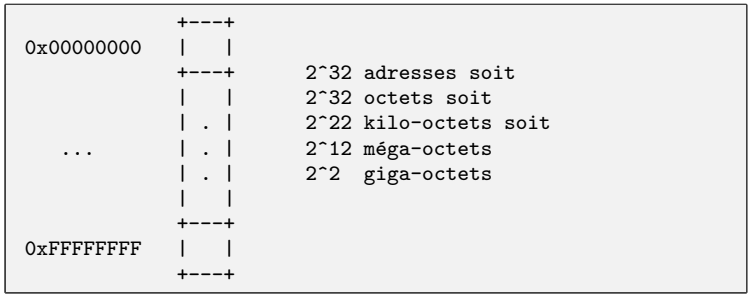
Allocation de la mémoire centrale

- Mémoire centrale (RAM)
- Organisation logique / physique
- Organisation en partitions
- Pagination et segmentation

La **mémoire centrale** (**RAM** pour **Random Access Memory**) est une zone de stockage composée **d'octets** (8 bits).

Chaque octet est repéré par un **adresse physique** (sur 32 ou 64 bits).



La mémoire physique est **contigüe** (les adresses varient de N à M).

Uniformité : tous les processeurs ont accès à tous les octets.

Les **mots** doivent être alignés sur une adresse physique multiple de leur taille :

```
struct S1 {
    char c1;
    int v1;
    char c2;
};

void main(void) {
    printf("sizeof(char) = %d\n", sizeof(char));
    printf("sizeof(int)  = %d\n", sizeof(int));
    printf("sizeof(S1)   = %d\n", sizeof(struct S1));
}
```

Exécution :

```
sizeof(char) = 1
sizeof(int)  = 4
sizeof(S1)   = 12
```

Placement des données en mémoire :

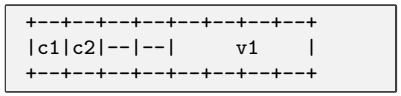
Il est **préférable** d'écrire :

```
struct S1 {
    char c1;
    char c2;
    int v1;
};
```

Exécution :

```
sizeof(char) = 1
sizeof(int)  = 4
sizeof(S1)   = 8
```

Placement des données en mémoire :



Avec la **RAM** il existe trois opérations :

- **lecture** d'un octet ou d'un mot
- **écriture** d'un octet ou d'un mot
- **lecture pour exécution** d'un mot

▶ Utilisation de la mémoire ◀

Un exemple :

```
int x;

void main(void) {
    sleep(1);
    printf("%x ", &x);
    sleep(1);
}
```

Exécution :

```
6008e8 6008e8 6008e8 6008e8 6008e8 6008e8 6008e8 6008e8 6008e8
```

Chaque processus travail dans une **mémoire logique** qui est une partie de la mémoire centrale.

Les octets de la mémoire logique sont repérés par des **adresses logiques**.

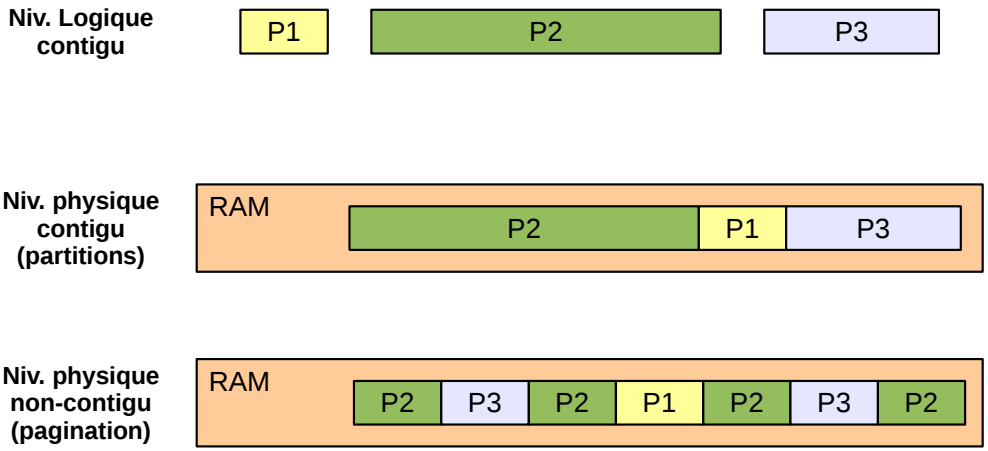
Lors de l'exécution, les processus génèrent des adresses logiques qui varient de 0 à $N-1$ (N étant la **taille de la mémoire logique du processus**).

▶▶ Allocation de mémoire ◀◀

- **Correspondance** entre adresses logiques et adresses physiques,
  - ▷ **fixe** : établie à la compilation
  - ▷ **statique** : établie au chargement
  - ▷ **dynamique** : variable dans le temps
- **Gestion** de la mémoire physique.
- **Partage** de données entre processus.
- **Protection** de chaque processus.

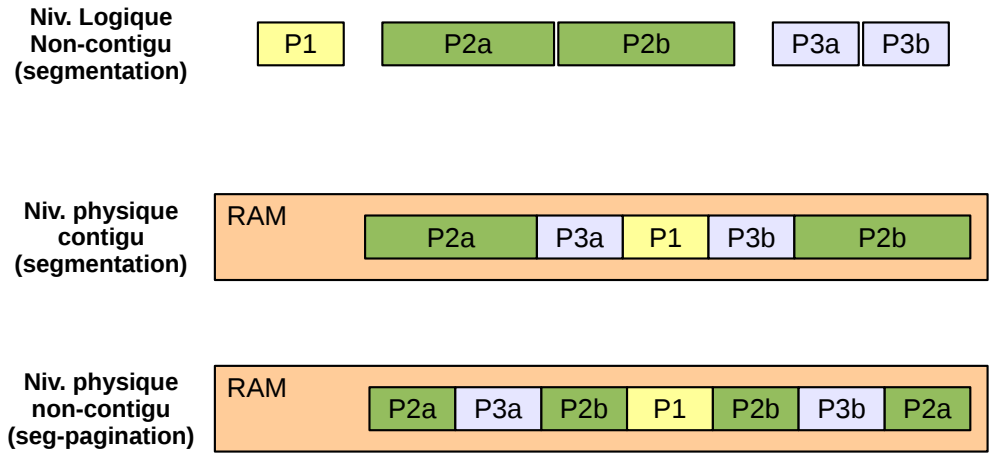
▶ Mémoire logique contiguë ◀

La mémoire logique des processus est constitué **d'un seul morceau** (**une partition**) :



▶ Mémoire logique non contiguë ◀

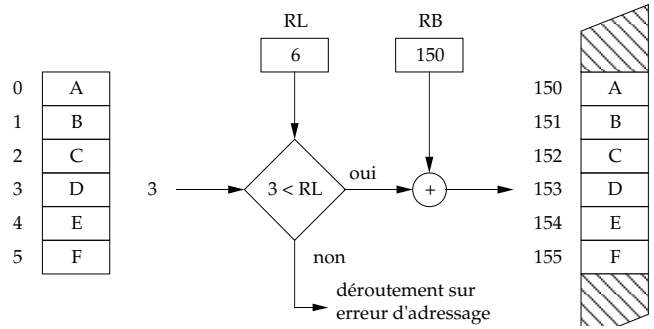
La mémoire logique des processus est constitué de **plusieurs morceaux** (**segments**) :



▶ Système à partitions variables ◀

Les partitions sont **allouées et libérées à la demande** (création ou fin d'un processus).

Le **registre de base** pointe sur la partition et le **registre limite** en indique la taille.



`RB` et `RL` sont utilisés **par la CPU** pour traduire **à chaque accès mémoire** les adresses logiques en adresses physiques.

▶ Fragmentation externe/interne ◀

Algorithme d'allocation :

- Algorithme de chaînage des zones libres
- Algorithme par subdivision

8					
A	2		4		Alloc. de A, longueur = 2
A	B	1	4		Alloc. de B, longueur = 1
A	B	1	C	2	Alloc. de C, longueur = 2
2	B	1	C	2	Libération de A
2	B	1	4		Libération de C
8					Libération de B

Conséquences :

- La **fragmentation externe** est due à l'émiettement de la mémoire lors des allocations/libérations.
- La **fragmentation interne** c'est l'unité de mémoire minimum que le S.E. est capable de gérer (généralement plusieurs Kilo-octets).

▶▶ Mémoire paginée ◀◀

- Principe
- Correspondance des adresses
- Mémoires associatives
- Partage de pages

▶ Mémoire paginée : principe ◀

La mémoire est divisée en **page de taille fixe** (quelques Kilo-octets). Cette taille est toujours une **puissance de deux** (2^m).

Une **adresse logique paginée** sur n bits (avec $n > m$) est un couple

$$\langle \underbrace{\text{n° de page logique}}_{\text{sur } n-m \text{ bits}} , \underbrace{\text{déplacement dans la page}}_{\text{sur } m \text{ bits}} \rangle$$

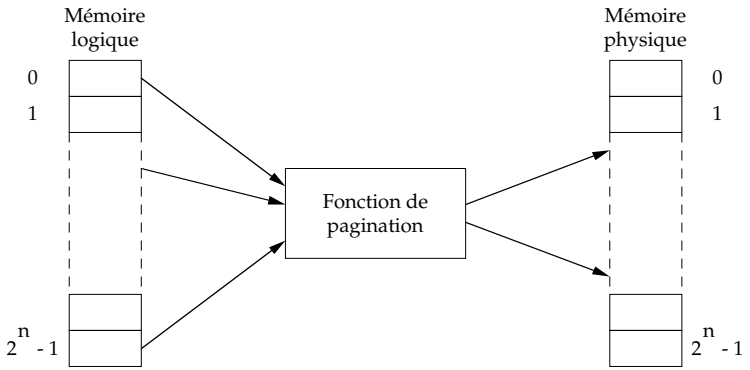
Une **adresse physique paginée** sur p bits (avec $p > m$) est un couple

$$\langle \underbrace{\text{n° de page physique}}_{\text{sur } p-m \text{ bits}} , \underbrace{\text{déplacement dans la page}}_{\text{sur } m \text{ bits}} \rangle$$

Exemple : avec une page de 4 ko (2^{12} octets) :

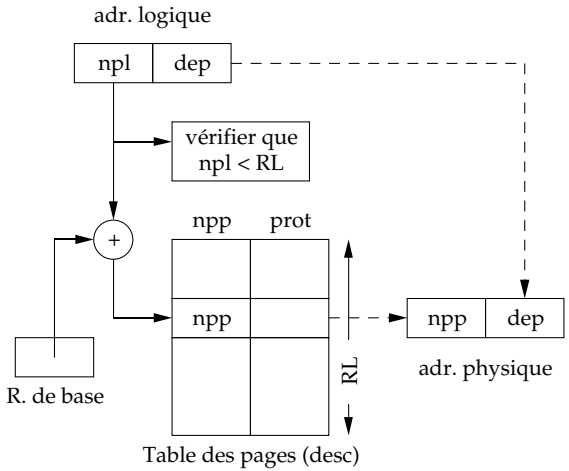
```
123.456.789  =  |00000111010110111100|110100010101|
                |<------- 20 ------->|<--- 12 --->|
                       30140              3349

     =      (30149 * 4096)   +    3349
```

La **fonction de pagination** assure la correspondance entre numéro de page logique et numéro de page physique.



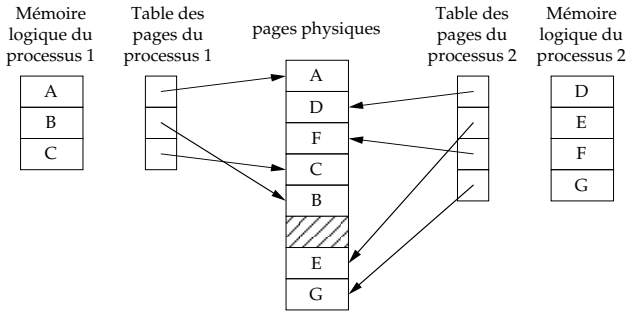
▶ Pages logiques versus pages physiques ◀

Pour chaque processus, le système prépare une **table de pages logiques** (notée `desc` ci-dessous).



▶ Exemple et discussion ◀

Un exemple avec deux processus :



Avantages :

- Gestion mémoire **plus simple** (liste des pages libres)
- Compactage **inutile**
- Protections différentes pour chaque page

Inconvénients :

- temps d'accès doublé
- nécessite une **PMMU** (**Page Memory Management Unit**)

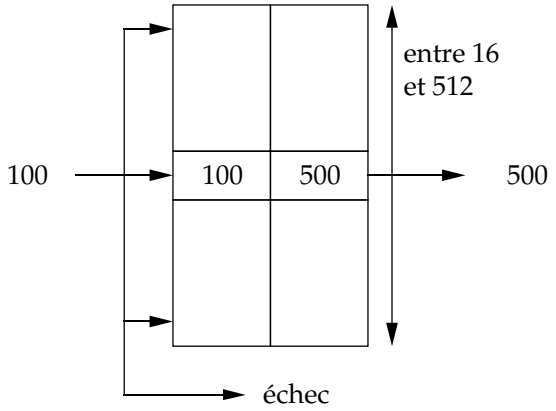
▶ Comportement des processus ◀

Comportement « **en moyenne** » des processus :

- **Non uniformité** : 20% des pages regroupent 90% des accès
- **Principe de localité** :
  - ▷ stabilité des accès sur une **courte période**
  - ▷ l'activité actuelle est une **bonne estimation** de l'activité future

▶ Mémoires associatives ◀

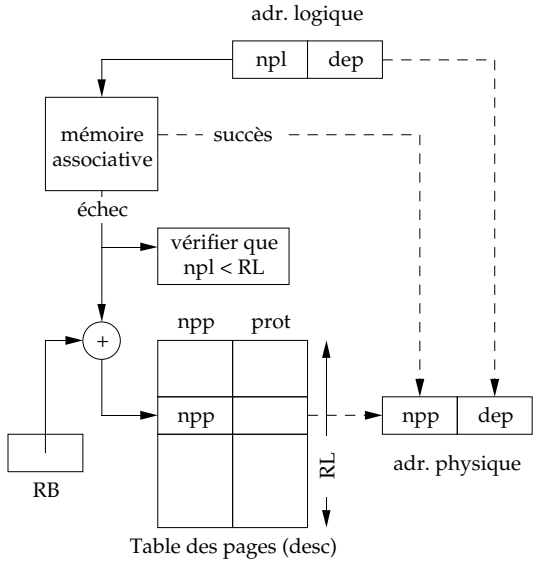
Principe des **mémoires associatives** :



- **Rapidité** : les tests sont faits **en parallèle** (quelques nanosecondes)
- Ces circuits sont **très couteux**

▶ Mémoires associatives et pagination ◀

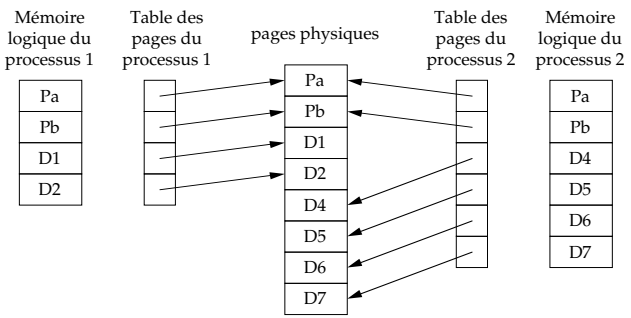
Principe : retenir les derniers couples (**page logique**, **page physique**), pour éviter l'accès mémoire à la table des pages.



Conséquences :

- Il faut mettre à jour la mémoire associative après les échecs
- Il faut vider la mémoire associative lors des changement de processus
- Le taux de réussite est lié à la taille de la M.A. (entre 80% et 95%).
  - ▷ $0,80 \times (100 + 20) + 0,20 \times (100 + 100 + 20) = 140$ ns
  - ▷ $0,95 \times (100 + 20) + 0,05 \times (100 + 100 + 20) = 125$ ns

▶ Partage de pages entre processus ◀



Les pages contenant le programme (Pa et Pb) sont partagées, mais les pages de données (D1, …, D7) ne le sont pas.

Les pages contenant le programme Pa et Pb sont **partagées**, tandis que les pages Dx ne le sont pas.

Pour une même page physique, il est possible d'avoir des **protections différentes** suivant le processus qui l'utilise.

Gestion d'une mémoire virtuelle paginée

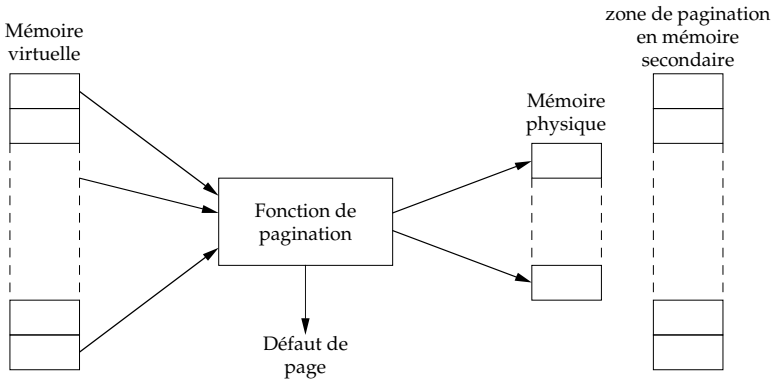
Principe : les programmes utilisent 20% de leur page, donc il est inutile de toutes les conserver en mémoire.

Exemple : 1000 pp = 10 processus de 100 pages logiques ou 50 processus de (100 x 0,2) pages utiles.

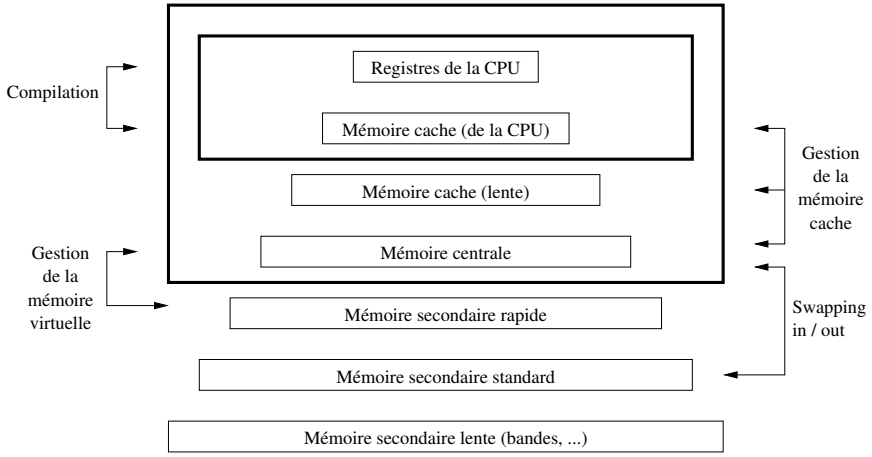
Le système doit détecter (avec l'aide du matériel) :

- les pages **inutilisées** (**réquisition**)
- les pages **utiles** et **présentes** en mémoire physique
- les pages **utiles** et **absentes** de la mémoire physique (**défaut de page**)

Fonction de pagination virtuelle :

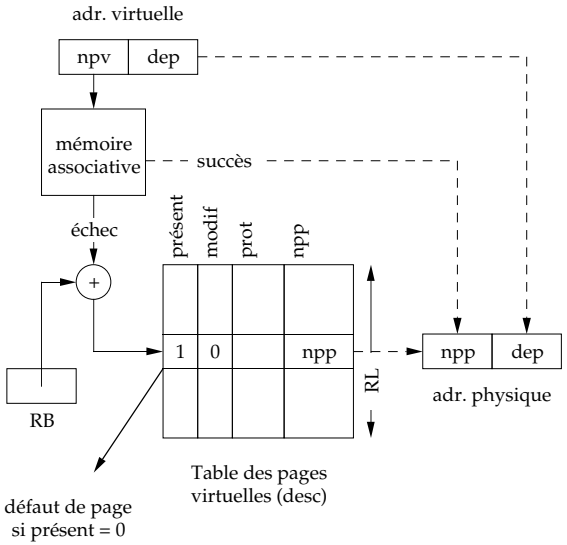


La mémoire virtuelle implante la **gestion d'un cache** :



► Adresses virtuelles versus adresses physiques ◄

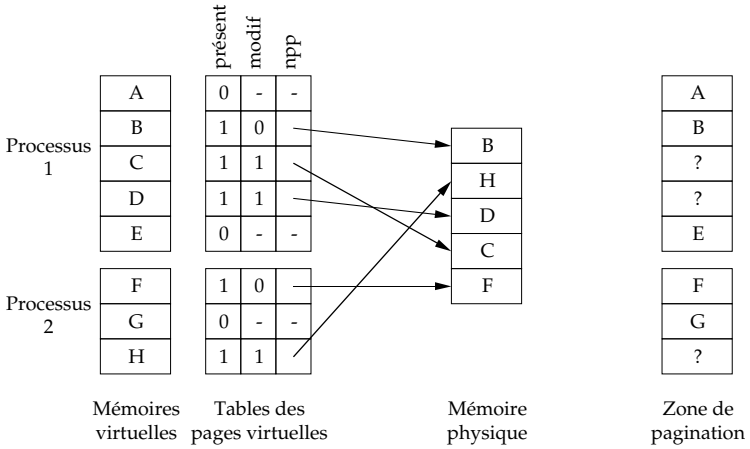
Pour chaque processus, le système prépare une **table des pages virtuelles** (pointée par le registre de base) :



Correspondance des adresses (c'est la partie **matérielle** de la pagination) :

```
fonction transformation( adr : adresse_virtuelle )
    ⟨npv, depl⟩ := adr
    si (npv >= RL) interruption erreur d'adressage

    -- lecture de la table des pages virtuelles
    ⟨présent, prot, npp⟩ := mem[ RB + npv ]

    -- vérification des protections
    si ⟨prot non respectées⟩ interruption violation de protec.

    -- vérification de la présence
    si (présent = 0) interruption défaut de page

    -- construction de l'adresse physique
    adresse_physique := ⟨npp, depl⟩
    renvoyer adresse_physique
```


► Un exemple sur deux processus ◄



► Algorithme du défaut de page ◄

Traitement de l'interruption défaut de page :

```
procédure defaut_de_page( v : numéro_de_page_virtuelle )
    ⟨suspendre le processus qui a provoqué le défaut⟩
    p := liberer_une_page_physique()
    ⟨charger la page virtuelle v dans la page physique p⟩   E/S
    desc[v].présent := 1
    desc[v].modif := 0
    desc[v].npp := p
    ⟨reprendre le processus qui a provoqué le défaut⟩
```

Algorithme de libération d'une page :

```
fonction liberer_une_page_physique( )
    | si ⟨il existe une page physique q libre⟩ alors
    |     | p := q
    | sinon
    |     | v := ⟨choisir un page virtuelle victime⟩
    |     | p := desc[v].npp
    |     | si (desc[v].modif = 1) alors
    |     |     | ⟨sauver la page physique p⟩                E/S
    |     | fin si
    |     | desc[v].présent := 0
    | fin si
    | renvoyer p
```

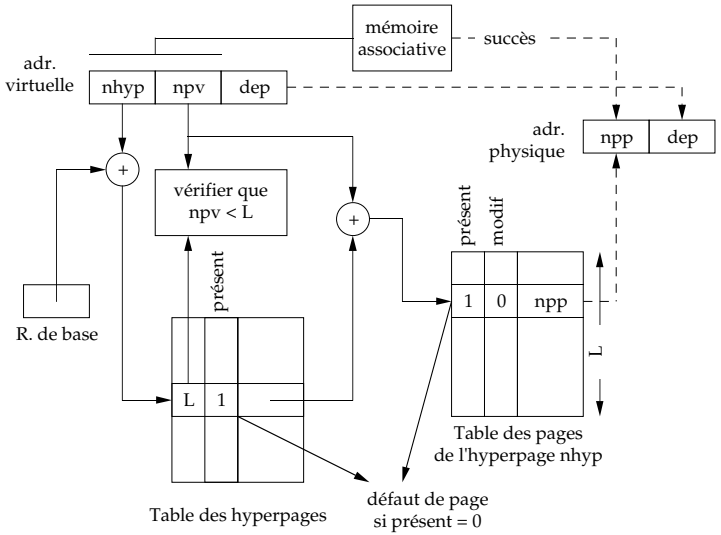

Principe : Si la mémoire est importante le nombre de pages augmente et la table des pages devient **imposante**.

Exemple : Une mémoire de 256 Mo (soit 2^{28} octets) est divisée en $2^{28}/2^{10} = 2^{18}$ pages. La table des pages a donc 2^{18} entrées soit **1 Mo**.

Conséquence : malgré la pagination, nous devons allouer des ensembles de pages **contigus** pour les tables de pages.

Solution : paginer la table des pages ce qui revient à faire une pagination à **deux niveaux**.

Organisation :



Il peut y avoir jusqu'à **5 niveaux** de pagination. Dans ce cas

$$\text{temps d'accès} = (0,98 \times 120) + (0,02 \times 520) = 128$$

La taille des pages doit être **grande** pour

- diminuer le **nombre de pages**, donc le nombre de défauts de page et la taille de la table des pages
- optimiser le **temps de transfert** vers ou depuis la zone de pagination
- utiliser des mémoires centrales de plus en plus **grandes**

La taille des pages doit être **petite** pour

- limiter la **fragmentation interne**
- définir avec **plus de précision** les pages utiles

Actuellement la taille des pages varie entre **1 ko** et **32 ko**.

Certains systèmes autorisent **plusieurs tailles** différentes.

▶▶ Algorithmes de remplacement ◀◀

Principe : On choisit en priorité les **pages virtuelles propres** (qui n'ont pas été modifiées).

Algorithmes :

- Algorithme **optimale** (base de référence) : choisir la page virtuelle qui est utilisée le **plus tard possible** ou qui n'est plus utilisée.
- Algorithme **aléatoire** (le moins bon).
- Algorithme **FIFO** (il ne tient pas compte de l'utilisation des pages).
- Algorithme **LRU** (*Least Recently Used*) est basé sur le **principe de localité** : choisir la page dont la date du dernier accès est la plus ancienne.

▶ Algorithme FINUFO (First In Not Used First Out) ◀

Principes :

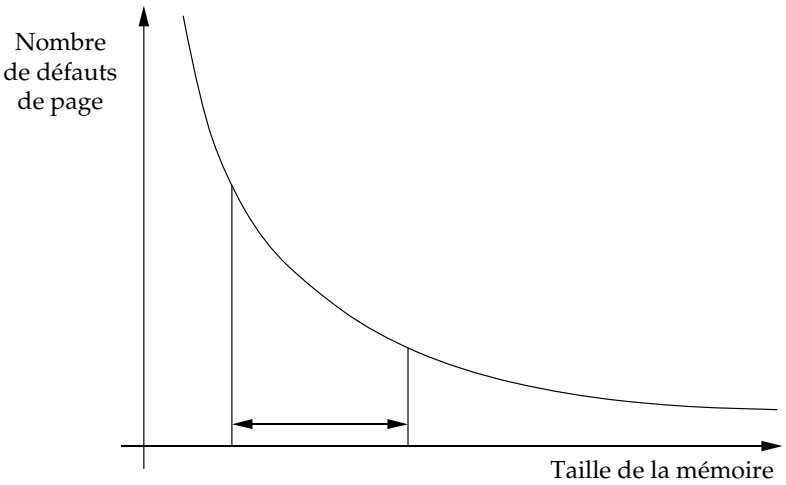
- Un **pointeur global** de page physique P
- Un **bit d'utilisation** par page physique noté `U[k]`
- `U[k]` est forcé à 1 après **chaque accès** à la page physique `k` (**PMMU**)

```
choisir_victime_FINUFO ()
    | tant que (U[P] = 1) faire
    |     | U[P] := 0
    |     | P := (P + 1) mod NB_PAGES_PHYSIQUES
    | fin faire
    | U[P] := 1
    | victime := P
    | P := (P + 1) mod NB_PAGES_PHYSIQUES
    | renvoyer victime
```

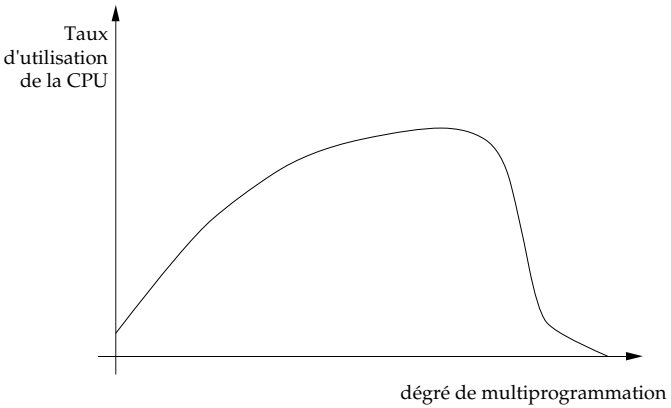
Performances :

OPT > LRU > LFU > FINUFO > FIFO > ALEA

▶ Comportement en mémoire virtuelle paginée ◀



▶ Écroulement d'un système paginé ◀



Faible taux de CPU → plus de processus → moins de mémoire → plus de défauts → baisse du Tx de CPU

Pour éviter l'écroulement on utilise

- la **régulation de charge** par variation du degré de multiprogrammation
- l'observation du **taux de défaut de pages**