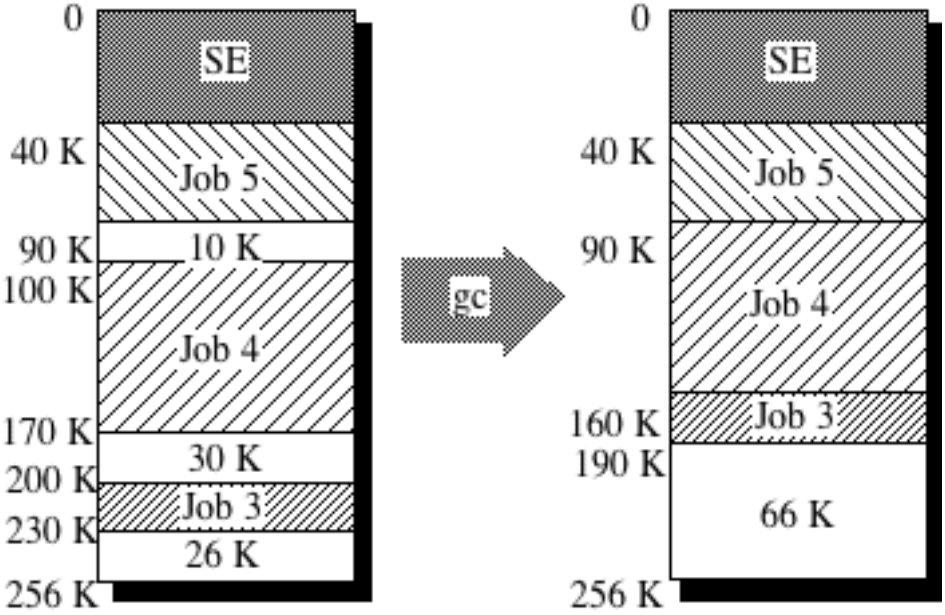
- Si plusieurs partitions sont susceptibles d'accueillir le processus à charger (partitions libres), plusieurs politiques de **placement** sont possibles :
  - *First Fit* : on place le processus dans la première partition libre dont la taille est suffisante
  - *Best Fit* : on place le processus dans 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 nombreuses partitions libres minuscules non-réutilisables, on utilise généralement *Worst Fit* en classant les partitions libres par ordre de tailles décroissantes, puis place le processus dans la première partition libre dont la taille est suffisante
- Dans le cas où il n'existe aucune partition libre assez grande, on peut faire du *garbage collecting* ou ramasse-miettes (cf. plus loin).

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 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 + ε/2 (ε> 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") = Nc/N
  - proba(M baisse de 1) = proba(on trouve un trou de bonne taille) + proba (on libère un bloc "a") = 1-p+(Na/N)
- A l'équilibre : proba (M augmente de 1) = proba (M baisse de 1) donc :
  - Nc/N = (Na/N)+1-p
  - puisque N-2M ~ Nc-Na on a M = N/2 * p
  - si p ~ 1 alors M ~ N/2 ou encore N = 2 * M

Note : ~ signifie ici « proche de »

- 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

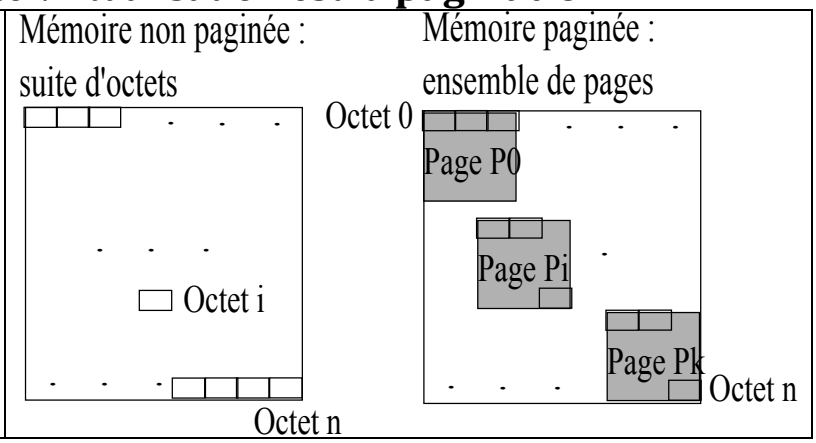
    - Partitions de taille fixe, de taille variable

\+ **3. Pagination et mémoire virtuelle**

Pagination (1/3) : la mémoire

- La mémoire est une ressource de taille finie, le système d'exploitation va en donner une représentation « logique », la virtualiser, pour qu'elle apparaisse comme une ressource disponible sans limitation.
- La première étape de cette virtualisation est la **pagination** :

La mémoire est divisée en sous-ensembles de même taille, appelés **pages physiques** (*frame* en anglais)



- Dans un espace mémoire paginé, les adresses sont structurées en paires :

adresse en mémoire physique (adresse physique) : (NP_{PHI}, DEP)

où NP_{PHI} est le numéro de page (*frame*) et DEP le déplacement, en octets, dans cette page. Par exemple, si la taille des pages est de 1Ko (1024 octets) :

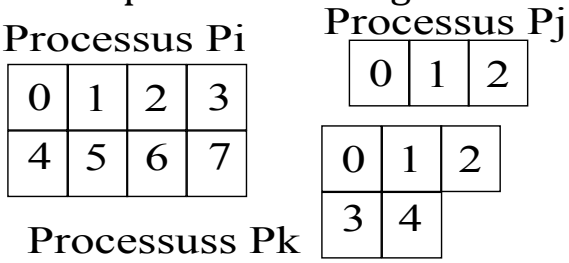
adresse physique 2026 = (1, 1002)

Pagination (2/3) : les processus

- L'espace mémoire alloué aux processus est lui aussi divisé en pages, appelées **pages logiques** (*page* en anglais) , numérotées de 0 à N_{proc} pour chaque processus.
- Par exemple, soit trois processus Pi, Pj et Pk, et des pages de 1 Ko :

Processus	P_i	P_j	P_k
Espace d'adressage nécessaire (en Kilo octets)	8	3	5

- Représentation de l'espace d'adressage de ces trois processus :



- Dans chaque processus, les adresses sont structurées en paires :

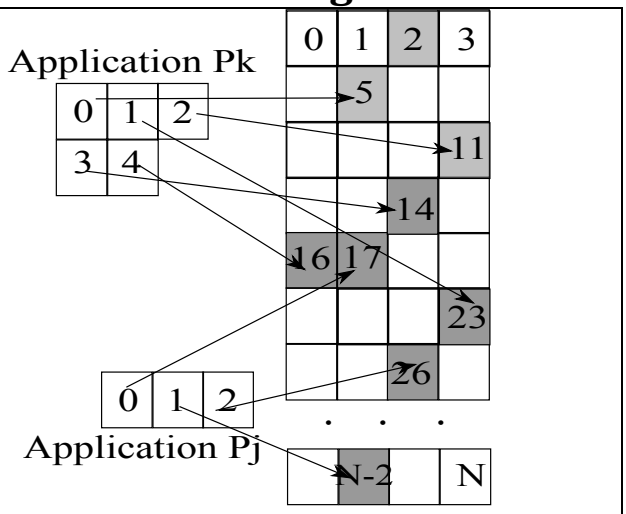
adresse en mémoire programme (adresse logique): (N_{PL}, DEP)

où N_{PL} est le numéro de **page logique** et DEP le déplacement dans la page, en octets, dans la page.

Pagination (3/3) : allocation des pages

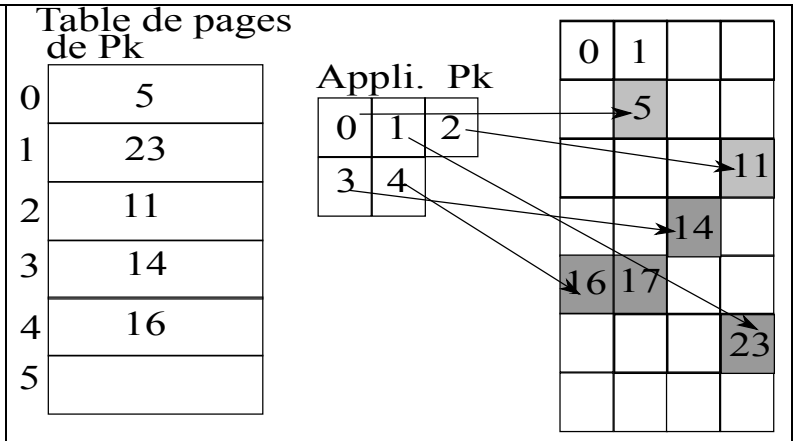
- Dans cet exemple, la taille des pages physiques est de 1Ko (1024 octets), les pages physiques libres sont représentées en grisé.
- On constate que le plus grand bloc libre fait 2Ko (page physiques 16 et 17 contiguës). En allocation contiguë, si on voulait maintenant charger en mémoire les processus Pj (3Ko) ou Pk (5Ko) il faudrait faire du *garbage collecting*.
- **La pagination va lever cette contrainte de contiguïté** :

Dans un espace paginé, aux algorithmes de placement se substituent des algorithmes de **remplacement :** les pages logiques sont chargées au fur et à mesure des besoins sur les pages physiques libres, même si ces dernières ne sont pas contigües.



La table de pages (1/3) : rôle

- Comment retrouver une information se trouvant dans une page logique numérotée NPL qui a été chargée sur une page physique NP_{PHI} ?
- On utilise une table, appelée **table de pages**. Il y a une table de pages par processus.
- Soit TP cette table, à chaque chargement d'une page logique NPL sur une page physique NP_{PHI}, on fait : TP[NPL] = NP_{PHI}

On aura donc pour le processus P_K précédent :	

La table de pages (2/3) : exemple

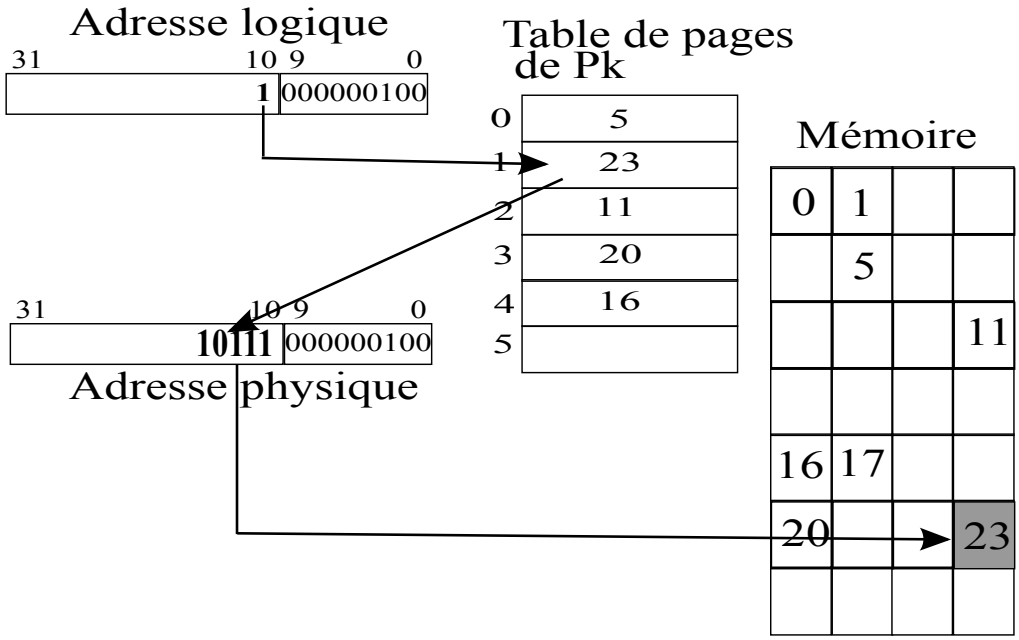
Soit la variable `i` qui a une adresse en mémoire programme (adresse logique) 1028 : on dit qu'elle est dans l'espace d'adressage de P_K .

On voit bien que l'exécution de `i=4` ne provoque pas d'écriture de la valeur 4 à l'adresse physique 1028, c'est-à-dire dans la page physique 1 en mémoire ; en effet, 1028 est ici une adresse en mémoire programme (adresse logique).

En réalité, la page logique 1 de P_K est chargée sur la page physique 23! L'adresse physique sera donc différente de l'adresse logique.

La table de pages (3/3) : exemple

• L'utilisation de la table de page permet la **traduction** des adresses logiques en adresses physiques :



• **Remarque :** chaque processus voit son propre espace d'adressage numéroté à partir de 0, mais chaque espace est appliqué sur un espace physique différent -> **isolation et protection** au niveau de la page (cf. Android, ...)

Adresses virtuelles (1/4)

- La pagination permet la **virtualisation** de l'espace d'adressage :

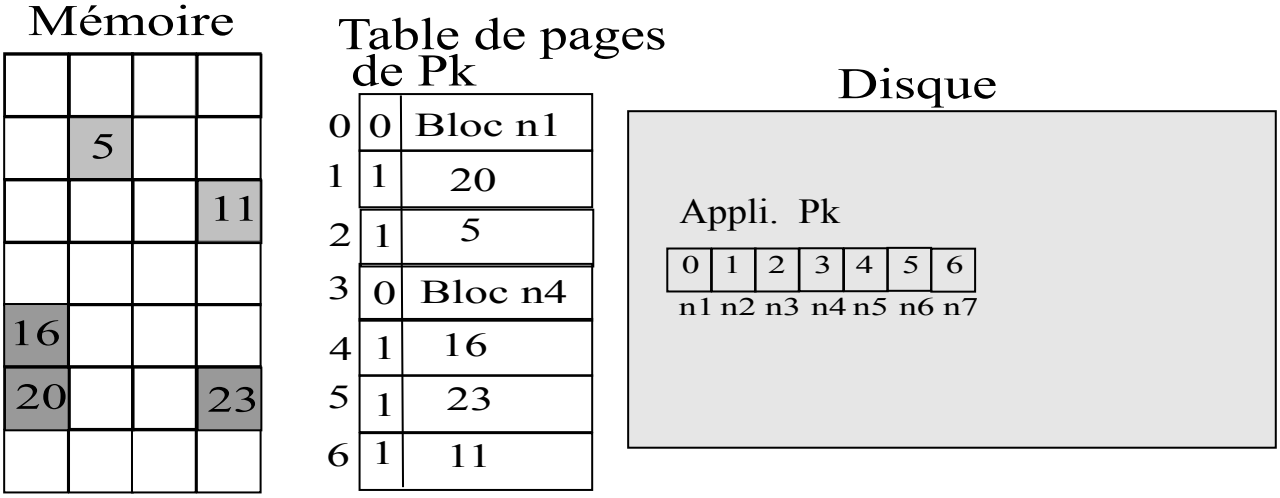
  1. elle permet de charger une application de façon non contigüe en mémoire,

  2. mais, de plus, elle autorise l'exécution d'une application **même si la taille de l'espace disponible en mémoire physique est inférieure à celle requise par l'application**.

  3. on ne charge en mémoire physique qu'un sous-ensemble des pages logiques de l'application, le reste des pages logiques peuvent être écrites sur un autre support comme un disque dur. La table de pages indique l'emplacement exact : c'est à dire si une page est en mémoire physique ou sur un autre support disque.

Adresses virtuelles (2/4)

- Exemple de scénario :
  - soit un processus P_k (7Ko), une mémoire à pages physiques de 1Ko et un disque dont les blocs font 1Ko. P_k occupe 7 blocs sur le disque: n1 à n7.
  - Il reste 5 pages physiques disponibles en mémoire : 5, 11, 16, 20 et 23.
  - En cours d'exécution de Pk, la configuration peut être la suivante :



- On remarque qu'un élément a été ajouté dans les entrées de la table de pages qui permet de déterminer l'emplacement d'une page logique: un indicateur de **validité** (1 : page en mémoire, donc l'entrée donne un numéro de page physique, 0 : numéro de bloc disque où se trouve la page)

Adresses virtuelles (3/4) : bilan

- Comme on vient de le voir, la mémoire requise par un processus peut être **supérieure à celle de la mémoire disponible** L'espace d'adressage d'un processus étant potentiellement plus grand que l'espace d'adressage de la mémoire physique, on parle donc d'espace d'adressage virtuel et d'adresses virtuelles pour les processus. :

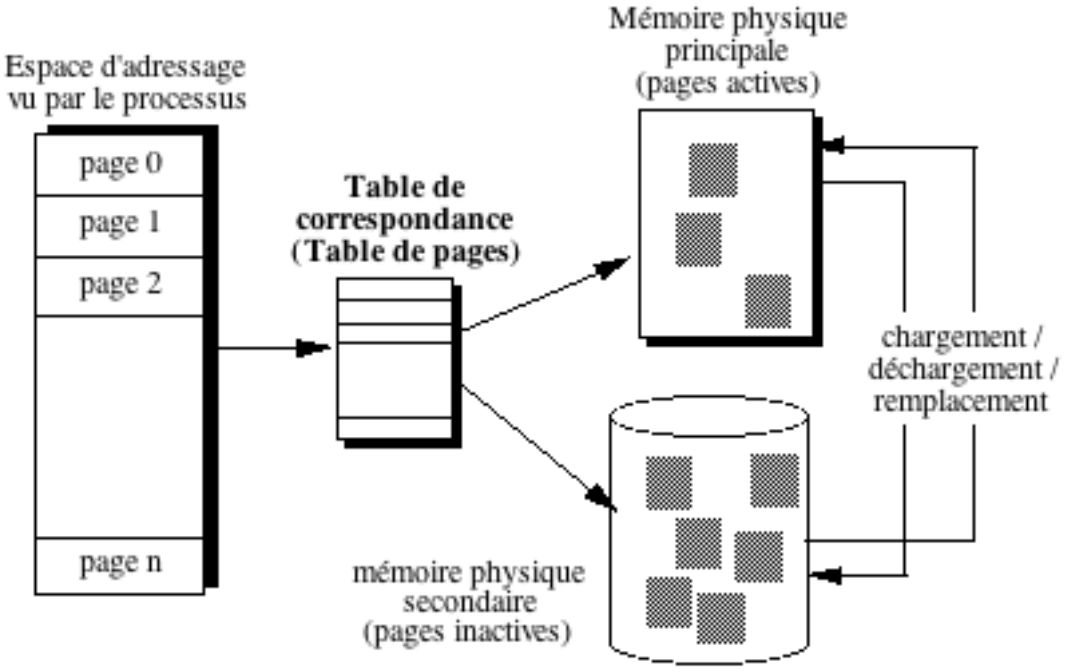
adresse virtuelle: (Numéro de page virtuelle, déplacement dans la page)

Adresses virtuelles (4/4) : bilan

- L'accès à une information en mémoire se fait comme suit :
  1. découpage de son adresse en numéro de page virtuelle (NPV) et déplacement,
  2. accès à l'entrée TP[NPL] de la table de pages TP du processus:
     a. indicateur de validité V =1 ➔passage de page logique en page physique, accès à l'information par traduction de l'adresse logique en adresse physique,
     b. indicateur de validité V = 0 ➔page sur le disque**, défaut de page** (*page fault*)
        - action : remplacer une des pages physique (appelée NPR) par NPV, recopier NPR sur disque si elle a été modifiée, mettre à jour TP[NPL] et TP[NPR].
        - accès à l'information par traduction d'adresse

Virtualisation de la mémoire : illustration

- L'espace d'adressage d'un processus est divisé en pages virtuelles :
  - dans la plupart des cas, seul un sous-ensemble de ces pages est effectivement en mémoire,
  - la table de pages indique, pour chaque page virtuelle utilisée, si elle est, ou non, présente en mémoire



- Sur cet exemple, seules 3 pages parmi 6 sont présentes en mémoire.

Pagination et 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 taille 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).**

Mémoire virtuelle : Point de vue utilisateur

- La mémoire virtuelle permet d'exécuter :
  - simultanément plusieurs processus dont la somme des espaces d'adressage est supérieure à la taille de la mémoire physiques
  - une application dont la taille est supérieure à la taille de la mémoire physique,

- La taille d'une application est donc limitée par celle d'une adresse :
  - n bits d'adresses -> 2^{32} octets d'espace d'adressage
  - en pratique un paramètre système limite cette taille

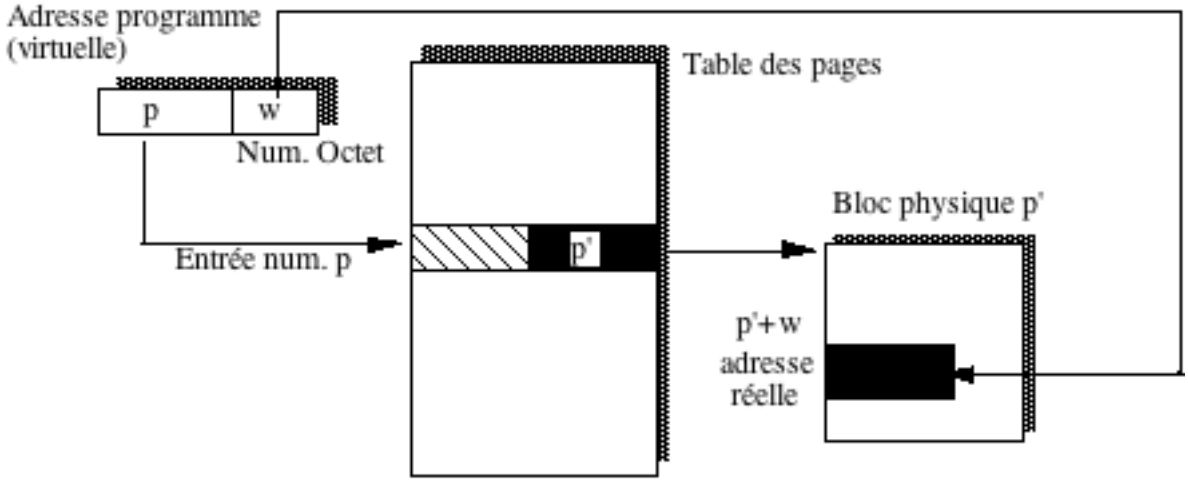
Passage de l'adresse virtuelle à l'adresse physique :

- 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.

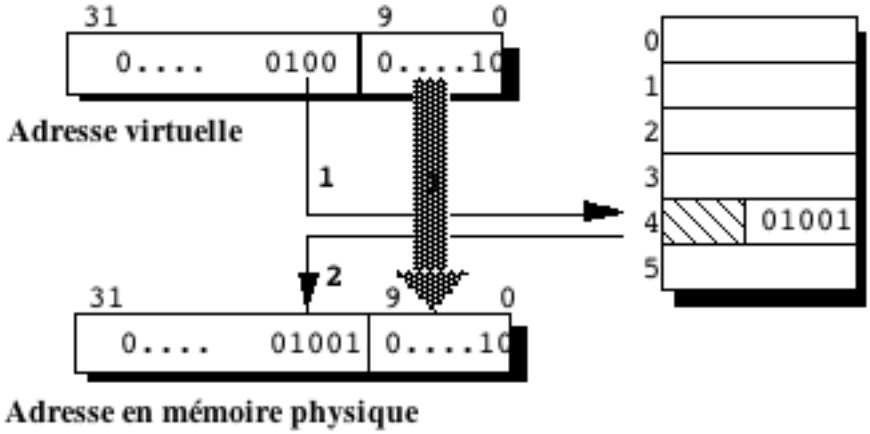
Lorsqu'on ne trouve pas la page que l'on cherche en mémoire, on parle de défaut de page (*page fault*, en anglais).

Traduction d'une adresse virtuelle en adresse physique

- Principe:



- **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 la traduction :



En allocation par partition, les stratégies sont des stratégies de **placement**, ici on utilise des stratégies de **REMPLACEMENT**, 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

- Ces stratégies permettent de choisir quelle page virtuelle doit être remplacée par la page virtuelle courante :
  - Least Recently Used (LRU), la page la moins récemment utilisée. C'est 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.

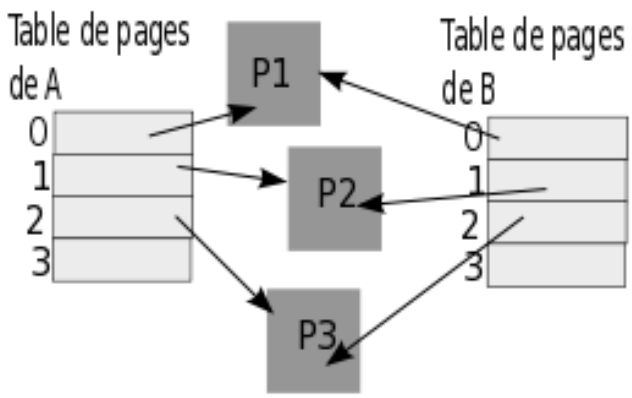
Exemple-1: partage de code (1/5)

- Contexte :
  - Taille des pages : 1K octets, algorithme de remplacement : LRU,
  - Une table de page par région (code, données et pile) pour chaque processus,
  - Il reste 6 pages libres (P1 à P6) en mémoire. **On ne s'intéresse qu'à la gestion de la région de code des processus.**

Exemple-1: partage de code (2/5)

- Scénario :
  - Un processus, le processus A, exécute un programme dont la taille est 3K octets. 3 pages en mémoire (P1, P2, P3) parmi ces 6 pages libres sont utilisées pour ranger ce programme.
    1. les trois premières entrées de la table de pages de A pointent donc vers P1, P2 et P3.
  - Ce processus A fait appel à `fork()` et crée le processus B :
    1. les trois premières entrées de la table de pages de la région de code de B pointent alors vers la mêmes pages mémoire que celles utilisées par A : P1, P2 et P3 (partage de code) comme illustré ci-après.

Exemple-1: partage de code (3/5)

Etat des tables de pages des processus A et B :	

• Remarques :

Ce partage des exécutables est mis en œuvre systématiquement (il est facile à faire : les pages n'étant pas modifiées, il n'y a pas de problème de concurrence):

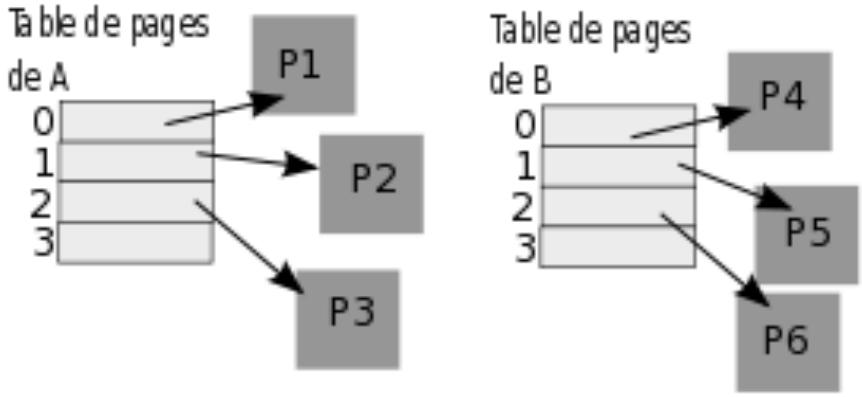
1. il n'y a qu'un seul exemplaire d'un programme en mémoire, l'espace qui lui est alloué est partagé par **tous** les processus qui l'utilisent
2. il n'y a qu'un seul exemplaire en mémoire **d'une bibliothèque dynamique** : cf. la bibliothèque d'entrées sorties (gain de place !)

Exemple-1: partage de code (4/5)

• Le processus B fait maintenant appel à `exec()` pour charger un fichier dont la taille est **4K octets**.

C'est maintenant que va se faire l'allocation en mémoire des pages de code pour le processus B :

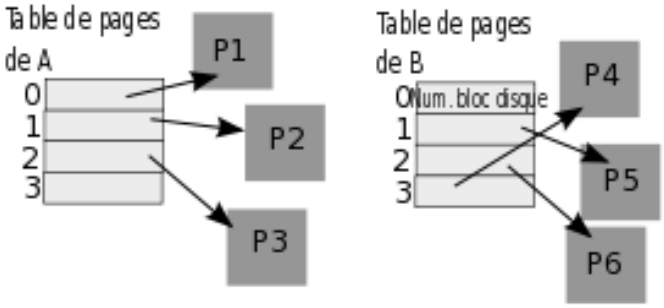
Les trois premières entrées de sa table de pages sont actualisées et pointent vers P4, P5 et P6, les trois dernières pages libres en mémoire,



Au chargement de la quatrième page du fichier, il y n'y plus de page libre, on applique donc LRU sur les trois pages déjà allouées en mémoire pour le processus B.

Exemple-1: partage de code (5/5)

• Etat des tables de pages des processus A et B après chargement de l'exécutable traité par le processus B :



• Remarques :

1. L'algorithme LRU travaille sur les dates **d'accès aux informations** se trouvant dans les pages, non pas sur les dates de chargement des pages en mémoire, la page LRU aurait pu être une autre que la page 0.
2. La page virtuelle remplacée n'est recopiée sur disque avant son remplacement que si elle a été modifiée.

Exemple-2: allocation d'un tableau (1/3)

• Contexte :

1. L'algorithme de remplacement est LRU, la taille des pages est de 4K octets ,
2. On suppose qu'il reste 3 pages libres en mémoire : P10, P20 et P30,
3. On ne s'intéresse qu'aux pages de données,
4. On ne prend pas en compte les indices de boucles : on suppose qu'ils sont implantés dans des registres.
5. Le tableau `Tab` n'est plus en mémoire avant l'exécution de la boucle du programme ci-dessous.

```
long Tab[4096];  /* sizeof(long) = 4 octets */
...
for (i=0; i<4096; j++) Tab[i]=0;
```

Exemple-2: allocation d'un tableau (2/3)

- Description du chargement du tableau `Tab` en mémoire et évolution de la table de pages lors de l'exécution de la boucle :
  1. Il faut 4 pages pour implanter le tableau, en effet : `4096` (2^{12}) mots de 4 octets = 16 Ko = 4 pages.
  2. Lors des références à `Tab[0]`, `Tab[1024]`, `Tab[2048]`, on charge respectivement les pages 0, 1 et 2 du tableau sur les pages physiques P10, P20 et P30,
  3. Lors de la référence à `Tab[3072]`, il n'y a plus de page libre. La première page du tableau (rangé sur la page mémoire P10) est la page LRU. Son contenu est donc recopiée sur le disque parce qu'elle a été modifiée, et la page P10 est utilisée pour stocker les valeurs suivantes de `Tab`

- Etat de la table de pages après chargement du dernier Ko du tableau :

0 (entrée pour 1/4)	Numéro bloc disque où se trouve la première page du tableau.
1 (entrée pour 2/4)	P20
2 (entrée pour ¾)	P30
3 (entrée pour 4/4)	P10
…	…

Exemple-2: allocation d'un tableau (3/3)

- Si on remplace la boucle précédente par ces instructions :

```
for (i=0; i<4096; j++){
  if (i %1024 == 0) Tab[0]=0;
  Tab[i]=0;
}
```

- A chaque changement de page, on fait accès à la première page. Elle n'est donc plus LRU lors du chargement de la quatrième page. C'est la deuxième qui est maintenant LRU :

- Etat de la table de pages après chargement complet du tableau :

0 (entrée pour 1 /4)	P10
1 (entrée pour 2/4)	Numéro bloc disque où se trouve la deuxième page du tableau.
2 (entrée pour ¾)	P30
3 (entrée pour 4/4)	P10
…	…

A propos des« fautes » sur les accès mémoire

• Défaut de segmentation (*Segmentation fault) :*

Il s'agit d'une référence vers une adresse dont l'accès est interdit dans le mode (lecture, écriture) demandé :

- l'accès demandé ne se fait pas. Sur occurrence de cette « faute », le système reçoit une interruption, qu'il traite en envoyant le signal SIGSEGV au processus ; par défaut, cela provoque la terminaison du processus fautif (sauf s'il a changé le comportement associé à SIGSEGV en utilisant la fonction signal).

• Défaut de page (*page fault*) :

Il s'agit d'une référence à une adresse dont le mode d'accès demandé est autorisé, mais qui se trouve dans une page qui n'est pas présente en mémoire :

- Il faut aller chercher cette page sur le disque.
- Une fois la page chargée en mémoire et faite la mise à jour de la table de page, on **recommence** l'exécution de l'instruction.