REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

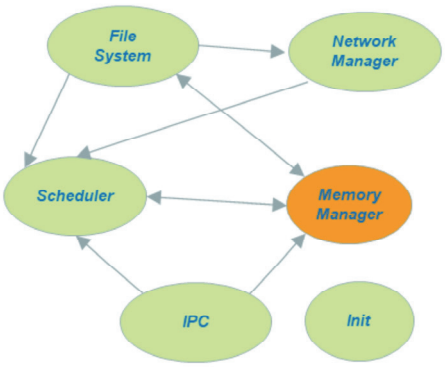
Systèmes d'exploitation
Mémoire virtuelle

Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

Version 3.4 (2017-2018)

Plan

- Extension mémoire
- Faute de page
- Algorithme **OPT**
- Algorithme **FIFO**
- Algorithme **LRU**
- Algorithme de la *seconde chance*
- Algorithme **WSClock**



Extension mémoire (1/4)

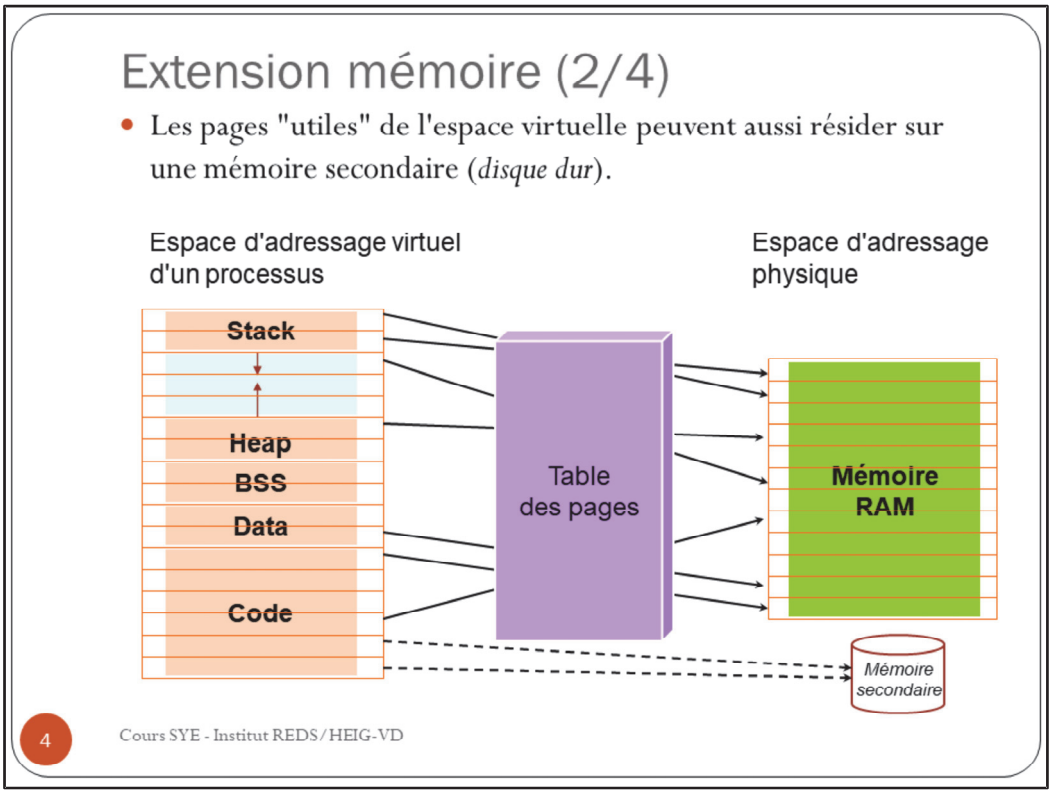
- Système paginé avec une MMU
  - Le processus ne sait rien sur l'espace d'adressage physique.
- Les pages physiques peuvent subir des **mouvements**.
  - Déplacement d'une page en RAM
  - Déplacement d'une page (momentanément inutile) sur le disque dur
  - Déplacement d'une page depuis le disque dur en RAM
  - *Swap* au niveau des pages

- Les **tables de page** sont mises à jour en conséquence.

3 Cours SYE - Institut REDS/HEIG-VD

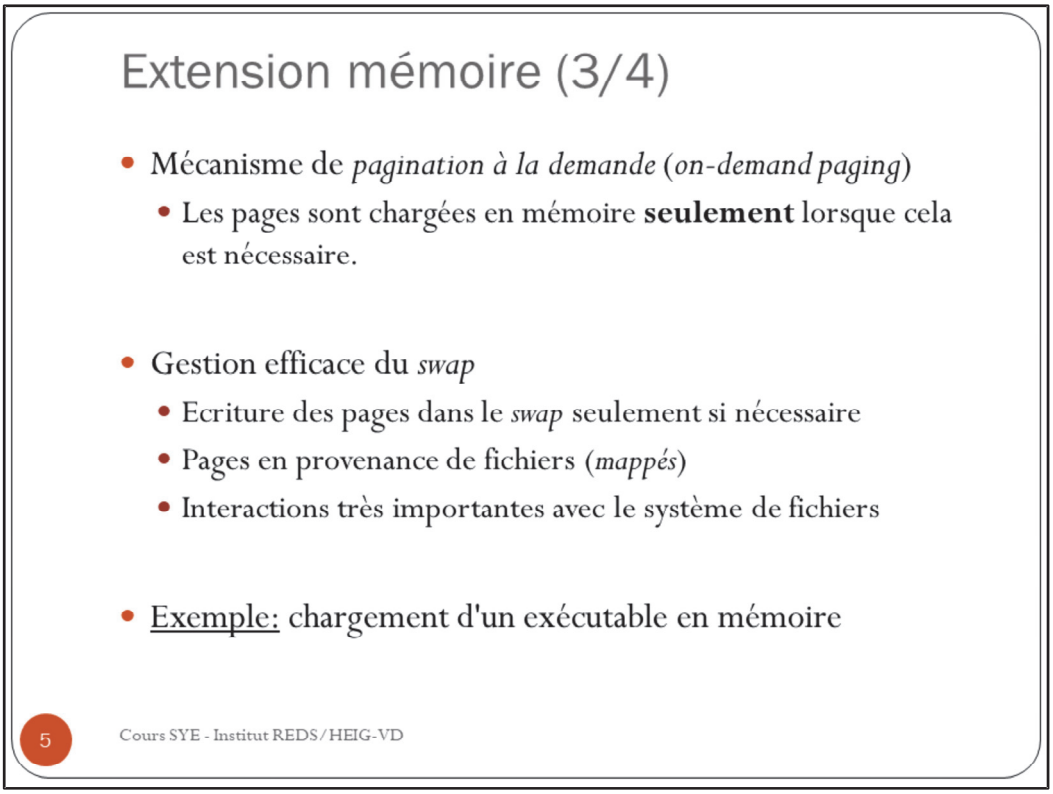
Un système paginé permet de "découper" un processus en pages mémoire de taille fixe (typiquement 4 Ko). La MMU effectue la translation d'une page virtuelle vers une page physique de manière totalement transparente pour le processus en cours d'exécution. C'est dire que les pages physiques peuvent se trouver n'importe où, et de plus, moyennant une mise à jour adéquate des tables de page, ces pages pourraient "changer de place" durant l'exécution. Fort de ce principe, on peut donc envisager de déplacer des pages mémoire (physiques) de la RAM vers un support de stockage, et de les "remonter" du disque vers la RAM lorsque le processeur doit les utiliser. Sur une architecture mono-cœur, seul une page sera "utilisée" à un temps donné, jamais deux "en même temps". Dans une architecture multi-cœur, en revanche, plusieurs pages pourraient être utilisées au même moment, mais les autres sont certainement pas.

Cette opération de déplacement de page (de la RAM vers le disque, ou du disque vers la RAM) est appelé *swapping*. Dès lors, lorsque la mémoire devient pleine, il est possible "d'éjecter" des pages physiques pour les stocker *temporairement* sur le disque dur, dans une partition de type *swap* ou un fichier spécial (*c:\pagefile.sys* sous Windows), permettant ainsi d'étendre la mémoire physique de manière *virtuelle* au-delà de sa taille réelle. La taille de la mémoire (virtuelle) dépendra alors directement de la taille de la partition *swap* (ou du fichier *swap*) prévue à cet effet.



En principe, le chargement d'un processus en mémoire consiste à charger les pages de celui-ci, c-à-d les pages mémoire qui vont contenir par la suite les différentes sections du processus (code, *data*, *BSS*, etc.).

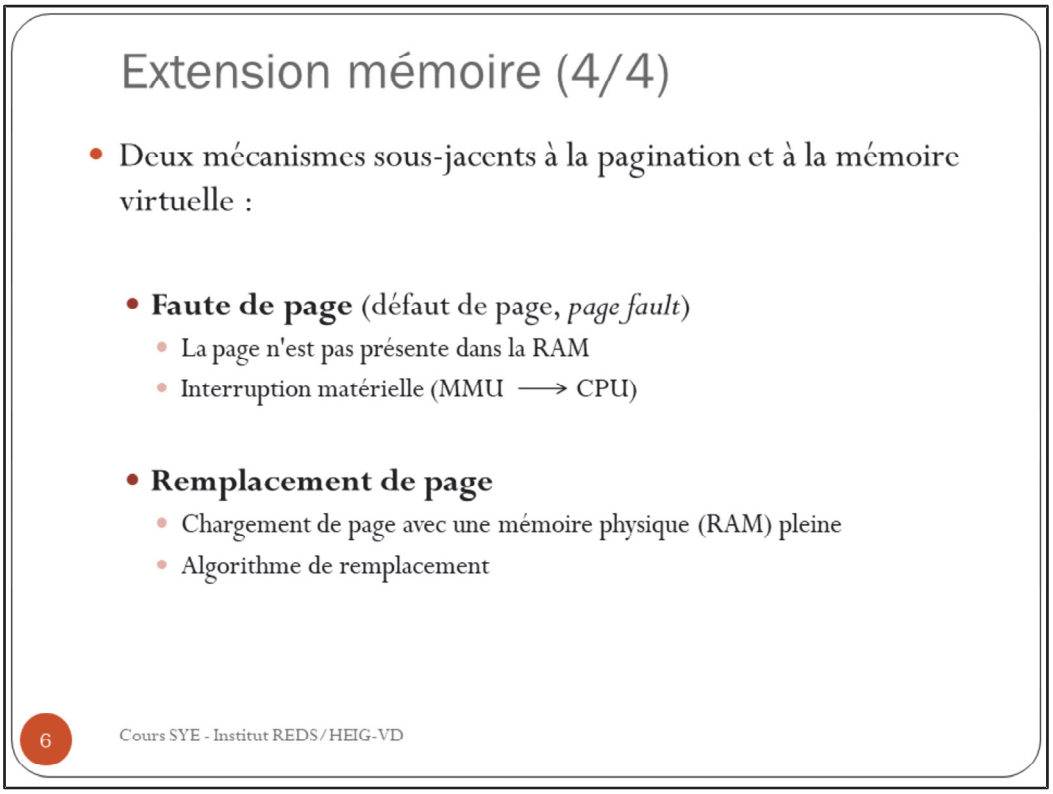
Bien évidemment, une section pouvant être répartie sur plusieurs pages (dépendant de sa taille).



Le chargement de l'intégralité des pages en mémoire physique n'est pas forcément nécessaire, puisque toutes ces pages ne sont pas utiles au même moment. Par conséquent - et comme évoqué précédemment - il devrait être possible d'avoir en mémoire seulement la page (voire quelques pages supplémentaires correspondant aux prochaines instructions à exécuter par exemple). Cette approche amène un concept intéressant permettant d'éviter de charger trop la mémoire au chargement d'un processus: la **pagination à la demande**. Les pages sont ainsi chargées *au fur et à mesure* que celles-ci sont nécessaires. On peut bien s'imaginer que beaucoup de pages risquent d'être chargées au début de l'exécution du processus. Les fonctions appelées plus tard - correspondant à des emplacements différents dans le code - nécessiteront alors un chargement de pages seulement lors de leurs invocations.

Lors d'un *swap* éventuel de pages, ces dernières peuvent souvent provenir d'un fichier; l'exemple typique étant des pages contenant du code issu de l'image binaire stocké sur disque. Le code n'étant pas altéré (en principe), il ne serait pas nécessaire de stocker ces pages de code dans le *swap*, puisque l'on sait les récupérer à partir du fichier exécutable. Cette interaction très forte entre le gestionnaire mémoire (gestionnaire du *swap*) et le système de fichiers nécessite que les pages mémoire puissent facilement être associées à un équivalent de page dans le fichier binaire.

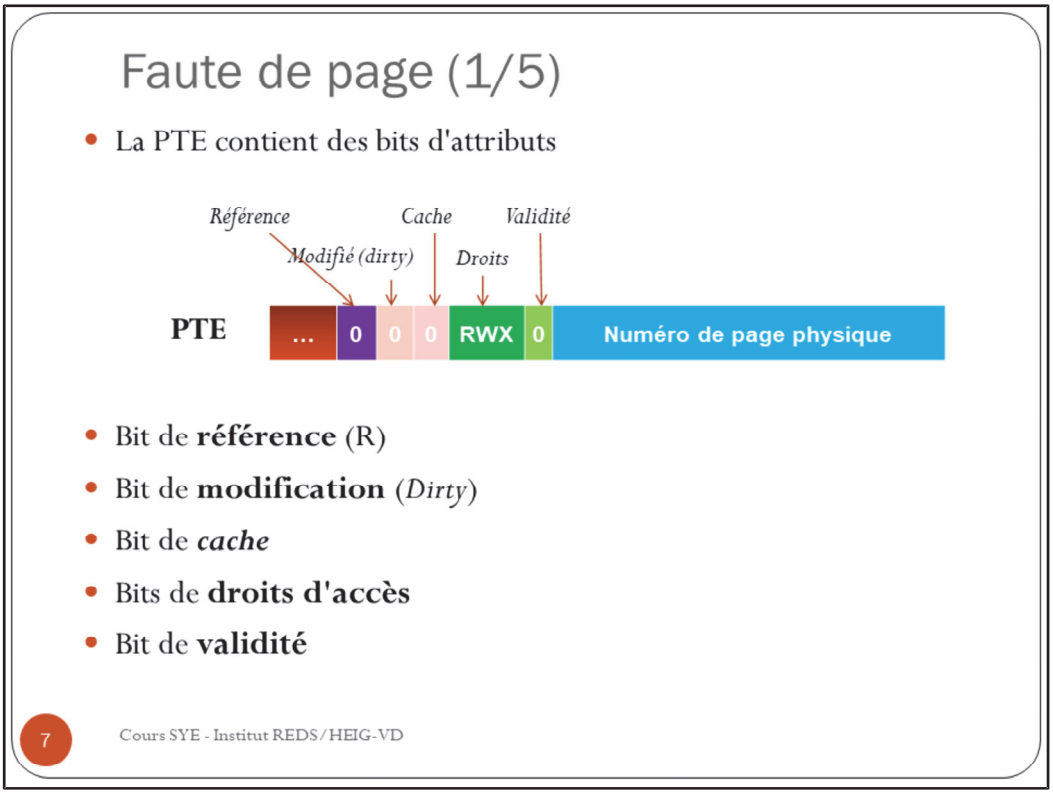
Dans ce contexte, on se rappelle aussi de la notion de **fichier mappé** consistant à projeter le contenu d'un fichier en mémoire. La décomposition d'un fichier sous forme de blocs similaires aux pages mémoires prend alors toute son importance.



La pagination à la demande et les mouvements de pages entre mémoire principale (RAM) et mémoire secondaire (disque dur) nécessitent de pouvoir détecter si une page est présente ou non en RAM. Lorsque le processeur effectue une référence mémoire, l'adresse virtuelle est traduite en adresse physique par la MMU (via les tables de page) et la donnée est ensuite accédée. Si la MMU ne peut accéder à la page physique (y compris d'ailleurs une page contenant une table de second niveau), celle-ci doit immédiatement informer le processeur que la page n'est **pas** présente en mémoire. C'est ce que l'on appelle la ***faute de page*** (ou encore défaut de page). Dans ce cas de figure la MMU envoie une interruption matérielle spécifique au CPU, permettant au noyau de l'OS de faire le nécessaire afin de récupérer la page manquante. Le gestionnaire mémoire effectuera donc une remontée de la page manquante en RAM.

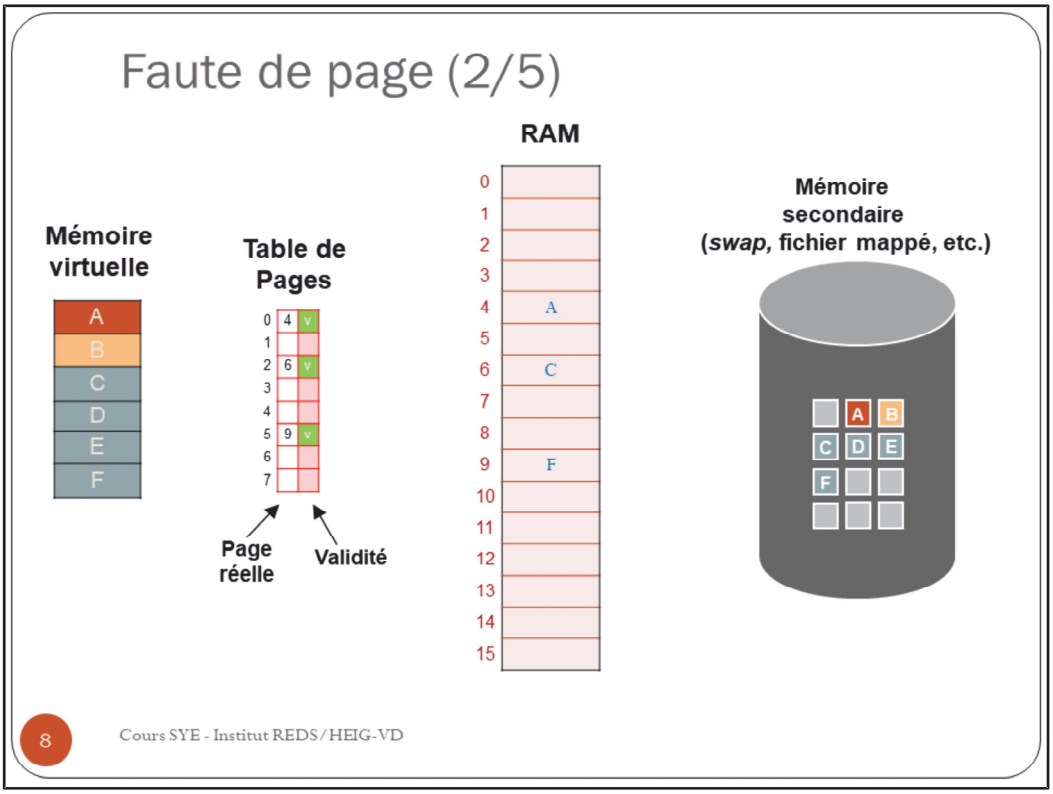
Lors de cette remontée (chargement) de la page en mémoire physique, deux situations peuvent se présenter: soit il reste des pages disponibles en RAM (mémoire vide), soit il n'y en a plus. Dans ce cas, il faut impérativement sélectionner une page physique, et l'éjecter de la mémoire en vue de libérer la place pour la page à charger. La sélection de la page à évincer de la mémoire se base sur un algorithme de recherche appelé algorithme de remplacement de page. De tels algorithmes sont présentés plus loin dans ce chapitre.

Nous allons maintenant examiner en détails le phénomène de *faute de page*.



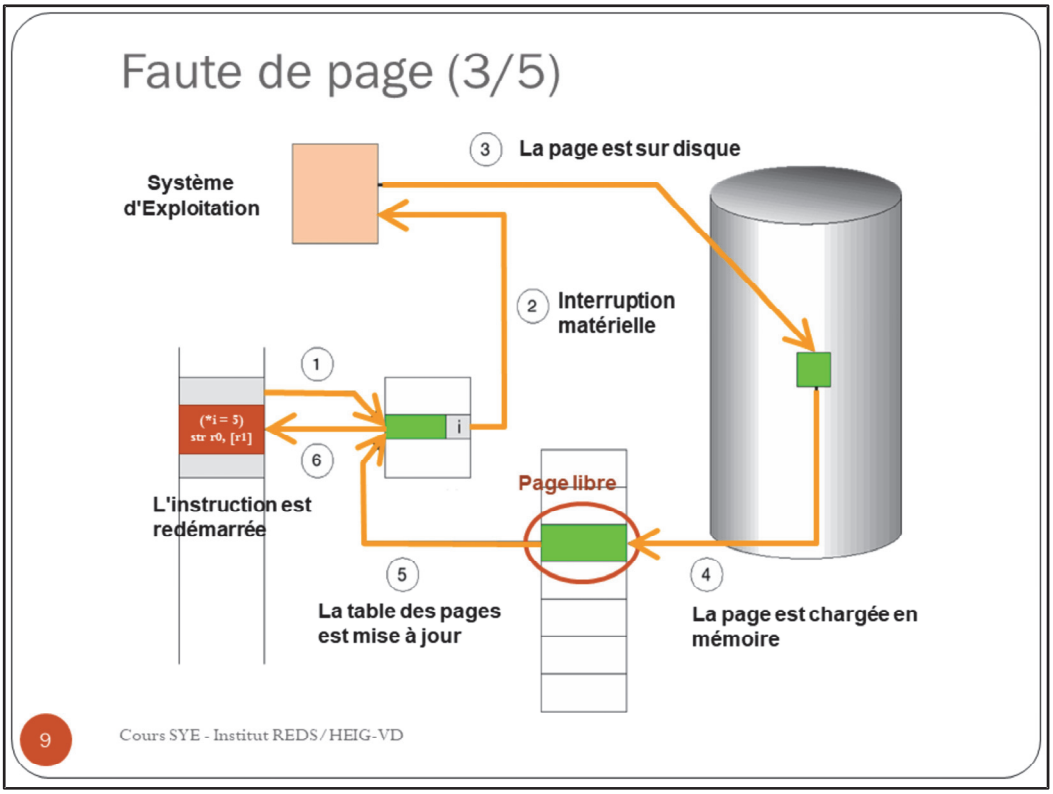
Pour que la MMU puisse détecter rapidement si une page est présente en RAM ou non, elle se servira de la PTE qu'elle récupère via le numéro de page virtuelle (ou l'index correspondant dans un système multi-niveau). Ces bits permettent de donner des informations relatives à la page physique référencée. Entre autre, le bit de **validité** (ou bit de *présence*) permet de savoir si la page se trouve en RAM ou non. Ainsi, si ce bit est à 0 (page absente), la MMU lève de suite l'interruption matérielle.
Les autres bits d'attribut sont les suivants:

- Le bit de "référence", qui est initialement mis à 0 au chargement, et à 1 dès que la **page est référencée**. En principe, ce bit sera donc rapidement à 1, mais il peut être remis à 0 par l'algorithme de remplacement de page et utilisé à des fins statistiques comme nous le verrons plus tard.
- Le bit "*modifié (dirty)*", qui est mis à 1 dès que la page **subit une modification**. Ainsi, si la page doit être évincée, elle doit être mis en *swap*, puisqu'elle ne correspond plus avec la page d'origine.
- Le bit "*cache*", qui indique si la page est **présente dans un *cache*** (processeur ou *cache virtuelle* définie en RAM).
- Les bits "RWX", qui indiquent les **droits d'accès de la page**. Grâce à ces bits, on peut protéger l'accès à certaines pages, comme les pages contenant le code. Ces dernières ne peuvent être modifiées, mais seulement en lecture seule et exécutables. Les pages de la pile ne devraient pas être exécutables (failles de sécurité exploitables le cas contraire).

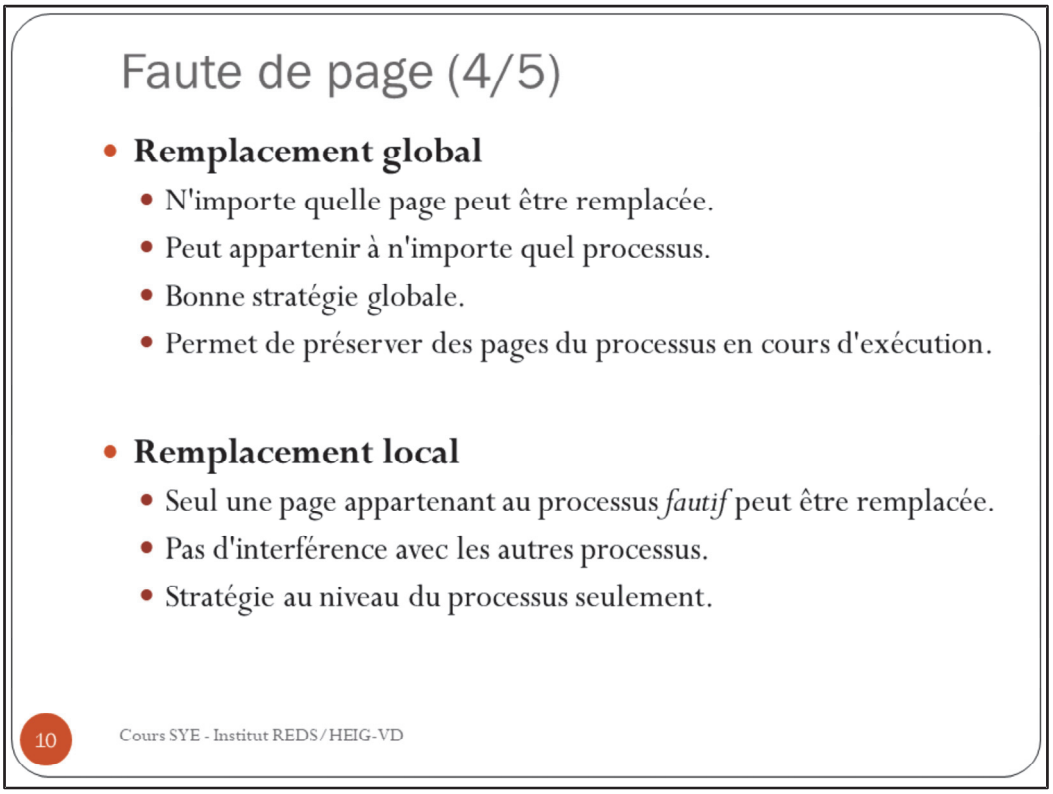


Lors d'une faute de page, le gestionnaire mémoire doit être capable de retrouver la page manquante afin de pouvoir la charger en mémoire. Le contenu associé à cette page peut provenir d'un fichier (exécutable, fichier de données, etc.) stocké sur disque, ou encore dans un espace mémoire de type *swap* (fichier ou partition *swap*).

Le lien entre la page référencée et son contenu est établi lors du chargement d'un fichier – pouvant être réalisé à l'aide d'un mappage de fichier par exemple – ou encore lors du transfert de la mémoire vers la zone de type *swap*; dans ce dernier cas, c'est le gestionnaire de *swap* en relation avec le système de fichiers qui maintiendra le lien.



Lors d'une tentative d'accès à une page non-présente en RAM, le scénario suivant se produit: en (1), l'instruction contenant la référence mémoire est exécutée. L'adresse est traduite par la MMU, la PTE récupérée, le bit de validité est consulté. Celui-ci indique que la page est absente: l'interruption de *faute de page* est levée (2). La routine de service correspondante exécutera la fonction qui permettra d'aller récupérer (3) la page manquante (peut-être localisé dans le *swap*), puis la chargera en mémoire en invoquant l'allocateur mémoire (4). S'il n'y a plus de place disponible (donc plus de page libre), l'allocateur devra sélectionner une page à évincer (en utilisant l'algorithme de remplacement approprié). Enfin, la table de pages (ou les tables) devra être mis à jour (5) afin d'associer le bon numéro de page virtuel avec le numéro de page physique. En (6), l'instruction qui a causé la faute de page est ré-exécutée et l'accès mémoire effectué cette fois-ci correctement.

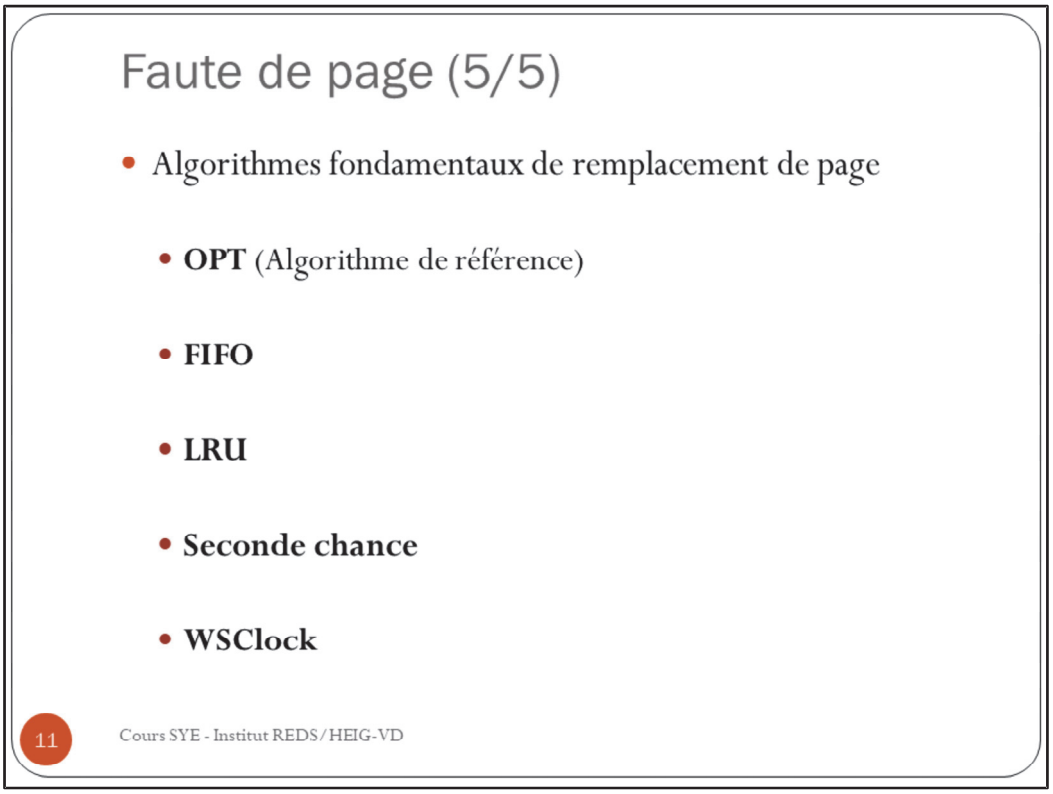


Le remplacement d'une page (dans la mémoire physique) nécessite de choisir une page "***victime***" qui sera (momentanément) **déchargée** dans la mémoire *swap,* s'il est pour autant nécessaire de faire un tel transfert. Bien qu'une page physique puisse être partagé par plusieurs processus, elle *appartient* à un processus. La page est libérée de la mémoire dès que le processus se termine.

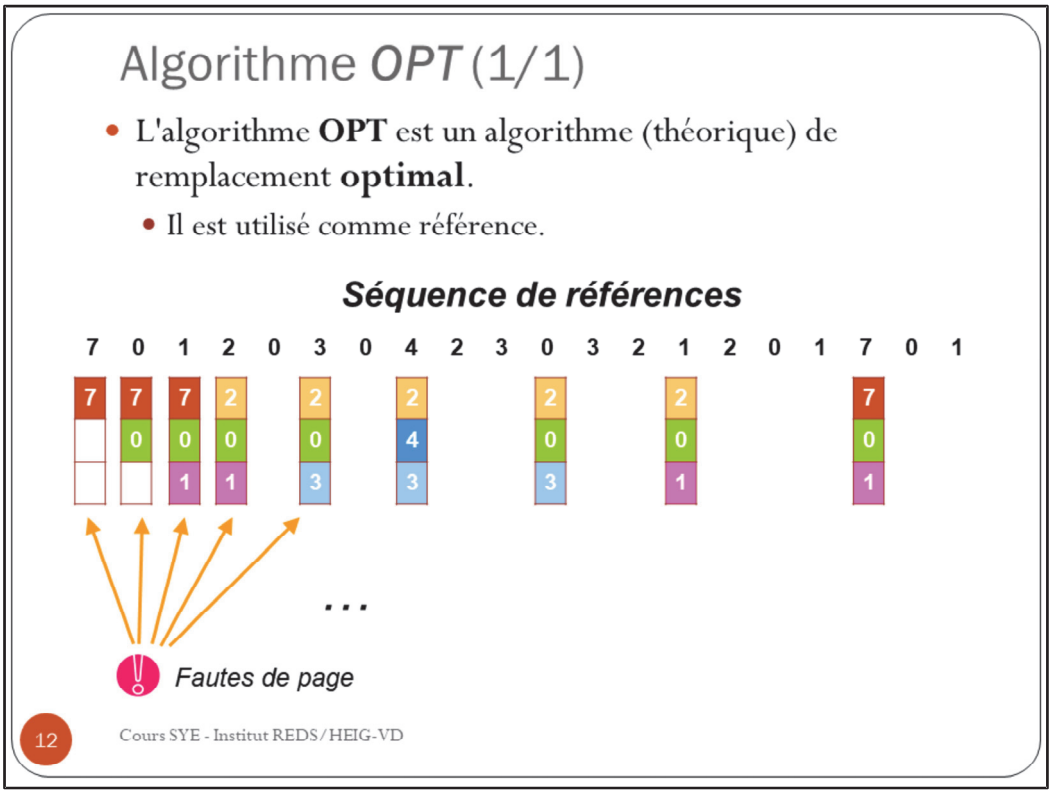
Lors du remplacement d'une page, il est donc possible de sélectionner une page physique existante appartenant soit au processus *fautif* (qui a déclenché la faute de page). Dans ce cas on parle de **remplacement local**. Soit la page sélectionnée appartient à tout autre processus (par exemple dans l'état *ready*): c'est le **remplacement global**.

L'une ou l'autre stratégie peut avoir un fort impact sur la disponibilité des pages d'un processus, et donc influencer le nombre de fautes de page potentielles. En principe, il convient de réduire le nombre de fautes de page *au minimum*, étant donné les transferts disque-mémoire que celles-ci peuvent engendrer, et donc ralentir le système.

Nous allons étudier maintenant différents algorithmes de remplacement de pages, qui peuvent être utilisés, pour certains, avec les deux stratégies de remplacement, et, pour d'autres, seulement avec l'une ou l'autre stratégie.



L'analyse des algorithmes de remplacement de page s'effectue en considérant une séquence de références de page (virtuelle), et en examinant une mémoire physique, c-à-d un ensemble prédéfini de pages physiques. Ces pages sont bien entendu libres au début de l'analyse. Le chargement des pages s'effectue *à la demande (on-demand paging*).

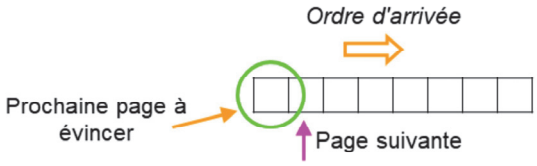


Les premières références provoquent forcément des fautes de pages, puisqu'aucune page n'est encore présente en mémoire. En revanche, les pages étant libres, il n'y a pas lieu d'effectuer un remplacement; les pages sont chargées dans la mémoire physique, dans l'ordre des pages disponibles.

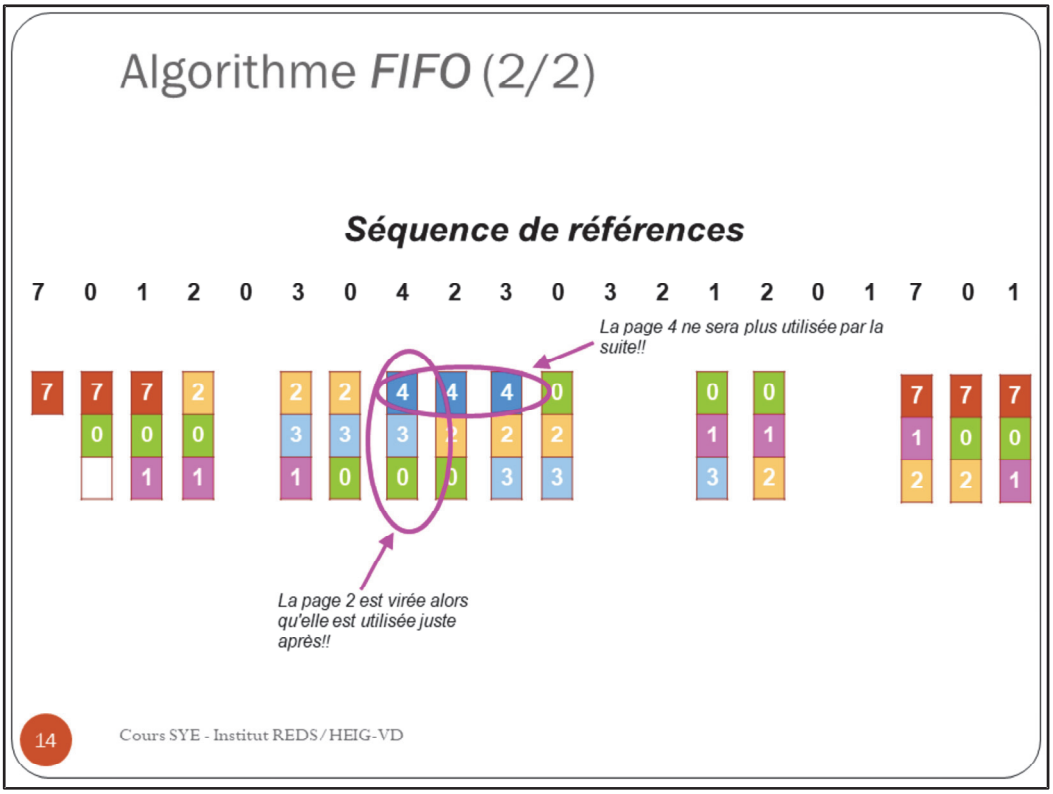
L'algorithme OPT est une algorithme optimal, mais essentiellement théorique. L'analyse des références dans le futur rend son implémentation difficile, voire impossible. Il sera utilisé pour comparer la performance des autres algorithmes.

Algorithme *FIFO* (1/2)

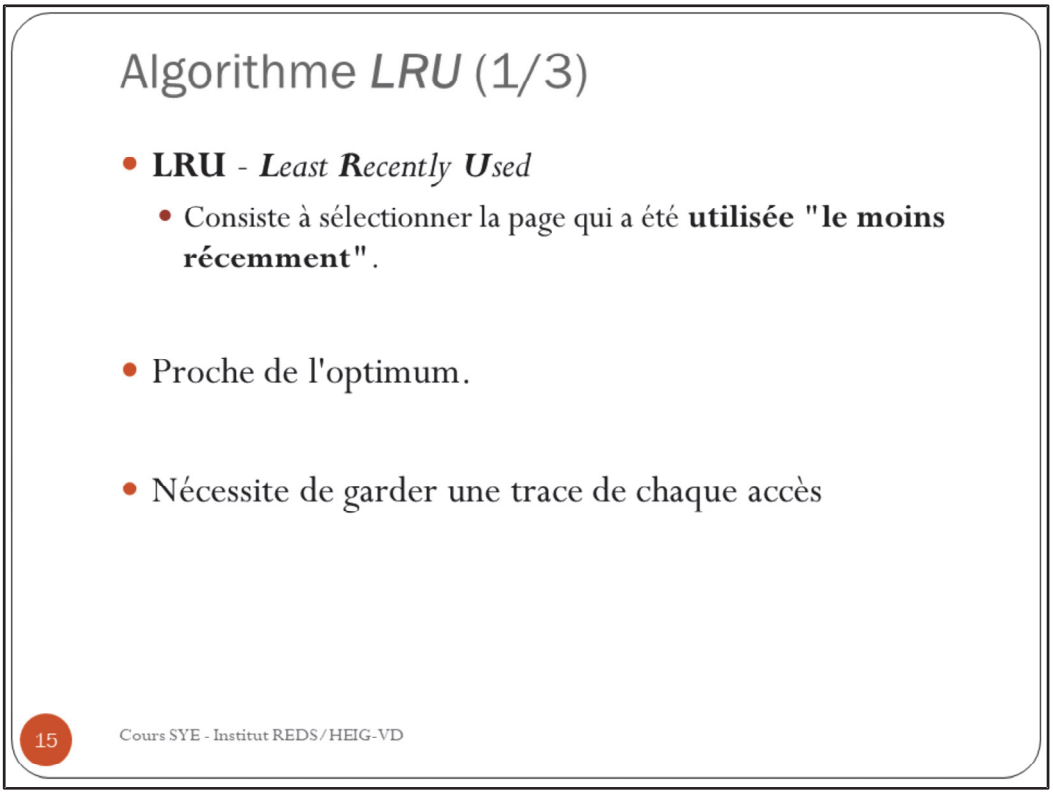
- L'algorithme **FIFO** consiste à remplacer les pages **selon leur ordre d'arrivée**.
  - **Facile** à implémenter et rapide
  - Ne tient pas compte de **l'utilisation récente et future** des pages.



L'algorithme FIFO est facile à implémenter, mais pas optimale car il y a un risque d'évincer des pages qui sont fréquemment utilisées. Par conséquent, il peut amener à de fréquentes fautes de page.

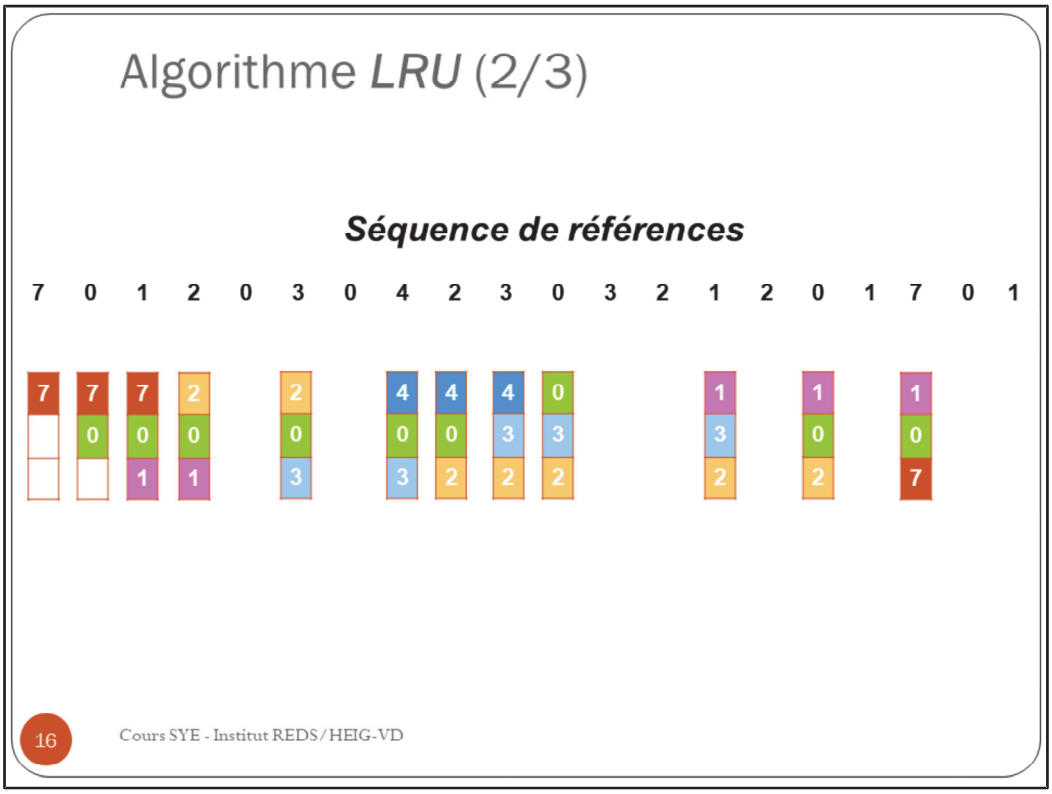


L'exemple ci-dessus montre bien l'inefficacité de l'algorithme *FIFO* : ne tenant pas compte d'une statistique d'utilisation des pages, celles en cours d'utilisation peuvent être virées, pour être rechargées peu de temps après.



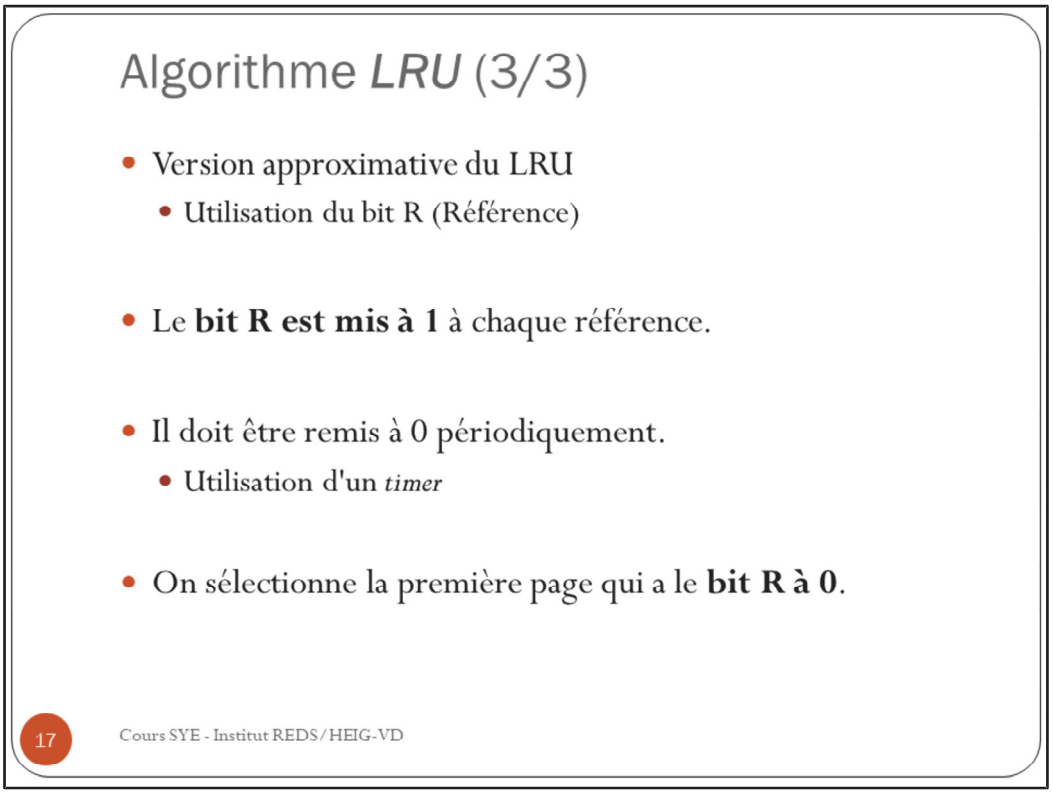
L'algorithme LRU consiste à sélectionner la page référencée le plus loin *dans le passé*. Cette approche se base complètement sur la notion de localité de référence étudiée précédemment. En effet, si une page (ou un ensemble de pages) vient juste d'être référencée, il se peut, avec une probabilité très élevée, qu'elle le sera à nouveau dans les toutes prochaines références. Dès lors, il vaut mieux la préserver afin d'éviter de provoquer une faute de page. On va donc chercher la page qui n'a plus été utilisée depuis longtemps.

Bien que proche de l'optimal, la mise en œuvre de cet algorithme reste difficile, car il faut garder une trace temporelle de la référence de chaque page. Au niveau de la PTE, par exemple, il n'y a pas assez de place pour stocker ce type d'information. Il faudra donc utiliser une structure plus sophistiquée afin de stocker cet estampillage.



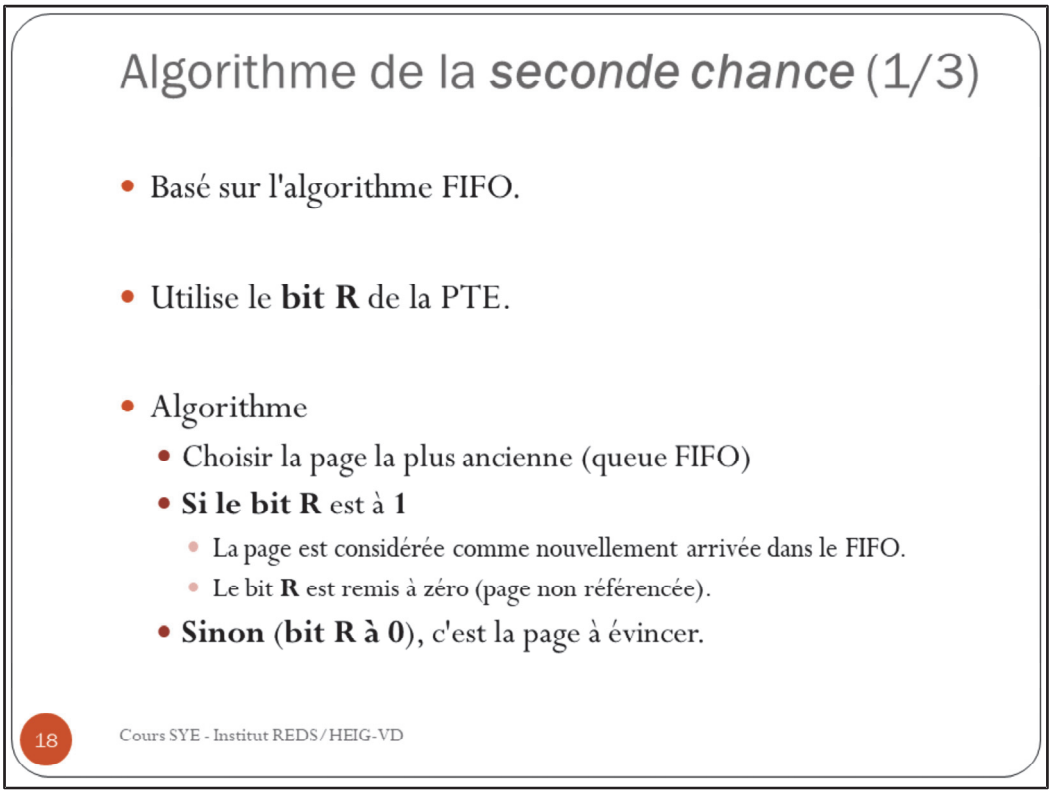
Dans l'étude du *LRU*, la séquence de références ne peut être établie de manière aléatoire. Elle doit correspondre à une certaine réalité d'utilisation, et tenir compte de ce fait du principe de localité de référence.

Une séquence aléatoire conduit inexorablement à un taux élevé de fautes de page, quelque soit l'algorithme de remplacement.



Finalement, il reste un moyen très simple d'implémenter une version approximative du LRU: c'est en utilisant **le bit référence** (bit R) de la PTE. Ce bit est automatiquement mis à 1 par la MMU lors de la référence à la page, que ce soit en lecture ou en écriture.

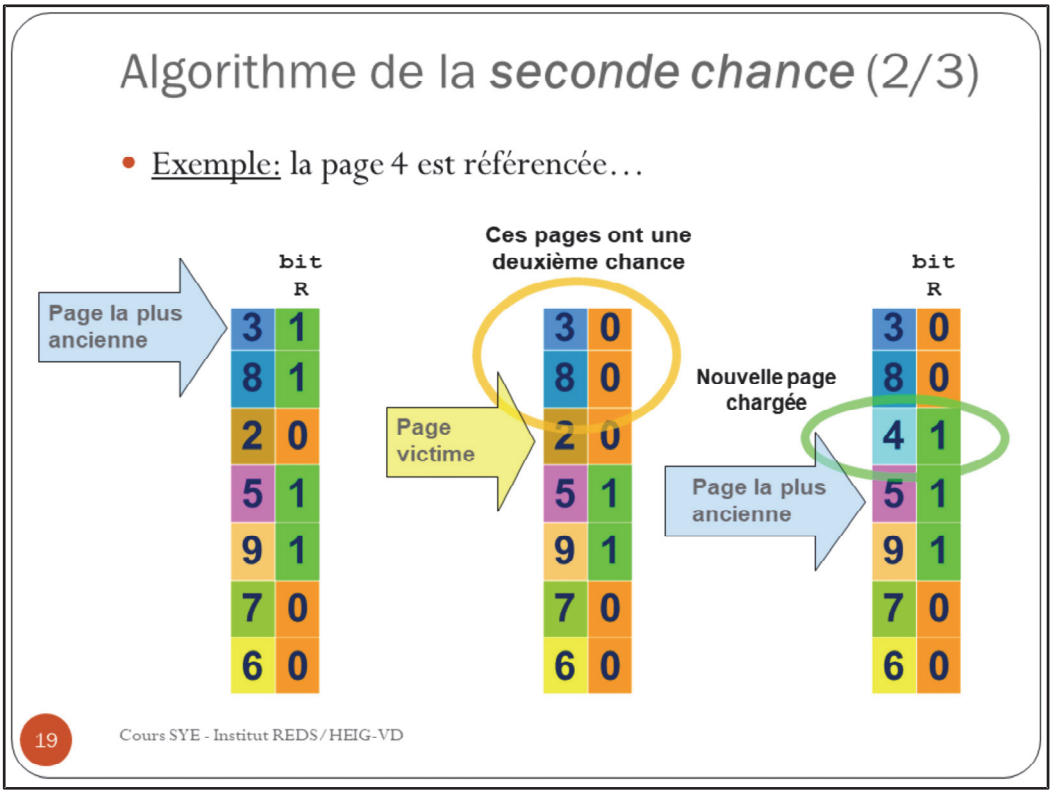
Si l'on dispose d'un *timer* périodique qui réinitialise le bit R de toutes les PTEs, alors l'ensemble des PTEs qui ont le bit R à 1 à un certain moment dans l'intervalle de deux réinitialisations représente l'ensemble des pages à préserver. Les pages dont le bit R est à 0 n'auront pas été référencées et peuvent donc être utiliser comme pages candidates au remplacement.



L'algorithme de la seconde chance se base sur l'algorithme *FIFO*, en examinant le bit de référence de la *PTE*. Il consiste à préserver les pages qui ont été référencées, en cherchant toujours à évincer une page avec le bit de référence à zéro.

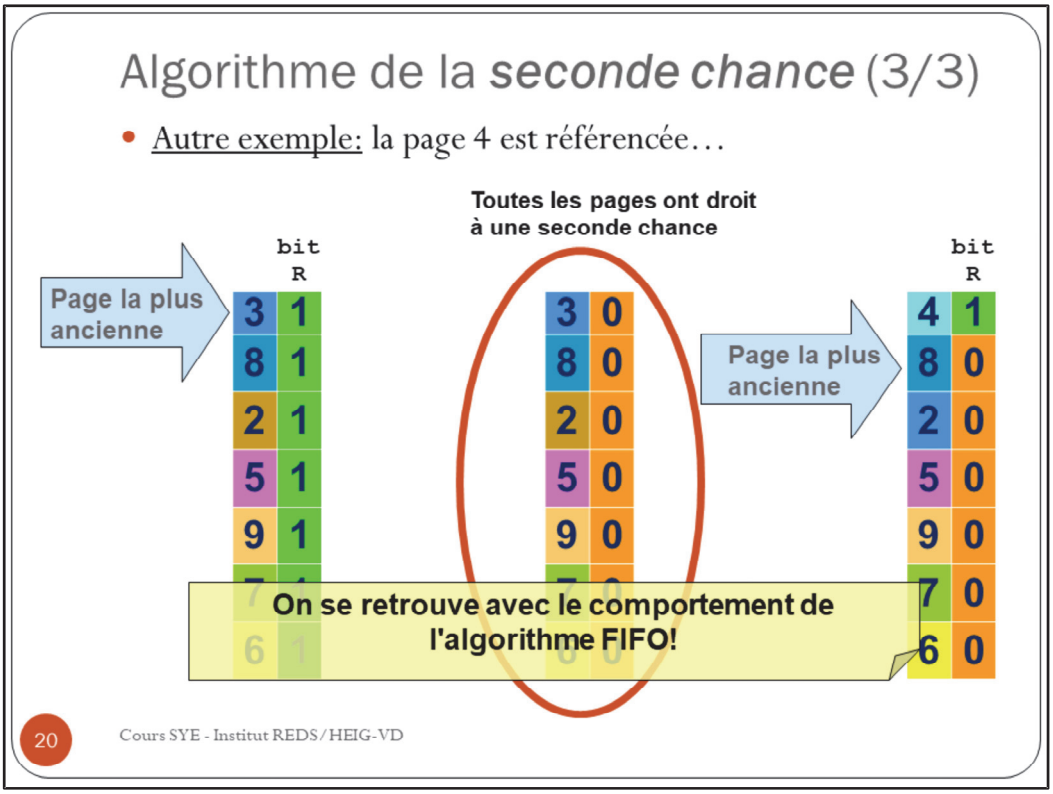
Le mécanisme consiste à parcourir l'ensemble des pages physiques de manière circulaire, jusqu'à trouver une page avec le bit R à zéro. Tant qu'une page a été référencée (son bit R est à un), **le bit de référence est alors remis à zéro** (on lui donne une seconde chance)**, mais la page reste en mémoire**, et on inspecte la page suivante. En revanche, la page examinée est "virtuellement" replacée dans le *FIFO*, comme si elle venait d'être chargée en mémoire.

Ainsi, selon cet algorithme, c'est lors d'un remplacement de page que les bits de référence peuvent être réinitialisés.

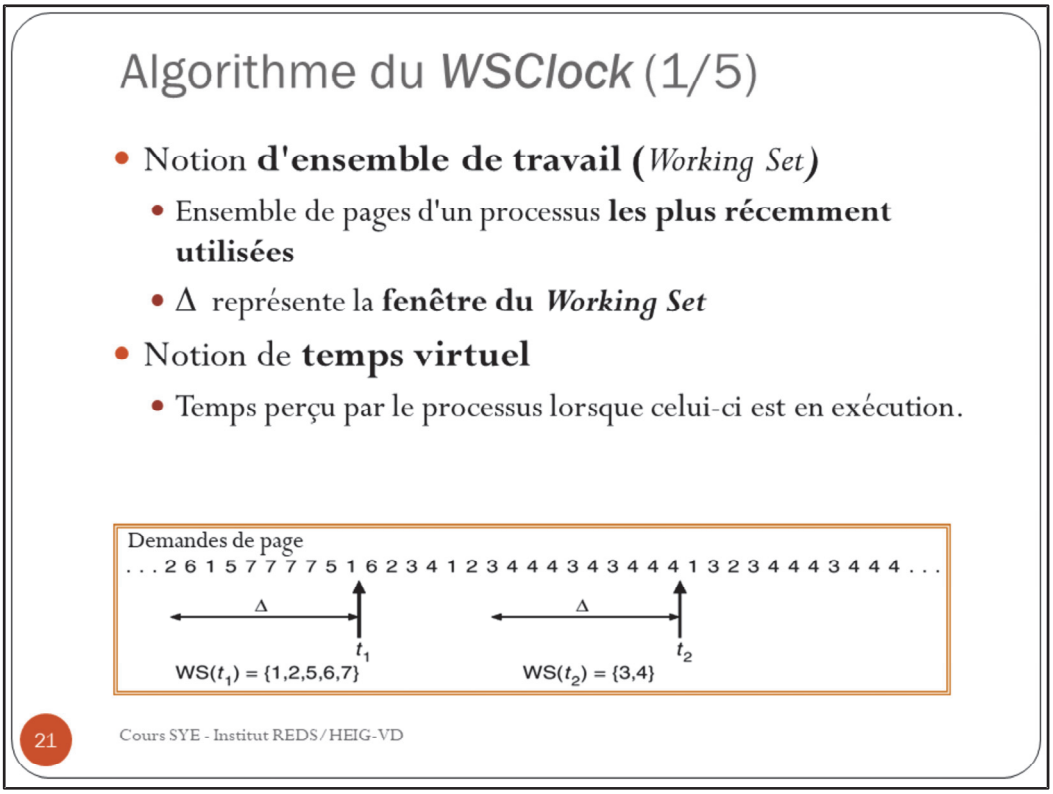


L'algorithme consiste à inspecter l'ensemble des pages physiques de manière circulaire. On utilise pour cela un pointeur qui maintient la position courante. Dès qu'une page est remplacée, le pointeur est maintenu à la position suivante, vers la page qui sera examinée lors du prochain remplacement.

Lorsque l'on a parcouru toute la mémoire, le pointeur reboucle sur le début de celle-ci. Cette façon de procéder conduit à appeler aussi cet algorithme "algorithme de l'horloge" en référence au parcours circulaire du pointeur.



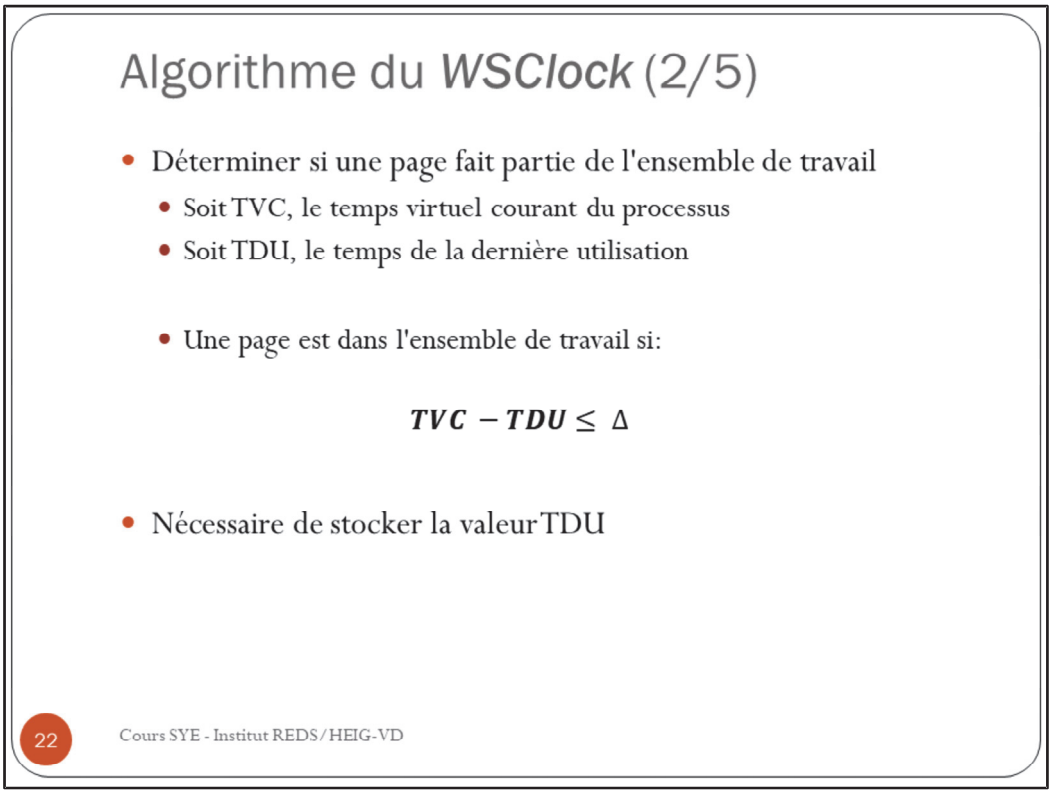
Lorsque toutes les pages ont leur bit de référence à un (ce qui arrive en premier lorsque la mémoire est pleine), l'algorithme dégénère en un algorithme FIFO lors du premier remplacement : en effet, toutes les pages voient leur bit de référence réinitialisé, et par circularité, la page remplacée sera la suivante.



L'algorithme du *WSClock* est un algorithme fondamental sur lequel se base aujourd'hui la plupart des algorithmes de remplacement de page. Il s'agit d'un algorithme utilisant **exclusivement un remplacement local**, i.e. seul les pages appartenant au processus fautif peuvent être évincées. Afin de déterminer la page à remplacer, l'algorithme utilise un *ensemble de travail (working set*) qui contient l'ensemble des pages les plus *récemment* utilisées sur une fenêtre d'observation d'une certaine taille (Δ). Cette taille peut être exprimée sous forme d'une durée ou simplement d'un nombre de page. La taille de cet ensemble de travail varie au fil du temps selon les pages référencées. Si toutes les pages référencées dans la fenêtre d'observation sont différentes, alors la taille sera égale à la largeur de la fenêtre d'observation (Δ). En revanche, si seul une page est référencée durant tout ce temps, la taille de l'ensemble de travail vaudra 1.

L'ensemble de travail est particulièrement intéressant pour faire du *pré-paging*. En effet, il est envisageable de *swapper* sur disque l'ensemble de travail complet si le processus se met en attente pour un long moment par exemple. Lors de sa réactivation, l'ensemble de travail pourra également être rechargé en un seul coup (*pré-paging*).

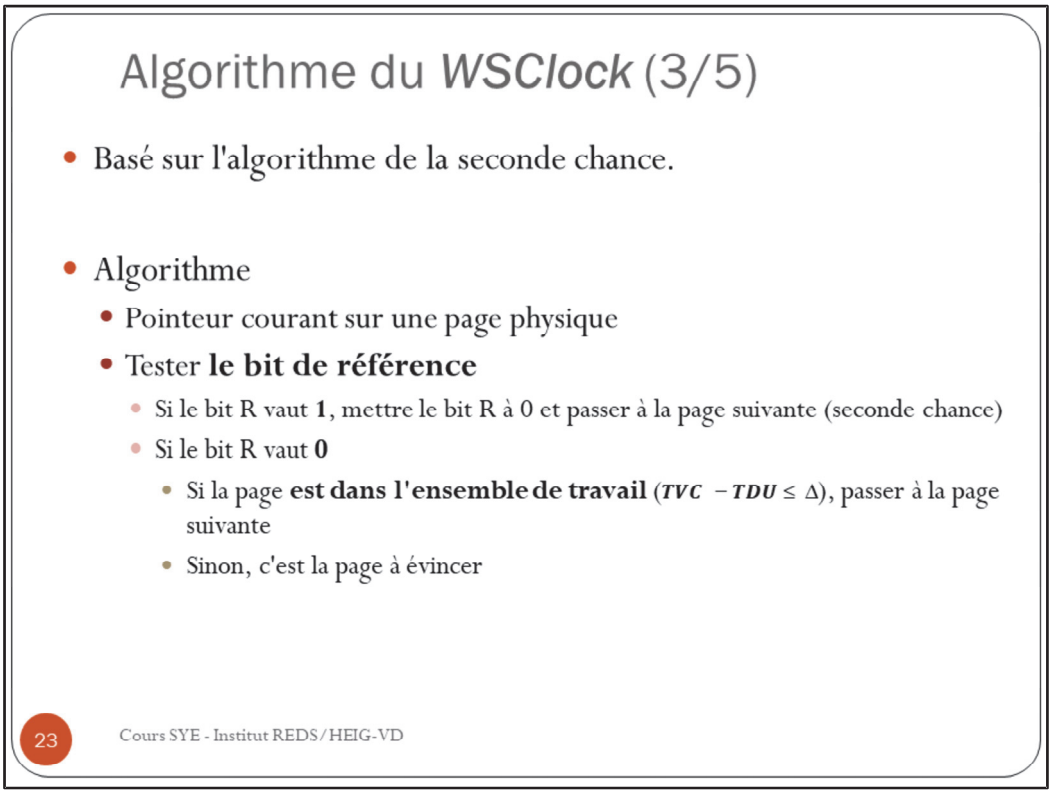
Le temps virtuel d'un processus est basé sur une horloge (virtuelle) interne au processus. Lorsque le processus n'est pas en exécution (*running*), le temps est momentanément suspendu. Le temps virtuel est nécessaire afin de déterminer si une page est présente ou non dans l'ensemble de travail.



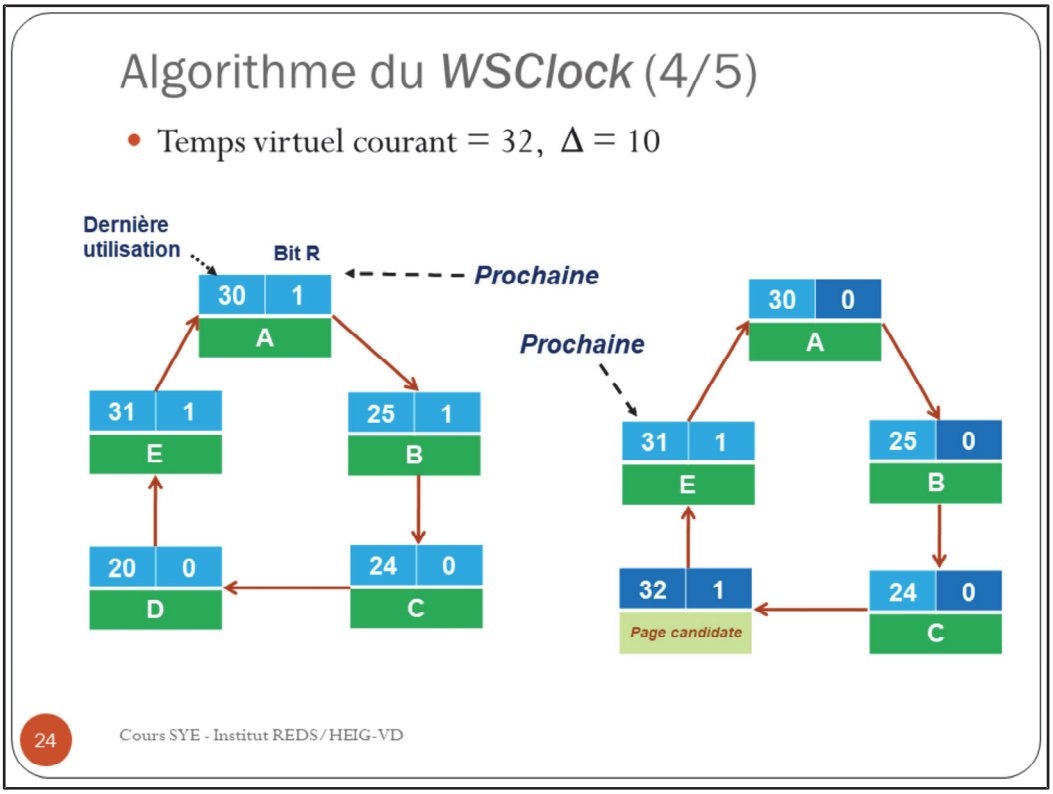
Avec l'algorithme du *Working Set*, le temps de la dernière utilisation est associée à chaque page physique. Cela signifie qu'à chaque accès mémoire, une estampille temporelle devrait être stockée dans la *PTE* correspondante (nous avions déjà un cas similaire avec l'algorithme *LRU*). Cela est quasiment impossible sans un support matériel adéquat.

C'est pourquoi, la mise à jour du *TDU* peut être effectuée de manière périodique (comme dans le cas du *LRU* approximé) pour les pages ayant le bit de référence à un. Dans ce cas, la valeur *TDU* est remplacée par le temps virtuel courant du processus (*TVC*). Il s'agit d'une approximation, mais différentes expérimentations ont montré qu'elle pouvait être satisfaisante.

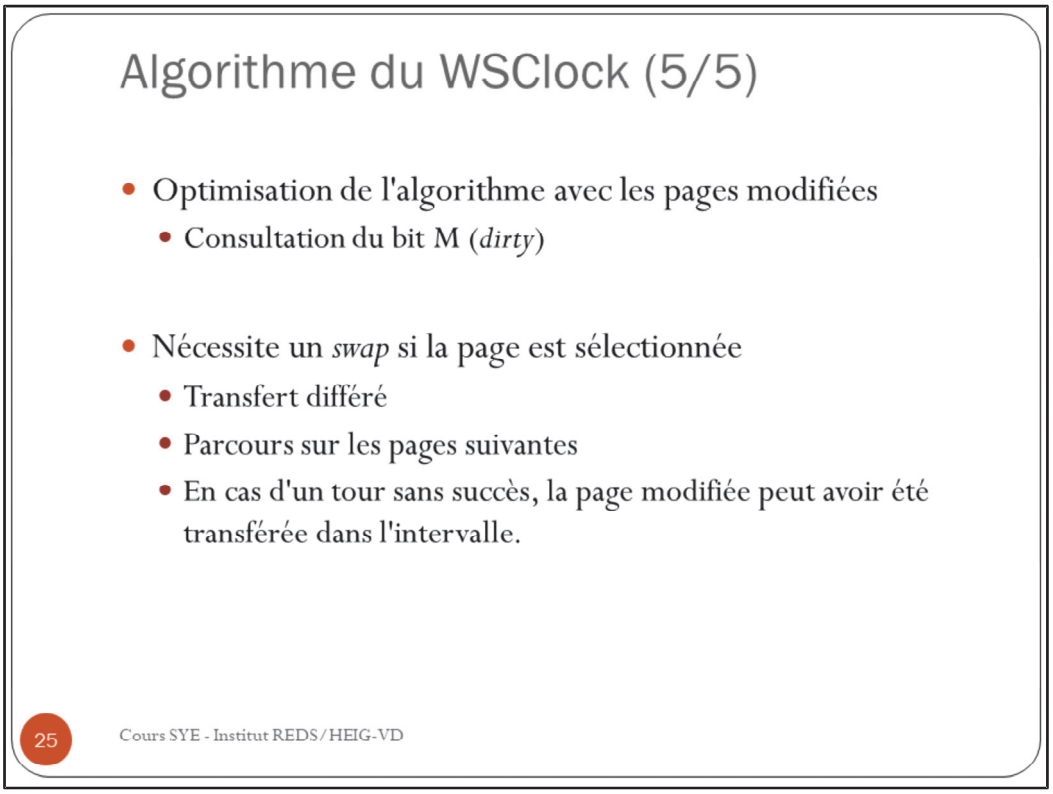
Comme nous le verrons dans la description de l'algorithme, les pages ayant le bit R à 0 – c'est-à-dire celles ayant "reçu une seconde chance" – ne verront pas leur *TDU* réactualisé, et sortiront progressivement de l'ensemble de travail.



L'algorithme du *WSClock* est relativement simple. Basé sur l'algorithme de la seconde chance, il commence par tester le bit de référence de la PTE, afin de déterminer, s'il y a lieu, de donner une seconde chance à la page ou non. Si le bit R vaut 0, il suffit de tester si la page appartient à l'ensemble de travail du processus courant ou non. Ce test est effectué à l'aide du temps virtuel courant et du temps de la dernière utilisation, information devant être associée à toutes les pages physiques. S'il n'existe aucune page externe à l'ensemble de travail, l'algorithme sélectionnera une page de l'ensemble de travail.



Dans l'exemple ci-dessus, le temps virtuel courant est égal à 32. On suppose qu'à ce moment, un remplacement de page est nécessaire. Le page référencée par le pointeur courant est examinée : le bit R est à 1, on donne donc une seconde chance à cette page, et on examine la page suivante; c'est l'algorithme de la seconde chance. Les pages suivantes sont examinées et le bit R mis à jour, jusqu'à ce qu'une page dont le bit R est à 0 est trouvée. A ce moment, on détermine si cette page fait partie de l'ensemble de travail (ici, la valeur *TDU* de la page contenant "C" est égale à 24. Par conséquent, elle fait partie de l'ensemble de travail (32 – 24 = 8, qui est inférieure à 10, taille de la fenêtre d'observation). On examine donc la page suivante. La page "D" ayant un *TDU* de 20, elle est clairement hors de l'ensemble de travail (32 – 20 = 12 > 10), et peut donc être remplacée.



D'une manière générale, les pages sélectionnées pour le remplacement doivent être *swappées* sur la mémoire secondaire, en particulier si elles ont subies des modifications. Or, le transfert prenant beaucoup de temps, une optimisation fréquente consiste à différer le transfert sur disque et de continuer le parcours de la mémoire à la recherche d'une autre page candidate. Le transfert peut s'effectuer en arrière-plan, et en cas d'échec de l'algorithme à trouver une autre page candidate, le temps de recherche, durant lequel des changements de contexte ont pu intervenir peut avoir donner la possibilité au gestionnaire mémoire d'avoir "eu le temps" de transférer la page sur disque. Dès lors, elle peut être sélectionnée de manière définitive. Si le gestionnaire n'a pas eu le temps, l'algorithme peut décider de re-parcourir une nouvelle fois les autres pages pour trouver une page candidate.

Références

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