Gestion de la mémoire

Thomas Vantroys
thomas.vantroys@polytech-lille.fr

Polytech'Lille
Université de Lille

IMA4
2020 - 2021

Rôle du gestionnaire de la mémoire

- Optimisation de l'utilisation de la mémoire principale (RAM) ;
- Le plus grand nombre possible de processus actifs doit y être gardé, de façon à optimiser le fonctionnement en multiprogrammation
  - Connaître les parties libres et/ou occupées ;
  - Allouer de la mémoire au processus ;
  - Récupérer de la mémoire en fin d'exécution ;
  - Traiter le va-et-vient entre le disque et la mémoire centrale

Concepts de ce chapitre

- Adresse physique et adresse logique
  - mémoire physique et mémoire logique
- Remplacement
- Allocation contiguë
  - partitions fixes
  - variables
- Pagination
- Segmentation
- Segmentation et pagination combinées

Adresse physique et logiques

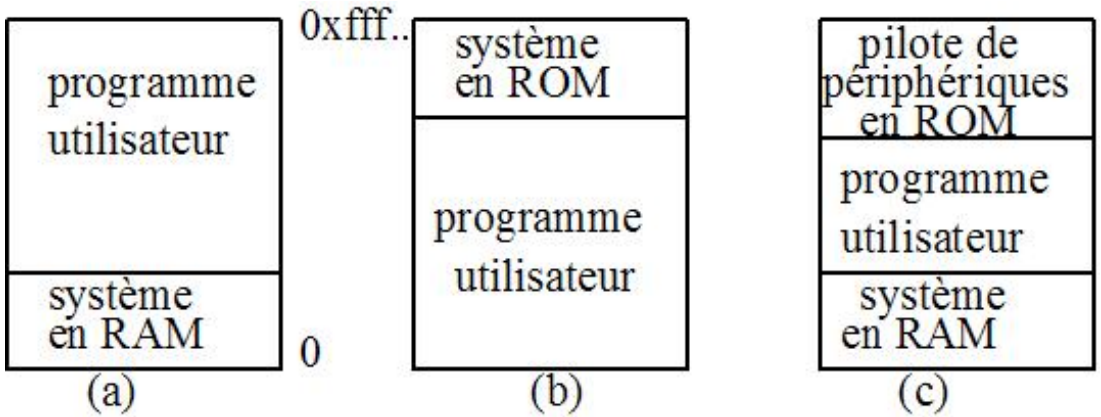
- Mémoire physique : mémoire principale (RAM) de la machine
- Adresse physique : les adresses de cette mémoire
- Mémoire logique : l'espace d'adressage d'un programme
- Adresse logique (ou adresse virtuelle) : les adresses dans cet espace
- **Besoin de séparer les concepts** : les programmes peuvent être chargés dans des positions différentes de la mémoire physique
  - donc adresse physique $\neq$ adresse logique

Liaison d'adresses

- Moment de la compilation
  - si on connaît l'emplacement du processus en mémoire au moment de la compilation on peut produite un code absolu.
- Moment du chargement
  - nécessité de produite, à la compilation, un code translatable. La liaison finale se fera au chargement
- Moment de l'exécution
  - si on peut déplacer un processus pendant son exécution, la liaison doit être retardée jusqu'au moment de l'exécution.

Mono-programmation

- Mémoire partagée entre le système d'exploitation et UN processus utilisateur ;
- Plusieurs solutions possibles :

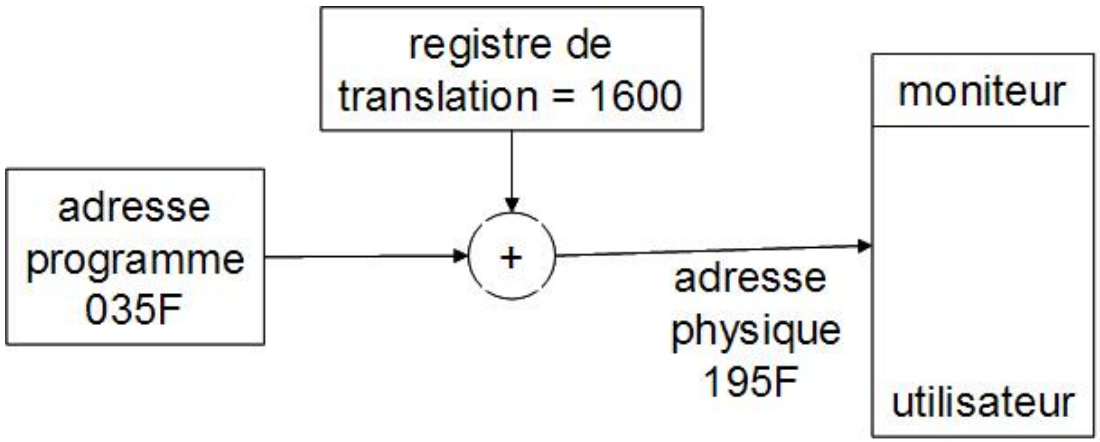


- Frappe d'une commande ➠ chargement du programme en mémoire ➠ exécution ➠ fin d'exécution ; rendre la main.

Code translatable

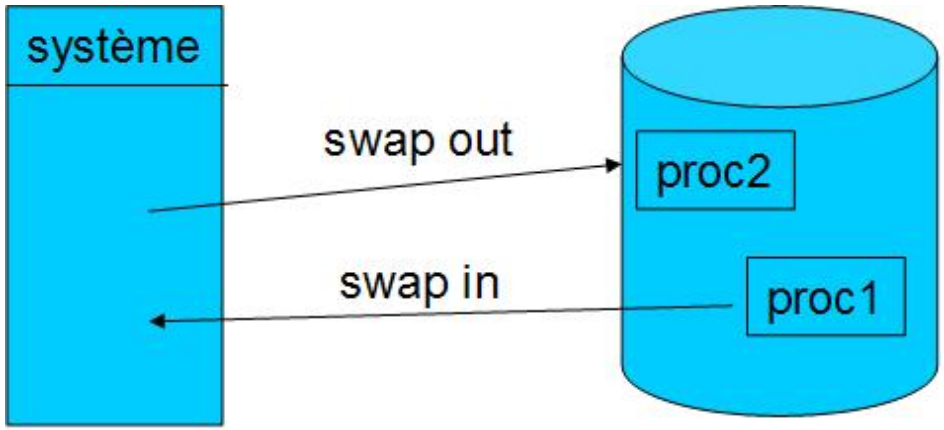
- traduction entre adresses logiques et physique effectuée par **l'unité de gestion mémoire** (MMU : *Memory Management Unit*)
- Utilisation de registres spéciaux :
  - lancement d'un processus
  - registre de translation donne la limite inférieure de la partition
  - ajout du registre de translation à chaque adresse mémoire générée (pas de modification de l'instruction)
- Avantage : les programmes peuvent être déplacés en mémoire après le début de leur exécution (changer la valeur des registres).

Code translatable



Le va-et-vient (swap)

- Mécanisme présent dès les premiers systèmes à temps partagé ;
- Recopier (*swap out*) sur une mémoire de réserve les programmes non actifs ;
- Ramener (*swap in*) les programmes en mémoire centrale dès que cela est nécessaire.



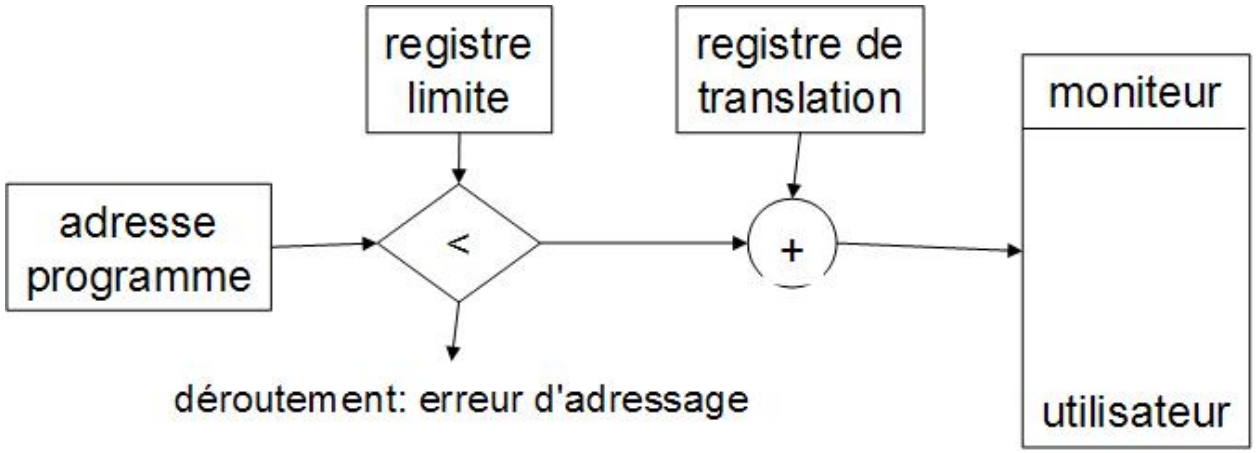
Code translatable et protection

- **problème** : les différentes tâches sont exécutées à des adresses différentes
- **solution** :
  - modification des instructions du programme chargé en mémoire
  - ex :
    - partition 1 : ajouter 300k aux des fichiers binaires chargés ;
    - partition 2 : ajouter 400k ;
    - etc
- problème : empêcher la lecture ou l'écriture dans n'importe quel mot mémoire
  - utilisation de deux registres spéciaux : registre de translation et registre de limite

Code translatable et protection

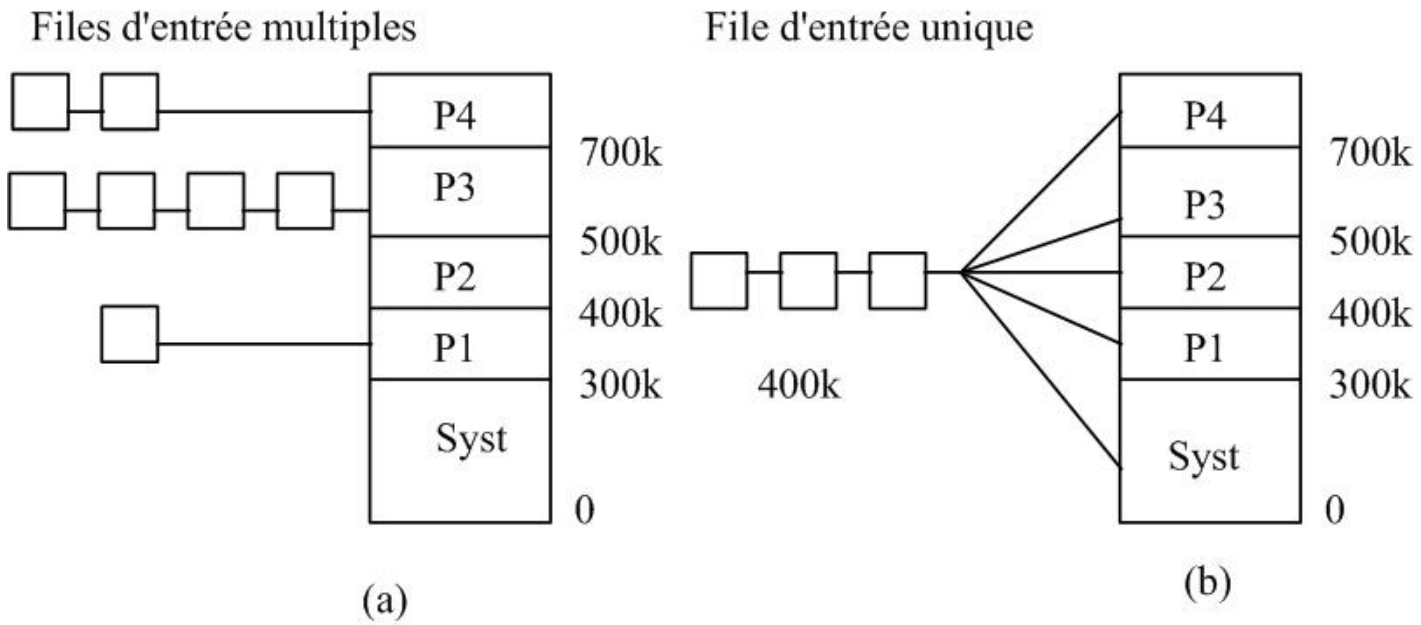
- Utilisation de registres spéciaux :
  - lancement d'un processus
  - registre de translation contient la limite inférieure de la partition
  - registre limite contient l'étendue des adresses logiques
  - vérification de non-dépassement
  - ajout du registre de translation à chaque adresse mémoire générée (pas de modification de l'instruction)
  - avantage : les programmes peuvent être déplacés en mémoire après le début de leur exécution (changer la valeur des registres)

Code translatable et protection



Multi-programmation

- Division de la mémoire en N partitions (pas forcément de tailles égales)

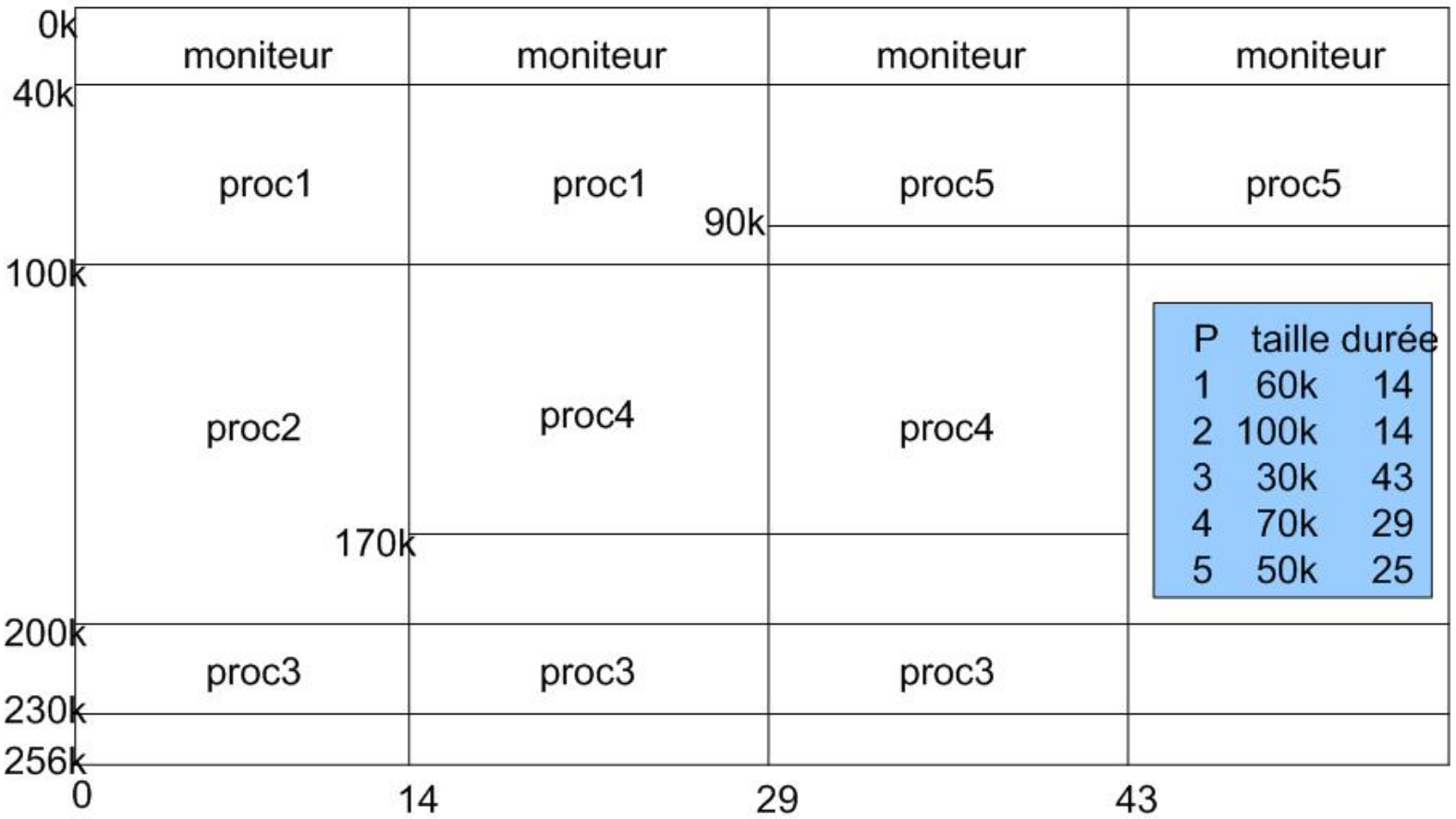


Multi-programmation

- (a) : une nouvelle tâche est placée dans la file d'attente de la plus petite partition pouvant la contenir
  - espace inutilisé perdu !
  - problème : une file d'attente peut être pleine alors qu'une autre est vide
- (b) : dès qu'une partition est libre, on y place le 1^{er} processus de la file d'attente pouvant y tenir ;
- autre solution pour (b) :
  - parcours de la file d'attente pour trouver la plus grande tâche pouvant y tenir (pénalise les petites tâches)
  - solutions :
    - garder au moins une partition de petite taille
    - interdire la non-sélection d'une tâche prête plus de k fois

Partitions variables

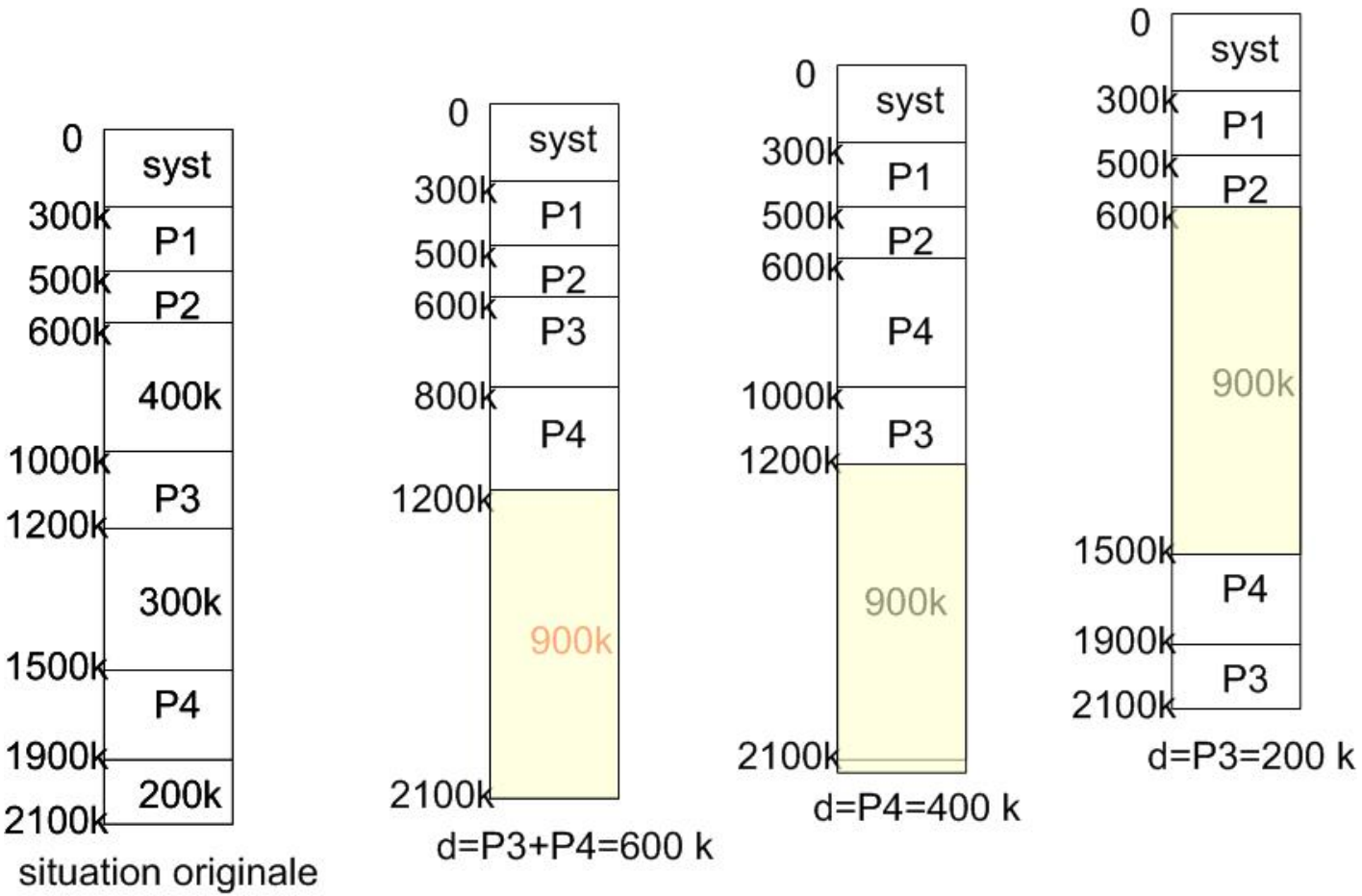
- Partitions variables
  - nombre et taille des processus varient au cours du temps
  - améliore l'usage de la mémoire MAIS complique son allocation et sa libération
- Solution
  - compactage de la mémoire mais ceci demande beaucoup de temps
- Problème
  - taille mémoire requise pour un processus ?
  - Si manque d'espace libre ?
    - déplacer le processus dans une partition plus grande
    - recopier les processus sur disque jusqu'à libération de place
    - tuer le processus


0k
40k
100k
200k
230k
256k
0
14
29
43
90k
170k
moniteur
moniteur
moniteur
moniteur
proc1
proc1
proc5
proc5
proc2
proc4
proc4
proc3
proc3
proc3
P taille durée
1 60k 14
2 100k 14
3 30k 43
4 70k 29
5 50k 25

Compactage

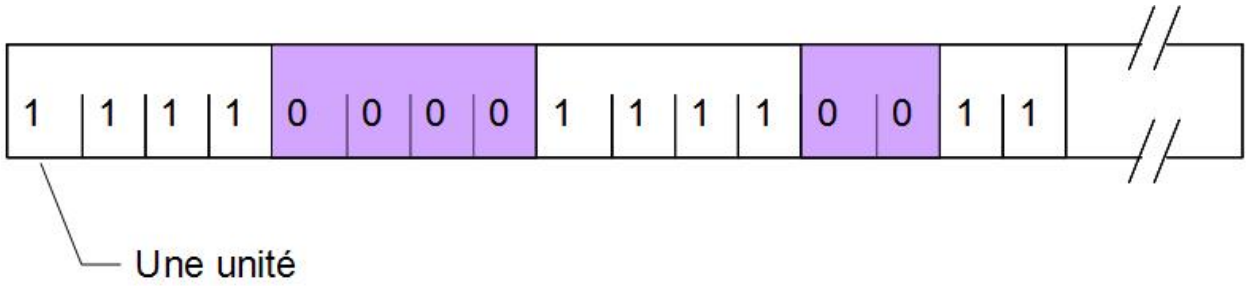
- si le programme est translable, la translation requiert seulement le déplacement des programmes et des données et ensuite la modification du registre de base de translation pour refléter la nouvelle adresse de vase
- la solution simple consiste à déplacer tous les processus vers une extrémité de la mémoire, tous les trous se déplacent alors dans l'autre sens, produisant ainsi un grand trou de mémoire disponible. Ceci peut être très couteux
- la sélection d'une stratégie de compactage optimale est très difficile

Compactage


0
syst
300k
P1
500k
P2
600k
400k
1000k
P3
1200k
300k
1500k
P4
1900k
200k
2100k
situation originale
0
syst
300k
P1
500k
P2
600k
P3
800k
P4
1200k
900k
2100k
d=P3+P4=600 k
0
syst
300k
P1
500k
P2
600k
P4
1000k
P3
1200k
900k
2100k
d=P4=400 k
0
syst
300k
P1
500k
P2
600k
900k
1500k
P4
1900k
P3
2100k
d=P3=200 k

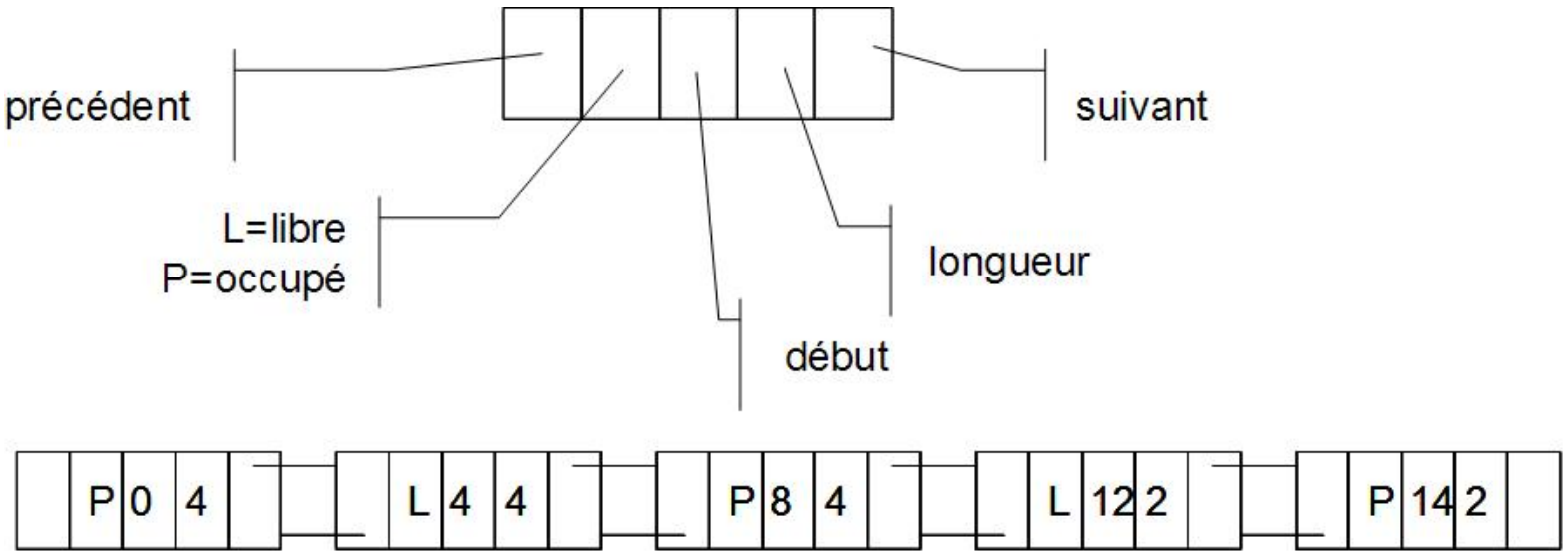
Gestion mémoire par table de bit (*bit map*)

- la mémoire est divisée en unité d'allocation
- une unité = un bit dans la table
- unité petite => grande table
- unité grande => perte de place mémoire (quand taille d'un processus non multiple de taille d'une unité)
- Problème : ramener en mémoire un processus de k unités
  - recherche de k octets consécutifs dans la table
  - lent

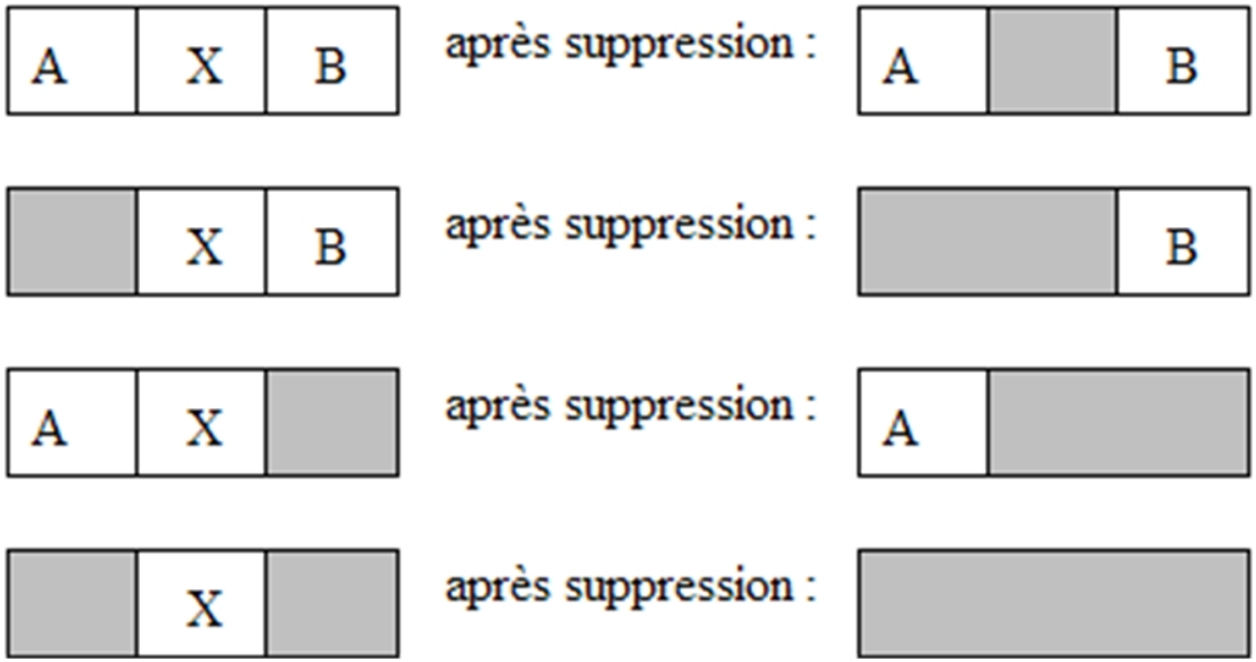


Gestion mémoire par liste chaînée

Gestion d'une liste chaînée des segments libres et occupés



Terminaison d'un processus X : 4 combinaisons



Variante : utiliser des listes séparées pour les processus et les espaces libres : accélération des algorithmes mais augmentation de la complexité et augmentation du temps de libération (MaJ des 2 listes, problème de fusion des emplacements libres)

Algorithmes de placement

- *first fit* : première zone libre suffisamment grande. Mémoire scindée en deux (une pour le processus, une l'espace mémoire inutilisé) : solution simple et rapide
- *next fit* : zone libre suivante. Variante du *first fit*. mémorisation de l'espace libre trouvé => la recherche suivante commence à cette position
- *best fit* : recherche de la plus petite zone libre qui convient (évite le fractionnement des grandes) mais plus lent (parcours de toute la liste). Paradoxalement : plus grande perte de place mémoire (multiplication de petites zones inutilisables)
- *worst fit* : plus grand résidu. Prendre toujours la plus grande zone libre. Pas de bon résultat en simulation.

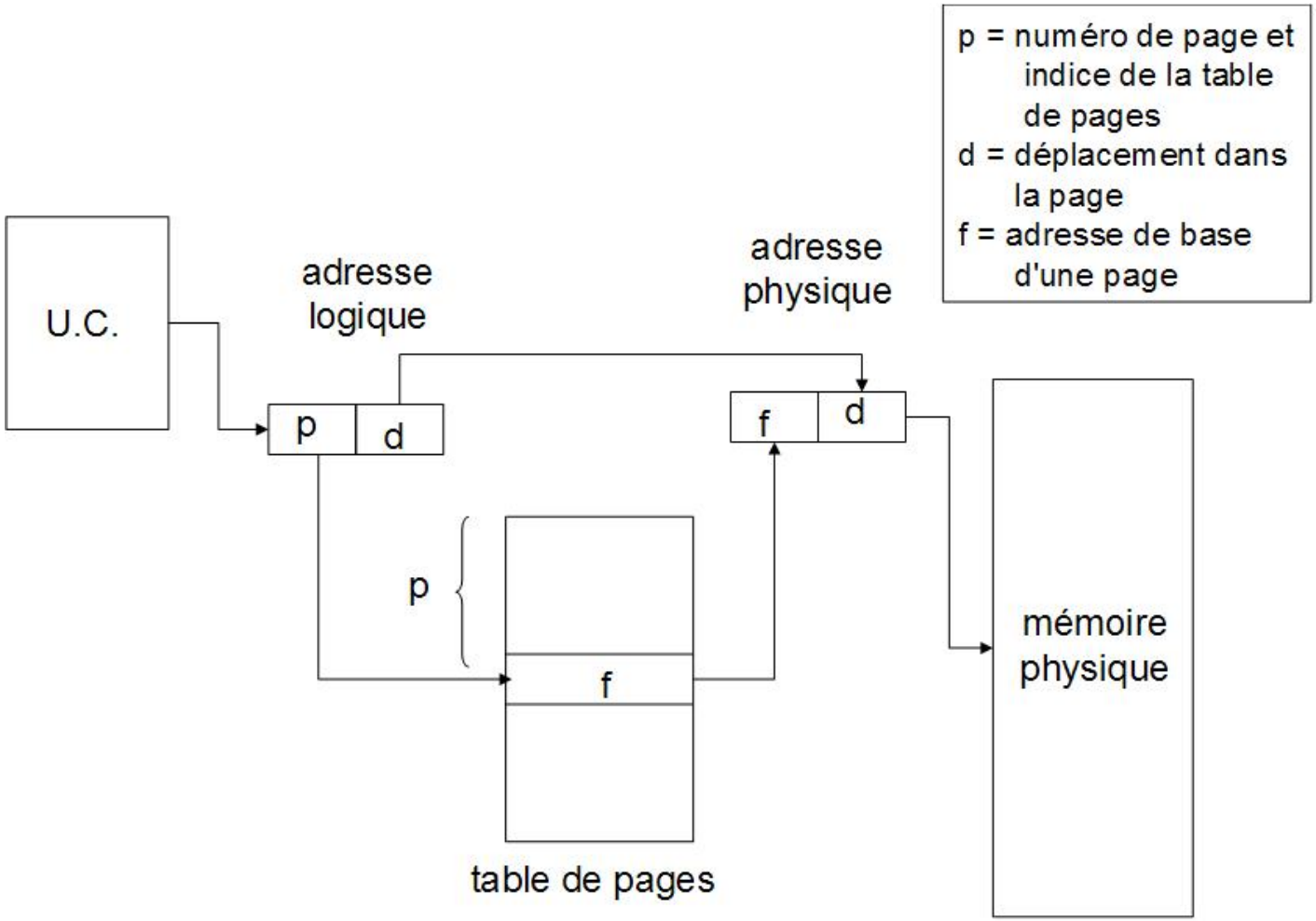
Allocation non contiguë

- Besoin de réduire la fragmentation : allocation non contiguë
  - idée de base : taille programme + données + pile ¿ mémoire disponible
  - diviser un programme en morceaux et permettre l'allocation séparée de chaque morceau
  - les morceaux sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace de la mémoire (réduction des trous)
  - garder en mémoire les parties de programme utilisées et stocker le reste sur disque (va-et-vient sur des parties du programme)
- Il existe deux techniques de base :
  - la pagination : utilisation de parties de programme arbitraires
  - la segmentation : utilisation des parties de programmes qui ont une valeur logique

Principe de la pagination

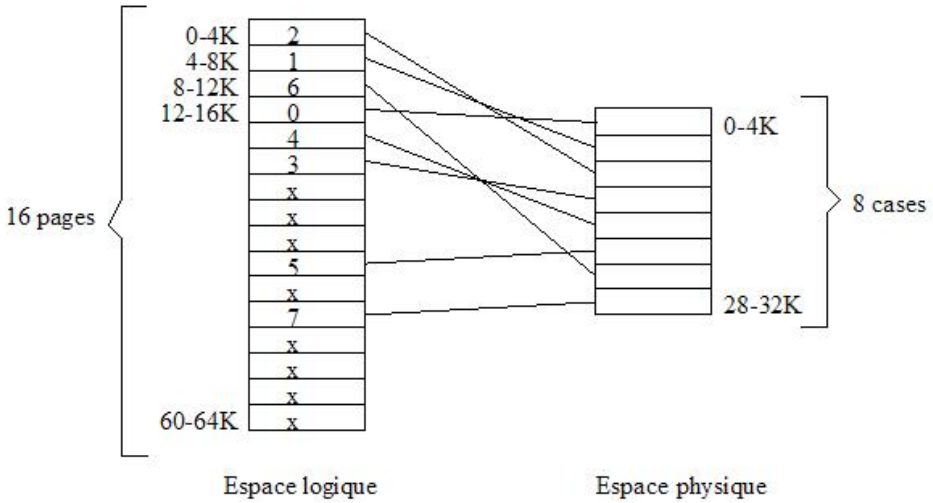
- adresse manipulées par le programme = adresse logiques : constituent l'espace d'adressage logique
- ordinateur sans mémoire virtuelle : les adresses sont directement placées sur le bus de la mémoire provoquant la lecture ou l'écriture à l'adresse spécifiée.
- avec mémoire logique : adresses envoyées à l'unité de gestion mémoire (*MMU : Memory Management Unit*) : traduit les adresses logiques en adresse physiques.

Principe de la pagination



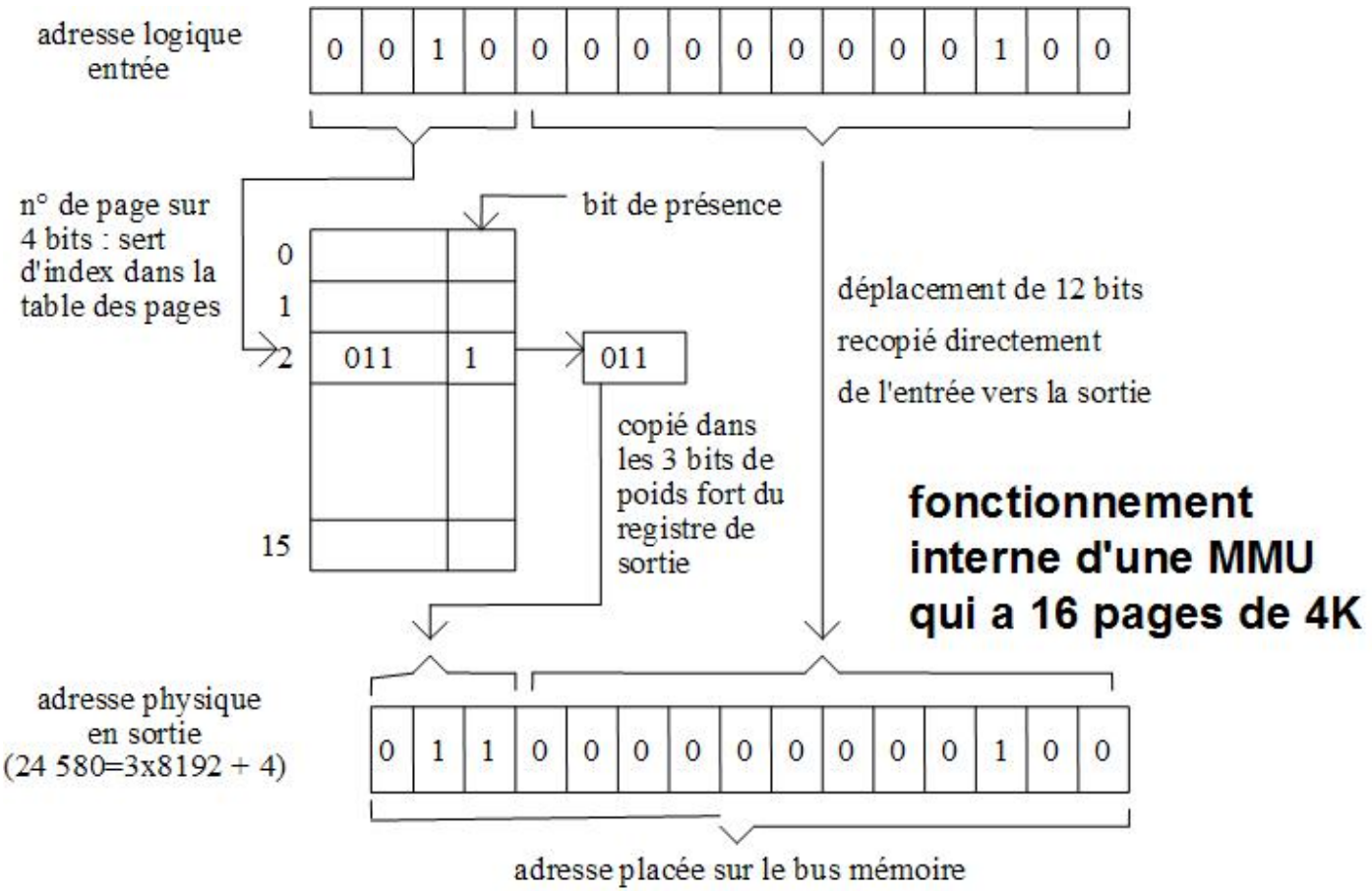
Exemple

- adresses logiques sur 16 bits : entre 0 et 64k
- adresses physiques sur 8 bits : entre 0 et 32k
- possibilité d'écrire des programmes de 64k mais pas de les charger en mémoire.
- espace logique divisé en pages
- mémoire physique divisée en cases mémoires (*page frames*)
- pages et cases mémoires sont de même taille
- table des pages : donne le "mappage" entre l'espace virtuel et l'espace physique
- Exemple : taille pages = 4k
  - 64k = 16 pages
  - 32k = 8 cases



`MOVE REG 8192` (8k) transformée en `MOVE REG 24576` (24k) car 8-12k mappée sur la case 6 (24-28k)
8 cases physiques ne peuvent mapper que 8 pages logiques. Une page non mappée, la MMU provoque un déroutement (`trap`) pour défaut de page.

Fonctionnement interne d'une MMU



La table des pages

Problèmes

1. la table des pages peut être très grandes.
   exemple :
   - adresse logiques sur 32 bits et pages de 4k
   - possibilité d'adressage logique de 0 à 2^{32}
   - nombre de page $= \frac{2^{32}}{4*2^{10}} = 2^{20}$ pages $=> 2^{20}$ entrées dans la table (plus d'un million ! !)

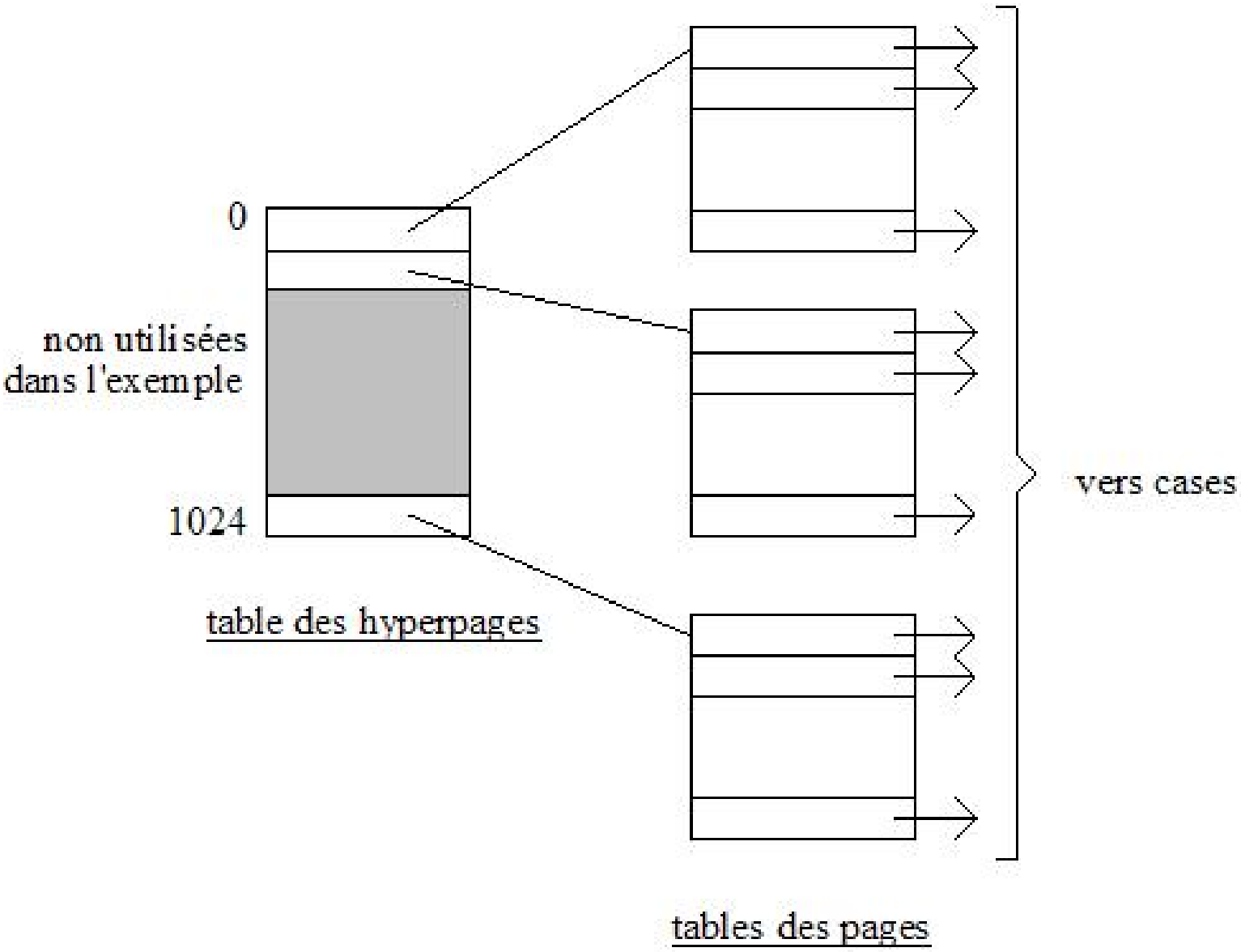
   Rappel :
   - chaque processus a besoin de sa propre table des pages
   - adresses logiques sur 64 bits donnent la possibilité d'adresser 2^{64} adresses logiques
2. le mappage doit être rapide : doit être effectué à chaque référence mémoire

La table des pages

La table des pages à plusieurs niveaux

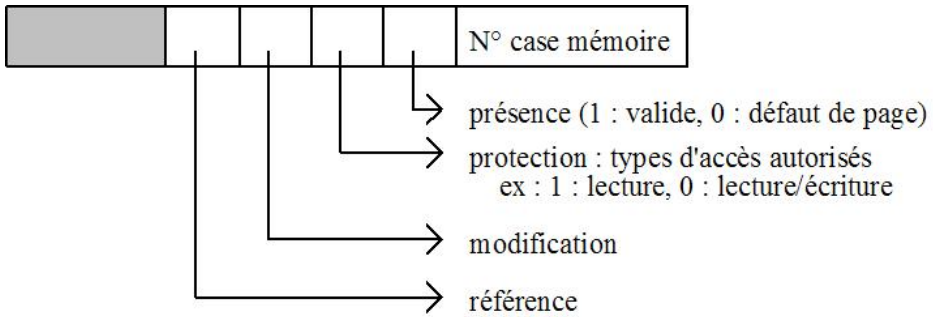
but : éviter d'avoir des pages trop grandes en mémoire exemple :
adresses logiques sur 32 bits >> espace logiques adressable = 2^{32}
adresse logique divisée en 3 champs :

- PT1 = 10 bits
- PT2 = 10 bits
- Déplacement = 12 bits
- taille des pages = 2^{12} = 4k
- nombre de pages = $\frac{2^{32}}{2^{12}} = 2^{20}$


0
non utilisées
dans l'exemple
1024
table des hyperpages
vers cases
tables des pages

PT1 : index dans la table des hyperpages : permet l'accès à une table des pages de 2^{nd} niveau
PT1 sur 10 bits $=>$ 1024 entrées
espace logique de 4 Go (2^{32})
$=>$ une entrée dans la table des hyperpages représente
$\frac{2^{32}}{2^{10}} = 2^{22} = 4Mo$
PT2 : index dans la table des pages de 2^{nd} niveau. Permet l'accès à une case mémoire
avantage : limite les tables de pages aux seules parties effectivement utilisées de la mémoire logiques.

Une entrée d'une table des pages



- *bit de modification* : positionné lorsqu'une page est accédée en écriture (permet d'indiquer si une écriture sur disque est nécessaire ou non lors du retrait de la page de la mémoire)
- *bit de référence* : positionné chaque fois qu'une page est accédée en lecture ou écriture : permet au système de deécider quelle page retirer lors d'un défaut de page.

La segmentation

Permet de travailler sur un espace virtuel à plusieurs dimensions
=> plusieurs espaces d'adresse indépendants appelés **segment**
(chaque segment est une suite d'adresse contiguës de 0 à max)
ex : un compilateur utilise plusieurs tables au cours de la compilation

- table des symboles
- table des constantes
- l'arbre d'analyse syntaxique du programme
- la pile utilisée pour les appels de procédure du compilateur

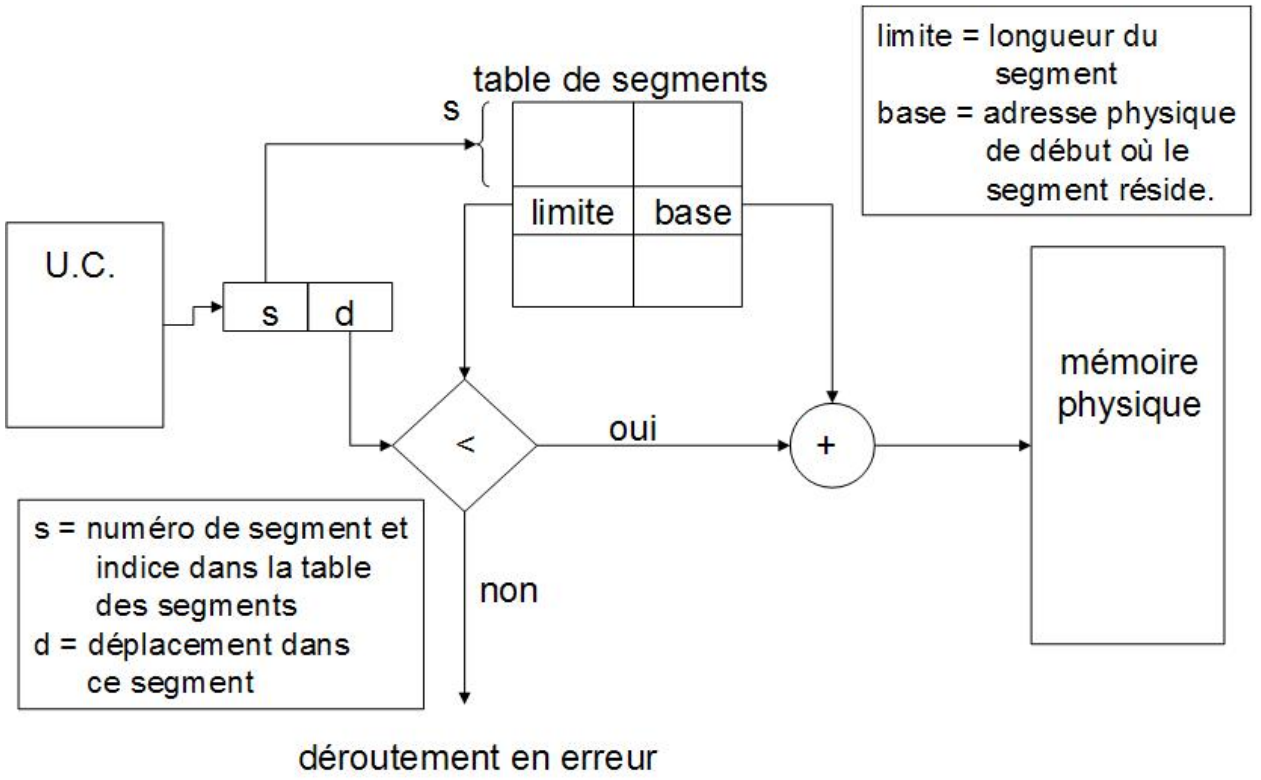
La segmentation

Problème : taille des pages varie dynamiquement

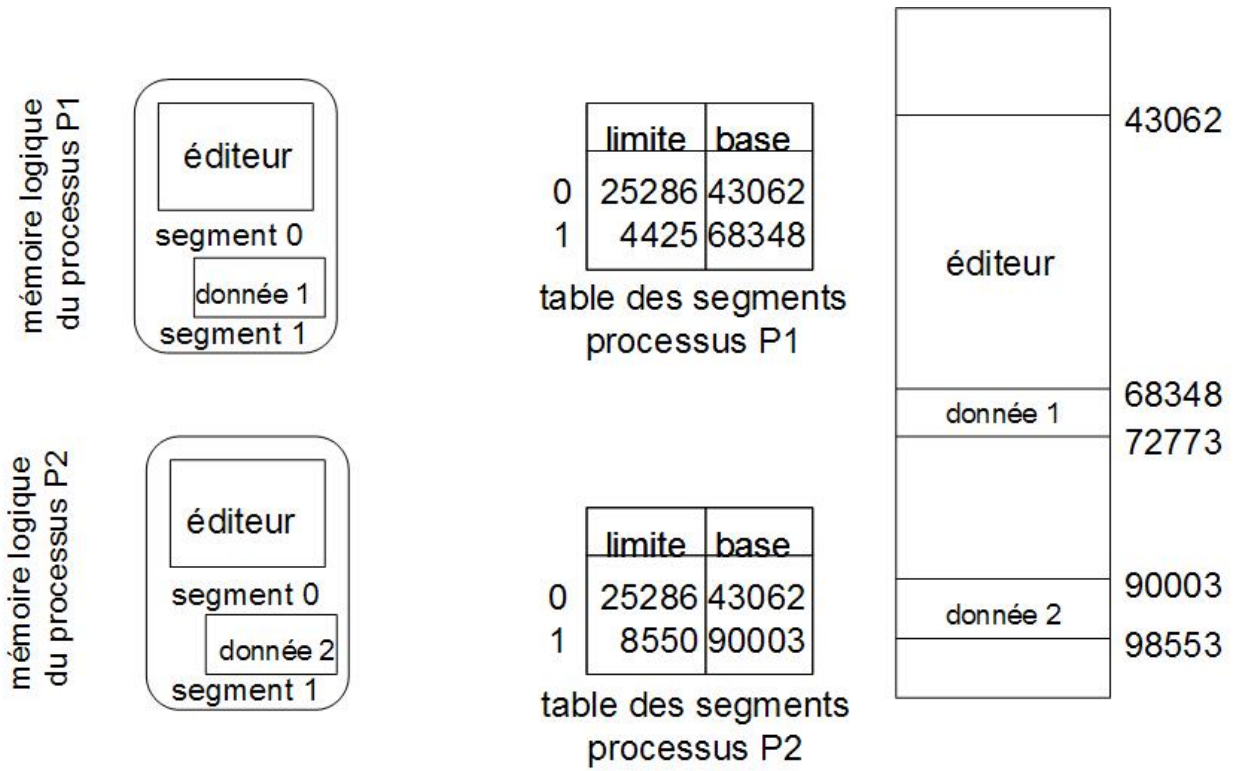
- espace à une dimension => attribuer un espace contigu à chacune des tables et gérer les recouvrements
- segmentation => un segment par table dont la taille peut varier dynamiquement (pas de problème de recouvrement puisque chaque segment à son propre espace d'adressage

=> une adresse est entièrement spécifiée par le numéro de segment et une adresse dans le segment

Méthode de base



Partage de code



Mémoire virtuelle

- Les problèmes :
  - sous utilisation de la mémoire : la totalité du programme est rarement utilisée en même temps et en permanence
  - nécessite d'exécuter des programmes plus gros que la mémoire disponible
- Les avantages :
  - dépendance moindre des programmes par rapport à la taille mémoire disponible
  - plus de processus en mémoire. Meilleur taux d'utilisation du processeur

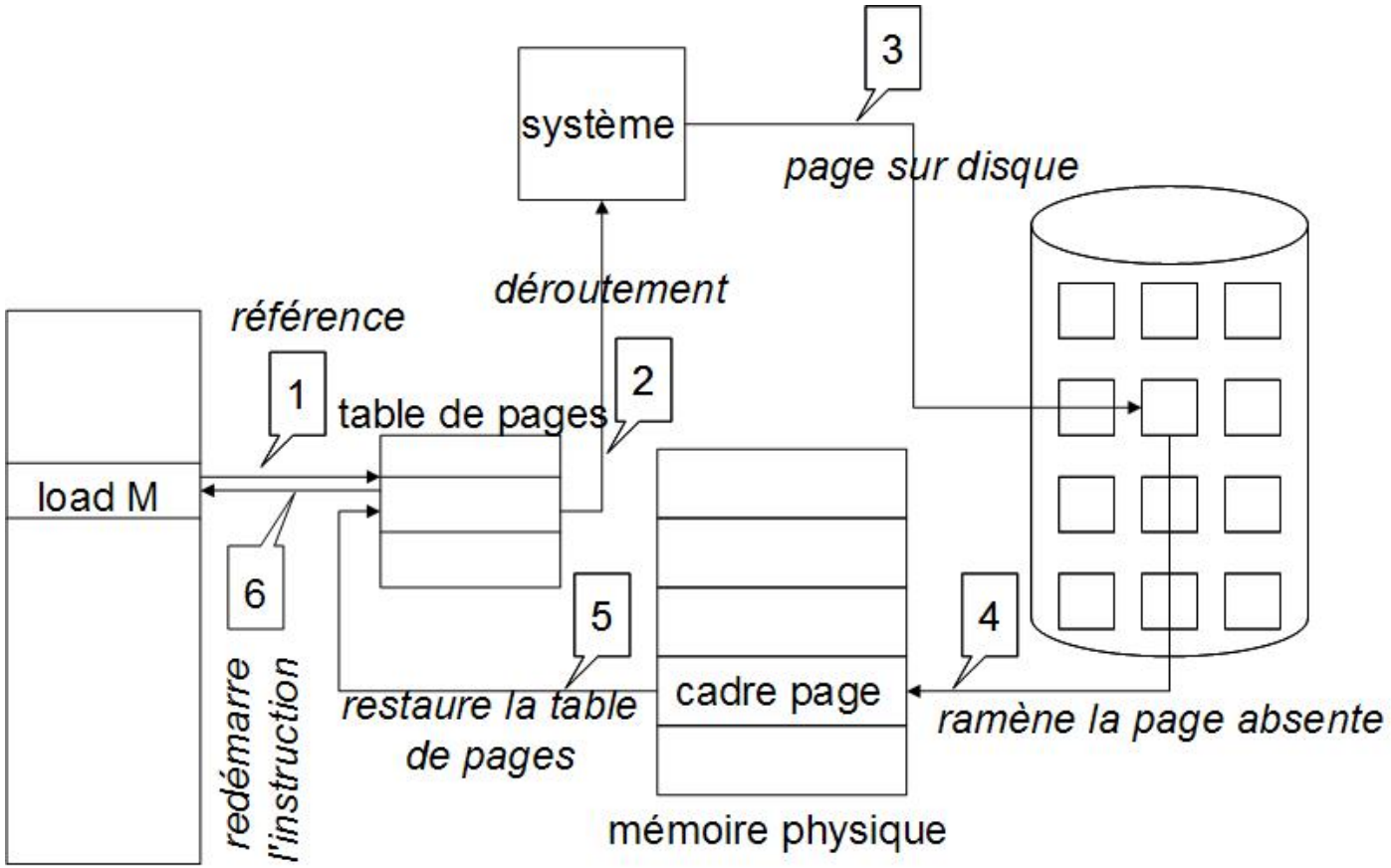
Overlays

- pas nécessité de matériel spécifique
- programme découpé en partie pour former une arborescence de façon que deux branches d'un même sommet n'aient pas besoin d'être en mémoire simultanément.
- détermination de l'arborescence effectué par le programmeur. Nécessite une connaissance détaillée du programme.

Pagination à la demande

- technique équivalent au *swap in* mais ne transfère que la page mémoire utile
- processus vu comme une séquence de pages
- distinction à faire entre les pages en mémoires et celles sur disque (bit valide/invalide)

Défaut de page



Remplacement de pages

- Trouver l'emplacement de la page désirée sur le disque
- Trouver un cadre de page libre
  - s'il en existe un, l'utiliser
  - sinon, utiliser un algorithme de remplacement de pages pour sélectionner un cadre de page *victime*
  - enregistrer la page *victime* sur le disque, modifier les tables de pages et de cadres de pages en conséquences
- Lire la page désirée dans le cadre de page libéré, modifier les tables de pages et cadres de pages
- Redémarer le processus

FIFO (*First In First Out*)

>> gestion d'une liste de pages
défaut de pages >> retrait de la page en tête de liste
et ajout de la nouvelle en queue
performances :
requiert peu de temps processeur mais ....
la page retirée peut être très utilisée
>> rarement utilisé tel quel

référencement. :	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
case 1 :	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
case 2 :	.	**0**	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
case 3 :	.	.	**1**	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**
défaut de page (15)	*	*	*	*		*	*	*	*	*	*			*	*			*	*	*

FIFO (anomalie de Belady)

Intuitivement, on peut penser que plus il y a de cases, moins il y aura de défauts de pages

avec 3 cases
9 défauts

référencement. :	0	1	2	3	0	1	4	0	1	2	3	4
case 1 :	**0**	0	0	**3**	3	3	**4**	4	4	4	4	**4**
case 2 :	.	**1**	1	1	**0**	0	0	**0**	0	**2**	2	2
case 3 :	.	.	**2**	2	2	**1**	1	1	**1**	1	**3**	3
défaut de page:	*	*	*	*	*	*	*			*	*	

référencement. :	0	1	2	3	0	1	4	0	1	2	3	4
case 1 :	**0**	0	0	0	0	0	**4**	4	4	4	**3**	3
case 2 :	.	**1**	1	1	1	1	1	**0**	0	0	0	**4**
case 3 :	.	.	**2**	2	2	2	2	2	**1**	1	1	1
case 4 :	.	.	.	**3**	3	3	3	3	3	**2**	2	2
défaut de page:	*	*	*	*			*	*	*	*	*	*

avec 4 cases
10 défauts

Algorithme optimal

- Il s'agit de remplacer la page qui sera réutilisée la plus tard
- cette stratégie est impossible à mettre en pratique car il est difficile de prévoir les références futures d'un programmes.

référencement:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
case 1 :	**7**	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
case 2 :	.	**0**	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
case 3 :	.	.	**1**	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1
défaut de page (9):	*	*	*	*		*		*			*			*				*		

LRU (*Least Recently Used*)

- Remplacer la page la moins récemment utilisée
- Il peut être considéré comme très bon

référérencement:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
case 1 :	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	**1**	1	1	1	1	1	1
case 2 :	.	**0**	0	0	0	0	0	0	0	**3**	3	3	3	3	3	**0**	0	0	0	0
case 3 :	.	.	**1**	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7
défaut de page (12):	*	*	*	*		*		*	*	*	*			*		*		*		