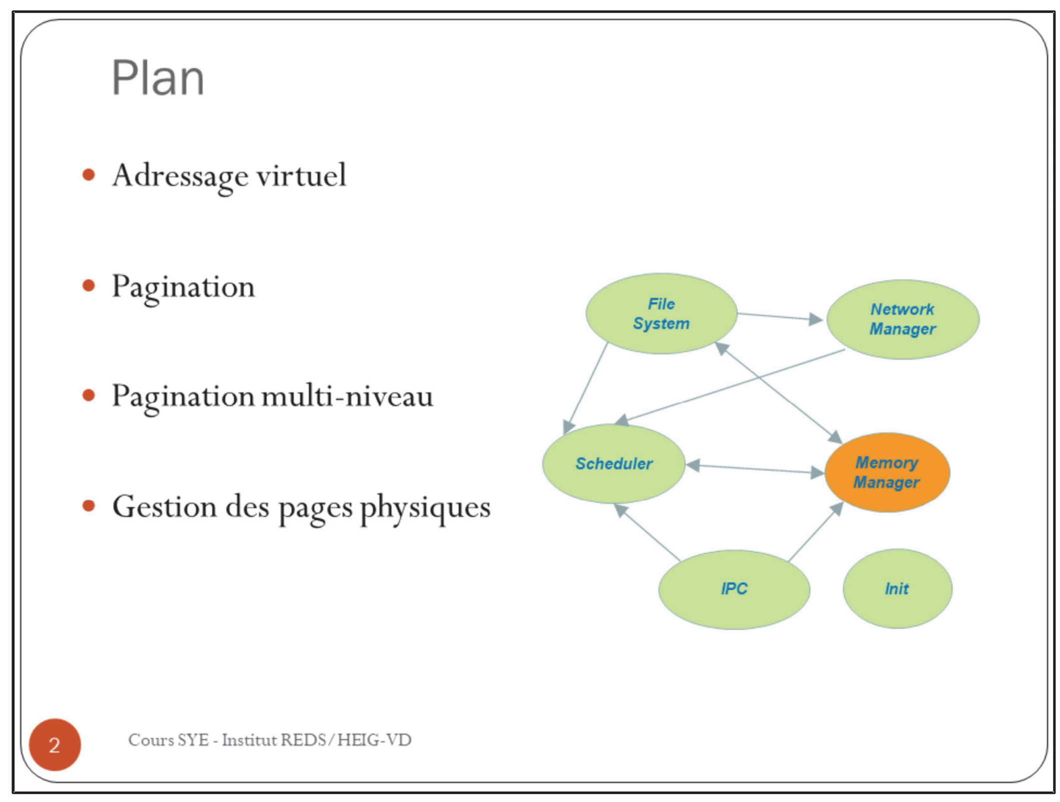
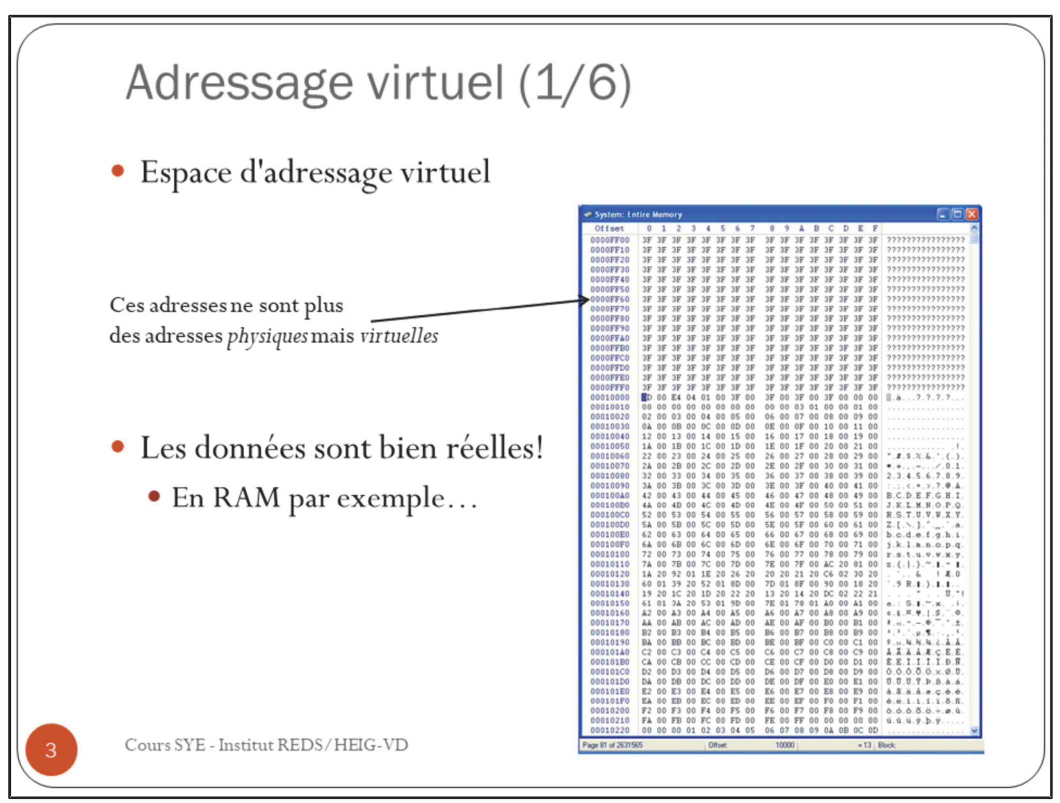
# Systèmes d'exploitation
## *Pagination*

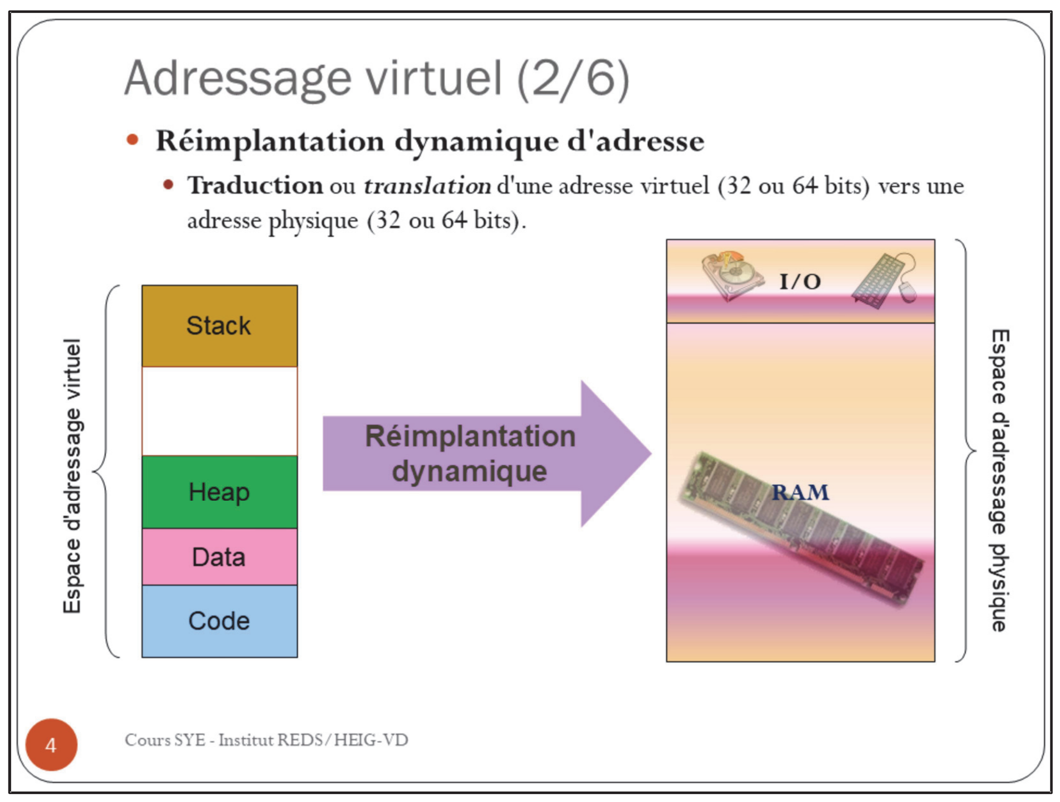
Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

Version 3.5 (2017-2018)


Plan
Adressage virtuel
Pagination
Pagination multi-niveau
Gestion des pages physiques
File System
Network Manager
Scheduler
Memory Manager
IPC
Init
2
Cours SYE - Institut REDS/HEIG-VD



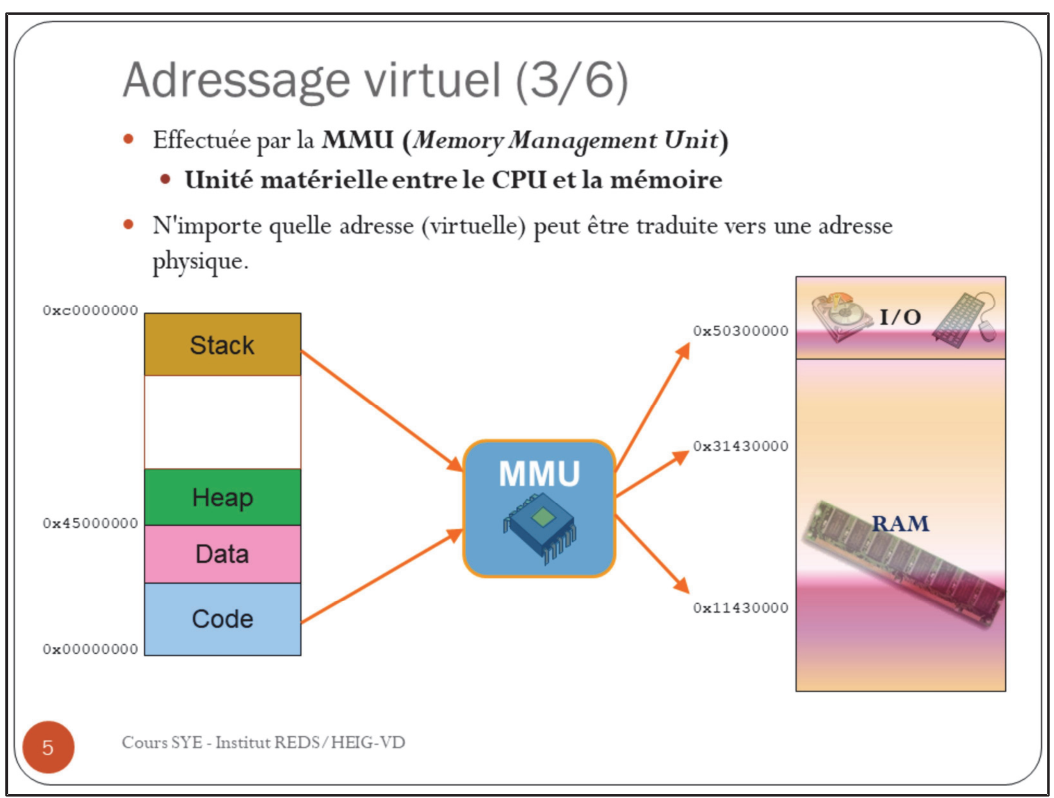

L'utilisation d'un espace d'adressage virtuel est le fondement de la notion de virtualisation qui date déjà des années 60, avec l'apparition du système d'exploitation *Multics*, et qui est couramment utilisé dans les OS depuis lors. C'est également grâce aux espace d'adressage virtuel qu'il est possible de gérer la mémoire virtuelle (en tant qu'extension de la mémoire physique) en utilisant des fichiers sur disque ou encore l'exécution de plusieurs OS sur une même machine grâce à un moniteur de machine virtuelle ou *hyperviseur*.




La réimplantation dynamique d'adresses virtuelles en adresses physiques consiste à effectuer une *translation* d'une adresse vers une autre, au moyen de mécanismes matériels. La translation est donc effectuée matériellement avec l'appui d'une ou plusieurs *tables de traduction* comme nous le verrons plus loin.

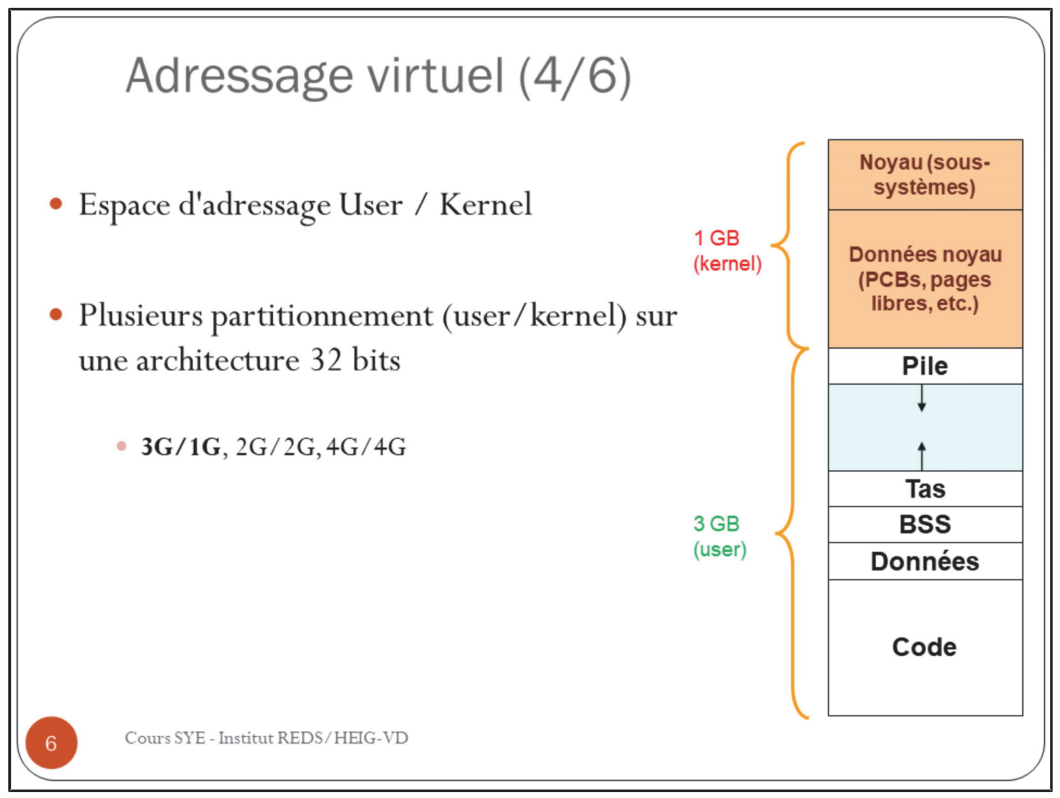
L'idée de la réimplantation dynamique d'adresse est de donner à tous les processus, i.e. à chaque contexte mémoire, une **traduction différente**. Cette traduction qui repose sur l'utilisation de tables nécessitera une **reconfiguration matérielle** à chaque changement de contexte de processus (changement de contexte mémoire).

Il faut noter que toutes les adresses virtuelles que constitue l'espace d'adressage virtuel n'ont pas forcément une équivalence en adresses physiques. Généralement, seules les adresses virtuelles "utiles" trouveront une traduction vers une adresse physique. Ainsi, un espace d'adressage virtuel peut se révéler beaucoup plus grand que l'espace d'adressage physique, dans sa représentation, mais en réalité seul un sous-ensemble de ses adresses trouveront une équivalence physique.




L'utilisation d'espaces d'adressage virtuel est donc possible grâce à l'intervention d'un composant matériel essentiel qui vient s'intercaler entre le processeur (CPU) et la mémoire: la ***MMU*** (***Memory Management Unit***). Les processeurs qui ne possèdent pas de MMU (microcontrôleur limité) ne peuvent pas gérer des espaces d'adressage virtuel.

Les changement de contexte de processus nécessiteront ainsi, systématiquement, un changement de traduction (translation) des adresses virtuelles vers les adresses physiques. De ce fait, la **MMU devra être reconfigurée** à chaque changement de contexte.



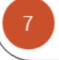

L'espace d'adressage représente l'ensemble des adresses "*visibles*" par le processus. Si l'espace d'adressage est défini sur l'ensemble possible des adresses (c-à-d 4 Go sur une architecture 32 bits), l'espace est scindé en deux: une partie **noyau** et une partie **utilisateur**. La partie noyau représente l'OS lui-même avec toutes ses structures de données internes. On se rappelle que le l'espace noyau contient le code noyau, et que celui-ci s'exécute dans le mode ***noyau*** du processeur (accès à tout le jeu d'instruction ainsi qu'à toutes les adresses mémoire). La partie utilisateur représente les données *utilisateur* du processus (code, données, piles, etc.). Par conséquent, lorsque le code du processus s'exécute, le processeur fonctionne en mode ***utilisateur***. Généralement, l'espace utilisateur occupe 3 Go, l'espace noyau 1Go (découpage 3G/1G). D'autres découpages sont cependant possibles comme 2G/2G, ou 4G/4G. Dans ce dernier cas, il y a un changement de tables de translation (donc une reconfiguration MMU) non seulement lors d'un changement de contexte entre processus, mais également lors d'une transition *user/kernel*. Cette configuration a été proposée dans le noyau *Linux*, mais par défaut, un découpage 3G/1G est utilisé.

L'approche qui consiste à considérer un espace complet de 4 Go par processus n'est possible que si l'on *virtualise* la mémoire disponible. Si un système ne dispose pas de MMU, l'espace d'adressage correspond alors à la taille de la RAM.

.

# Adressage virtuel (5/6)

- Avantages des espaces d'adressage virtuels
  - Isolation garantie entre les processus
  - Gestion facilitée de l'adressage, notamment au chargement
    - Utilisation possible d'adresses absolues
  - Permet l'utilisation d'une mémoire physique *étendue*






L'introduction des espaces d'adressage virtuels s'est vite fait ressentir afin de faciliter la gestion des adresses au niveau de l'image binaire: si l'on pense au chargement de celle-ci, une allocation directe en mémoire physique nécessite de *reloger* le code en fonction de sa position de départ. De plus, tout changement de position peut entraîner la nécessité de recalculer dynamiquement les adresses.

Or, avec un espace d'adressage virtuelle, les différentes sections d'un processus peuvent se trouver à des adresses fixes, même si en mémoire physique la position est différente: seul les tables de traduction devront être adaptées. On peut ainsi envisager de charger la section code d'un processus à l'adresse (virtuel) 0x00000000, de placer le sommet de la pile (descendante) à 0xc0000000 (3Go), etc.
**L'utilisation d'adresses absolues** est à nouveau possible dans le contexte d'un système multi-programmé (on se rappelle que le processeur ne manipule plus que des adresses virtuelles, la traduction étant assurée, matériellement, par une unité dédiée - la MMU - entre le processeur et la mémoire).

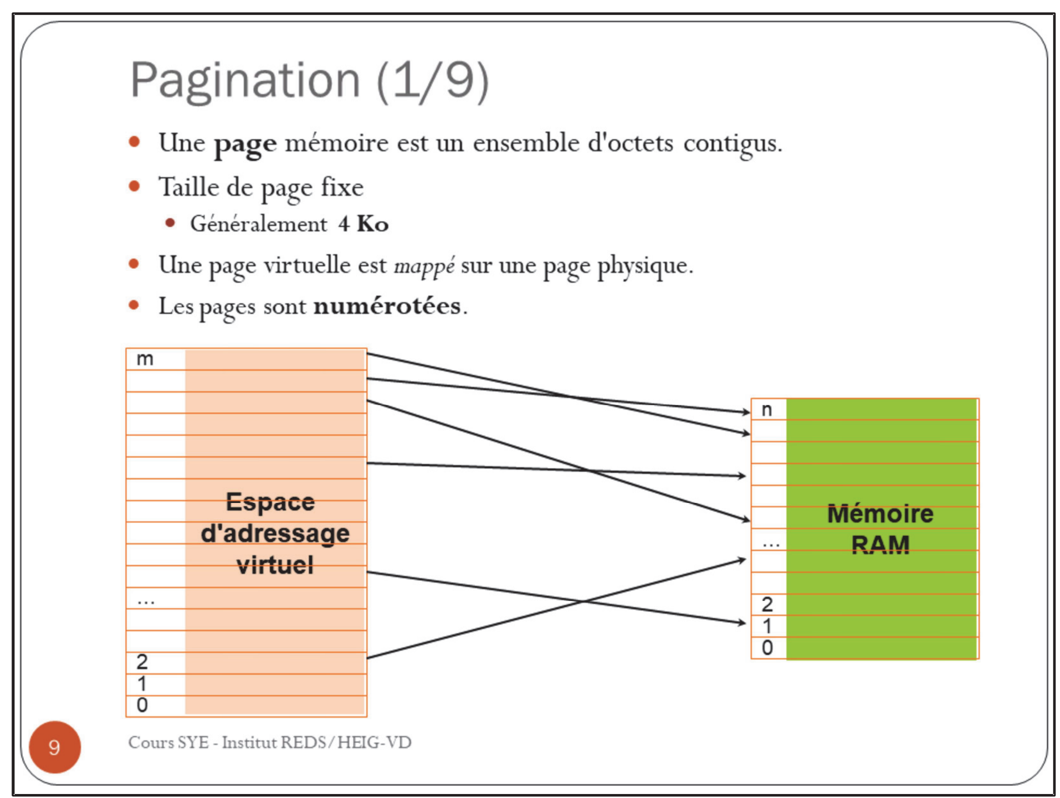
## Adressage virtuel (6/6)

- Avec un mécanisme d'adressage virtuel, comment pourrait-on gérer au mieux **l'allocation** en mémoire physique et les problèmes liés à la **fragmentation** en RAM ?
  - …en découpant l'espace d'adressage de telle manière à pouvoir charger des "petites" zones de mémoire
  - On adapte la (les) table-s de translation en conséquence
- La **pagination** permet de découper un processus en ***pages mémoire.***

Cours SYE - Institut REDS/HEIG-VD

Le mécanisme de traduction d'adresses permet également de gérer l'allocation d'un processus en mémoire physique et de réduire considérablement les problèmes de fragmentation. En effet, si l'on découpe le processus en zones mémoire plus petites, il serait possible d'adapter les tables de traduction **sans** que cela n'est le moindre impact sur l'espace d'adressage virtuel: les adresses virtuelles ne subissant aucune modification, l'image binaire reste inchangée. En revanche, les tables de traduction doivent être adaptées en conséquence: chaque zone virtuelle devra alors "pointer" vers les bonnes zones de mémoire physique.

On comprend aisément que les tables ne peuvent pas contenir une traduction pour chaque adresse: le mécanisme serait trop lent. C'est pourquoi, la traduction se fait par "bloc" d'adresse, et l'on considère plutôt une translation d'une région virtuelle vers une région physique. Une telle région représentera une ***page*** mémoire et sera de taille fixe. Le mécanisme réalisant ce découpage et permettant de traduire une page virtuelle vers une page physique s'appelle la **pagination**.




L'espace d'adressage virtuel est alors décomposé en pages *virtuelles* (de taille constante), et chaque page est *mappée* sur une page physique (de taille identique).

Aujourd'hui, les OS utilisent des systèmes paginés avec une taille de page de 4 Ko. Cette taille a été déterminée de manière empirique et convient bien à l'utilisation de la mémoire dans un système moderne. Cette taille de 4 Ko (4096 octets, référencés par 4096 adresses) correspond également à la taille habituelle d'un bloc de système de fichiers (NTFS, *ext2fs*, *reiserfs*, etc.) afin de faire correspondre rapidement un bloc de fichier avec un bloc (*page*) mémoire.

L'allocation en mémoire physique s'effectuera ainsi par page: c'est dire qu'il sera impossible d'allouer moins que la taille d'une page (par exemple 4096 octets si la taille est de 4 Ko). Allouer une page de 4 Ko consistera donc à trouver un "trou" de 4 Ko, n'importe où dans la mémoire: ce qui résoud presque complètement le problème que l'on avait avec une allocation en mémoire physique sans MMU, c-à-d de trouver un trou générant des résidus minimaux. Seul, des résidus inférieurs à la taille de la page (résidu *interne*) peuvent subvenir, mais ils restent tolérables par rapport à la taille de la mémoire.

## Pagination (2/9)

- **Mécanisme de translation**
- L'adresse virtuelle de 32 bits est décomposée en deux parties:
  - **Numéro** de page
  - Déplacement à l'intérieur de la page (*offset*)
- Utilisation d'une **table de page**

Cours SYE - Institut REDS/HEIG-VD

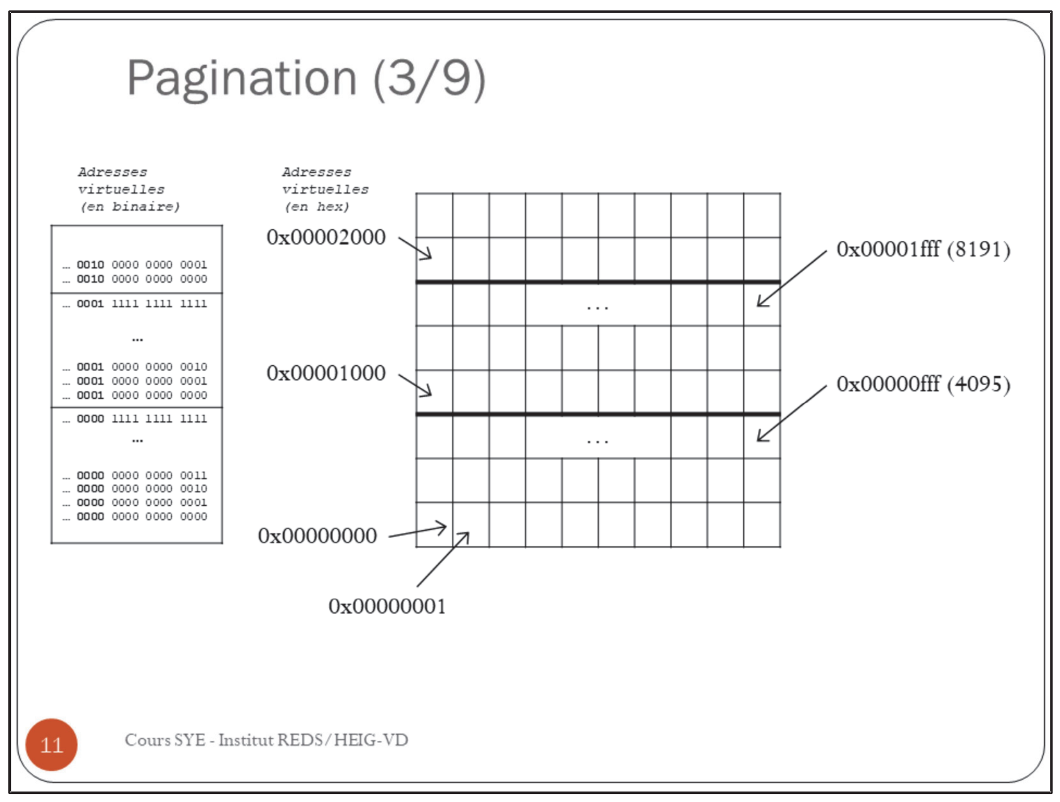
La traduction (ou *translation*) d'une page virtuelle vers une page physique s'effectue en prenant l'adresse virtuelle et en récupérant deux informations essentielles: le **numéro de page** et le **déplacement** (ou ***offset***) à l'intérieur de la page.

La MMU effectue cette décomposition et pourra ainsi, à partir du numéro de page, retrouver le numéro de page physique grâce à l'utilisation d'une table de traduction, appelée **table de pages**.

Examinons d'abord la structure de l'adresse virtuelle.

Au niveau de l'adressage, le découpage en numéro de page et déplacement est complètement transparent, dans la mesure où il n'influe en aucune façon la taille de l'espace d'adressage, ni la signification intrinsèque de l'adresse (une adresse permet toujours de référencer un seul *byte*).

L'illustration sur la page suivante montre comment le découpage de l'adresse virtuelle fonctionne.



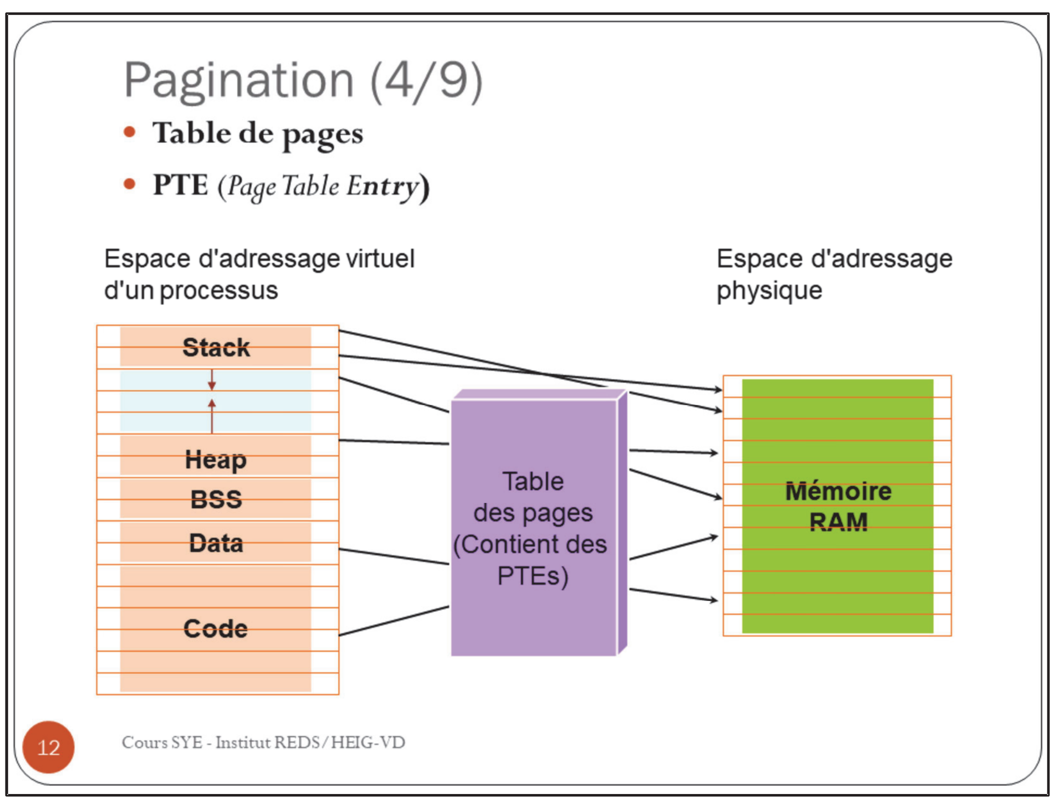

L'adresse est représentée sur 32 bits (pour une architecture de 64 bits, l'adresse sera représentée sur 64 bits).

Il faut considérer un *byte* dans la mémoire et son adresse (virtuelle) correspondante; le **numéro de page** correspond aux **bits de poids forts** de l'adresse, alors que le **déplacement** est donnée par les **bits de poids faibles**. Le déplacement représente la position du *byte* à l'intérieur de la page. Ainsi, si la page fait 4 Ko, il faudra 12 bits pour représenter le déplacement à l'intérieur de cette page ($2^{12}$ = 4096 octets). Si l'adresse fait 32 bits, il reste donc 20 bits pour représenter le numéro de page.

Avec 20 bits, nous obtenons donc $2^{20}$ pages, soit 1'048'576 pages. Le nombre total de *bytes* adressables est bien de $2^{20}$ **x** $2^{12}$ **=** $2^{32}$ ***bytes*** (taille maximale d'un espace d'adressage sur 32 bits).

On notera que la taille de la page dépend directement du nombre de bits représentant l'*offset*: si 8 bits sont utilisés pour "coder" le déplacement, la taille de page correspondante sera de 256 *bytes*.

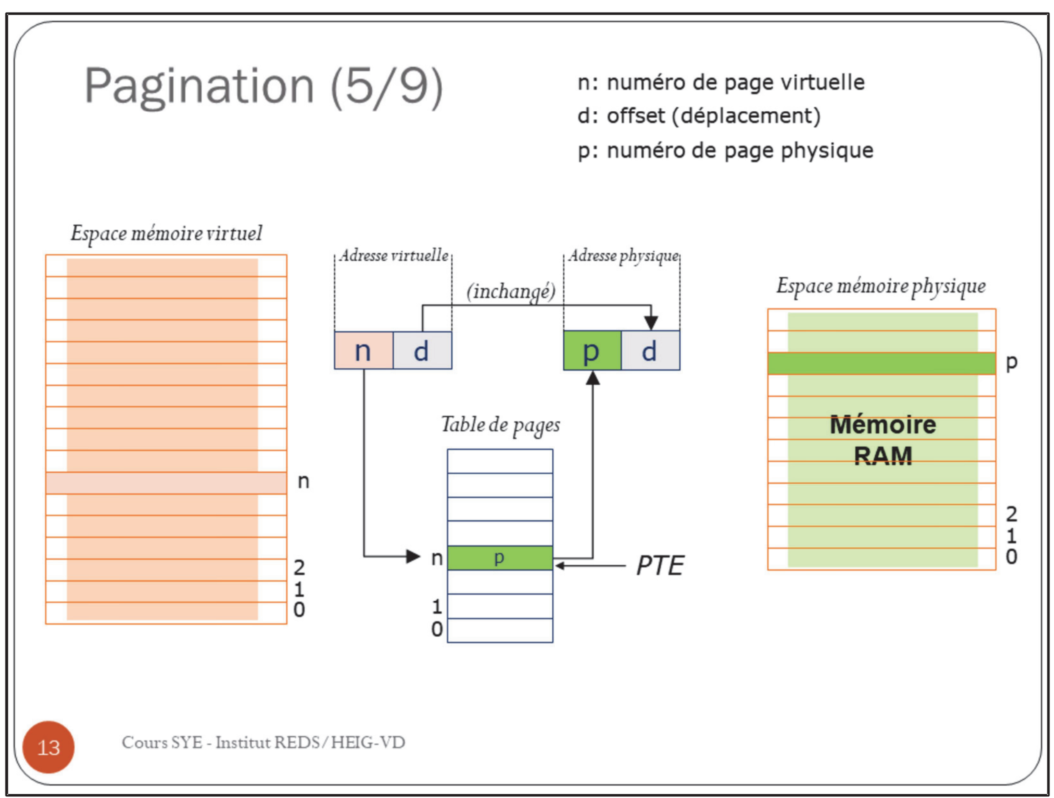
**Numéro de page** **Offset**




Le *mappage* d'une page virtuelle sur une page physique s'opère via une table de pages. Il s'agit d'une table d'éléments – appelées **PTE** (*Page Table Entry*) – contenant le **numéro de page physique** correspondant, plus un certain nombre d'informations caractérisant cette page (droits d'accès, présence en mémoire ou non, page ayant subit des modifications ou non, etc.). Habituellement, une telle entrée occupe 32 bits (4 octets).

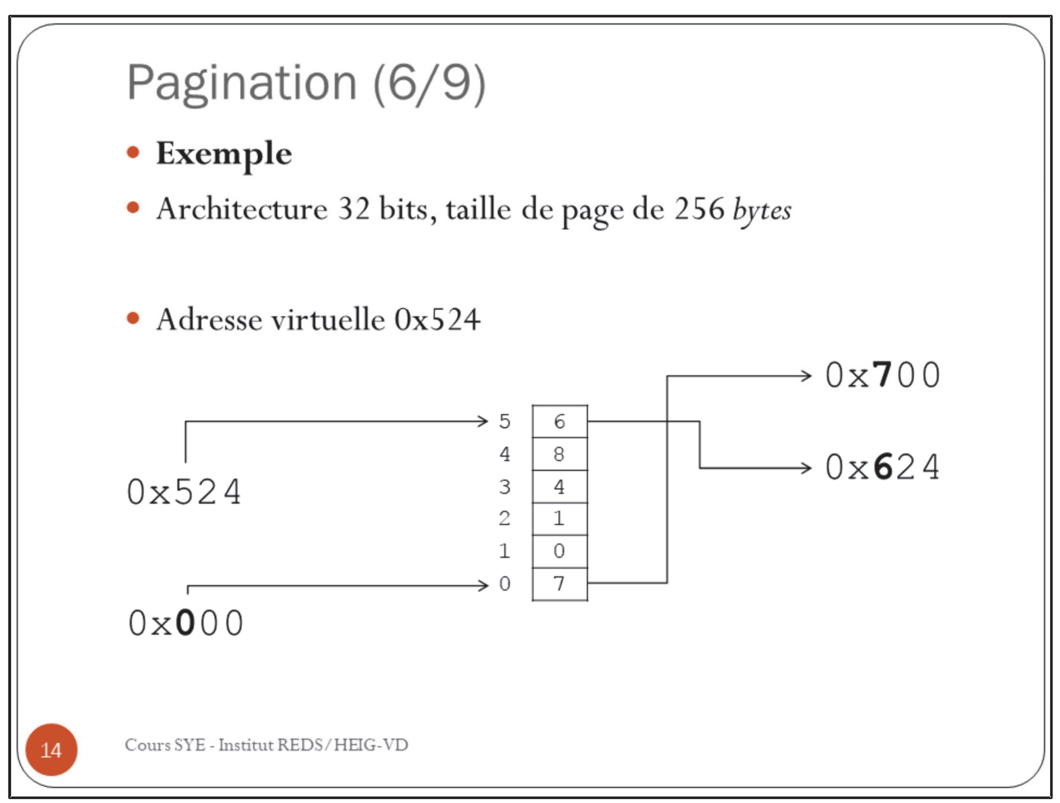
La traduction ne s'opérant qu'au niveau de la page, on notera qu'il n'y a pas lieu de stocker une quelconque information relative au déplacement (bits de poids faible de l'adresse virtuelle) lors de la traduction. Ce que la MMU doit récupérer sera donc le numéro de page physique afin de pouvoir calculer l'adresse physique résultante.

Le calcul de l'adresse physique (réelle) résultant consistera ainsi à récupérer **la PTE se trouvant à l'entrée no N**, N étant le numéro de page virtuelle. Le numéro de page physique constituera les bits de poids fort de l'adresse, et les bits de poids faible de l'adresse virtuelle correspondront à l'*offset*. La taille des pages étant égale des deux côtés (virtuelles et physiques), le déplacement sera parfaitement identique.

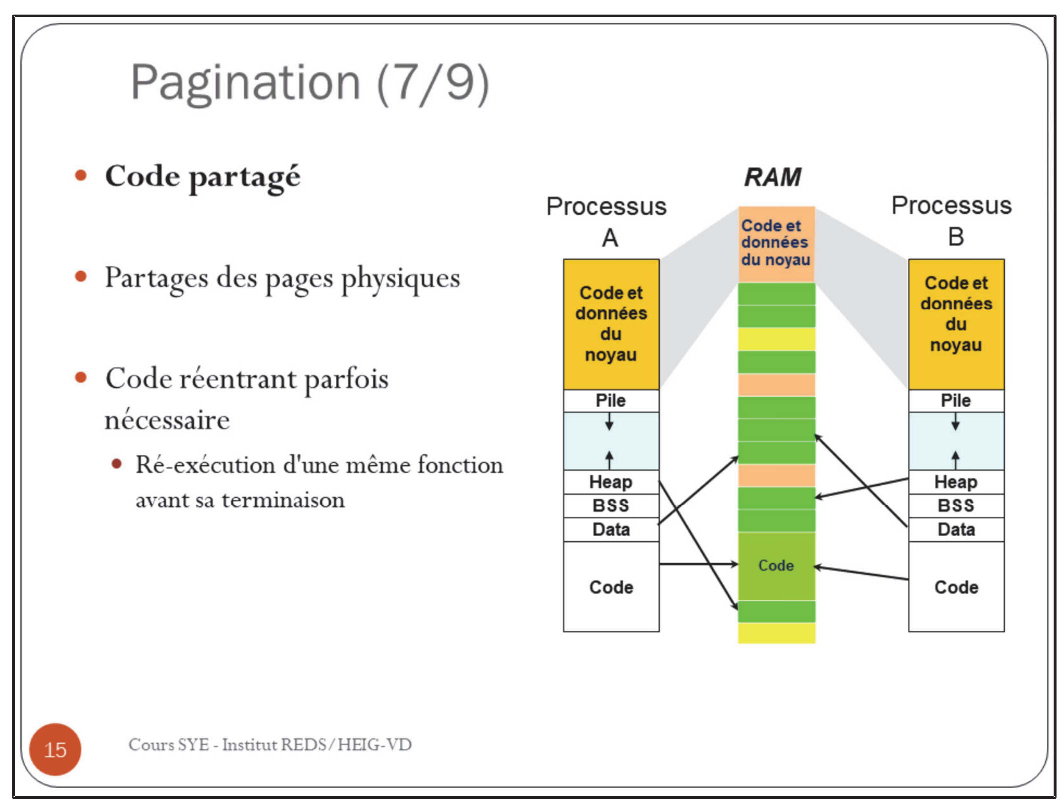



L'adresse virtuelle est décomposée en deux parties : ***n*** représente le numéro de page et ***d*** le déplacement (*offset*). La valeur *n* est utilisée comme index dans la table de pages pour rechercher la *PTE* correspondante.

La valeur ***p*** contenue à cette position dans la table représentera le numéro de page physique contenant le *byte* référencé. Cette valeur constituera ainsi les bits de poids forts de l'adresse physique. **Le déplacement *d* reste inchangé,** puisque la taille de la page virtuelle est identique à celle de la page physique.




Dans l'exemple ci-dessus, la page a une taille de 256 octets ($2^8$ octets), ce qui signifie que **8** bits sont nécessaire pour se déplacer à l'intérieur de la page : l'adresse virtuelle et physique auront donc 8 bits de poids faible (2 positions hexadécimales).




Lorsqu'un processus utilise le même code qu'un autre processus, ou qu'il désire partager une portion de la mémoire avec un autre processus, certaines zones de son espace d'adressage peuvent être *mappées* sur des **pages physiques identiques.**

De même, les zones appartenant à l'espace noyau (noyau OS) seront mappées sur les mêmes zones physiques. Nous comprenons ainsi comment un espace d'adressage virtuel peut contenir du code et des données identiques à un autre espace d'adressage virtuel, **sans qu'il y ait réellement duplication du contenu**.

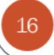
Cette technique est largement utilisée pour le code partagé, notamment celui de la *libc* (appels systèmes par exemple).

Le code pouvant être exécuté par plusieurs processus de manière totalement indépendante implique que les fonctions soient ***réentrantes***, c-à-d qu'elles ont été développées de telle sorte à gérer les **accès concurrents** aux variables globales et autres ressources partagées.

L'initialisation des tables de pages au démarrage d'un processus est pris en charge par le *chargeur* de l'OS (*loader*) qui analysera l'image binaire – dans les format *ELF* par exemple – et déterminera quelles pages physiques existantes doivent être mappées dans l'espace virtuel du processus (code & données partagés).

## Pagination (8/9)

- La table de page est stocké en mémoire physique.
  - La MMU connaît l'emplacement de la table de pages.
  - Adresse physique de la table de pages
  - *Page Table Base Register* (registre cr3 (*Pentium*), CP15:c1 (ARM))
- Chaque processus dispose de sa **propre** table de pages.
- Changement de contexte au niveau du processus
  - Changement de contexte mémoire
  - **Reconfiguration de la MMU**




Cours SYE - Institut REDS/HEIG-VD

Un système paginé présente l'avantage d'offrir des espaces d'adressage virtuels individuels et isolés aux différents processus. Il est donc nécessaire, à chaque changement de contexte, de reconfigurer la MMU de telle manière que celle-ci *pointe* vers la bonne table de page: chaque processus dispose de son propre mappage de son espace d'adressage vers une région de mémoire physique. Dès lors, on peut facilement se représenter la situation ou plusieurs processus *mappent* des zones mémoire virtuelle vers une zone mémoire physique **identique**. Cette approche permettra d'envisager le partage de code (librairies par exemple) entre plusieurs processus.

Il faut se rendre compte que la table de pages doit être stockée en mémoire. L'adresse physique de la table de pages sera donc transmise à la MMU à chaque changement de contexte mémoire.

Un autre aspect important d'un système paginé est la nécessité de multiplier les accès mémoire. En effet, pour accéder un octet en mémoire, il faut d'abord faire **un premier accès** afin de récupérer la PTE correspondante dans la table de page (la PTE se trouvant en mémoire). Puis, **un second accès** pour aller chercher la donnée avec l'adresse physique. On peut donc considérer qu'un tel système est **deux fois** plus lent qu'un système sans MMU. C'est pourquoi, sur les architectures avec MMU, on a recours à des mémoires *caches* entièrement dédiées au stockage temporaire de PTEs, afin d'accélérer les accès mémoire.

## Pagination (9/9)

- **TLB** (*Translation Look-aside Buffer*)

- *Caches* mémoire de niveau 2 (*Level 2* – **L2**)
  - Table associative (no page virtuelle, no page physique)
  - Entre 32 Ko et 2 Mo (plusieurs centaines de milliers d'entrée)
  - *Flushé* entre deux changements de contexte
  - Algorithme de remplacement des entrées

- Au fil du temps, plus de 98% de chance de trouver l'entrée en *cache*.

Cours SYE - Institut REDS/HEIG-VD

Les TLBs (*Translation Look-aside Buffer*) sont des entrées stockées dans la mémoire *cache* de niveau 2 (voire de niveau 3) sous forme d'une table associative. A chaque référence mémoire, la MMU va parcourir **d'abord** les TLBs pour voire s'il y a déjà la PTE correspondante en *cache*, et d'obtenir ainsi rapidement la page physique correspondante. Si tel n'est pas le cas, l'accès en mémoire (RAM) est requis, et la PTE sera copiée en cache.

L'utilisation de TLBs dans les systèmes paginés se fonde sur le principe de **localité des références.** En effet, un processus contient du code généralement organisé en fonction avec des variables locales. Lorsqu'un processus s'exécute, il passera donc un certain temps à l'intérieur d'une fonction, puis "sautera" dans une autre fonction pour y passer également un certain temps. Durant l'exécution au sein d'une fonction, les références mémoire seront souvent les mêmes (accès aux mêmes objets, variables locales, boucles, etc.). De ce fait, ces références se feront toujours dans la même page, voire un même **ensemble de pages**. On a donc tout intérêt à conserver les PTEs en *cache* comme ceux-ci seront très sollicités durant un certain temps, avant de changer de localité de référence, et de devoir charger d'autres PTEs.

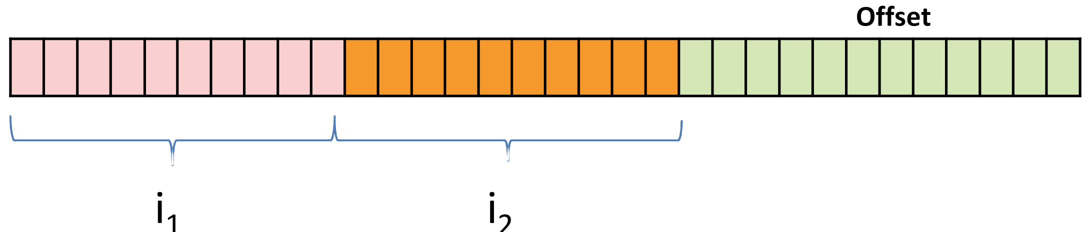
## Pagination multi-niveau (1/5)

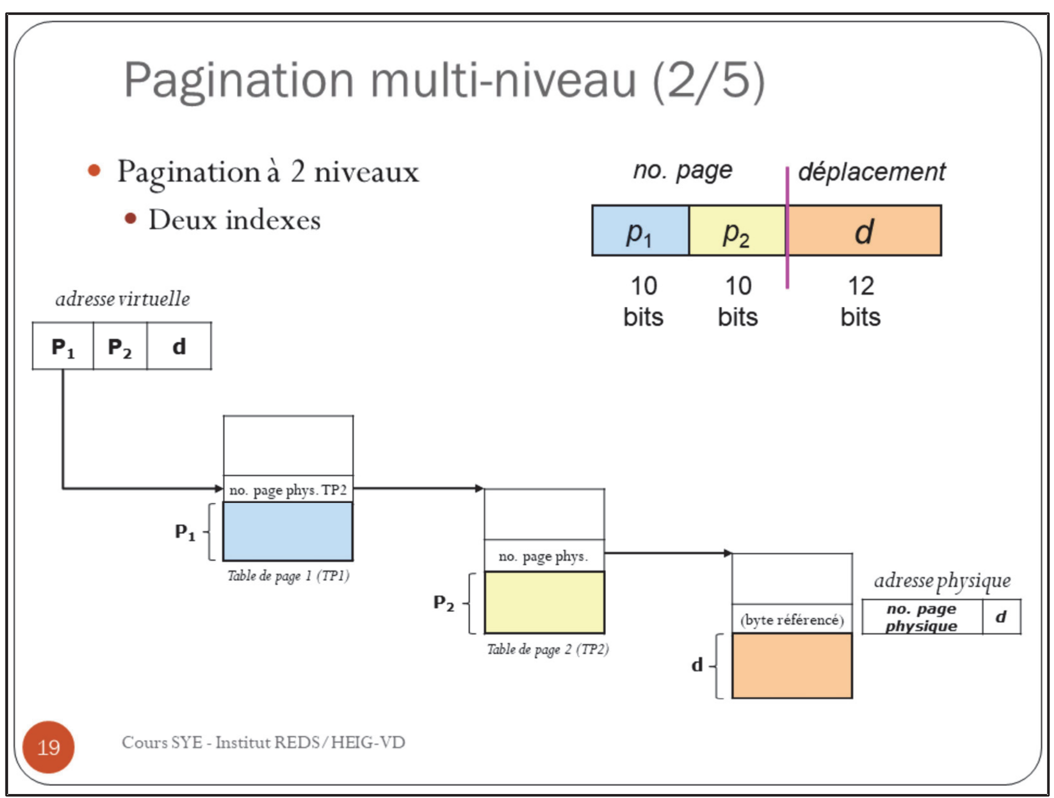
- La pagination à un niveau (sur une architecture 32 bits) entraîne l'utilisation de table de pages **très grande**.
  - 32 bits x $2^{20}$ entrées = **4 Mo** !
  - On doit donc rechercher un "trou" de 4 Mo
    - Problème déjà connu !...
- **Pagination à deux niveaux**
  - Utilisation de deux tables de page, à la place d'une, mais plus petite.
  - **Indirection** supplémentaire dans la traduction d'adresse

18 Cours SYE - Institut REDS/HEIG-VD

Jusqu'à maintenant, nous avons étudié le fonctionnement d'un système paginé utilisant une table de pages contenant des PTEs pour récupérer le numéro de page physique lors de la traduction d'adresse. Nous savons que l'utilisation de page de 4 Ko facilite grandement l'allocation en mémoire physique. Or, la table de page doit être également chargé en mémoire physique, et on se rappelle que chaque processus dispose de sa propre table de pages. Une table de pages contient autant d'entrée (PTE) qu'il y a de page virtuelle *potentiellement* utilisable: si chaque entrée occupe 4 octets, la taille de la table sera de 4 Mo. Et on se trouve à nouveau avec un problème d'allocation en mémoire physique pour trouver une zone contiguë de 4 Mo. Cela est inacceptable en terme d'utilisation mémoire.

C'est pourquoi, sur une architecture 32 bits, nous allons introduire une étape supplémentaire dans le mécanisme de traduction qui consistera à passer par une seconde table de page. Le mécanisme consiste à décomposer les 20 bits de poids forts en deux parties égales: **deux indexes** seront utilisés pour "naviguer" dans deux tables différentes, mais plus petite.



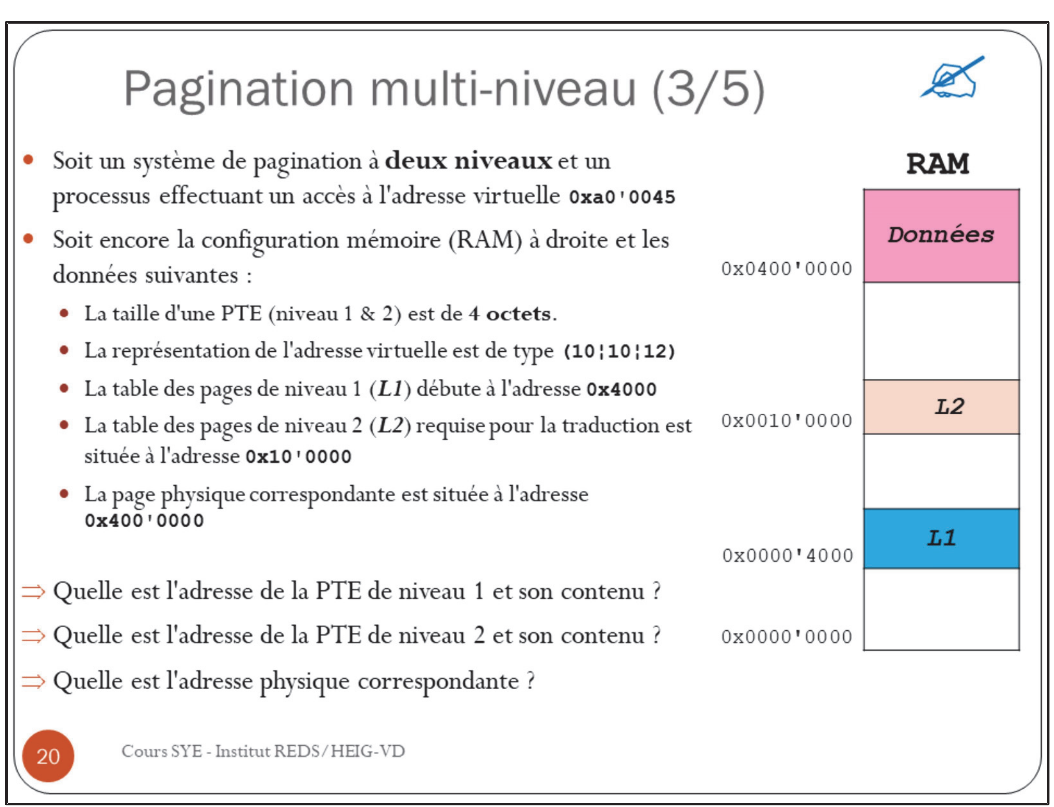



Si l'on considère 10 bits pour l'utilisation d'un index dans une table, la table pourra contenir $2^{10}$ entrées, soit 1'024 entrées (PTEs). Si chaque PTE reste encodée sur 4 octets, la taille d'une table de page sera égal à 4'096, soit exactement la taille d'une page (de 4 Ko) ! Cette propriété très intéressante permet de gérer les tables de page d'une manière optimale, car chaque table occupera l'entièreté d'une page.

Le mécanisme fonctionne exactement de la même manière qu'un système à un niveau, **à l'exception** que la première table de page donne la page physique contenant une **seconde table de page** (et non la donnée directement). Une fois la seconde table de page récupérée, on utilise le second index (index de niveau 2) afin de récupérer la PTE correspondante. Cette PTE contiendra, enfin, la page physique contenant la donnée adressée.

Ce mécanisme fait intervenir une indirection de plus, donc un accès mémoire supplémentaire. Comme précédemment, les PTEs de niveau 1 et de niveau 2 pourront se trouver dans les TLBs, accélérant ainsi les accès mémoire.

Pour chaque processus, il y aura ainsi une table de page de premier niveau (1024 entrées), où chaque entrée "pointera" vers une table de page de second niveau contenant à son tour chacune 1024 entrées**. 1024 entrées x 1024 entrées** donnent bien les 1 millions d'entrées de la table de page dans le système à un niveau. Avec ce système, nous n'avons pas plus, pas moins d'entrées; nous avons seulement paginé la table de page elle-même.

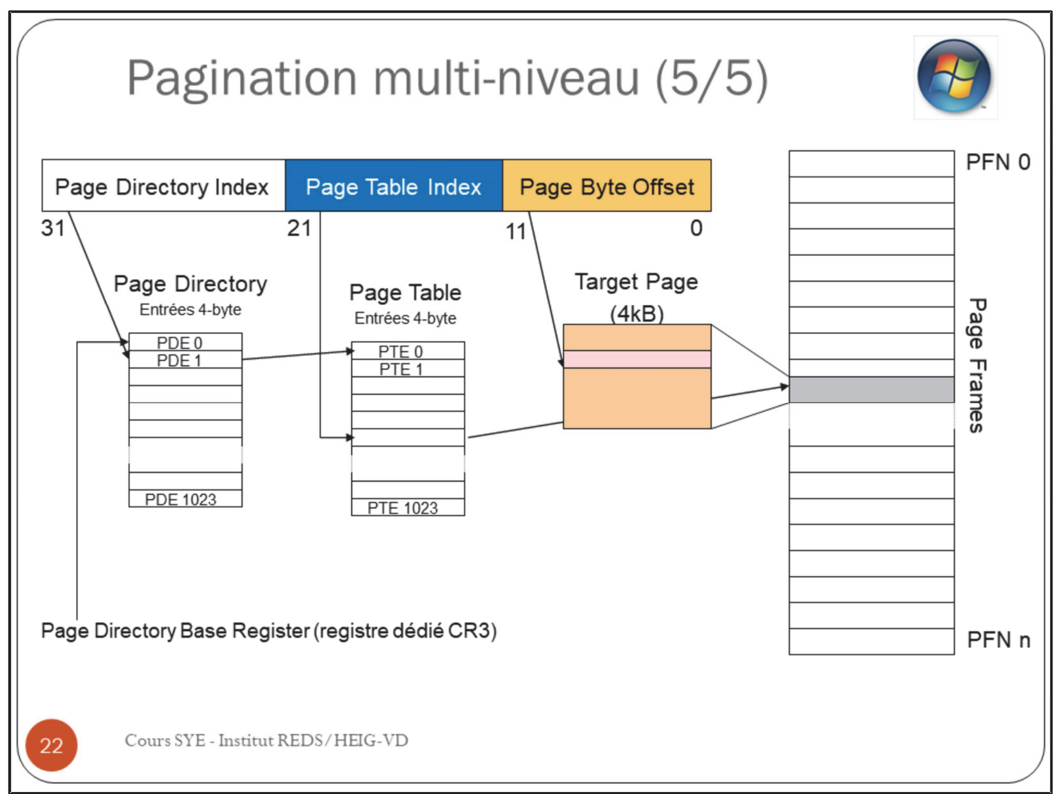

Pagination multi-niveau (3/5)
Soit un système de pagination à **deux niveaux** et un processus effectuant un accès à l'adresse virtuelle `0xa0'0045`
Soit encore la configuration mémoire (RAM) à droite et les données suivantes :
La taille d'une PTE (niveau 1 & 2) est de 4 **octets**.
La représentation de l'adresse virtuelle est de type (10¦10¦12)
La table des pages de niveau 1 (*L1*) débute à l'adresse `0x4000`
La table des pages de niveau 2 (*L2*) requise pour la traduction est située à l'adresse `0x10'0000`
La page physique correspondante est située à l'adresse `0x400'0000`
⇒ Quelle est l'adresse de la PTE de niveau 1 et son contenu ?
⇒ Quelle est l'adresse de la PTE de niveau 2 et son contenu ?
⇒ Quelle est l'adresse physique correspondante ?
RAM
Données
0x0400'0000
L2
0x0010'0000
L1
0x0000'4000
0x0000'0000
20
Cours SYE - Institut REDS/HEIG-VD

# Pagination multi-niveau (4/5)

- Soit un mécanisme de pagination à deux niveaux où la représentation de l'adresse virtuelle est comme suit: **(8¦4¦8)**
- Soit l'adresse virtuelle **0x2F09**
- La taille d'une PTE est de **8 bits**

⇒ Esquisser le contenu des tables de page pour une adresse physique valant **521** *(en décimal)* et si sa table de page de second niveau se trouve à l'adresse **1024**.

⇒ Quelle sera la taille de chaque table ?




Dans la terminologie *Windows*, la table de page de premier niveau s'appelle *Page Directory* et la table de second niveau *Page Table.* La taille de la page est de 4 Ko pour la plupart des pages mémoire. La page physique est aussi appelée *Page Frame* (le numéro de la page physique est donc la *pfn* pour *Page Frame Number*).

L'index de la table de premier niveau est codé sur 10 bits, tout comme l'index de la table de second niveau. Chaque entrée de table (*PTE*) faisant exactement 4 octets, la taille de chaque table sera donc de 4 Ko ($2^{10}$ multiplié par 4 octets).

Dans les OS modernes, la pagination s'effectue ***à la demande*** (*on-demand paging*) : cela signifie qu'un processus au démarrage n'a aucunement besoin de charger l'ensemble de toutes ses pages, mais de se contenter de charger les pages **au fur et à mesure** des besoins, réduisant ainsi l'utilisation de la mémoire physique. Cette approche nécessite une gestion efficace du chargement des pages et sera décrite dans le chapitre suivant.

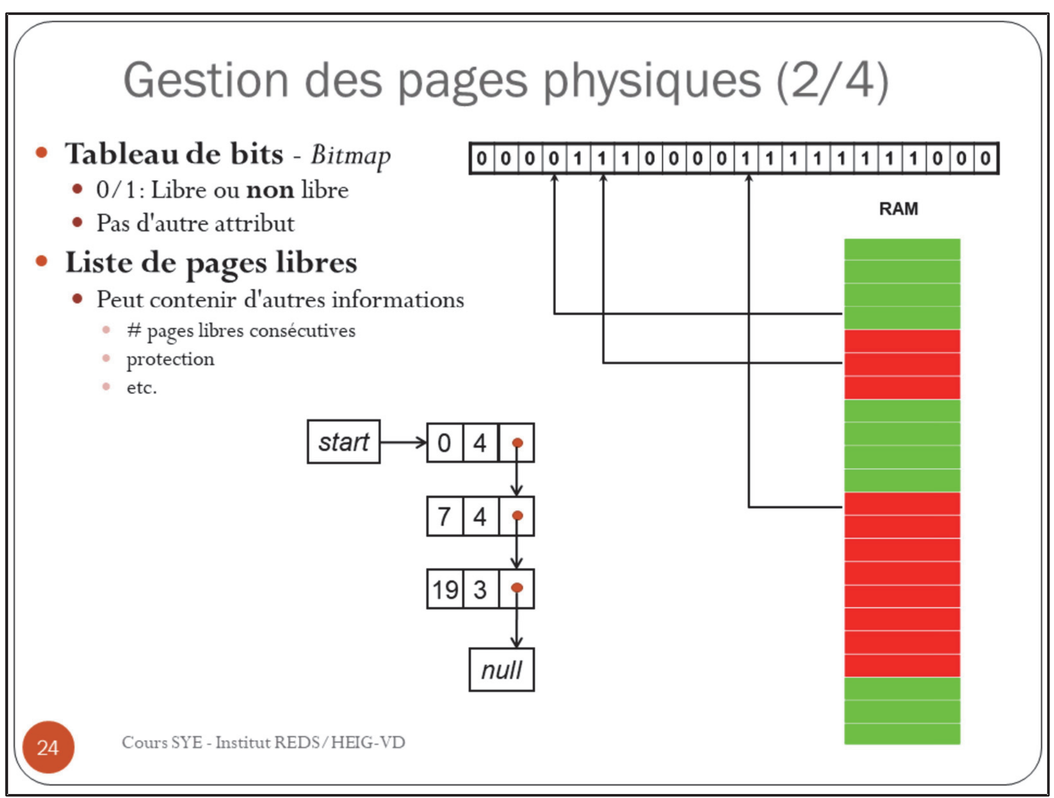
## Gestion des pages physiques (1/4)

- La mémoire physique (RAM) découpée en page constitue un *récipient* de pages.
- Le gestionnaire mémoire doit offrir un allocateur de page physique.
- Les pages sont restituées lorsqu'un processus se termine.

Cours SYE - Institut REDS/HEIG-VD

S'il n'est plus nécessaire de trouver une zone contiguë de pages physiques au chargement d'un processus, il est cependant nécessaire de gérer convenablement l'ensemble des pages physiques qui peuvent être allouées par les processus.

Par ailleurs, certaines zones de l'espace virtuelle nécessitent des pages physiques contiguës lorsqu'il s'agit de transfert *DMA* (*Direct Memory Access*) entre le périphérique et le CPU. Les zones mémoires utilisées pour de tels transferts sont limités à quelques mégaoctets, mais nécessitent quasiment toujours une contiguïté dans l'espace physique.




La gestion des pages libres nécessite l'utilisation d'une structure de donnée adéquate. Généralement, deux structures sont utilisées: un tableau de bit (*bitmap*) où chaque bit représente une page physique, et une liste de pages libres.

La *bitmap* permet de récupérer un numéro de page libre très rapidement, et de déterminer s'il existe des pages contiguës: il peut être en effet utile de récupérer des pages contiguës dans le cas particulier d'accès direct mémoire (DMA) lorsqu'un périphérique doit transférer des blocs de données rapidement. La taille de la bitmap dépendra donc directement du nombre de la taille de la mémoire physique, et cette taille restera constante.

Une liste de pages libres en revanche permettra de stocker d'autres informations relatives à la page physique: son type, ses droits d'accès, etc. La liste présente l'avantage d'avoir une taille variable, et d'être réduite à néant lorsque toute la mémoire est utilisée.

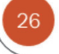
# Gestion des pages physiques (3/4)

- Soit une mémoire de 256 Mo.
- Taille de page: 4 Ko

Quelle sera la taille de la *bitmap*?

## Gestion des pages physiques (4/4)

- Si toutes les pages physiques sont allouées, deux cas de figures:
  - *Memory Overflow*
  - Nécessité d'effectuer un *swapping* de page
    - Une page peut être transférée sur le disque et libérer la place.
    - Il s'agit de sélectionner quelle page doit être remplacée.
    - *Chapitre* "**Mémoire virtuelle**"




Cours SYE - Institut REDS/HEIG-VD

Lorsque la mémoire physique (RAM) est pleine, il n'y a plus de possibilités de charger de nouveaux processus, ou d'effectuer de nouvelles allocations mémoire. Ainsi, il est nécessaire de retirer des pages mémoire existantes afin de faire de la place, en stockant *temporairement* ces pages sur disque par exemple, jusqu'à ce que l'on en a besoin à nouveau. Cette technique sera étudiée en détail dans le chapitre suivant.

## Références

- A. Silberschatz et al.:
  "Operating Systems Concepts", 6th edition, Wiley
- Andrew Tanenbaum,
  "Systèmes d'exploitation", 2ème édition