

# Noyau d'un système d'exploitation

## INF2610

### Chapitre 7 : Gestion de la mémoire

Département de génie informatique et génie logiciel

POLYTECHNIQUE  
MONTRÉAL

AFFILIÉE À  
L'UNIVERSITÉ DE MONTRÉAL



Automne 2016

# Chapitre 7 - Gestion de la mémoire

- Généralités
- Comment organiser la mémoire physique ?
- Comment organiser l'espace d'adressage d'un processus ?
  - La pagination pure
  - Segmentation avec ou sans pagination
- Cas de Linux



# Généralités

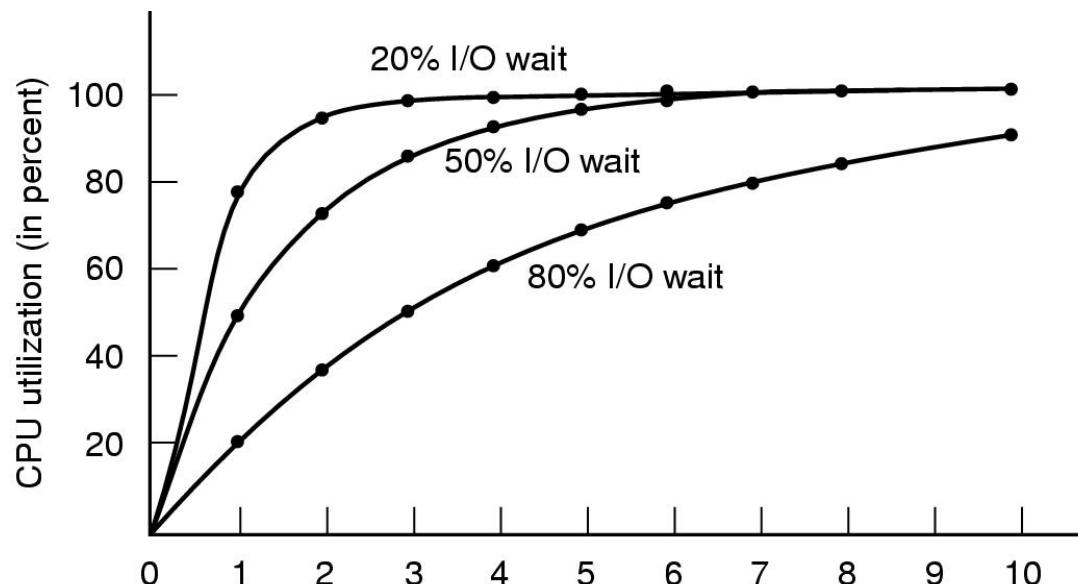
- Le gestionnaire de la mémoire est le composant du système d'exploitation qui se charge de gérer l'allocation d'espace mémoire au SE et aux processus :
  - Comment organiser la mémoire physique ?
  - Comment mémoriser l'état de la mémoire? Parmi les parties libres en mémoire, lesquelles allouer au processus? (politique de placement)
  - Faut-il allouer tout l'espace nécessaire avant l'exécution ou à la demande ? (politique d'allocation).
  - S'il n'y a pas assez d'espace en mémoire, doit-on libérer de l'espace en retirant des parties ou des processus entiers ? Si oui lesquels ? (politique de remplacement).
  - Les adresses figurant dans les instructions sont-elles relatives (logiques / virtuelles) ? Si oui, comment les convertir en adresses physiques.
  - Si plusieurs processus peuvent être résidents en mémoire, comment assurer la protection des processus (éviter qu'un processus soit altéré par un autre) ?



## Généralités (2)

### Exigences :

- Multiprogrammation : La mémoire physique doit être partagée entre le système d'exploitation et plusieurs processus. Le but est d'optimiser le taux d'utilisation du processeur.



Taux d'utilisation =  $1 - P^n$   
où P est le taux d'attente d'E/S et n est le nombre de processus.

## Généralités (3)

- Efficacité : La mémoire doit être allouée équitablement et à moindre coût tout en assurant une meilleure utilisation et partage des ressources (mémoire, processeurs et périphériques) entre les processus.
- Relocation : La possibilité de déplacer un processus en mémoire.
- Protection : Les processus ne peuvent pas se corrompre.

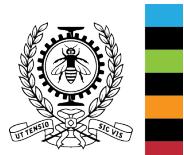
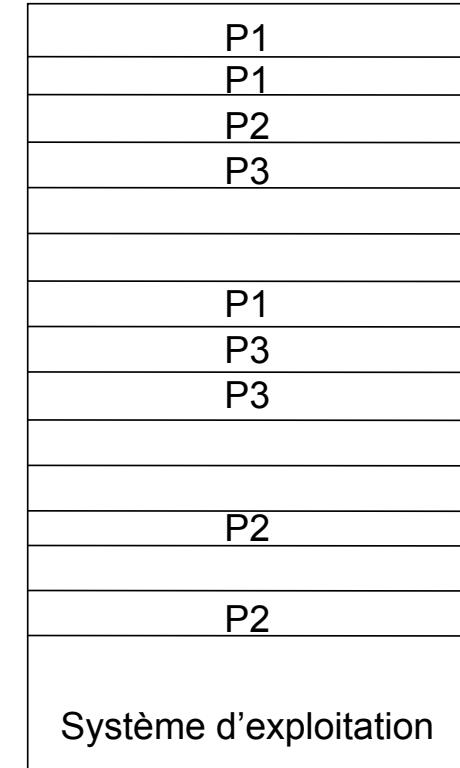


- Chaque processus a un espace d'adressage privé et accède à la mémoire physique via son espace d'adressage (adresses logiques / linéaires).
- Chaque adresse virtuelle référencée par le processus est analysée pour vérifier sa validité avant de la convertir en adresse physique (accéder à la mémoire physique).
- L'allocation d'espace physique (nécessaire à son exécution) à la demande (durant l'exécution avec ou sans pré-allocation). L'espace d'adressage d'un processus peut être donc beaucoup grand que celui de la mémoire physique (mémoire virtuelle).



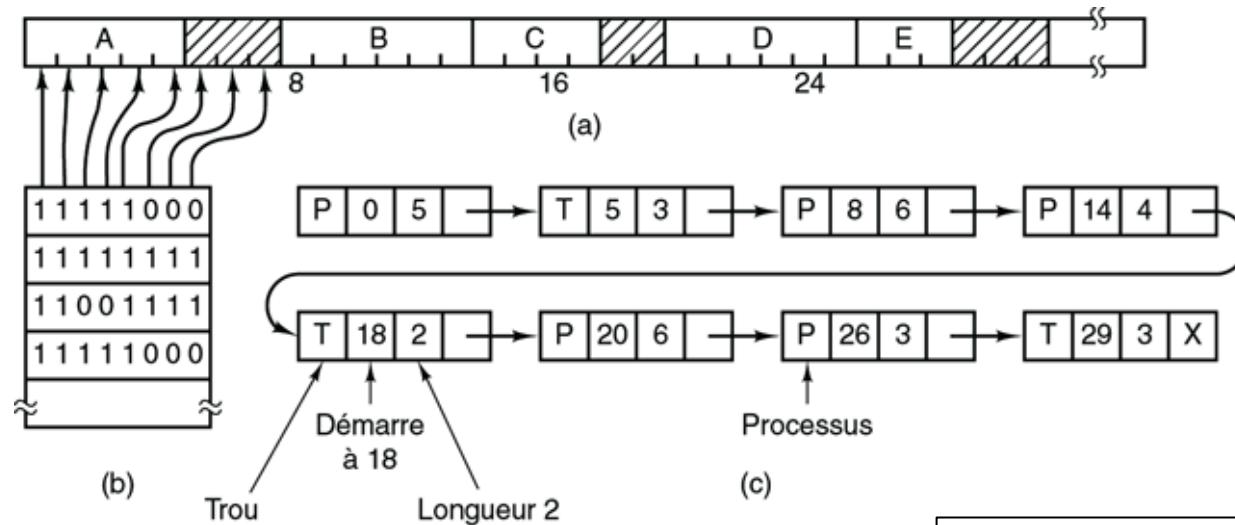
# Comment organiser la mémoire physique ?

- La mémoire physique est composée d'une zone réservée au système d'exploitation (résidente) et d'une autre zone composée de cadres (cases ou frames) de même taille (4Kio, 8Kio).
- L'unité d'allocation est le cadre avec possibilité d'allouer plusieurs cadres contigus (segment).
- Un processus peut occuper plusieurs cadres contigus ou non (segmentation ou pagination)
- Il n'est pas nécessaire de charger tout l'espace d'adressage d'un processus avant son exécution.
- Pour gérer l'allocation/libération de l'espace mémoire, le gestionnaire doit connaître l'état de la mémoire : tableaux de bits (1 bit par cadre) ou listes chaînées (triées par ordre croissant des adresses).

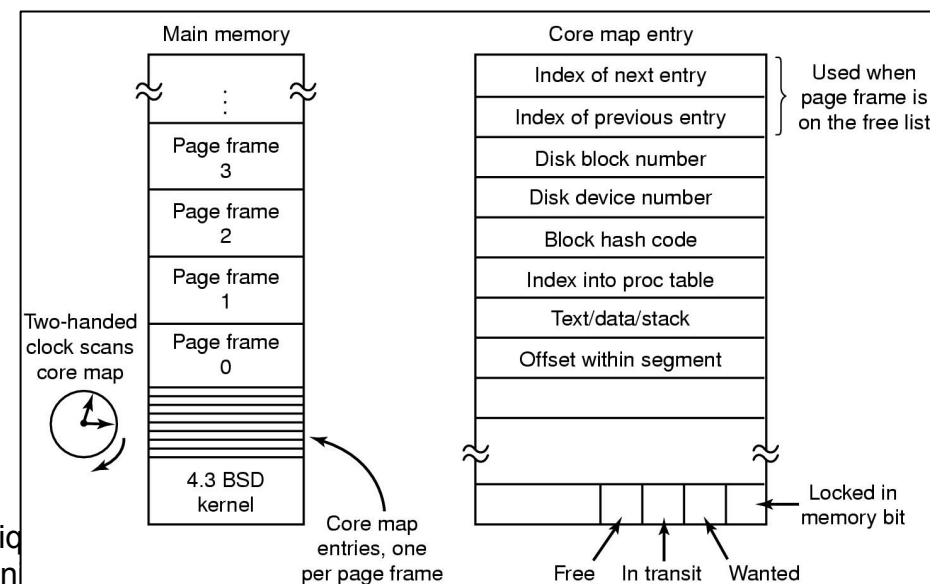


# Comment organiser la mémoire physique ? (2)

État de la mémoire :



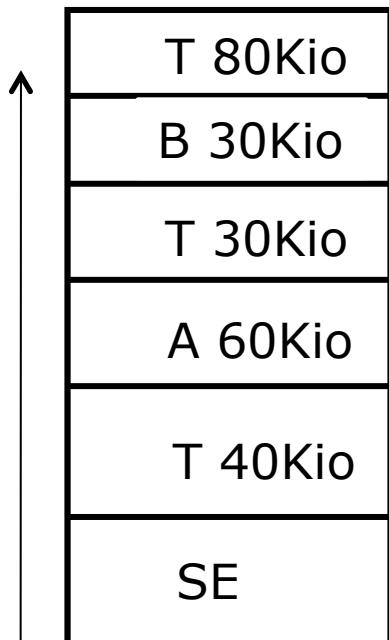
État de la mémoire cas de BSD 4 :  
 → Liste doublement chaînée de cadres libres.



# Comment organiser la mémoire physique ? (3)

## Politique de placement (allocation de zones contigües) :

- Premier ajustement (First-fit) : première zone libre qui convient (de taille suffisante).
- Meilleur ajustement (Best-fit) : plus petite zone libre qui convient.
- Pire ajustement (Worst-fit) : plus grande zone libre qui convient.
- Par subdivision (Linux).



- La zone allouée pour une demande d'allocation contiguë de 20 Kio (taille d'un cadre = 1 Kio) :  
First-fit: premiers 20 Kio de la zone de 40Kio.  
Best-fit: premiers 20 Kio de la zone de 30Kio.  
Worst-fit: premiers 20 Kio de la zone de 80Kio.
- La libération de la zone occupée par A devrait regrouper les zones T 30 Kio, A 60 Kio, T 40 Kio en une seule zone T 130Kio.

→ Fragmentation externe

# Comment organiser la mémoire physique ? (4)

## Par subdivision (cas de Linux) :

La mémoire centrale est gérée comme suit :

- Initialement, la mémoire est composée d'une seule zone libre.
- Lorsqu'une demande d'allocation arrive, la taille de l'espace demandé est arrondie à une puissance de 2. La zone libre initiale est divisée en deux. Si la première est trop grande, elle est, à son tour, divisée en deux et ainsi de suite... Sinon, elle est allouée au demandeur.
- Le gestionnaire de la mémoire utilise un tableau qui contient des têtes de listes. Le premier élément du tableau contient la tête de la liste des zones de taille 1. Le deuxième élément contient la tête de la liste des zones de taille 2...
- Lors de la libération de l'espace, les zones contiguës de même taille sont regroupées en une seule zone.

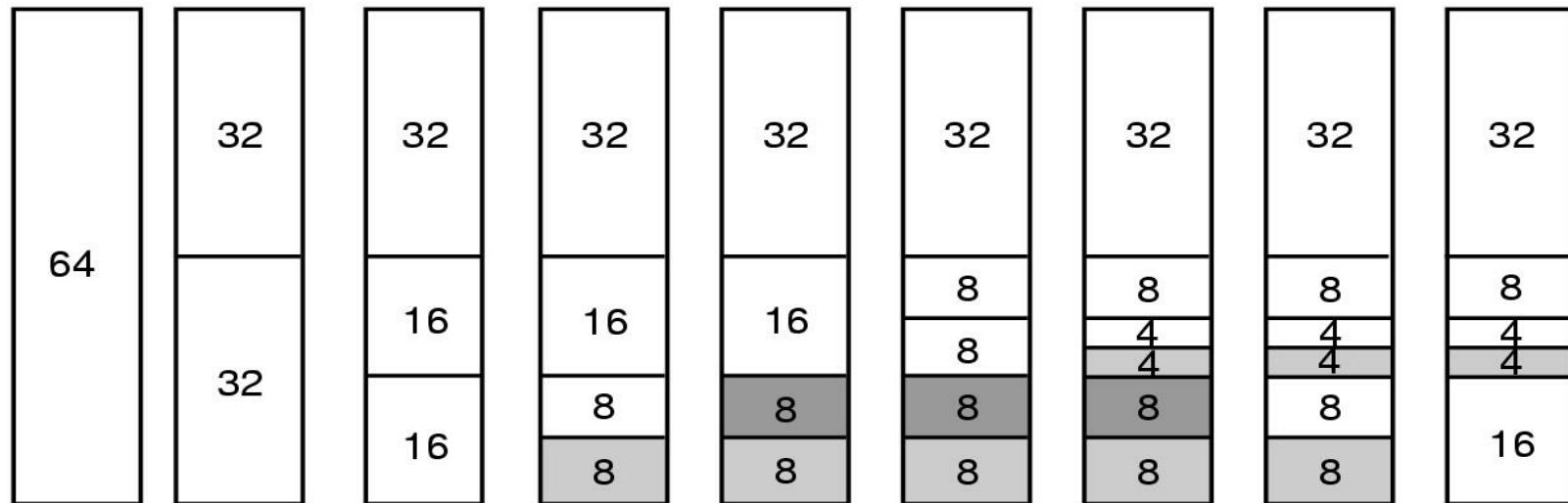
→ Allocation très rapide.

→ **Fragmentation interne** (espaces alloués mais non utilisés).



# Comment organiser la mémoire physique ? (5)

Par subdivision (cas de Linux) :



Demande de 6 pages → 8 pages

Demande de 5 pages → 8 pages

Demande de 3 pages → 4 pages

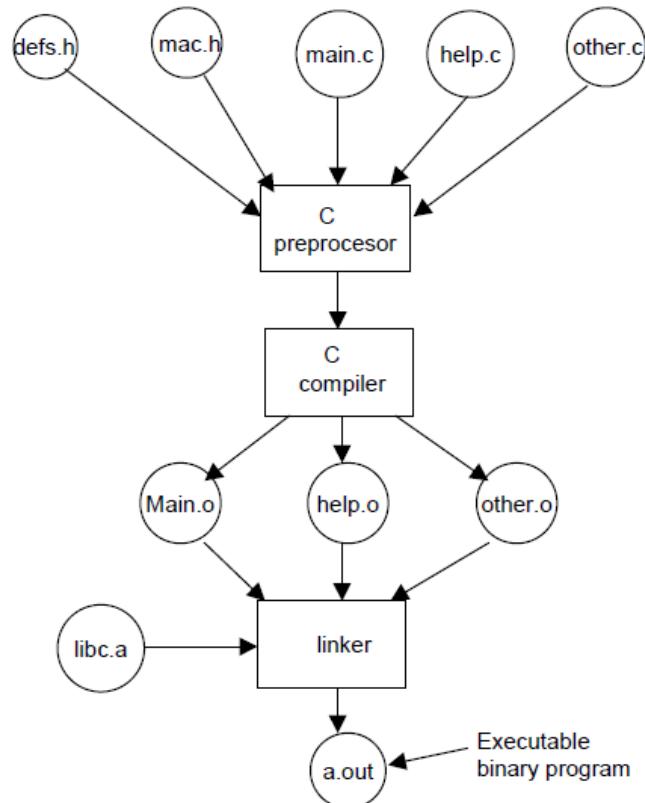
Libération de 8 pages + Libération de 8 pages → 16 pages



Tableau de têtes de listes : L'entrée  $i$  est un pointeur vers la liste de blocs libres de taille  $2^i$ .

# Comment gérer l'espace d'adressage d'un processus ?

Compilation et édition de liens



ELF (Executable and Linkable Format)

File Offset	File	Virtual Address
0	ELF header	
0x100	Other information	0x8048100
0x2bf00	Text segment ... 0x2be00 bytes	0x8073eff
0x30d00	Data segment ... 0x4e00 bytes	0x8074f00
	Other information ...	0x8079cff



# Comment gérer l'espace d'adressage d'un processus ?

ELF (Executable and Linkable Format)

File Offset	File	Virtual Address
0	ELF header	0x8048000
	Other information	0x8048100
0x100	Text segment	0x8048100
	...	
	0x2be00 bytes	0x8073eff
0x2bf00	Data segment	0x8074f00
	...	
	0x4e00 bytes	0x8079cff
0x30d00	Other information	
	...	

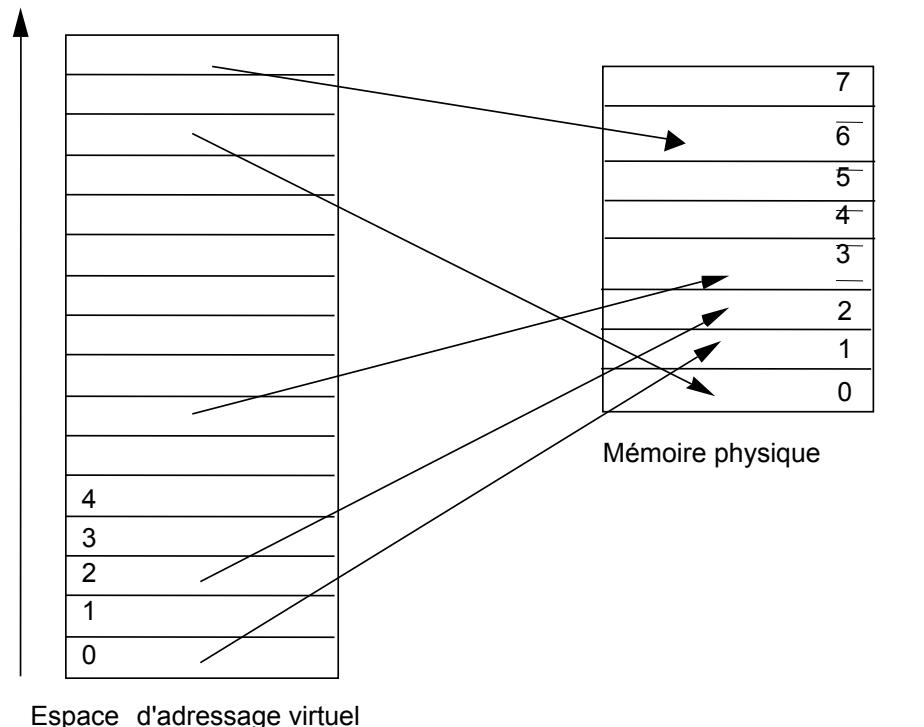
Espace d'adressage virtuel

Virtual Address	Contents	Segment
0x8048000	<i>Header padding</i> 0x100 bytes	Text
0x8048100	Text segment	
	...	
0x8073eff	0x2be00 bytes	
0x8074f00	<i>Data padding</i> 0x100 bytes	
0x8074000	<i>Text padding</i> 0xf00 bytes	Data
0x8074f00	Data segment	
	...	
0x8079cff	0x4e00 bytes	
0x8079d00	Uninitialized data 0x1024 zero bytes	
0x807ad24	<i>Page padding</i> 0x2dc zero bytes	



# Comment gérer l'espace d'adressage d'un processus ? La pagination pure

- Espace d'adressage d'un processus = ensemble de pages de même taille.



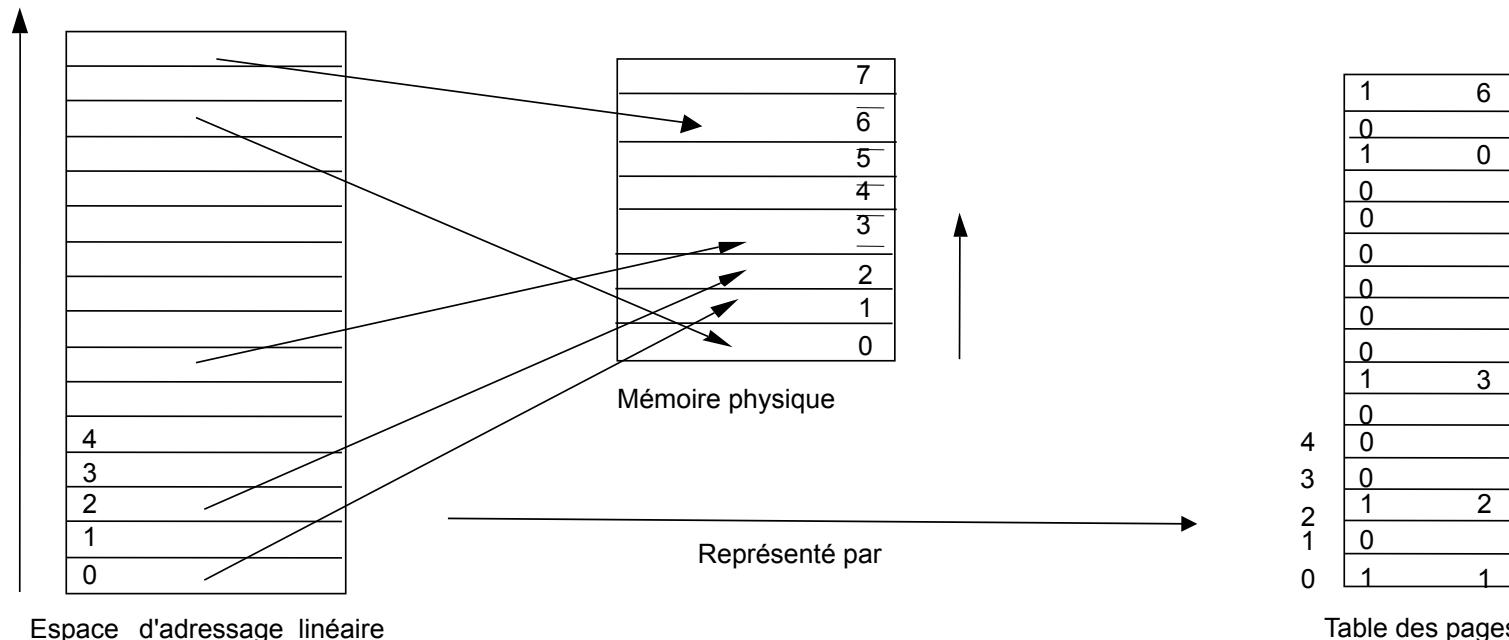
- Espace d'adressage linéaire.
- Unité d'allocation : cadre ( taille d'un cadre = taille d'une page).
- Chargement à la demande.
- Fragmentation interne** (espace alloué mais non utilisé) pour les cadres non pleins.



## La pagination pure (2)

**Exemple :** Soit un programme de 64 Kio sur une machine 32 Kio de mémoire physique. La taille d'une page (case) est de 4 Kio.

- L'espace d'adressage virtuel d'un processus est composé de 16 pages de 4 Kio.
- La mémoire physique est composée de 8 cases (cadres) de 4 Kio.



L'adresse de la table des pages fait partie du contexte d'exécution du processus.

## La pagination pure (3)

### Conversion d'adresses virtuelles

- Adresse virtuelle = (numéro de page, déplacement dans la page).
- Les adresses virtuelles référencées par l'instruction en cours d'exécution doivent être converties en adresses physiques.
- La correspondance entre les pages et les cases est mémorisée dans la table de pages. Le nombre d'entrées dans la table est égal au nombre de pages virtuelles.
- La table des pages d'un processus doit être (en totalité ou en partie) en mémoire centrale lors de l'exécution du processus. Elle est nécessaire pour la conversion des adresses virtuelles en adresses physiques.
- Chaque entrée de la table des pages est composée de plusieurs champs, notamment :
  - Le bit de présence
  - Le bit de référence (R)
  - Les bits de protection (un, deux ou trois bits)
  - Le bit de modification (M appelé bit dirty/clean)
  - Le numéro de case correspondant à la page
  - son emplacement sur disque



# **La pagination pure (4)**

## **Conversion d'adresses virtuelles**

**Exemple :**

- Supposons que l'adresse virtuelle est sur 16 bits :  
(numéro de page (4 bits), déplacement dans la page (12 bits)).
- La conversion est réalisée en examinant l'entrée dans la table des pages correspondant au numéro de page.
- Si le bit de présence est à 0, la page n'est pas en mémoire, il faut alors lancer son chargement en mémoire.
- Sinon, on détermine l'adresse physique sur 15 bits en recopiant dans :
  - les 3 bits de poids le plus fort, le numéro de case (110) correspondant au numéro de page (0010) et
  - les 12 bits de poids le plus faible, les 12 bits de poids le plus faible de l'adresse virtuelle.

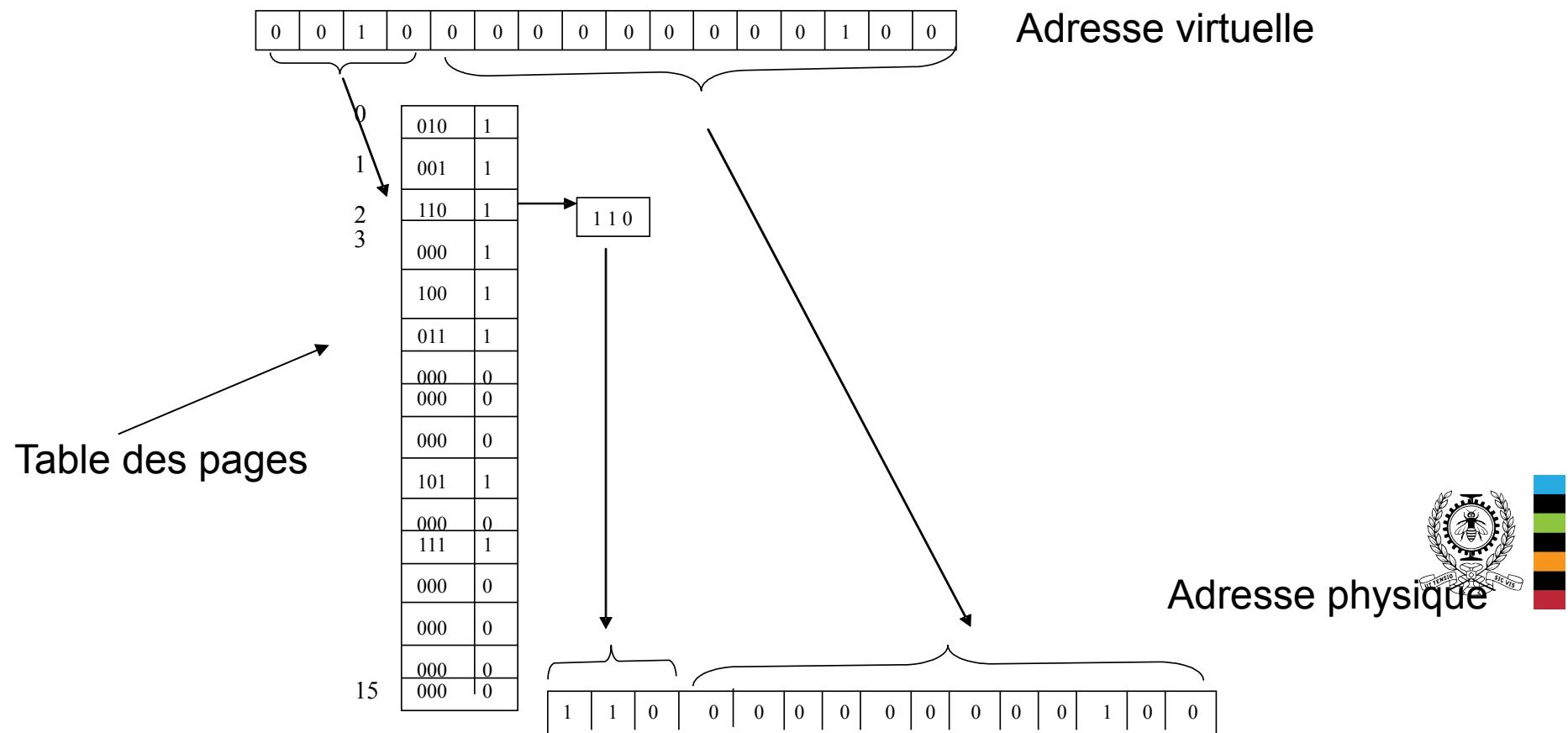


# La pagination pure (5)

## Conversion d'adresses virtuelles

**Exemple (suite) :**

- L'adresse virtuelle 8196 (0010 0000 0000 0100) est convertie en adresse physique 24580 (110 0000 000 0100).



# La pagination pure (6)

## Conversion d'adresses virtuelles

### Exercice 1

- Dans un système de mémoire virtuelle, la taille des pages virtuelles et physiques est 1024 octets et l'adressage virtuel est sur 16 bits. Un processus P1 utilise les 9 premières pages de son espace virtuel. Les champs P (bit de présence) et Cadre (numéro de cadre physique) de la table des pages de l'espace virtuel de P1 sont :

	P	Cadre
0	1	2
1	1	1
2	1	5
3	0	-
4	0	-
5	1	0
6	0	-
7	0	-
8	0	-

- Quelle est la taille maximale (en nombre de pages) de l'espace virtuel d'un processus ?
- Quelle est l'adresse physique correspondant à chacune des adresses virtuelles de P1 suivantes codées en hexadécimal : 0x0A2A et 0x01F1 ?



# La pagination pure (7)

## Conversion d'adresses virtuelles

### MMU

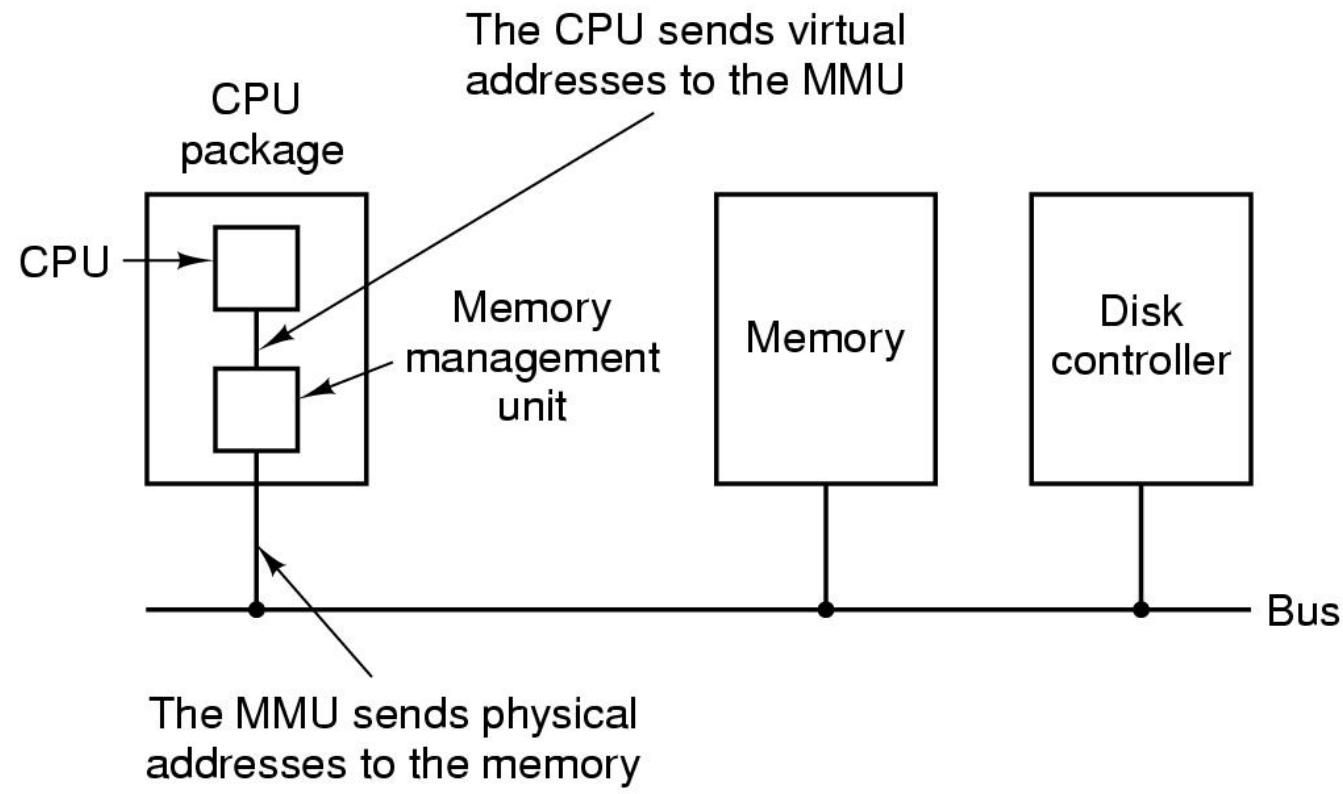
- Cette conversion d'adresse est effectuée par un composant matériel du processeur le MMU : *Memory Management Unit* (MMU).
- Le MMU vérifie si l'adresse virtuelle reçue correspond à une adresse en mémoire physique (en consultant la table des pages).
- Si c'est le cas, le MMU transmet sur le bus de la mémoire l'adresse réelle, sinon il y a un **défaut de page**.
- Un défaut de page provoque un déroutement (trap) dont le rôle est de ramener à partir du disque la page manquante référencée (l'unité de transfert est la page).



# La pagination pure (8)

## Conversion d'adresses virtuelles

### MMU



# La pagination pure (9)

## Conversion d'adresses virtuelles

### MMU avec mémoire associative TLB

- Pour accélérer la translation d' adresse, le MMU est doté d' un composant, appelé mémoire associative, composé d' un petit nombre d' entrées (8 à 128).
- Ce composant appelé aussi TLB (Translation Lookaside Buffers) contient des informations sur les dernières pages référencées. Chaque entrée est composée de :
  - Un bit de validité
  - Un numéro de page virtuelle
  - Un bit de modification (M)
  - Un, deux ou trois bits de protection
  - Un numéro de case

1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75



# La pagination pure (10)

## Conversion d'adresses virtuelles

### MMU avec mémoire associative TLB

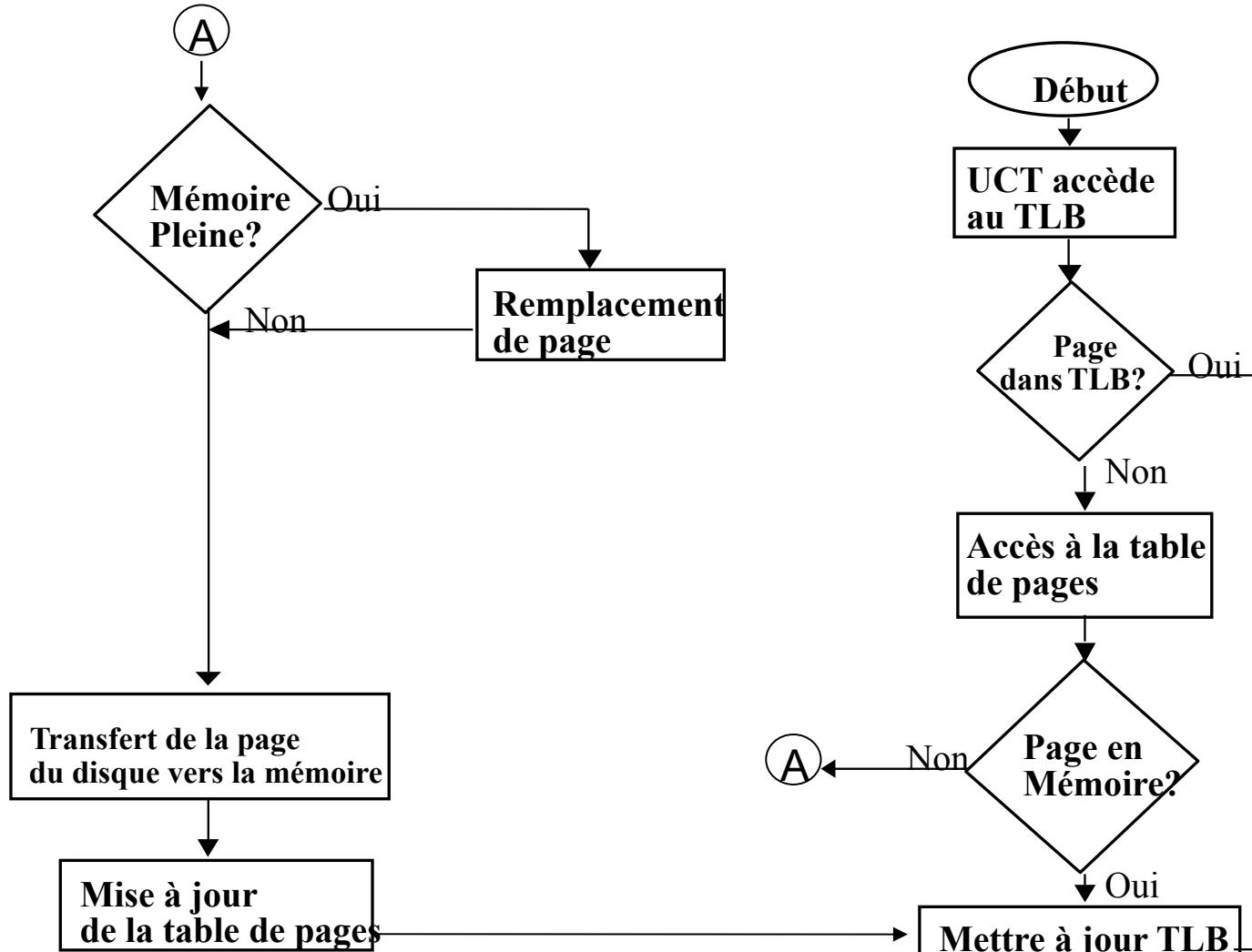
- Lorsqu'une adresse virtuelle est présentée au MMU, il vérifie d'abord si le numéro de la page virtuelle est présent dans la mémoire associative (en le comparant simultanément à toutes les entrées).
- S'il le trouve et le mode d'accès est conforme aux bits de protection, le numéro de case est pris directement de la mémoire associative (sans passer par la table des pages).
- Si le numéro de page est présent dans la mémoire associative mais le mode d'accès est non conforme, il se produit un défaut de protection.
- Si le numéro de page n'est pas dans la mémoire associative, le MMU accède à la table des pages à l'entrée correspondant au numéro de page.
- Si le bit de présence de l'entrée trouvée est à 1, le MMU remplace une des entrées de la mémoire associative par l'entrée trouvée. Sinon, il provoque un défaut de page.



# La pagination pure (11)

## Conversion d'adresses virtuelles

### MMU avec mémoire associative TLB



# **La pagination pure (12)**

## **Conversion d'adresses virtuelles**

### **MMU avec mémoire associative TLB**

- Supposons que les temps d'accès à la table des pages et à la mémoire associative sont respectivement 100ns et 20ns.
- Si la fraction de références mémoire trouvées dans la mémoire associative (taux d'impact) est  $s$ , le temps d'accès moyen est :  
$$s * 20 + (1-s) * 100.$$



# **La pagination pure (13)**

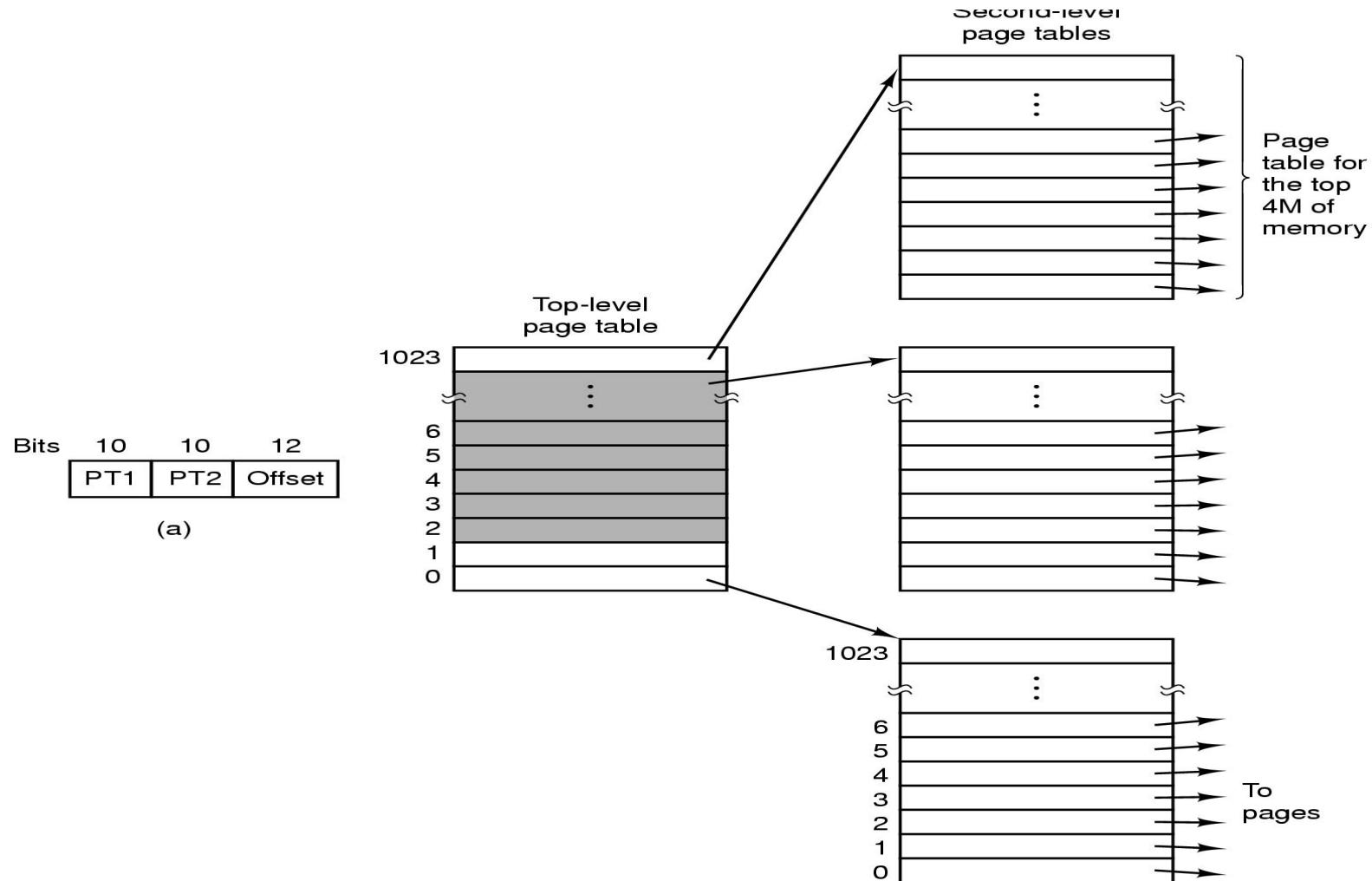
## **Table des pages à plusieurs niveaux**

- La taille de la table des pages peut être très grande:  $2^{20}$  entrées (plus d'un million) pour un adressage virtuel de 32 bits et des pages de 4 Kio.
- Pour éviter d'avoir des tables trop grandes en mémoire, de nombreux ordinateurs utilisent des tables à plusieurs niveaux.
- Par exemple, une table des pages à deux niveaux, pour un adressage sur 32 bits et des pages de 4 Kio, est composée 1025 tables de 1024 entrées. Il est ainsi possible de ne charger que les tables nécessaires.
- Dans ce cas, une adresse virtuelle de 32 bits est composée de trois champs :
  - un pointeur sur la table du 1er niveau (10 bits),
  - un pointeur sur une table du 2nd niveau (10 bits) et
  - un déplacement dans la page (12 bits).



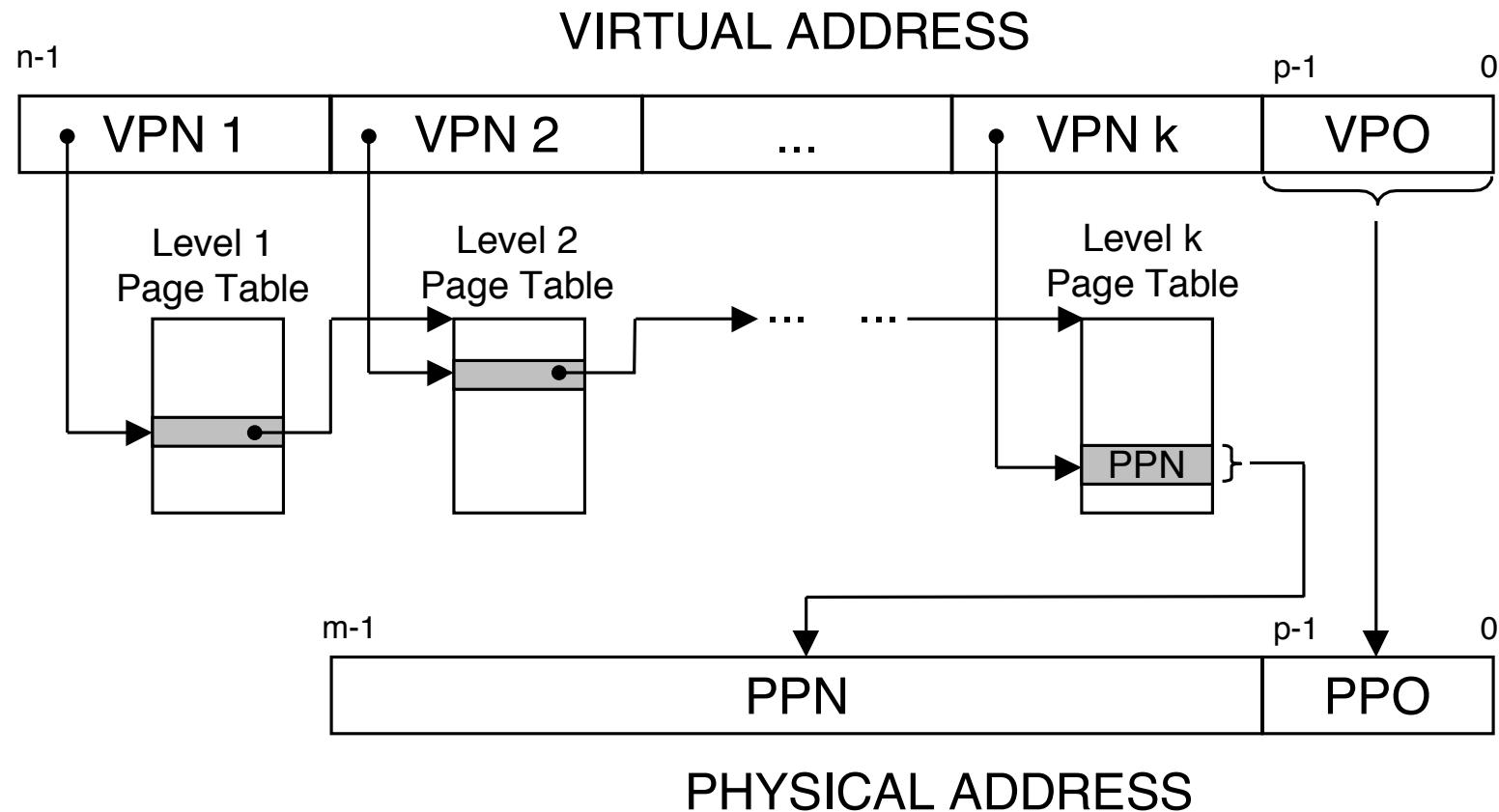
# La pagination pure (14)

## Table des pages à deux niveaux



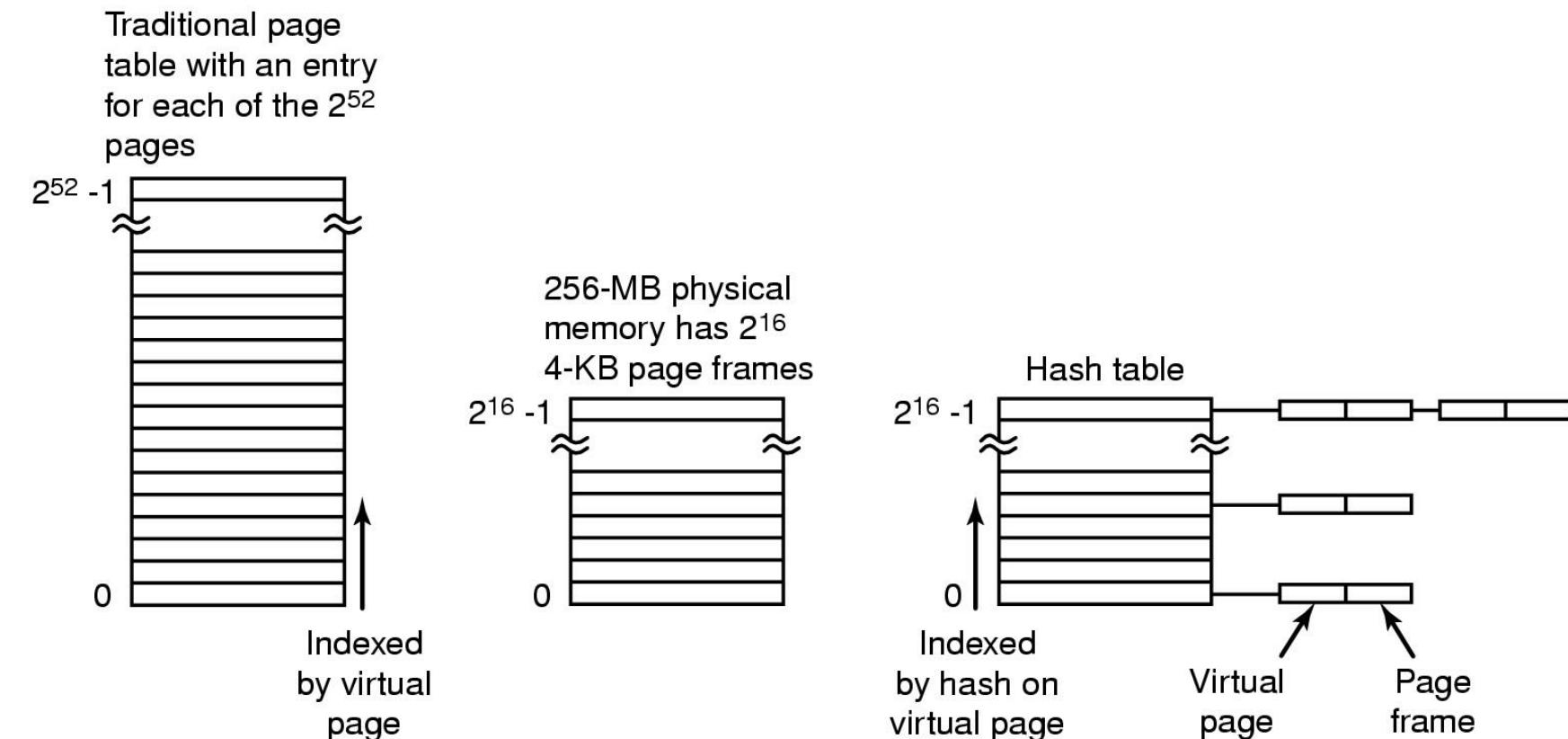
# La pagination pure (15)

## Table des pages à k niveaux



# La pagination pure (16)

## Table des pages inversée



# **La pagination pure (17)**

## **Algorithmes de remplacement de page**

- A la suite d'une faute de page (lourde), le système d'exploitation doit ramener en mémoire la page manquante à partir du disque.
- S'il n'y a pas de cases libres en mémoire, il doit retirer une page de la mémoire pour la remplacer par celle demandée.
- Si la page à retirer a été modifiée depuis son chargement en mémoire, il faut la réécrire sur le disque.
- Quelle est la page à retirer de manière à minimiser le nombre de défauts de page ?
  - Le choix de la page à retirer peut se limiter aux pages du processus qui a provoqué le défaut de page (allocation locale) ou à l'ensemble des pages en mémoire (allocation globale).
  - En général, l'allocation globale produit de meilleurs résultats que l'allocation locale.
- Ces algorithmes mémorisent les références passées aux pages. Le choix de la page à retirer dépend des références passées.



# La pagination pure (18)

## Algorithme optimal (BELADY)

- Critère : remplacer la page qui sera référencée le plus tard possible dans le futur.
- Non réalisable.
- Version locale et globale.
- Intérêt pour faire des études analytiques comparatives.

### Exemple 1 : avec 3 cadres

Nombre d'accès: 20  
Fautes de page: 9 (45%)

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



# La pagination pure (19)

## Algorithme FIFO

- Critère : la page dont le temps de résidence est le plus long
- Implémentation facile : pages résidentes en ordre FIFO (on expulse la première)  
→ file circulaire
- Ce n'est pas une bonne stratégie : critère non fondé sur l'utilisation de la page
- Anomalie de Belady : en augmentant le nombre de cadres on augmente le nombre de défauts de page au lieu de le diminuer.

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

Nombre d'accès: 20

Fautes de page: 15 (75%)



# La pagination pure (20)

## Algorithme FIFO

### Anomalie de Belady

#### Exemple 2 avec 3 cadres

Nombre d'accès: 20

Fautes de page: 15 (75%)

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

#### Exemple 2 avec 4 cadres

Nombre d'accès: 20

Fautes de page: 16 (80%)

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



# La pagination pure (21)

## Algorithme LRU

- Critère : page résidente la moins récemment utilisée.
- Basé sur le principe de localité : une page a tendance à être réutilisée dans un futur proche.
- Difficile à implémenter sans support matériel.

Nombre d'accès: 20  
Fautes de page: 12 (60%)

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



# La pagination pure (22)

## Algorithme LRU

### Comment l'implémenter ?

- Mémoriser pour chaque page en mémoire la date de la dernière référence.
- Vieillissement (aging) :
  - Un registre de  $n$  bits est associé à chaque page,
  - Le bit le plus significatif est mis à 1 à chaque référence
  - Régulièrement, décaler vers la droite les bits de ce registre,
  - choisir la page dont la valeur est la plus petite
- Utilisation d'une pile :
  - Ajouter ou déplacer, en sommet de pile, le numéro de la page référencée,
  - Remplacer la page située au fond de la pile.



# **La pagination pure (23)**

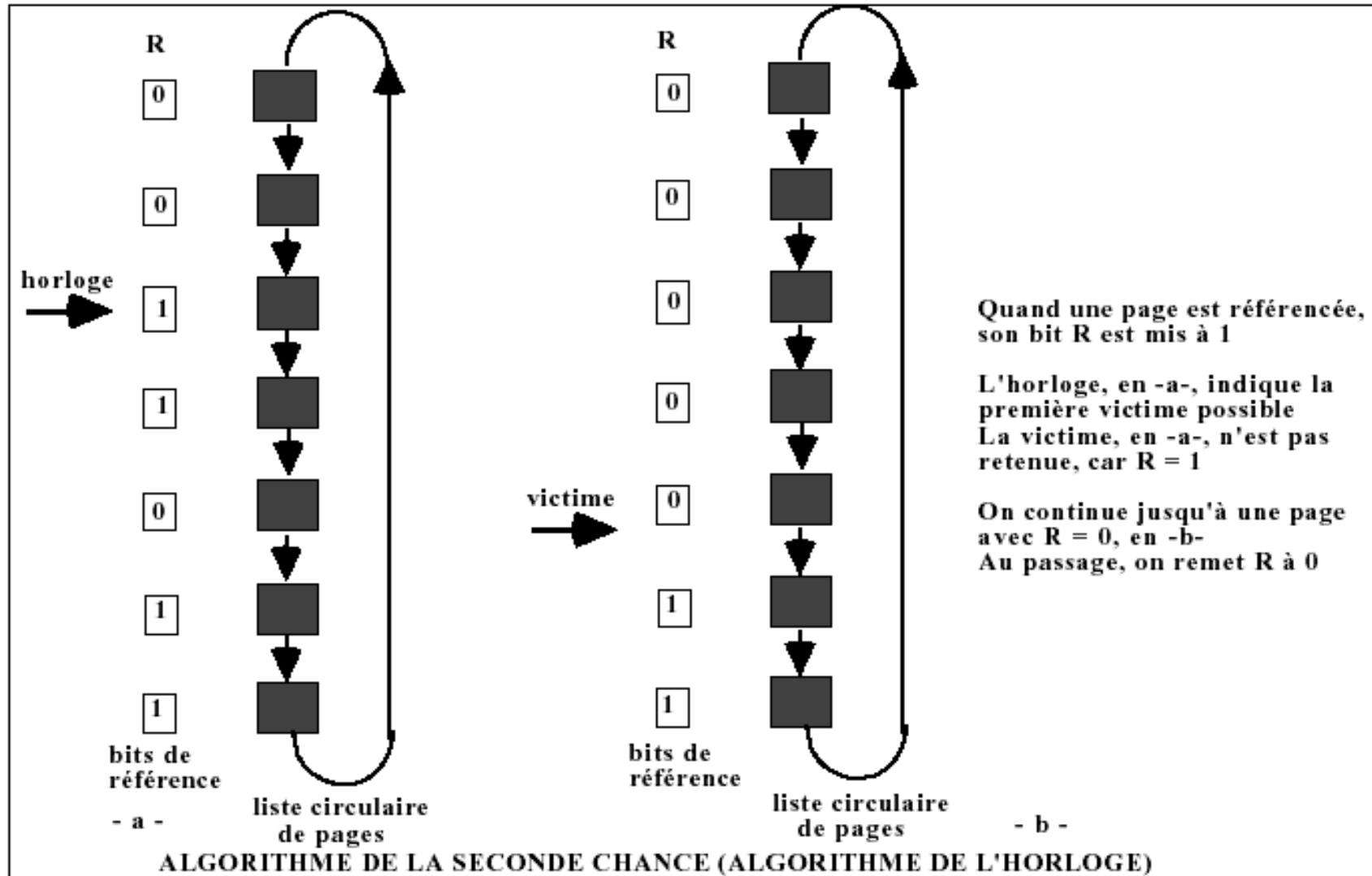
## **Algorithme de l'horloge (seconde chance)**

- Approximation de LRU
- Les pages en mémoire sont mémorisées dans une liste circulaire en forme d'horloge.
- On a un indicateur sur la page la plus ancienne et un bit de référence R est associé à chaque page.
- Lorsqu'un défaut de page se produit, les pages sont examinées, une par une, en commençant par celle pointée par l'indicateur.
- La première page rencontrée ayant son bit de référence R à 0 est remplacée. Le bit R de la page ajoutée est à 1.
- Le bit R des pages examinées est remis à 0.
- Une variante de cet algorithme, tient compte du bit de modification M.



# La pagination pure (24)

## Algorithme de l'horloge (seconde chance)



# La pagination pure (25)

## Remplacement de pages

« Bufferisation » de pages (Page buffering) : VAX VMS et Mach

- Pour améliorer les performances, lorsqu'une page doit être retirée de la mémoire, son descripteur ou numéro est insérée, selon son état à la queue de la liste de pages libres ou la liste de pages modifiées. La page n'est pas effacée physiquement de la mémoire.
  - Lorsqu'une page doit être chargée en mémoire, la victime est la page en tête de liste de pages libres (ou, à défaut, celle de la liste des pages modifiées). Si la victime a été modifiée, elle est sauvegardée sur le disque avant le chargement.
  - Si la page retirée (mais non encore effacée) est référencée, elle est retirée de la liste des pages libres (ou modifiées). Elle devient résidente.
- Les listes de pages libres et modifiées agissent comme un cache de pages.



# La pagination pure (26)

## Comment répartir la mémoire entre les processus ?

- Allocation fixe : Allouer un même nombre de cadres (cases) à chaque processus. Par exemple, si la mémoire totale fait 100 cadres et qu'il y a 5 processus, chaque processus recevra 20 cadres  
→ Allocation locale.
- Allocation variable : Allouer les cases proportionnellement aux tailles des programmes ou en tenant compte des priorités, des défauts de pages, etc.  
→ Allocation locale ou allocation globale.
- L'allocation de cases avant l'exécution ou à la demande, au cours de l'exécution (pre-pagination ou pagination à la demande).

Espace de travail (working set) (allocation variable mais locale) :

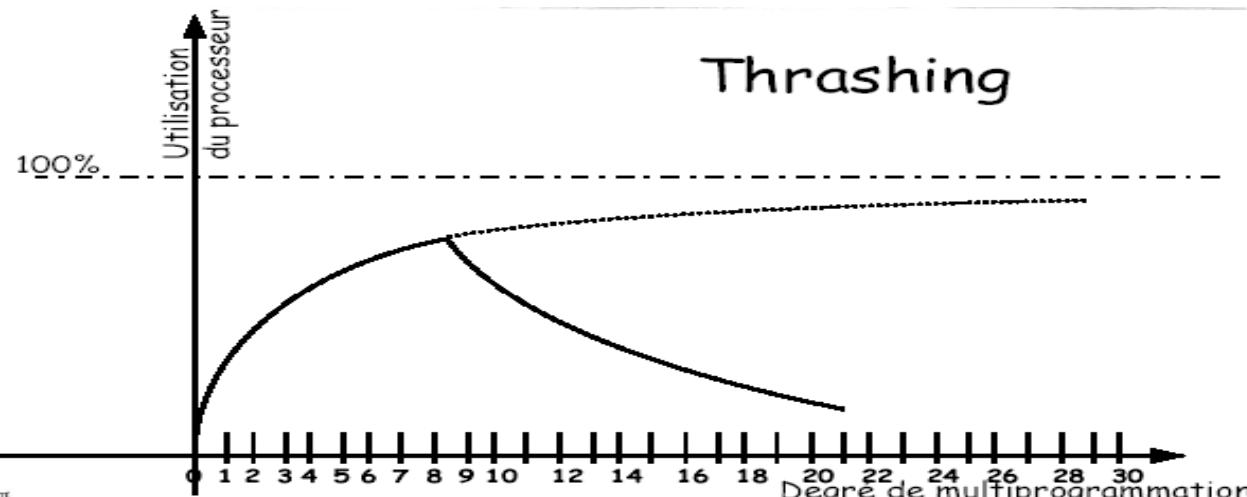
- $W(t, \Delta)$  est l'ensemble des pages qui ont fait l'objet d'au moins une référence entre le temps  $t - \Delta$  et  $t$ .
- On conserve en mémoire les pages référencées entre  $t - \Delta$  et  $t$ .
- Cet espace de travail ne doit pas excéder une certaine limite.



# La pagination pure (27)

## Problème de l'écroulement du système

- Le système passe plus de temps à traiter les défauts de page qu'à exécuter des processus ← le nombre de processus est trop grand et l'espace propre à chacun est insuffisant.
- On peut limiter le risque d'écroulement en surveillant le nombre de défauts de page provoqués par chaque processus :
  - Allouer des cadres supplémentaires aux processus qui génèrent trop de défauts de pages (au dessus d'une limite supérieure).
  - Retirer des cadres aux processus qui génèrent très peu de défauts de pages (au dessous d'une limite inférieure).
  - Suspendre un ou plusieurs processus.



Noyau d'un système d'exploitation



# **La pagination pure (28)**

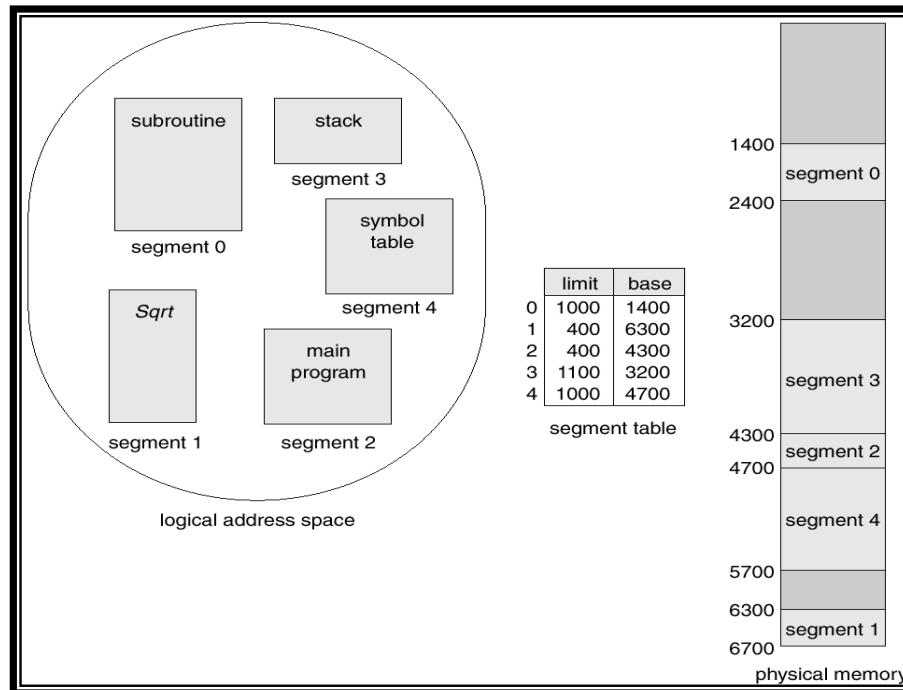
## **Retour sur instruction**

- Si un défaut de page se produit au milieu d'une instruction, le processeur doit revenir au début de l'instruction initiale, avant de lancer le chargement de la page manquante en mémoire.
- Ce retour sur instruction n'est possible qu'avec l'aide du matériel.

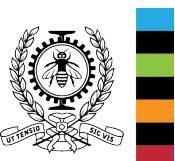


# Segmentation pure

- Espace d'adressage d'un processus = **ensemble de segments**  
Segments = **zones contiguës de tailles différentes**



L'adresse de la table des segments fait partie du contexte d'exécution du processus.



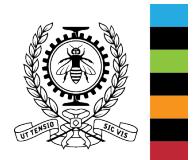
- Adresse logique = (numéro de segment, déplacement dans le segment).
- L'adresse physique de l'adresse logique (3,200) est  $3200+200=3400$ .
- Problème de **fragmentation externe** (espaces non alloués et non allouables (trop petits)) → nécessité de compactage (opération très coûteuse).
- Noyau d'un système d'exploitation

# Segmentation avec pagination

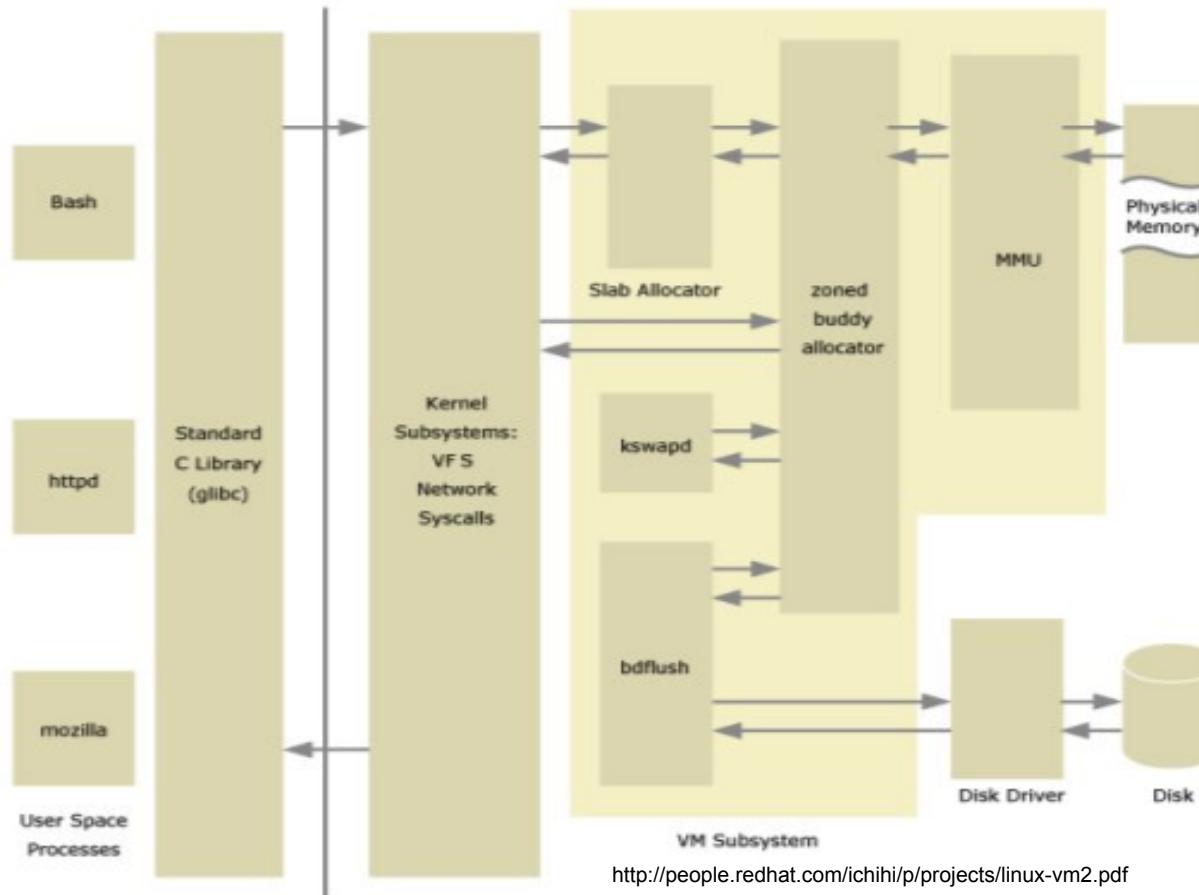
- Espace d'adressage d'un processus = ensemble de segments.
- Segment = ensemble de pages.
- On dispose dans ce cas **d'une table de segments et d'une table de pages par segment (MULTICS)**.
- Adresse logique :  
(numéro de segment, numéro de page, déplacement dans la page).
- **Fragmentation externe et interne.**



# Cas de Linux



# Cas de Linux



- kswapd (démon de pages) qui gère la zone de swap
- bdflush (démon, buffer-dirty-flush) qui gère la copie des blocs de fichiers modifiés --> pdflush
- Allocateurs Slab et par division

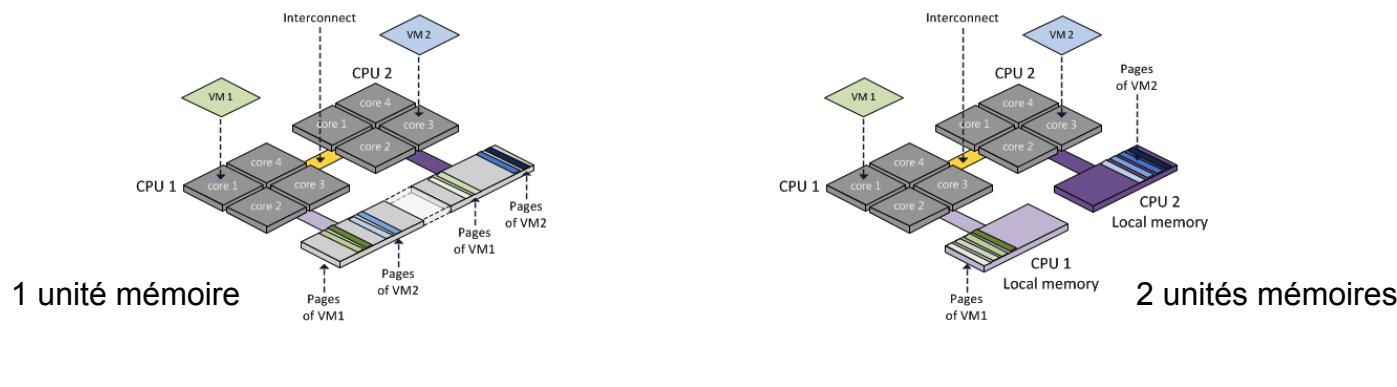


- Un système de pagination à la demande, sans pré-pagination ni concept d'ensemble de travail, allocation globale.

# Cas de Linux (2)

## Mémoire physique

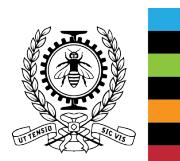
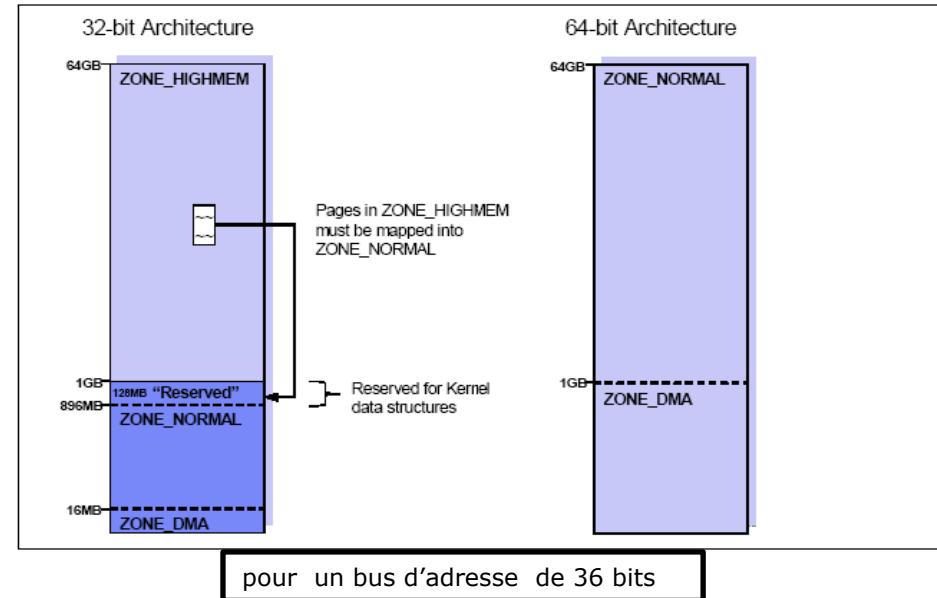
- Linux supporte des architectures NUMA (Non Uniform Memory Access) qui utilisent des unités mémoires hétérogènes avec des caractéristiques différentes.
- Linux offre la possibilité de gérer ces unités mémoires en associant à chaque unité un nœud (de type `struct_pglist_data`). Dans une architecture UMA, un seul nœud est présent.



# Cas de Linux (3)

## Mémoire physique

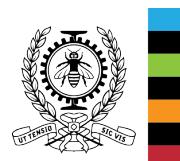
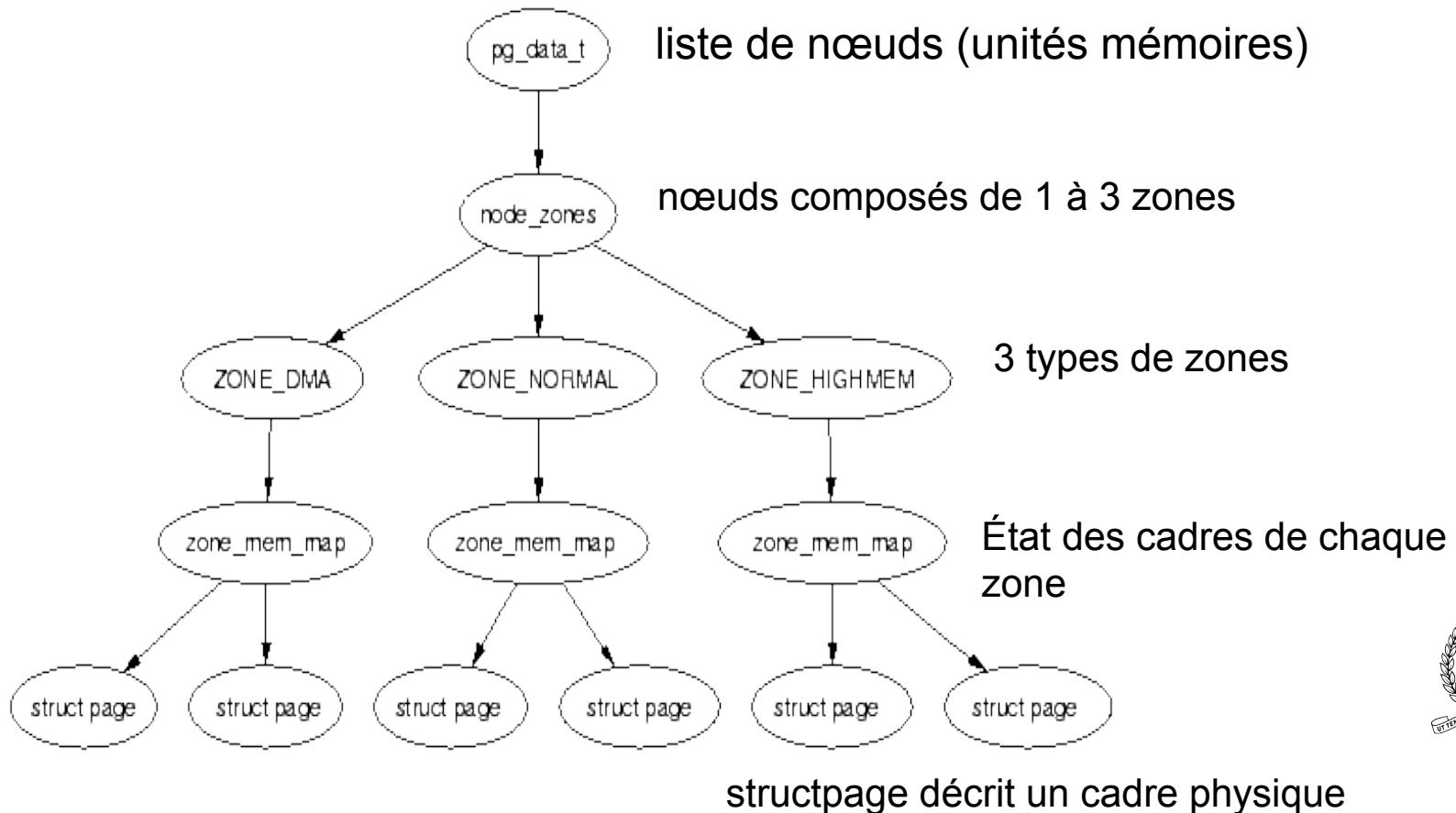
- Pour certaines architectures (ex. x86), il n'est pas possible de traiter toute la mémoire physique (unité mémoire) de la même manière.
- Chaque nœud (unité mémoire) est composé de trois zones mémoires :
  - ZONE\_DMA (16 Mio sur x86) réservée pour l'accès DMA des périphériques (qui ne peuvent accéder qu'aux 16 premiers Mio de la mémoire physique),
  - ZONE\_NORMAL (à mappage direct dans l'espace noyau) et
  - ZONE\_HIGHMEM utilisée si la taille des deux zones précédentes est supérieure à 896 Mo sur x86 (adresses non mappées de façon permanente).
- ZONE\_DMA et HIGHMEM peuvent être vides.



# Cas de Linux (4)

## Mémoire physique

### Structures de données associées



# Cas de Linux (5)

## Mémoire physique

### Structures de données associées

- L'état de la mémoire :
  - Tableau Mem\_map[] qui a autant d'entrées qu'il y a de cadres physiques. Les éléments sont de type structpage.
  - La structure structpage décrit une page physique (un cadre) <http://www.tldp.org/LDP/tlk/tlk.html>.

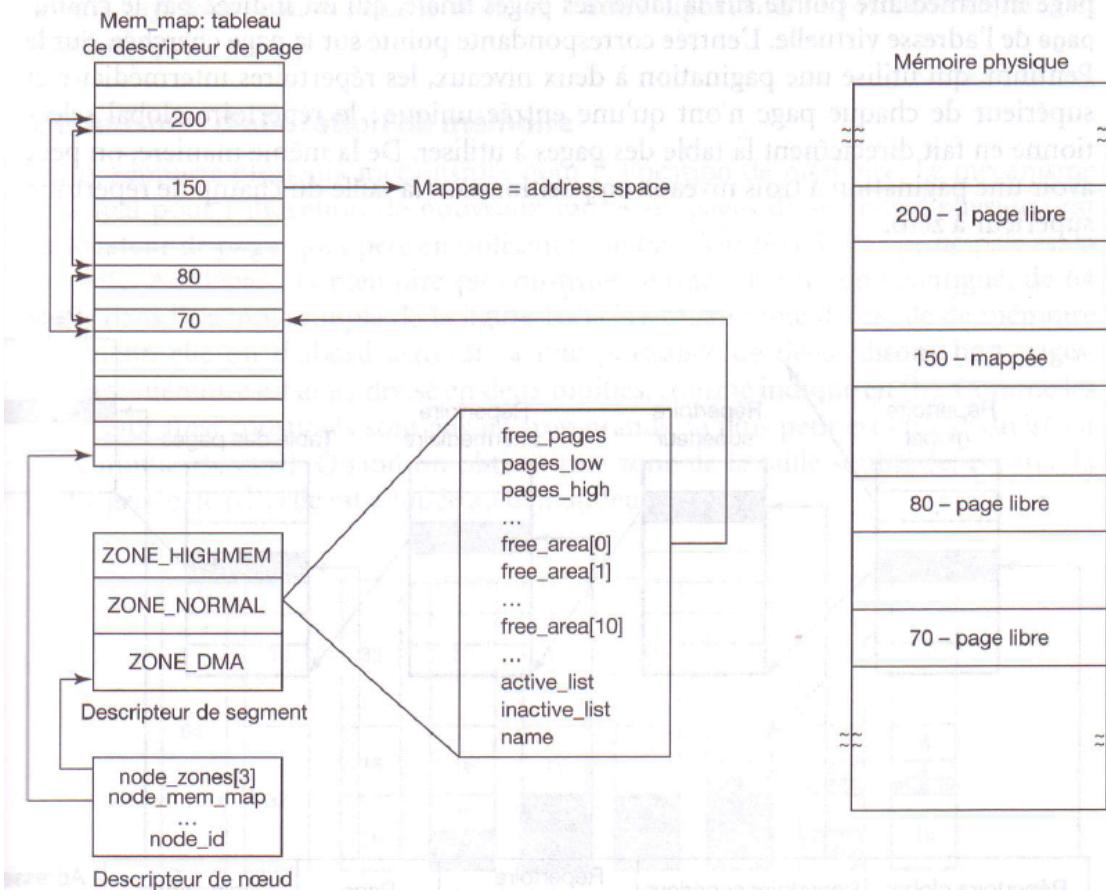
```
typedef struct page {  
    struct page *next; struct page *prev; // pour la liste doublement chainée de pages libres  
    struct inode *inode; unsigned long offset; // Son emplacement sur le disque  
    atomic_t count; // nombre de processus partageant cette page  
    unsigned flags; // indique si la page est libre, modifiée, verrouillée, etc.  
    ....  
    unsigned long map_nr; // numéro de la page physique (cadre)  
} mem_map_t;
```



# Cas de Linux (6)

## Mémoire physique

### Structures de données associées (vue d'ensemble)



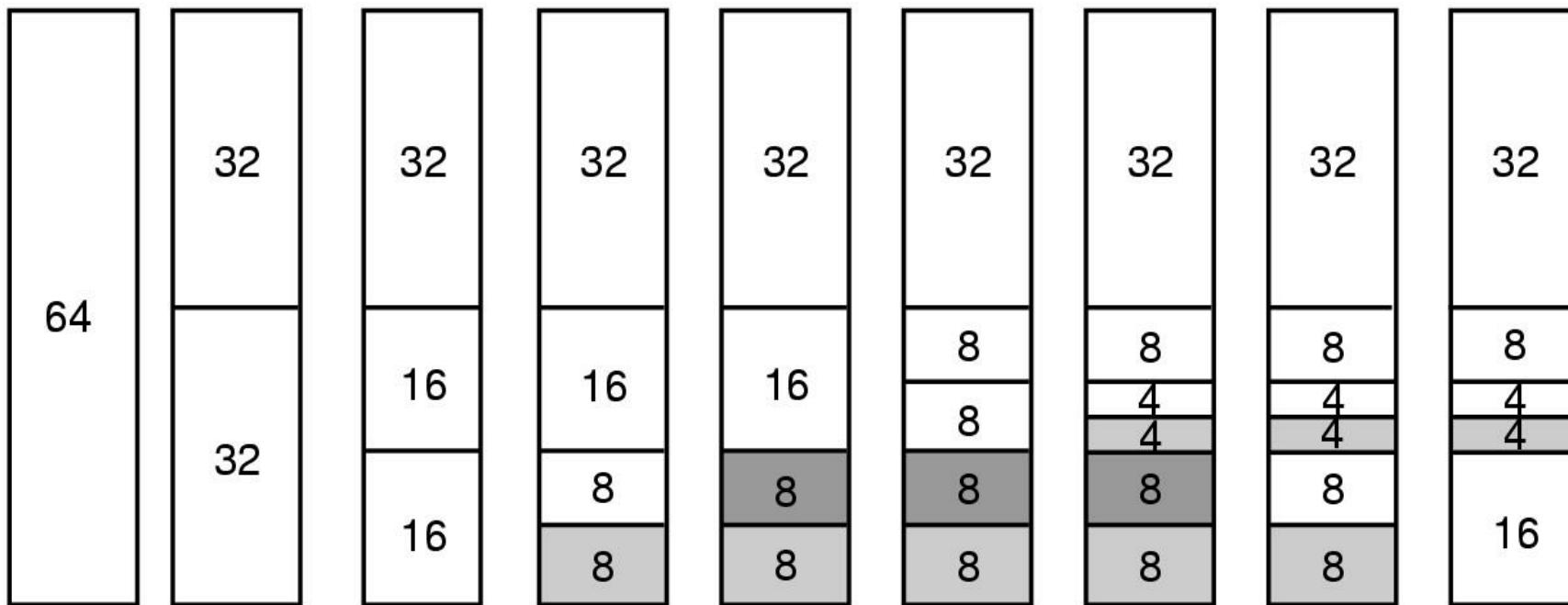
- L'algorithme de base d'allocation d'espace physique est un allocateur par subdivision.



# Cas de Linux (7)

## Politique d'allocation d'espace physique

Allocateur par subdivision :



- Cet algorithme conduit vers une importante fragmentation interne. Les espaces non utilisés (de la fragmentation interne) sont récupérés et gérés différemment par d'autres allocateurs de mémoire (allocateur SLAB, allocateur pour l'espace virtuel de processus).

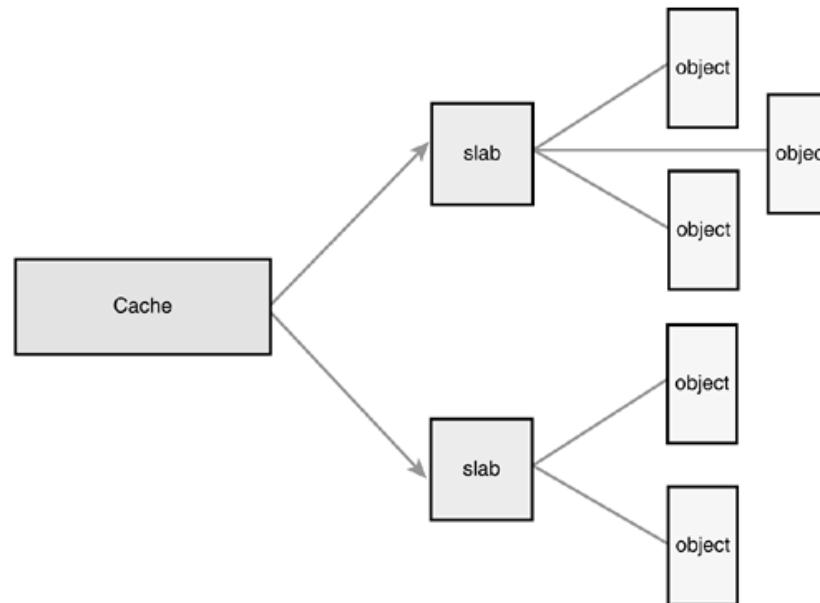


# Cas de Linux (8)

## Politique d'allocation d'espace physique

### Allocateur Slab

- Linux dispose d'un allocateur pour les objets noyau. L'idée de base est de disposer de "caches d'objets" réservés aux objets utilisés par le noyau (un cache par type d'objet, ex. task\_struct, mm\_struct).
- Certains de ces caches sont prêts à l'emploi dès le démarrage du système.
- Chaque cache d'objets est composé d'un ensemble de blocs mémoires appelés slabs.



<http://www.makelinux.net/books/lkd2/ch11lev1sec6>



# Cas de Linux (9)

## Politique d'allocation d'espace physique

### Allocateur Slab

- Sans ces caches, le noyau va perdre beaucoup de temps à allouer, initialiser et libérer le même type d'objet.
- Lorsqu'un objet est libéré d'un cache d'objets, son espace est préservé et son état initial est rétabli. Il est donc prêt à être alloué à un autre objet de même type.
- L'allocateur Slab gère un nombre variable de caches d'objets.
- La commande « cat /proc/slabinfo » donne la liste de caches d'objets.
- `kmem_cache_create()` permet de créer un cache pour un type d'objet.
- `Kmem_cache_malloc()`, `kmem_cache_free()` permettent d'allouer et de libérer un objet d'un cache d'objets.
- `kmem_cache_destroy()` permet de supprimer un cache d'objet.



# Cas de Linux (10)

## Politique d'allocation d'espace physique

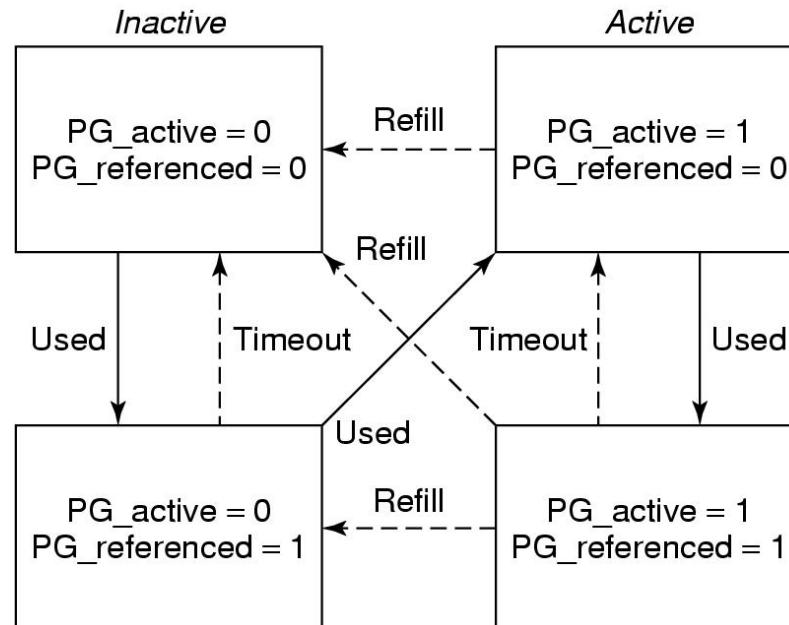
- Linux offre, via `vmalloc()`, la possibilité d'allouer un espace physique non forcément contigu à un espace contigu de l'**espace virtuel d'un processus**. Cette fonction retourne l'adresse virtuelle du début de la région dans l'espace d'adressage du processus appelant.
- La fonction `vfree()` permet de libérer un espace alloué par `vmalloc`.
- La fonction `kmalloc()` permet d'allouer un espace physique contigu à un espace contigu de l'espace virtuel.
- La fonction `kfree()` permet de libérer un espace alloué par `kmalloc`.



# Cas de Linux (11)

## Politique de remplacement

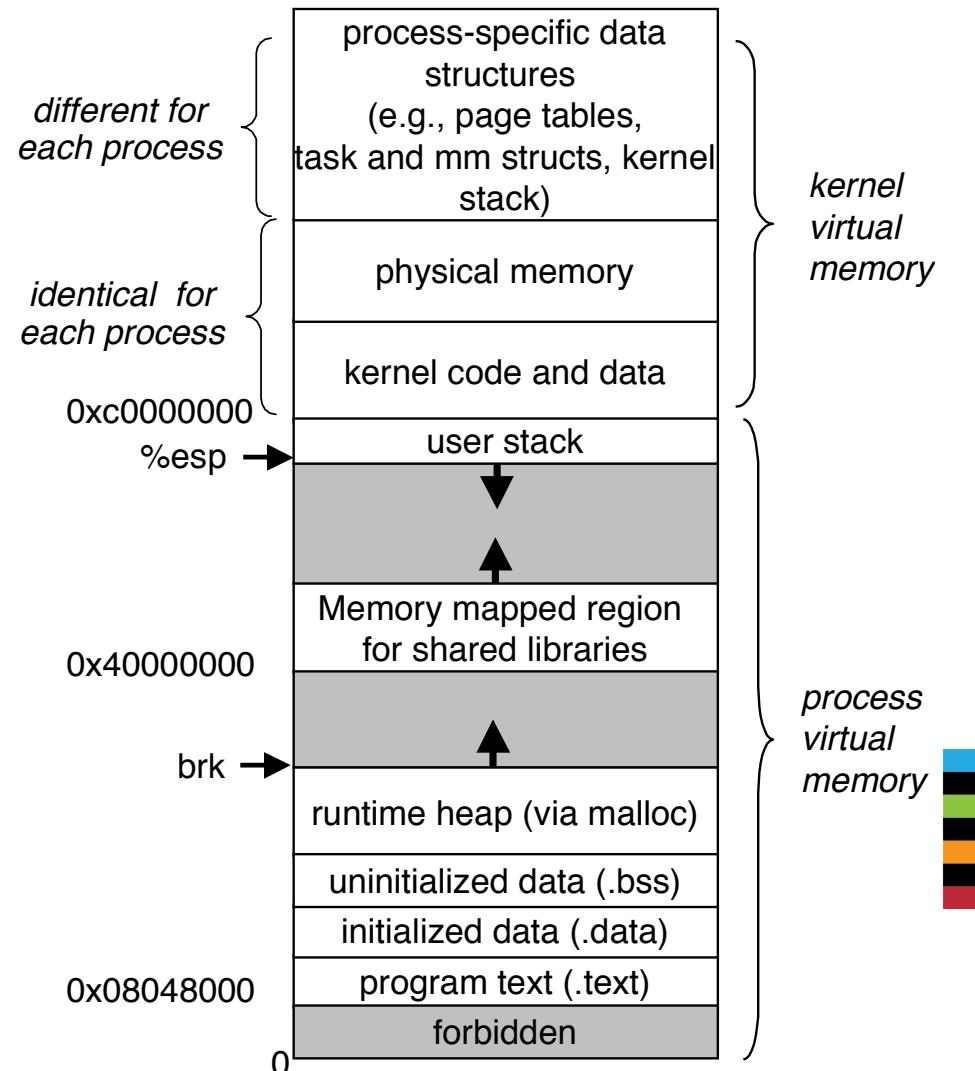
- Au démarrage, le processus Init lance un démon de pages (processus kswapd qui exécute l'algorithme de remplacement de pages de type « horloge ») pour chaque nœud mémoire.
- Ce démon est réveillé périodiquement ou suite à une forte allocation d'espace mémoire, pour vérifier si le nombre de cases libres en mémoire est trop bas (inférieur à un seuil min). Si ce n'est pas le cas, il se remet au sommeil.
- Sinon, Il parcourt les listes active et inactive de chaque zone à la recherche de pages à libérer.



# Cas de Linux (12)

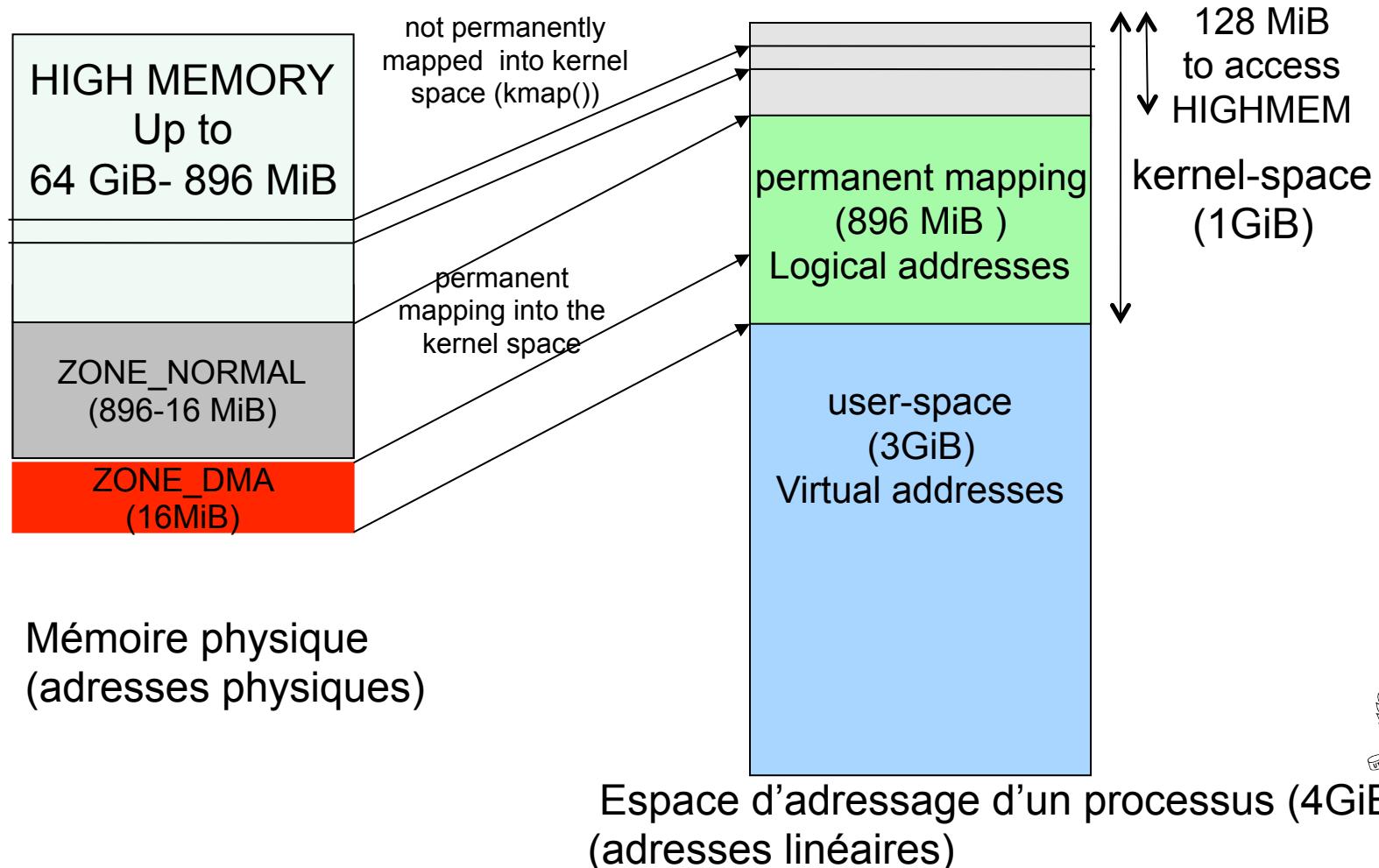
## Espace d'adressage d'un processus

- Chaque processus sur une machine de 32-bit a 3 GB d'espace d'adressage virtuel accessible en mode utilisateur,
- le GB restant est réservé aux tables des pages, la pile d'exécution, données et code du système d'exploitation, accessible en mode noyau.



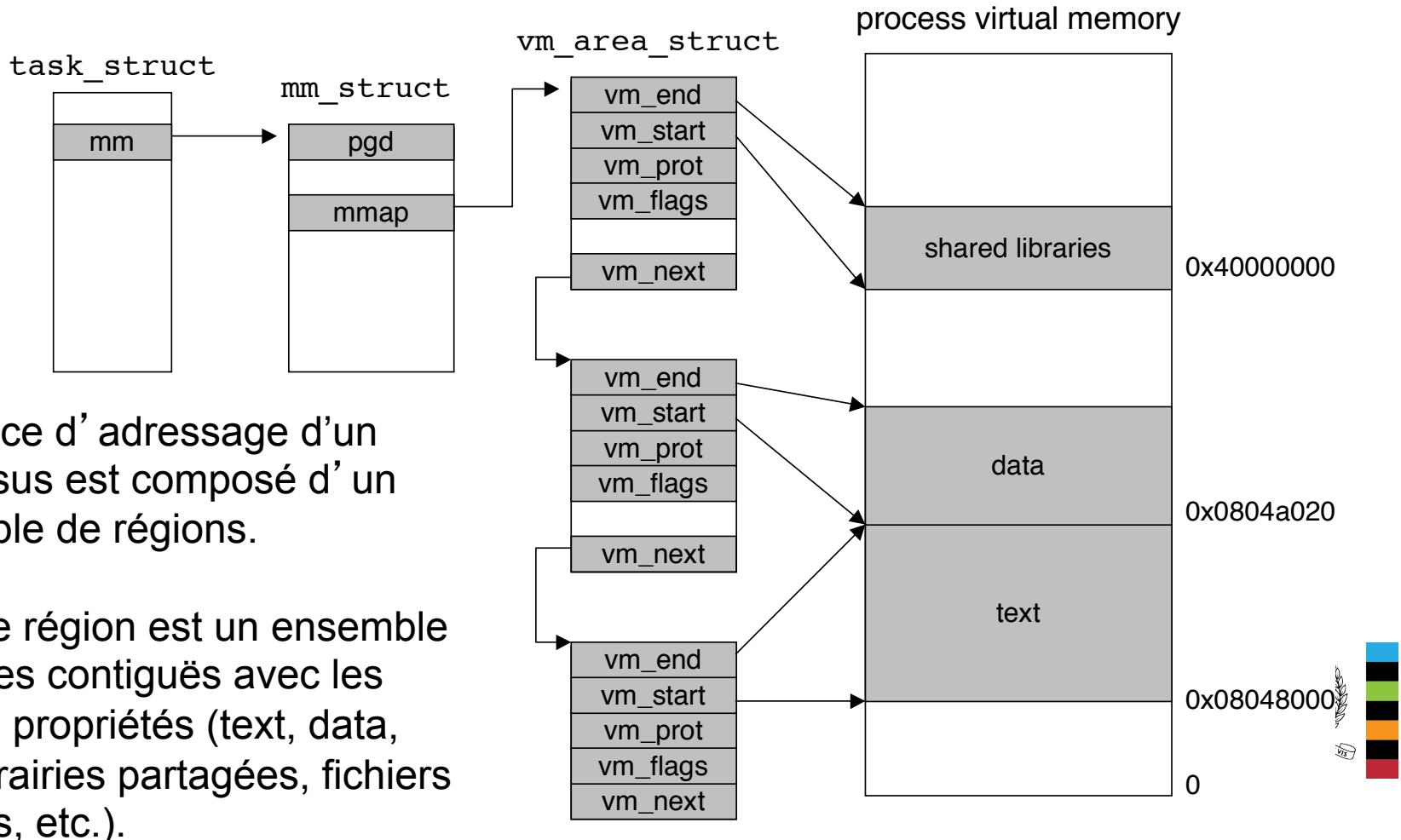
# Cas de Linux (13)

## Espace d'adressage d'un processus



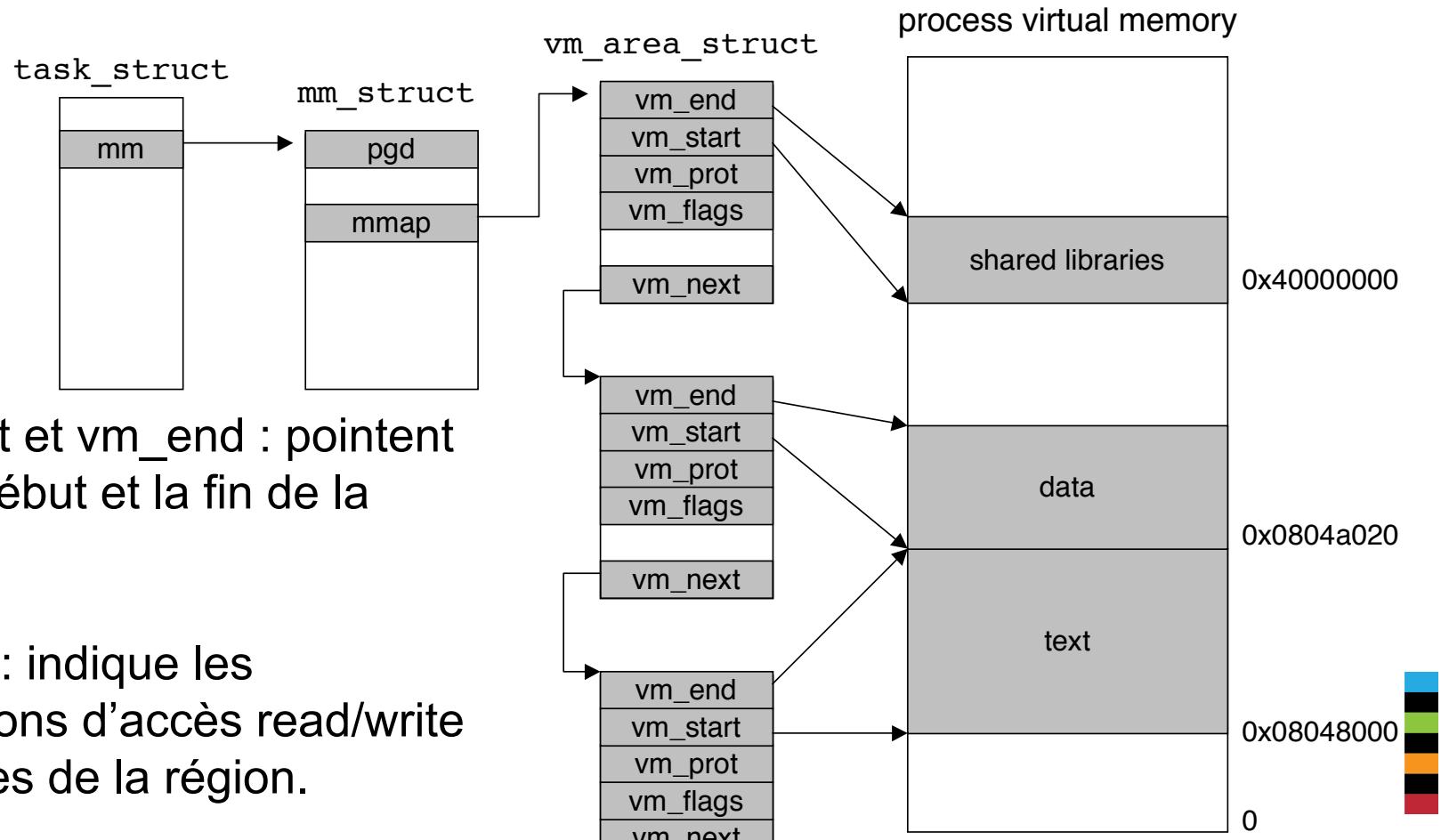
# Cas de Linux (15)

## Espace d'adressage d'un processus



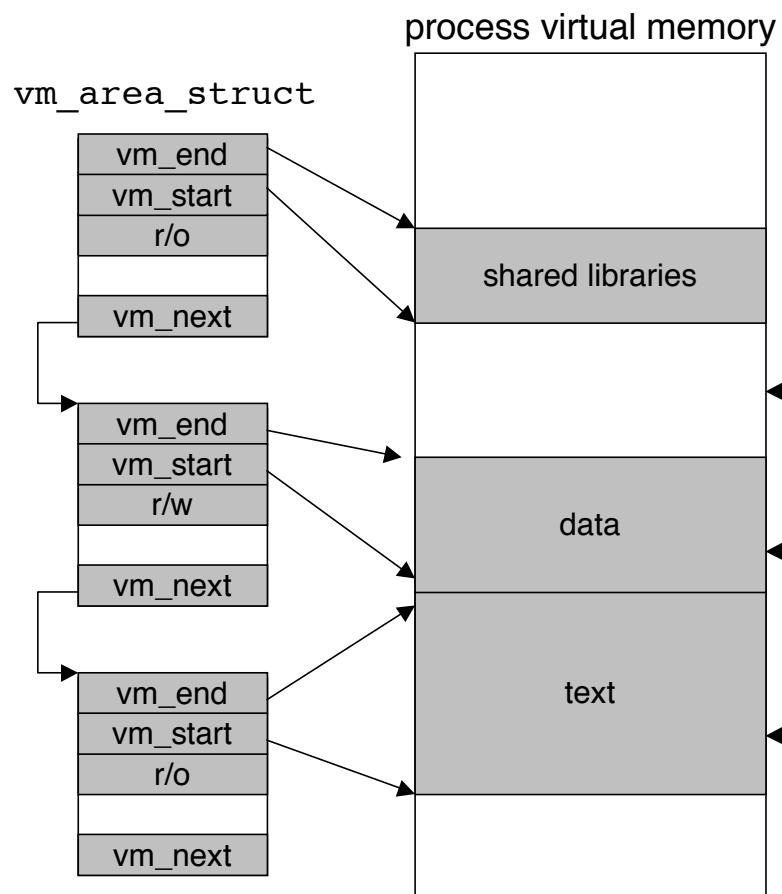
# Cas de Linux (16)

## Espace d'adressage d'un processus



# Cas de Linux (17)

## Espace d'adressage d'un processus

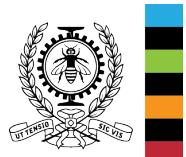


- Comment vérifier si une adresse virtuelle est légale ?
- Comment vérifier si l'accès est légal ?
- Comment vérifier si la page de l'adresse virtuelle est en mémoire ?
- Comment récupérer l'adresse physique ?

① segmentation fault:  
accessing a non-existing page

③ normal page fault

② protection exception:  
e.g., violating permissions by  
writing to a read-only page

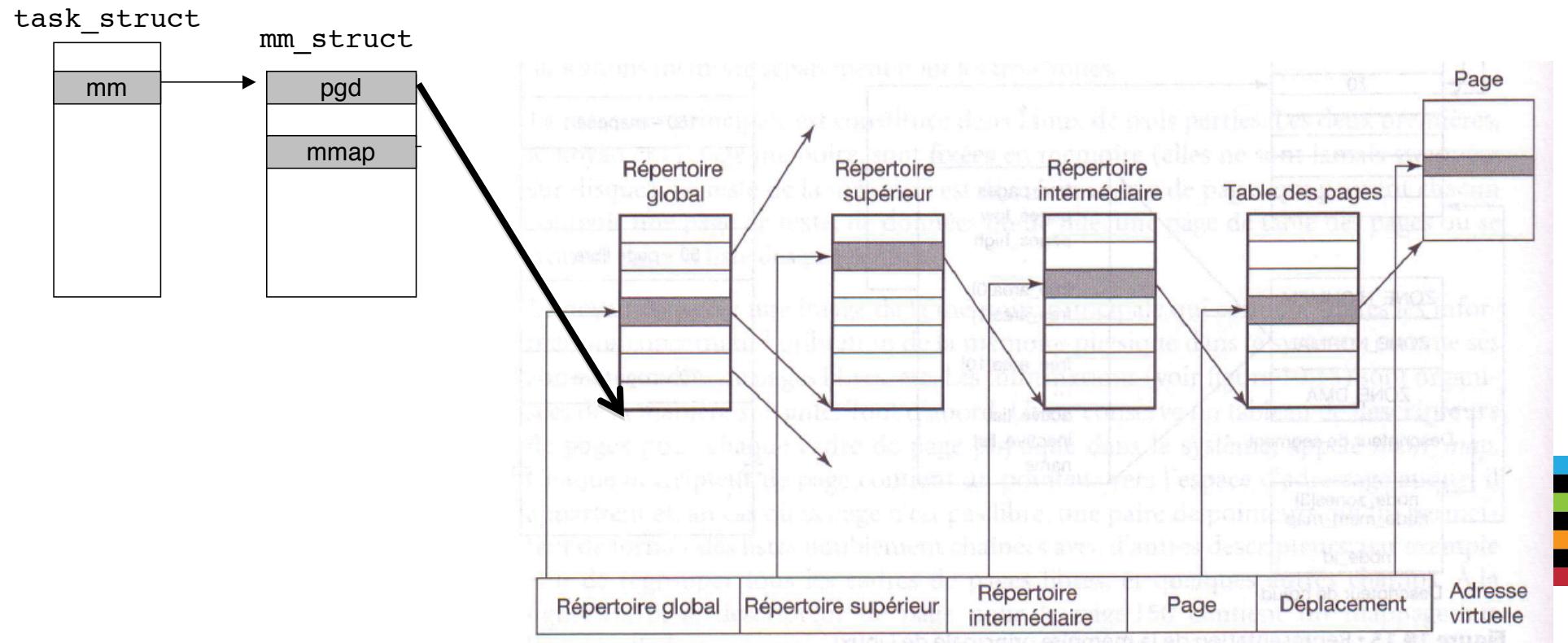


# Cas de Linux (18)

## Espace d'adressage d'un processus

### Adresse virtuelle

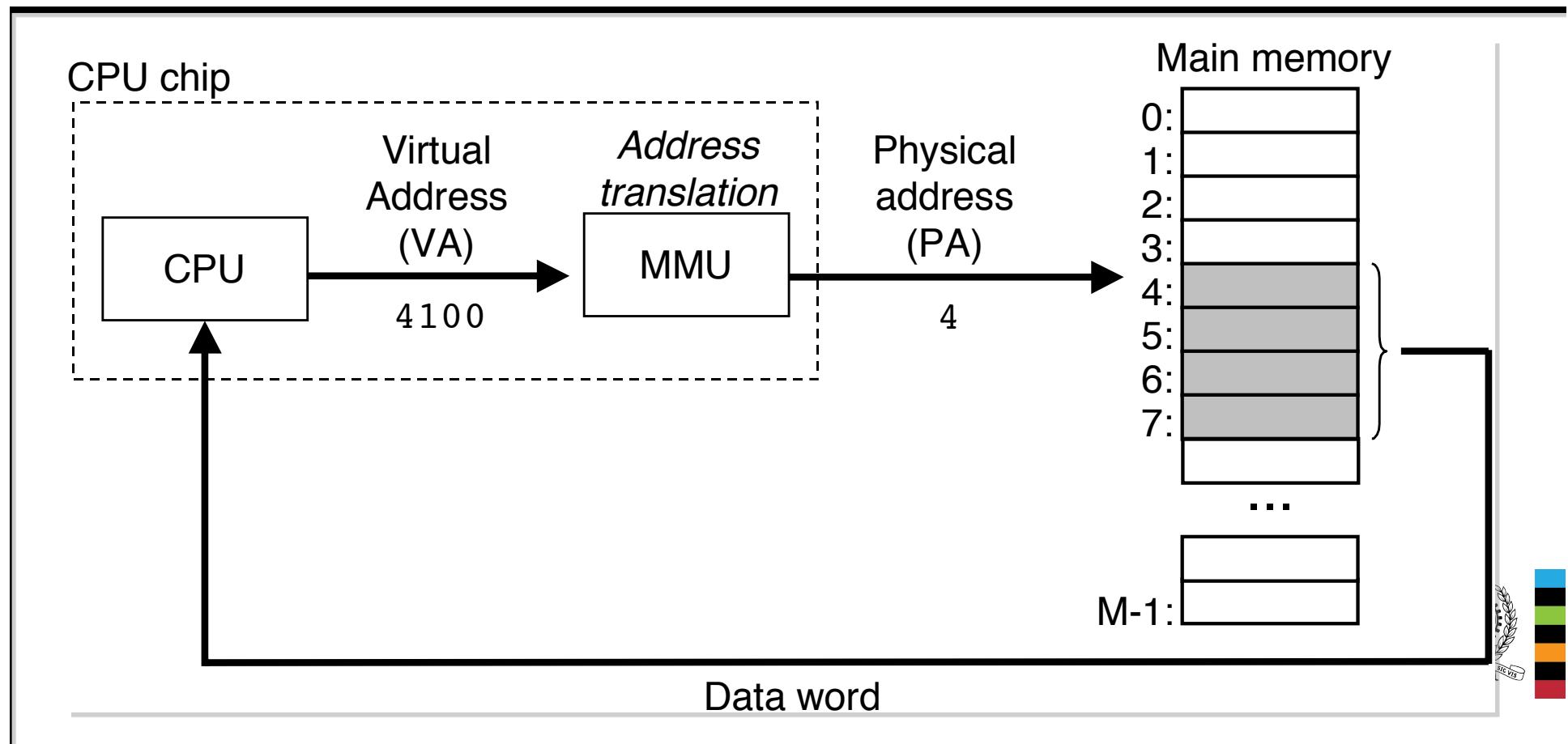
Table des pages à 4 niveaux



# Cas de Linux (19)

## Espace d'adressage d'un processus

### Translation d'adresses



# Cas de Linux (20)

## Espace d'adressage d'un processus

Quelques appels système :

- fork() : crée un nouveau processus avec un espace d'adressage partagé selon le principe COW (Copy-On-Write).
- mmap() crée une région dans l'espace d'adressage virtuel du processus appelant.
- mremap() déplace ou modifie la taille d'une région déjà mappée dans l'espace d'adressage virtuel du processus appelant.
- munmap() supprime une partie ou toute une région de l'espace d'adressage virtuel du processus appelant.
- shmat() attache une région partagée à l'espace d'adressage virtuel du processus appelant.
- shmdt() permet de détacher une région partagée à l'espace d'adressage virtuel du processus appelant.



# Cas de Linux (21)

## Espace d'adressage d'un processus

### Quelques appels système :

- brk() change la taille du segment de données.
- execve() remplace l'espace d'adressage du processus appelant par un autre, construit à partir d'un fichier exécutable (en général de format ELF <http://manpagesfr.free.fr/man/man5/elf.5.html>).
- exit() libère tout l'espace d'adressage du processus appelant.
- Les appels système splice et sendfile permettent d'éviter les copies. Ils transfèrent les pages d'un espace d'adressage virtuel à un autre par le biais des tables de pages.



## Exercice 2

### Caractéristiques du système :

Mémoire physique de 64 mots de 8 bits

Taille d'une page = 4 mots

Table des pages à deux niveaux:

Adresse virtuelle sur 6 bits : 2 bits (niv. 1), 2 bits (niv. 2), 2 bits (décalage)

Une entrée dans la table des pages sur 8 bits:

- 2 bits de contrôle : bit de présence, bit de référence (10,11)  
si bit de présence =0 et bit de référence =1, la page est dans la zone de « «swap » »
- 6 bits pour l'adresse

Deux processus P1 et P2 sont en mémoire.

La table de premier niveau de P1 commence à 32

La table de premier niveau de P2 commence à 52.

Supposez que l'état courant de la mémoire est :



0	1	1	1	16	0	0	0	32	0	0	4	48	1	0	8
1	1	1	2	17	0	0	0	33	1	1	36	49	1	0	0
2	0	0	0	18	0	0	0	34	1	1	20	50	0	0	0
3	0	0	0	19	0	0	0	35	0	0	52	51	0	0	0
4	1	0	6	20	1	1	4	36	0	1	2	52	1	1	48
5	0	1	9	21	1	1	44	37	0	0	0	53	1	1	24
6	0	0	59	22	0	1	3	38	0	0	0	54	0	0	28
7	1	0	62	23	1	1	12	39	0	0	0	55	0	0	12
8	1	0	7	24	0	0	0	40				56	1	0	8
9	1	0	6	25	0	0	0	41				57	1	1	7
10	1	0	5	26	0	1	0	42				58	1	0	6
11	0	1	2	27	1	0	16	43				59	1	0	5
12	0	0	0	28				44	1	0	48	60	1	0	0
13	0	0	30	29				45	1	0	52	61	1	0	56
14	0	0	0	30				46	0	0	56	62	0	0	0
15	0	0	0	31				47	0	0	60	63	0	0	0

Pages déplacées en mémoire secondaire (va-et-vient):

Page 0

0	0	40
0	0	0
0	0	0
0	0	0

Page 1

0	1	10
0	0	12
0	0	14
1	0	16

Page 2

0	1	1
0	0	0
0	0	0
1	0	0

Page 3

1	0	63
0	0	29
0	1	0
0	0	0



Figure 1

a) Pour chaque page du processus P1, indiquez laquelle des situations suivantes s'applique (utilisez le tableau à la dernière page):

- Présente en mémoire (spécifiez dans quel cadre)
- Absente de la mémoire ou invalide
- Dans le va-et-vient (indiquez à quelle position)

b) Pour chacun des 16 cadres de la mémoire, indiquez laquelle des situations suivantes s'applique (utilisez le tableau à la dernière page):

1. Contient une page d'un processus (indiquez le processus)
2. Contient une table de pages (indiquez le processus)
3. Libre
4. Autre

c) Le processus P1 veut accéder aux adresses logiques 37 et 40? Dans chaque cas, indiquez s'il y aura faute de page, l'adresse physique obtenue, ainsi que le contenu de l'octet à cette adresse. *Remarque:* s'il y a faute de page, indiquez les changements à la mémoire, une fois complété le traitement de cette faute de page.



d) Supposons que le processus P1 ne peut occuper plus de trois cadres en mémoire et que l'algorithme de l'horloge est utilisé pour le remplacement de page. Supposons aussi que le système n'utilise aucun autre algorithme de remplacement global. Voici des exemples de séquences d'accès à des pages réalisées par le processus P1 depuis le début de son exécution jusqu'au moment où la mémoire se retrouve dans l'état illustré à la figure 1:

- 4, 10, 8, 9, 11, 8
- 8, 9, 11, 8, 9
- 10, 9, 8, 11

Indiquez lesquelles, parmi ces séquences, peuvent mener à l'état de la mémoire illustré à la figure 1? (Justifiez votre réponse)



# Lectures suggérées

- Notes de cours: Chapitres 9 et 10  
(<http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap9.pdf>  
<http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap10.pdf> )



## Annexe - Exemple 1 : Espace virtuel d'un processus

Où seront stockées les données des variables déclarées (cas de C) ?

```
// http://ilay.org/yann/articles/mem/mem0.html (à lire avant le prochain lab) //
memoirevirtuelle.cpp
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
static int i_stat = 4; /* Stocké dans le segment data */
int i_glob;           /* Stocké dans le segment bss */
int *pi_pg;          /* Stocké dans le segment bss */
int main(int nargs, char **args) {
    int *pi_loc;      /* dans la frame 1 de la pile */
    void *sbrk0 =(void *) sbrk(0); /*l'adresse de base avant le 1er malloc */
    if (!(pi_loc = (int *) malloc(sizeof(int) * 16)))
        return 1;
    if (!(pi_pg = (int *) malloc(sizeof(int) * 8)))
    {
        free(pi_loc);
        return 2;
    }
```



# Exemple 1 : Espace virtuel d'un processus

Où seront stockées les données des variables déclarées ?

```
// afficher les adresses
printf("adresse de i_stat = 0x%08x (zone programme, segment data)\n", &i_stat);

printf("adresse de i_glob = 0x%08x (zone programme, segment bss)\n", &i_glob);

printf("adresse de pi_pg = 0x%08x (zone programme, segment bss)\n", &pi_pg);

printf("adresse de main = 0x%08x (zone programme, segment text)\n", main);

printf("adresse de nargs = 0x%08x (pile frame 1)\n", &nargs);

printf("adresse de args = 0x%08x (pile frame 1)\n", &args);

printf("adresse de pi_loc = 0x%08x (pile frame 1)\n", &pi_loc);

printf("sbrk(0) (heap) = 0x%08x (tas)\n", sbrk0);

printf("pi_loc = 0x%08x (tas)\n", pi_loc);

printf("pi_pg = 0x%08x (tas)\n", pi_pg);
```



## Exemple 1 : Espace virtuel d'un processus

Où seront stockées les données des variables déclarées ?

```
// récupérer le contenu /proc/pid/maps du processus
char buf[128];
printf("Affichage du fichier /proc/%d/maps\n",getpid());
sprintf(buf,"/proc/%d/maps",getpid());
int fd1 = open(buf,O_RDONLY);
while (read(fd1,buf,128)>0) write(1, buf,128);
write(1, "\n",2);
close(fd1);
free(pi_pg);
free(pi_loc);
return 0;
}
```



# Exemple 1 : Espace virtuel d'un processus

## Où seront stockées les données des variables déclarées ?

```
jupiter$ g++ memoirevirtuelle.cpp -o memoirevirtuelle  
jupiter$ ./memoirevirtuelle  
adresse de i_stat = 0x006012d4 (zone programme, segment data)  
adresse de i_glob = 0x006012e0 (zone programme, segment bss)  
adresse de pi_pg = 0x006012e8 (zone programme, segment bss)  
adresse de main = 0x00400096c (zone programme, segment text)  
adresse de nargs = 0xfd058cac (pile frame 1)  
adresse de args = 0xfd058ca0 (pile frame 1)  
adresse de pi_loc = 0xfd058d38 (pile frame 1)  
sbrk(0) (heap) = 0x00c24000 (tas)  
pi_loc = 0x00c24010 (tas)  
pi_pg = 0x00c24060 (tas)
```



## **Exemple 1 : Espace virtuel d'un processus**

**Où seront stockées les données des variables déclarées ?**

## Affichage du fichier /proc/2850/maps

00400000-00402000 r-xp 00000000 00:26 125178476 memoirevirtuelle

00601000-00602000 rw-p 00001000 00:26 125178476 memoirevirtuelle

00c24000-00c45000 rw-p 00000000 00:00 0 [heap]

1

Les champs sont : adresses (deb et fin), permissions (p pour privé et Copy-On-Write), offset, périph, i-nœud, chemin d'accès (pour localiser le texte et les données initialisées)

**Lancez une seconde fois le programme.  
Est-ce que vous obtenez les mêmes adresses ?**



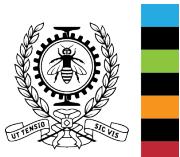
## Exemple 2 : Exécution d'un programme (évolution de la pile d'exécution)

(Livre sur les SE de Bort LAMIROY, Laurent NAJMAN, , Hugues TALBOT)

Le code assembleur dépend de l'architecture. La commande `g++ -S prog.cpp` produit le code dans `prog.s`.

Trois paramètres suffisent pour caractériser l'état d'exécution d'un programme. Ces paramètres sont mémorisés dans des registres. Par exemple, pour Intel x86, ces paramètres sont :

- `%cs` contient l'adresse de base (le premier mot mémoire) du programme.
- `%eip` est le compteur ordinal. Il pointe en permanence sur la prochaine instruction à exécuter. Il est initialisé à l'adresse de la fonction `main`.
- `%esp` est le pointeur de pile. Il évolue dynamiquement avec l'utilisation de la pile, au fur et à mesure des allocations de données intermédiaires (données locales, les paramètres d'une fonction, etc.). On utilise souvent un autre registre `%ebp` (Extended Base Pointer) pour y stocker à chaque appel à une fonction, l'adresse du sommet de la pile. À la fin de l'exécution de la fonction, le contenu de `%ebp` est copié dans `%esp`.



# Exemple 2 : Exécution d'un programme (évolution de la pile d'exécution)

(Livre sur les SE de Bort LAMIROY, Laurent NAJMAN, , Hugues TALBOT)

```
int addition(int a) {return a+5; }
static int arg1= 5;
static int arg2= 1;
int main() {
    int param= arg1 + arg2;
    arg2 = addition(param);
}
```

Instructions ass.	Fonction addition
pushl %ebp	Sauvegarder le contenu de %ebp (avant l'exécution de la fonction) dans la pile.
movl %esp, %ebp	Mettre à jour %ebp
movl 8(%ebp), %eax	Charger dans %eax la valeur du paramètre a (qui se trouve à 8 octets de l'adresse contenu dans %ebp).
addl \$5 %eax	Ajouter de 5 à %eax
popl %ebp	Récupérer dans %ebp, le contenu de %ebp sauvegardé avant l'exécution de la fonction.
ret	Retour à l'appelant (dépiler %eip, ...)



# Exemple 2 : Exécution d'un programme (évolution de la pile d'exécution)

(Livre sur les SE de Bort LAMIROY, Laurent NAJMAN, , Hugues TALBOT)

```
int addition(int a)
{   return a+5; }
static int arg1= 5;
static int arg2= 1;
int main() {
    int param= arg1 + arg2;
    arg2 = addition(param);
}
```

Instructions ass.	Fonction main
pushl %ebp	Sauvegarder le contenu de %ebp (avant l'exécution de la fonction) dans la pile.
movl %esp, %ebp	Mettre à jour %ebp
subl \$4, %esp	Réserver 4 octets pour param. %esp contient l'adresse de param ( (%ebp)-4)
movl arg1, %edx	%edx = arg1
movl arg2, %eax	%eax = arg2
leal (%edx,%eax), %eax	%eax= %edx + %eax
movl %eax, -4(%ebp)	param = %eax
pushl -4(%ebp)	Empiler param comme argument
call addition	Appeler la fonction addition (empiler %eip puis aller au début de la fonction)
addl \$4, %esp	Mettre à jour %esp (dépiler l'argument)
movl %eax, argv2	arg2 = %eax
leave	Restorer %ebp et %esp.
ret	Retour à l'appelant (dépiler %eip,...)



# Exemple 3 : Alignement de la mémoire

(Livre sur les SE de Bort LAMIROY, Laurent NAJMAN, , Hugues TALBOT)

```
typedef struct { double valeur; short int index; char code; char id;} S1;
typedef struct {char id; short int index; double valeur; char code;} S2;
#include <stdio.h>
#include <stdlib.h>
int main()
{ S1* s1 = (S1*) malloc(sizeof(S1));
  S2* s2 = (S2*) malloc(sizeof(S2));
  char * a_S1 = (char*) s1;
  char* a_S2 = (char*) s2;
  printf("Taille d'un double = %lu\n", sizeof(double));
  printf("Taille d'un entier court = %lu\n", sizeof(short int));
  printf ("Taille d'un caractère = %lu \n", sizeof(char));

  printf("\n Taille de S1 (double,short,char,char)= %lu\n", sizeof(S1));
  printf("Champ \t Offset \t Taille \n");
  printf("1 \t %ld \t %lu\n", ((char*)(&(s1->valeur))) - a_S1,sizeof(double));
  printf("2 \t %ld \t %lu\n", ((char*)(&(s1->index))) - a_S1,sizeof(short int));
  printf("3 \t %ld \t %lu\n", ((char*)(&(s1->code))) - a_S1,sizeof(char));
  printf("4 \t %ld \t %lu\n", ((char*)(&(s1->id))) - a_S1,sizeof(char));
```



## Exemple 3 : Alignement de la mémoire

(Livre sur les SE de Bort LAMIROY, Laurent NAJMAN, , Hugues TALBOT)

```
printf("\n Taille de S2 (char,short,double,char)= %lu\n", sizeof(S2));
printf("Champ \t Offset \t Taille \n");
printf("1 \t %ld \t %lu\n", ((char*)(&(s2->id))) - a_S2,sizeof(char));
printf("2 \t %ld \t %lu\n", ((char*)(&(s2->index))) - a_S2,sizeof(short int));
printf("3 \t %ld \t %lu\n", ((char*)(&(s2->valeur))) - a_S2,sizeof(double));
printf("4 \t %ld \t %lu\n", ((char*)(&(s2->code))) - a_S2,sizeof(char));
return 0;
}
// cas Mac OS X 10.8.5
jupiter$ ./alignement
Taille d'un double = 8
Taille d'un entier court = 2
Taille d'un caractère = 1
```

char	aucun
short int	adresse paire
double	Adresse multiple de 8
struct	Adresse multiple de 8

Champ	Offset	Taille
1	0	8
2	8	2
3	10	1
•	11	1

Champ	Offset	Taille
1	0	1
2	2	2
3	8	8
4	16	1

## Exemple 4 : Parcours d'un tableau multidimensionnel

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ char A[512][4096];
  int i,j;
  for(i=0; i<512; i++)
    for(j=0; j<4096; j++)
      A[i][j] = 'A';
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ char A[512][4096];
  int i,j;
  for(j=0; j<4096; j++)
    for(i=0; i<512; i++)
      A[i][j] = 'A';
  return 0;
}
```

```
jupiter$ time ./parcours_Lig_Lig
real 0m0.017s
user 0m0.010s
sys 0m0.003s
```

```
jupiter$ time ./parcours_Col_Col
real 0m0.035s
user 0m0.030s
sys 0m0.003s
```

