Mémoire

LIFASR5 - Systèmes d'exploitation

L. Gonnord, N. Louvet

22 mars 2022

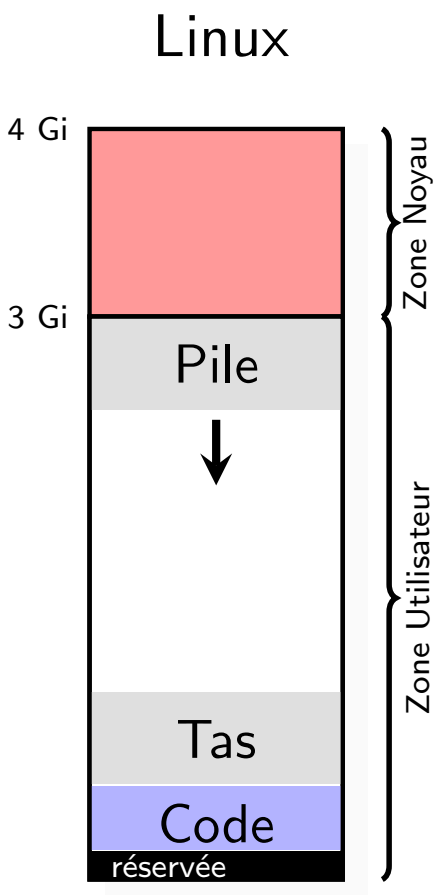
Mémoire utilisée par un processus

Pour un processus, la mémoire permet de :

- stocker le code du programme,
- stocker le code des fonctions partagées,
- stocker des données globales sur le tas,
- stocker les variables locales sur la pile.

En particulier, vous savez que :

- à chaque appel d'une fonction, la valeur de retour, les paramètres et les variables locales se voient réserver un emplacement sur la pile,
- il est possible de réserver de la mémoire sur le tas avec `new`, et de la libérer avec `delete`.



Si le compilateur attribue une adresse fixe à chaque variable ou fonction, alors un processus doit toujours être rangé au même endroit en mémoire !

Au âges farouches, un ordinateur faisait tourner un seul programme, et tout allait pour le mieux ; mais ça fait longtemps que ça n'est plus le cas :

- on veut utiliser plusieurs (beaucoup de) processus sur une machine,
- les différents processus doivent cohabiter en mémoire,
- le compilateur ne peut pas prévoir comment partager la mémoire !

D'où la problématique : comment faire cohabiter en même temps de nombreux processus dans un volume fini de mémoire ?

Historiquement, il y a plusieurs façons de régler le problème : translation d'adresse, notion de segment, . . . , mémoire virtuelle.

Organisation générale

Idée générale :

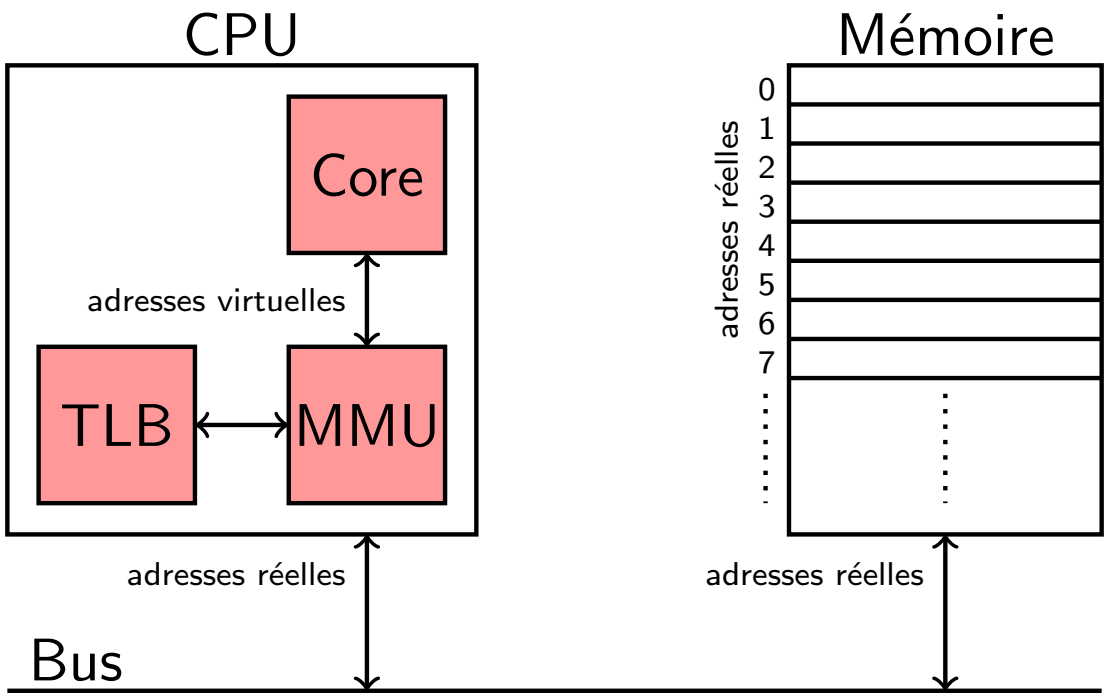
- Le processeur travaille avec des adresses virtuelles, sans rapport avec les adresses physiques de la RAM.
- À chaque accès (lecture ou écriture) à la mémoire, l'adresse physique est calculée d'après l'adresse virtuelle.

Un matériel est utilisé pour faire efficacement les conversions d'adresses virtuelles en adresses réelles : la MMU (Memory Management Unit).

Tous les processus disposent ainsi du même espace d'adressage, et les adresses produites par le compilateur sont toutes dans cet espace.

La capacité d'adressage du processeur est le nombre total d'adresses que l'on peut utiliser en mémoire physique. On suppose pour l'instant qu'elle est plus grande que l'espace d'adressage.

Le processeur ne manipule que des adresses virtuelles, mais seules des adresses réelles circulent au dehors ; la MMU sert d'interface :

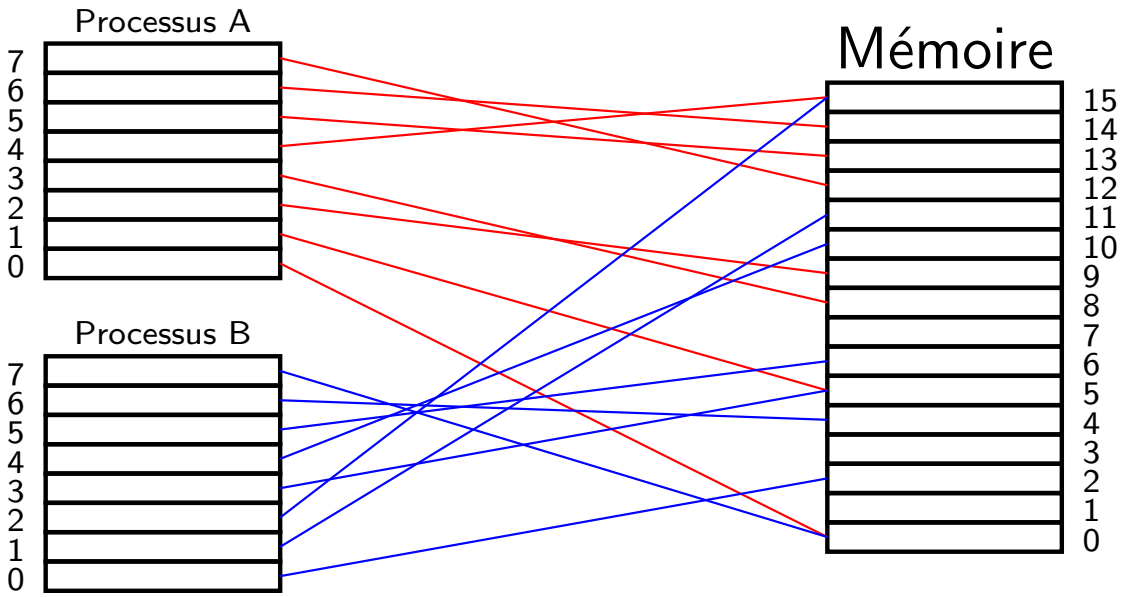


Comme la conversion peut présenter une latence importante, la MMU utilise un petit cache, le TLB (Translation Lookaside Buffer), pour conserver temporairement les conversions des dernières adresses.

Sachant que tous les processus ont le même espace d'adressage, il faut

- que les adresses de chaque processus soient bien mappées en mémoire,
- que chaque adresse en mémoire soit attribuée à au plus un processus,
- que l'accès à la mémoire se fasse avec un surcoût raisonnable.

Voici une tentative de solution où « ça commence mal » :



Formalisons un peu...

On note $\mathcal{P}$ l'ensemble des processus, $\mathcal{V}$ leur espace d'adressage, et $\mathcal{R}$ l'ensemble des adresses réelles de la mémoire.

Il nous faudrait une fonction $F : \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{R}$ telle que :

- F soit totale, c.-à-d. définie sur tout $\mathcal{P} \times \mathcal{V}$

  $\hookrightarrow$ toutes les adresses de tous les processus doivent avoir une place en mémoire.

- F soit injective, i.e., pour toute adresse réelle $a_r \in \mathcal{R}$ il existe au plus un processus p et une adresse virtuelle a_v telle que $F(p, a_v) = a_r$.

  $\hookrightarrow$ les processus ne doivent pas partager d'adresses réelles entre-eux.

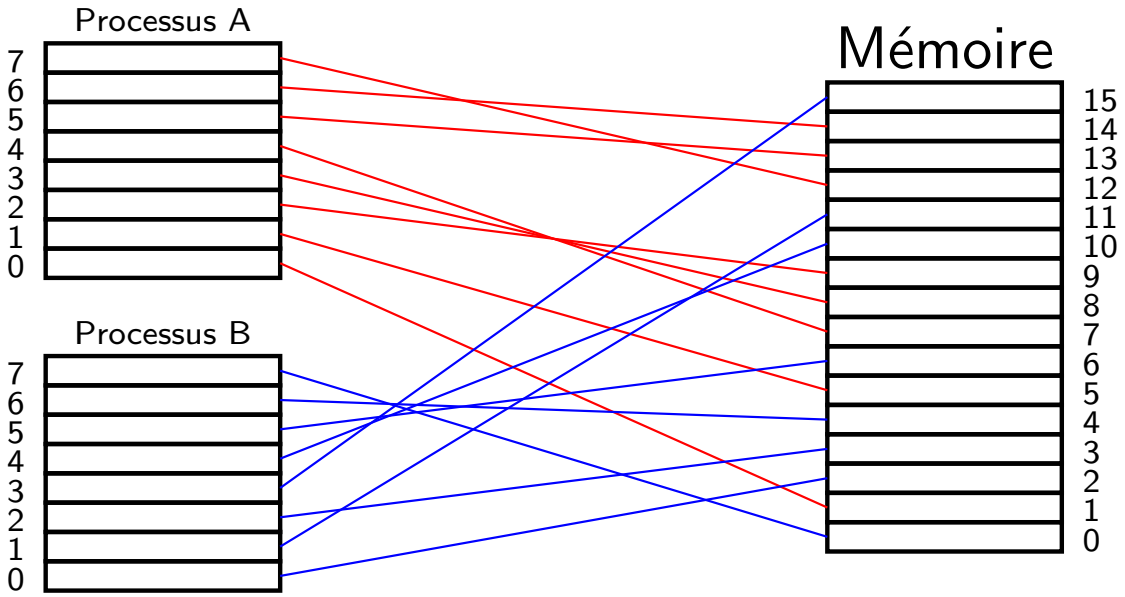
Il existe beaucoup (combien ?) de fonctions vérifiant les conditions précédentes, et ça n'est pas très difficile d'imaginer des solutions.

Ce qui est plus difficile, c'est d'identifier une fonction F :

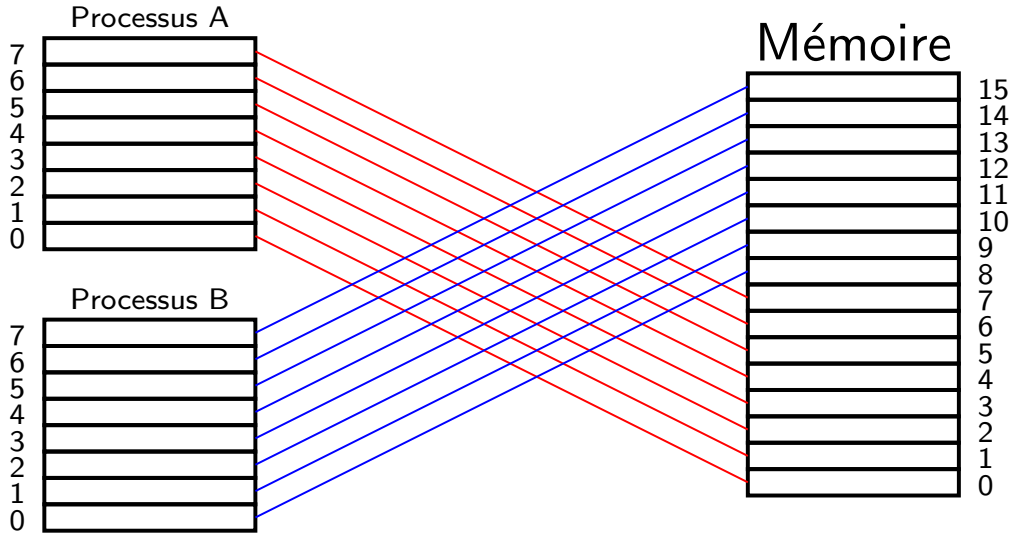
- qui puisse être calculée efficacement (en temps et en mémoire),
- qui introduise peu de surcoût lors des accès à la mémoire.

Pour que F introduise peu de surcoût lors des accès à la mémoire, elle doit respecter le principe de localité spatiale : un processus tend à utiliser des données proches de données qu'il a déjà utilisées.

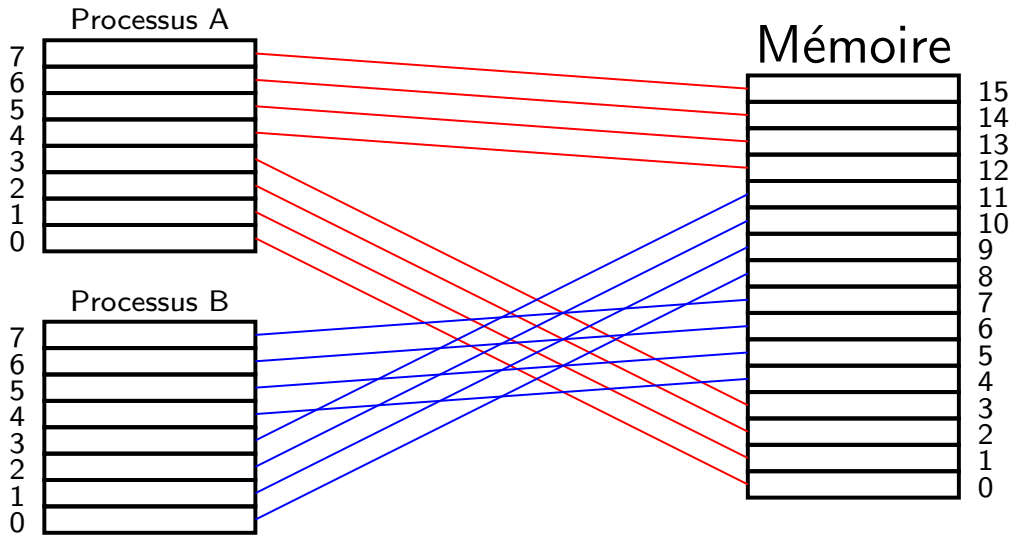
Un exemple de fonction F, totale et injective, mais qui ne respecte pas le principe de localité spatiale :



Un exemple où F respecte idéalement le principe de localité spatiale :

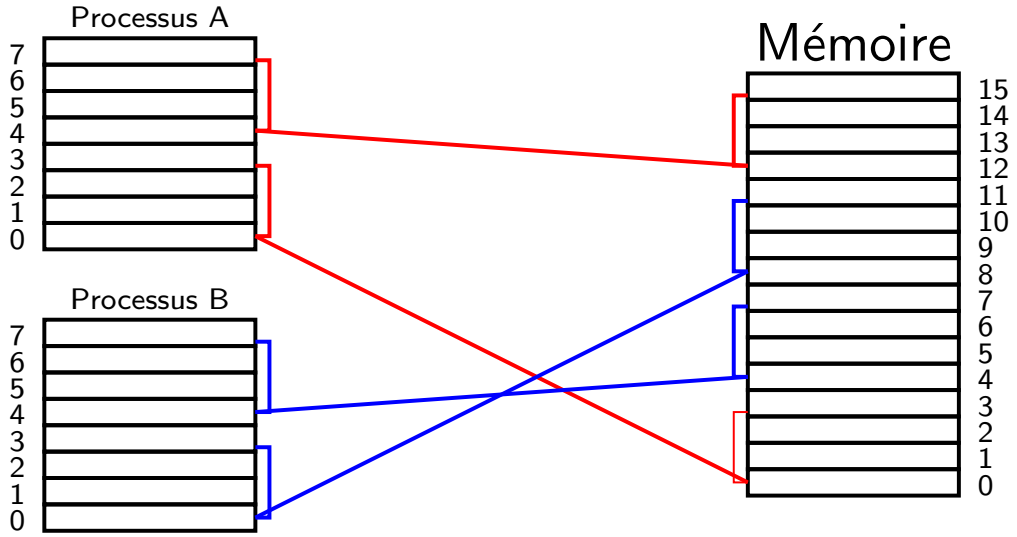


Un compromis dans lequel le principe de localité est bien respecté :



Pour ce qui est du calcul efficace de F, on ne peut pas stocker l'image de chaque adresse virtuelle de chaque processus : par processus, il faudrait stocker autant d'images qu'il y a d'adresses dans l'espace d'adressage !

On découpe donc mémoire virtuelle et réelle en blocs d'adresses consécutives, et on ne stocke la correspondance que pour chaque bloc :



Le décalage par rapport à la 1ère adresse d'un bloc est identique entre mémoire virtuelle et réelle : on calcule ainsi l'ad. réelle d'après la virtuelle.

Le principe sur un exemple

On suppose : ad virtuelles sur 24 bits, ad réelles sur 32 bits.

L'espace d'adressage d'un processus est divisé en blocs de taille fixe s : chaque bloc est appelé une page.

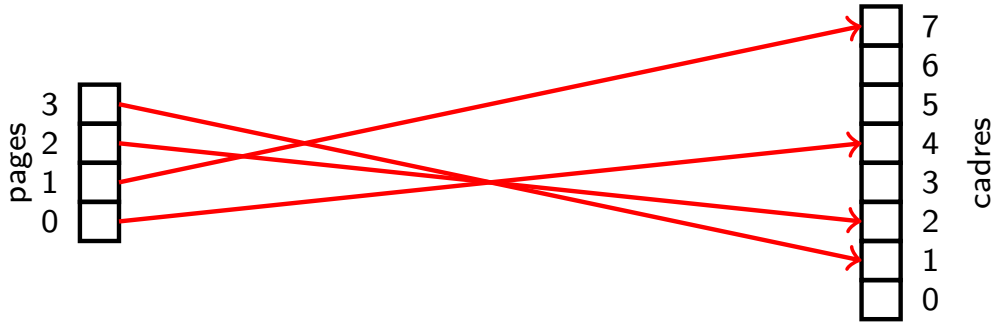
On divise aussi la mémoire réelle en blocs de la même taille s : chacun de ces blocs est appelé cadre.

A chaque page de la mémoire d'un processus, le système va faire correspondre un cadre dans la mémoire vive.

On considère un bloc de mémoire, dont l'ad la plus petite est b. Si a est une ad du bloc, on appelle $r = a - b$ le décalage de a. Notez que $r < s$.

Intérêt de la notion de décalage : le décalage d'une ad est inchangé entre la page et le cadre associé

Comme les pages et les cadres sont tous de même taille s, on attribue un indice à chaque page et à chaque cadre :



Si a_V est une adresse virtuelle on décide que

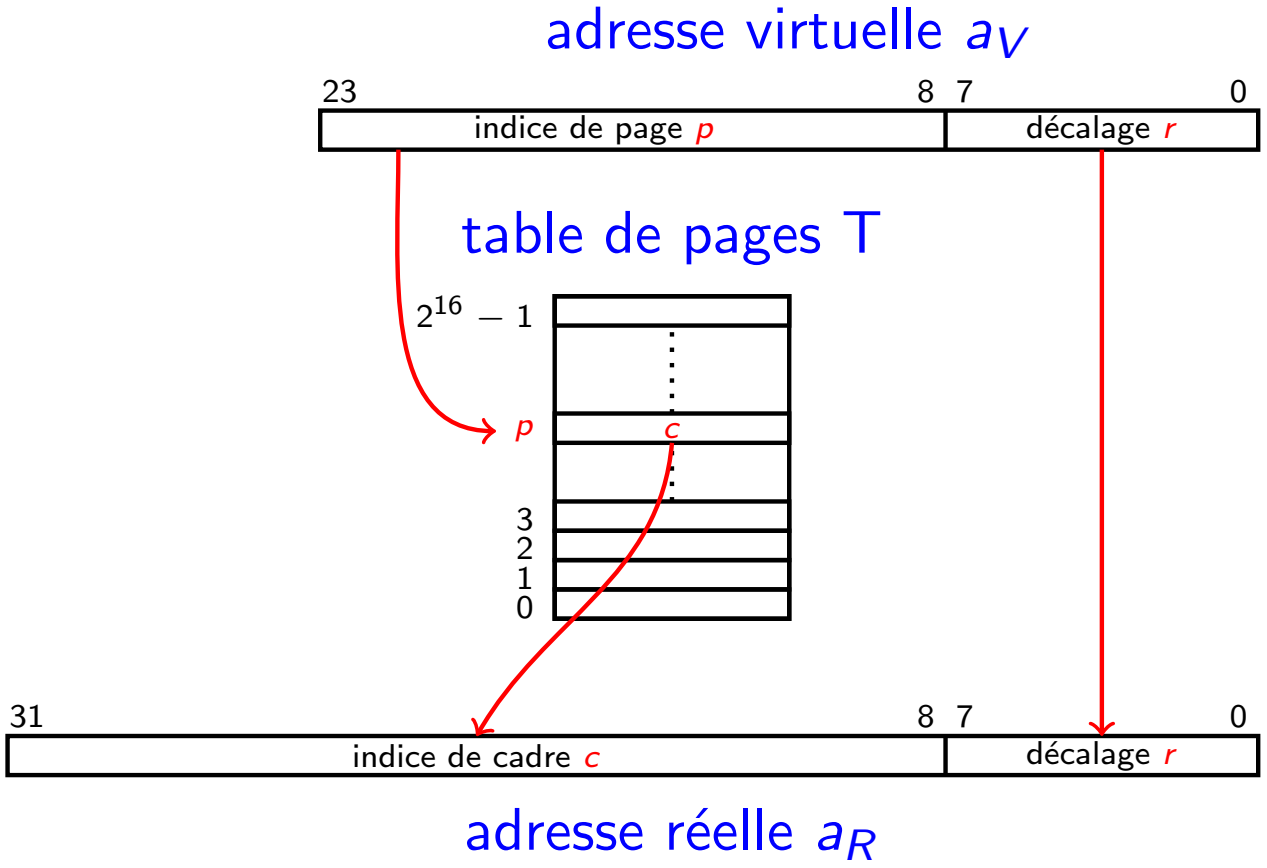
- l'indice de la page à laquelle elle appartient est $p = a_V \text{ div } s$,
- le décalage sur la page est $r = a_V \bmod s$.

Pour chaque processus une table des pages associe un indice de cadre $c = \mathrm{T}[p]$ à p, on obtient ainsi une unique adresse réelle $a_R = c \times s + r$.

On choisit une puissance de 2 pour s, pour que les calculs soient efficaces : dans la suite, on prend $s = 2^8$.

- ad virtuelle a_V : 24 bits, dont 8 bits de décalage,
- ad réelle a_R : 32 bits, dont 8 bits de décalage,
- table des pages T : tableau de 2^{16} indices de cadres sur 24 bits.

$a_V = p2^8 + r$ est associée à l'ad réelle $a_R = c2^8 + r$ où $c = \mathrm{T}[p]$.



Quelques observations :

- On peut supposer que le système chargera dans la MMU la table des pages d'un processus donné quand il lui attribuera le processeur.
- Le système peut partager la mémoire réelle entre plusieurs processus.
- Des ads consécutives d'une page restent consécutives dans le même cadre de mémoire réelle : bon respect du principe de localité spatiale.

Quelques problèmes :

- Mais où sont stockées les tables des pages des processus ? !
- Une table occupe 24×2^{16} bits $= 1,5$ Mio : bcp pour un processus !
- Le système peut accueillir au plus $2^{32-24} = 2^8$ processus : c'est peu !
- Si un processus n'utilise pas tout son espace d'adressage, est-il judicieux de lui allouer autant de cadres que de pages ?

Il faut remettre en cause l'organisation proposée dans notre exemple...

Des « ingrédients » supplémentaires

La structure de données qui permet l'association entre les pages et les cadres doit être maintenue en mémoire vive, pour que le système puisse changer le processus en cours d'exécution.

On n'alloue au processus que les pages qu'il utilise effectivement :

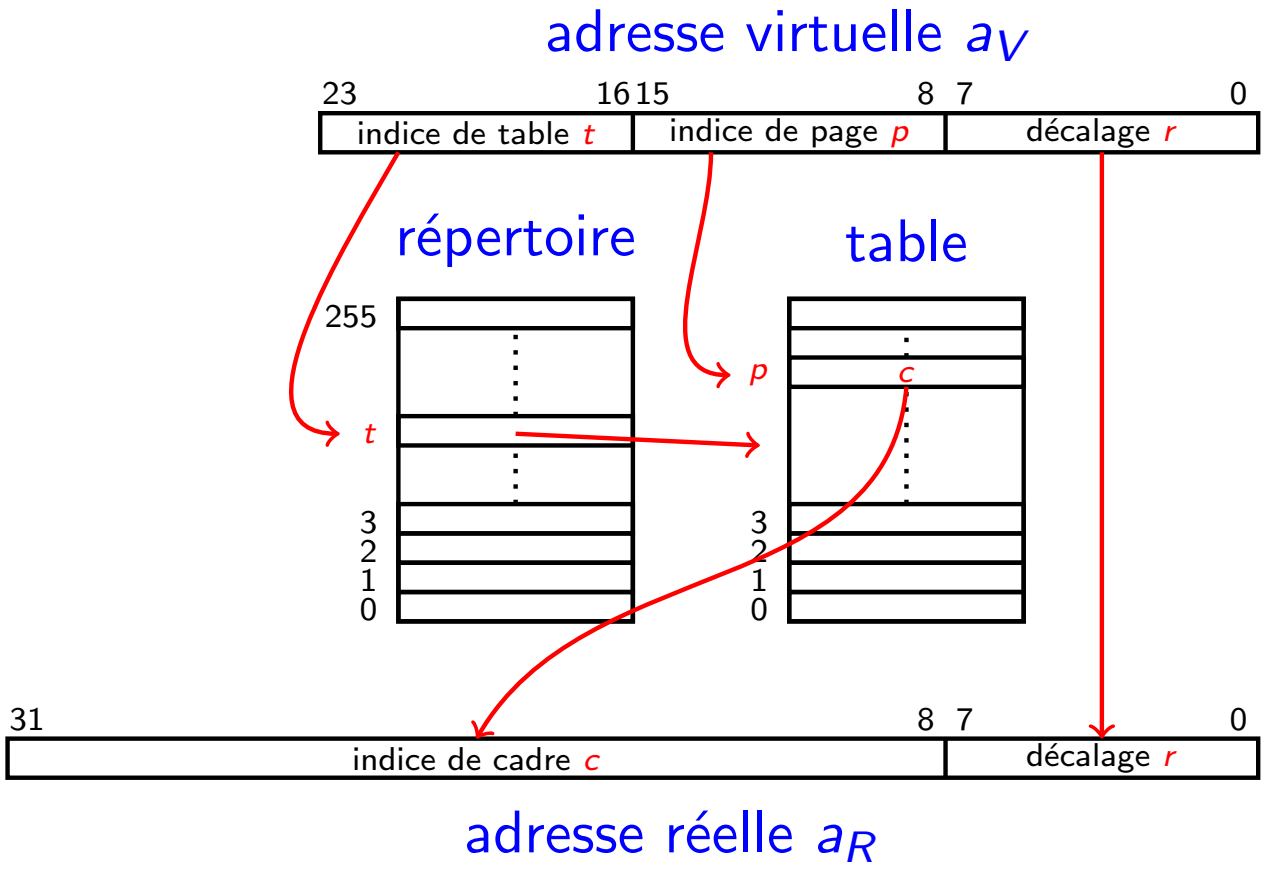
- seules les pages utilisées seront répertoriées et associées à un cadre,
- l'allocation évolue au cours de l'exécution du processus.
- Cela sera géré par le système en lien avec la MMU.

On va introduire un niveau d'indirection :

- pour chaque processus, on introduit un répertoire des tables de pages, dans lequel chaque entrée pointe vers une table de pages.
- Le répertoire est moins « gros » que la table de l'exemple précédent.
- Lors d'un changement de contexte, le système remplace le répertoire utilisé par la MMU.

On reprend notre exemple, avec des ads virtuelles sur 24 bits, dont :

- 8 bits pour l'indice de la table dans le répertoire des tables de pages,
- 8 bits pour l'indice du cadre dans la table des pages.



Cela permet de gérer un ≪ petit ≫ répertoire au lieu d'une ≪ grosse ≫ table.

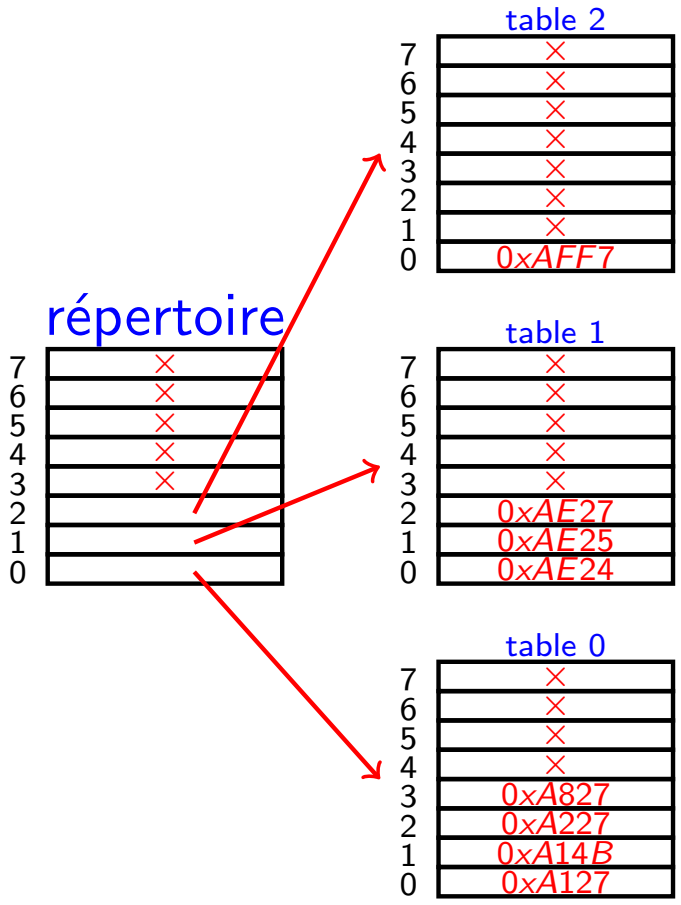
En outre, rien n'oblige à attribuer dès le lancement du processus un cadre à chaque page.

Le système peut

- attribuer un nouveau cadre dans une table,
- allouer une nouvelle table,

selon les besoins du processus.

L'espace d'adressage est inchangé, mais un processus n'occupe en mémoire que les adresses dont il a besoin.



Allocation des pages et gestion des cadres

Un processus peut se voir attribuer de nouvelles pages quand :

- son code est en cours de chargement,
- il fait des allocations sur la pile ou le tas.

Le système doit alors :

- attribuer de nouvelles pages au processus,
- leur associer des cadres en mémoire réelle.

Quand un processus libère de la mémoire ou se termine, le système libére les cadres alloués, pour qu'ils soient disponibles pour d'autres processus.

Le système doit donc :

- maintenir une structure contenant les indices des cadres libres,
- utiliser un algorithme d'allocation judicieux des cadres libres.

En outre, le nombre de pages allouables est généralement plus grand que le nombre de cadres utilisables en mémoire vive.

On peut donc se retrouver dans une situation où :

- un processus demande l'allocation de nouvelles pages,
- mais plus aucun cadre n'est disponible en mémoire vive.

Pour remédier à ce problème, le système peut transférer sur un support de mémoire de masse (le disque) des cadres appartenant à d'autres processus : c'est le principe de la mémoire swap.

Quand un processus alloue de la mémoire :

- soit il reste assez de place sur les pages déjà allouées,
- soit il n'y en a plus assez ! Il y a défaut de page, et le système doit en attribuer (au moins) une nouvelle au processus :
  - soit il reste des cadres disponibles en mémoire vive :
    $\hookrightarrow$ le système alloue les pages et leur associe des cadres.
  - soit il ne reste plus de cadres disponibles en mémoire vive :
    $\hookrightarrow$ le système doit sélectionner des cadres à dépacer en swap.

Quand un processus accède à une page de la mémoire virtuelle :

- soit le cadre correspondant à cette page est en mémoire vive,
- soit il n'y est pas ! Il y a défaut de page :
  $\hookrightarrow$ le système doit sélectionner des cadres à dépacer en swap, avant de charger les cadres accèdés en mémoire vive.

Les défauts de pages sont coûteux, le système doit tenter de les éviter !

L'algorithme d'allocation doit être choisi par le développeur du système de façon à essayer de réduire le nombre de défauts de pages.

Cet algorithme intervient dans deux situations :

- Lors de l'allocation de $k > 0$ pages pour un processus.
  Différentes stratégies sont possibles, par exemple :
  - choix de la 1ère zone de mémoire pouvant accueillir les k cadres,
  - choix de la plus petite zone pouvant accueillir les k cadres...
- Lors du choix d'une ou plusieurs pages à évincer de la mémoire. A nouveau, différentes stratégies sont possibles pour le cadre à évincer :
  - FIFO $\rightarrow$ le premier cadre alloué est le premier à être évincé,
  - LRU $\rightarrow$ le cadre qui n'a pas été utilisé depuis le plus longtemps,
  - La stratégie optimale serait d'évincer le cadre qui ne sera pas utilisé pendant le plus longtemps : mais on ne peut pas lire l'avenir !

Il n'existe pas de stratégie optimale pour tous les processus. Mais l'application du principe de localité par le programmeur contribue aussi à réduire le nombre de défauts.

Conclusion

On a présenté sur un exemple le principe de la mémoire virtuelle paginée, qui permet à plusieurs processus de s'exécuter tout en disposant tous du même espace d'adressage.

L'exemple peut se généraliser d'une façon assez évidente :

- pour d'autres tailles d'adresses réelles et virtuelles,
- en adaptant la taille des pages ou du répertoire des tables.

De nombreuses autres techniques existent pour améliorer l'utilisation de la mémoire, comme le partage de pages de code entre plusieurs processus.

Aussi, nous n'avons introduit qu'un seul niveau d'indirection avec le répertoire des tables de pages : en pratique, il peut y en avoir beaucoup plus, 5 par exemple sous Linux sur un processeur x86-64 [1].

1. https://en.wikipedia.org/wiki/Intel_5-level_paging