Architecture et Système 2

Partie 4 : La gestion mémoire

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

(1ère version par P.Saliou)

Partie 3 : La mémoire

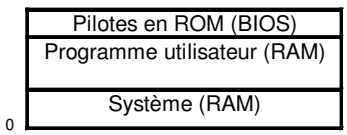
1. Introduction
2. Les concepts fondamentaux
3. L'allocation
4. Le swap (ou va-et-vient)
5. La pagination
6. La mémoire virtuelle
7. La segmentation

1. Introduction

- La **mémoire principale** est le lieu où se trouvent les programmes et les données quand le processeur les exécute.
- On l'oppose au concept de **mémoire secondaire**, représentée par les disques, de plus grande capacité, où les programmes peuvent séjourner avant d'être exécutés.
- La nécessité de gérer la mémoire de manière **optimale est toujours d'actualité**.
  - Malgré des capacités de plus en plus grandes et des coûts moindres.
  - Elle n'est en général, jamais suffisante, ceci en raison de la **taille continuellement grandissante des programmes**.

1.1 La multiprogrammation (1)

- Le concept de multiprogrammation (multi tâche) s'oppose à celui de monoprogrammation (mono tâche) qui ne permet qu'à un seul processus utilisateur d'être exécuté.
- On trouve alors en mémoire, par exemple dans le cas de MS-DOS :
  - le système en mémoire basse,
  - les pilotes de périphériques en mémoire haute
  - et un programme utilisateur entre les deux.



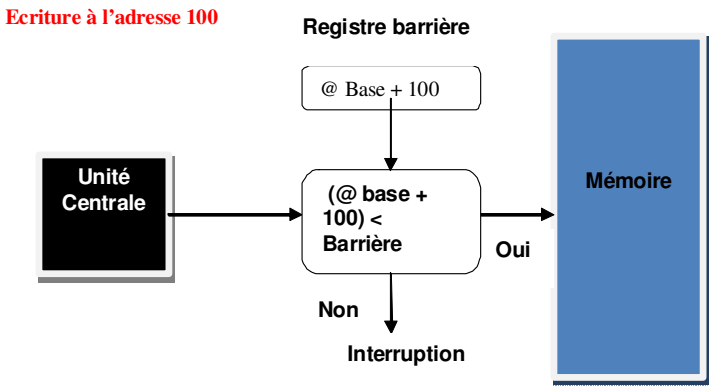
1.1 La multiprogrammation (2)

- La multiprogrammation autorise l'exécution de plusieurs processus indépendant à la fois.
- Cette technique permet d'optimiser le taux d'utilisation du processeur en réduisant notamment les attentes sur des entrées/sorties.
- La multiprogrammation implique le **séjour de plusieurs programmes/processus en même temps en mémoire** et c'est cette technique qui a donné naissance à la gestion moderne de la mémoire.

1.2 Les registres matériels (1)

- Pour assurer une protection, la plupart des processeurs disposent de deux registres délimitant le domaine d'un processus:
  - **Le registre de base** contient l'adresse de base du processus en mémoire.
    - L'utilisateur/système continue à utiliser des adresses logiques qui commencent à 0000.
    - A chaque utilisation d'une adresse logique du programme, on ajoute à cette adresse la valeur du registre de base pour trouver l'adresse physique.
  - **Le registre barrière** (limite) contient une adresse limite.
    - Cette adresse limite est comparée à toute adresse d'instruction ou de données manipulées par un programme utilisateur.

1.2 Les registres matériels (2)



2. Concepts fondamentaux

2.1 La production d'un programme/processus
2.2 Les principes de gestion

2.1 Production d'un programme/processus

- Plusieurs étapes avant l'exécution d 'un programme/processus
  - Au début, le programmeur écrit son programme dans un langage.
  - Un compilateur transforme ce programme en **un module objet**.
  - Les appels à des procédures externes sont laissés comme des points de branchements.
  - L'éditeur de liens fait correspondre ces points à des fonctions contenues dans les bibliothèques et produit, dans le cas d'une liaison *statique*, une image binaire.
  - Certains systèmes, notamment Unix, autorisent des liaisons *dynamiques* et reportent la phase d'édition jusqu'au chargement.
- C'est ensuite le « **chargeur** » qui effectue le chargement de l'exécutable en mémoire.
  - C'est également lui qui effectue les liaisons des appels système avec le noyau, tel que l'appel *write* par exemple.

2.2 Principes de gestion

- Pour effectuer le chargement, le système :
  - **alloue** un espace de mémoire libre,
  - il y **place** l'espace d'adressage du processus,
  - Il **libère** cet emplacement une fois le programme terminé.
- Dans beaucoup de cas :
  - il n'est pas possible de faire tenir tous les programmes ensemble en mémoire,
  - parfois même, la taille d'un seul programme est trop grande.
- Les systèmes d'exploitation mettent en œuvre des stratégies de chargement qui libèrent le programmeur :
  - ***le swap*** *(ou va-et-vient),*
  - ***et la mémoire virtuelle***.

3. L'allocation

- Avant d'implanter une technique de gestion de la mémoire centrale par ***va-et-vient (ou swap),*** il est nécessaire :

  1. de **connaître son état** : les zones libres et occupées,
  2. de disposer d'une **stratégie d'allocation**,
  3. et enfin de **procédures de libération**.

- Les techniques présentées dans les transparents suivants servent de base au va-et-vient.

3.1 État de la mémoire

- La mémoire est découpée en **unités ou blocs d'allocation**.
- Le système garde la trace des blocs de mémoire occupés par l'intermédiaire:
  - d'une **table de bits** (bitmap).
  - ou bien d'une **liste chaînée**.

3.1.1 Tables de bits

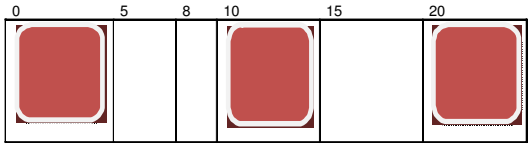
- On peut conserver l'état des blocs de mémoire grâce à une ***table de bits***.
  - Les unités d'allocation libres étant notées par 0
  - et ceux occupées par un 1 (ou l'inverse).

0	0	1	1	0	0	

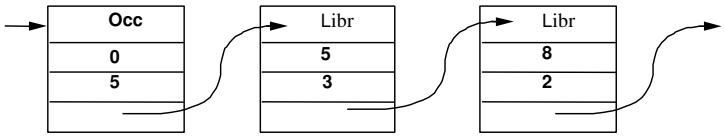
- La technique des tables de bits est **simple** à implanter, mais elle est peu utilisée.
  - plus **l'unité d'allocation** est petite, moins on a de pertes lors des allocations,
  - mais en revanche, plus cette table occupe de place en mémoire.

3.1.2 Listes chaînées

- On peut représenter la mémoire par une ***liste chaînée*** de structures:
  - le type (Libre ou Occupé),
  - l'adresse de début, la longueur,
  - et un pointeur sur l'élément suivant.
- Pour une mémoire ayant l'état suivant :



- On aurait la liste :



3.2 Politiques d'allocation

- L'allocation d'un espace libre pour un processus peut se faire suivant trois stratégies principales :
  - Le "**premier ajustement**" (first fit) prend le premier bloc libre de la liste qui peut contenir le processus qu'on désire charger.
  - Le "**meilleur ajustement**" (best fit) tente d'allouer au processus l'espace mémoire le plus petit qui puisse le contenir.
  - Le "**pire ajustement**" (worst fit) prend le plus grand bloc disponible et le fragmente en deux. Dans cette stratégie, il s'agit de conserver le plus grand morceau de mémoire libre.
- Certaines simulations ont montré que :
  - le " premier ajustement " était meilleur que les autres ,
  - paradoxalement, le " meilleur ajustement ", qui est plus coûteux, n'est pas optimal car il produit de la **fragmentation**.

3.3 Libération

- La libération se produit quand un processus est évacué de la mémoire.
  - On marque alors le bloc à libérer
  - Et on le fusionne éventuellement avec des blocs adjacents.
- Supposons que X soit le bloc qui se libère, on a les schémas de fusion suivants.

Avant			Après		
A	X	B	A	L	B
A	X	L	A	L	
L	X	B	L		B
L	X	L	L		

3.4 La récupération de mémoire

- Les deux aspects contre lesquels il faut se prémunir dans la gestion de la mémoire sont :
  - La **fragmentation** (**interne** et **externe**) de la mémoire qui peut être diminuée à l'aide d'un mécanisme de compactage (défragmentation).
  - Les **fuites de mémoire** qui peuvent être récupérées sur certains systèmes à l'aide d'un mécanisme de récupération automatique appelé **ramasse-miettes** (garbage collector) → ex: malloc() sans free().

3.4.1 Le compactage de la mémoire

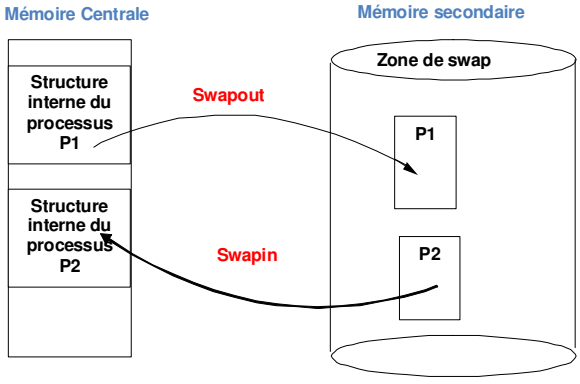
- La fragmentation de la mémoire est particulièrement dommageable.
  - Elle peut saturer l'espace disponible rapidement.
- Pour la diminuer, on peut la compacter régulièrement.
  - C'est-à-dire déplacer les processus en mémoire de façon à rendre contiguës les zones de mémoire libres.
  - L'objectif est d'aboutir à des zones de mémoire libres de taille suffisante pour être utilisées.
  - Cette opération est coûteuse et nécessite parfois des circuits spéciaux.

3.4.2 Le ramasse-miettes (garbage collector)

- Une fois qu'un objet ou une zone a été utilisé, le programmeur système doit récupérer la mémoire " à la main " par une libération du pointeur sur cette zone (free()).
  - Ceci est une source d'erreurs car on oublie parfois cette opération ce qui provoque alors une " fuite de mémoire " (memory leak).
- Certains langages ou systèmes incorporent la récupération automatique de mémoire **(garbage collector)** qui libère ainsi le programmeur de cette tâche.
  - C'est le cas de Java. Il est alors très facile d'éliminer immédiatement une zone par l'affectation objet = null.
  - La récupération automatique de mémoire peut poser des problèmes car périodiquement le système s'arrête brutalement pour laisser la place au **ramasse-miettes**.

4. Le swap (ou va-et-vient) (1)

- Le swap est mis en œuvre lorsque tous les processus ne peuvent pas tenir simultanément en mémoire.
- On doit alors en déplacer temporairement certains sur disque dans une zone de swap.



4. Le swap (ou va-et-vient) (2)

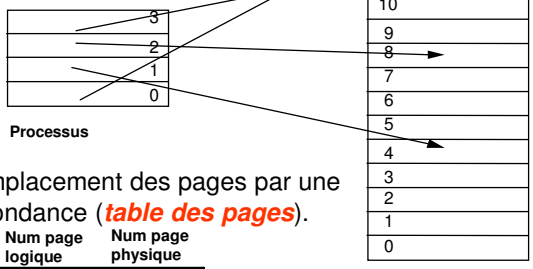
- Sur le disque, la zone de swap d'un processus peut être allouée :
  - **A la demande** dans la zone de swap générale.
  - **Au début de l'exécution**; lors du déchargement, le processus est alors sûr d'avoir une zone d'attente libre sur le disque.
- Le système de va-et-vient
  - s'il **permet de pallier le manque de mémoire nécessaire à plusieurs utilisateurs/processus**,
  - ***Il n'autorise cependant pas l'exécution de programmes de taille supérieure à celle de la mémoire centrale***.

5. La pagination (1)

- Pour accélérer les mécanismes d'allocation de mémoire, la notion de **page** a été introduite :
  - la mémoire physique est découpée en pages de taille fixe (quelques kilo-octets, généralement 4ko),
  - l'espace d'adressage d'un processus est également découpé en pages.
- Grâce à ce système :
  - Il n'est plus nécessaire de placer le processus dans une zone contiguë de mémoire.
  - Il devient possible d'allouer de la mémoire à un processus sans forcément avoir à réaliser de compactage.

5. La pagination (2)

- Les pages du processus sont chargées à des pages libres de la mémoire.

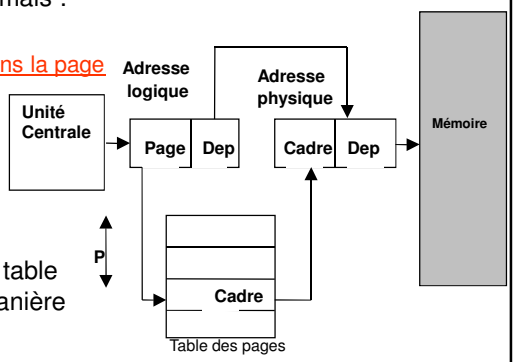


- On conserve l'emplacement des pages par une table de correspondance (***table des pages***).

Num page logique	Num page physique
0	14
1	4
2	8
3	12

5. La pagination (3)

- La pagination nécessite de nouvelles possibilités matérielles car toute adresse comprend désormais :
  - Un numéro de page.
  - Une position (déplacement) dans la page



- Le calcul d'une adresse avec la table des pages peut se faire de la manière suivante :
  - A : adresse logique
  - P : taille de page
  - Numéro de page = A div P
  - Position = A modulo P

5. La pagination (4)

- La taille du processus n'est plus le critère prépondérant de l'ordonnancement qui devient plus indépendant de la mémoire.
- Un avantage de la pagination est une plus grande simplicité du partage de la mémoire entre différents processus.
  - Quand plusieurs processus partagent le même code, celui-ci sera contenu dans des pages partagées qui de plus pourront être protégées en écriture.
- Les protections d'accès des pages sont faites au niveau de la table des pages qui est globale au système.
- Attention à la **fragmentation interne** car on alloue une page entière alors que le processus ne l'utilise qu'en partie.

6. La mémoire virtuelle

6.1. Présentation
6.2. Transcodage des adresses logiques
6.3. Algorithmes de remplacement de pages
6.4. Conception des systèmes paginés

6.1 Présentation (1)

- Le **swap** et la **pagination** sont fondés sur l'idée que l'ensemble de l'espace logique adressable d'un processus doit être en mémoire pour pouvoir l'exécuter.
  - Un coût de swap important.
  - L'impossibilité de créer de très gros processus.
- Si on regarde de près l'exécution des programmes
  - Il y a des portions de code qui gèrent des cas très inhabituels.
  - Les tableaux, les listes et autres tables sont en général initialisés à des tailles plus grandes que ce qui est réellement utile.
  - Certaines fonctions d'applications sont très rarement utilisées.

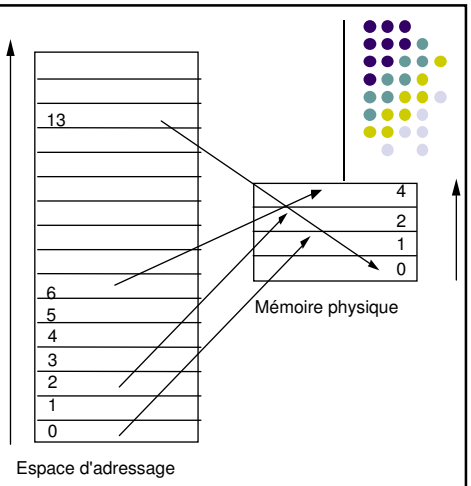
…et donc tous les morceaux d'un programme ne sont pas utilisés en même temps.

6.1 Présentation (2)

- D'où l'idée de la **mémoire virtuelle** **qui permet d'exécuter des programmes qui ne tiennent pas entièrement en mémoire centrale**.
  - Pour ceci, on découpe (on " pagine ") les processus ainsi que la mémoire réelle en pages de quelques kilo-octets (1, 2 ou 4 ko généralement).
  - L'encombrement total des processus constitue l'espace d'adressage ou **la mémoire virtuelle**. Cette mémoire virtuelle réside en mémoire ou/et sur le disque.
  - A la différence de la pagination, ***on ne charge qu'un sous-ensemble de pages en mémoire***. Ce sous ensemble est appelé l'espace physique (réel).

6.1 Présentation (3)

- La mémoire virtuelle permet d'exécuter des programmes dont la taille excède la taille de la mémoire réelle car on dispose *d'une* ***mémoire linéaire/virtuelle beaucoup plus grande que la mémoire physique***.



- Autres avantages :
  - Les utilisateurs consommant individuellement moins de mémoire, plus d'utilisateurs peuvent travailler en même temps.
  - Moins d'entrée/sorties sont effectuées pour l'exécution d'un processus, ce qui fait que celui-ci s'exécute plus rapidement.

6.2 Transcodage des adresses logiques

6.2.1 Les adresses virtuelles (1)

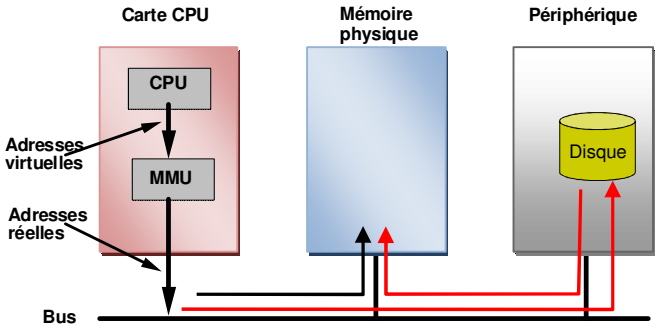
- Les adresses manipulées par les programmes sont appelées des adresses virtuelles et constituent l'espace d'adressage virtuel.
  - L'espace d'adressage virtuel est divisé en petites unités appelées pages.
  - Les unités correspondantes de la mémoire physique sont les cadres mémoire (ou cases mémoire ou *frame*).
  - Les pages et les cadres sont toujours de la **même taille**.
  - Les transferts entre la mémoire et le disque se font toujours par pages entières (au moins).

6.2.1 Les adresses virtuelles (2)

- Sur les architectures **sans mémoire virtuelle** :
  - ces adresses sont directement placées sur le bus de mémoire,
  - et provoque la lecture ou l'écriture du mot à l'adresses spécifiée.
- Lorsque la mémoire virtuelle est utilisée
  - les adresses virtuelles **ne sont pas** directement placées sur le bus de mémoire ,
  - elles sont envoyées à l'unité de gestion de la mémoire ou **MMU** (Memory Management Unit), composant qui **traduit les adresses virtuelles en adresses physiques**.

6.2.2 Le Memory Management Unit (MMU) (1)

- Lorsqu'une adresse est générée, elle est transcodée grâce à une table des pages et à des circuits matériels de gestion : *Memory Management Unit* (MMU).
  - Si cette adresse correspond à une adresse en mémoire physique, le MMU transmet sur le bus l'adresse réelle.
  - Sinon il se produit un **défaut de page**.

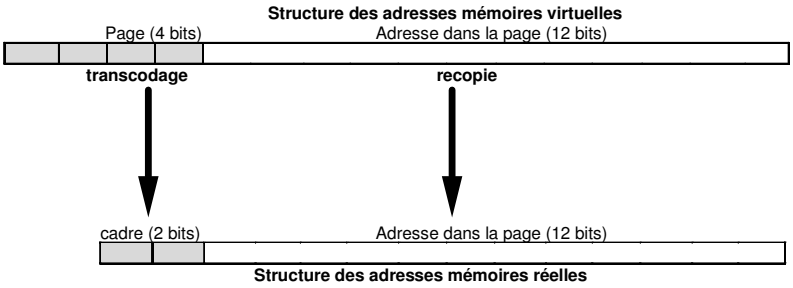


6.2.2 Le Memory Management Unit (MMU) (2)

- Lorsqu'un défaut de page se produit, il faut la charger la page virtuelle correspondante en mémoire réelle.
  - On choisit parmi les pages réelles une page "victime ".
  - Si cette dernière a été modifiée on la reporte en mémoire virtuelle (sur le disque).
  - Et on charge à sa place la page à laquelle on désirait accéder.

6.2.3 Algorithme de transcodage/traduction (1)

- Il est plus facile d'implanter un algorithme de transcodage avec une taille de page correspondant à une puissance de 2 :
  - Pour une adresse virtuelle ou réelle, on réserve **les bits de poids forts** nécessaires **pour coder les pages réelles et virtuelles**.
  - **Les bits de poids faibles codent les décalages** à l'intérieur de chacune de ces pages.



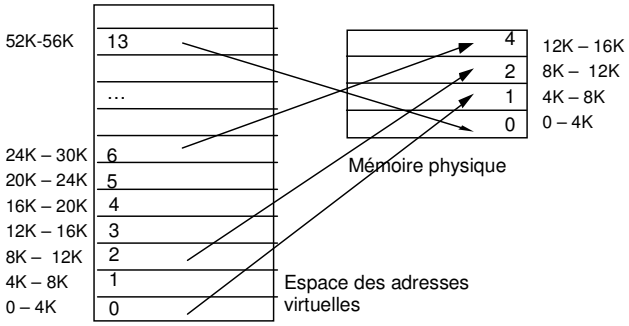
6.2.3 Algorithme de transcodage (2)

- Les informations de la table des pages servent au transcodage.
  - Un bit de présence précise si la page est en mémoire réelle.
  - Un bit de modification signale si on a écrit dans la page. Dans ce cas, la page devra être reportée sur le disque si on désire la remplacer par une autre.

P. Virt.	P. Réelle	Présence	Modif.
0	01	1	
1	–	0	
2	10	1	
3	11	0	
4	00	0	
5	01	0	
6	11	1	
13	00	1	
15	–		

6.2.4 Exemple de mappage (transcodage)

- Quand un programme essaie de lire l'adresse 0, par exemple au moyen de l'instruction « MOV REG, 0 ».
  - L'adresse virtuelle 0 est envoyé au MMU qui constate que cette adresse se situe à la page 0 (adresse de 0 à 4095) qui appartient à la case/cadre 1 (4096 à 8191).
  - Le MMU transforme l'adresse en 4096 et place cette valeur sur le bus.



6.2.4 La problématique de la table des pages (1)

- **Rappel** : le but de la **table des pages** est d'établir la correspondance entre les pages virtuelles et les cadres mémoires.
  - L'adresse virtuelle est composée d'un numéro de page virtuelle (bits de poids forts) et d'un déplacement (bits de poids faibles).
  - Le numéro de page virtuelle sert d'index dans la table des pages pour trouver l'entrée de la page virtuelle qui contient le numéro du cadre mémoire.
  - Le numéro du cadre mémoire remplace le numéro de la page virtuelle et constitue avec le déplacement l'adresse physique

6.2.4 La problématique de la table des pages (2)

- Deux points sont à prendre en considération.
  - **La table des pages peut être très grande.**
    - En utilisant des pages de 4 Ko, un espace d'adressage de 32 bits abouti à 1 million de pages.
    - La table des pages doit donc contenir 1 million d'entrées.
    - Chaque processus a besoin de sa propre table des pages.
  - **La mappage doit être très rapide.**
    - Le mappage de l'adresse virtuelle doit être effectué à chaque référence mémoire.
    - Une instruction typique nécessite une, deux ou davantage de références à la table des pages.
    - Si une instruction prend 10 ns, la recherche dans la table des pages doit être effectuée en quelques nanosecondes pour éviter un ralentissement trop important.

6.2.4 La problématique de la table des pages (3)

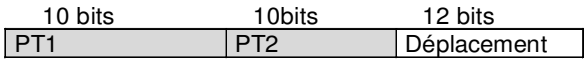
- Une **1ère approche** consiste à avoir une seule table des pages constituées d'un tableau de registres machine rapides (Translation Lookaside Buffer ou TLB parfois appelé mémoire associative), avec une entrée par page virtuelle, cette table étant indexée au moyen du numéro de page virtuelle.
  - Lorsqu'un processus est activé, le système d'exploitation charge les registres *avec la table des pages* de ce processus à partir d'une copie située en mémoire.
  - Au cours de l'exécution du processus, la table des pages ne nécessite plus de références mémoire pendant le mappage.
  - Cette méthode peut être coûteuse si la table des pages est grande et forcément pénalisante à chaque changement de contexte de processus.

6.2.4 La problématique de la table des pages (4)

- Une **2ème approche** consiste à stocker la table des pages entièrement en mémoire centrale.
  - Le matériel n'a alors besoin que d'un seul registre qui pointe sur le début de la table des pages en mémoire.
  - Cette conception permet de changer la "mappe" mémoire en modifiant un seul registre lors d'un changement de contexte.
  - Elle présente bien sûr le désavantage de nécessiter une ou plusieurs références mémoire pour lire les entrées de la table des pages lors de l'exécution de chaque instruction.

6.2.5 La table des pages à plusieurs niveaux (1)

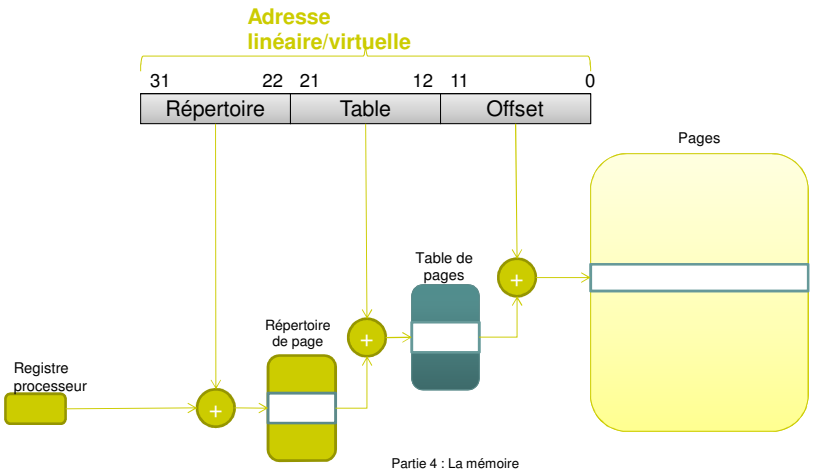
- Les tables de pages à plusieurs niveaux évitent de garder en permanence toutes les tables des pages en mémoire.
- Avec deux niveaux et une adresse virtuelle de 32 bits, on peut décomposer celle-ci de la manière suivante :
  - Un champ PT1 de 10 bits.
  - Un champ PT2 de 10 bits.
  - Un champ déplacement de 12 bits correspondant à des pages de 4K.



6.2.5 La table des pages à plusieurs niveaux (2)

- Le fonctionnement de la table à deux niveaux est le suivant :
  - Le MMU qui reçoit une adresse virtuelle extrait le champ PT1 et l'utilise en tant qu'index dans la table des pages (répertoires) de haut niveau.
  - Cette table de haut niveau contient les 1024 entrées correspondant au champ PT1 de 10 bits.
  - Chacune des 1024 entrées représente 4 Mo puisque l'espace d'adressage virtuel de 4 giga octets (32 bits) est divisée en 1024 parties.
  - L'entrée trouvée dans la table des pages de haut niveau donne l'adresse ou le numéro de case mémoire d'une table de second niveau.
  - Le champ PT2 est alors utilisée en tant qu'index dans la table de second niveau pour trouver le numéro de la case mémoire.

6.2.5 La table des pages à plusieurs niveaux (3)



6.3 Algorithmes de remplacement de pages

6.3.1 Présentation
6.3.2 Remplacement de page optimal
6.3.3 Premier entré - Premier sorti
6.3.4 La moins récemment utilisée (LRU)

6.3.1 Présentation des algorithmes de remplacement de pages (1)

- A la suite d'un défaut de page.
  - Il faut retirer une page de la mémoire pour libérer de la place pour la page manquante.
  - Si la page à retirer a été modifiée depuis son chargement en mémoire, il faut la sauvegarder sur le disque.
- Le choix de la page à retirer.
  - L'algorithme de remplacement de page optimal consiste à choisir la page qui sera appelée le plus tard possible.
  - La technique First In-First Out (FIFO) est assez facile à implanter.
  - L'algorithme de remplacement de la page la moins récemment utilisée (Least Recently Used) est l'un des plus efficaces.

6.3.2 L'algorithme de remplacement de page optimal

- Le meilleur algorithme de remplacement est facile à décrire.
  - Au moment du défaut de page, il existe en mémoire un certain nombre de pages. Une de ces pages risquent d'être référencée très vite, d'autres ne le seront que dans 10, 100 ou 1000 instructions.
  - On peut donc imaginer de numéroter chaque page avec le nombre d'instructions qui seront exécutées avant qu'elles ne soit référencées.
  - L'algorithme de remplacement optimal consiste alors à retirer la page qui porte le plus haut numéro.
- Malheureusement cet algorithme est irréalisable car le système d'exploitation ne peut pas connaître à l'avance le moment où les différentes pages seront référencées.

6.3.3 Premier entré – premier sorti

- **L'algorithme FIFO** (First In-First Out) a l'avantage de ne requérir que très peu de temps processeur.
  - Le système d'exploitation mémorise une liste de toutes les pages en mémoire, la première page de cette liste étant la plus ancienne et la dernière la plus récente.
  - Lorsqu'il se produit un défaut de page, on retire la première et on place la nouvelle à la fin de cette liste.
- Le problème de cet algorithme est qu'il peut retirer une page peu utilisée comme une page très utilisée. Il est donc très rarement utilisé.

6.3.4 La moins récemmment utilisé (LRU) (1)

- Dans le cas du **LRU**, c'est le **vieillissement d'une page** et non plus l'ordre de création de la page qui est utilisé. Il est basé sur l'hypothèse suivante.
  - Les pages qui ont été récemment utilisées le seront dans un proche avenir.
  - Alors que les pages qui n'ont pas été utilisées depuis longtemps ne sont plus utiles.
- L'algorithme LRU est un bon algorithme mais il pose de nombreux problèmes d'implémentation et peut nécessiter des dispositifs matériels particuliers pour le mettre en œuvre (compteur, pile, masque)

6.3.4 La moins récemmment utilisé (LRU) (2)

- Utilisation d'une **pile**
  - A chaque fois que l'on accède à une page, la page est placée en sommet de pile.
- Utilisation de **masques**
  - On utilise un octet à chaque page. Le système positionne à 1 le bit de poids fort à chaque accès à la page.
  - Toutes les N millisecondes le système fait un décalage à droite de l'octet associé à chaque page.
  - L'octet 00000000 = page non utilisée depuis 8 cycles.
  - Si on interprète ces octets comme des entiers non-signés, c'est la page ayant le plus petit octet qui a été utilisée le moins récemment.

6.4 Conception des systèmes paginés

6.4.1 Le modèle de l'ensemble de travail
6.4.2 Allocation locale et allocation globale
6.4.3 L'écroulement (Thrashing)
6.4.4 Le retour sur instructions
6.4.5 La taille des pages
6.4.6 Le verrouillage des pages en mémoire

6.4.1 Le modèle de l'ensemble de travail (1)

- Dans un système paginé, les processus sont lancés sans qu'aucune de leurs pages ne soit en mémoire :
  - Dès que le processeur essaie d'exécuter la première instruction, il se produit un défaut de page, ce qui amène le système d'exploitation à charger la page correspondante.
  - D'autres défauts de page apparaissent rapidement pour les variables globales ou la pile.
  - C'est seulement au bout d'un certain temps que le processus dispose de la plupart de ses pages et que l'exécution peut se poursuivre avec relativement peu de défauts de page.
- Cette stratégie est appelée **pagination à la demande** car les pages sont chargées à la demande et non à l'avance.

6.4.1 Le modèle de l'ensemble de travail (2)

- Généralement, les processus font des références groupées, c'est-à-dire que pendant une certaine phase de leur exécution ils ne référencent qu'un nombre restreint de leurs pages.
  - Cet ensemble de pages utilisées pendant un certain temps par un processus constitue **l'ensemble de travail du processus**.
  - Si l'ensemble de travail est entièrement en mémoire, le processus s'exécute sans provoquer de défauts de page.
- De nombreux systèmes de pagination mémorisent l'ensemble de travail de chaque processus et le chargent en mémoire avant de lancer le processus.
  - Cette approche est appelée " Modèle de l'ensemble de travail ".
  - Le chargement des pages avant exécution est appelé La **Préchargement** ou **Prépagination** .

6.4.1 Le modèle de l'ensemble de travail (3)

- Pour mettre en œuvre le modèle de l'ensemble du travail, il faut que le système d'exploitation mémorise les pages qui appartiennent à l'ensemble de travail de chaque processus.
- Cela peut se faire avec la technique du masque proposé dans l'algorithme LRU (Moins récemment utilisé)
  - Elle permet de ne conserver que les N pages référencées au cours des N derniers tops d'horloge.
  - Le paramètre N est déterminé expérimentalement pour chaque système.

6.4.2 Allocation locale et allocation globale?

- De quelle manière doit se faire l'allocation de la mémoire entre les processus concurrents prêts ?
  - L'allocation de la mémoire doit-elle se faire
    - de manière **équitable** avec un même nombre de pages pour chaque processus,
    - ou doit-elle se faire **proportionnellement** à la taille du processus ?
  - L'allocation de la mémoire doit-elle se faire
    - de manière **locale** (la recherche de la page à remplacer se faisant parmi les pages du processus ayant provoqué le défaut de page,)
    - ou de manière **globale** (parmi toutes les pages de tous les processus prêts) ?

6.4.2.1 Allocation à part égale ou proportionnelle

- L'**allocation à part égale** consiste
  - à déterminer régulièrement le nombre de processus en cours d'exécution,
  - et à allouer à chaque processus le même nombre de cadres de mémoire physique.
- L'**allocation proportionnelle** consiste
  - à allouer les pages proportionnellement aux tailles des programmes,
  - si un processus est deux fois plus grand qu'un autre, il recevra (à peu près) le double de cadres.

6.4.2.2 Remplacement de page local ou global

- L'**allocation locale** correspond
  - à un algorithme de remplacement local dont le rôle est de chercher la page à remplacer uniquement parmi celles du processus qui provoque le défaut de page.
- L'**allocation globale** correspond
  - à un algorithme de remplacement global qui étend la recherche à toutes les pages des processus prêts.

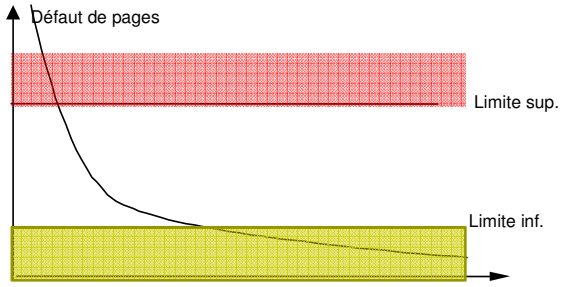
- L 'allocation **globale produit de meilleurs résultats** surtout si la taille de l'ensemble de travail peut varier dans le temps.
  - Avec un algorithme local, si l'ensemble de travail augmente de taille, il se produira forcément un écroulement (**thrashing**) et ce même s'il y a des pages mémoire libres.
  - Si la taille de l'ensemble de travail diminue, les algorithmes locaux font perdre de la mémoire car certaines pages allouées par le processus ne sont pas forcément utiles.

6.4.3 L'écroulement (thrashing) (1)

- L'écroulement/thrashing du système se produit lorsqu'un processus provoque trop de défauts de page.
  - Aucun des mécanismes d'allocation (à part égale ou proportionnelle) et de remplacement (local ou global) ne résout le problème de l'écroulement (trashing).
- L'écroulement du système ne peut être que surveillé. C'est le rôle de l'algorithme de la fréquence des défauts de page ou PFF **(Page Fault Frequency).**
  - Il limite le risque d'écroulement du système en maintenant le taux de défauts de page de chaque processus dans des limites acceptables.

6.4.3 L'écroulement (thrashing) (2)

- Lorsqu'un processus provoque trop de défauts de page (au dessus d'une limite supérieure), le système lui allouera plus de pages.
- Lorsqu'un processus dispose d'un taux de défauts de page trop faible (en dessous d'une limite inférieure), le système lui retirera une ou plusieurs pages.
- Si le système ne peut garder tous les processus au dessous de la limite supérieure (parce qu'il y a trop de processus en mémoire) le système retire un processus de la mémoire .



6.4.4 Le retour sur instruction

- Sur la plupart des processeurs, les instructions se codent sur plusieurs opérandes.
  - Si un **défaut de page se produit au milieu d'une instruction**, le processeur doit revenir au début de l'instruction initiale et la ré exécuter.
  - Ce retour sur instruction n'est possible qu'avec **l'aide du matériel**.
    - Le compteur ordinal est copié dans un 1er registre avant l'exécution de chaque instruction.
    - Les registres qui ont déjà été incrémentés ou décrémentés sont mémorisés dans un deuxième registre avec la valeur d'incrément correspondant.
  - Le système d'exploitation peut alors sans ambiguïté annuler les effets de l'instruction fautive (provoquant le défaut) et la réexécuter.

6.4.5 Le choix d'une taille de page

- La fragmentation interne
  - Un segment de code, de données ou de pile remplit très rarement un nombre entier de pages (en moyenne la moitié de la dernière page).
  - S'il y a N segments en mémoire et si la taille est de p octets, la fragmentation interne fait perdre en moyenne N*p/2 octets.
- Une taille de page trop élevée
  - Plus la taille d'une page est élevée, plus il y a du code/données inutile en mémoire. L'ensemble de travail du processus n'est pas forcément aussi large que la page de code/données chargée.
- Une taille de page trop faible
  - Si la taille des pages est faible, les programmes auront davantage de pages ce qui va nécessiter une plus grande table de pages.
  - Les transferts entre le disque et la mémoire se font par page. Le transfert d'une petite page prend pratiquement autant que le transfert d'une grande.

6.4.6 Le verrouillage des pages en mémoire

- La mémoire virtuelle et les entrées/sorties interagissent.
  - Soit un processus qui vient d'effectuer un appel système pour lire un fichier ou un périphérique et placer les données lues dans un tampon contenu dans son espace d'adressage.
  - Pendant qu'il attend la fin de l'E/S (il est à l'état Bloqué/Endormi en mémoire), le processus actuellement en cours d'exécution (le processus ELU) provoque un défaut de page.
  - L'algorithme de pagination étant global, c'est la page contenant le tampon d'E/S qui est retirée de la mémoire.
  - Le périphérique d'E/S était sur le point de faire un transfert DMA dans la page retirée, une partie des données sont écrites dans la page du tampon et une autre partie dans la page qui vient d'être chargée.
- La solution à ce type de problème consiste à *verrouiller les pages* qui sont concernées par le opérations d'E/S afin qu'elles ne soient pas retirées de la mémoire.

7. La segmentation

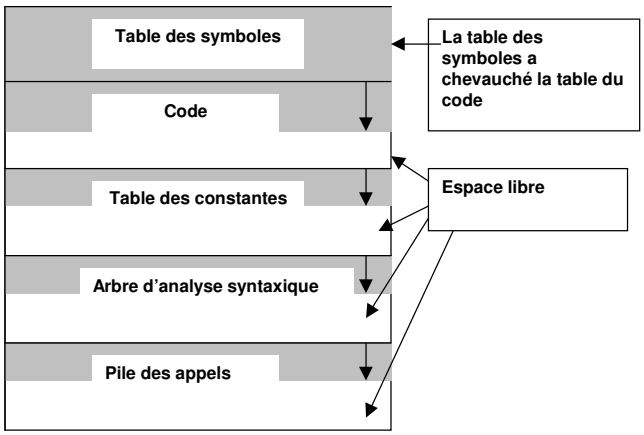
7.1. La segmentation par l'exemple
7.2. Intérêt de la segmentation
7.3. Comparaison de la pagination et de la segmentation

7.1 La segmentation par l'exemple (1)

- **La pagination**
  - propose un espace d'adressage plat et indifférencié,
  - les adresses virtuelles étant comprise entre 0 et une adresse maximale.
- Dans de nombreux cas, il est cependant préférable d'avoir d'avantages d'espaces d'adresses virtuelles.
- Soit un compilateur qui exploite de nombreuses tables :
  - Le code source sauvegardé.
  - La table des symboles contenant le nom des variables.
  - La table des constantes utilisées.
  - L'arbre d'analyse syntaxique du programme.
  - La pile utilisée pour les appels de procédures.

7.1 La segmentation par l'exemple (2)

- Dans un espace mémoire à une dimension, il faut attribuer à chacune de ces 5 tables un espace contigu.



7.1 La segmentation par l'exemple (3)

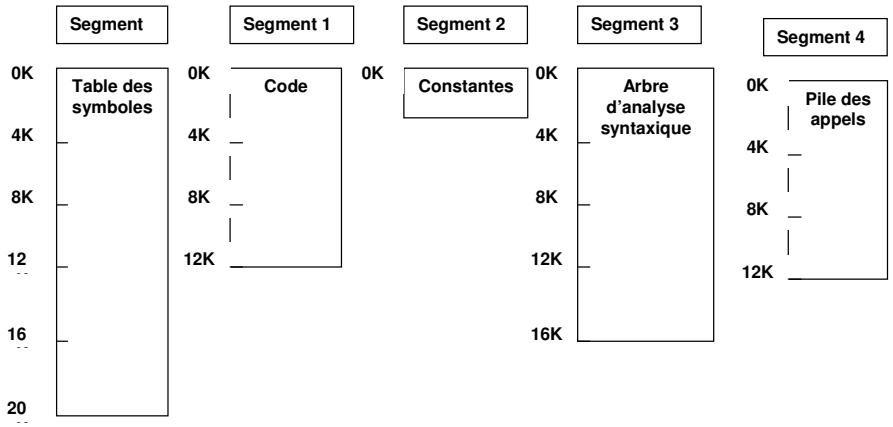
- Considérons le cas d'un programme qui comporte un très grand nombre de variables.
- La partie de l'espace des adresses allouée à la table des symboles peut se remplir alors qu'il reste beaucoup d'espace dans les autres tables.
  - Le compilateur peut évidemment signaler que la compilation ne peut être menée à terme en raison d'un trop grand nombre de variables, mais ceci ne serait pas très juste puisqu'il reste de l'espace libre dans les autres tables.
  - Une autre possibilité consiste à prendre de l'espace libre dans les tables vides et à l'attribuer aux tables pleines. Ce transfert est possible, mais revient à gérer ses propres recouvrements ce qui est gênant et pénible.

7.1 La segmentation par l'exemple (4)

- La solution consiste à disposer de plusieurs espaces d'adresses indépendants appelés **segments** (ou régions).
  - Chaque segment est une suite d'adresses contiguës de 0 à une adresse maximale.
  - Les segments ont en général des tailles différentes.
  - La taille d'un segment varie en cours d'exécution. La taille d'un segment de pile peut augmenter lorsque quelque chose est empilé et diminuer lorsque quelque chose est dépilé.
  - Un segment est une entité logique que le programmeur doit manipuler. Il contient en général des objets de même type.
  - Pour spécifier une adresse dans un espace mémoire à segmentation, le programme doit fournir une adresse en 2 parties :
    - un numéro de segment,
    - une adresse au sein du segment.

7.1 La segmentation par l'exemple (5)

- Chaque table de notre compilateur peut désormais croître indépendamment des autres.



7.2 Intérêts de la segmentation

- Simplification du traitement des structures de données dont la taille varie.
- Chaque segment formant une entité logique connue du programmeur, il peut avoir des **protections différentes**.
- Simplification du **partage** des procédures et des données entre plusieurs processus.
- Edition de lien facilitée si chaque procédure occupe un segment distinct dont l'adresse de départ est 0.
  - L 'appel à la procédure du segment n utilisera l'adresse en deux parties (n,0) pour adresser l'octet 0 qui est le point d'entrée du segment.
  - On peut modifier et recompiler la procédure du segment sans avoir à refaire une édition de liens, l'adresse de base 0 n 'ayant pas été modifiée.

Petite conclusion

- Concurrence + facilité de développement ⇨ techniques de gestion de la mémoire
- **Swap**: plus grand nombre d'espaces d'adressage chargés
- **Pagination**: éviter la fragmentation externe
- **Mémoire virtuelle**: charger des parties d'espace d'adressage seulement
- **Segmentation**: différencier fonctionnellement les régions de l'espace d'adressage

Références bibliographiques

- Andrew Tanenbaum « *Systèmes d'exploitation* » 2ème édition Pearson Education.
- Jean-Marie Rifflet « *La programmation sous Unix* » 3ème édition EdiScience.
- Beauquier Joffroy « *Systemes d'exploitation - concepts et algorithmes* » Ediscience
- Daniel Bovet, Marco Cesati « *Le noyau Linux* » 3ème édition O'Reilly.