Cours de systèmes d'exploitation centralisés 2

1. Gestion de la mémoire principale
2. Gestion de la mémoire virtuelle
3. Pagination à la demande
4. Gestion de la mémoire secondaire
5. Systèmes de gestion des fichier(SGF)

Cours de systèmes d'exploitation centralisés

Chapitre 1

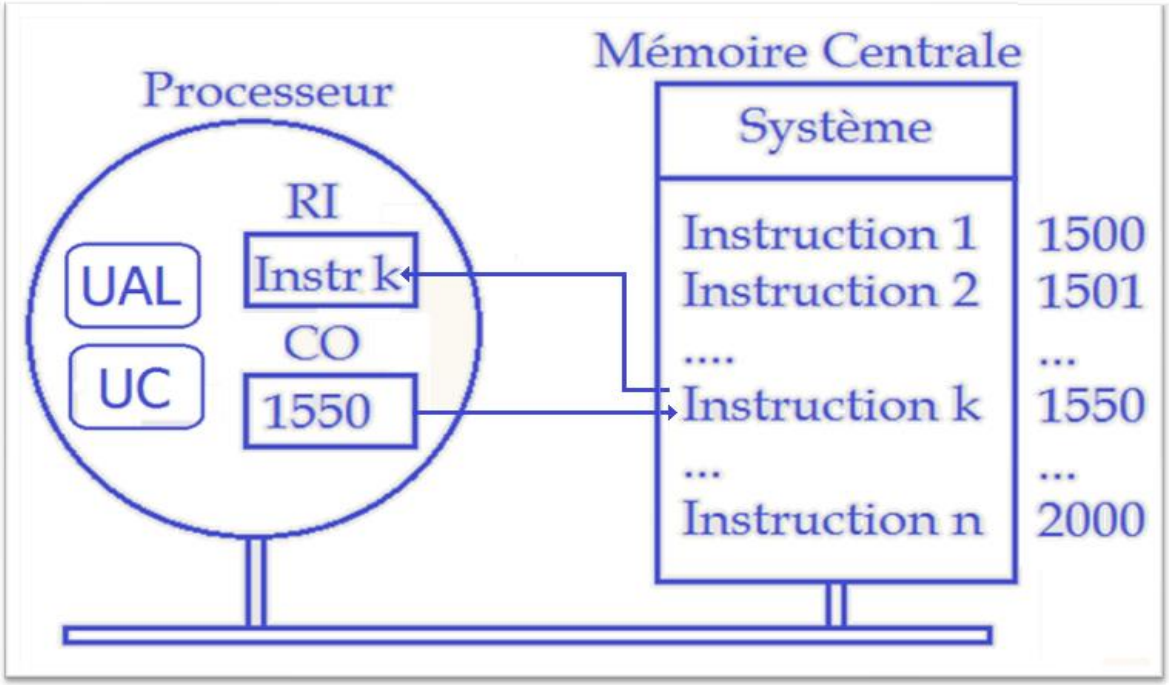
Gestion de la mémoire principale

N.TEMGLIT & M.RAHMANI

Gestion de la mémoire principale

1. INTRODUCTION

- Tout processus a besoin d'espace mémoire pour y charger son programme et ses données.
- Le processeur prélève les instructions à exécuter à partir de la mémoire centrale.



Qui charge le programme en mémoire centrale ?

- C'est un module appelé **chargeur**.
- Le programme chargé peut être :
  - Un module exécutable absolu (ex: fichiers *.com* de DOS), dans ce cas le chargement s'effectue à des emplacements fixes de la mémoire,
  - Un module exécutable translatable (ou relogeable) dans ce cas, le programme peut subir une opération de translation des adresses.
- Selon la technique de gestion mémoire utilisée, le programme peut être chargé, en MC : entièrement ou partiellement.

1.1 Gestionnaire de la mémoire

- C'est un module système dont la fonction est de gérer la mémoire principale ➔ Il doit :
  - Garder, en permanence, des informations concernant l'espace libre et l'espace occupé de la mémoire.
  - Allouer de la mémoire aux processus qui en font la demande.
  - Récupérer la mémoire libérée par un processus.
  - Protéger les espaces(codes et données) des différents processus.
  - Permettre, éventuellement, le partage de la mémoire.

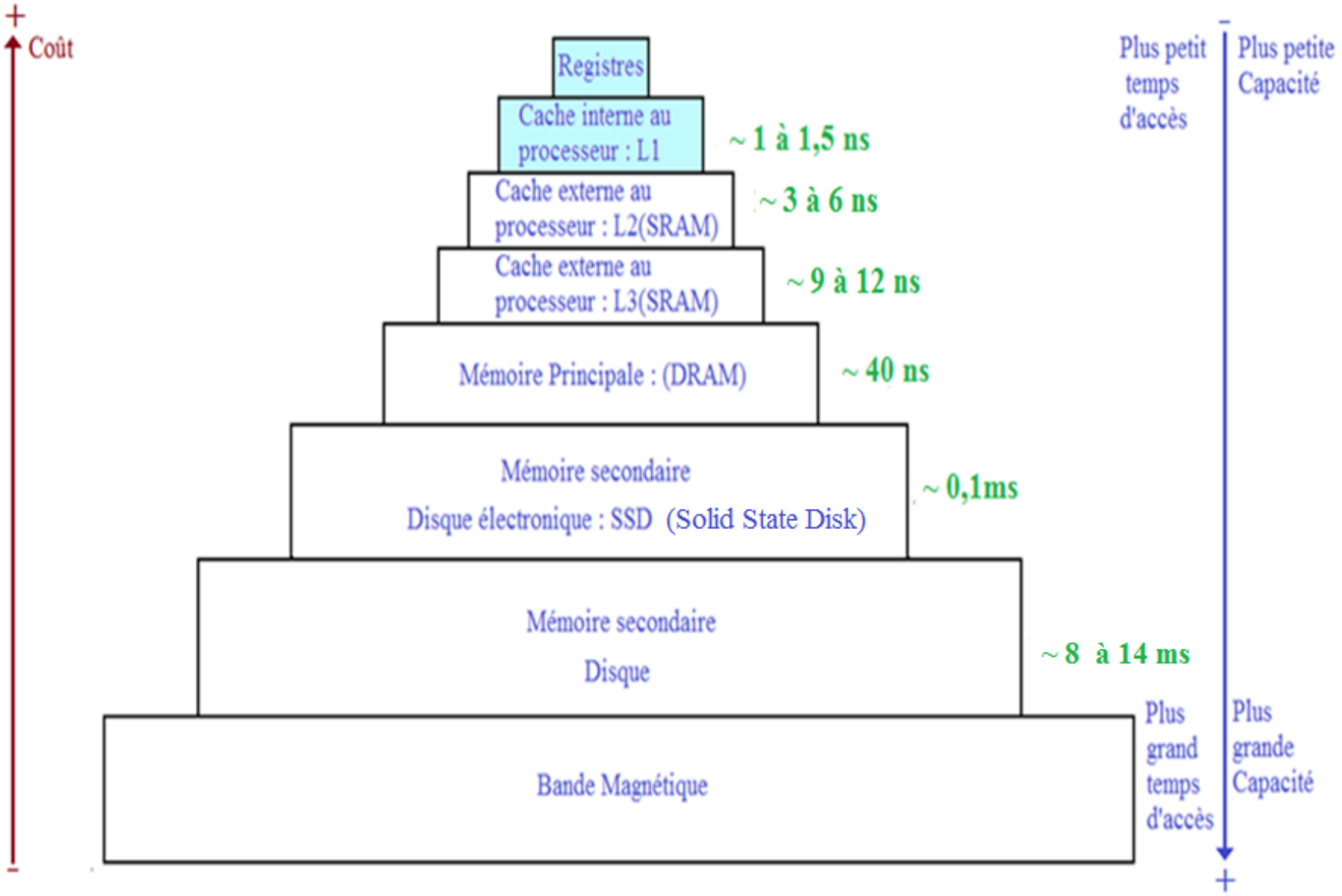
- Le gestionnaire ne peut pas charger tous les processus en mémoire centrale ➔
  - Décharger un programme (ou une partie d'un programme) de la mémoire principale et
  - Charger un autre programme (une partie ou tout le programme).

  - **Remarque** : Les programmes déchargés de la mémoire principale sont sauvegardés sur disque (mémoire secondaire ou auxiliaire).

1.2 Hiérarchie des mémoires

- Les supports de stockage d'information utilisés sont très différents :
  - Capacités de stockage et temps d'accès très hétérogènes.
  - En général, plus le temps d'accès à un support est rapide, plus sa capacité est faible.
  - Cependant, le coût du support est inversement proportionnel au temps d'accès: plus un support est lent, moins il est cher.

- La hiérarchie des mémoires consiste à organiser les supports de stockage par temps d'accès croissants ou par taille croissante.



Hiérarchie des Mémoires

a) Objectifs de la hiérarchie des mémoires :

- Offrir à l'utilisateur l'espace de la mémoire la plus grande avec le temps d'accès de la mémoire la plus rapide.
- Accéder le plus rapidement possible aux informations (données ou instructions) les plus utilisées.

b) Comment satisfaire ces objectifs ?

Conserver à tout instant l'information la plus utilisée dans la mémoire la plus rapide.
Cette mémoire est appelée mémoire cache.

- L'efficacité de cette technique dépend de la localité des programmes et des données.

1.3 Propriété de localité

- Localité c'est l'ensemble des zones (pages) mémoire d'un programme qui sont actives en même temps.
- On distingue deux types de localités:

1. Localité temporelle :

Une adresse mémoire qui a été référencée récemment, a de forte chance d'être référencée prochainement (dans un futur proche).

➔ Garder en mémoire cache les dernières données/instructions référencées par un processus.

Exemples :

- Boucle, - Variable utilisée comme compteur, - Pile.

2. Localité spatiale :

Les références sont concentrées dans une région mémoire: si une adresse vient d'être référencée, il y a de forte chance qu'une adresse voisine soit référencée. ➔

Charger à l'avance les données/instructions contiguës à une donnée/instruction référencée.

- **Exemples** : - Parcours d'un tableau, - Exécution séquentielle d'une séquence d'instructions ou d'une Fonction,

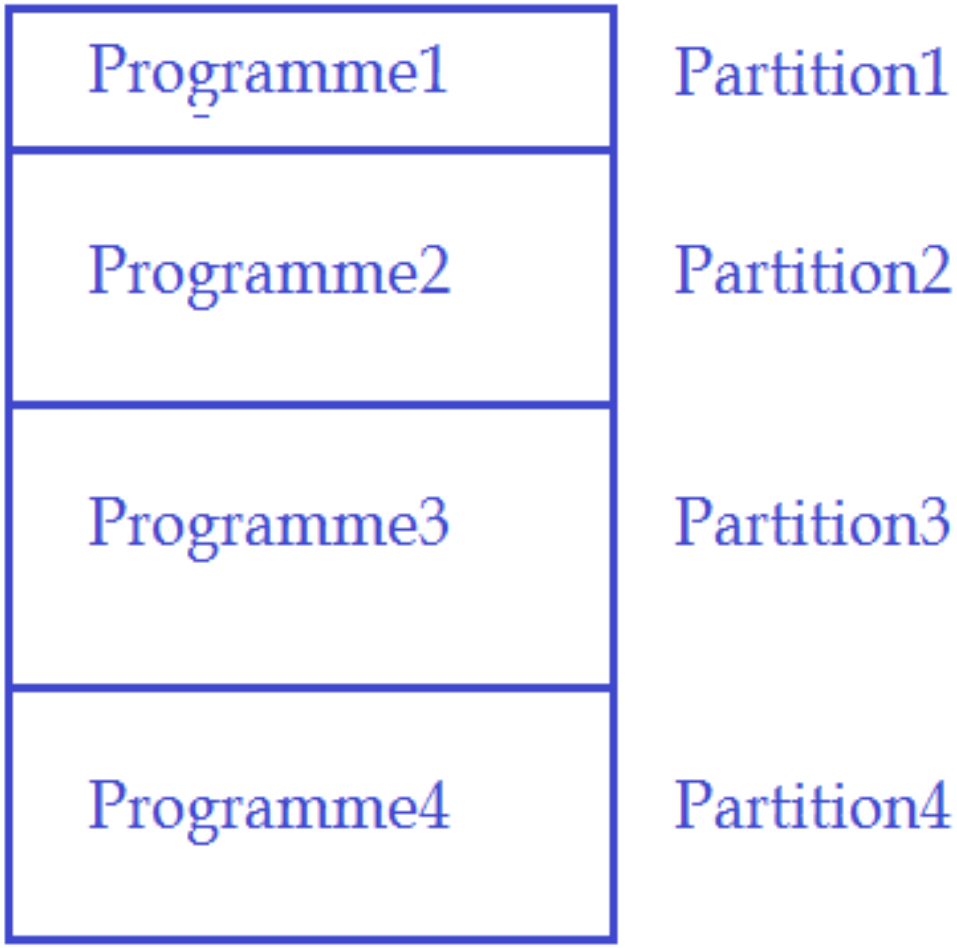
- **Contres exemples:** - Table en accès direct, - Structure avec allocation dynamique, - If … goto dans un grand programme.

2. Allocation contiguë de la mémoire principale dans les systèmes multiprogrammés

- La multiprogrammation permet de charger plusieurs programmes utilisateurs en mémoire centrale.

2.1 La technique des partitions fixes

- Dans un système à partitions fixes, la mémoire est partagée de manière statique en un **nombre fixe de zones ou partitions**.
- Les tailles et les adresses début de ces partitions sont définies lors de la « *génération du système* » ou bien par l'opérateur, au moment du chargement du système.
- Cette technique permet la coexistence de plusieurs programmes en mémoire centrale.



- Pendant son exécution un programme ne doit pas accéder en dehors de sa partition.

2.1.1 Protection et translation

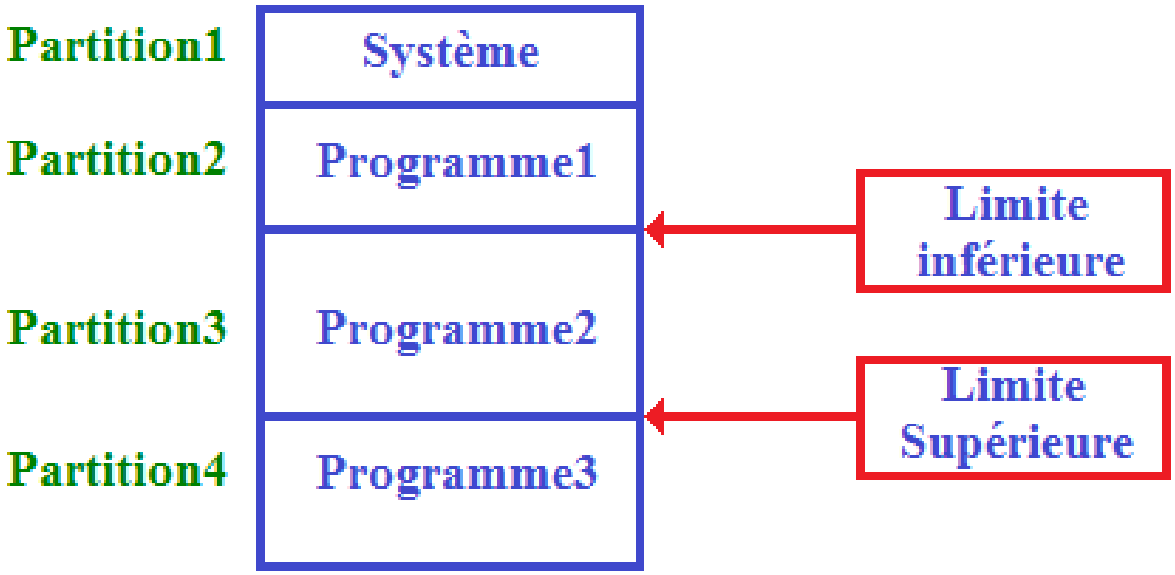
- Coexistence du système et de plusieurs programmes en mémoire principale ➔ Problème de protection du système et des programmes utilisateurs.

- La protection des programmes est réalisée au moyen de deux registres qui définissent :
  la limite inférieure et la limite supérieure des adresses qui peuvent être utilisées par un programme.

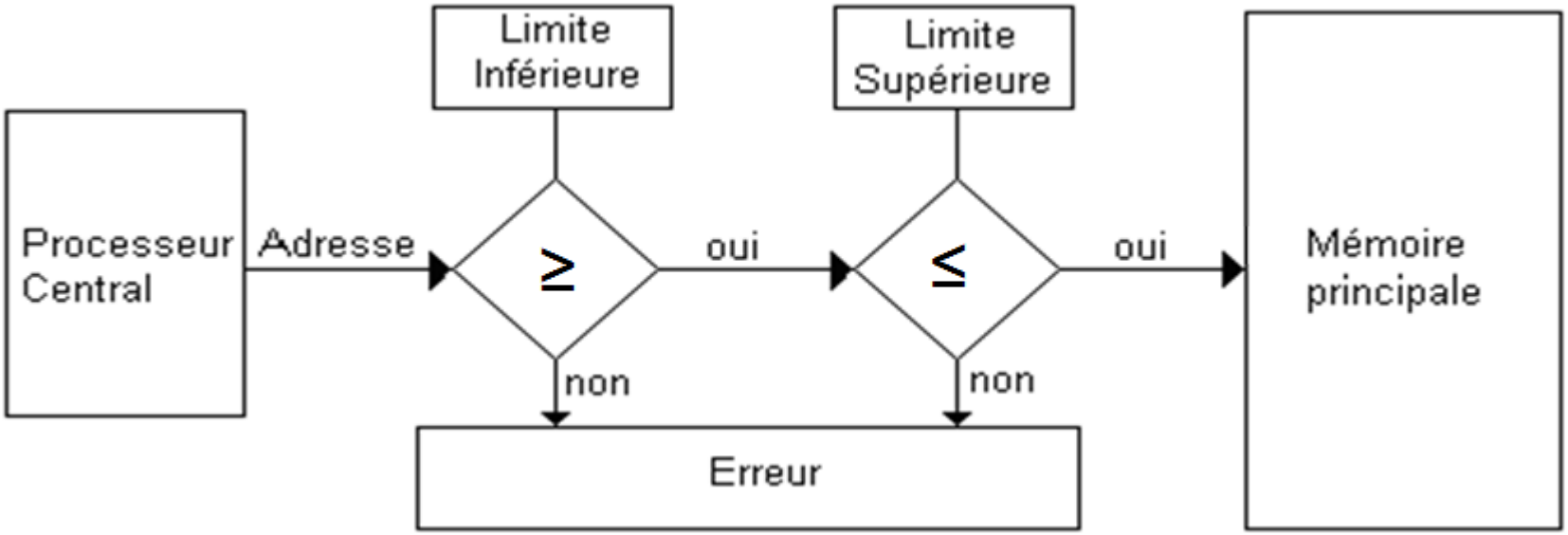
- Ces limites peuvent être définies de deux manières :

1. Registres limite

- La plus petite et la plus grande adresse de la partition sont chargées dans des registres limite.



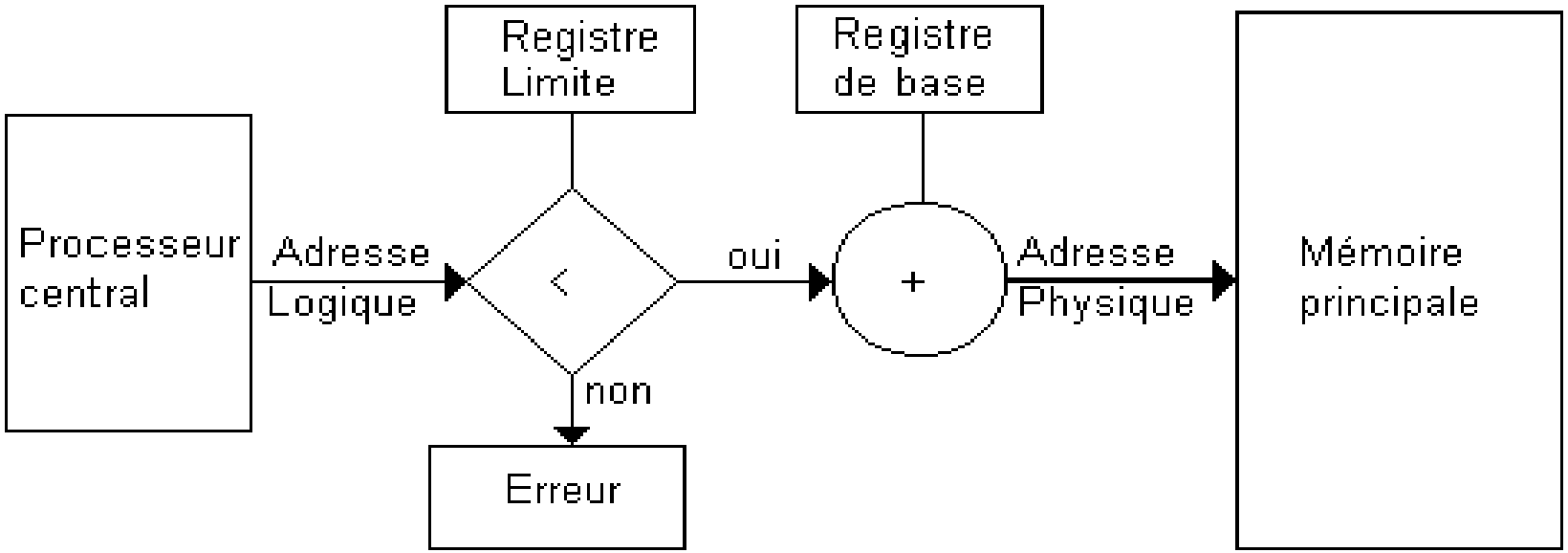
- Toutes les adresses de l'utilisateur sont comparées aux registres limite.



- La translation des adresses est faite de manière statique à l'assemblage/compilation ou au chargement du programme.

2. Registre de base et registre limite

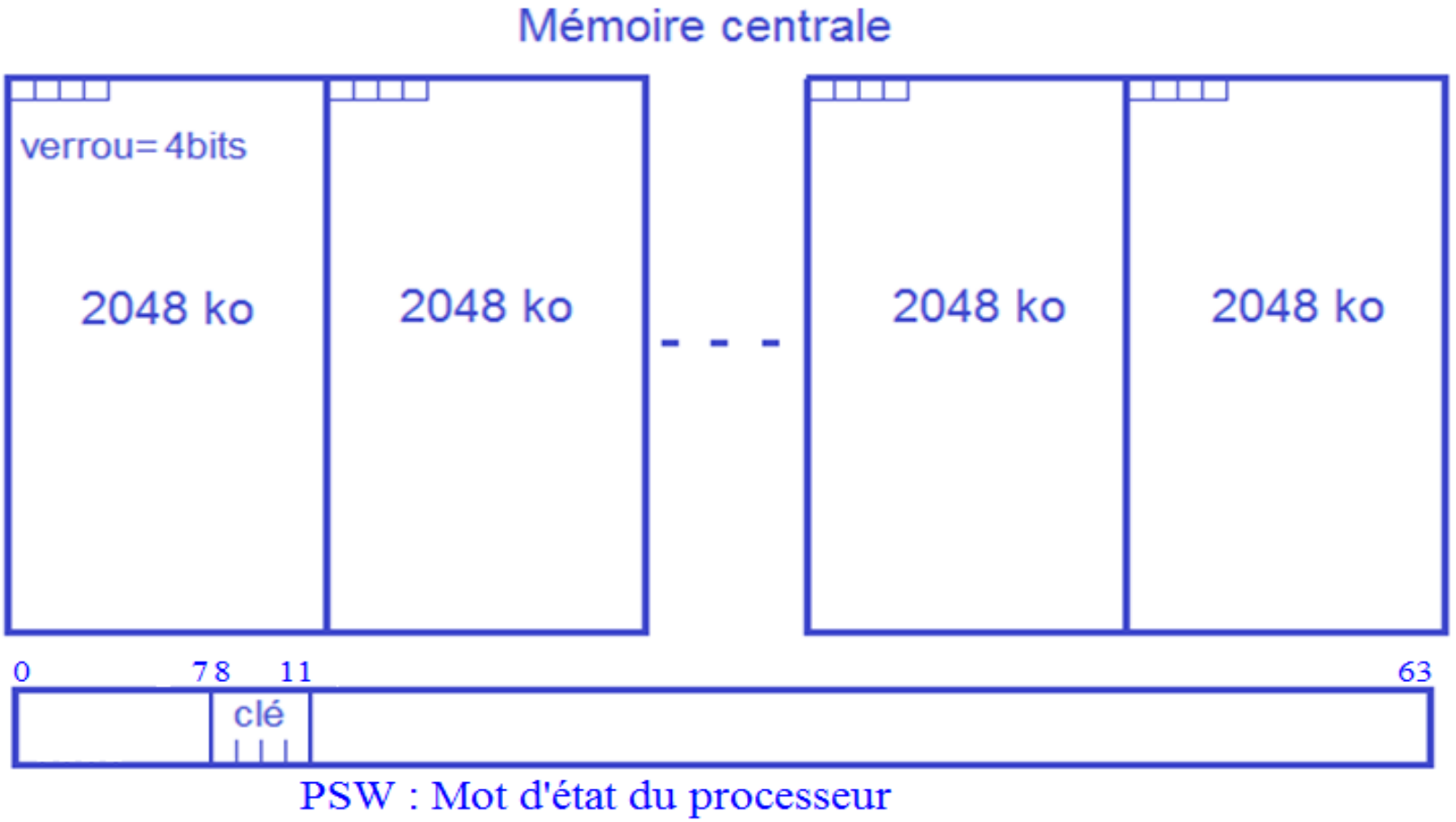
- La translation des adresses est faite au moment de l'exécution du programme c'est ce que l'on appelle **translation dynamique**.
- Les adresses logiques du programme sont comparées seulement au **registre limite.**
- **Adresse physique = adresse logique + registre de base**



- Cette technique a été utilisée par le CDC6600 et sa descendance.

- **Exemple : L'IBM/360**
  - La technique des partitions fixes a été utilisée dans le système OS/360-MFT d'IBM (MFT : **M**ultiprogramming with a **F**ixed number of **T**asks).
  - La mémoire principale est divisée en blocs de 2048 octets.
  - Chaque bloc possède un code de protection(verrou) composé de 4bits.
  - Le mot d'état du processeur (PSW) contient une clé constituée de 4bits.
  - Lors de l'exécution, à chaque accès mémoire, on compare la **clé** au **code de protection**(verrou) de la partition; s'ils sont différents ➔ **erreur**.

- **L'IBM/360 :**



- La clé « 0000 » permet tout accès à la mémoire.
- La clé et le code de protection(verrou) ne peuvent être modifiés que par le système à l'aide d'instructions privilégiées.

2.1.2 L'ordonnancement des processus

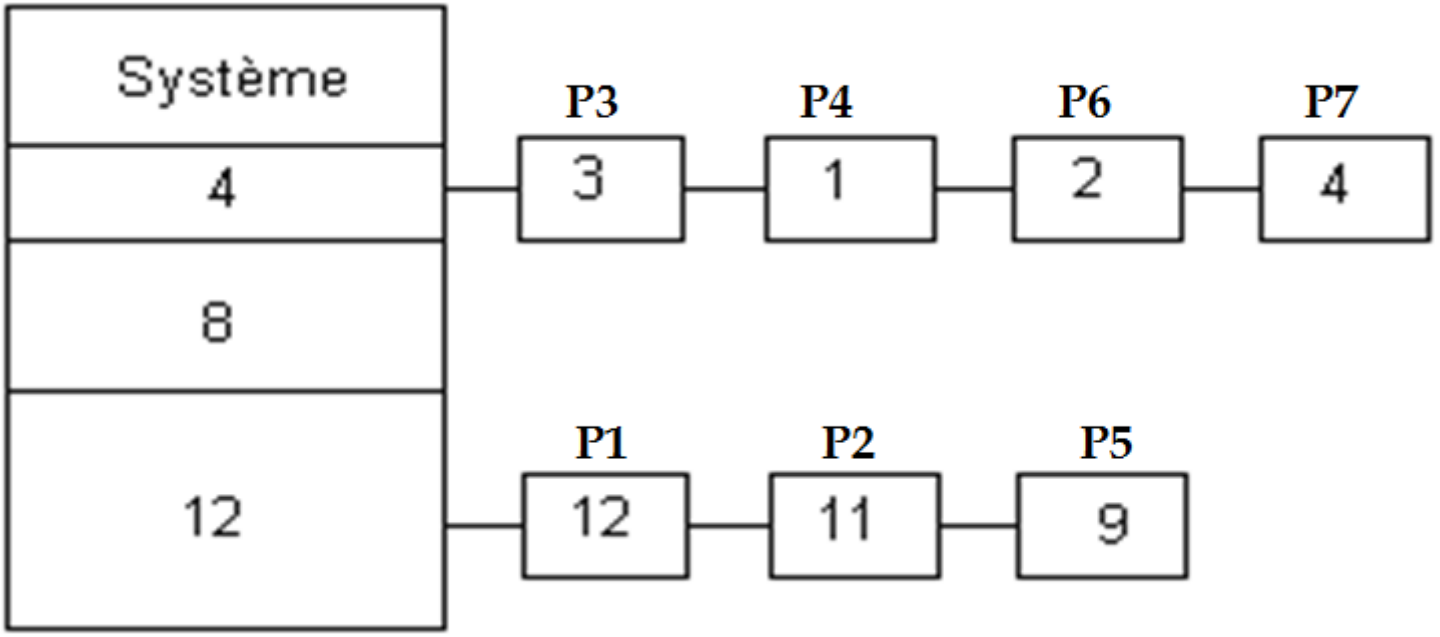
- Tout processus doit faire une demande explicite de l'espace mémoire nécessaire à l'exécution des programmes qui le composent.
- Cette demande est exprimée au moyen d'un langage de commandes.
- L'ordonnanceur peut utiliser:

a) Une file de processus par partition :

- Chaque processus est inséré dans la file de la partition dont la taille est la plus proche (supérieure ou égale) de la taille demandée.
- **Exemple :**

P7	P6	P5	P4	P3	P2	P1
4	2	9	1	3	11	12

P7	P6	P5	P4	P3	P2	P1
4	2	9	1	3	11	12

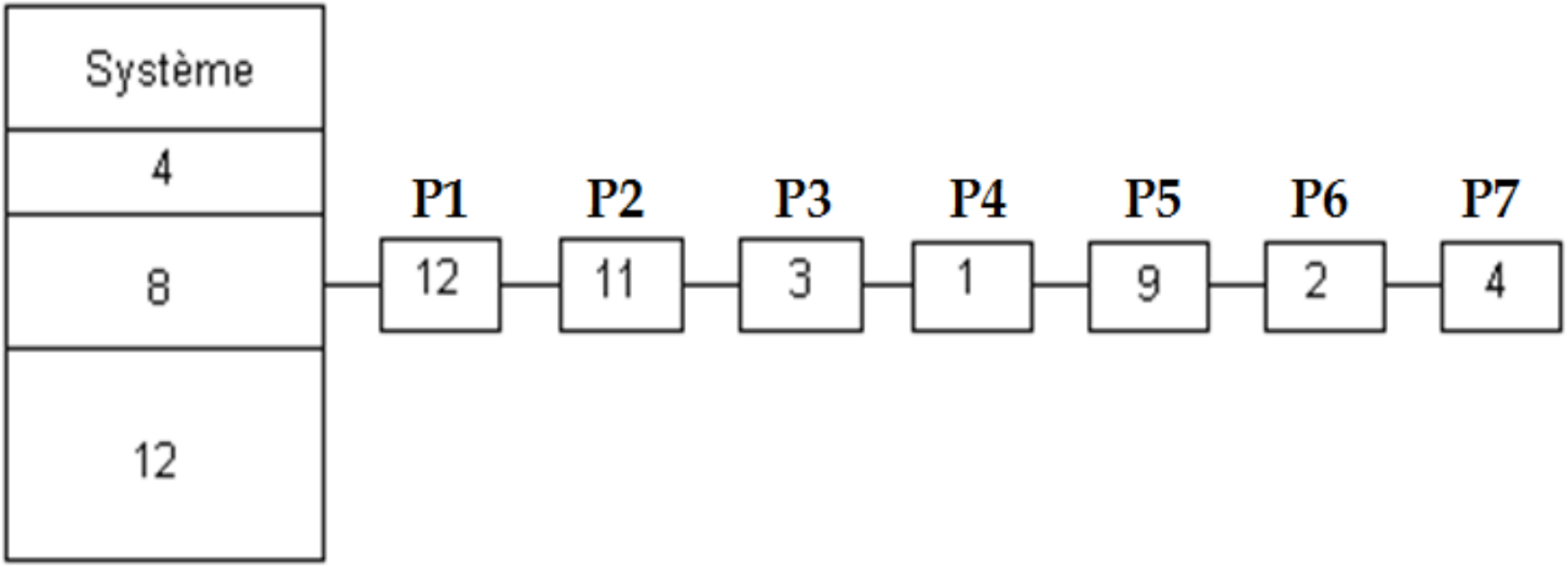


Inconvénients:

On peut trouver des files vides alors que d'autres files contiennent beaucoup de travaux en attente: Mauvaise répartition des charges sur les différentes partitions.

L'espace inutilisé dans une partition est perdu : C'est la **fragmentation interne**.

b) Une seule file pour tous les processus



- Dans ce cas, on alloue au processus choisi la première partition dont la taille est supérieure ou égale à la taille demandée.
- Si on utilise un algorithme d'ordonnancement FIFO, tant que la requête du premier travail de la file n'est pas satisfaite aucun autre travail ne passera.

c) Choix des tailles des partitions

- Les tailles des partitions sont définies en fonction de la nature des travaux à exécuter et des statistiques d'utilisation de la machine.

2.1.3 Fragmentation de la mémoire

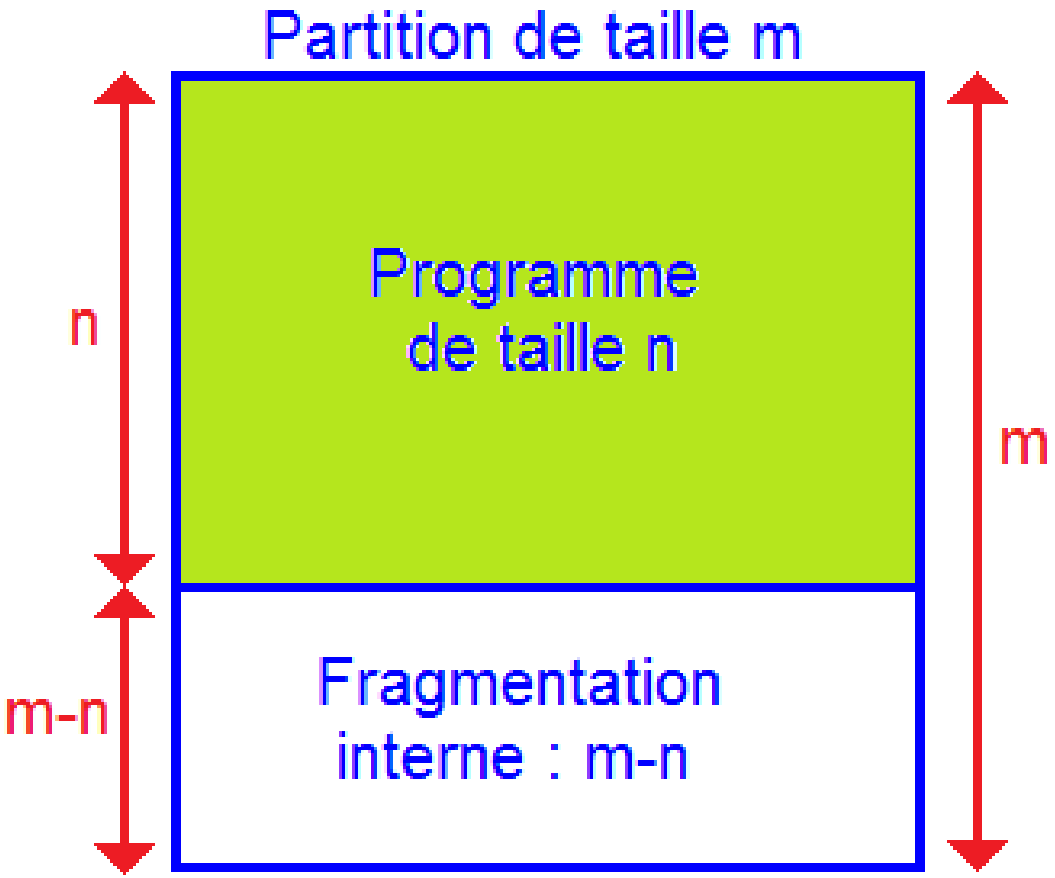
- Le principal inconvénient de cette technique est le **fragmentation** de la mémoire principale.
- On distingue deux types de fragmentations :

a) La fragmentation interne

- Le choix des tailles des partitions doit être fait de telle sorte qu'il minimise la fragmentation interne de la mémoire centrale.

- On dit qu'il y a **fragmentation interne** quand un programme de taille **n** est chargé dans une partition de taille **m** avec **m>n :**
- Fragmentation interne =

  taille de la partition - taille du programme = **m - n**.



b) La fragmentation externe

- **Exemple:**

  Soit un programme qui demande un espace mémoire de 50 unités et une mémoire avec 2 partitions libres de taille respective 30 et 40 unités.

- La requête de ce programme ne peut pas être satisfaite, car il n'existe pas de partition libre dont la taille est supérieure ou égale à la taille demandée par le programme.

- On dit qu'il y a **fragmentation externe,** si un programme de taille **n** est en attente de la mémoire alors qu'il existe un espace mémoire total suffisant pour satisfaire la requête (demande) du programme mais cet espace n'est pas contigu :
  Il est constitué de plusieurs petites partitions (la somme des tailles des partitions libres ≥ taille du programme).
  Exemple: (30+40)>50.

2.2 La technique des partitions variables

- Technique des partitions fixes ➔ **fragmentation interne et fragmentation externe.**
- Comment minimiser ou éliminer la fragmentation interne ?
- **Solution** : Utiliser des partitions de taille variable.
- Une partition est définie :
  - par sa taille et
  - son adresse.
- Au début, toute la mémoire disponible (libre) est constituée d'**une seule partition**.
- Les partitions sont créées au fur et à mesure de l'exécution des processus(allocation/libération de partition).

2.2.1 Allocation d'une partition

- L'allocation d'une partition de taille **t** à un processus, consiste à trouver une zone de taille convenable parmi les zones (partitions) libres.
- La taille de la partition choisie doit être supérieure ou égale à la taille **t** demandée par le processus.

- Si la partition sélectionnée a une taille trop grande, elle est découpée en deux (c'est ce que l'on appelle un éclatement de partition):
  - Une partie de taille **t** est allouée au processus;
  - L'autre est insérée dans «la liste » des partitions libres.

2.2.2 Libération d'une partition

- Quand un processus se termine, il libère sa partition qui est de nouveau placée dans la « liste » des zones libres.
- **Libération** ➔ Fusionner les partitions chaque fois que cela est possible.
  Fusion : Créer, à partir de plusieurs partitions contiguës, une seule partition de grande taille.

- La libération d'une partition peut créer trois situations différentes selon la position de la partition libérée:

a) La partition libérée est entourée de 2 partitions libres

b) La partition libérée est entourée d'une partition libre et d'une partition occupée

c) La partition libérée est entourée de 2 partitions occupées

2.2.3 Algorithmes de sélection d'une partition

- Les principaux algorithmes pour le choix(ou la sélection) d'une partition sont **first-fit, best-fit et buddy system**.

a) First-fit

1. Choisir la première partition dont la taille est suffisamment grande (taille de la partition choisie ≥ taille demandée).

   La recherche s'arrête dés que l'on trouve une partition libre de taille assez grande.

2. Si la taille de la partition choisie est égale à la taille demandée
   ➔ Allouer toute la partition.

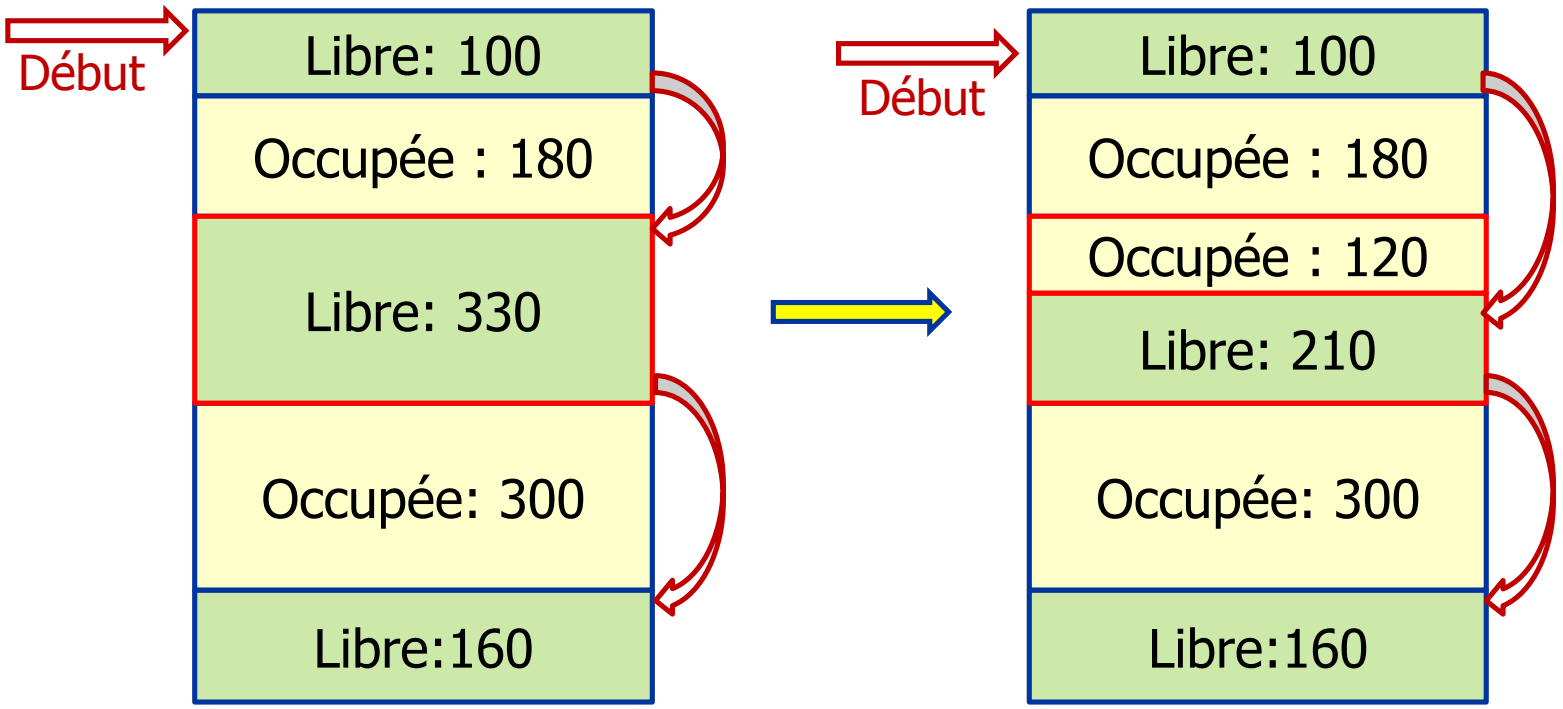
   Si la taille de la partition choisie est supérieure à la taille demandée ➔ Découper la partition en deux :
   - Allouer une partition de taille égale à la taille demandée et
   - Le reste devient une nouvelle partition libre(plus petite).

- La recherche commence toujours au début de la liste.

- Si la recherche commence là où la dernière recherche a fini ➔ l'algorithme **next-fit**. Dans ce cas, la liste est traitée de manière circulaire.

- La liste des partitions est ordonnée par **ordre croissant des adresses**.

Exemple :



- Demande d'une partition de taille = 120 .
- ➔ Partition choisie : partition de taille 330.

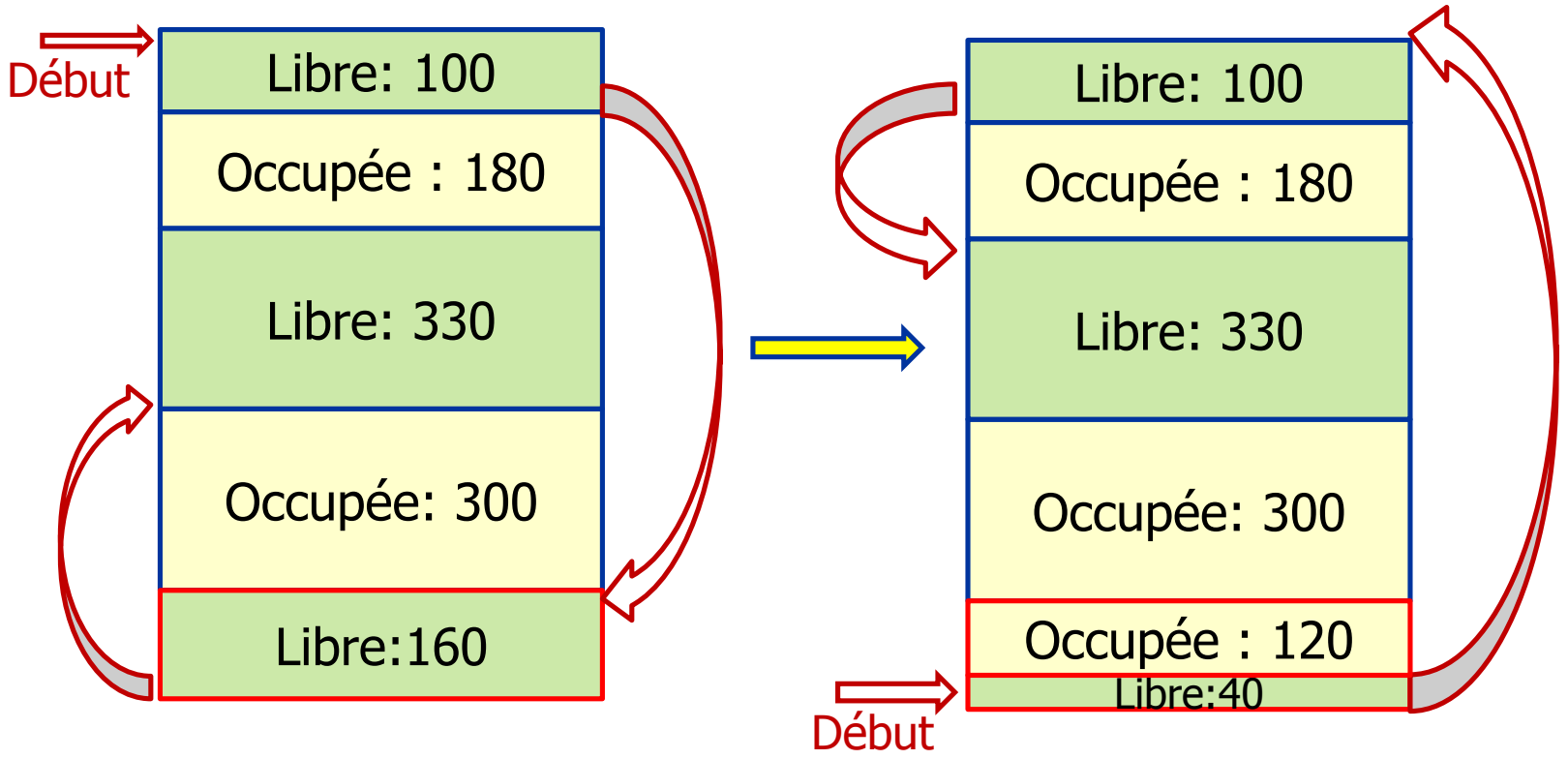
b) Best-fit

1. Choisir la partition dont la taille est la plus proche de la taille demandée.
2. Si la taille de la partition choisie est égale à la taille demandée
   ➔ Allouer toute la partition.

   Si taille de la partition choisie est supérieure à la taille demandée
   ➔ Découper la partition en deux:
   - Allouer une partition de taille égale à la taille demandée et
   - Le reste devient une nouvelle partition libre(plus petite).

- Liste ordonnée par ordre croissant des tailles des partitions libres.
- Si la liste n'est pas ordonnée par ordre croissant des tailles des partitions libres ➔ Recherche effectuée dans toute la liste des partitions libres.

- Best-fit évite de découper inutilement une grande partition; Cependant, il génère plus rapidement de très petites partitions inutilisables.



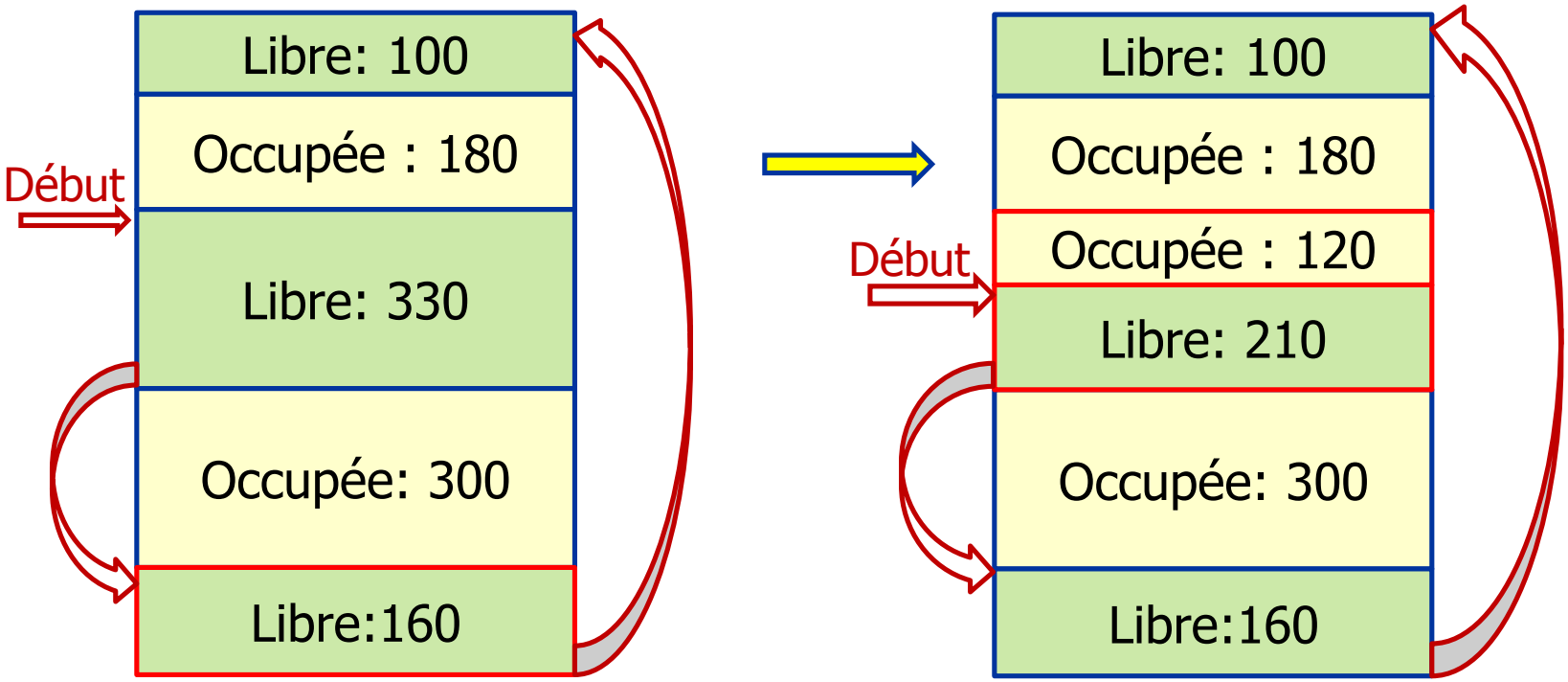
- Demande d'une partition de taille = **120**.
- ➔ Partition choisie : **Partition de taille 160**.

c) Worst-fit

1. Choisir la plus grande partition libre(minimiser l'émiettement de la mémoire).
2. Allouer une partie de taille suffisante pour le programme et le reste devient une nouvelle partition libre.

- La liste est ordonnée par **ordre décroissant** des tailles des partitions libres.
- La recherche est effectuée dans toute la liste des partitions libres, Si la liste n'est pas ordonnée par ordre décroissant des tailles des partitions libres.
- Limite les trop petites partitions ➔ minimiser l'émiettement de la mémoire.

Exemple :



- Demande d'une partition de taille = 120.
- ➔ Partition choisie : partition de taille 330.

Exemple1 : à l'instant t on a 2 partitions libres non contigûes.

	First-fit		Best-fit		Worst-fit	
Etat initial :	1300	1200	1300	1200	1300	1200
Demande: 1000	300	1200	1300	200	300	1200
Demande : 1100	300	100	200	200	300	100
Demande : 250	50	100	échec		50	100

Exemple2 : à l'instant t on a 2 partitions libres (non contigûes):

	First-fit		Best-fit		Worst-fit	
Etat initial :	2000	1500	2000	1500	2000	1500
Demande : 1200	800	1500	2000	300	800	1500
Demande : 1400	800	100	600	300	800	100
Demande : 300	500	100	600	0	500	100
Demande : 600	Echec		0	0	Echec	

d) Algorithme Buddy system ou système de compagnons

1) Principe de fonctionnement de l'algorithme buddy system :

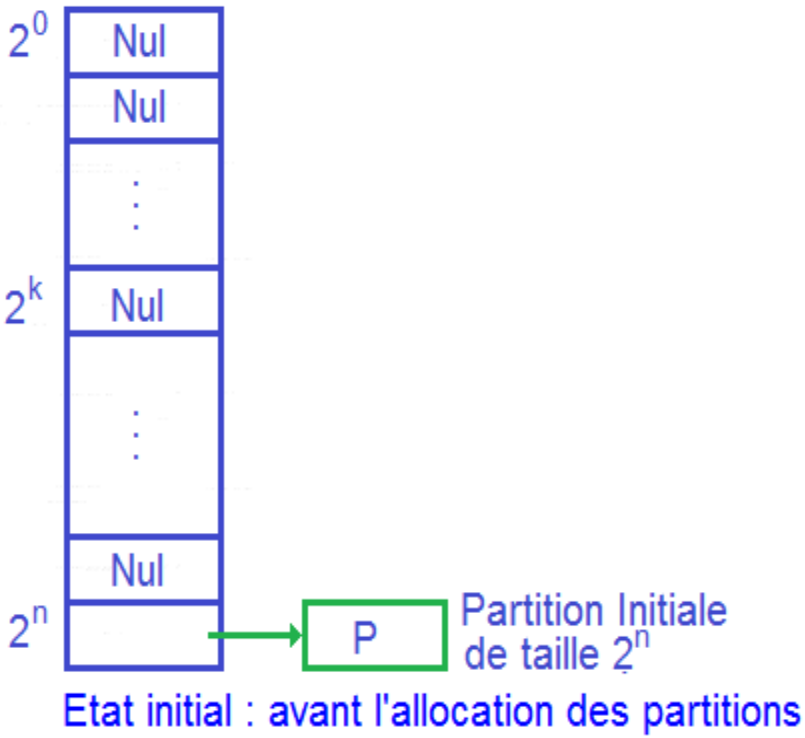
- Les tailles des partitions (zones ou blocs) mémoires sont des puissances de 2 : 2^k.
- **Exemple** : $2^0, 2^1, 2^2, 2^3, \ldots, 2^n$ ➔ $1, 2, 4, 8, \ldots, 2^n$
- La taille t d'une partition demandée doit être arrondie à la puissance de 2 supérieure ou égale à t : 2^k avec $t \leq 2^k$.

Selon Donald Knuth, l'algorithme buddy system a été conçu en 1963 par Harry Markowitz(Prix Nobel d'économie en 1990) et a été publié par Kenneth C. Knowlton en 1965.

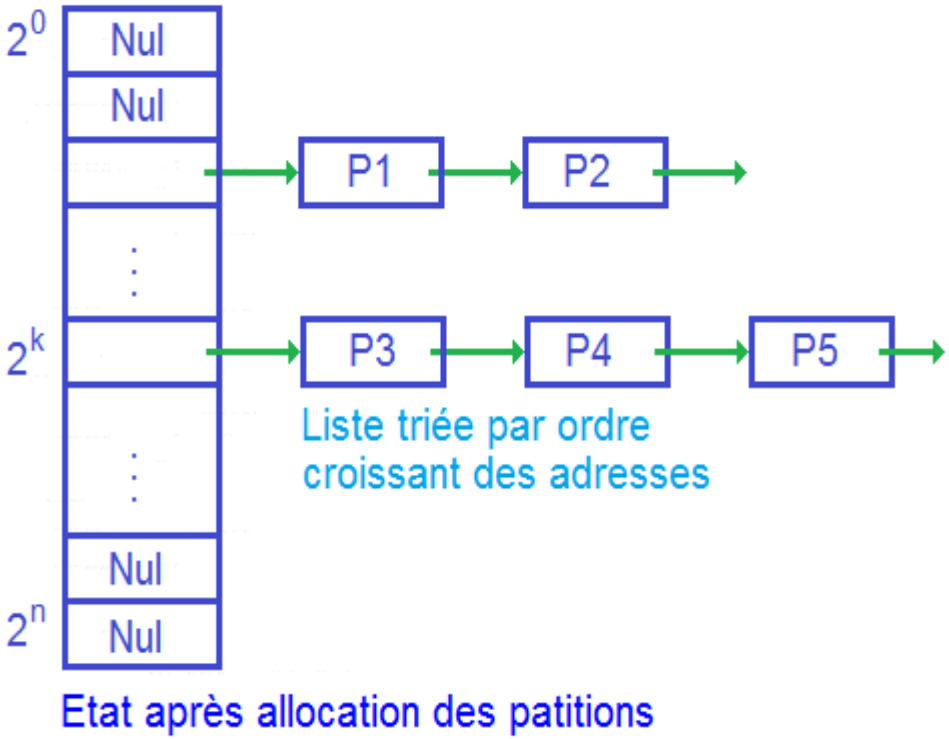
- Si la taille maximale de la mémoire est de 2^n et
  la taille de la plus petite partition est 2^m :
  La taille t ($t = 2^k$) d'une partition doit être : **$2^m \leq t \leq 2^n$**

2) Représentation des partitions libres

- Les différentes listes de partitions libres constituent un tableau de listes des partitions libres(ou tableau de pointeurs de listes).
- Toutes les partitions libres de même taille sont chaînées dans une même liste.
- Nombre d'éléments(entrées) du tableau des listes des partitions libres = **Log2(taille maxi) - log2(taille min) + 1**



Etat initial : avant l'allocation des partitions



Etat après allocation des patitions

3) Allocation d'une partition de taille t

- Arrondir t à 2^k ➔ $t \leq 2^k$
- Chercher une partition libre de taille 2^k :
  - Si l'entrée k du tableau des listes des partitions libres n'est pas vide(la liste contient au moins une partition) : Allouer une partition de taille 2^k .
  - Si l'entrée k du tableau des listes est vide (il n'existe pas de partition libre de taille 2^k).
    - ✓ Allouer une partition de taille 2^{k+1} , s'il en existe;
    - ✓ Découper cette partition en deux partitions libres(deux compagnons) de tailles 2^k :
      - ➢ Allouer la partition de plus petite adresse(compagnon gauche) et
      - ➢ L'autre partition (compagnon droit) est ajoutée à la liste k des partitions libres.

- S'il n'existe pas de partition libre de taille 2^{k+1},
  - Poursuivre les recherches dans les listes de tailles supérieures 2^{k+2}, 2^{k+3}, …;
  - Si toutes les recherches son infructueuses, la demande n'est pas satisfaite.
- Sinon, la partition libre trouvée de taille 2^j ($j \geq k+1$) est décomposée en deux partitions(2 compagnons):
  - La partition de droite est ajoutée à la liste des partitions libres correspondant à sa taille et
  - La partition gauche est de nouveau divisée en deux partitions et ainsi de suite, jusqu'à pouvoir satisfaire exactement la demande (partition de taille 2^k).

Remarque : Pour faciliter l'allocation et la fusion, il faut insérer les adresses des partitions libres par ordre croissant des adresses.

4) Libération d'une partition de taille 2^k

- Lorsqu'une partition est libérée et que son compagnon est libre, les deux partitions sont fusionnées.

  Les fusions sont poursuivies tant que le compagnon de la partition résultat de la fusion est libre (fusionner récursivement).

Exemple :

- On dispose d'une mémoire centrale de **1Mo** initialement libre.
  La taille **t** d'une partition : **1ko ≤ t ≤ 1Mo .**
  Taille de la plus petite partition = **1ko**.
- Les requêtes successives suivantes sont faites :
  - Demande d'allocation d'un bloc A de 70k octets;
  - Demande d'allocation d'un bloc B de 35k octets;
  - Demande d'allocation d'un bloc C de 130k octets;
  - Libération du bloc A;
  - Libération du bloc C;
  - Libération du bloc B;

- Nombre d'entrées de la table des partitions libres :

  Une partition peut avoir une taille de 1024 octets à 1Mo ou 2^{10} à 2^{20} octets ➔ Pour représenter toutes les partitions libres , on aura besoin d'une table de 11 entrées(0 à 10).

  Nombre d'entrées = Log2(taille max de la mémoire) – Log2(taille de la plus petite partition) + 1

  Nombre d'entrées = $\text{Log2}(2^{20}) - \text{Log2}(2^{10})+1 = 20-10+1 =$ **11**

Remarque :

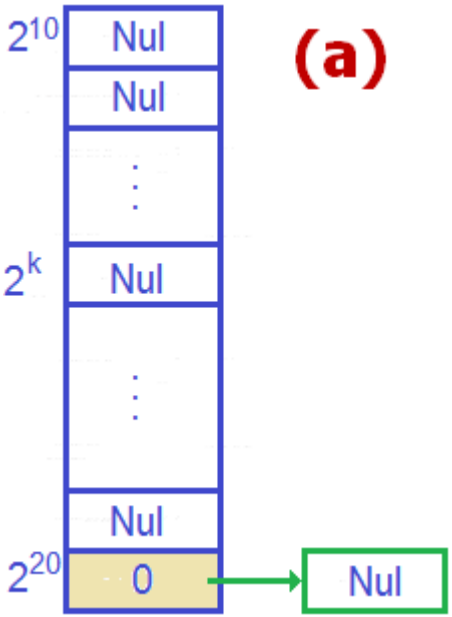
Taille de la mémoire et des partitions : **puissance de 2**.

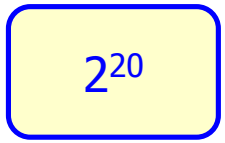
Etat initial :

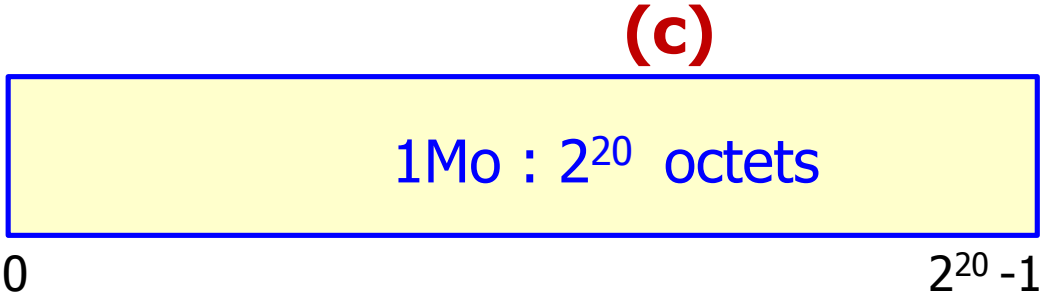
a) L'espace libre représenté à l'aide d'un tableau de listes des partitions libres.

b) L'espace libre et occupé représentés à l'aide d'un arbre binaire.

c) Représentation schématique de l'espace mémoire(libre, occupé).

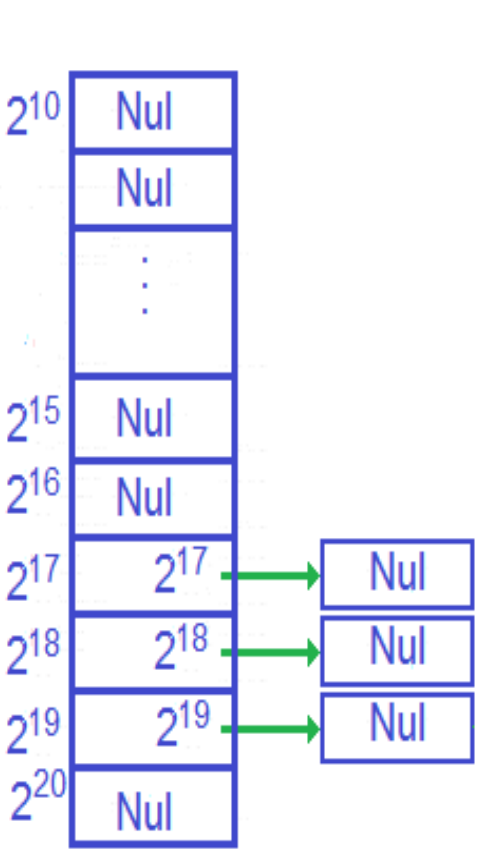


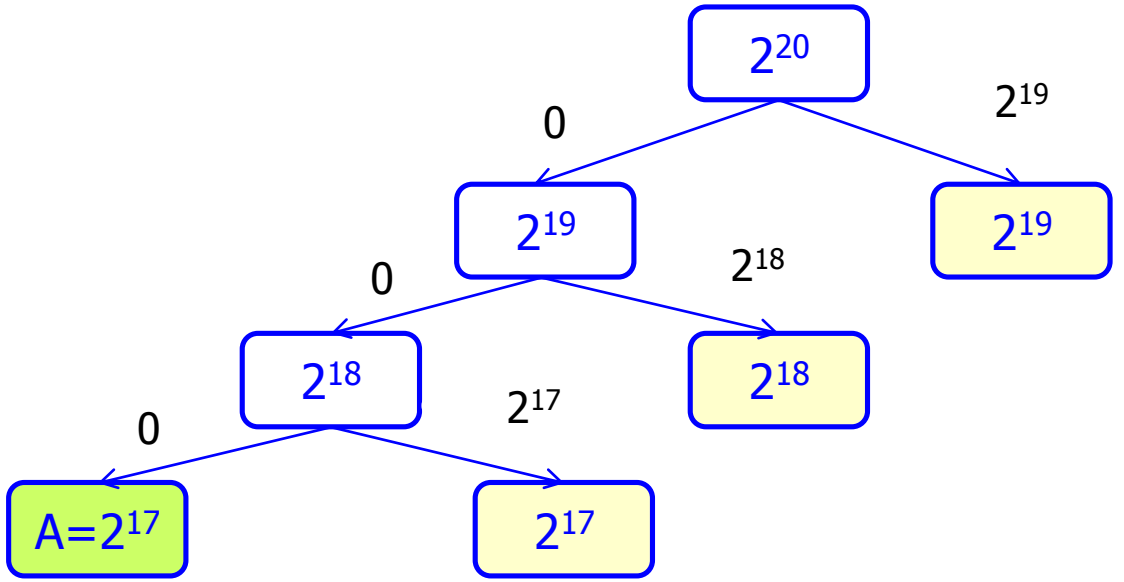


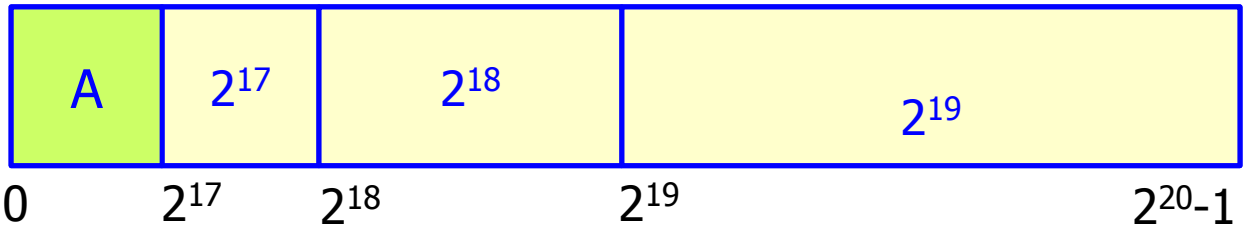


Demande d'allocation d'un bloc A de 70k octets;

- A=70k ➔ 128k ➔ 2^{17}

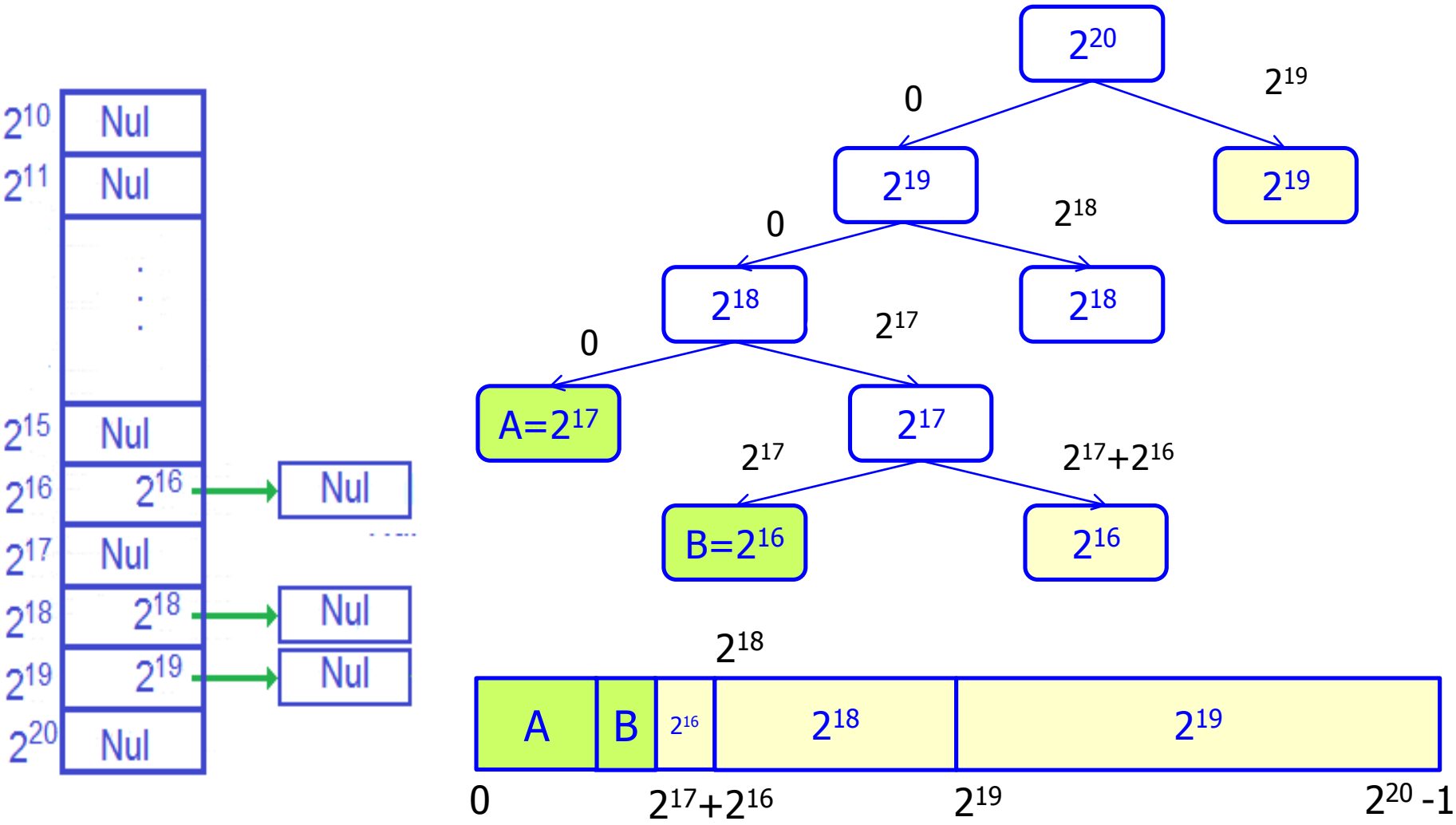




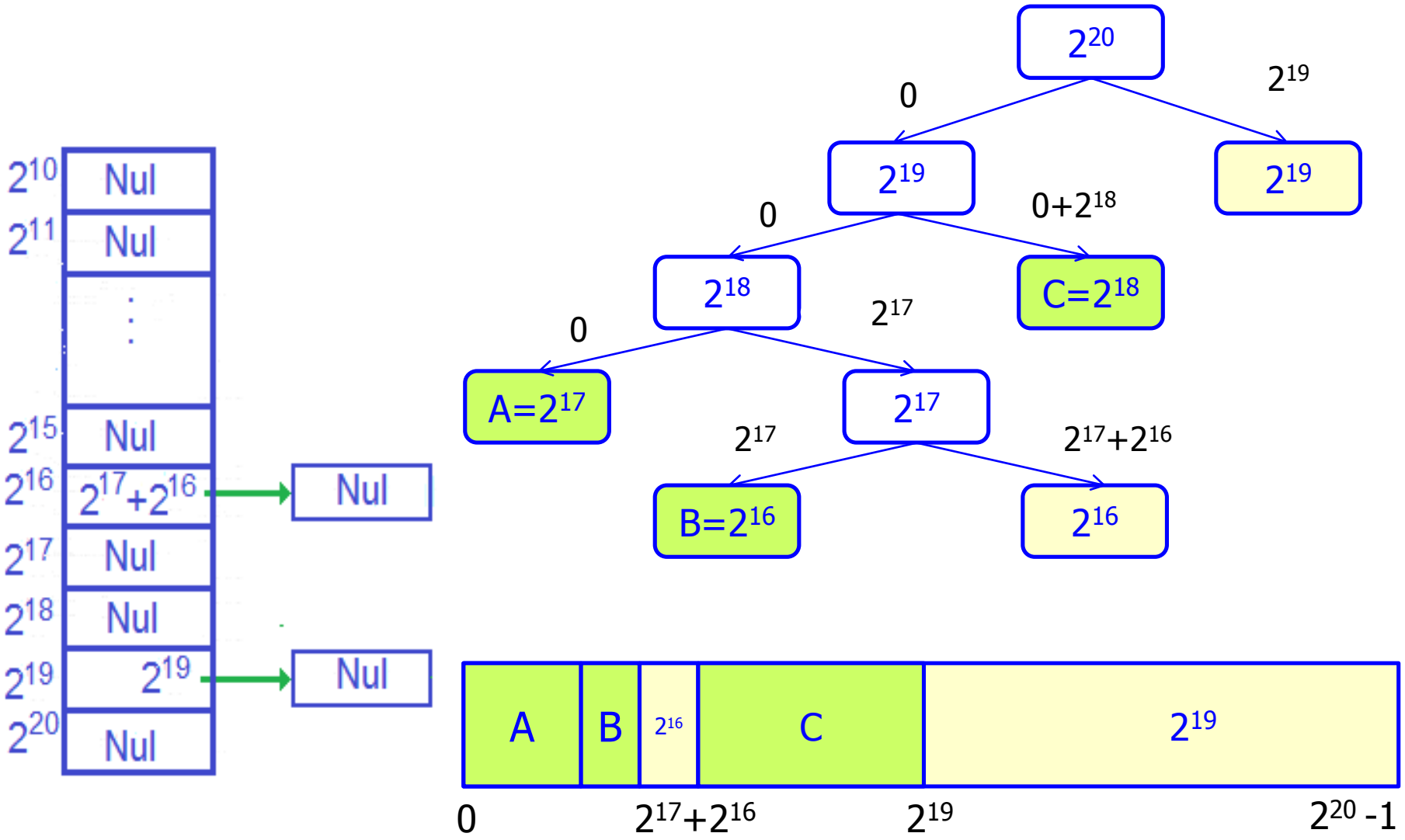


Demande d'allocation d'un bloc B de 35k octets;

- B=35k ➔ 64k ➔ 2^{16}



- Demande d'allocation d'un bloc C de 130k octets;
  - C=130k➔ 256k ➔ 2^{18}



- Comment identifier le compagnon d'une partition d'adresse **adr** et de **taille 2^k** (on ignore s'il s'agit d'un compagnon de droite ou de gauche), parmi plusieurs partitions d'une liste donnée ?
  Un compagnon est identifié par son adresse : soit **adrc**
- Calcul de l'adresse **adrc** du compagnon.
- Adrc = adr $\oplus$ taille ;
  $\oplus$: ou exclusif

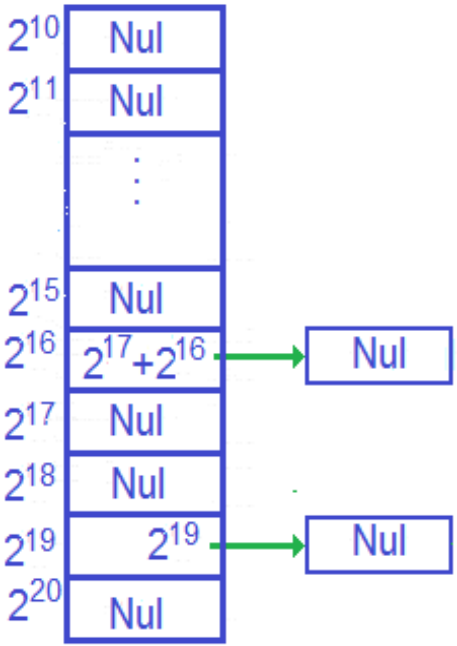
a	b	a $\oplus$ b
0	0	0
0	1	1
1	0	1
1	1	0

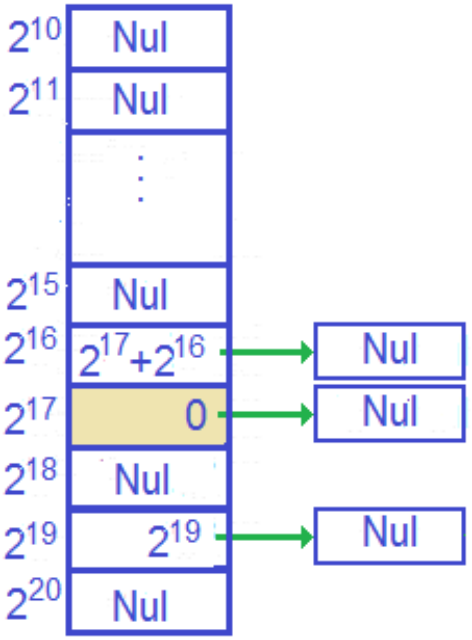
➢ Libération du bloc A; L'adresse de A = 0 et taille = 2^{17}

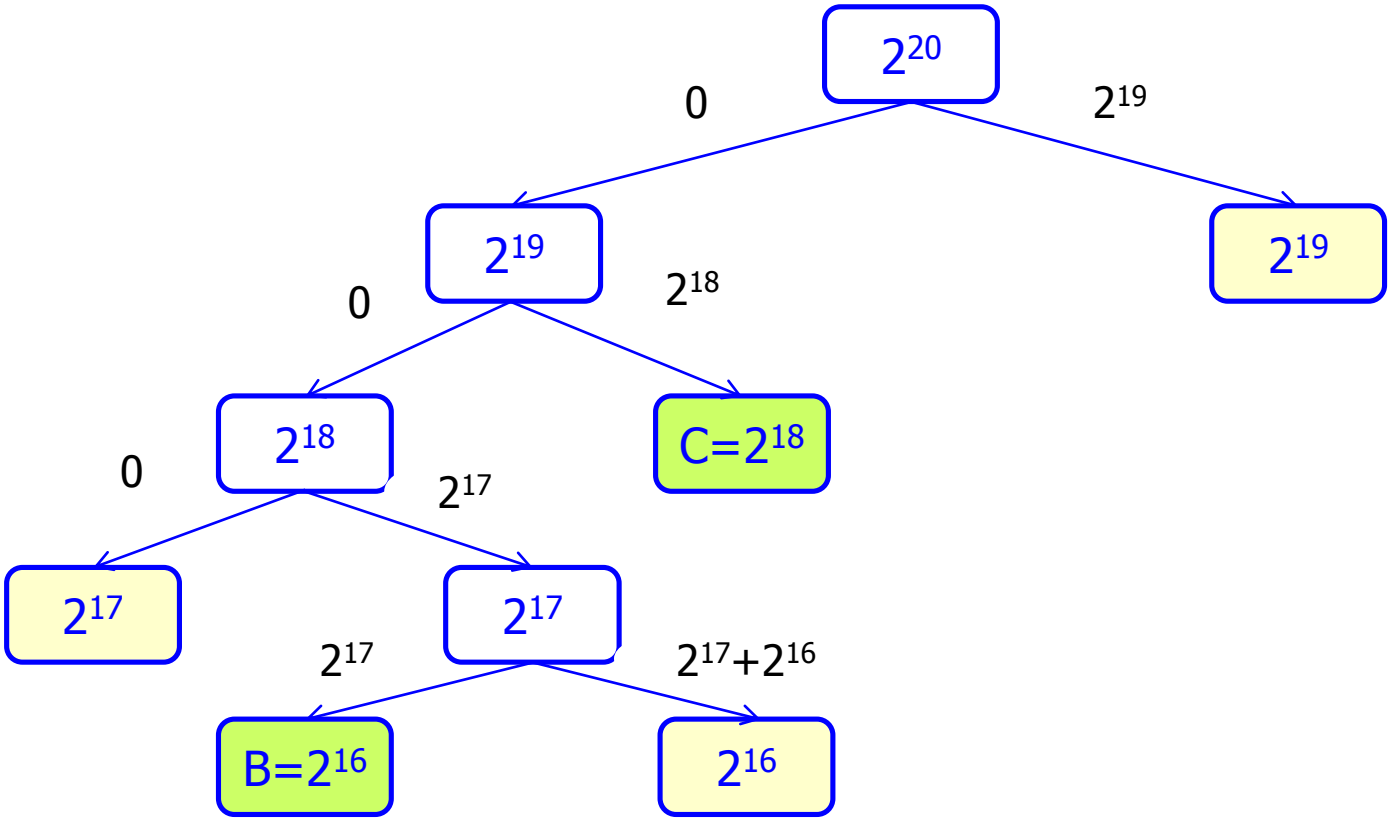
- Calculer l'adresse du compagnon : adrc = 0 ⊕ 2^{17} ➔ 0x00000 ⊕ 0x20000 = 2^{17} ;

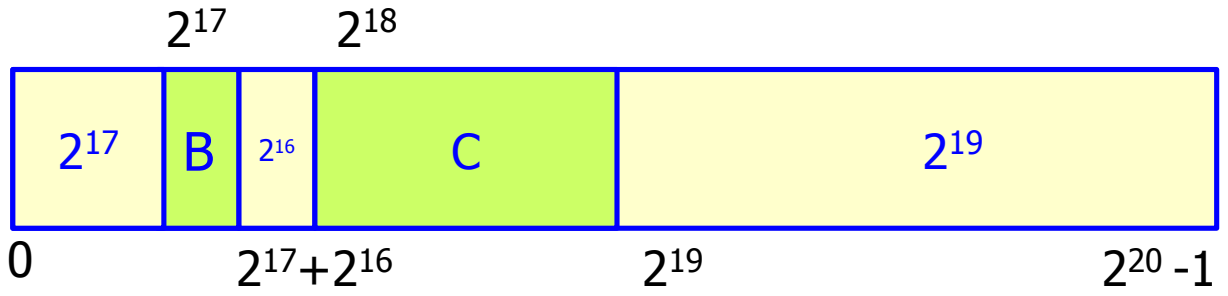
	0000 0000 0000 0000 0000	00000
⊕	0010 0000 0000 0000 0000	20000 : 2^{17}
	0010 0000 0000 0000 0000	20000 : 2^{17}

- La partition d'adresse 2^{17} et de taille 2^{17} n'existe pas dans la liste des partitions libres ➔ Insérer.





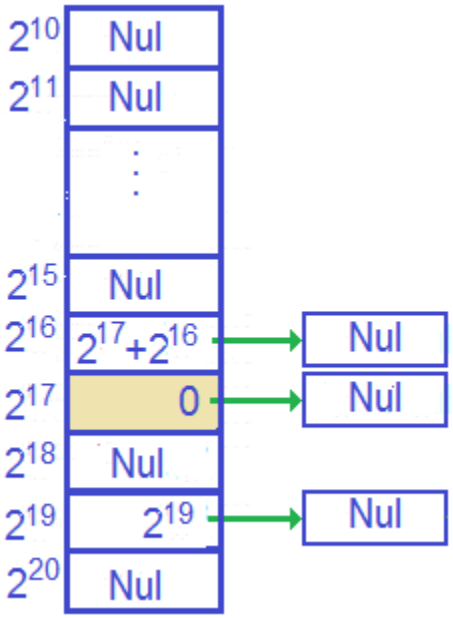

2^{20}
0
2^{19}
2^{19}
2^{19}
0
2^{18}
2^{18}
$C=2^{18}$
0
2^{17}
2^{17}
2^{17}
$2^{17}+2^{16}$
$B=2^{16}$
2^{16}

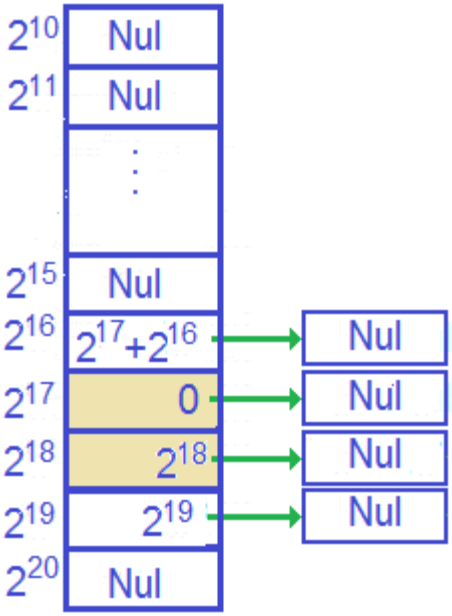

2^{17}
2^{18}
2^{17}
B
2^{16}
C
2^{19}
0
$2^{17}+2^{16}$
2^{19}
$2^{20}-1$

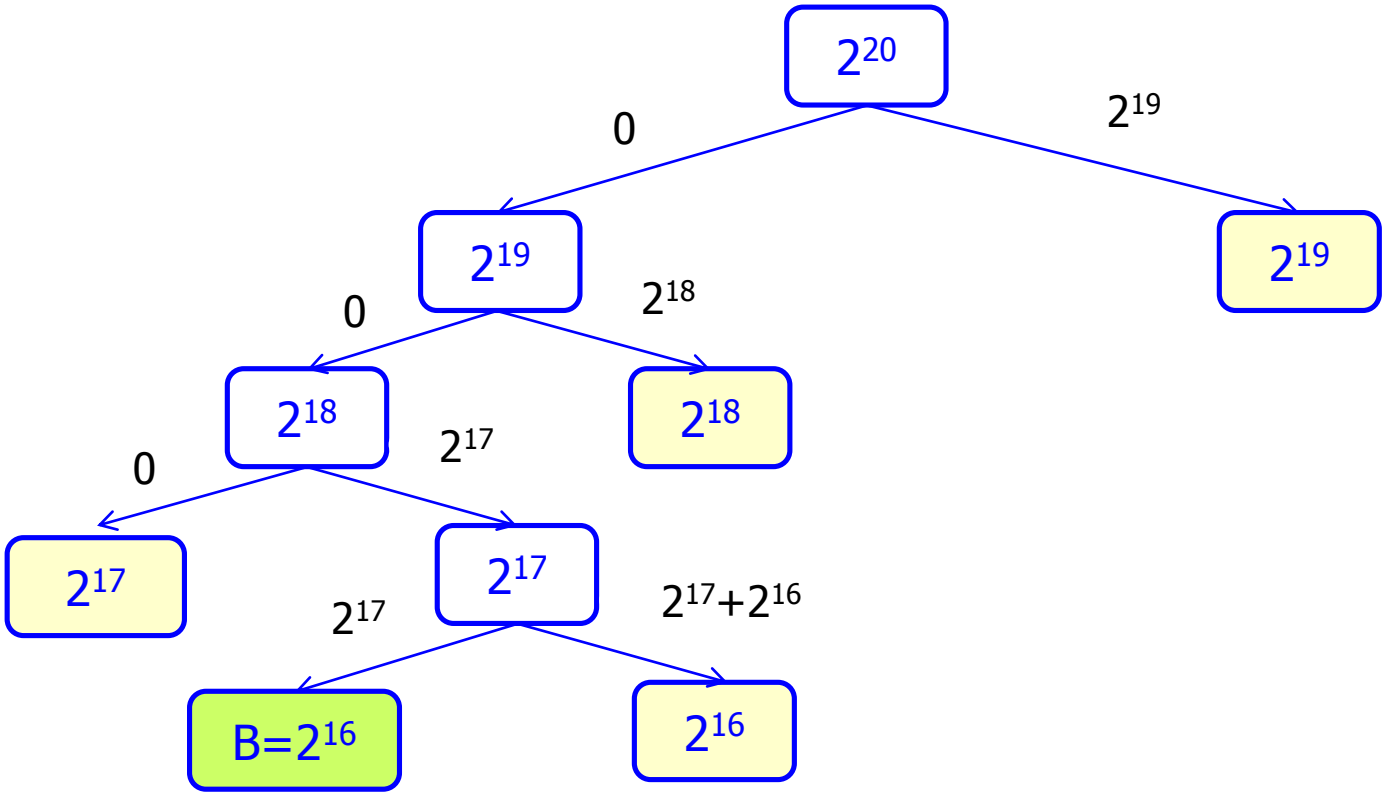
➢ Libérer C; Adrc = $2^{18} \oplus 2^{18}$ ➔ 0x40000 ⊕ 0x40000 = 0

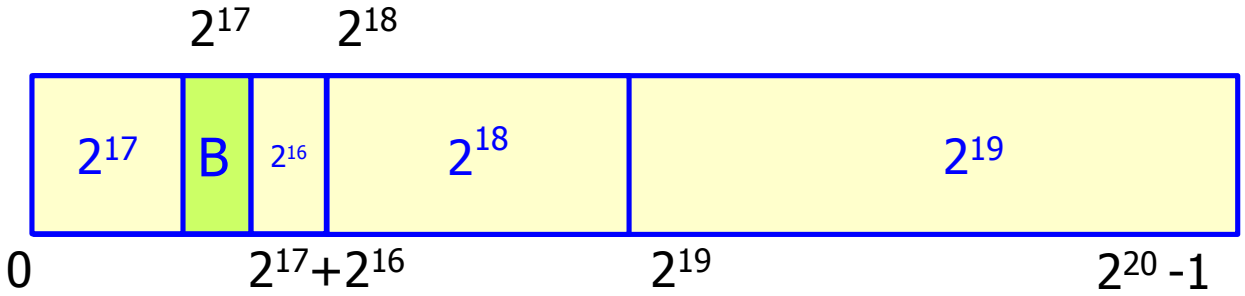
	0100 0000 0000 0000 0000	40000 : 2^{18}
⊕	0100 0000 0000 0000 0000	40000 : 2^{18}
	0000 0000 0000 0000 0000	00000 : 0

- La partition d'adresse 0 n'existe pas dans la liste des partitions libres de taille 2^{18} ➔ insérer.



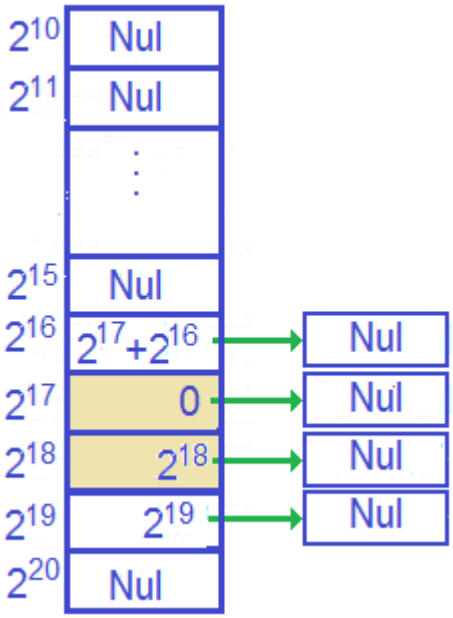



2^{20}
0
2^{19}
2^{19}
2^{19}
0
2^{18}
2^{18}
2^{18}
0
2^{17}
2^{17}
2^{17}
$2^{17}+2^{16}$
B=2^{16}
2^{16}


2^{17}
2^{18}
2^{17}
B
2^{16}
2^{18}
2^{19}
0
$2^{17}+2^{16}$
2^{19}
2^{20} -1

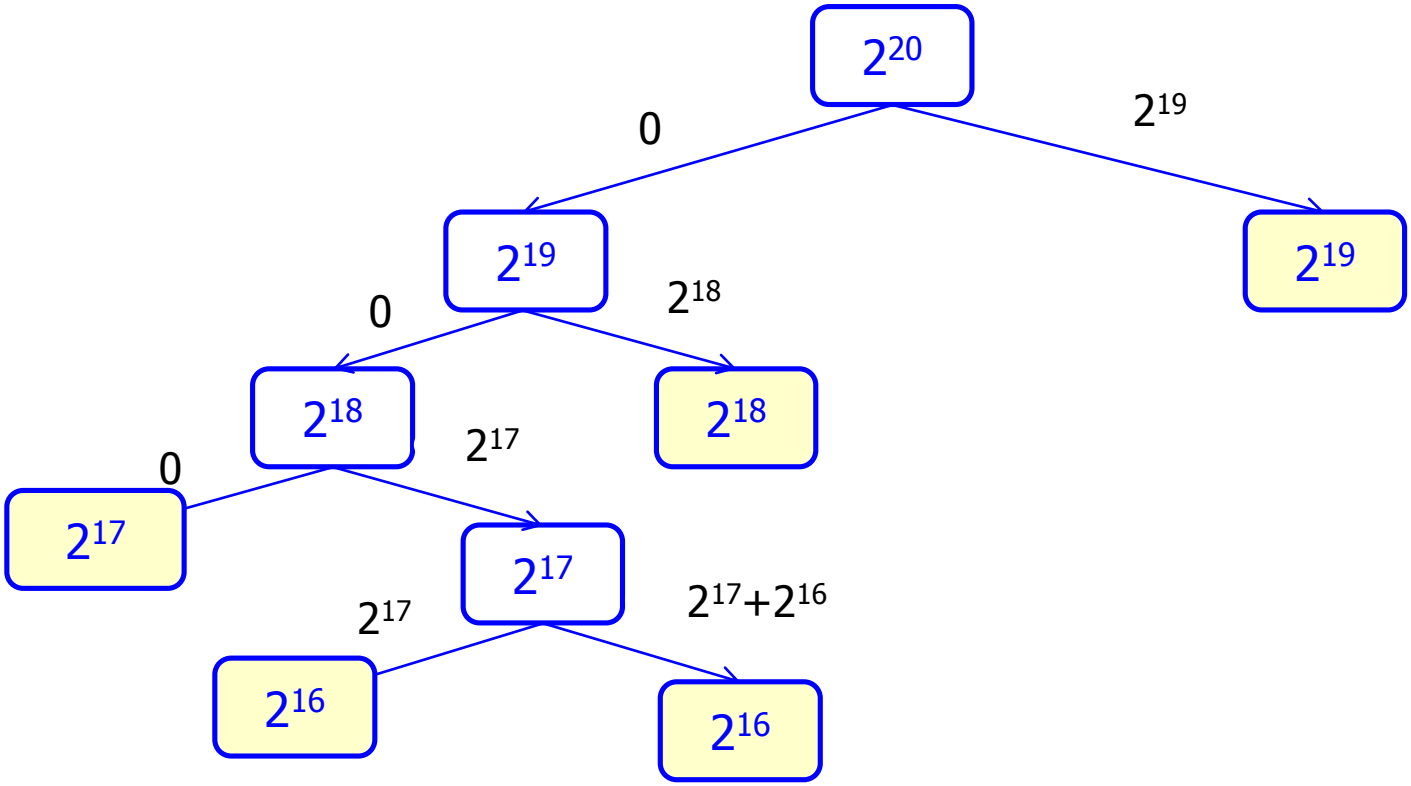
- Libérer B; adresse de B = 2^{17} et taille = 2^{16}
- Adrc = $2^{16} \oplus 2^{17}$ ➔ 0x10000 ⊕ 0x20000 = $2^{17}+2^{16}$ existe ➔ fusion

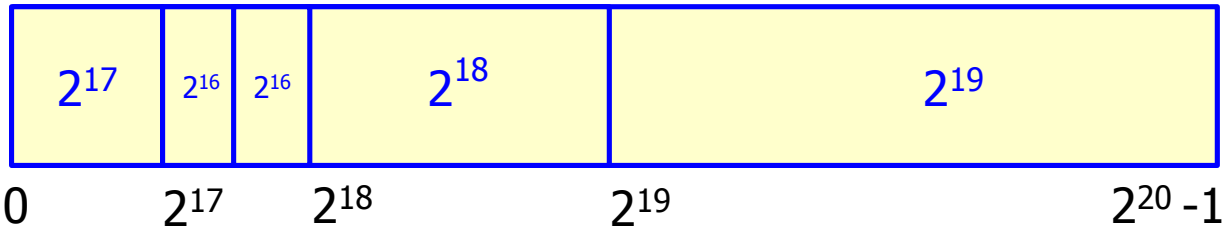
	0001 0000 0000 0000 0000	10000 : 2^{16}
⊕	0010 0000 0000 0000 0000	20000 : 2^{17}
	0011 0000 0000 0000 0000	30000 : $2^{17}+2^{16}$



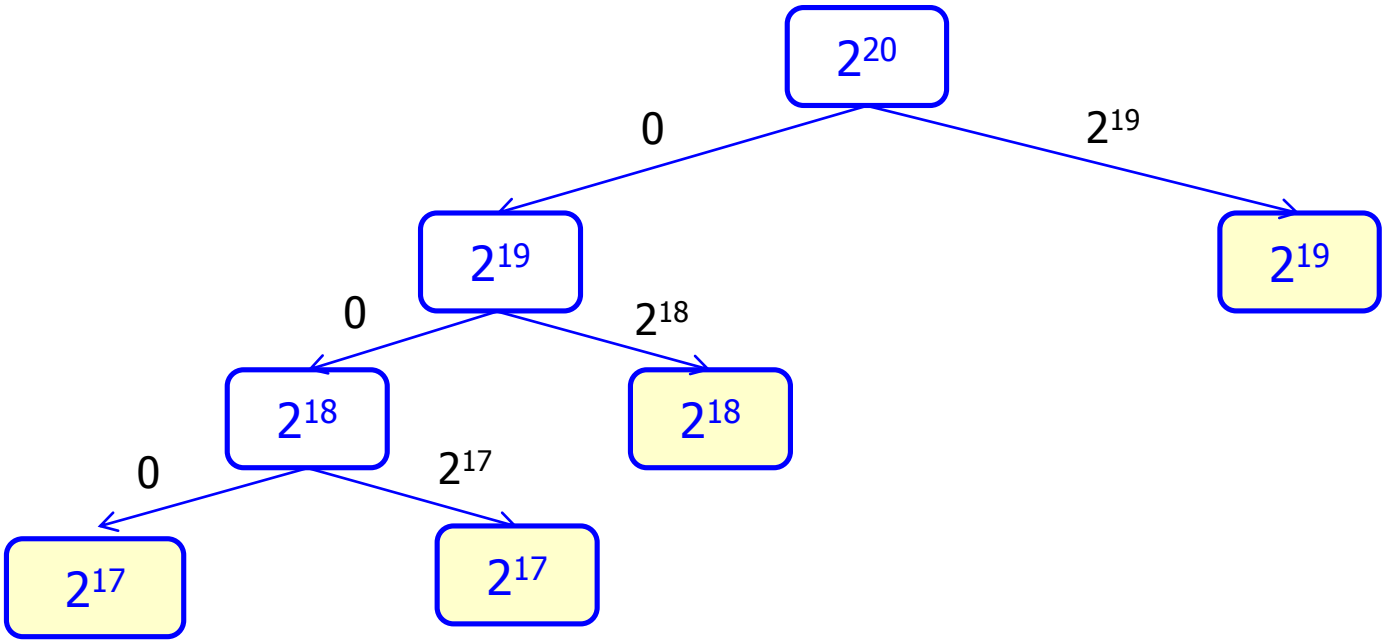
- Le résultat de la fusion est une partition de taille=2^{17} et d'adresse=2^{17}.
- Fusionner tant qu'il y a des compagnons (partitions adjacentes de même taille).

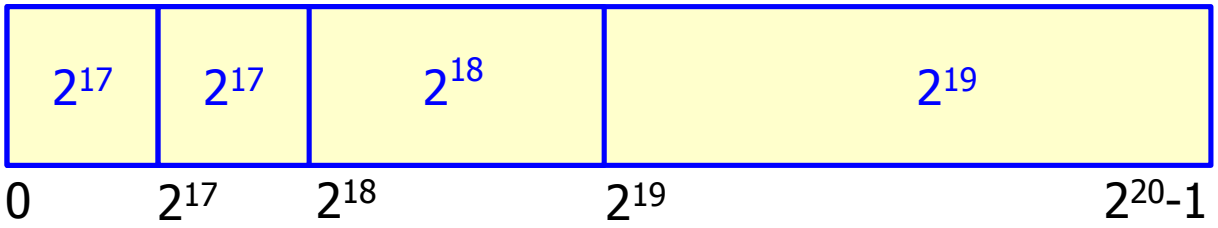
Suite de Libérer B : Avant la fusion de B(2^{16}) avec son compagnon





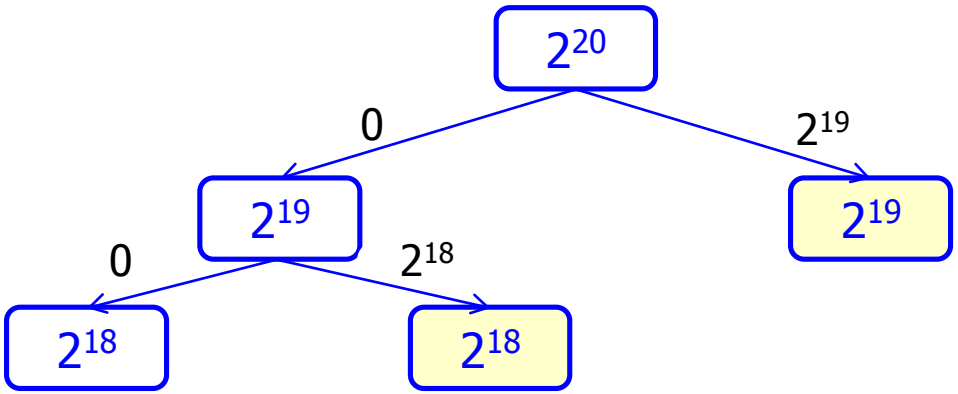
- Suite de Libérer B : Après la fusion de B avec son compagnon

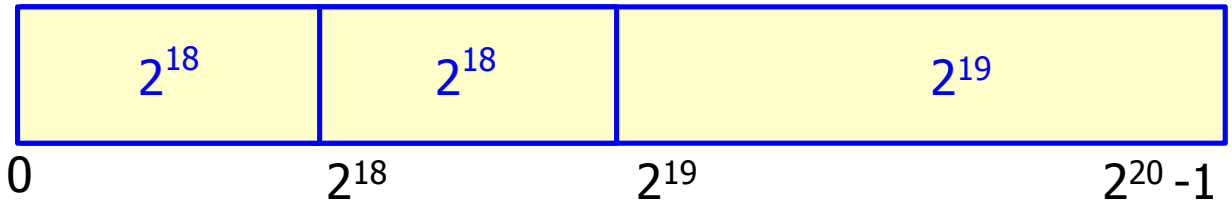




- Après la fusion, on obtient une partition de taille=2^{17} et d'adresse=0.
- Adrc = 0 ⊕ 2^{17} ➔ 0x00000 ⊕ 0x20000 = 2^{17} existe ➔ fusion

	0000 0000 0000 0000 0000	00000 : 0
⊕	0010 0000 0000 0000 0000	20000 : 2^{17}
	0010 0000 0000 0000 0000	20000 : 2^{17}

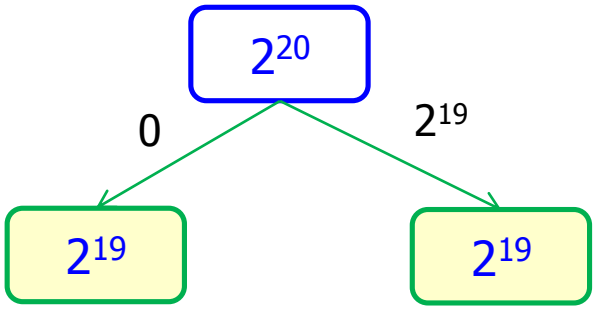


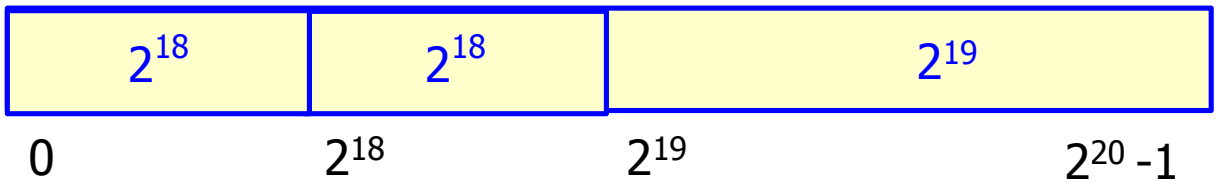


- Après la fusion, on obtient une partition de taille=2^{18} et d'adresse=0
- Adrc = 0 ⊕ 2^{18} ➔ 0x00000 ⊕ 0x40000 = 2^{18} existe ➔ fusion

	0000 0000 0000 0000 0000	00000 : 0
⊕	0100 0000 0000 0000 0000	40000 : 2^{18}
	0100 0000 0000 0000 0000	40000 : 2^{18}

- Après la fusion, on obtient une partition de taille=2^{19} et d'adresse=0.

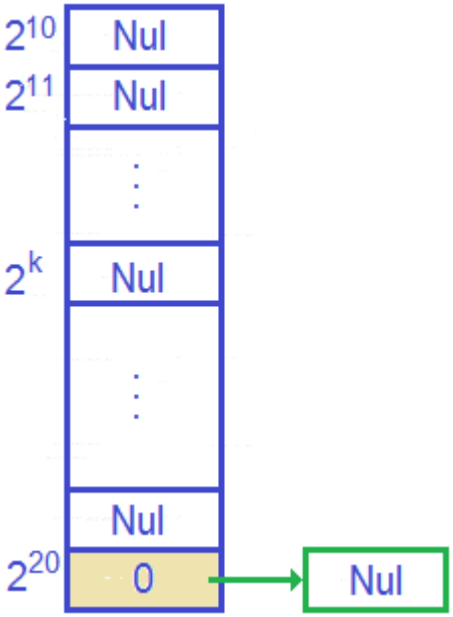


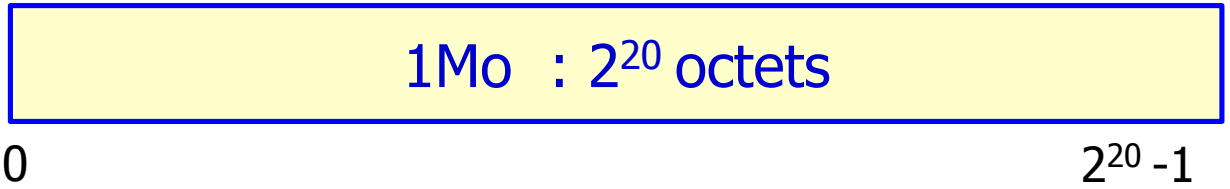


- Résultat de la fusion: une partition de taille=2^{19} et d'adresse=0.
- Adrc = 0 ⊕ 2^{19} ➔ 0x00000 ⊕ 0x80000 =2^{19} existe ➔ fusion : **fin**

	0000 0000 0000 0000 0000	00000 : 0
⊕	1000 0000 0000 0000 0000	80000 : 2^{19}
	1000 0000 0000 0000 0000	80000 : 2^{19}

- Résultat de la fusion: une partition de taille=2^{20} et d'adresse=0.



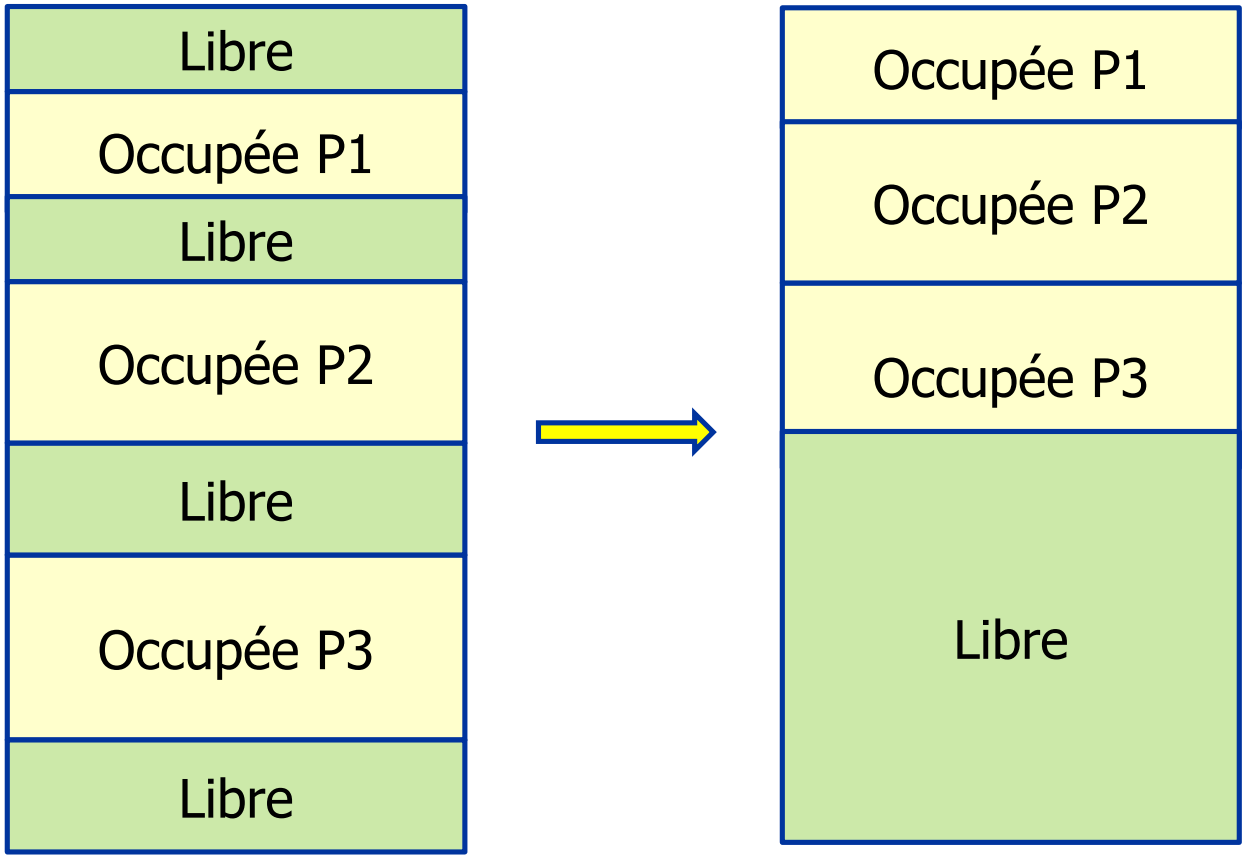


2.2.4 Fragmentation et compactage(défragmentation)

- Dans l'allocation avec partitions variables, le phénomène le plus gênant est celui de la fragmentation externe de la mémoire,
- Il se manifeste au bout d'un certain temps de fonctionnement: L'allocateur n'arrive pas à trouver une partition de taille suffisante.
- Une solution consiste à compacter les zones ou partitions allouées en les déplaçant, toutes, vers une extrémité de la mémoire (en bas ou en haut).
  Ce qui fait apparaître de l'autre bout une partition libre unique dont la taille est la somme des tailles des partitions libres existantes.

- Le compactage nécessite beaucoup de temps processeur : il faut le faire avec beaucoup de précaution ➔ **Il n'est pas très recommandé.**

Exemple :



2.3 Technique de va-et-vient (swapping)

- La mémoire est partagée entre
  - le système et
  - un ou plusieurs programmes utilisateurs.

- Les autres programmes sont stockés sur une mémoire secondaire (disque).
- Un programme est chargé, en M.C. avant de commencer son exécution.

3. Chargement des programmes en mémoire centrale

- Un fichier exécutable (module objet exécutable) contient :
  - Le code (lecture seule),
  - Les données initialisées,
  - Les données non initialisées(n'occupent pas d'espace disque).
- L'opération de chargement est réalisée par **le chargeur.**
- Le chargement consiste à implanter le programme (code et les données initialisées) en mémoire centrale à partir du fichier exécutable ➔ Effectuer les opérations suivantes:
  - Allouer l'espace mémoire, pour le code , les données (initialisées et non initialisées) et la pile.
  - Charger le code et les données initialisées à partir du fichier exécutable.
  - Faire les translations nécessaires(réalisée par le chargeur).