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 2017

Chapitre 7 - Gestion de la mémoire

- **Généralités**
- **Espace d'adressage d'un processus**
- **La pagination pure**
- **Segmentation avec ou sans pagination**
- **Comment organiser la mémoire physique ?**

Généralités

- Le gestionnaire de la mémoire est le composant du système d'exploitation qui se charge de gérer l'allocation d'espace mémoire nécessaire à l'exécution du SE et des processus.

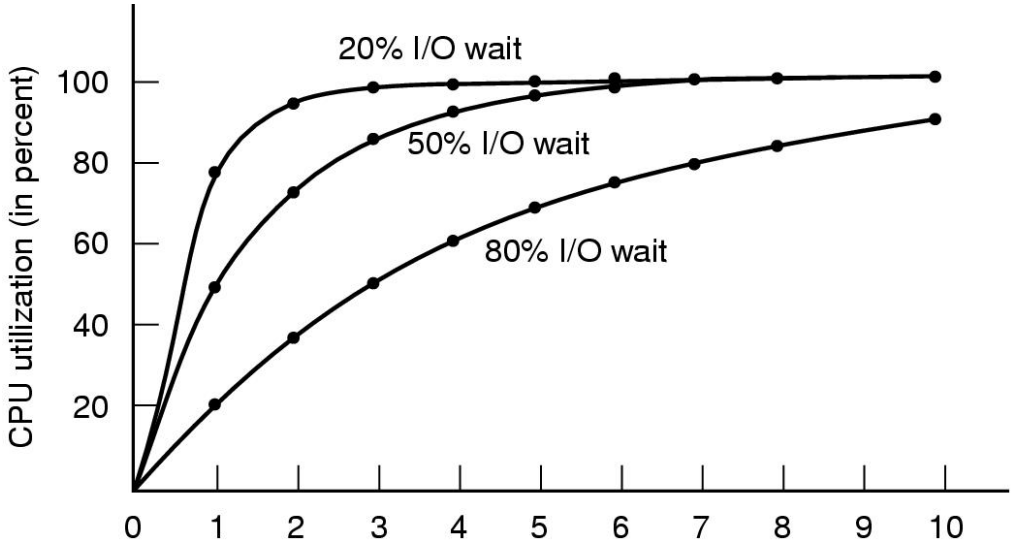
- Il distingue deux types d'espace : la mémoire physique (espace partagé) et les espaces d'adressage des processus (espaces privés).

- Il se charge de gérer les deux types d'espace :
  - **organisation et représentation des deux types d'espace,**
  - **politique de placement (premier ajustement (First-fit)**, meilleur ajustement (Best-fit), pire ajustement (Worst-fit), **par subdivision** (Linux)).
  - **politique de remplacement** (FIFO, LRU, Optimal, espace de travail (working set), etc.)
  - **politique d'allocation d'espace aux processus** (avant l'exécution, à la demande (durant l'exécution avec ou sans pré-allocation)).

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.

- 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.

Généralités (3)

- Relocation : La possibilité de retirer et déplacer un processus en mémoire.
- Protection : Les processus ne doivent pas se corrompre.

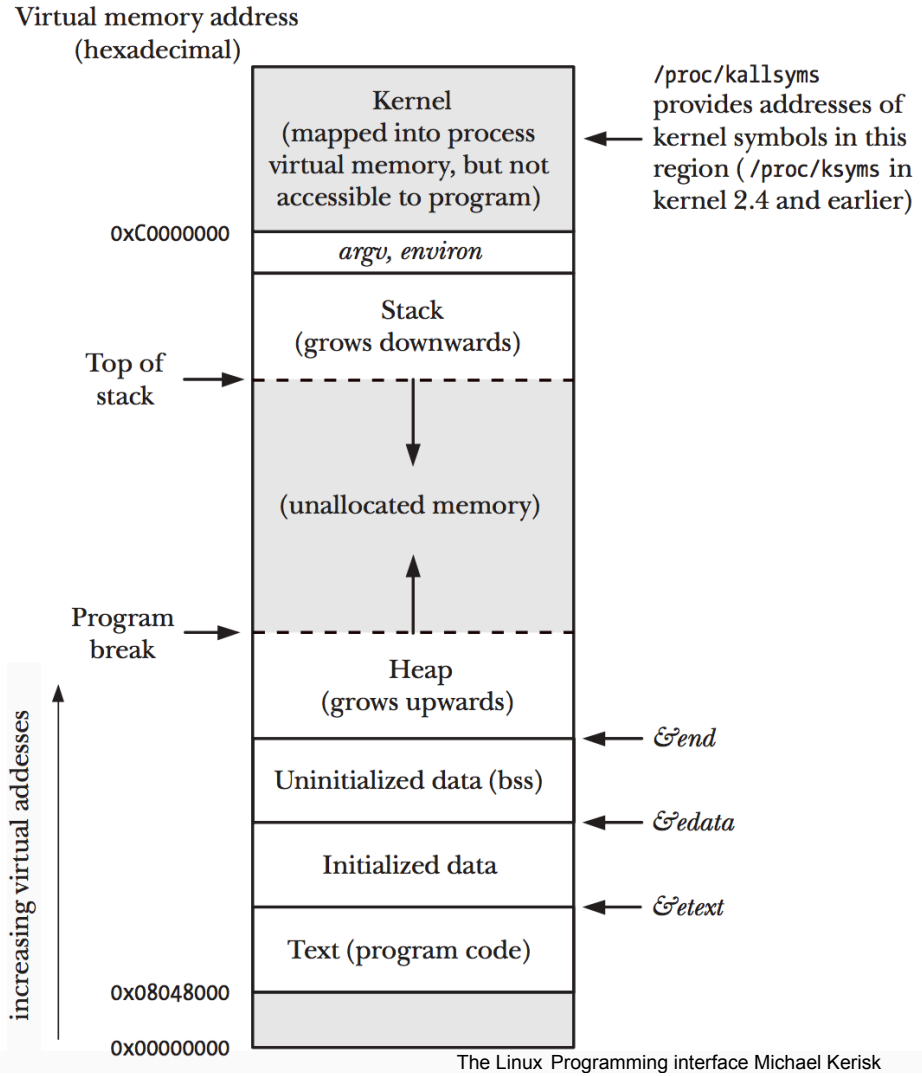
➔

- Chaque processus 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).
- La taille de l'espace d'adressage d'un processus peut être donc beaucoup plus grande que celle de la mémoire physique ➔ mémoire virtuelle (2^n pour un adressage virtuel sur n bits).

Espace d'adressage d'un processus

- L'espace d'adressage d'un processus est généré, en partie, par les compilateurs et les éditeurs de liens => mémoire virtuelle du processus ou espace d'adressage virtuel du processus.
  - Text *segment :* instructions du programme du process (lecture seul et partagé);
  - *Initialized data segment :* variables globales et statiques explicitement initialisées.
  - *Uninitialized data segment :* variables globales et statiques non explicitement initialisées.

=> (**commande size** pour récupérer la taille de chaque segment).



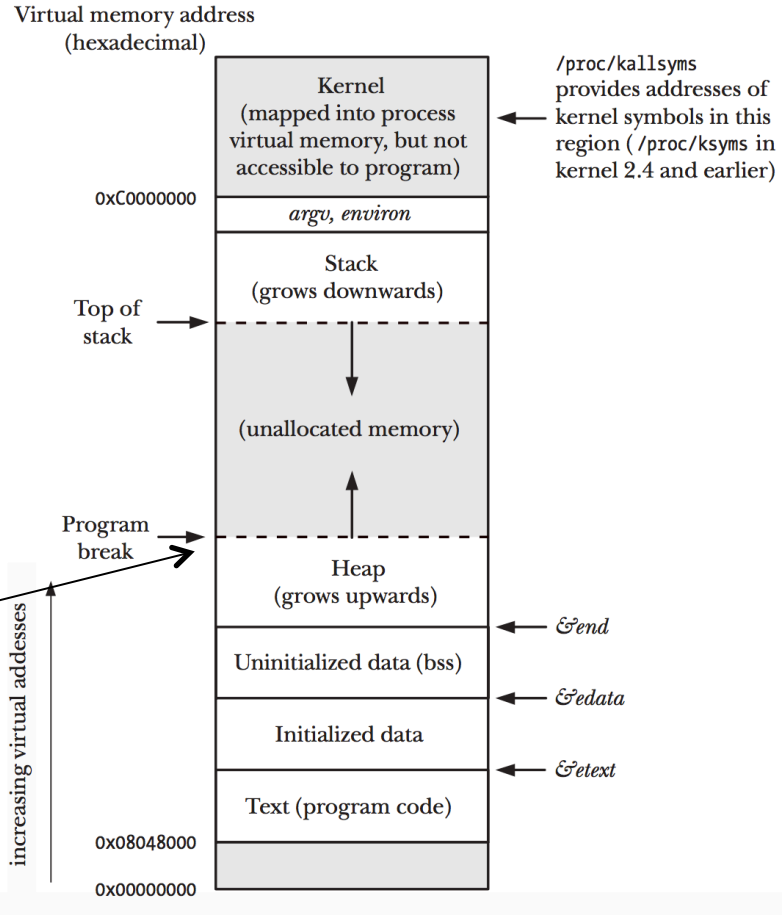
The Linux Programming interface Michael Kerisk

Espace d'adressage d'un processus (2)

- *Stack* : sections (frames) allouées dynamiquement (1 par fonction en cours d'exécution). Chaque section contient :
  - les variables locales, les arguments,
  - un espace pour la valeur de retour de la fonction et
  - un espace pour la sauvegarde de certains registres de la fonction appelante (comme le sommet de pile, et le compteur ordinal avant l'appel).

- H*eap : espace disponible pour les allocations dynamiques (fonctions malloc, calloc, brk, sbrk):*
  - *brk et sbrk permettent d'incémenter* ***break,*** *sbrk(0) retourne* ***break****.*
  - *malloc/*calloc alloue un espace contiguë qui peut être ensuite libéré par la fonction *free* (*malloc/calloc* peut faire appel à *sbrk*)

  ➔ *liste doublement chainées de blocs libres.*



Espace d'adressage d'un processus (3)
Exemple 1

```
//   http://ilay.org/yann/articles/mem/mem0.html
// memoirevirtuelle.cpp
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <fcntl.h>
   static int i_stat = 4; /* Stocké dans le segment données initialisées */
   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)))    /* allocation dynamique
                    return 1;
        if (!(pi_pg  = (int *) malloc(sizeof(int) *  8)))
                {     free(pi_loc);
                      return 2;
                }
```

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

Espace d'adressage d'un processus (4)
Exemple 1

```
// 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);
```

Espace d'adressage d'un processus (5)
Exemple 1

```
//  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;
}
```

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

Espace d'adressage d'un processus (6)
Exemple 1

```
jupiter$ g++ memoirevirtuelle.cpp -o memoirevirtuelle

jupiter$ ./memoirevirtuelle

adresse de i_stat = 0x0060207c (zone programme, segment data)
adresse de i_glob = 0x00602088 (zone programme, segment bss)
adresse de pi_pg  = 0x00602090 (zone programme, segment bss)
adresse de main   = 0x00400870 (zone programme, segment text)
adresse de nargs  = 0xdf12b7bc (pile frame 1)
adresse de args   = 0xdf12b7b0 (pile frame 1)
adresse de pi_loc = 0xdf12b848 (pile frame 1)
sbrk(0) (heap)    = 0x01a17000 (tas)
pi_loc            = 0x01a17010 (tas)
pi_pg             = 0x01a17060 (tas)
```

Espace d'adressage d'un processus (7)
Exemple 1

Affichage du fichier /proc/29430/maps

00400000-00401000 r-xp 00000000 00:2a 148650549	memoirevirtuelle
00601000-00602000 r--p 00001000 00:2a 148650549	memoirevirtuelle
00602000-00603000 rw-p 00002000 00:2a 148650549	memoirevirtuelle
01a17000-01a38000 rw-p 00000000 00:00 0	[heap]

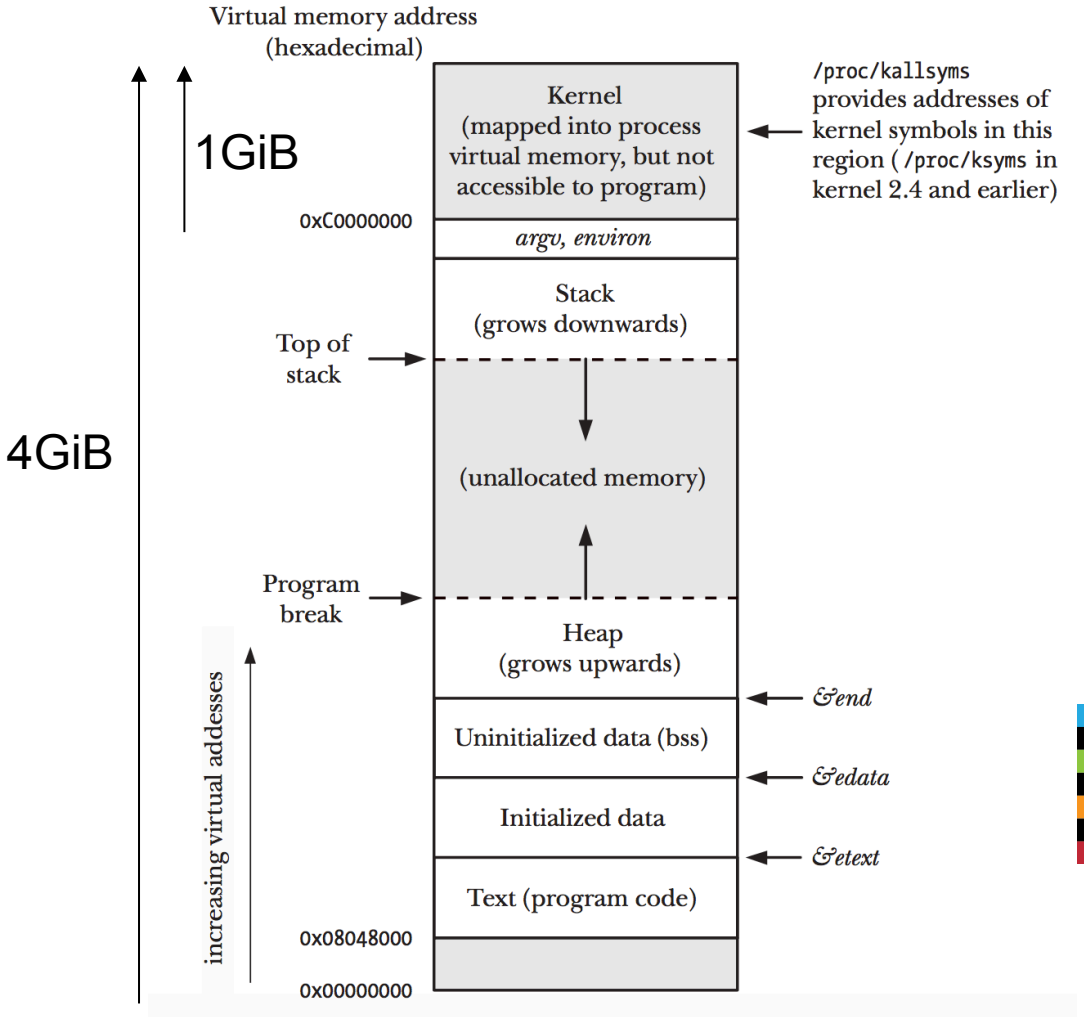
.....

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

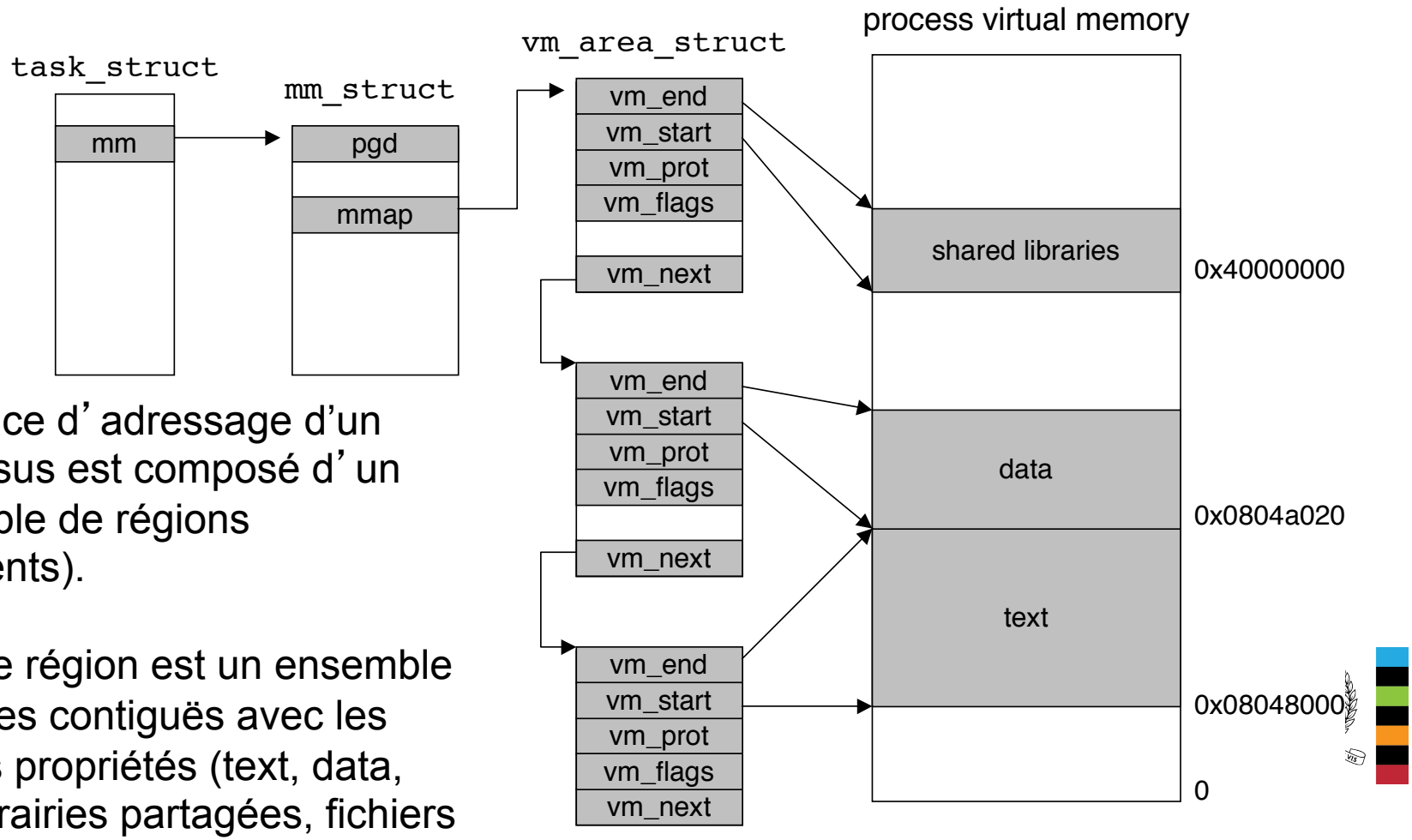
Lancez une seconde fois le programme.
Est-ce que vous obtenez les mêmes adresses ?

Espace d'adressage d'un processus (8)
Cas de Linux

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

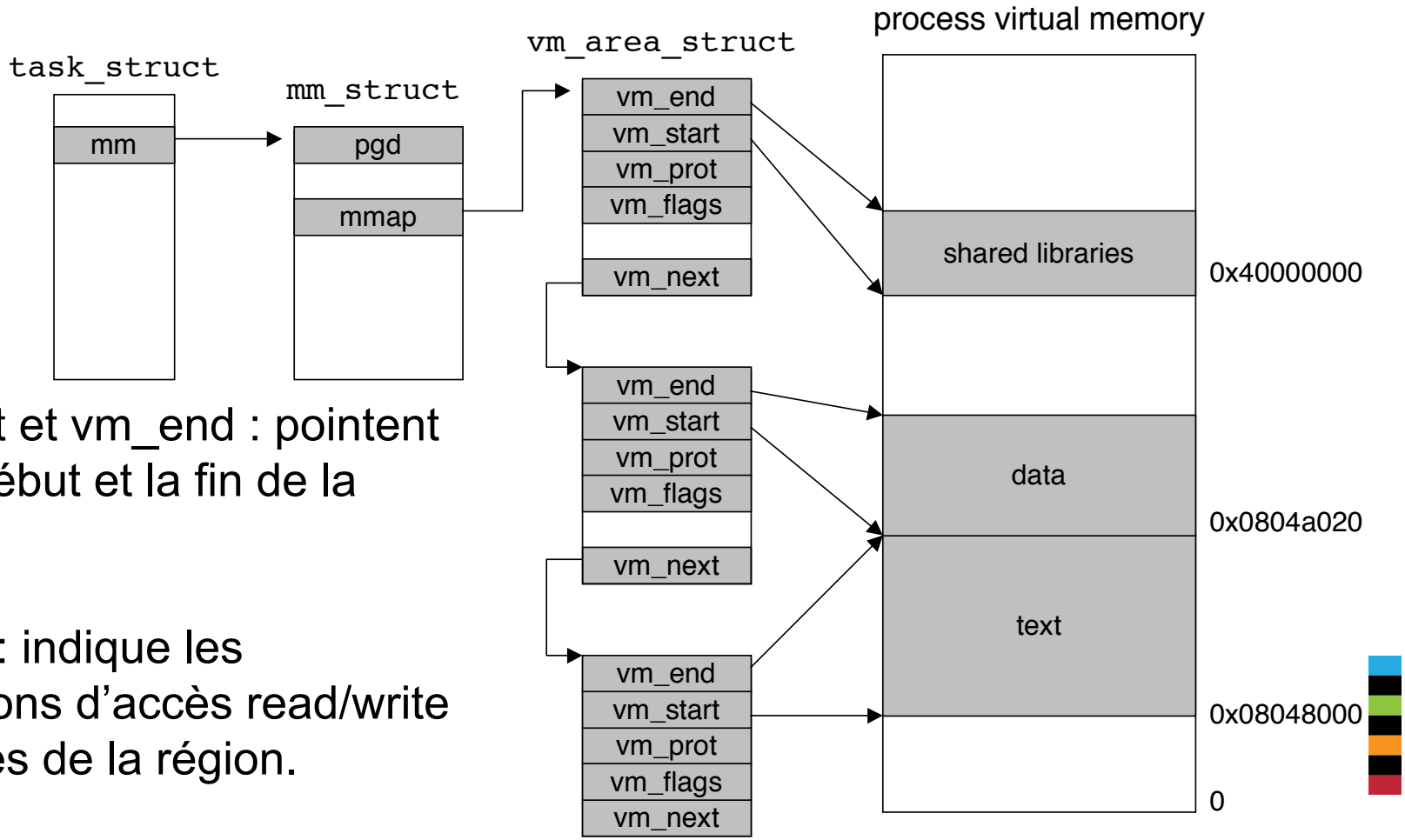


Espace d'adressage d'un processus (9)
Cas de Linux



- L'espace d'adressage d'un processus est composé d'un ensemble de régions (segments).
- Chaque région est un ensemble de pages contiguës avec les mêmes propriétés (text, data, pile, librairies partagées, fichiers mappés, etc.).

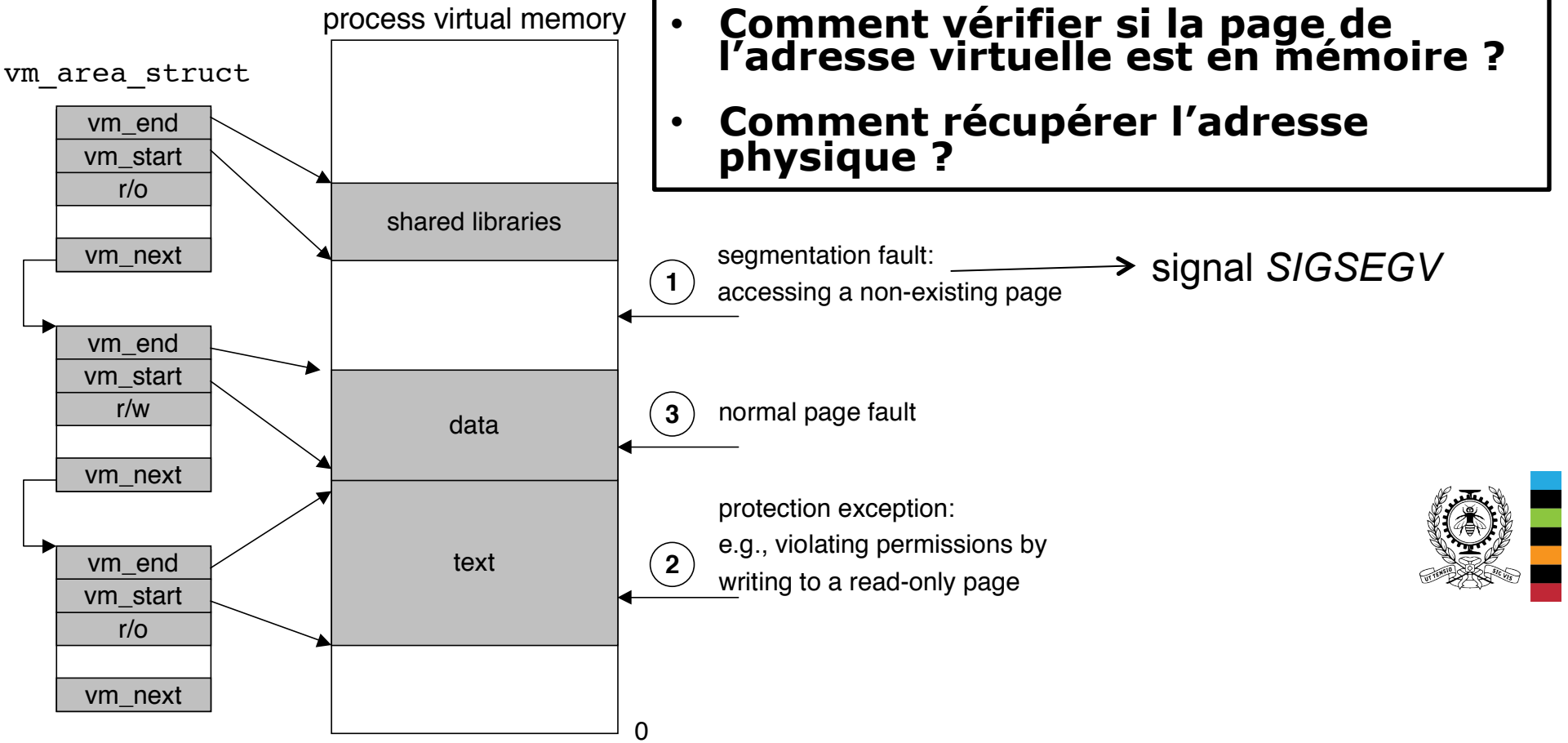
Espace d'adressage d'un processus (10)
Cas de Linux



- vm_start et vm_end : pointent vers le début et la fin de la région.
- vm_prot : indique les permissions d'accès read/write aux pages de la région.
- Vm_flags : indique notamment si la région est privée ou non.

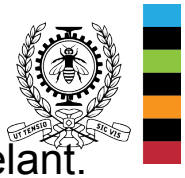
Espace d'adressage d'un processus (11)
Cas de Linux

- 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 ?**



Espace d'adressage d'un processus (12)
Cas de Linux (quelques appels système)

- fork() crée un nouveau processus avec un espace d'adressage partagé selon le principe COW (Copy-On-Write).
- execve() remplace l'espace d'adressage du processus appelant par un autre, construit à partir d'un fichier exécutable.
- exit() permet de libérer tout l'espace d'adressage du processus appelant.
- mmap() crée une région (segment) 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 de l'espace d'adressage virtuel du processus appelant.



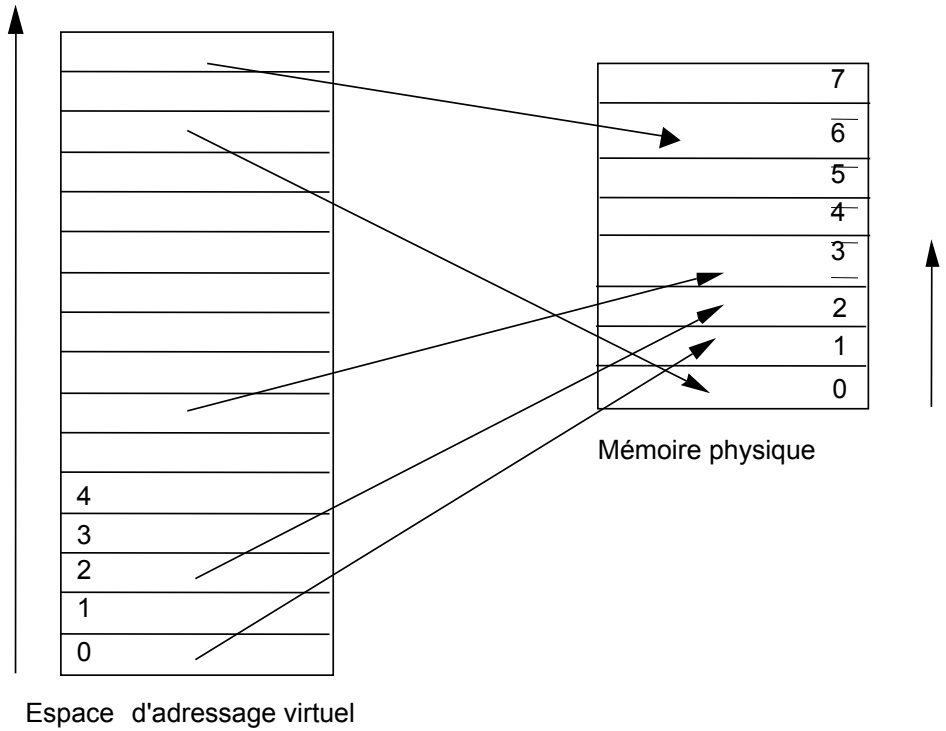
Espace d'adressage d'un processus (13)
Cas de Linux (quelques appels système)

- 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.

Pagination pure

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

int getpagesize();

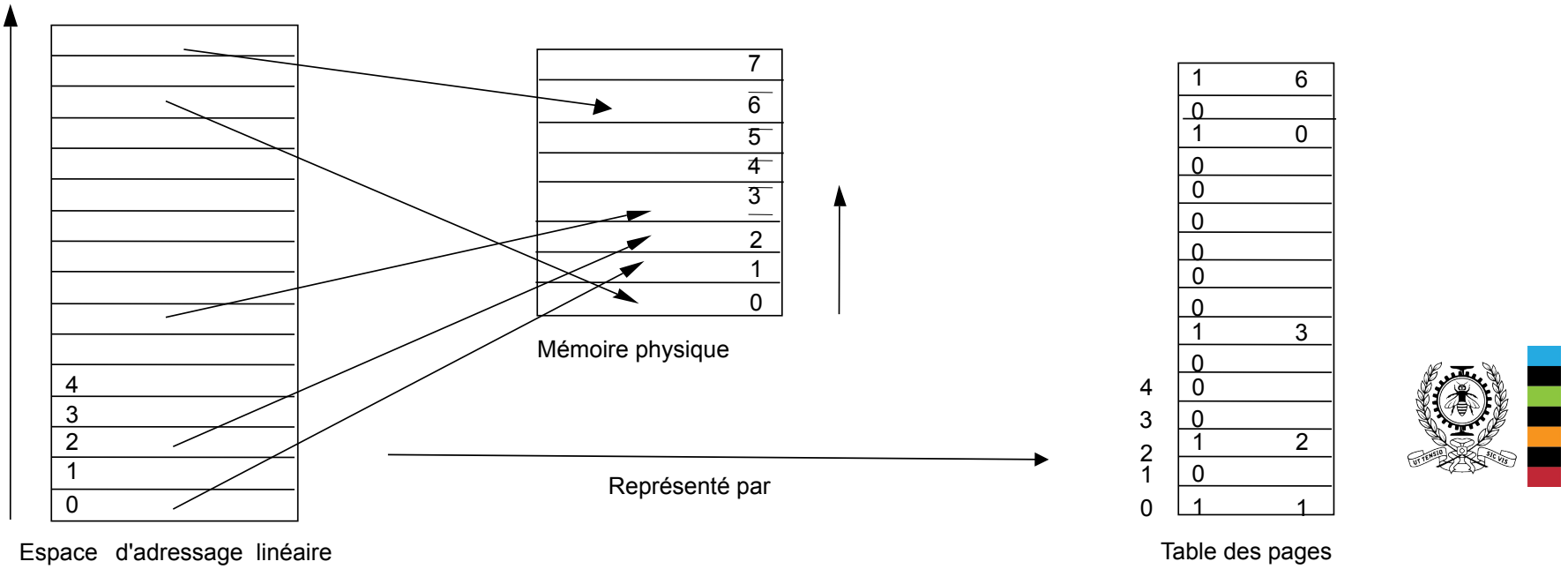


- Espace d'adressage linéaire.
- Unité d'allocation : cadre (taille d'un cadre = taille d'une page).
- Chargement en mémoire à la demande.
- **Fragmentation interne** (espace alloué mais non utilisé) pour les cadres non pleins.

Pagination pure (2)

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

Pagination pure (3)

- 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 en mémoire physique (P)
  - 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

Pagination pure (4)

Exemple de conversion d'adresse :

- 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.

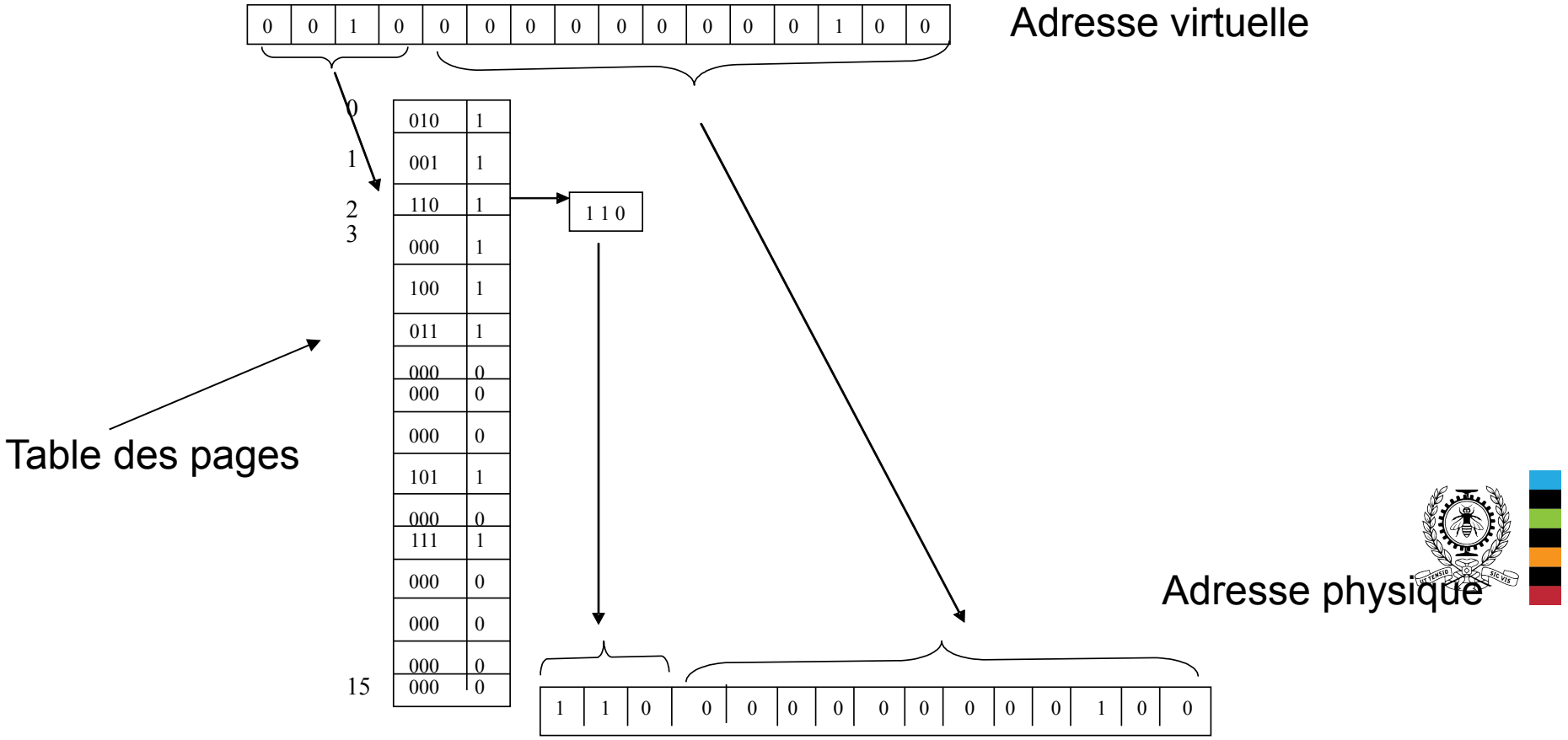
- Si le bit de présence est à 0, la page n'est pas en mémoire, il faut alors lancer son chargement en mémoire.

- 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.

Pagination pure (5)

Exemple (suite) :

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



Pagination pure (6)

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	-

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

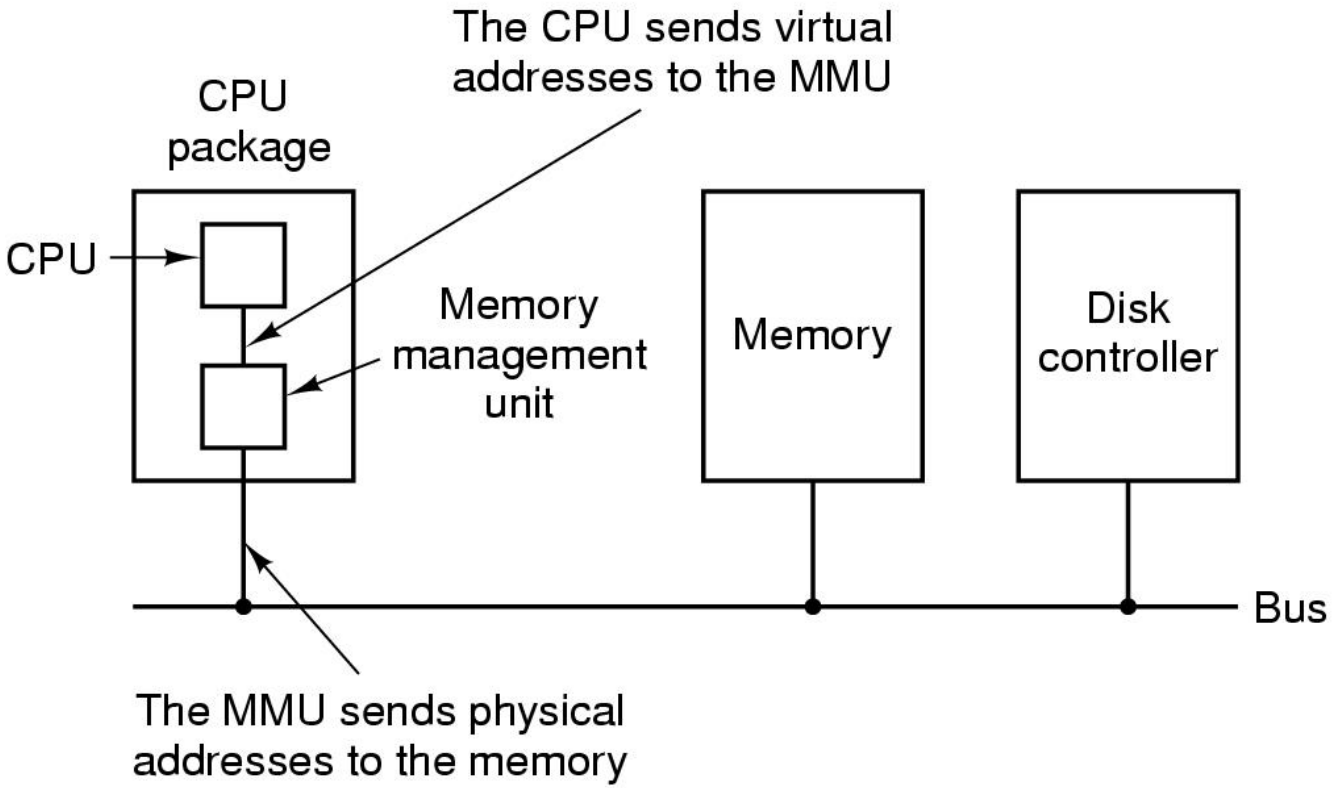
Pagination pure (7)

MMU

- La 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 lancer le chargement à partir du disque de la page manquante référencée (l'unité de transfert est la page).

Pagination pure (8)

MMU



Pagination pure (9)

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

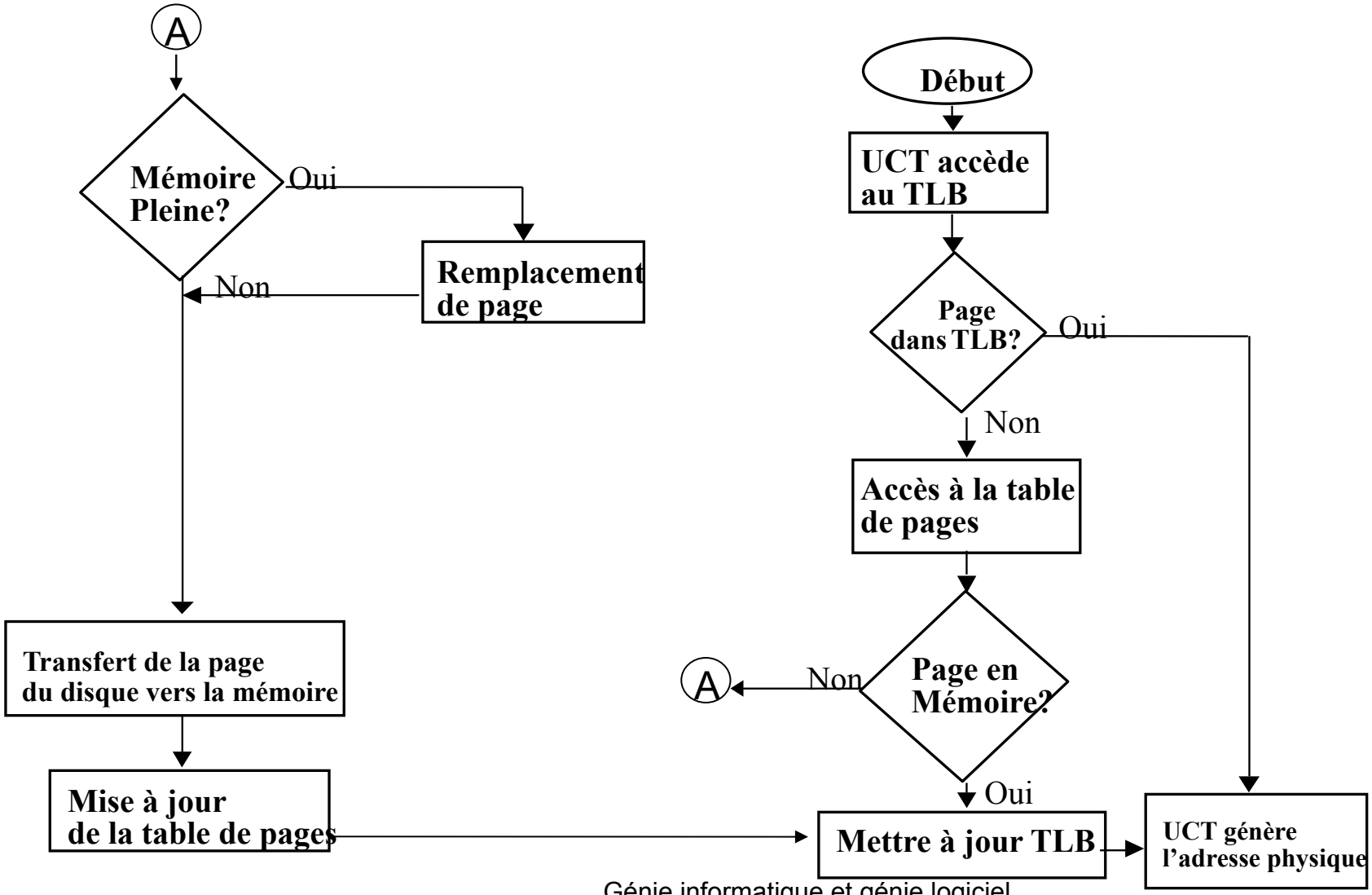
Pagination pure (10)

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.

Pagination pure (11)

MMU avec mémoire associative TLB



Pagination pure (12)

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.

Pour s=0.5, temps d'accès moyen = 60 ns

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.

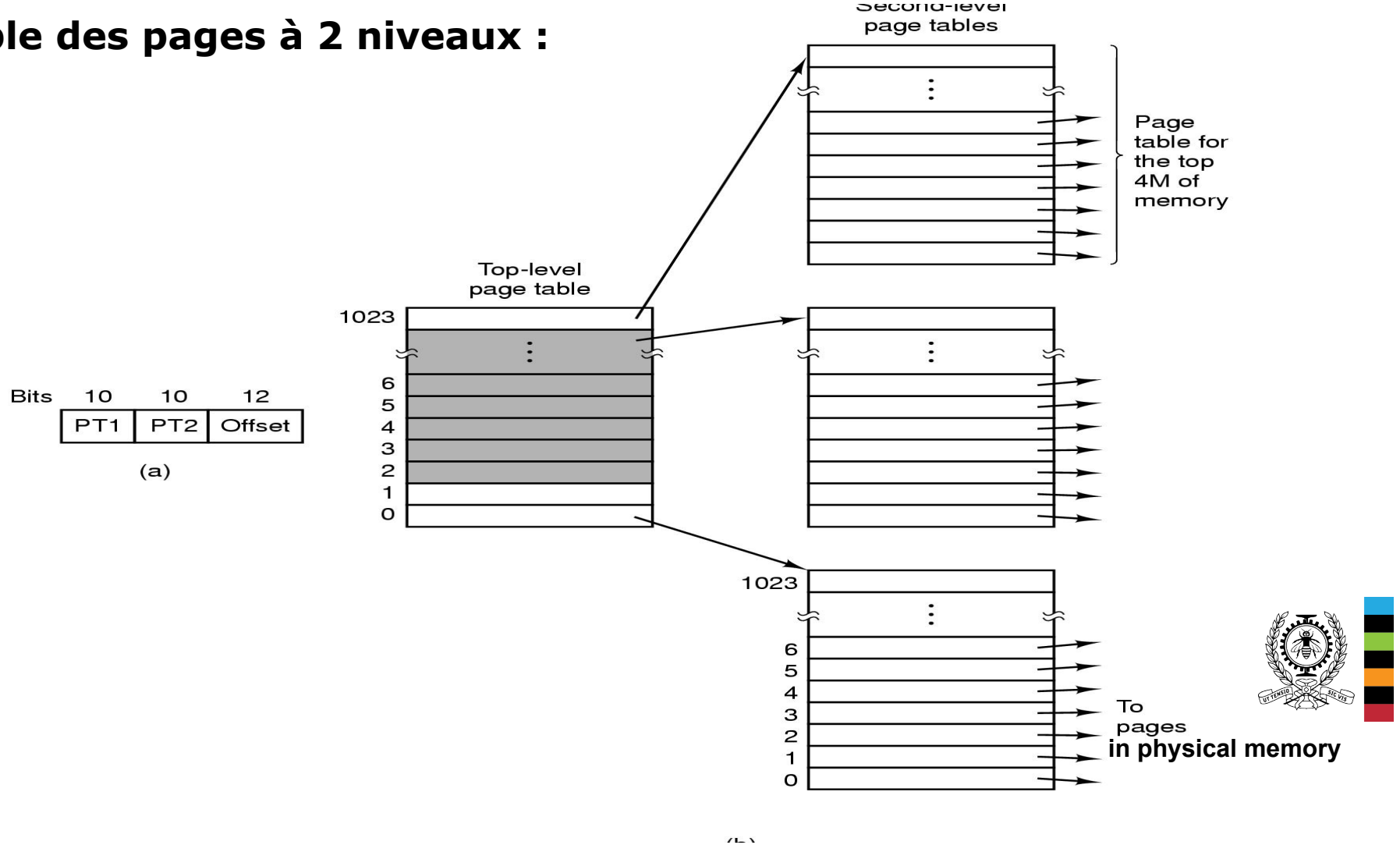
- Pour éviter d'avoir des tables trop grandes en mémoire, de nombreux ordinateurs utilisent des tables de pages à plusieurs niveaux.

- 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 en mémoire 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).

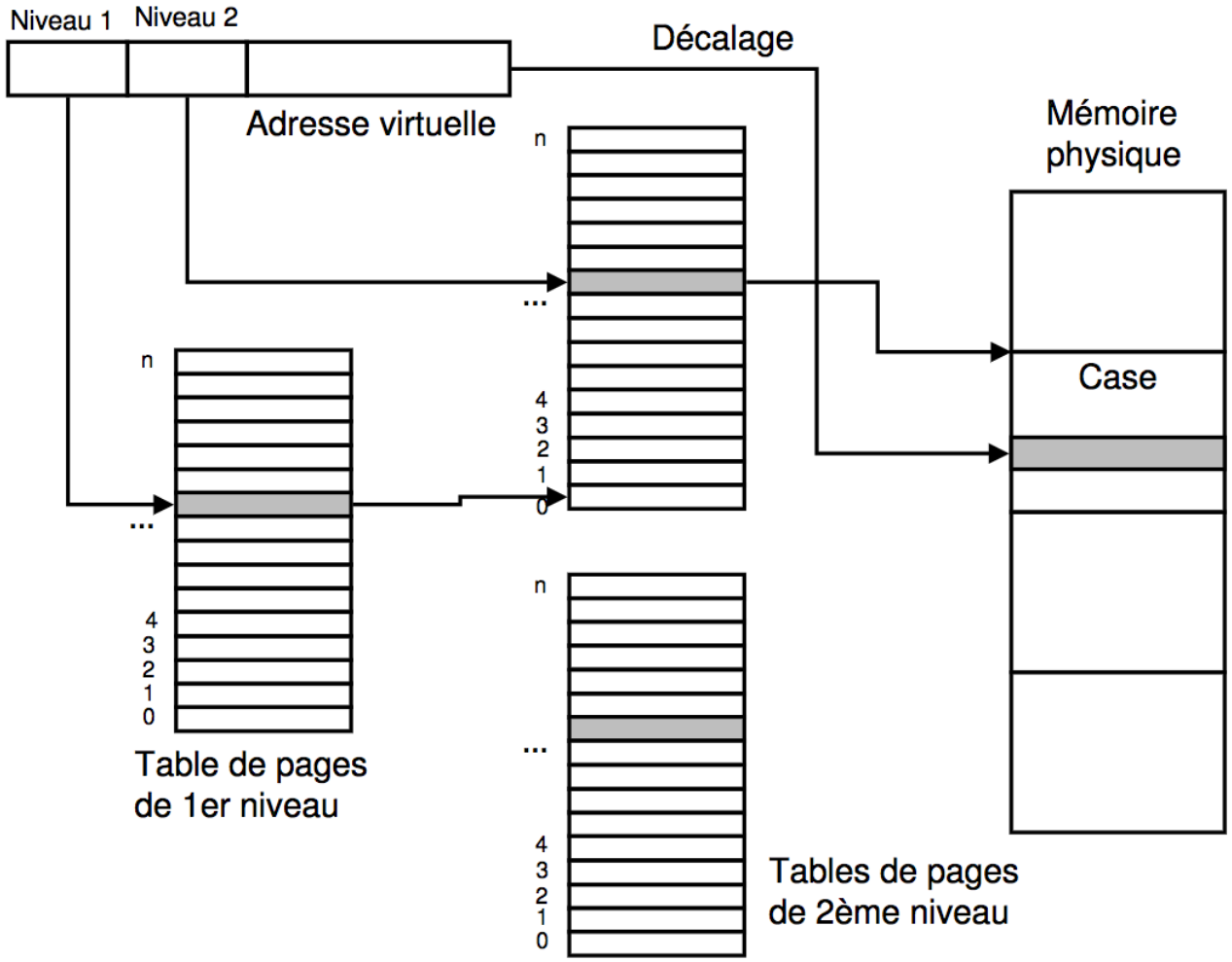
Pagination pure (14)

Table des pages à 2 niveaux :



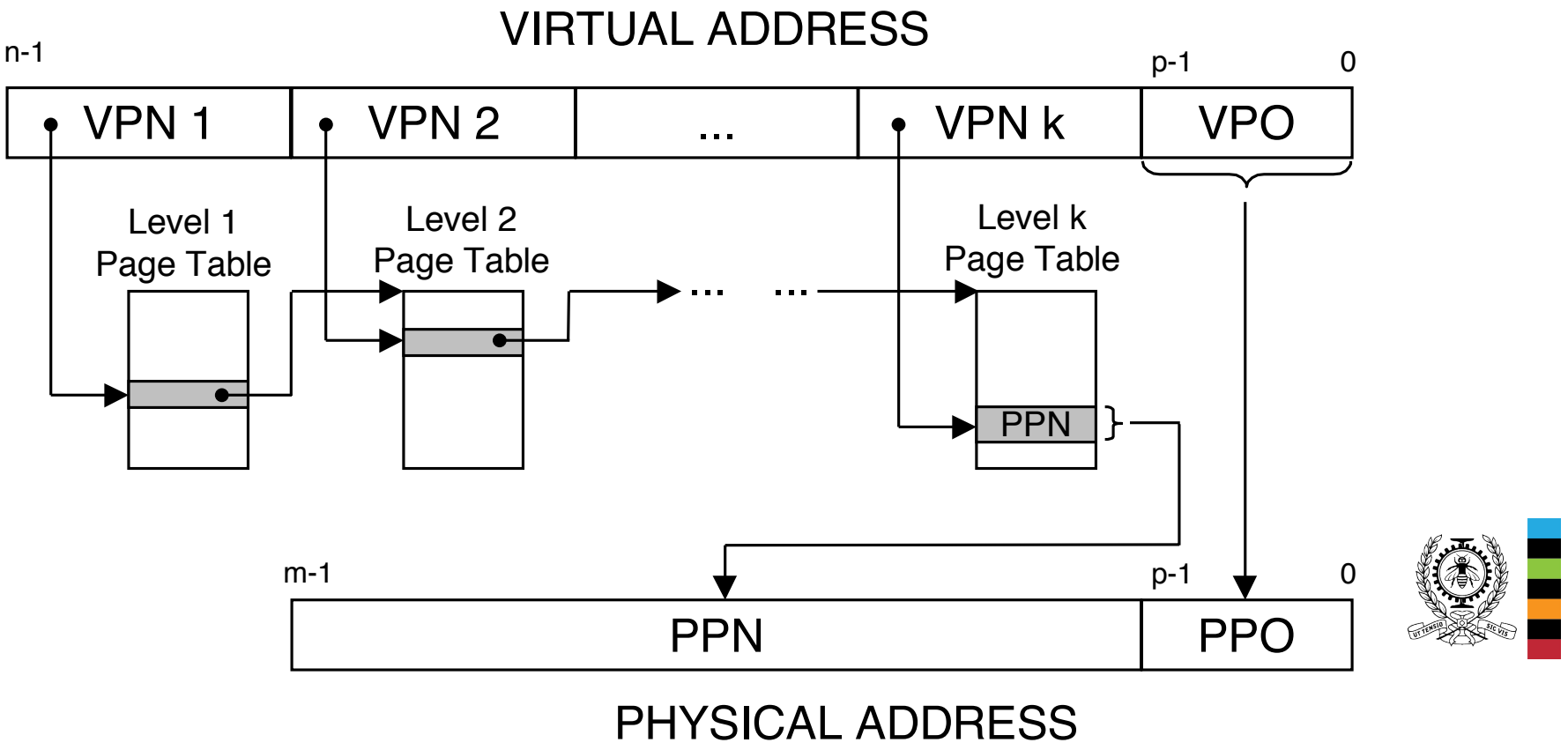
Pagination pure (15)

Table des pages à 2 niveaux :



Pagination pure (16)

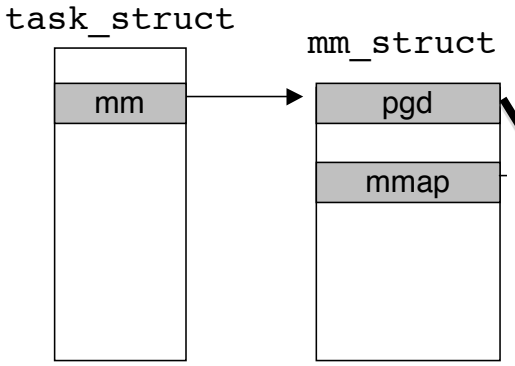
Table des pages à k niveaux

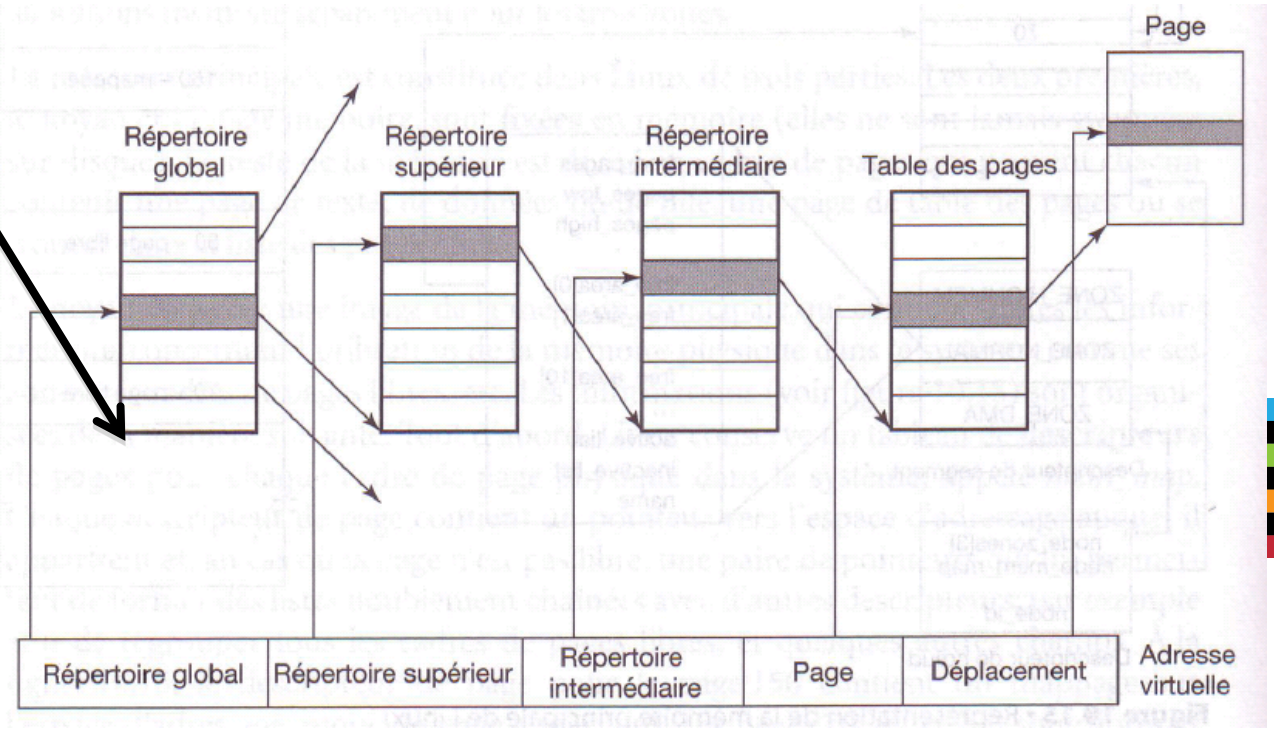


Pagination pure (17)

Cas de Linux : Table des pages à 4 niveaux

- **Comment vérifier si la page de l'adresse virtuelle est en mémoire ?**
- **Comment récupérer l'adresse physique ?**





Pagination pure (18)

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.

Pagination pure (19)

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

Pagination pure (20)

Algorithme FIFO

- Critère : remplacer la page dont le temps de résidence est le plus long
- Implémentation facile : pages résidentes en ordre FIFO (on expulse la première)
- 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	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	0	0	0	3	3	3	3	3	2	2	2	2	2	1

Nombre d'accès: 20
Fautes de page: 15 (75%)

Pagination pure (22)

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

Pagination pure (23)

Algorithme LRU

- Critère : remplacer la page résidente la moins récemment utilisée.
- Basé sur le principe de localité : une page a tendence 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	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7

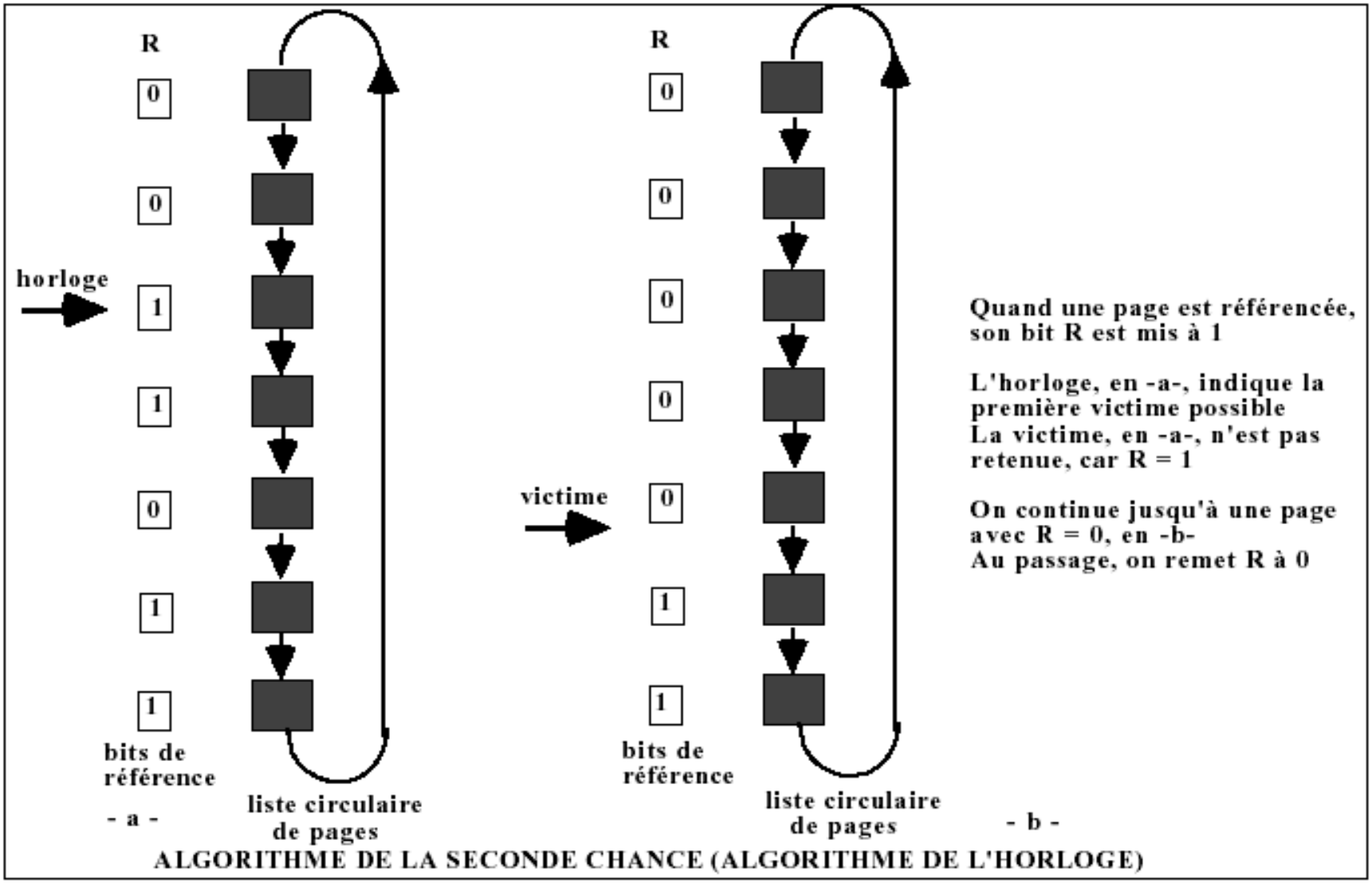
Pagination pure (24)

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.

Pagination pure (25)

Algorithme de l'horloge (seconde chance)



ALGORITHME DE LA SECONDE CHANCE (ALGORITHME DE L'HORLOGE)

Pagination pure (26)

« 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 sont 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.

Pagination pure (27)

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

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

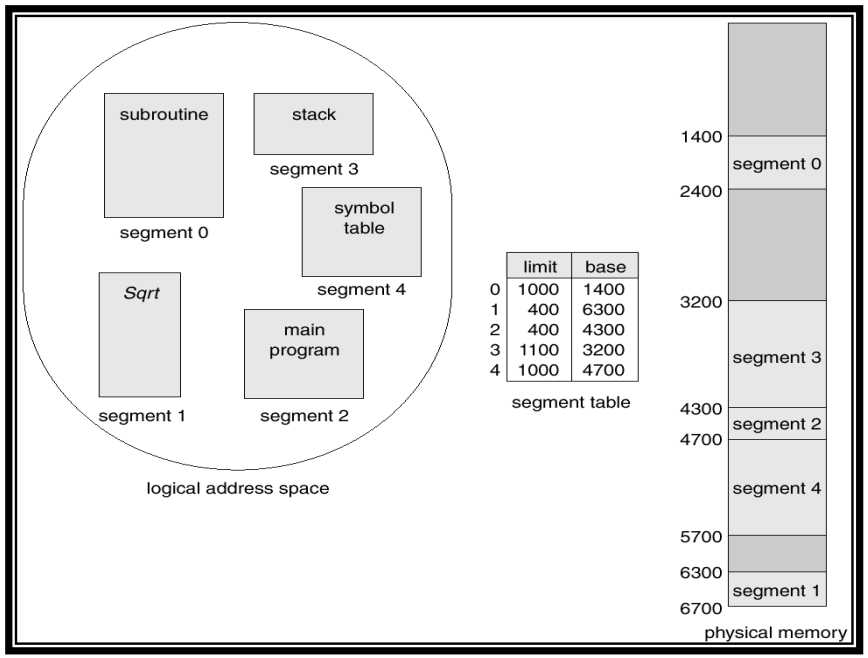
➔ Cas de Windows.

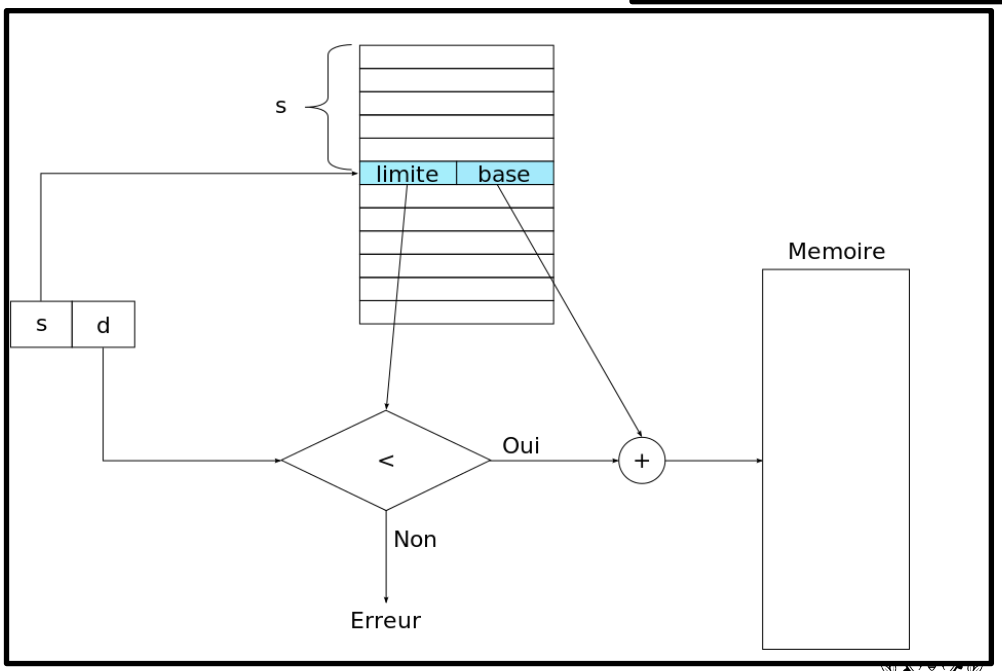
Segmentation sans ou avec pagination

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

Segmentation pure

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





- Adresse logique = (numéro de segment (s), déplacement dans le segment (d)).
- 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).

Segmentation sans ou avec pagination (2)

Segmentation avec pagination

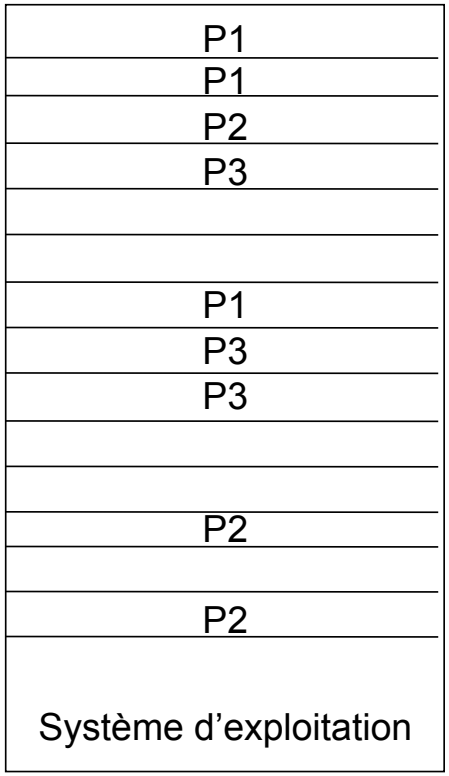
- Espace d'adressage d'un processus = ensemble de segments.
- Segment = ensemble de pages.
- On dispose dans ce cas **d'une table de segments et d'une table de pages par segment** (MULTICS).
- Adresse logique :

  (numéro de segment, numéro de page, déplacement dans la page).

- **Fragmentation externe et interne**.

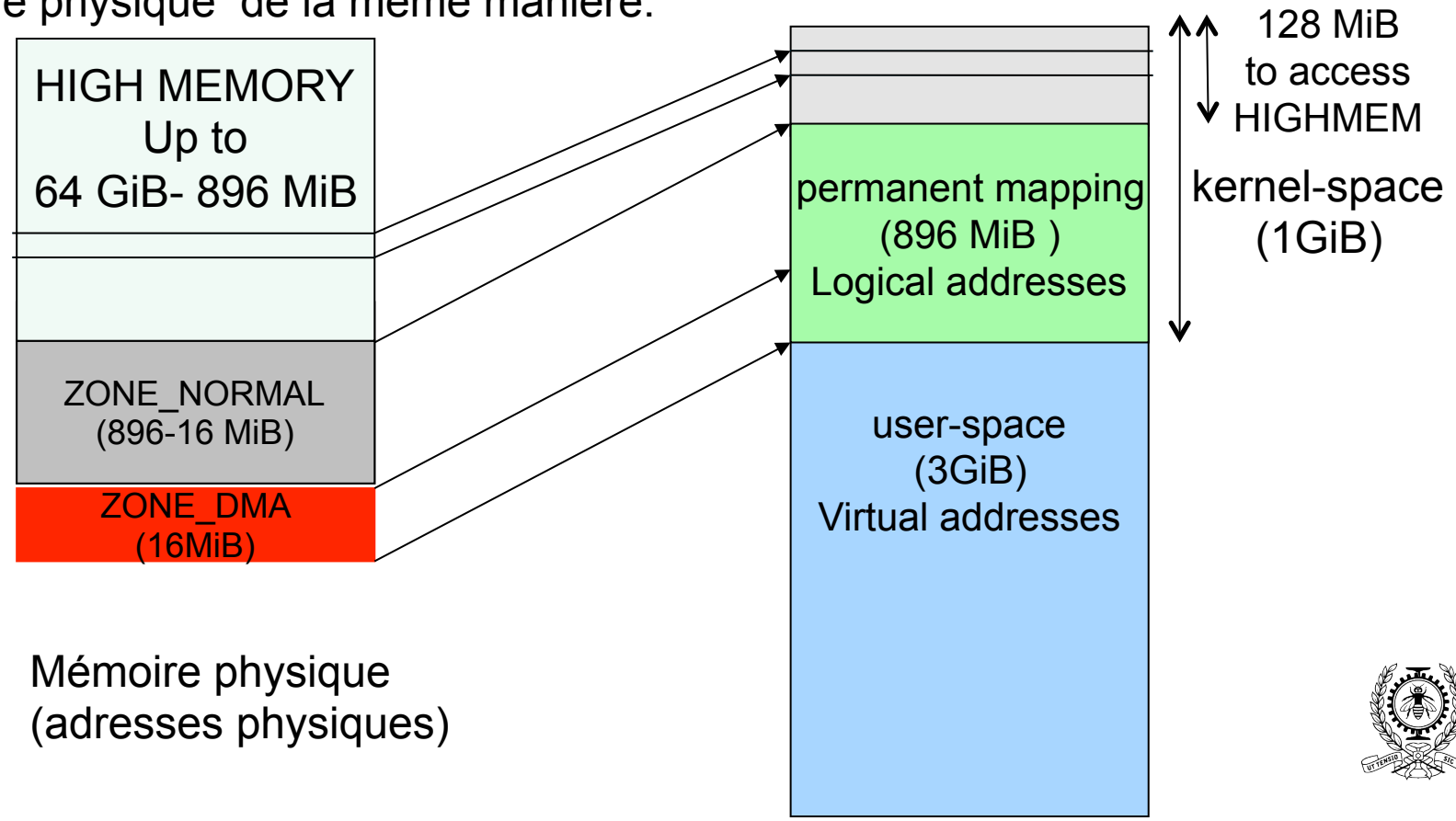
Comment organiser la mémoire physique ?

- La mémoire physique est composée d'une zone réservée au système d'exploitation (espace noyau) 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.



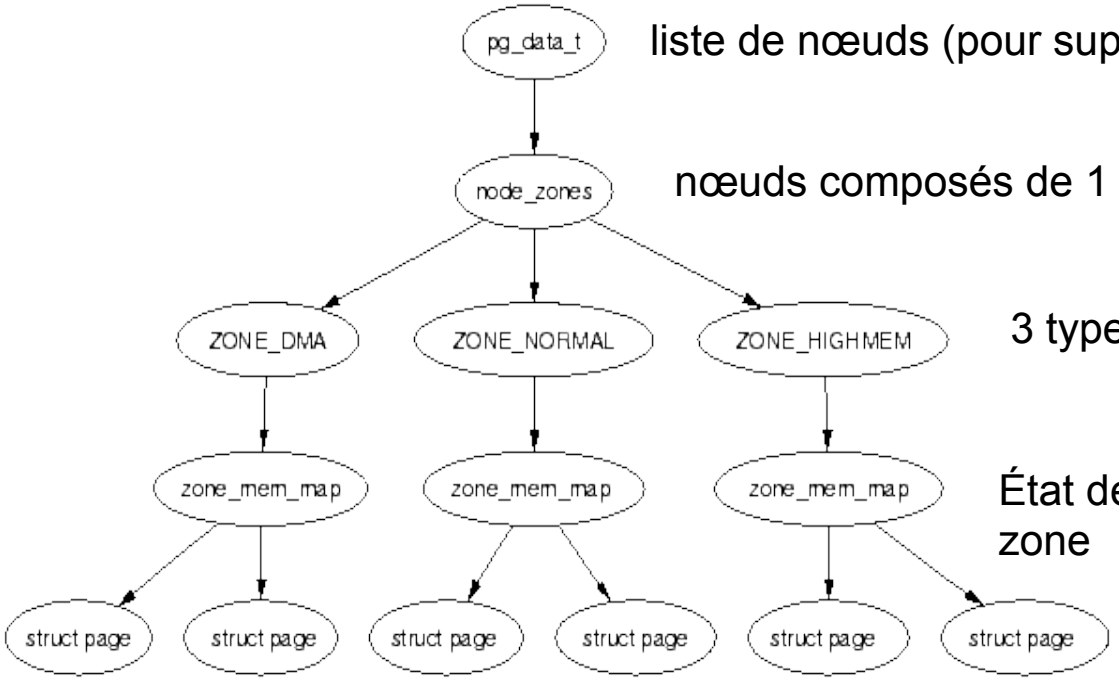
Comment organiser la mémoire physique ? (2)
Cas de Linux

- Pour certaines architectures (ex. x86), il n'est pas possible de traiter toute la mémoire physique de la même manière.



- ZONE_DMA et HIGHMEM peuvent être vides.

Comment organiser la mémoire physique ? (3)
Cas de Linux : Structures de données associées



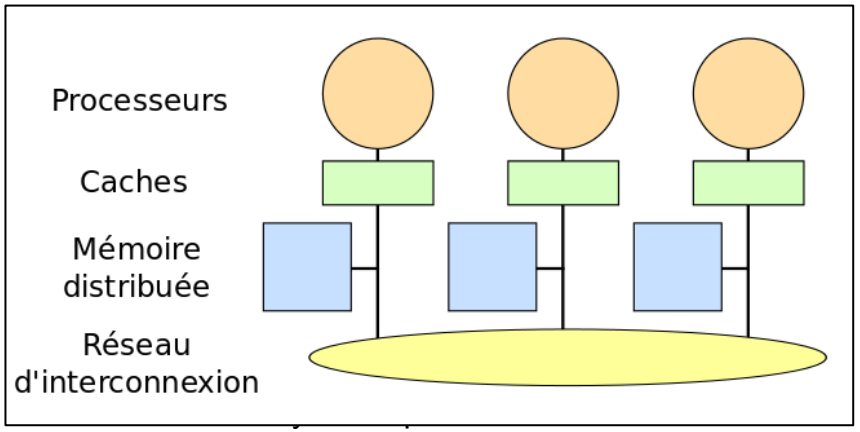
liste de nœuds (pour supporter une architecture NUMA)

nœuds composés de 1 à 3 zones

3 types de zones

État des cadres de chaque zone

struct page décrit un cadre physique



Architecture NUMA
https://fr.wikipedia.org/wiki/Non_uniform_memory_access

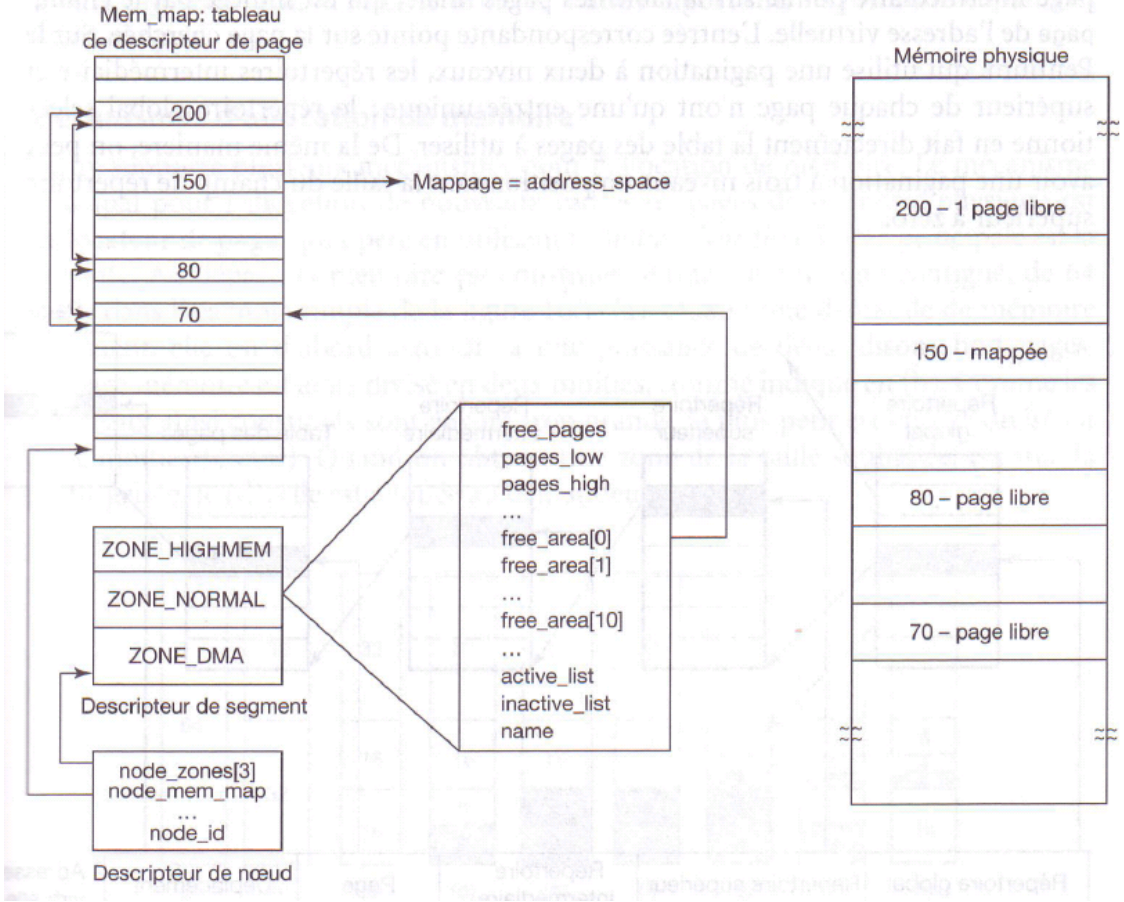
Comment organiser la mémoire physique ? (4)
Cas de Linux : Structures de données associées

- 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 struct page.
  - La structure page 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 cadres libres
struct inode *inode; unsigned long offset; // Son emplacement sur le disque
atomic_t count;   // nombre de processus partageant cette page en mémoire
unsigned flags; // indique si la page est libre, modifiée, verrouillée, etc.
….
unsigned long map_nr; // numéro de la page physique  (cadre)
} mem_map_t;
```

Comment organiser la mémoire physique ? (5)
Cas de Linux

Structures de données associées (vue d'ensemble)



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

Comment organiser la mémoire physique ? (6)
Cas de Linux

Allocateur par subdivision

La mémoire centrale est gérée comme suit :

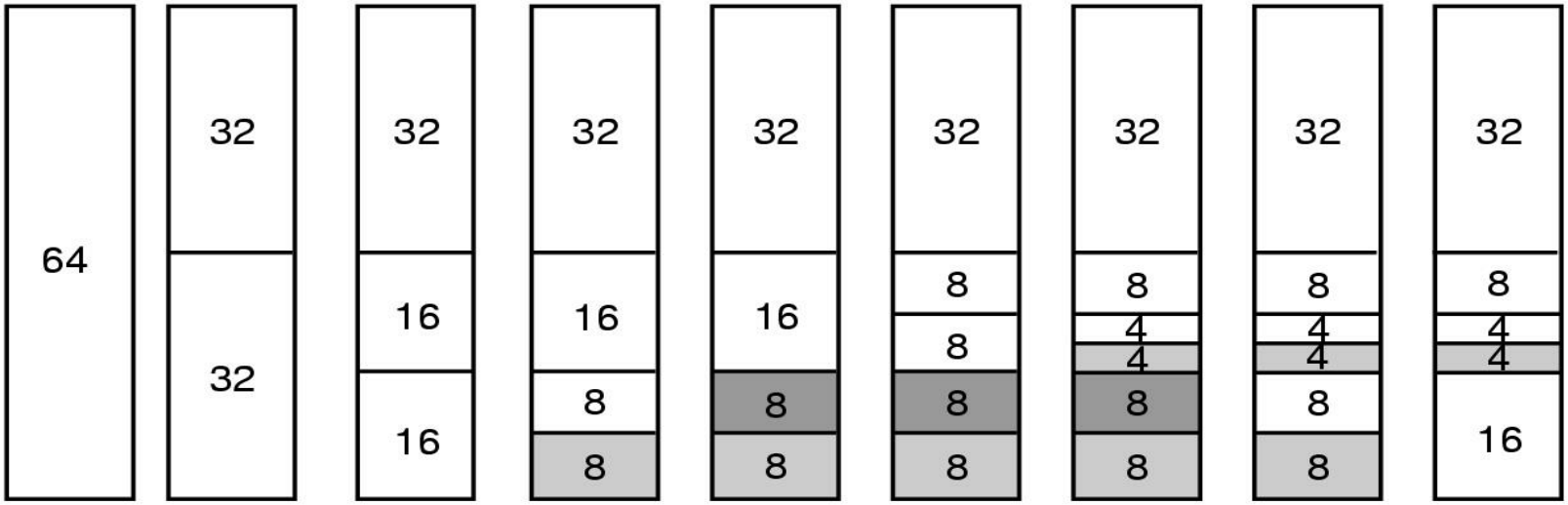
- Initialement, la mémoire est composée d'une seule zone libre.
- Lorsqu'une demande d'allocation arrive, la taille de l'espace demandé est arrondie à une puissance de 2. La zone libre initiale est divisée en deux. Si la première est trop grande, elle est, à son tour, divisée en deux et ainsi de suite... 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 (free_area(0)). Le deuxième élément contient la tête de la liste des zones de taille 2 (free_area(1)), …
- 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 ? (7)
Cas de Linux

Allocateur par subdivision



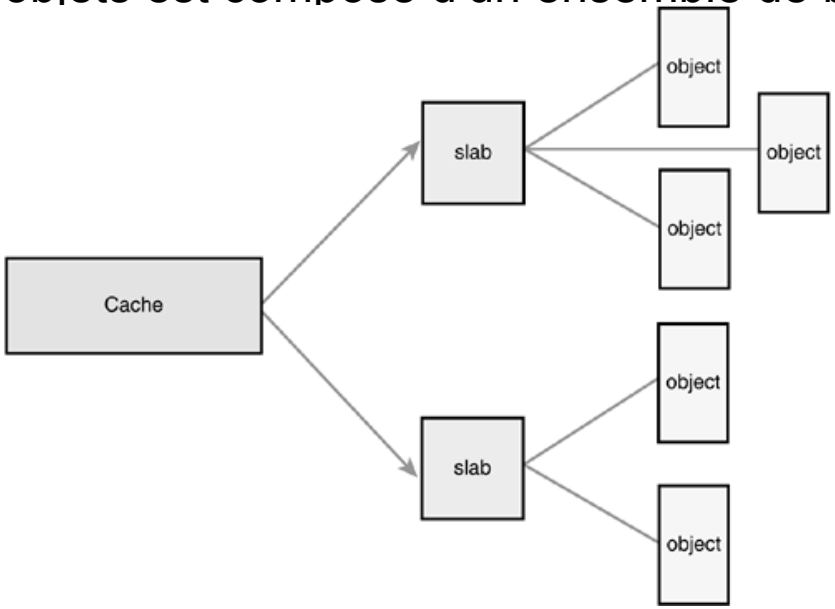
Demande de 6 pages → 8 pages
Demande de 5 pages → 8 pages
Demande de 3 pages → 4 pages
Libération de 8 pages + Libération de 8 pages → 16 pages

Tableau de têtes de listes : L'entrée i est un pointeur vers la liste de blocs libres de taille 2^i ➔ (free_area[i])

Comment organiser la mémoire physique ? (8)
Cas de Linux

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 dés le démarrage du système.
- Chaque cache d'objets est composé d'un ensemble de blocs mémoires appelés slabs.



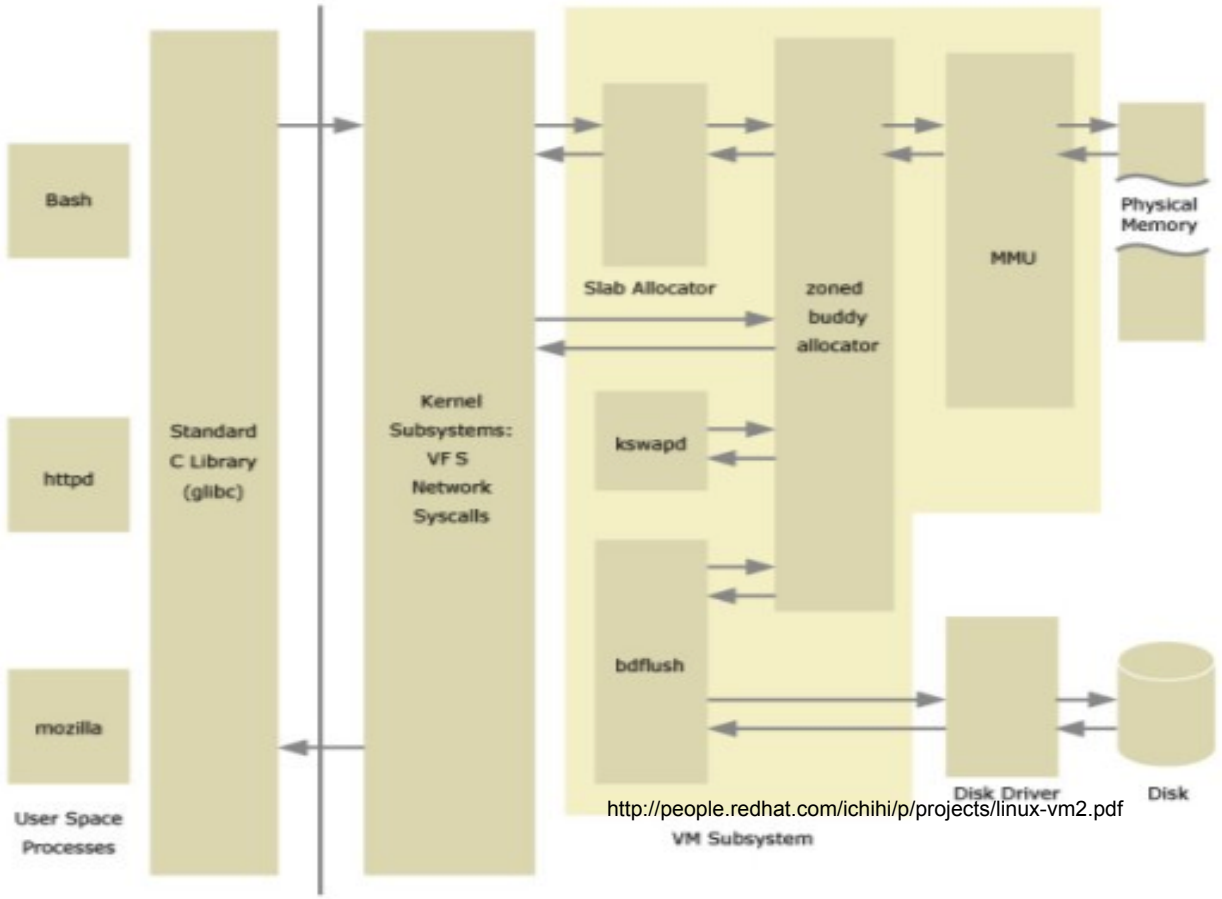
http://www.makelinux.net/books/lkd2/ch11lev1sec6

Comment organiser la mémoire physique ? (9)
Cas de Linux

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. (http://docs.oracle.com/cd/E19253-01/816-5180/6mbbf02cc/index.html)

Cas de Linux (vue d'ensemble)



http://people.redhat.com/ichihi/p/projects/linux-vm2.pdf

- kswapd (démon de pages) = algorithme de remplacement de pages.
- bdflush (démon, buffer-dirty-flush) qui gère la copie des blocs de fichiers modifiés --> pdflush
- Allocateurs Slab et par subdivision

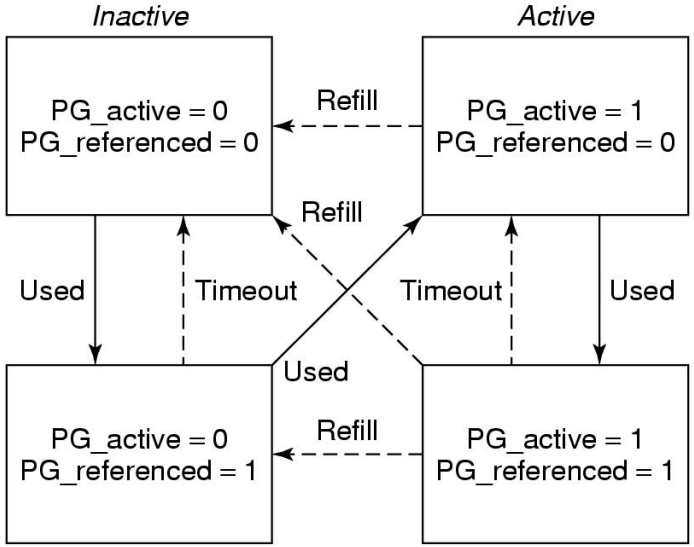
- Un système de pagination à la demande, sans pré-pagination ni concept d'ensemble de travail, allocation globale.

Cas de Linux (vue d'ensemble) (5)

- Au démarrage du système, 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.



Évolution de l'état d'une page en mémoire

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

(Livre sur les SE de Bort LAMIROY, Lourent 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.

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

(Livre sur les SE de Bort LAMIROY, Lourent 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 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, …)

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

(Livre sur les SE de Bort LAMIROY, Lourent 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,…)

Annexe : Exemple 3
Alignement de la mémoire

(Livre sur les SE de Bort LAMIROY, Lourent 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));
```

Annexe : Exemple 3
Alignement de la mémoire

(Livre sur les SE de Bort LAMIROY, Lourent 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

```
Taille de S1 (double,short,char,char)= 16
Champ       Offset     Taille
1              0        8
2              8        2
3             10        1
4             11        1
```

```
Taille de S2 (char,short,double,char)= 24
Champ    Offset     Taille
1              0        1
2              2        2
3              8        8
4             16        1
```

Annexe :
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
```