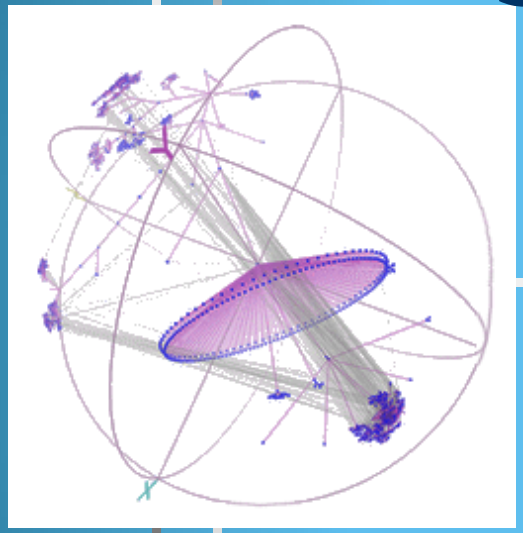
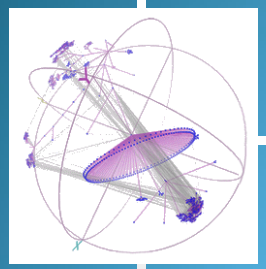


Systèmes d'exploitation
Chapitre III

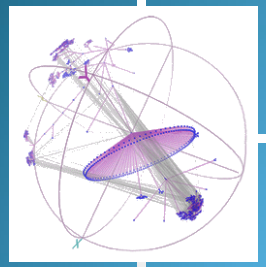
Gestion de la Mémoire



Objectifs

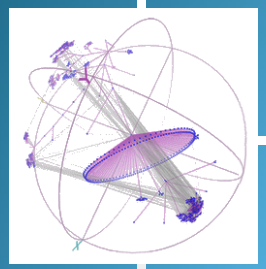
Organisation de la mémoire principale :

- Savoir quelles zones sont libres et quelles zones sont utilisées.
- Règles d'allocation : qui obtient de la mémoire, combien, quand, etc.
- Techniques d'allocation – choix des lieux d'allocation et mises à jour des informations d'allocation.
- Règles de désallocation : à qui reprendre de la mémoire, quand, etc.



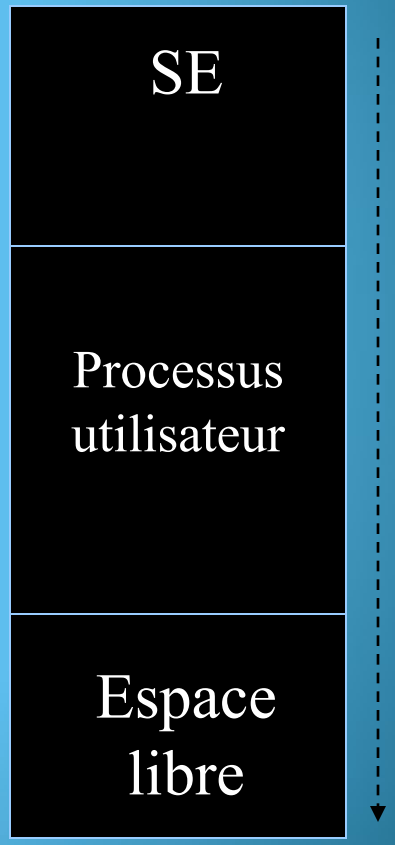
Principaux critères

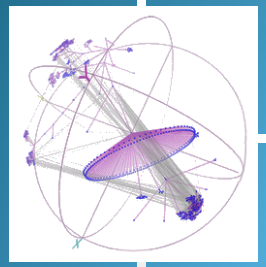
- Protection : plusieurs programmes/utilisateurs en mémoire simultanément. Il faut donc éviter les interférences.
- Partitionnement statique ou dynamique de la mémoire.
- Placement des programmes dans la mémoire : où, qui enlever si la mémoire est pleine ? Partage de code ou de données…
- Le partage doit être transparent : l'utilisateur ne doit pas savoir si la mémoire est partagée ou pas (illusion d'une mémoire infinie).



Cas mono-processus

- Système de gestion simple
- La mémoire est divisée entre le SE et les processus utilisateur.
- Le SE est protégé des programmes de l'utilisateur (pour éviter que l'utilisateur n'écrive sur le code du SE)





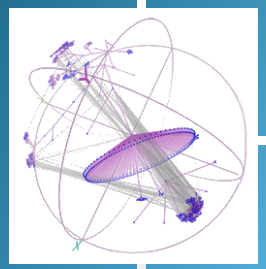
Cas mono-processus

Avantages :

- Simplicité
- SE très réduit

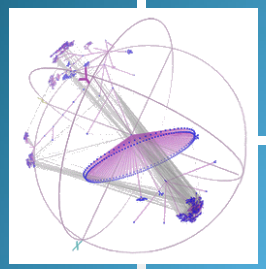
Inconvénients :

- Mauvaise utilisation de la mémoire (un seul processus) et du processeur (attente des E/S).
- Manque de flexibilité : les programmes sont limités à la mémoire existante.



Cas multi-utilisateur

- Plus d'un processus à la fois.
  - Maximiser le degré de multiprogrammation.
  - Meilleure utilisation de la mémoire et du CPU.
- Deux problèmes :
  - Réallocation.
  - Protection.

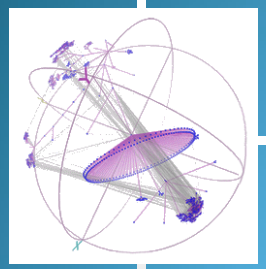


Adresses physiques et logiques

- Mémoire physique:
  - la mémoire principale RAM de la machine
- Adresses physiques:
  - les adresses de cette mémoire
- Mémoire logique:
  - l'espace d'adressage d'un programme
- Adresses logiques:
  - les adresses dans cet espace

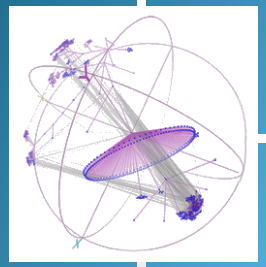
- Il faut séparer ces concepts car normalement, les programmes sont à chaque fois chargés à des positions différentes dans la mémoire

Donc adresse physique ≠ adresse logique



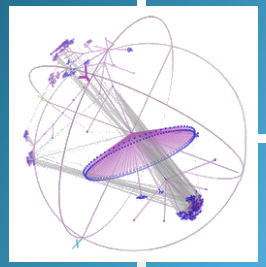
Adresses physiques et logiques

- Une adresse logique est une adresse à une location de programme
  - par rapport au programme lui-même seulement
  - indépendante de la position du programme en mémoire physique
- Plusieurs types d'adressages p.ex.
  - les adresses du programmeur (noms symboliques) sont traduites au moment de la compilation dans des
    - adresses logiques
    - ces adresses sont traduites en adresses physiques après chargement du programme en mémoire par l'unité de traduction adresses (MMU)



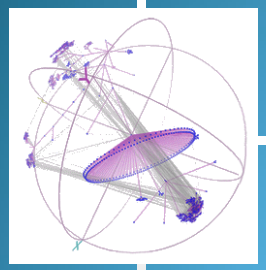
Liaison(Binding) d'adresses logiques et physiques

- La liaison des adresses logiques aux adresses physiques peut être effectuée à des moments différents:
  - Compilation: quand l'adresse physique est connue au moment de la compilation (rare)
    - p.ex. parties du SE
  - Chargement: quand l'adresse physique où le programme est chargé est connue, les adresses logiques peuvent être traduites (rare aujourd'hui)
  - Exécution: normalement, les adresses physiques ne sont connues qu'au moment de l'exécution
    - p.ex. allocation dynamique



Traduction d'adresses logiques en adresses physiques

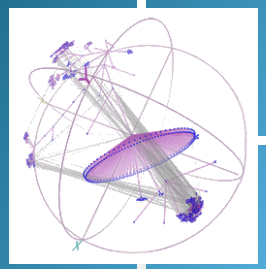
- Dans les premiers systèmes, un programme était toujours lu aux mêmes adresses de mémoire
- La multiprogrammation et l'allocation dynamique ont donc engendré le besoin de lire un programme dans des positions différentes
- Au début, ceci était fait par le chargeur (loader) qui changeait les adresses avant de lancer l 'exécution
- Aujourd'hui, ceci est fait par le MMU au fur et à mesure que le programme est exécuté
- Ceci ne cause pas d'hausse de temps d'exécution, car le MMU agit en parallèle avec autres fonctions d'UCT
  - P.ex. l'MMU peut préparer l'adresse d'une instruction en même temps que l'UCT exécute l'instruction précédente



Protection et Réallocation

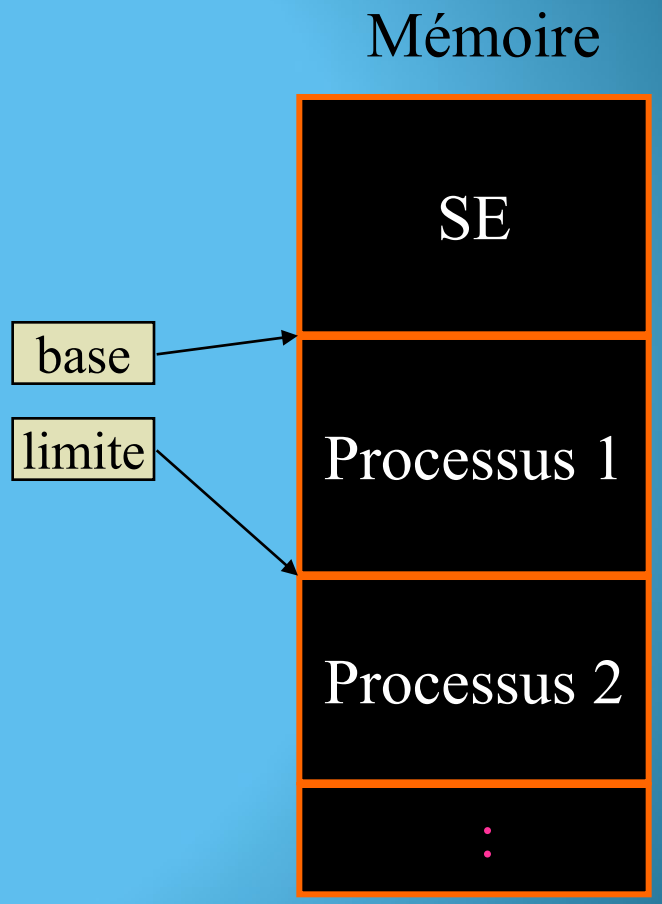
- **Protection :** empêcher qu'un processus ne puisse écrire dans l'espace d'adressage d'un autre processus.

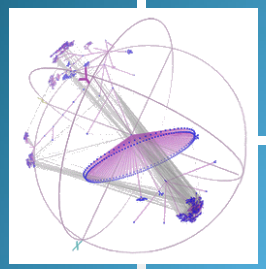
- **Réallocation :** si le même code est placé à différents endroits de la mémoire cela peut poser des problèmes si l'adressage est absolu :



Solution

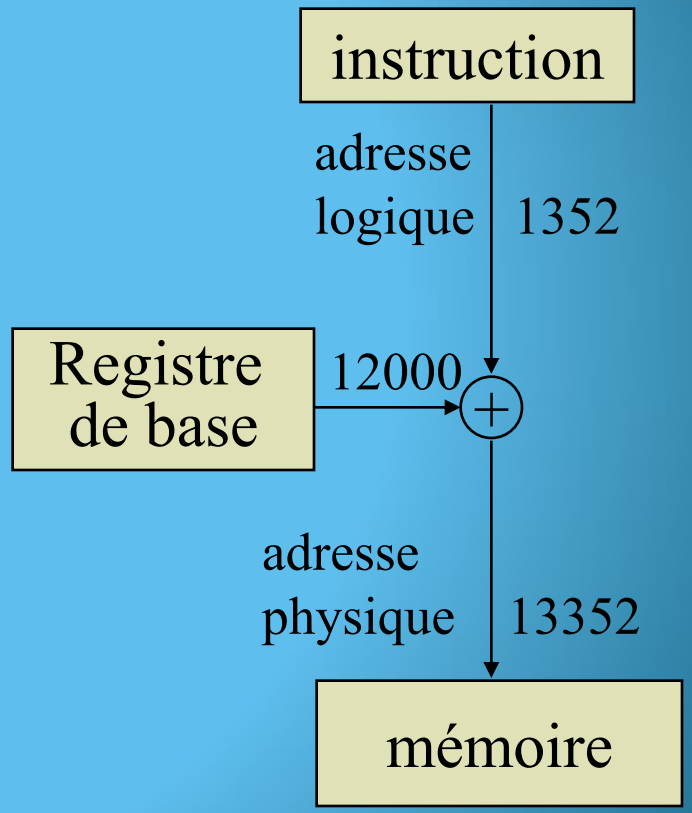
- Avec les registres base et limite, les deux problèmes sont résolus :
- Le registre base stocke l'adresse du début de la partition et le registre limite sa taille.
- Toute adresse générée doit être inférieure à limite et est ajoutée au registre base.
- Si un processus est déplacé en mémoire, il suffit alors de modifier son registre base.

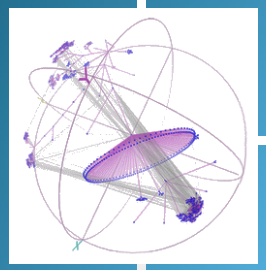




Adresses logiques/physiques

- Le concept d'adressage logique distinct de la zone d'adressage physique est central pour les systèmes de gestion de la mémoire.
  - *Adresse logique ou virtuelle* – adresse générée par le CPU.
  - *Adresse physique* – adresse utilisée par le périphérique mémoire.



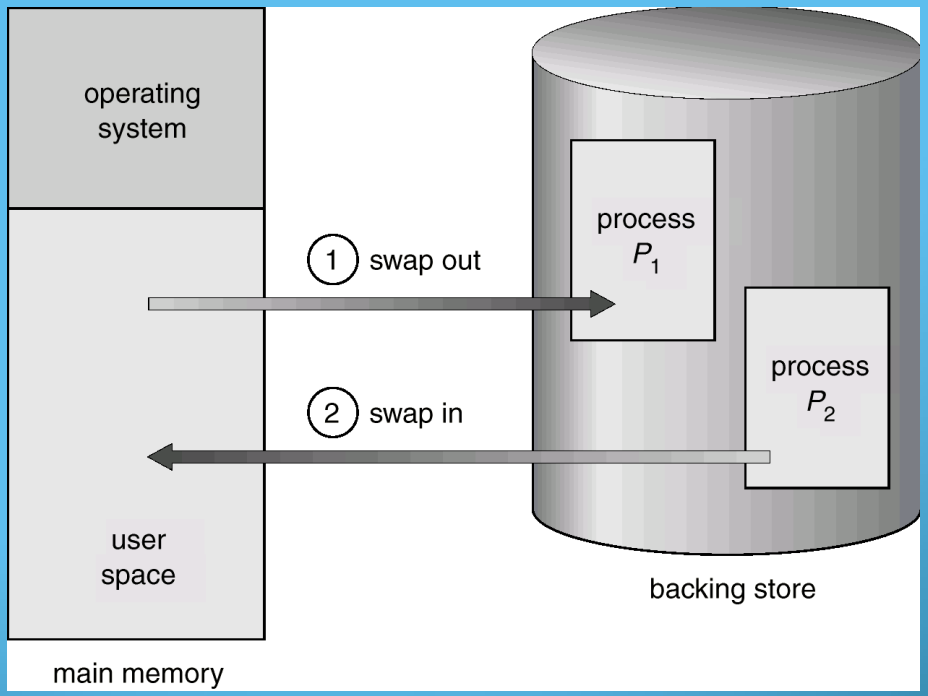


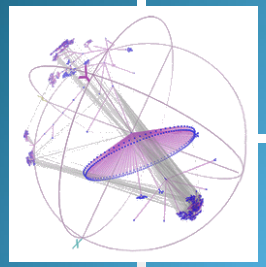
Unité de gestion de la mémoire

- L'unité de gestion de la mémoire (MMU) est un périphérique qui converti les adresses logiques en adresses physiques.
- Avec un MMU, le registre base est appelé registre de réallocation et il est ajouté à toute adresse générée par un processus au moment de l'envoi vers la mémoire physique.
- Quand l'ordonnanceur choisi un programme, les registres de réallocation et limite sont mis à jour.
- L'utilisateur n'utilise jamais d'adresse physique.

Swapping

- Un processus peut être sorti de la mémoire et stocké dans une zone annexe puis rapatrié plus tard pour terminer son exécution.





Swapping (2)

Changement de contexte : exemple

Processus = *100 Ko*

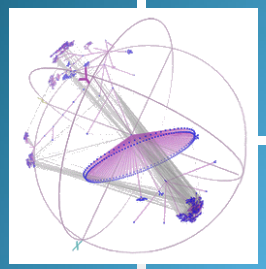
Stockage à un taux de 1Mo/s

Transfert = 100 ms

+ latence de 8 ms, temps total de swap : 216 ms.

- Pour bien utiliser le CPU, il faut que le temps d'exécution soit long par rapport au temps de swap. Tourniquet : q > 216 ms.
- Le swap standard est trop coûteux et peu utilisé en pratique.
- Des versions modifiées sont utilisées sur la plupart des systèmes (UNIX, Windows, etc.).

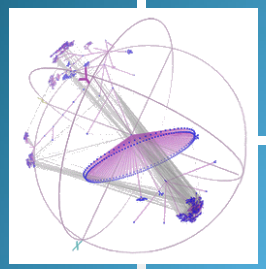
Techniques de gestion Méthodes d'allocation



- Pour un espace donné on peut choisir deux modes d'allocation :

- **Allocation contiguë :** consiste à placer la totalité d'un programme à des adresses consécutives

- **Allocation non contiguë :** consiste à fractionner le programme et à placer les différents fragments à des adresses dispersées

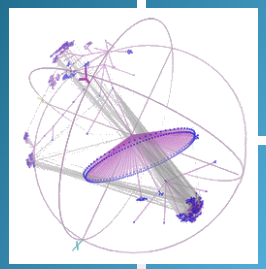


Techniques de gestion

Divers types de gestion de la mémoire ont été utilisés.

- Dans l'allocation contiguë
  - Partitionnement statique
    - Partitions fixes
  - Partitionnement dynamique
    - Partitions variable
- Dans l'allocation non contiguë
  - Segmentation
  - Pagination simple
  - Pagination à la demande

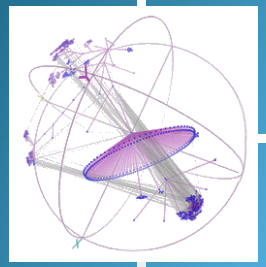
Les systèmes récents utilisent la mémoire virtuelle



Partitionnement statique

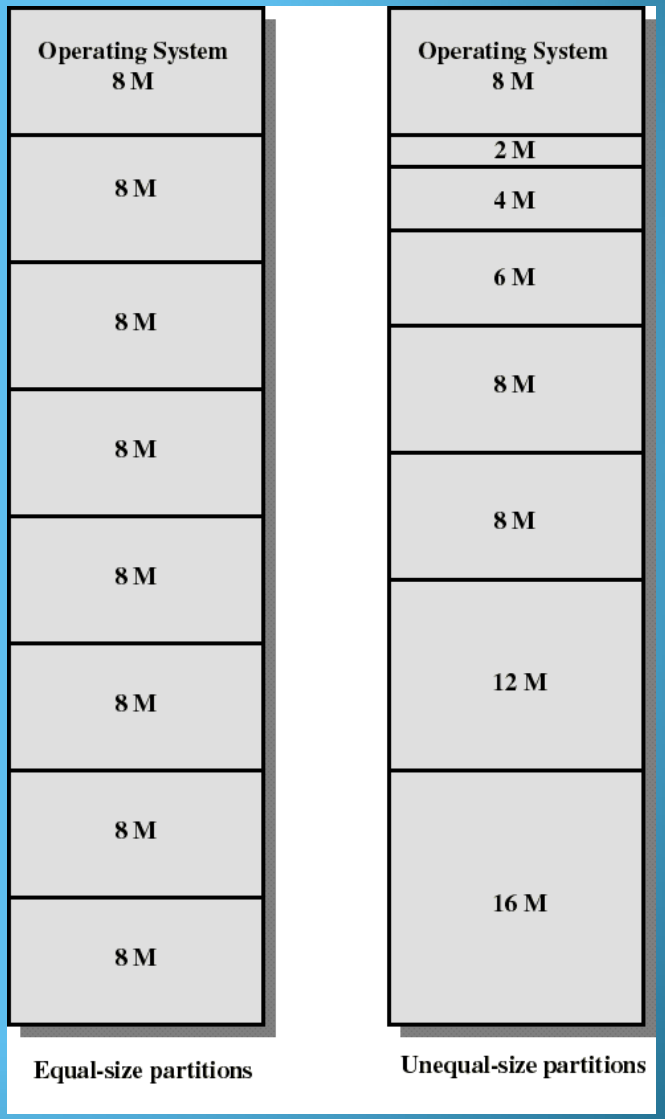
- Méthode la plus simple :
  - La mémoire est partagé en plusieurs partitions de la même taille.
  - Chaque partition peut contenir un processus (limite du nombre de processus au nombre de partitions).
  - Quand une partition est libérée, un autre processus est choisi.

Partitionnement statique
Partitions fixes

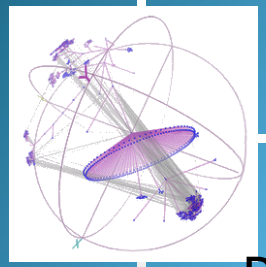


Première organisation de l'allocation contiguë

- Mémoire principale subdivisée en régions distinctes: partitions
- Les partitions sont soit de même taille ou de tailles inégales
- N'importe quel programme peut être affecté à une partition qui soit suffisamment grande



Algorithme de placement partitions fixes

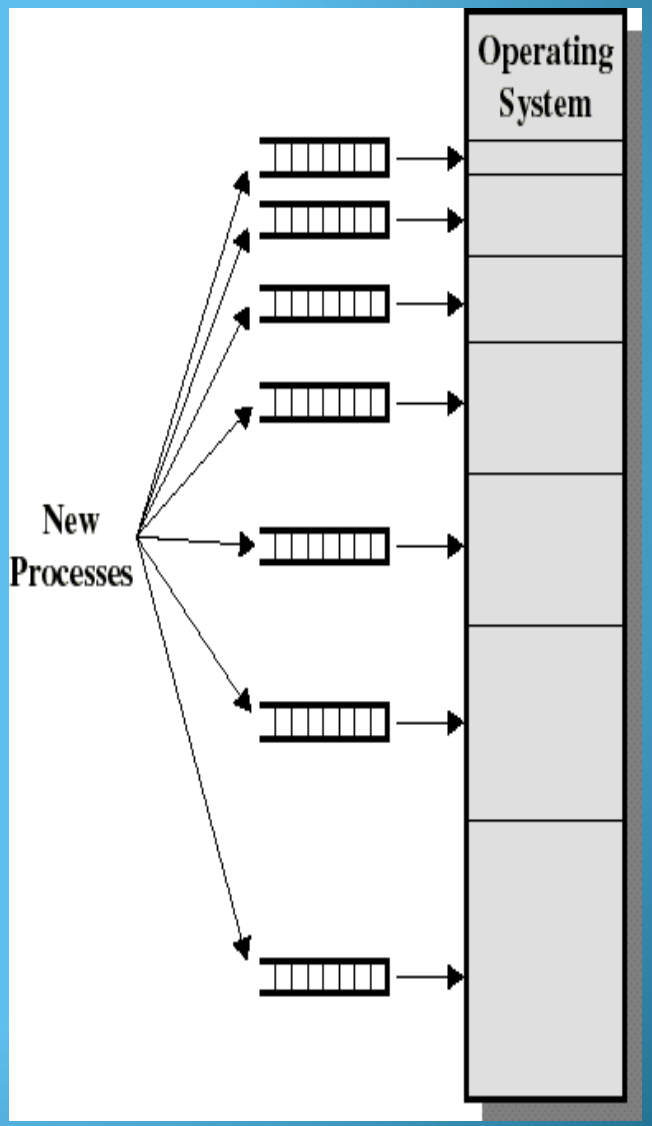


Partitions de tailles inégales: utilisation de plusieurs files d'attentes

- assigner chaque processus à la partition de la plus petite taille pouvant le contenir
- Une file par taille de partition
  - tente de minimiser la fragmentation interne

Problème:

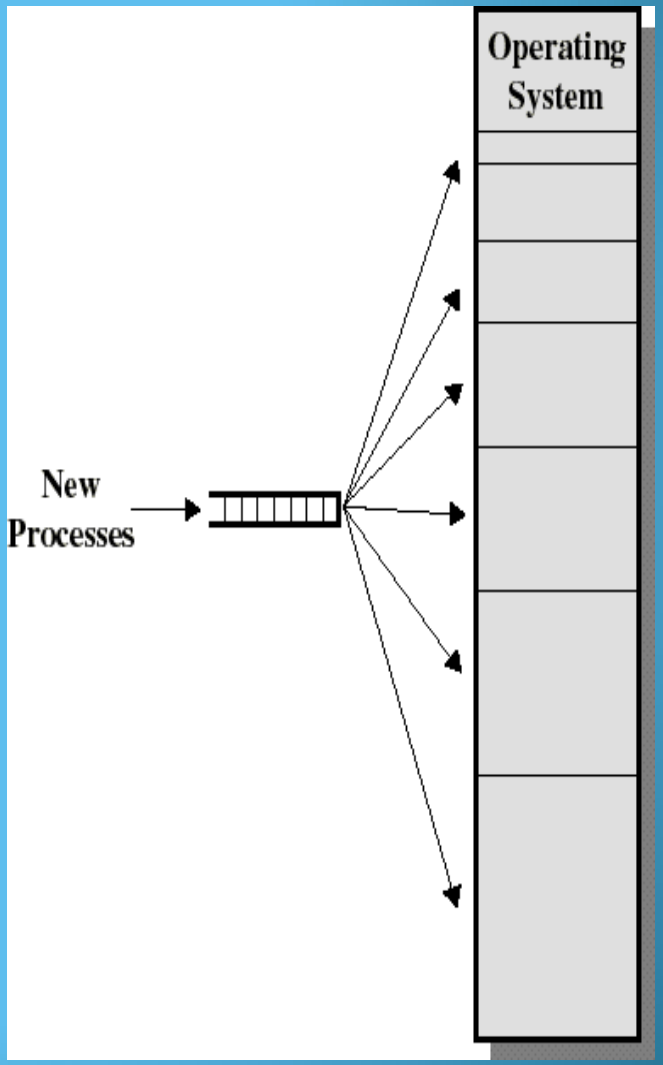
certaines files seront vides s'il n'y a pas de processus de cette taille (fragemantation externe)



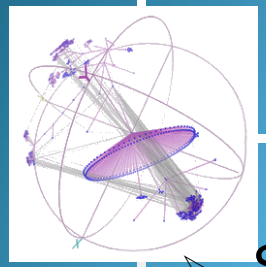
Algorithme de placement partitions fixes

Partitions de tailles inégales: utilisation d'une seule file

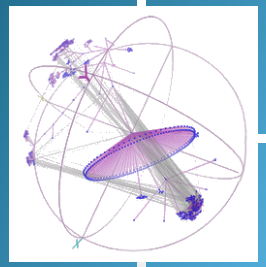
- On choisit la plus petite partition libre pouvant contenir le prochain processus
- le niveau de multiprogrammation augmente au profit de la fragmentation interne



Partitions fixes



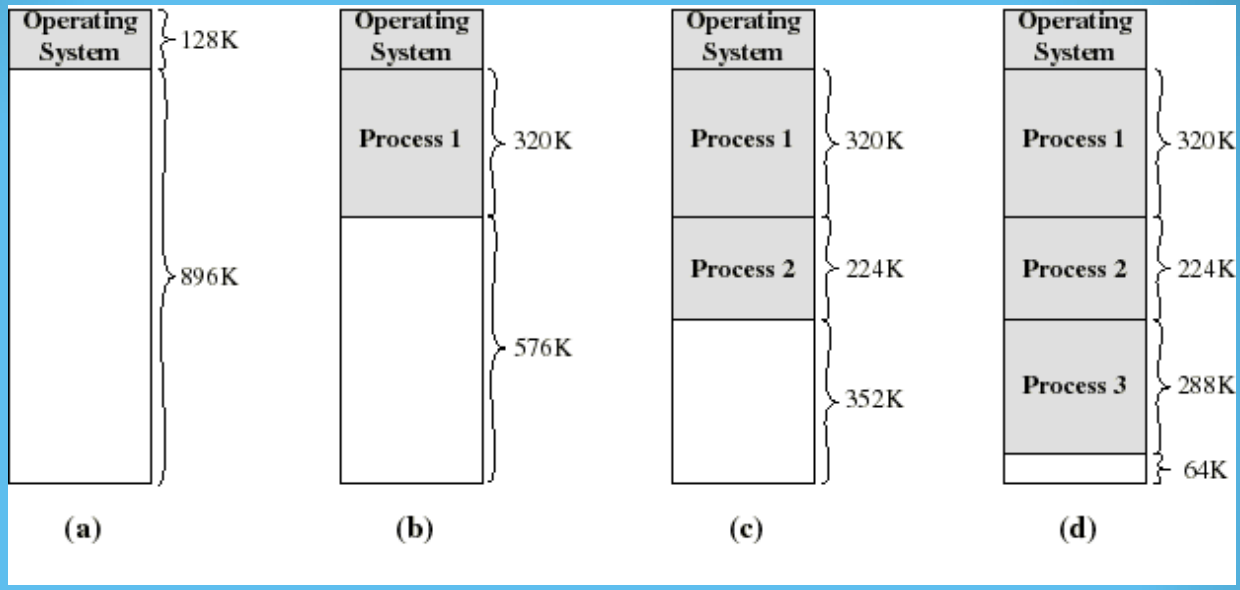
- Simple, mais...
  - Inefficacité de l'utilisation de la mémoire: tout programme, si petit soit-il, doit occuper une partition entière. Il y a fragmentation interne.
  - Les partitions à tailles inégales atténue ces problèmes mais ils y demeurent...



Partitions dynamiques

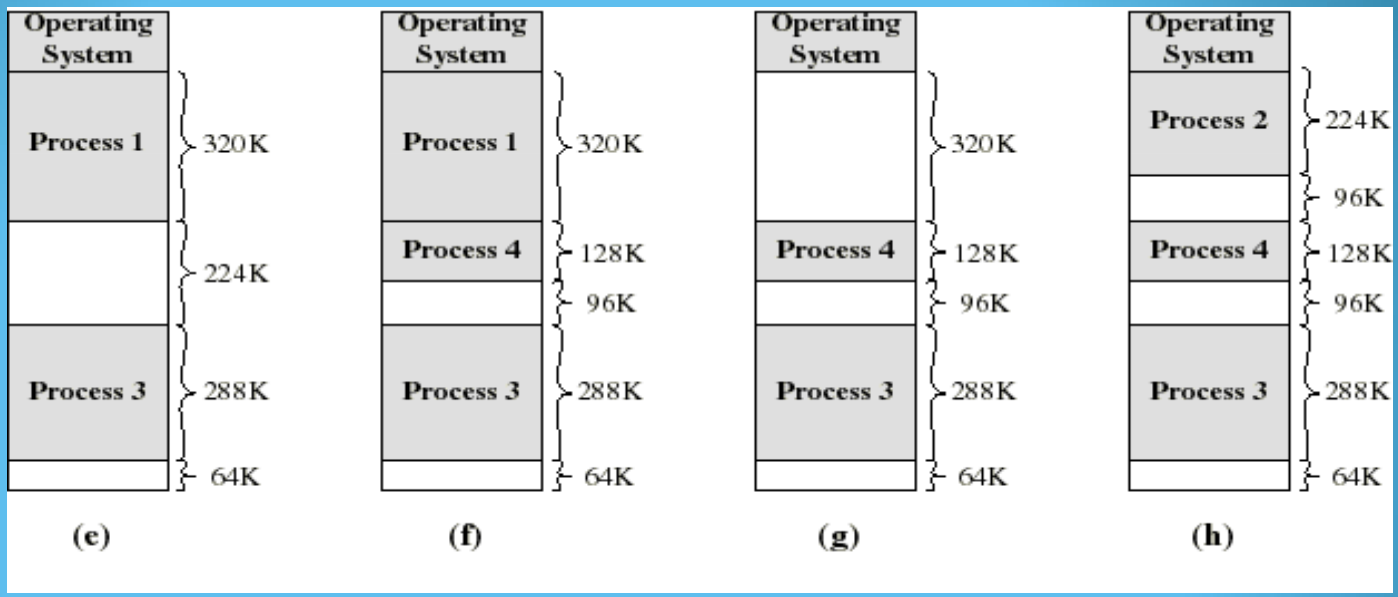
- Partitions en nombre et tailles variables
- Chaque processus est alloué exactement la taille de mémoire requise
- Probablement des trous inutilisables se formeront dans la mémoire: c’est la fragmentation externe

Partitions dynamiques

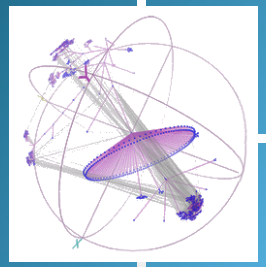


- (d) Il y a un trou de 64K après avoir chargé 3 processus: pas assez d'espace pour un autre processus
- Si tous les processus se bloquent (p.ex. attente d'un événement), P2 peut être permuté et P4=128K peut être chargé.

Partitions dynamiques



- (e-f) P2 est suspendu, P4 est chargé. Un trou de 224-128 =96K est créé (fragmentation externe)
- (g-h) P1 se termine ou il est suspendu, P2 est chargé à sa place: produisant un autre trou de 320-224=96K...
- Nous avons 3 trous petits et probablement inutiles. 96+96+64=256K de fragmentation externe
- COMPRESSION pour en faire un seul trou de 256K

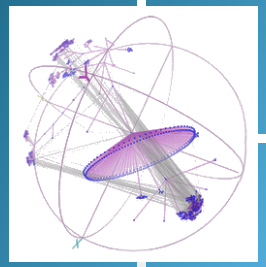


Gestion de la mémoire

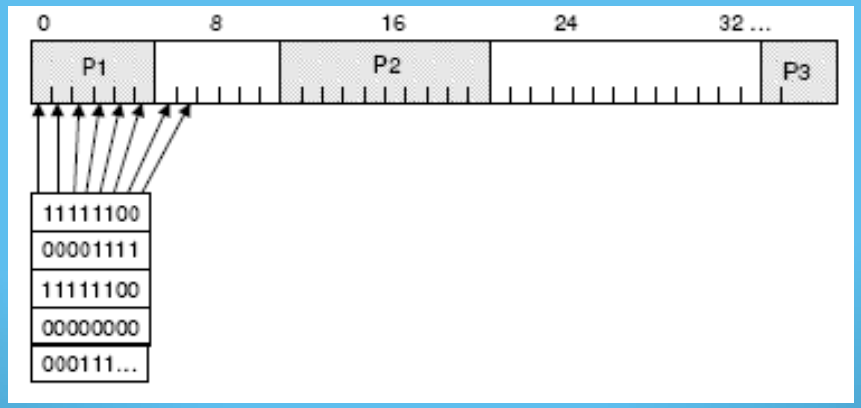
- Le système garde la trace des emplacements occupés de la mémoire par l'intermédiaire :
  - D'une table de bits ou bien
  - D'une liste chaînée.

La mémoire étant découpée en unités, en blocs, d'allocation

État de la mémoire
Tables de bits

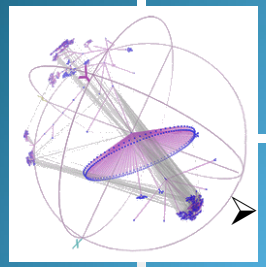


- On peut conserver l'état des blocs de mémoire grâce à une table de bits. Les unités libres étant notées par 0 et ceux occupées par un 1. (ou l'inverse).
- La technique des tables de bits est simple à implanter, mais elle est peu utilisée.
- plus l'unité d'allocation est petite, moins on a de pertes lors des allocations, mais en revanche, plus cette table occupe de la place en mémoire.

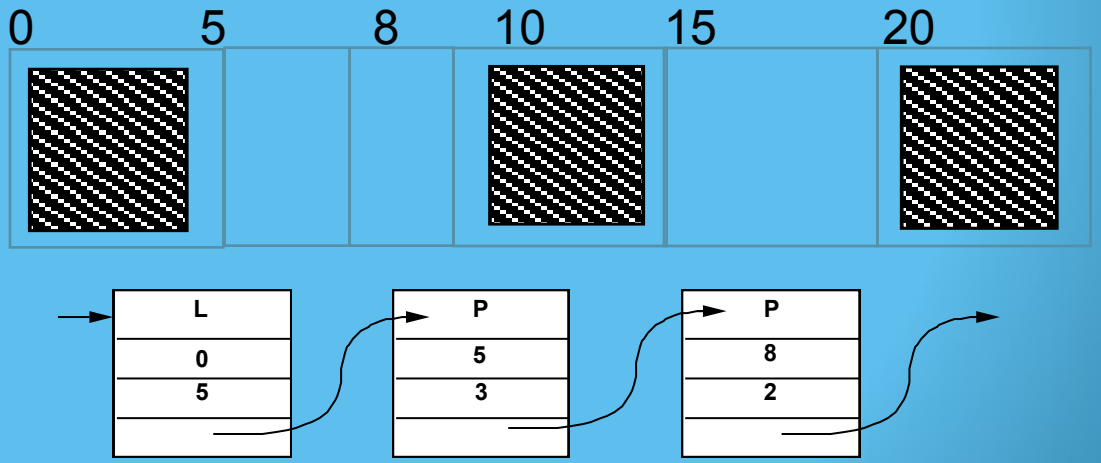


État de mémoire avec trois processus et son bitmap

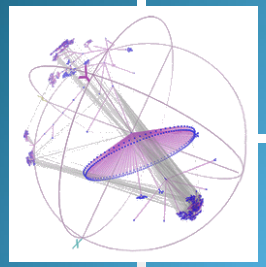
Gestion de la mémoire
Listes chaînées



- On peut représenter la mémoire par une liste chaînée de structures dont les membres sont :
  - le type (libre ou occupé),
  - l'adresse de début,
  - la longueur, et
  - un pointeur sur l'élément suivant.



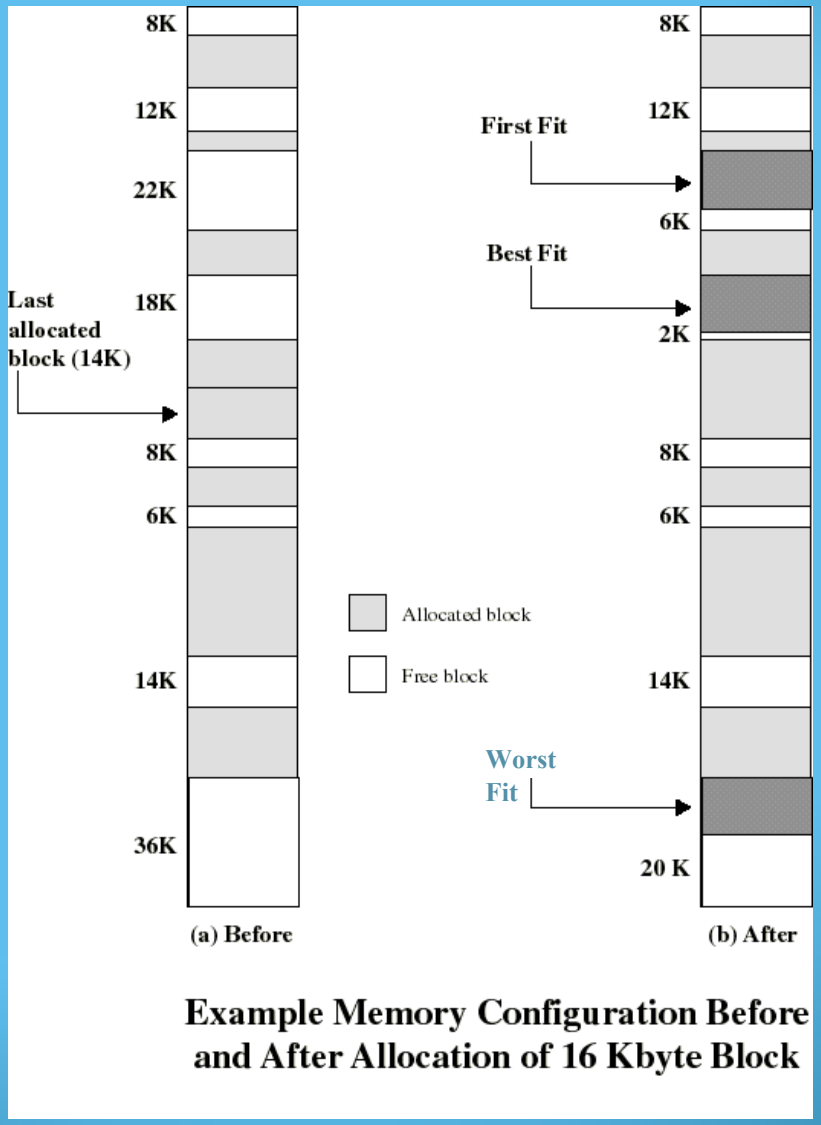
- On peut modifier ce schéma en utilisant deux listes
  - une pour les processus et
  - l'autre pour les zones libres.



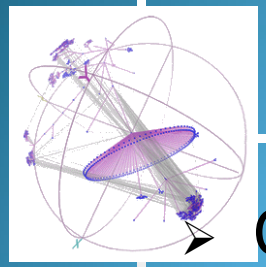
Allocation d’espace mémoires

- L'allocation d'un espace libre pour un processus peut se faire suivant quatre stratégies principales:
  - First-fit (1ère zone libre) : allouer le premier trou de taille suffisante dans la liste.
  - Next-fit (Zone libre suivante) : La recherche suivante commencera à partir de la position courante et non à partir du début.
  - Best-fit (Meilleur ajustement) : allouer le plus petit trou qui soit suffisamment grand (garder la liste triée par taille).
  - Worst-fit (pire ajustement): allouer le plus grand trou.

Algorithmes de Placement Exemple



Example Memory Configuration Before and After Allocation of 16 Kbyte Block



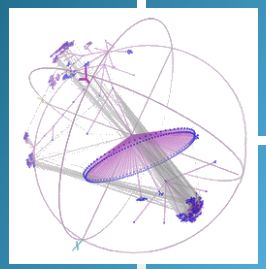
Algorithmes de placement

Quel est le meilleur?

critère principal: diminuer la probabilité de situations où un processus ne peut pas être servi, même s'il y a assez de mémoire...

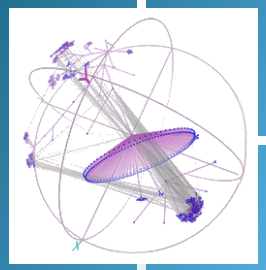
- ✓ La simulation montre qu'il ne vaut pas la peine d'utiliser les algorithmes les plus complexes
- ✓ "Best-fit": cherche le plus petit bloc possible: l'espace restant est le plus petit possible
- ✓ la mémoire se remplit de trous trop petits pour contenir un programme
- ✓ "Worst-fit": les allocations se feront souvent à la fin de la mémoire et donc trop de temps

... D'où le first fit



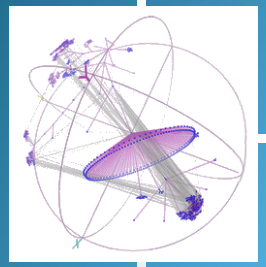
Fragmentation mémoire non utilisée

- Un problème majeur dans l'affectation contiguë:
  - Il y a assez d'espace pour exécuter un programme, mais il est fragmenté de façon non contiguë
    - externe: l'espace inutilisé est entre partitions
    - interne: l'espace inutilisé est dans les partitions



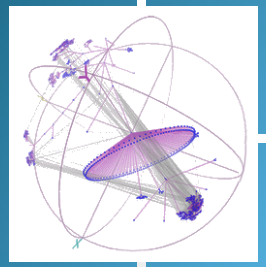
Compaction

- Une solution pour la fragmentation externe
- Les programmes sont déplacés en mémoire de façon à réduire à 1 seul grand trou plusieurs petits trous disponibles
- Effectuée quand un programme qui demande d'être exécuté ne trouve pas une partition assez grande, mais sa taille est plus petite que la fragmentation externe existante
- Désavantages:
  - temps de transfert programmes
  - besoin de rétablir tous les liens entre adresses de différents programmes



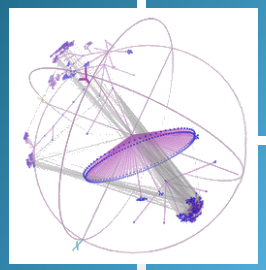
Allocation non contiguë

- A fin réduire le besoin de compression, le prochain pas est d'utiliser l'allocation non contiguë
  - 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
    - les petits trous peuvent être utilisés plus facilement



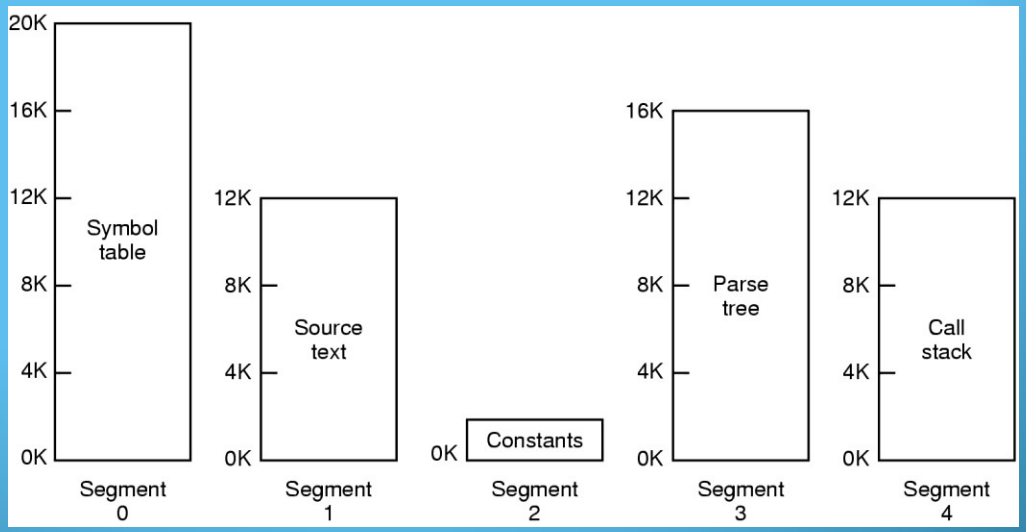
Allocation non contiguë

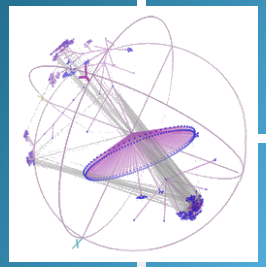
- Il y a deux techniques de base pour faire ceci: la pagination et la segmentation
  - la segmentation utilise des parties de programme qui ont une valeur logique (des modules)
  - la pagination utilise des parties de programme arbitraires (morcellement du programmes en pages de longueur fixe).
  - La combinaison des deux



Segmentation

- Segmentation : plusieurs espaces d'adressage distincts.
  - L'espace logique correspond alors à un ensemble de segments de taille variable indépendants.
  - Les segments peuvent croître ou diminuer dynamiquement et indépendamment des autres segments.



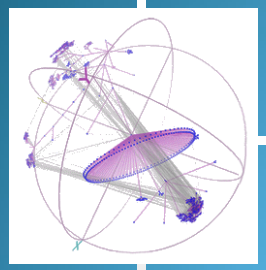


Segmentation

- Pour un processus, chaque module ou structure de données occupe un segment différent.
- Chaque segment a un nom/numéro et une longueur.
- Chaque adresse est définie par un numéro de segment et un décalage.
- Le compilateur crée les segments de manière automatique.

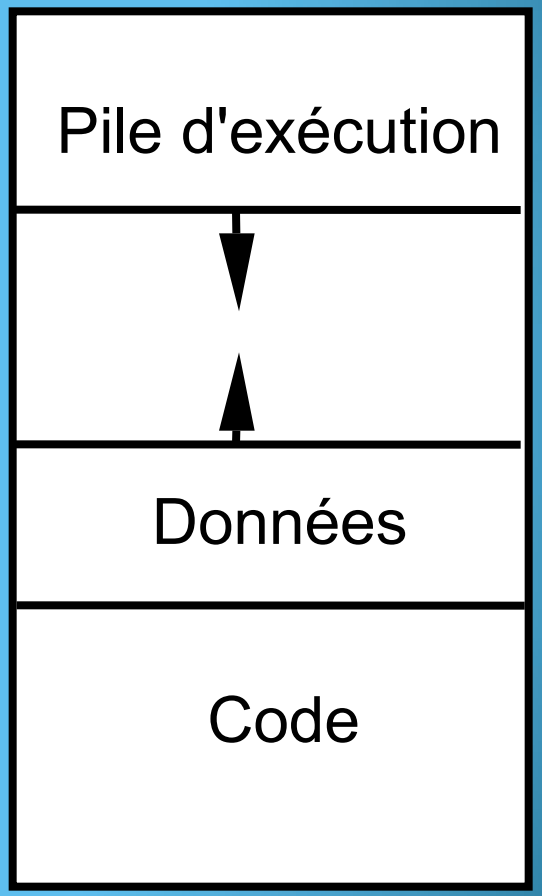
Avantages :

- La taille des segments est variable.
- Permet d'utiliser des structures dynamiques.
- Protection : les segments peuvent être associés à différents niveaux de privilèges.
- Code partagé plus simple à implémenter (segment partagé).

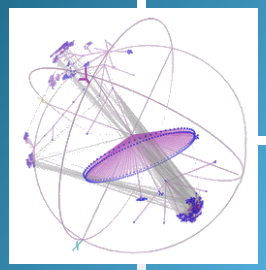


Espace de travail

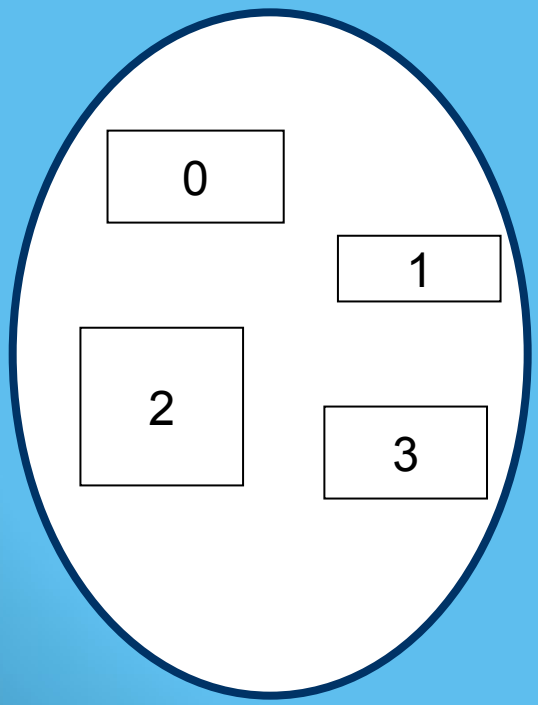
- Les processus sont composés d'un espace de travail en mémoire formé d'au moins trois segments



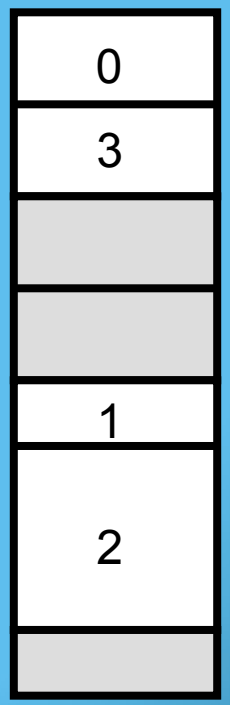
Les segments comme unités d'allocation mémoire



- Étant donné que les segments sont plus petits que les programmes entiers, cette technique implique moins de fragmentation (qui est externe dans ce cas)



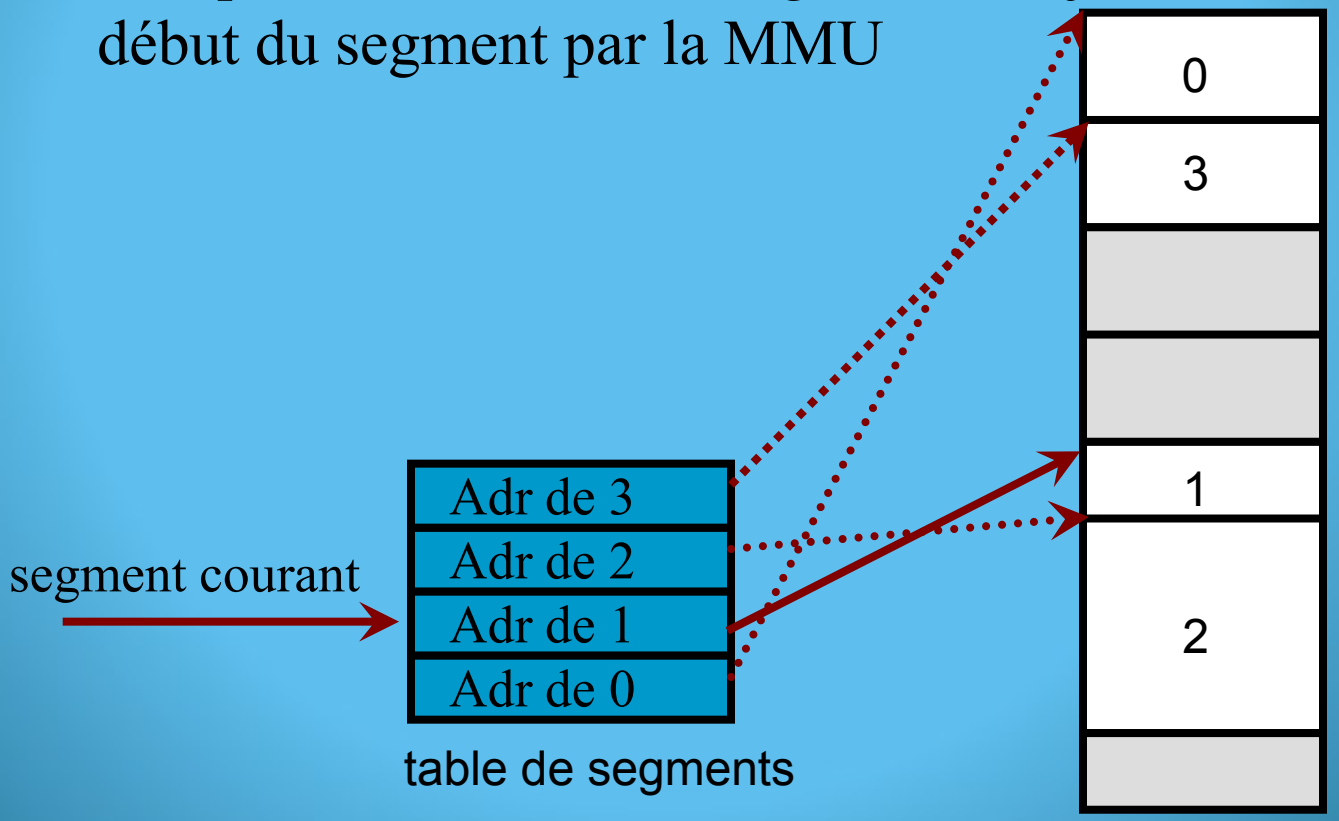
espace usager



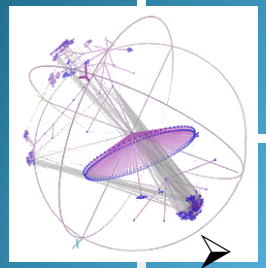
mémoire physique

Mécanisme de la segmentation

- Un tableau contient l'adresse de début de tous les segments dans un processus
- Chaque adresse dans un segment est ajoutée à l'adresse de début du segment par la MMU



Mécanisme de la segmentation



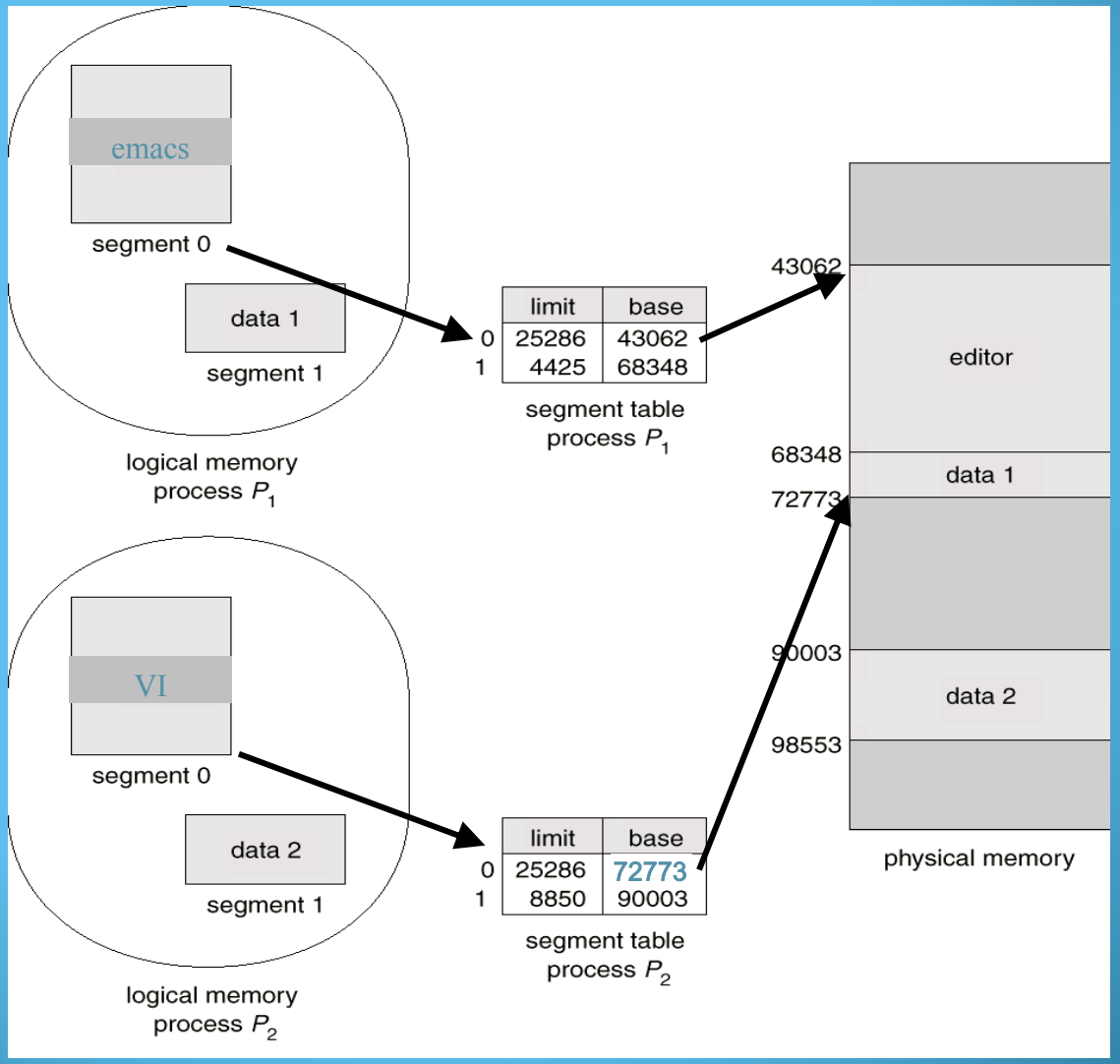
- L'adresse logique consiste el la paire:

  <No de segment, décalage>

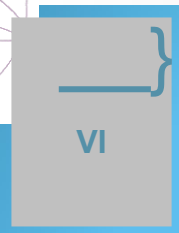
  où décalage est l'adresse *dans* le segment
- le table des segments contient: les descripteurs de segments tels que:
  - adresse de base
  - longueur du segment
  - Informations de protection
- Dans le PBC du processus il y a:
  - un pointeur à l'adresse en mémoire de la table des segments
  - le nombre de segments dans le processus

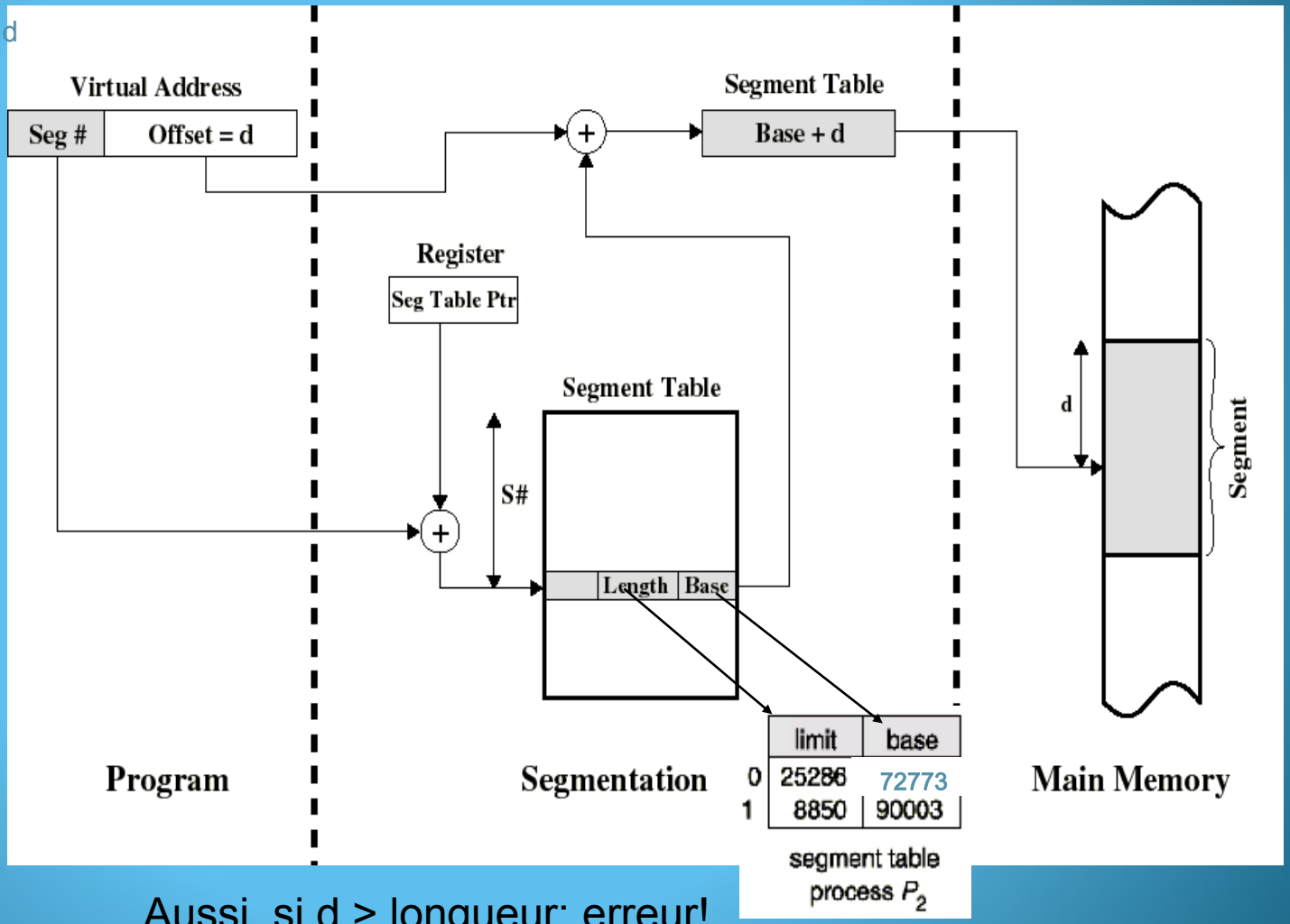
Au moment de la commutation de contexte, ces infos seront chargées dans les registres appropriés d'UCT

Exemple de segmentation

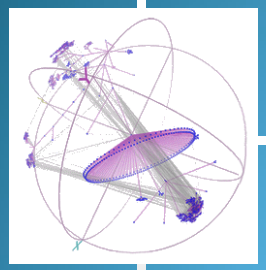


Traduction d'adresses dans la segmentation





Aussi, si d > longueur: erreur!

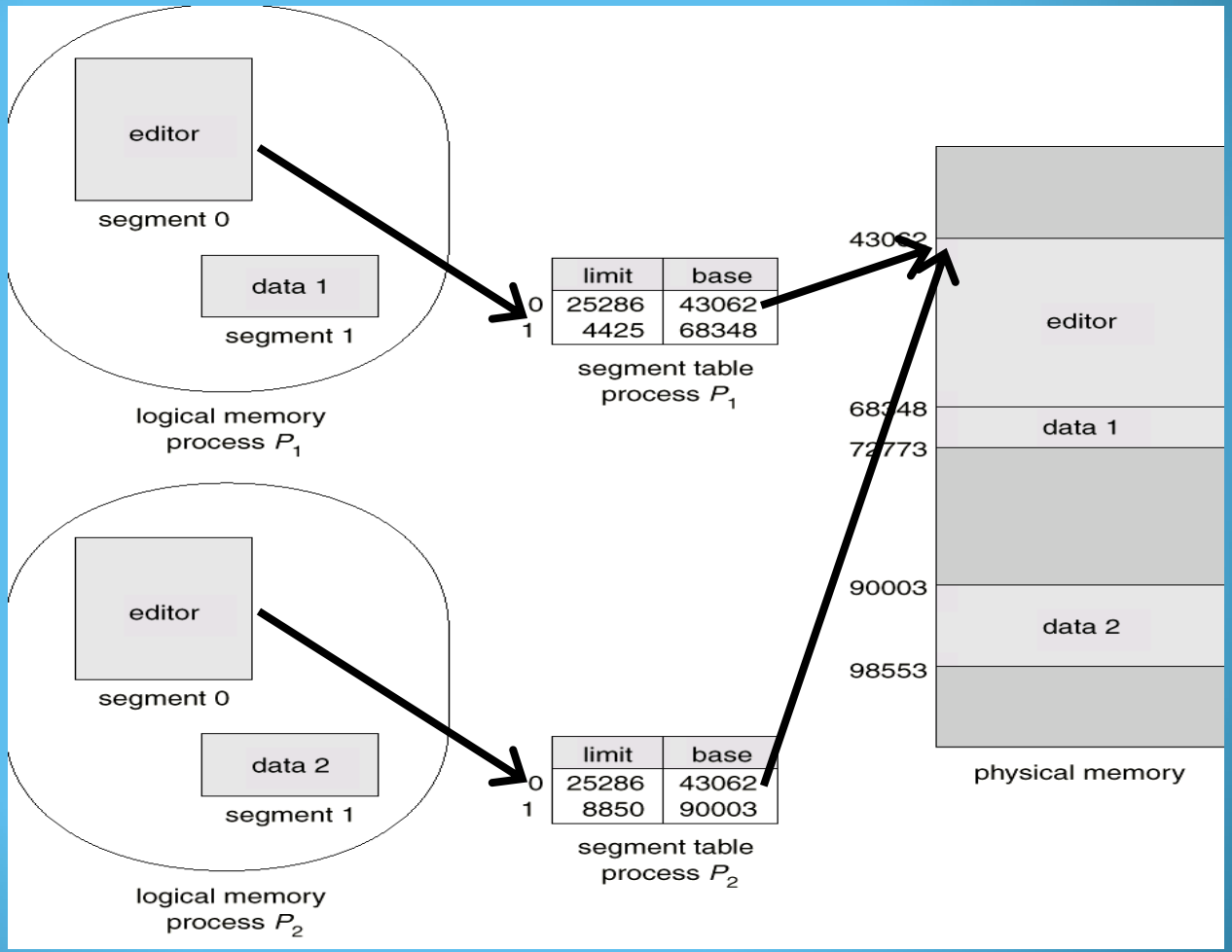


Partage de segments

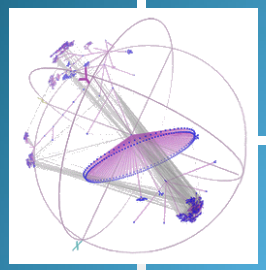
- Un des avantages : partage de code commun. Particulièrement important pour les systèmes à temps partagé où plusieurs utilisateurs partagent un programme (éditeur de texte, base de données, etc.)

- Code partagé
  - Une copie en lecture seule (code réentrant) est partagé entre plusieurs utilisateurs. Seul la position dans le code change selon les utilisateurs.
  - Le code partagé doit apparaître au même endroit dans l'espace logique de tous les processus.

Partage de segments



P.ex: DLL utilisé par plus usagers

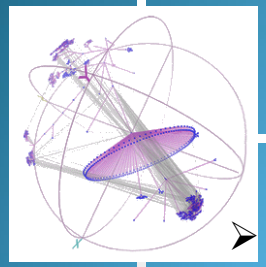


Segmentation et protection

- Chaque entrée dans la table des segments peut contenir des infos de protection:
  - longueur du segment
  - privilèges de l'usager sur le segment: lecture, écriture, exécution
    - Si au moment du calcul de l'adresse on trouve que l'usager n'a pas droit d'accès→interruption
    - ces infos peuvent donc varier d'un usager à autre, par rapport au même segment!

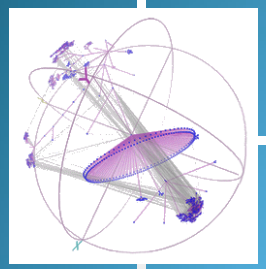
limite	base	read, write, execute?

Évaluation



- Avantages: l'unité d'allocation de mémoire (segment) est :
  - plus petite que le programme entier
  - une entité logique connue par le programmeur
  - les segments peuvent changer de place en mémoire
  - la protection et le partage de segments sont aisés (en principe)
- Désavantage: le problème des partitions dynamiques:
  - La fragmentation externe n'est pas éliminée: trous en mémoire, compression?

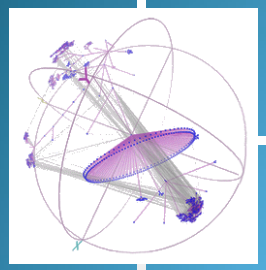
Une autre solution serait d'essayer de simplifier le mécanisme en utilisant des unités d'allocation mémoire de tailles égales : PAGINATION



Pagination

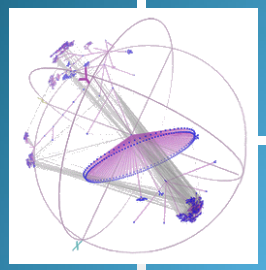
Le problème avec la segmentation est que l'unité d'allocation de mémoire (le segment) est de longueur variable

- La pagination utilise
  - des unités d'allocation de mémoire fixe, éliminant donc ce problème
  - l'allocation non contiguë de blocs de mémoire



Pagination

- Schéma de pagination :
  - La mémoire physique est divisée en blocs de taille fixe (des frames) dont la taille est généralement une puissance de 2 (entre 512 et 8192 octets).
  - La mémoire logique est divisée en blocs de la même taille (des pages)
- Conséquences:
  - Les pages logiques d'un processus peuvent donc être assignés aux cadres disponibles n'importe où en mémoire principale
  - un processus peut être éparpillé n'importe où dans la mémoire physique.
  - la fragmentation externe est éliminée



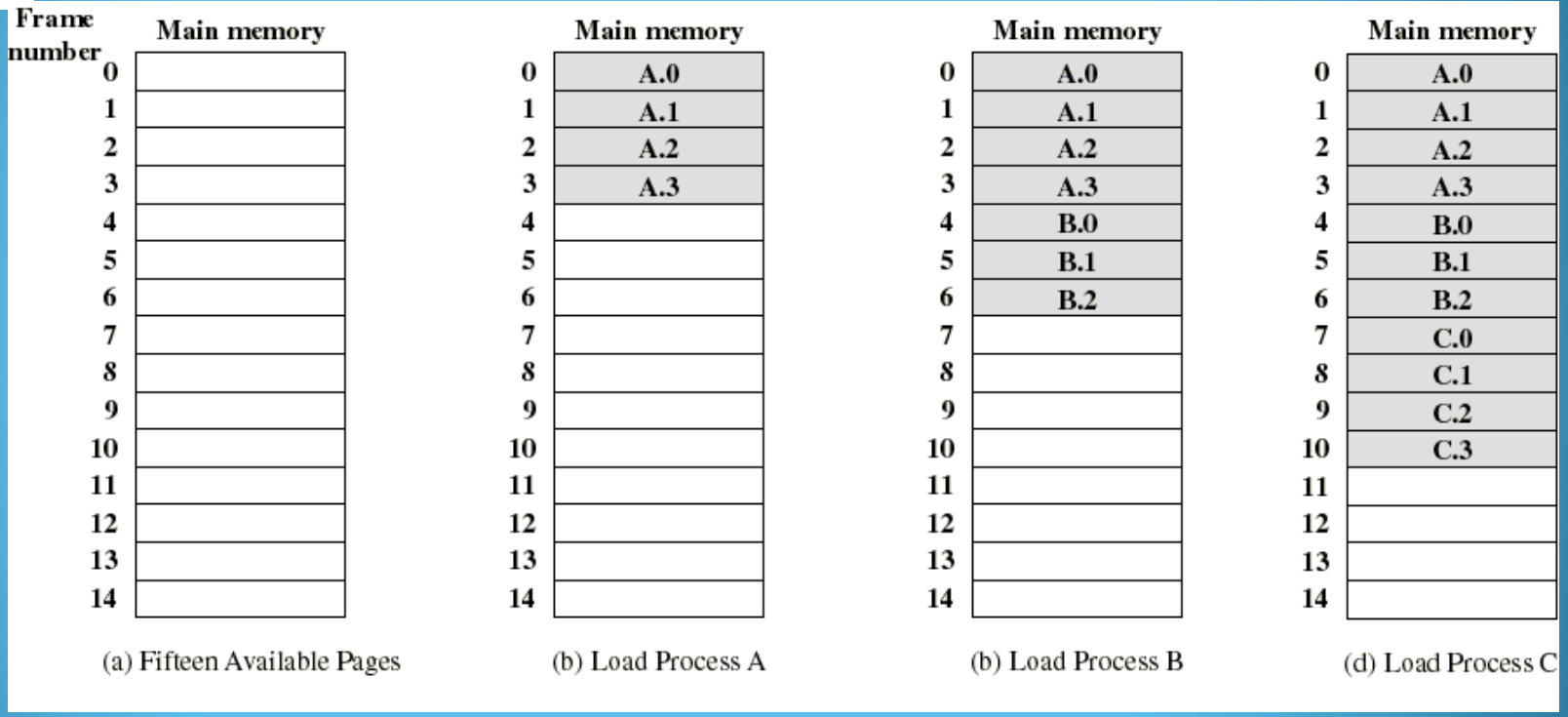
Pagination

Le SE doit en permanence :

- Savoir quelles sont les frames libres.
- Trouver n frames libres si une programme en a besoin.
- Pouvoir gérer les correspondance entre adresses physiques et logiques.

- Risque de fragmentation interne à cause des pages.

Exemple



- **Supposons que le processus B se termine ou est suspendu**

Exemple

- Nous pouvons maintenant transférer en mémoire un processus D, qui demande 5 cadres, bien qu'il n'y ait pas 5 cadres contigus disponibles
- La fragmentation externe est limitée en cas où le nombre de cadres disponibles n'est pas suffisant pour exécuter un programme en attente
- Seule la dernière page d'un processus peut souffrir de fragmentation interne

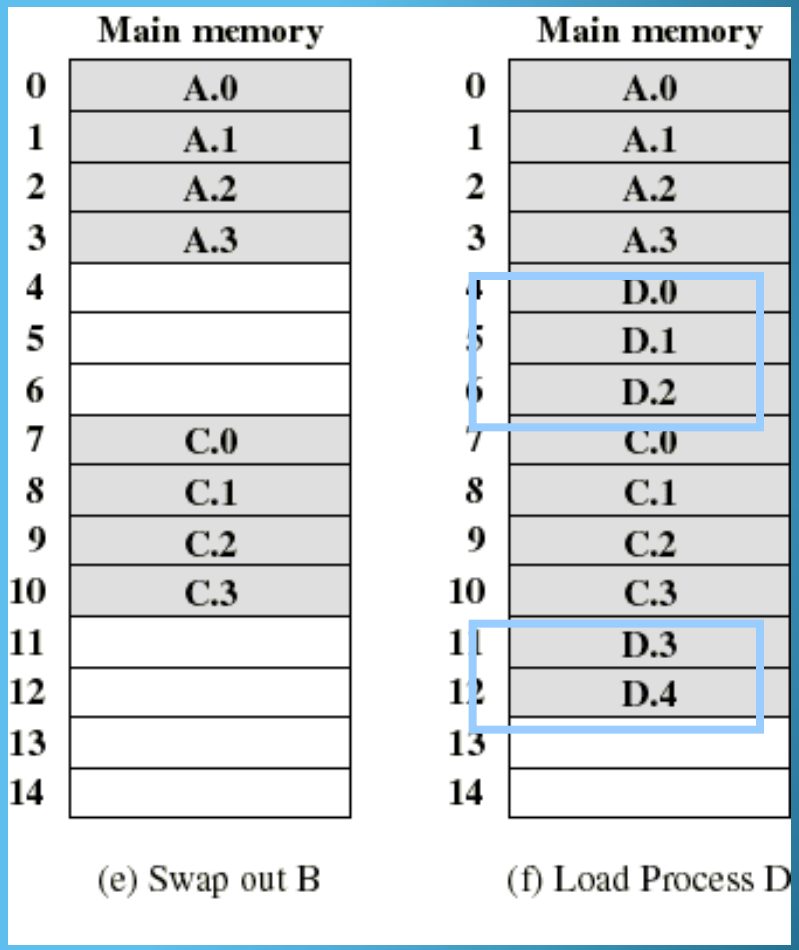
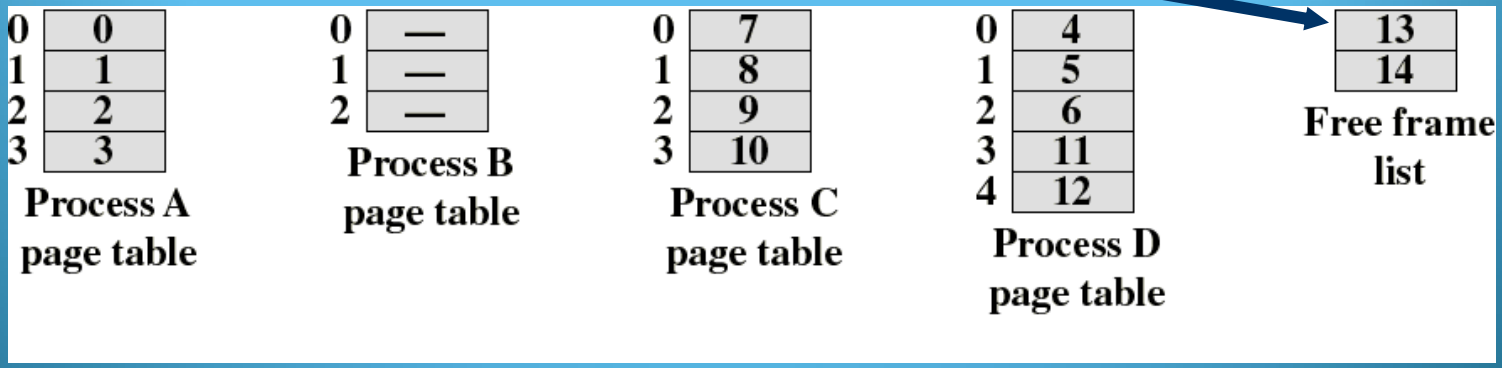
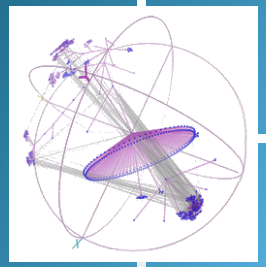


Table des pages

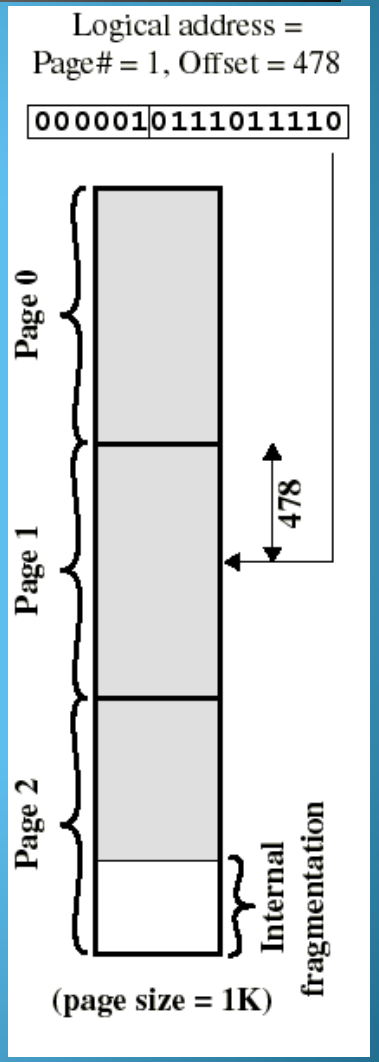
- Le SE doit maintenir une table de pages pour chaque processus
- Chaque entrée de la Table de pages est composée de plusieurs champs, notamment :
  1. Le bit de présence.
  2. Le bit de référence (R).
  3. Les bits de protection.
  4. Le bit de modification (M).
  5. Le numéro de cadre contenant la page.
- Une liste de cadres disponibles est également maintenue (free frame list)



Adresse logique



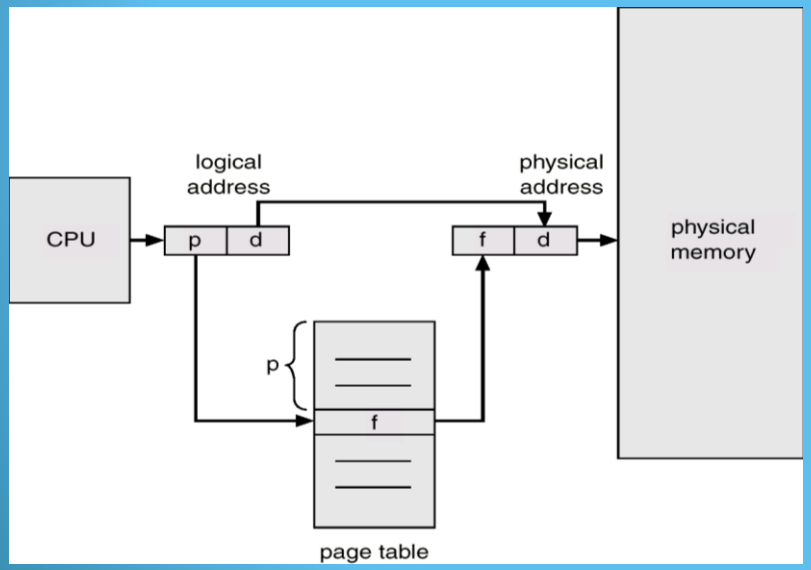
- L'adresse logique (n,d) est traduite en adresse physique (k,d) en utilisant n comme index sur la table des pages et en le remplaçant par l'adresse k trouvée
- d ne change pas
- Donc les pages sont invisibles au programmeur,
- La traduction des adresses au moment de l'exécution est facilement réalisable par le matériel: MMU
- Un programme peut être exécuté sur différents matériels employant des dimensions de pages différentes

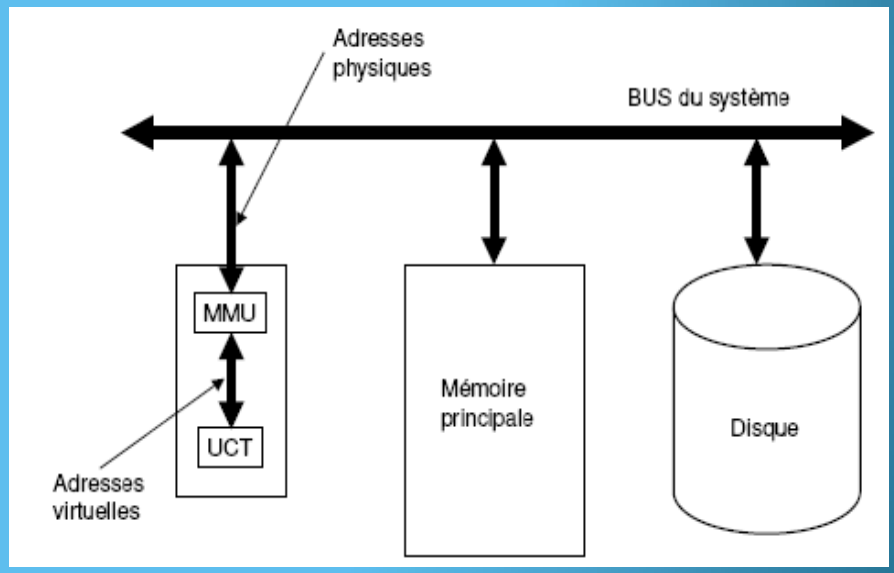


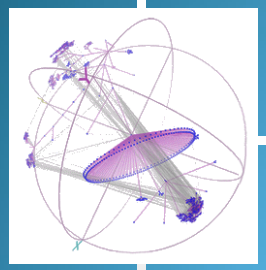
Traduction d'adresses par le MMU

- Traduction d'adresses:

  Tant dans le cas de la segmentation, que dans le cas de la pagination, nous ajoutons donc le décalage à l'adresse du segment ou page.





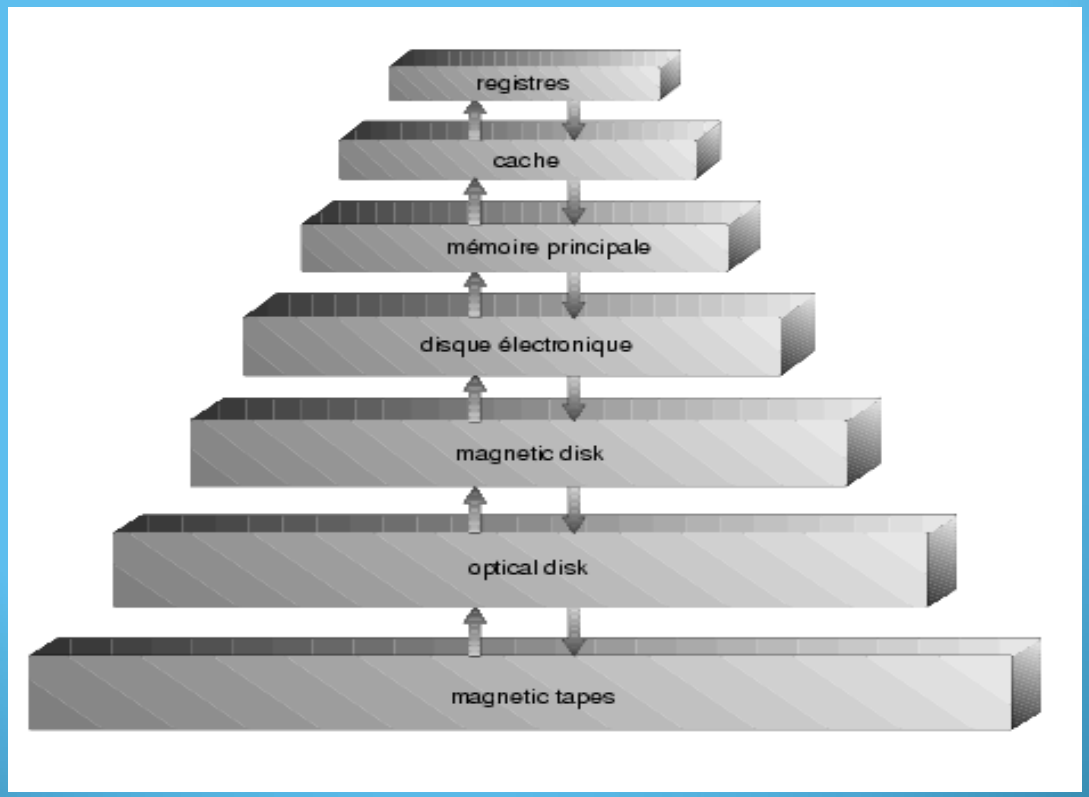


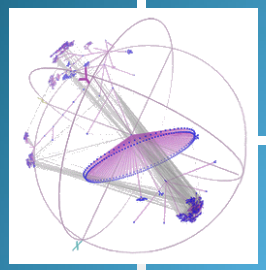
Pagination et segmentation

- Techniques combinées utilisées dans de nombreux systèmes modernes :
  - Espace logique divisé en plusieurs segments.
  - Chaque segment est divisé en pages de taille fixe.
  - Si un segment est plus petit qu'une page, il occupe une seule page.
  - Table des pages pour chaque segment.
  - Décalage de segment = numéro de page + décalage de page.
  - Le numéro de frame est combiné avec le décalage de page pour obtenir l'adresse physique.
- Plus de fragmentation externe, mais fragmentation interne et table plus volumineuses.

La mémoire virtuelle

- La mémoire virtuelle est une application du concept de hiérarchie de mémoire

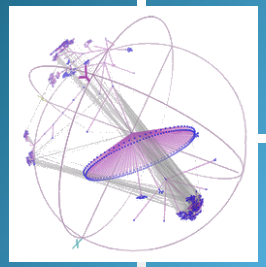




La mémoire virtuelle

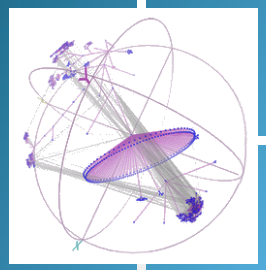
- Séparation de la mémoire logique et de la mémoire physique.
  - Seuls de petites parties des programmes ont besoin d'être en mémoire pour l'exécution.
  - L'adressage logique peut donc être plus étendu que l'espace physique réel.
  - Les pages peuvent donc être ou pas en mémoire physique (mécanismes de swap).

- La mémoire virtuelle peut être implémentée via :
  - Pagination
  - Segmentation



De la pagination et segmentation à la mémoire virtuelle

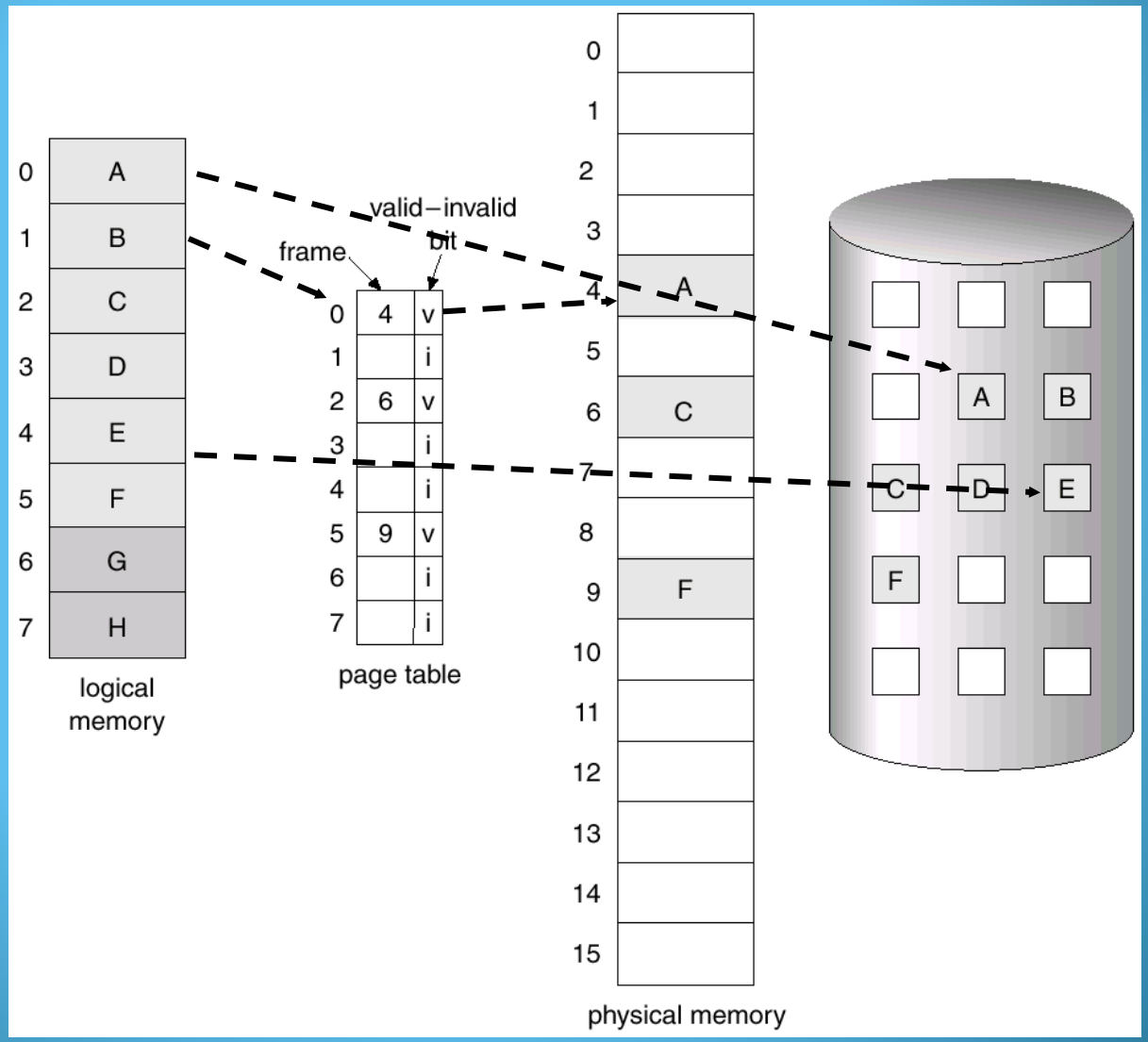
- Références à la mémoire sont traduites en adresses physiques au moment d'exécution
  - ✓ Un processus peut être déplacé à différentes régions de la mémoire, aussi mémoire secondaire!
  - ✓ Donc: tous les morceaux d'un processus ne nécessitent pas d'être en mémoire principale durant l'exécution
  - ✓ L'exécution peut continuer à condition que la prochaine instruction (ou donnée) soit dans une page se trouvant en mémoire principale
- La somme des mémoires logiques des processus en exécution peut donc excéder la mémoire physique disponible
- Le concept de base de la mémoire virtuelle
  - ✓ Une image de tout l'espace d'adressage du processus est gardée en mémoire secondaire (normal. disque) d'où les pages manquantes pourront être prises au besoin
- Mécanisme de va-et-vient ou swapping

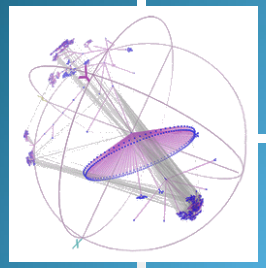


Pages en RAM ou sur disque

Page A en RAM **et** sur disque

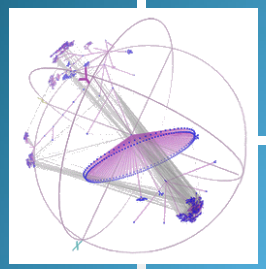
Page E **seulement** sur disque





Pagination

- Ne mettre une page en mémoire que si besoin:
  - Moins d'E/S nécessaires
  - Besoin plus faible en mémoire
  - Plus d'utilisateurs possibles
- Il faut des références sur les pages :
  - Référence invalide ⇒ erreur
  - Pas en mémoire ⇒ charger la page
  - En mémoire ⇒ ok

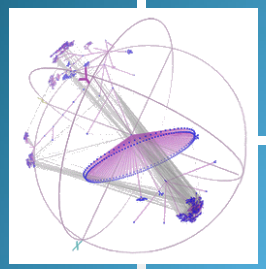


Bit de validité (présence)

- À chaque page est associé un bit :
  1 $\Rightarrow$ en mémoire, 0 $\Rightarrow$ pas en mémoire
- Initialement tous les bits sont mis à 0.

- Au moment de la conversion des adresses, si le bit est à 0 ; défaut de page(page fault).

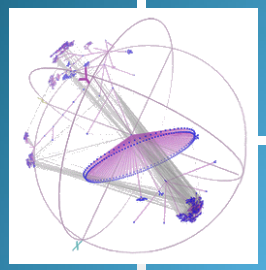
- Sauvegarde de l'état du processus.
- Le SE détermine s'il s'agit bien d'une faute de page.
- Le CPU est donné à un autre processus pendant la gestion de la faute.



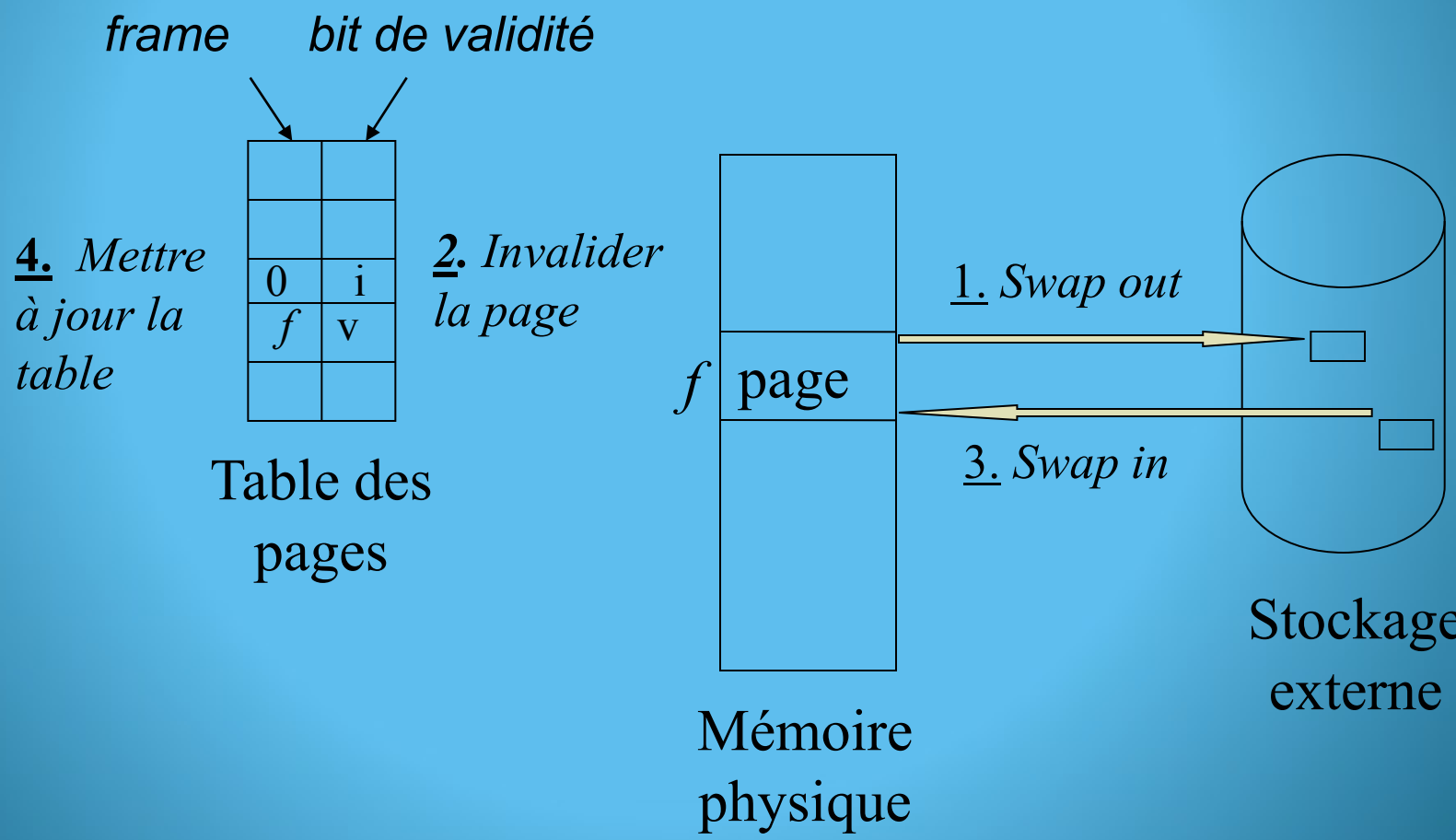
Plus de pages libres ?

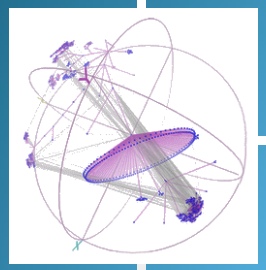
Deux problèmes principaux à gérer. Il faut développer des algorithmes pour :

1. Le remplacement de pages :
   - Trouver une page à sortir de la mémoire : page victime.
   - Évidemment, plusieurs cadres de mémoire ne peuvent pas être `victimisés`: p.ex. cadres contenant le noyau du SE, tampons d 'E/S...
   - La 'victime' doit-elle être récrite en mémoire secondaire?
     - Oui, si elle a été modifiée depuis qu'elle a été chargée en mémoire principale
     - sinon, sa copie sur disque est encore fidèle
   - Essayer d'éviter de faire trop de remplacements.
2. L'allocation des frames.



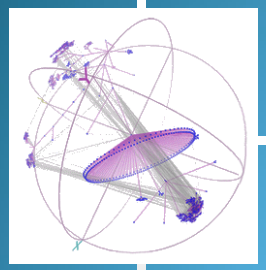
Remplacement de page





Algorithmes de remplacement

- Choisir la victime de façon à minimiser le taux de défaut de pages
- Plusieurs méthodes pour choisir qui sortir de la mémoire :
  - First-in-First-Out
  - Optimal
  - Least Recently Used
  - Deuxième chance : horloge (clock)
  - …
- On cherche un algorithme qui fait peu de défauts de pages.
- Évaluation de l'algorithme sur une suite de demandes.



Algorithme FIFO et LRU

FIFO

- Pour chaque page on connaît son heure d'arrivée en mémoire.
- Quand une page doit être remplacée, on choisit la plus vieille.
  - Problème :Les premières pages amenées en mémoire sont souvent utiles pendant toute l'exécution d'un processus!
  - variables globales, programme principal, etc.

Least-recently-used

La moins récemment utilisée

- Remplacer la page qui n'a pas été utilisée depuis le plus longtemps
- Associer à chaque page son dernier instant d'utilisation.

Comparaison de FIFO avec LRU

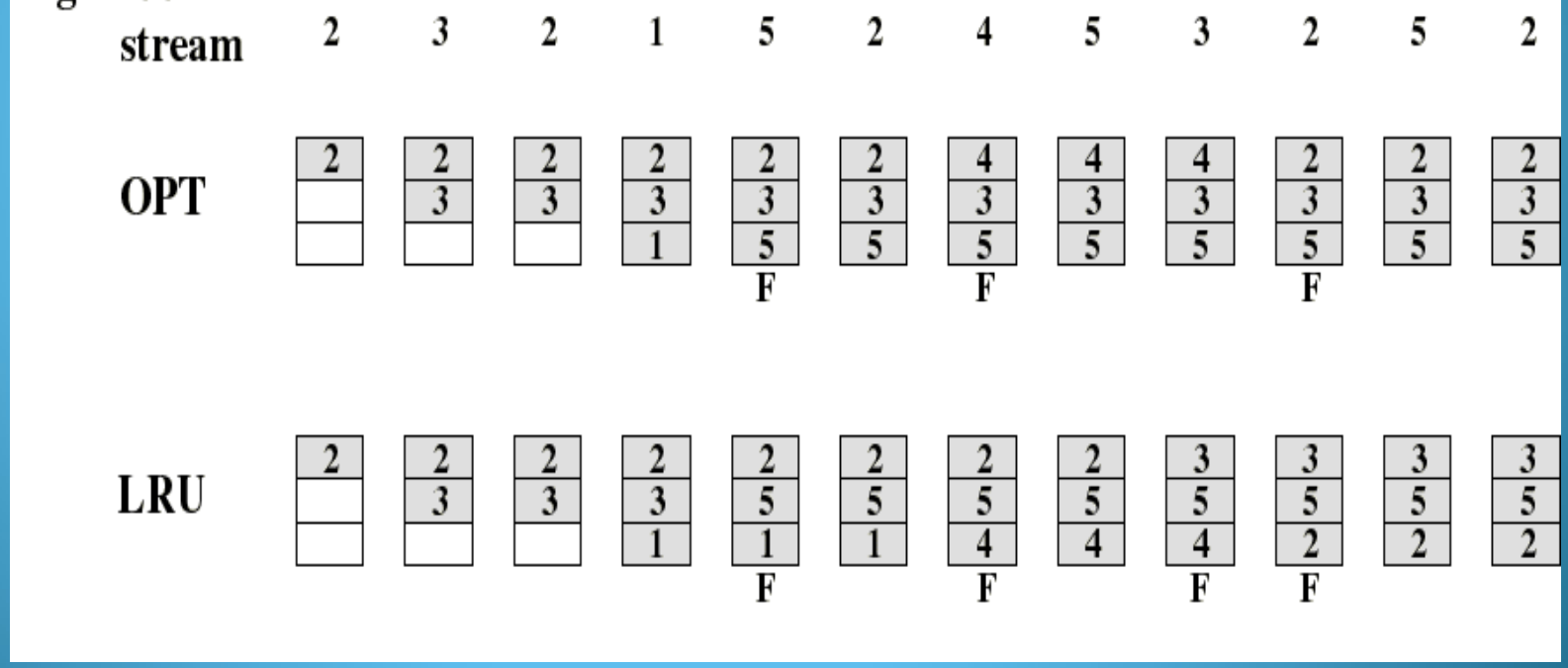
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F

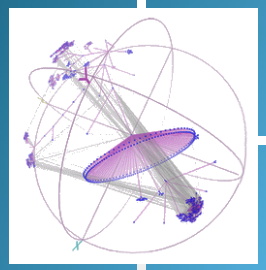
- Contrairement à FIFO, LRU reconnaît que les pages 2 and 5 sont utilisées récemment
- La performance de FIFO est moins bonne:

  LRU: 3+4=7, FIFO: 3+6=9

Comparaison OPT-LRU

OPT: 3+3=6, LRU 3+4=7 .

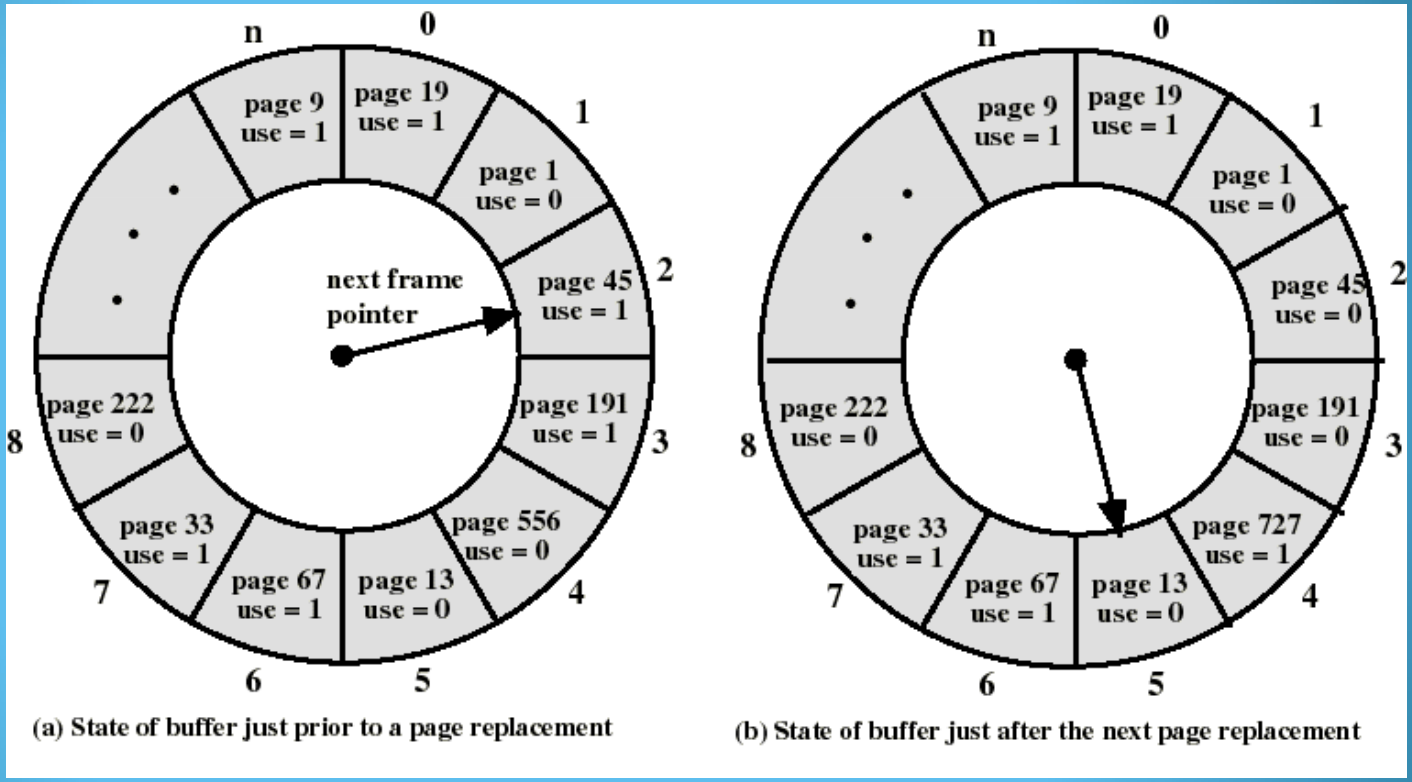




L'algorithme de l'horloge

- Semblable à FIFO, mais les cadres qui viennent d'être utilisés (bit=1) ne sont pas remplacées (deuxième chance)
- Les cadres forment conceptuellement un tampon circulaire
- Lorsqu'une page est chargée dans un cadre, un pointeur pointe sur le prochain cadre du tampon
- Pour chaque cadre du tampon, un bit "utilisé" est mis à 1 (par le matériel) lorsque:
  - ✓ une page y est nouvellement chargée
  - ✓ sa page est utilisée
- Le prochain cadre du tampon à être remplacé sera le premier rencontré qui a son bit "utilisé" = 0.
- Durant cette recherche, tout bit "utilisé" = 1 rencontré sera mis à 0

Algorithme de l'horloge Remplacement global

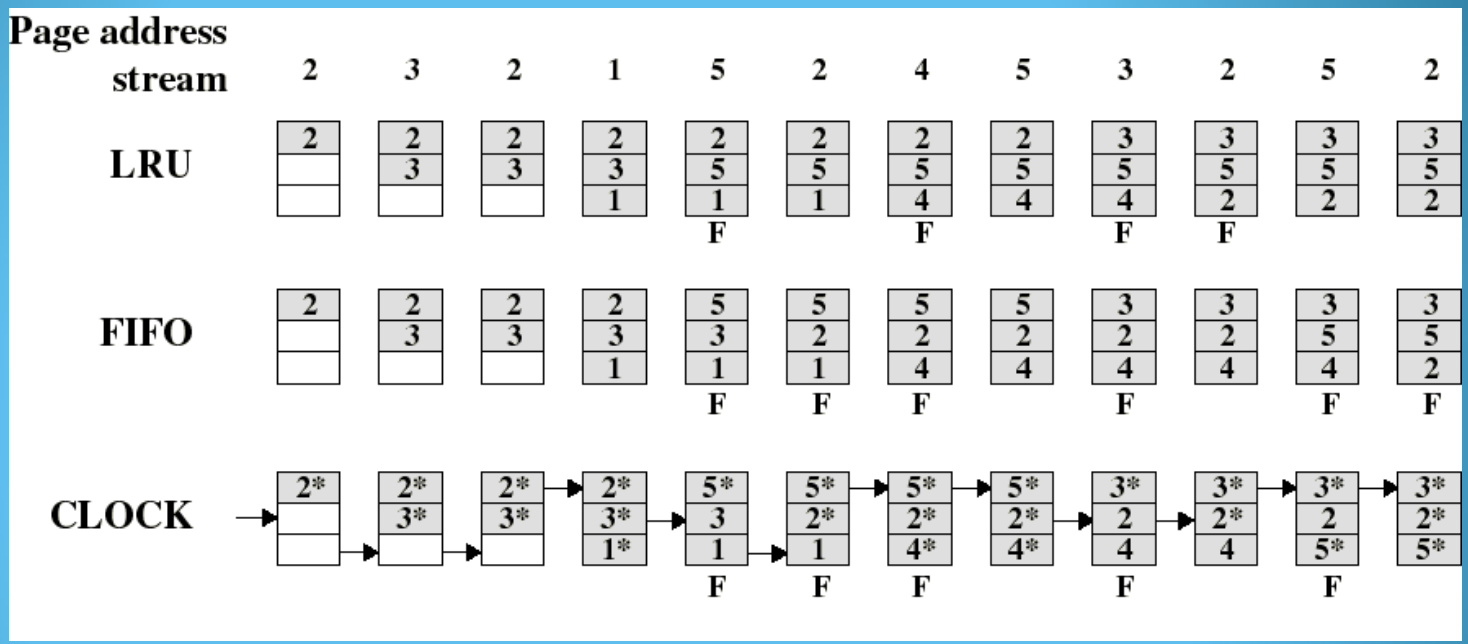


(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

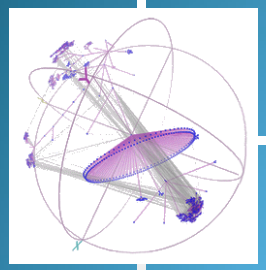
La page 727 est chargée dans le cadre 4.
La prochaine victime est 5, puis 8.

Algorithme de l'horloge Remplacement local



- Astérisque indique que le bit utilisé est 1
- L'horloge protège du remplacement les pages récemment utilisées en mettant à 1 le bit "utilisé" à chaque référence

  OPT: 3+3=6, LRU: 3+4=7, Horloge: 3+5=8 , FIFO: 3+6=9

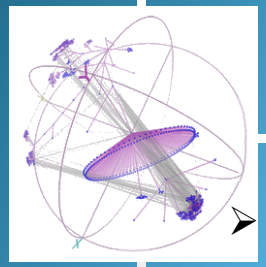


Anomalie de Belady

1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

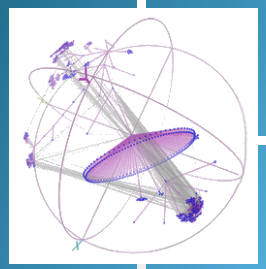
- 9 fautes de page.
- En général, plus il y a de frames, moins il y a de fautes.
  - *Anomalie de Belady*.
- 9 fautes de page.

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3



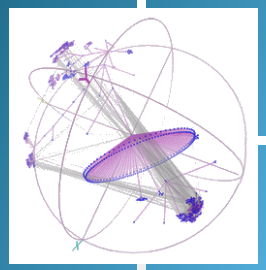
Comparaison des algorithmes

- Les simulations montrent que l'horloge est presque aussi performant que LRU
- variantes de l'horloge ont été implantées dans des systèmes réels
- Lorsque les pages candidates au remplacement sont locales au processus souffrant du défaut de page et que le nombre de cadres alloué est fixe, les expériences montrent que:
  - ✓ Si peu (6 à 8) de cadres sont alloués, le nombre de défaut de pages produit par FIFO est presque double de celui produit par LRU, et celui de CLOCK est entre les deux
  - ✓ Ce facteur s'approche de 1 lorsque plusieurs (plus de 12) cadres sont alloués.
- Cependant le cas réel est de milliers et millions de pages et cadres, donc la différence n'est pas trop importante en pratique...
- On peut tranquillement utiliser LRU



Algorithmes compteurs

- Garder un compteur pour les références à chaque page
- LFU: Least Frequently Used: remplacer la pages avec le plus petit compteur
- MFU: Most Frequently Used: remplacer les pages bien usées pour donner une chance aux nouvelles
- Ces algorithmes sont d'implantation couteuse et ne sont pas très utilisés

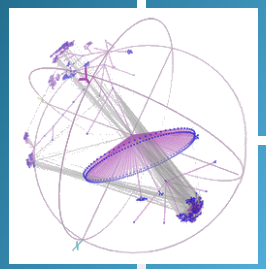


Thrashing

- S'il y a trop de défauts de page, un processus peut passer plus de temps en attente de pagination qu'en exécution : trashing.
  - Performance très amoindrie du système.

Exemple :

- Utilisation faible du CPU : de nouveaux processus sont admis. Si un processus a besoin de plus de frames, il va faire des fautes et prendre des frames aux autres processus. Ceux-ci vont à leur tour prendre des frames aux autres, etc.
- Ces processus doivent attendre que les pages soient chargés et vont donc être en attente, donc l'utilisation du CPU diminue.



Thrashing (2)

Comment gérer le trashing ?

- Algorithme de remplacement local : un processus qui fait du trashing ne peut pas prendre de frames aux autres processus.
- Fournir à un processus autant de frames qu'il en a besoin. Utiliser par exemple le *Working Set Model.*
  - Working set : ensemble de travail = ensemble des pages qu'un processus utilise.
  - Si l'ensemble de travail est entièrement en mémoire, le processus ne fera pas de faute.
  - Le système de pagination doit s'assurer qu'il l'ensemble de travail est en mémoire avant que le processus ne puisse avoir accès au processeur.