Croisière au cœur d'un OS*
Étape 4 : Mise en place de la pagination

Résumé

Grâce à l'article 3, nous pouvons gérer la totalité de la RAM disponible. Dans cet article, nous allons commencer la mise en place de la "mémoire virtuelle" qui permettra d'utiliser du disque en plus de la RAM, facilitera le multitâche et nous permettra de détecter des erreurs de programmation. Pour cela nous allons activer et configurer la "*pagination*". Nous terminerons la mise en place de la mémoire virtuelle en fin de la série d'articles.

Introduction

Sur les premiers ordinateurs, la mémoire physique présente (de type RAM) avait une petite capacité car elle était chère. Les programmeurs devaient ainsi tenir compte de sa taille et les programmes devaient être réécrits dès qu'ils étaient utilisés sur des machines disposant de moins de mémoire que prévu. Depuis la fin des années 1950, des mécanismes de "mémoire virtuelle" ont été développés pour dépasser ces limitations. Aujourd'hui, les processeurs travaillent "sur du vent", avec des données et des instructions qui sont situées dans un espace d'adresses purement "virtuelles", totalement décorrélé de l'espace des adresses physiques. Ces données/instructions peuvent être en réalité stockées en RAM ou bien sur le disque dur (*swap*, fichiers exécutables ou bibliothèques de fonctions) ou sur d'autres machines du réseau local (NFS), etc.

Dans l'article 2 de la série, nous avions déjà évoqué d'une manière incomplète une technique de mise en œuvre de la mémoire virtuelle : la segmentation sur `x86` en mode protégé. Dans cet article, nous allons nous concentrer sur une autre technique de mémoire virtuelle : la "*pagination*" sur `x86`.

Nous présenterons d'abord quelques généralités sur les principes de la mémoire virtuelle et de la traduction d'adresses, en faisant abstraction de la technique choisie (segmentation ou pagination). Nous donnerons davantage de détails que dans l'article 2 puisqu'avec la pagination nous allons pouvoir explorer quelques raffinements de la mémoire virtuelle. Nous nous concentrerons ensuite sur le cas particulier de la pagination, en exposant sa mise en œuvre dans l'architecture `x86`. Enfin, nous décrirons la mise en place de la pagination dans SOS, puis la petite démo habituelle. L'article offre en annexe quelques compléments d'informations sur les principes plus abstraits et plus généraux de la pagination et ses différences par rapport à la segmentation.

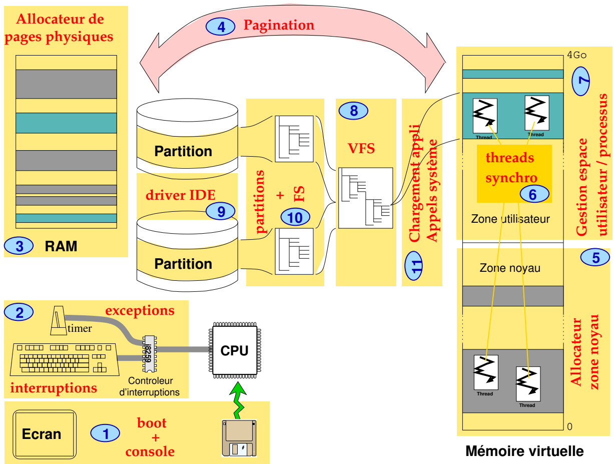


FIG. 1 – Programme des articles

1 Principes de la mémoire virtuelle

Dans cette partie, nous détaillerons les principes généraux relatifs à la mémoire virtuelle et à la traduction d'adresses.

1.1 Mémoire virtuelle, MMU et traduction d'adresses

L'objectif associé à la "mémoire virtuelle" est de pouvoir écrire des programmes en considérant que la mémoire disponible est "infinie" ou du moins indépendante de la RAM disponible. À cette fin, les autres ressources disponibles (telles que le disque dur local ou celui d'une autre machine sur le réseau) peuvent être utilisées pour stocker les parties de cette mémoire virtuelle qui ne servent pas souvent. Ceci permet de libérer de la RAM pour y stocker ce qui sert plus fréquemment. C'est le mécanisme de *swap* qu'on retrouve sur la plupart des OS génériques, tels que Linux.

Ce mécanisme correspond au fonctionnement d'un *cache* : la RAM sert de tampon pour les accès

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 65 – Octobre 2004 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

aux données/instructions qui se situent dans un espace beaucoup plus grand (la mémoire virtuelle). On l'intègre souvent dans la pyramide de la "hiérarchie mémoire" (figure 2) dans laquelle chaque niveau sert en quelque sorte de cache au niveau du dessous : les accès se font de haut en bas dans cette pyramide et lorsqu'une donnée n'est pas présente à un niveau n, elle est demandée au niveau du dessous. En pratique, ce schéma purement théorique est légèrement démenti puisqu'il n'y a pas de réelle séparation qui isole la RAM de la mémoire virtuelle en bas de la hiérarchie : les deux sont très intimement reliées.

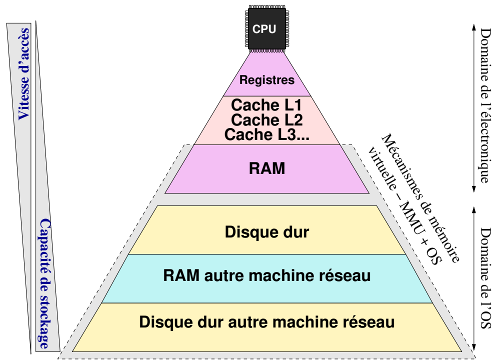


FIG. 2 – Schéma vulgaire de la hiérarchie mémoire

Toute la mise en œuvre de la mémoire virtuelle repose en fait sur un seul mécanisme fondamental et plus général : la *traduction d'adresses*. Cette dernière consiste à découpler *i)* les adresses des données et des instructions manipulées par le processeur de *ii)* leur emplacement physique (en RAM ou sur le disque ou ailleurs). Elle est prise en charge par l'*unité de gestion de la mémoire* (*MMU*). En pratique, la MMU peut être une puce externe au processeur située entre le processeur et la RAM (figure 3) ou être intégrée directement dans la puce qui contient le processeur.

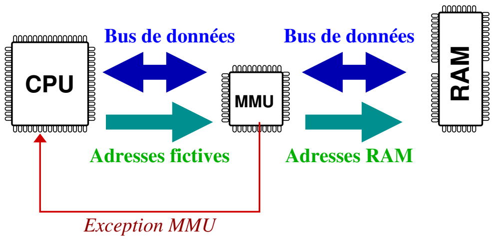


FIG. 3 – Relations entre processeur, MMU et RAM (MMU externe au CPU sur cet exemple)

Tous les processeurs n'utilisent pas nécessairement de MMU : la plupart des processeurs ou microcontrôleurs embarqués s'en dispensent. Dans ce cas, le processeur est obligé de travailler avec l'espace des adresses RAM (fini et restreint). Par contre, tous les processeurs génériques grand public actuels (x86, ppc, ...) utilisent voire intègrent une MMU (depuis la fin des années 1980 seulement !). La suite de l'article ne s'intéressera qu'à ces derniers.

1.2 Synopsis de fonctionnement

Avec un mécanisme de traduction d'adresses, le processeur travaille "sur du vent", avec des adresses de données et d'instructions qui sont purement *fictives* : il n'a aucune notion des adresses physiques en RAM. Il ne se soucie pas de savoir si les données et instructions qu'il manipule se trouvent initialement en RAM ou sur le disque ou ailleurs. Il dépend de la MMU et du système d'exploitation que ces données et ces instructions soient ramenées en RAM avant de pouvoir être utilisées. Le déroulement de ces opérations se fait en deux étapes (voir la figure 4).

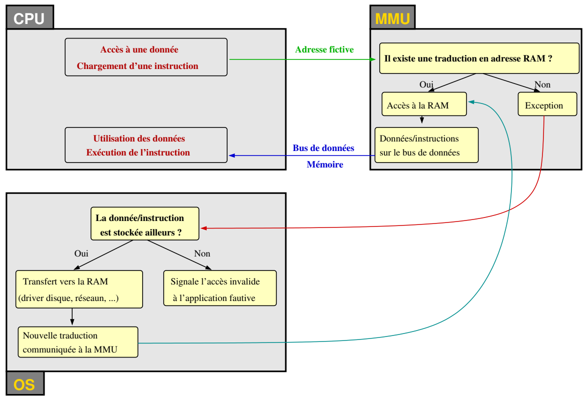


FIG. 4 – Principe de la mémoire virtuelle

1.2.1 Première étape (systématique) : la MMU

Le processeur commence par fournir chaque adresse qu'il manipule (purement fictive) à la MMU. Celle-ci s'occupe de les traduire en adresses en RAM (et en RAM **seulement**). Deux cas peuvent se produire.

Si la MMU dispose d'une traduction pour l'adresse fictive fournie par le processeur, alors elle se charge de l'accès à la RAM. Le processeur récupère ensuite la donnée/instruction sur le bus mémoire et peut continuer son travail.

Si la traduction par la MMU échoue, c'est que les données ne sont pas en RAM. Dans ce cas, la MMU signale une exception au processeur et c'est ensuite au système d'exploitation de la prendre en charge.

1.2.2 Deuxième étape (si besoin seulement) : le système d'exploitation

Le système d'exploitation définit et maintient ses propres structures de données internes pour la gestion des adresses fictives. Lorsqu'il prend le relais pour traiter l'exception levée par la MMU, il utilise ces données pour déterminer où sont les données/instructions référencées par l'adresse fictive qui a provoqué l'exception (à quel endroit dans le swap ? dans quel fichier ?) ou pour signaler que l'adresse fictive ne référence rien de connu.

Si l'OS constate que l'adresse fictive correspond à un emplacement sur le disque dur, il peut transférer les données depuis cet emplacement du disque vers la RAM si de la place est disponible. Ce mécanisme porte le nom générique de "demand paging" et est au cœur du fonctionnement du *file mapping* (*projection de fichier*) ou du *swap* (entre autres). Si aucune place en RAM n'est disponible, il peut transférer un morceau peu utilisé de la RAM vers le disque (en veillant à mettre à jour les tables de traduction d'adresses) et il peut ainsi profiter de ce nouvel emplacement disponible. Ce mécanisme est également au cœur du fonctionnement du *file mapping* et du *swap*. Nous évoquons ces mécanismes ici mais il faudra attendre quelques articles avant que nous les détaillions.

Si le système d'exploitation constate que l'adresse fictive ne correspond à rien (ni en RAM, ni sur disque, ni sur aucune autre ressource de stockage), il peut avertir l'application qui a travaillé avec l'adresse fictive erronée. Ceci correspond au fameux "*segmentation fault*" sous Unix. Ainsi, la traduction d'adresses permet aussi de détecter et de signaler des erreurs de programmation.

1.2.3 L'OS est en dehors de la MMU et du CPU ?!

Lorsqu'on dit que "l'OS prend le relais quand la MMU n'a pas pu traduire l'adresse fictive fournie par le processeur", cela laisse à penser que l'OS constitue une entité en dehors de tout le reste (processeur et MMU). En réalité il n'en est rien : l'OS est fait de code et de données qui aussi sont traités par le processeur. Ce qui vient d'être décrit s'applique donc à l'OS lui-même. Ceci induit ainsi un niveau de récursivité supplémentaire (pour traiter les défauts de traduction de la MMU, l'OS utilise la MMU). Cet aspect aurait été difficile à faire apparaître sur la figure et dans le synopsis.

1.3 Tables de traduction d'adresses

Pour effectuer la traduction d'adresses, la MMU utilise une ou plusieurs *tables de traduction d'adresses* qui se trouvent dans la RAM ou qui sont internes à la MMU pour certains modèles de MMU. Ces tables servent de dictionnaire à la MMU : pour une adresse fictive donnée, elles contiennent l'adresse en RAM associée. Il serait fastidieux voire impossible d'utiliser des tables qui permettraient de stocker la traduction `adresse fictive` $\rightarrow$ `adresse en RAM` pour **chaque adresse fictive de l'espace virtuel** (*ie* chaque octet de mémoire virtuelle) : ceci nécessiterait une quantité faramineuse de données de traduction puisque l'espace des adresses fictives est souvent extrêmement grand.

L'espace des adresses fictives est en fait découpé en morceaux (de taille finie) et il suffit de retrouver les adresses physiques associées à un morceau fictif pour avoir la traduction d'une adresse fictive en son adresse physique. On ne s'occupe que de la traduction des adresses de *début* de chaque morceau (figure 5) et l'emplacement dans un morceau est conservé à l'identique par la traduction d'adresses : c'est le "déplacement" dans le morceau (ou l'*offset*), une notion que nous avons déjà rencontrée plusieurs fois. Ce découpage en morceaux ainsi que la traduction des adresses de chaque début de morceaux est commun à toutes les techniques de traduction d'adresses.

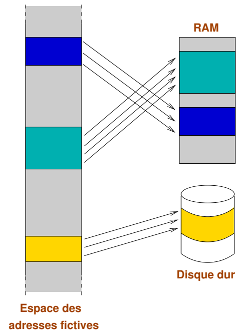


FIG. 5 – Traduction d'adresses par morceaux

1.4 Intérêts de la mémoire virtuelle

1.4.1 Intérêts initiaux

Nous avons déjà évoqué certains intérêts des mécanismes de mémoire virtuelle. Ils permettent de :
- s'abstraire de la taille de la RAM disponible ;
- utiliser des ressources de stockage autres que la RAM : disque dur et pourquoi pas, la RAM ou le disque dur d'une machine distante sur le réseau ;
- signaler des erreurs de programmation grâce au contrôle de tous les accès aux adresses fictives par l'intermédiaire de la MMU.

1.4.2 Mémoire virtuelle et multi-programmation

Nous verrons en fin de la série que les techniques de mémoire virtuelle permettent également de faciliter la *multi-programmation*, c'est-à-dire la faculté de pouvoir exécuter simplement plusieurs applications sur le même processeur. Sans entrer dans les détails, en voici le principe.

Pour faire fonctionner une application A sur le processeur, on met en place les tables de traduction d'adresses pour cette application. Ceci établit les liens entre les adresses fictives manipulées par le processeur figurant dans le fichier exécutable de l'application A et les adresses physiques (en RAM ou sur le disque dur). On peut très bien imaginer de faire fonctionner une autre application B dans le système en définissant d'autres tables de traduction d'adresses (voir la figure 6). On a alors deux jeux d'adresses fictives dans le système : celles pour l'application A quand le processeur l'exécute relativement aux tables de traduction d'adresses de A, et celles pour l'application B quand le processeur l'exécute relativement aux tables de traduction d'adresses de B. Pour passer de A à B et inversement, il s'agit donc (entre autres) de changer les tables de traduction utilisées par la MMU. On dit qu'on a ainsi défini deux *espaces d'adresses fictives* distincts, ou plus couramment deux "*espaces d'adressage*". Sous Unix, ceci amène le concept de "*processus*".

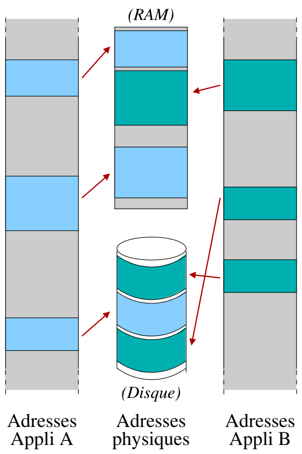


FIG. 6 – Mémoire virtuelle et multi-programmation

Cloisonnement mémoire. Puisqu'une application n'a accès qu'aux données qui sont couvertes par ses tables de traduction d'adresses, on a un cloisonnement naturel entre applications. Il suffit pour cela que les tables de traduction d'adresses associées aux applications ne recouvrent aucune adresse identique en RAM ou sur disque. On accroît de cette manière la robustesse et la sécurité du système en ne permettant pas à une application d'écraser le code ou les données des autres applications, ni d'y avoir accès en lecture.

Partage de mémoire. À l'inverse, l'OS peut modifier les tables de traduction d'adresses en cours de fonctionnement des applications. On peut par exemple modifier certaines traductions de deux applications A et B afin qu'elles recouvrent une série d'adresses physiques (en RAM ou sur disque) identiques (figure 7). De la sorte, on a réalisé très simplement un partage de mémoire physique "à chaud", c'est-à-dire alors même que les applications avaient déjà commencé leur exécution.

Ce principe est utilisé abondamment dans les Unix (`man mmap`) pour le chargement de programmes, de bibliothèques de fonctions, ou même l'accès aux fichiers pour certains OS (Hurd par exemple). Il permet de limiter la consommation d'espace mémoire physique lorsqu'un même programme ou fichier est chargé/ouvert plusieurs fois. Par exemple, le code d'un programme ne sera présent qu'une seul fois en RAM même si plusieurs processus en émanent : ainsi le code de l'interpréteur de commandes `bash` sera en RAM en un seul exemplaire pour tous les utilisateurs logés dans le système et pour tous les scripts `bash` qui sont lancés.

Signalons que cette technique fonctionne entre applications différentes mais également à l'intérieur d'une même application. Les mêmes données/instructions (physiques) peuvent très bien être présentes à plusieurs endroits dans l'espace d'adressage d'une application donnée (c'est le cas en bas à droite de la figure 7). Dans la petite démo en fin d'article, nous rencontrerons cette situation.

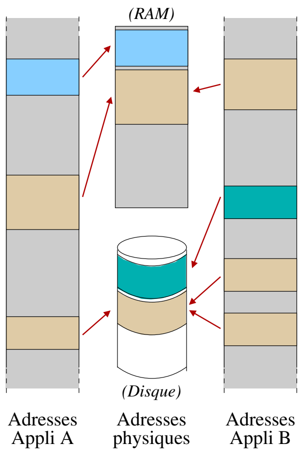


FIG. 7 – Partage de mémoire physique (disque et RAM)

2 Pagination sur `x86`

Dans le deuxième article de la série nous avons vu un premier type de traduction d'adresses sur `x86` en mode protégé : la segmentation. Avec la segmentation, un "morceau" au sens de la section 1.3 correspond à un segment. Dans cet article, nous décrivons la pagination qui est une autre démarche de traduction d'adresses, aujourd'hui plus répandue.

2.1 Petit rappel

Sur l'architecture `x86` (en mode protégé), Intel a préféré ne pas trancher entre segmentation et pagination. Leur MMU supporte en effet les deux mécanismes qui peuvent être utilisés en même temps.

Comme nous l'avons vu dans l'article 2, l'adresse fictive manipulée par le processeur, appelée adresse *logique* dans l'architecture `x86`, est traduite en deux étapes :

1. `adresse logique` → `adresse linéaire` par la partie *segmentation* de la MMU du `x86`,
2. `adresse linéaire` → `adresse physique` (*ie* en RAM) par la partie *pagination* de la MMU du `x86`;

En mode protégé, la segmentation est systématique et la pagination est optionnelle. Nous ne parlons pas du mode réel puisqu'aucune MMU digne de ce nom n'est utilisée.

Dans toute cette section, nous ne nous intéressons qu'à la partie *pagination* de la MMU. La partie *segmentation* a été évoquée dans l'article 2, aussi nous n'y revenons pas ici.

2.2 Caractéristiques générales

Le terme "pagination" vient de "page". En informatique, une "page" de mémoire (fictive, physique, ou sur le disque dur) est un bloc de taille donnée dépendante de l'architecture, de l'ordre de quelques kilo-octets le plus souvent, ou de quelques mega-octets plus rarement. Avec la pagination, un "morceau" au sens de la section 1.3 correspond à une page.

2.2.1 Espaces d'adresses fictif et physique

Les tables de traduction d'adresses pour la pagination permettent à la MMU de traduire les adresses fictives de début des pages, en adresses des pages en RAM. La pagination sur l'architecture `x86` permet de traduire des adresses linéaires sur 32 bits vers des adresses physiques sur 32 bits (*ie* on peut accéder à 4 Go de RAM au maximum), avec des pages de 4 Ko [1, section 3.6] comme nous l'avions annoncé dans l'article 3. Lorsque la MMU ne trouve aucune traduction en adresse physique ou détecte un problème de droit d'accès, elle signale l'exception 14 "`Page Fault`" au processeur [1, section 5.12]. Les tables de traduction d'adresses sont des tables à deux niveaux d'indirection (voir la figure 8), la table de premier niveau permettant d'accéder aux tables du second niveau.

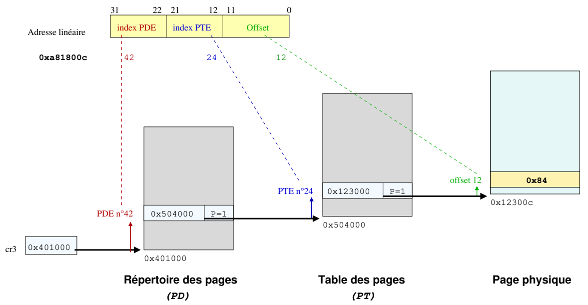


FIG. 8 – Pagination sur `x86` : Exemple avec l'adresse linéaire `0xa81800c` traduite en adresse physique 0x12300c et contenant la valeur `84`

En fait, l'architecture `x86` supporte d'autres types de tables de traduction : à 3 niveaux avec l'extension PAE [1, section 3.8] ; à 4 niveaux dans l'architecture `x86-64` [2, section 5][3, section 1.6.5]. Il supporte aussi d'autres tailles de pages (4Mo ou 2Mo avec l'extension PSE [1, section 3.6.2] et/ou PAE) et des adresses physiques sur 36 bits au lieu de 32 (extension PAE). Nous n'utiliserons aucune de ces extensions dans SOS, donc nous n'en parlerons pas dans ce qui suit.

Dans tous les cas, sur `x86`, toutes les tables de traduction d'adresses se situent dans la RAM.

2.2.2 Fonctionnement de la MMU

Une adresse linéaire (32 bits) se décompose de la façon suivante pour la partie *pagination* de la MMU :

bits 31-22 : L'index dans la table de premier niveau, où est stockée l'adresse de début de la table de second niveau en RAM ;

bits 12-21 : L'index dans la table de second niveau, où est stockée l'adresse de début de la page en RAM ;

bits 0-11 : Le déplacement dans la page.

La table de premier niveau s'appelle le "*répertoire des pages*" (*Page Directory* ou *PD*) et les tables de second niveau s'appellent les "*tables des pages*" (*Page Table*s ou *PT*s). Toutes ces tables contiennent chacune 1024 entrées.

Les entrées dans le répertoire des pages (resp. dans les tables des pages) indiquent bien sûr l'adresse en RAM de la table des pages (resp. la page) quand elle est en RAM (adresses dans la figure précédente). Elles indiquent également si la table des pages (resp. la page) associée est effectivement présente en RAM ("`P=1`" sur la figure précédente), ainsi que d'autres informations telles que les droits d'accès ou les niveaux de privilèges requis pour y accéder (voir la section 2.3).

Pour traduire une adresse linéaire, la partie *pagination* de la MMU :

1. récupère l'adresse RAM du répertoire des pages (voir plus bas),
2. récupère l'adresse de début (en RAM) de la table des pages parmi les 1024 adresses contenues dans le répertoire des pages (à partir des bits 31-22 de l'adresse linéaire),
3. récupère l'adresse de début de la page en RAM parmi les 1024 adresses contenues dans la table des pages (à partir des bits 21-12 de l'adresse linéaire),
4. ajoute le déplacement dans la page (à partir des bits 11-0 de l'adresse linéaire).

Si la MMU détecte une erreur à l'une de ces trois étapes (table des pages déclarée non présente en RAM, page déclarée non présente en RAM, problèmes de droits d'accès ou de privilège), elle signale une exception au processeur afin que le système d'exploitation prenne le relais.

2.2.3 Un peu d'arithmétique simple

Une entrée dans une table des pages correspond à une page d'adresses linéaires, qu'il y ait une page en RAM associée ou non. Donc une entrée dans la table des pages couvre 4 Ko d'adresses linéaires.

Une entrée dans le répertoire des pages correspond à une table des pages. Puisqu'une table des pages possède 1024 entrées, elle couvre donc un espace total de 1024 * 4 Ko, soient 4 Mo d'adresses linéaires contiguës. Chaque entrée du répertoire des pages couvrira donc 4 Mo d'adresses linéaires.

Pour terminer, puisqu'il y a 1024 entrées dans le répertoire des pages, ce dernier couvre donc un espace total de 1024 * 4Mo = 4 Go d'adresses linéaires. On retrouve bien la capacité de 2^{32} octets de l'espace des adresses linéaires (voir la section 2.2.1).

Tout ceci permet d'identifier très simplement à quel index dans le répertoire des pages et à quel index dans la table des pages associée correspond telle adresse linéaire. Par exemple l'adresse linéaire `0xa81800c` s'écrit $42 * 4Mo + 24 * 4Ko + 12$: on retrouve la décomposition de la figure 8.

2.2.4 Cache de traduction d'adresses

Si on s'en tient au fonctionnement de la traduction d'adresses décrit ci-dessus, on constate que chaque accès à une donnée/instruction dont on connaît l'adresse linéaire implique en fait 3 accès en RAM : un accès dans le répertoire des pages pour récupérer l'adresse de la table des pages concernée, un accès dans cette table des pages pour récupérer l'adresse de la page concernée, et un accès à la donnée dans cette page.

Pour accélérer les traductions d'adresses, la MMU dispose d'un cache de traduction d'adresses interne à accès très rapide : le TLB [1, section 3.7]. Ce cache conserve un nombre limité de traductions `adresse linéaire → adresse physique` (typiquement 32 à quelques centaines suivant les MMU, mais le programmeur n'a pas à connaître sa taille exacte). Il est mis à jour au gré des traductions complètes (3 accès à la RAM). Il permet de court-circuiter au maximum les 2 premiers accès à la RAM lorsqu'un accès à la même page a été précédemment effectué. Évidemment ce cache ne conserve pas tous les résultats des traductions effectuées, seulement les plus récentes. Donc la traduction complète à 3 accès à la RAM continue d'avoir lieu, mais plus rarement.

2.3 Répertoires et tables des pages

Nous avons vu que le répertoire des pages et les tables des pages sont des tableaux de 4ko qui contiennent chacun 1024 entrées de 4 octets : les "PDE" (*Page Directory Entries*) pour le répertoire de pages et les "PTE" (*Page Table Entries*) pour les tables des pages (voir la figure 8 pour un aperçu).

La norme Intel [1, section 3.6.4] spécifie que ces deux types d'entrée ont la même structure, mis à part pour deux bits. Parmi les éléments importants de leur structure figurent :

bits 31-12 : les 20 bits de poids fort de l'adresse en RAM de la table des pages (pour les *PDE*) ou de la page (pour les *PTE*) associée. Ces bits permettent d'identifier totalement une table des pages/une page. Les autres bits d'une adresse en RAM d'une table de pages/d'une page (*ie* les 12 bits de poids faible) valent en effet 0 car une table de pages/une page commence toujours à une adresse multiple de 4 Ko (*ie* 2^{12}).

bit 2 : indique le privilège requis pour accéder à la table des pages (*PDE*) ou à la page (*PTE*) associée : vaut 0 si la table des pages/la page n'est accessible qu'en mode superviseur (voir l'article 2), vaut 1 sinon.

bit 1 : indique si la table des pages (*PDE*) ou la page (*PTE*) associée est accessible en lecture seule (vaut 0), ou si elle est accessible en lecture/écriture (vaut 1).

bit 0 (bit "`P`" pour "*present*") : indique si la table des pages (*PDE*) ou la page (*PTE*) associée est présente en RAM.

2.4 Registres liés à la pagination

Pour mettre en place la pagination, l'élément fondamental à configurer est l'adresse du répertoire des pages. Une fois qu'on a indiqué à la MMU où se situe ce répertoire des pages, tous les accès mémoire se font relativement à l'espace d'adressage défini par ce répertoire des pages et les tables des pages qu'il référence.

Sur `x86`, l'adresse en RAM du répertoire des pages est contenue dans le registre `cr3` (*Crontrol Register 3*) [1, section 2.5] appelé aussi *Page Directory Base Register* (*PDBR* ; voir la figure 8 pour le situer). En fait, seuls les 20 bits de poids fort doivent être positionnés, ce qui veut dire que le répertoire des pages est toujours situé à une adresse en RAM multiple de 4 Ko. En pratique, les 12 bits de poids faible sont à 0 : la plupart sont "réservés" par Intel, sauf 2 qui valent 0 dans notre cas.

Pour revenir sur la section 1.4.2, remarquons que si on modifie la valeur du registre `cr3`, alors on change d'espace d'adressage fictif (cela revient à utiliser d'autres tables de traduction d'adresses). Il est très impressionnant de réaliser qu'en changeant un simple registre, on modifie un élément fondamental de l'exécution sur le processeur : tout le contexte mémoire.

Le deuxième registre lié à la pagination est `cr0` [1, section 2.5] et plus précisément son bit 31. La partie pagination de la MMU (désactivée par défaut) est activée dès qu'on positionne ce bit à 1.

2.5 Mise en place de la pagination

La mise en place de la pagination sur `x86` procède de 3 étapes :

1. configuration de toutes les tables de traduction d'adresses (répertoire et tables des pages),
2. initialisation du registre `cr3` avec l'adresse en RAM du répertoire de pages,
3. positionnement du bit 31 de `cr0` à 1 pour activer la pagination et utiliser immédiatement toutes les tables de traduction initialisées ci-dessus.

La difficulté principale est qu'il faut garantir que les tables de traduction d'adresses seront correctement initialisées avant d'activer la pagination. Sinon, une fois l'étape 3 validée, le processeur exécutera n'importe quoi à des adresses incorrectes ou invalides.

Pour cela, le plus simple est d'initialiser la pagination en "*Identity Mapping*" : c'est une configuration particulière des tables de traduction d'adresses. Elle permet que la mise en place de la pagination ne perturbe pas le déroulement de l'exécution du code. En effet, avant la mise en place de la pagination, le processeur travaille avec des adresses physiques. Et juste après la mise en place de la pagination il travaille avec des adresses linéaires. Le plus simple pour le programmeur est de ne pas avoir à tenir compte de ce changement. Donc le plus simple est de faire en sorte que les adresses processeur immédiatement avant (adresses physiques) et immédiatement après (adresses linéaires) la mise en place de cette pagination soient égales. L'identity mapping correspond justement à cette configuration particulière des tables de traduction d'adresses : les adresses linéaires sont égales aux adresses en RAM lors de la mise en place de la pagination.

Par la suite, on pourra modifier les tables de traduction d'adresses pour briser cette correspondance directe entre adresses fictives et adresses en RAM, sinon la pagination n'a aucun intérêt.

2.6 Modification des tables de traduction d'adresses

En théorie, la modification des tables de traduction d'adresses consiste à modifier les entrées du répertoire des pages et/ou les entrées des tables des pages.

Mais en pratique il y a deux précautions importantes à prendre :

1. Les tables de traduction d'adresses doivent aussi avoir une adresse virtuelle. Sinon le processeur ne peut pas les modifier. Autrement dit, les tables de traduction doivent contenir au moins une traduction d'adresse vers elles-mêmes ;
2. Quand on modifie les tables de traduction d'adresses, il faut enlever les traductions du TLB devenues obsolètes : on parle d'"invalider" des entrées du TLB. Sinon on continue d'accéder à l'ancienne donnée même après la modification des tables.

Sur `x86` il y a principalement deux manières d'invalider des traductions stockées dans le TLB :

– en enlevant *toutes* les traductions stockées dans le TLB d'un coup. Pour ce faire, il suffit de remettre l'adresse du répertoire des pages dans le registre `cr3`. Cette opération provoque systématiquement le vidage complet du TLB ;
– en exécutant l'instruction `invlpg`. Cette instruction prend en paramètre une adresse linéaire `laddr` et enlève la traduction `laddr` → `adresse physique` du TLB si elle y est présente. Sinon elle ne fait rien.

Le TLB est un élément déterminant dans l'efficacité de la MMU. On veillera donc à invalider ses entrées avec le plus de parcimonie possible.

3 Gestion de la pagination dans SOS

Dans SOS (voir l'article 2), nous avons configuré la segmentation suivant le modèle *flat* afin d'avoir l'équivalence entre adresses logiques et linéaires. Ceci signifie que nous ne gérerons la mémoire virtuelle que par l'intermédiaire d'un seul des deux mécanismes de la MMU de l'architecture `x86` : la pagination. Dans ce qui suit, nous distinguons explicitement le type des adresses physiques `sos_paddr_t` de celui des adresses linéaires (également logiques) `sos_vaddr_t` (fichier `sos/types.h`).

Dans les premières sections du présent article, nous avons évoqué un grand nombre d'aspects de la mémoire virtuelle : de la traduction d'adresses par la MMU jusqu'au *file mapping* ou au *swap* par l'OS. Dans toute la suite, nous implantons uniquement le nécessaire pour configurer et modifier les tables de traduction pour la pagination sur `x86` (répertoire et tables des pages). L'implantation des mécanismes de plus haut niveau (*file mapping*, *swap*, ...) sera vue en toute fin de la série d'articles.

Tout le code associé à cet article se résume à deux fichiers : `hwcore/paging.h` pour les déclarations et `hwcore/paging.c` pour les définitions et l'implantation. Ces fichiers sont très courts et ne semblent qu'implanter "tout simplement" une interface de programmation pour la gestion des tables de traduction de la pagination. Qu'on ne s'y trompe pas cependant : le principe fondamental sur lequel ils reposent est relativement subtil et mérite qu'on le détaille.

3.1 Principe central : le *mirroring*

3.1.1 Rappel et présentation

Nous avons fait observer (section 2.6) qu'il faut que les tables de traduction soient accessibles par des adresses linéaires si on veut pouvoir les modifier après que la pagination ait été activée.

Le "*mirroring*" que nous décrivons ici est un moyen simple de maintenir automatiquement à jour les traductions entre les adresses linéaires des tables de traduction (répertoire et tables des pages) et leurs adresses physiques associées, même lorsque les entrées du répertoire des pages (*ie* les adresses des tables des pages) sont modifiées. Ce mécanisme est assez original et nous avait été présenté par Christophe Avoinne au début de KOS. Nous employons le terme "mirroring" sans savoir s'il est le terme consacré.

3.1.2 Observations fondamentales

Tout le principe du mirroring s'appuie sur les caractéristiques particulières des structures des tables de traduction pour la pagination sur `x86` :

– Le répertoire des pages (PD) sur `x86` est un tableau de 4 Ko formé de 1024 PDE. Et chaque table des pages (PT) est un tableau de 4 Ko formé de 1024 PTE. Les pages normales ont elles-mêmes une taille de 4 Ko ;
– Les PDE et les PTE ont presque la même structure (voir la section 2.3).

Autrement dit, on a envie de se dire que le répertoire des pages et les tables des pages d'une part, et les PDE et les PTE d'autre part, sont "presque" interchangeables. Le seul point délicat est que PDE et PTE n'ont pas tout à fait la même structure. En pratique cependant tous ces éléments sont bien interchangeables : les 2 bits dont la signification diffère entre PDE et PTE sont "compatibles". En effet, même s'ils ne signifient pas la même chose pour les PDE et les PTE, la valeur qu'ils prennent pour ces 2 types d'entrées est la même.

3.1.3 Mise en œuvre

La mise en œuvre est extrêmement simple mais s'y habituer nécessite un peu d'efforts. C'est là l'sort de la marine et de la récursivité qui n'est pas toujours facile à appréhender. Tout se résume en une phrase :

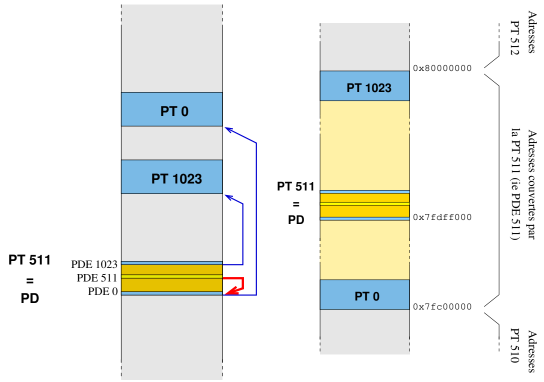
> Lorsqu'on construit le répertoire des pages (disons : à l'adresse $AddrPD$ en RAM), on indique dans une entrée arbitraire de ce répertoire des pages (disons : celle à l'index $idxMirror$) que l'adresse de la table des pages associée est... $AddrPD$.

Ou, plus formellement :

$$PDE[idxMirror] = addrPD$$

Les effets de cette petite mise en place qui n'a l'air de rien sont les suivants (voir la figure 9) :

- La table des pages référencée par la PDE d'index $idxMirror$ n'est pas une table des pages anodine : c'est le répertoire des pages lui-même.
  - Dit autrement : la table des pages qui couvre les 4 Mo à partir de l'adresse linéaire $idxMirror * 4Mo$ n'est autre que le répertoire des pages lui-même.
- Dans cette table des pages, les PTE sont en fait des PDE puisque cette table des pages est le répertoire des pages lui-même.
  - Autrement dit : la PTE d'index i dans cette table des pages ne renvoie pas à une page physique anodine mais à la table des pages référencée par la PDE d'index i.
  - Dit encore autrement : pour modifier le contenu (*ie* les PTE) de la i^{eme} table des pages (parmi les 1024 possibles), il suffit d'accéder à la page d'adresse linéaire $idxMirror * 4Mo + i * 4Ko$.
  - En particulier : pour modifier le répertoire des pages (*ie* les PDE), qui se trouve aussi être la $idxMirror^{eme}$ table des pages, il suffit d'accéder à la page d'adresse linéaire $idxMirror * 4Mo + idxMirror * 4Ko$.



(a) Adresses des tables en RAM

(b) Adresses linéaires correspondantes

FIG. 9 – Configuration obtenue par mirroring avec $idxMirror = 511$. Exemple avec les PT 0, 511 et 1023.

En résumé, avec cette configuration toute simple du répertoire des pages, on accède à toutes les PTE possibles par les 4Mo situés à l'adresse linéaire $idxMirror * 4Mo$ et on accède aux 1024 PDE par la page située à l'adresse linéaire $idxMirror * 4Mo + idxMirror * 4Ko$.

Une partie de l'espace des adresses linéaires *reflète* ainsi la configuration de cet espace des adresses linéaires (en termes de tables de traduction d'adresses), d'où le terme "mirroring". L'intérêt de cette méthode est que si on modifie la i^{eme} PDE, la nouvelle table des pages associée est automatiquement accessible à l'adresse $idxMirror * 4Mo + i * 4Ko$ sans aucune autre opération. Toute autre solution aurait nécessité de changer à la main une PTE pour garder la table des pages accessible quelque part dans l'espace des adresses linéaires. Ici, puisque les PDE sont aussi les PTE d'une table des pages particulière (le répertoire des pages en l'occurrence), ceci se fait automatiquement dès qu'on modifie la PDE.

3.1.4 Remarque

Insistons sur le fait que cette technique de mirroring est très particulière à l'architecture `x86` : elle repose sur les observations faites en section 3.1.2 qui sont vraiment propres à cette architecture. Le portage du *mirroring* sur d'autres architectures risque d'être difficile voire impossible. Il suffirait néanmoins d'implanter `hwcore/paging.c` différemment, l'interface de programmation déclarée dans `hwcore/paging.h` s'abstrayant de la façon dont sont gérées les tables de traduction.

3.2 Initialisation de la pagination

En plus de définir les fonctions déclarées dans `hwcore/paging.h`, le fichier `hwcore/paging.c` définit la structure des PDE (`struct x86_pde`) et celle des PTE (`struct x86_pte`) en reprenant littéralement les spécifications d'Intel [1, section 3.6.4].

La fonction `hwcore/paging.c:sos_paging_setup()` se charge de construire les tables de traduction d'adresses (établissement de l'identity mapping et du mirroring), puis d'activer la pagination.

3.2.1 Mise en place de l'identity mapping

La fonction `hwcore/paging.c:sos_paging_setup()` prend deux adresses en paramètre qui indiquent le début et la fin des adresses physiques de la zone noyau pour lesquelles on établit l'*identity mapping* (voir la section 2.5). Initialement on choisit en effet de ne pas *mapper* (*projeter*) à l'identique *toute* la mémoire RAM, mais seulement les pages pour lesquelles l'adresse linéaire doit être maintenue égale à l'adresse physique. Ceci correspond à 3 parties de la mémoire physique : la mémoire vidéo mappée en mémoire physique (sinon on ne peut plus rien afficher), le code et les données du noyau chargés par Grub et enfin le tableau des descripteurs de pages physiques (voir l'article 3).

C'est la fonction intermédiaire `hwcore/paging.c:paging_setup_map_helper(pd, ppage, vpage)` qui sera appelée avec `vpage == ppage` pour établir l'identity-mapping sur chaque page de ces 3 parties :

```
static sos_ret_t paging_setup_map_helper(struct x86_pde * pd,
                                         sos_paddr_t ppage,
                                         sos_vaddr_t vpage)
{
  /* Get the page directory entry and table entry index for this
     address */
  unsigned index_in_pd = virt_to_pd_index(vpage);
  unsigned index_in_pt = virt_to_pt_index(vpage);

  /* Make sure the page table was mapped */
  struct x86_pte * pt;
  if (pd[index_in_pd].present)
    {
      pt = (struct x86_pte*) (pd[index_in_pd].pt_paddr << 12);

      /* Indicate the PT has one more PTE */
      if (! pt[index_in_pt].present)
        sos_physmem_ref_physpage_at((sos_paddr_t)pt);
    }
  else
    {
      /* No : allocate a new one */
      pt = (struct x86_pte*) sos_physmem_ref_physpage_new(FALSE);
      pd[index_in_pd].present  = TRUE;
      pd[index_in_pd].pt_paddr = ((sos_paddr_t)pt) >> 12;
    }

  /* Actually map the page in the page table */
  pt[index_in_pt].present = 1;
  pt[index_in_pt].paddr   = ppage >> 12;

  return SOS_OK;
}
```

Les macros `virt_to_pd_index(vaddr)` et `virt_to_pt_index(vaddr)` permettent d'obtenir respectivement l'index de la PDE et l'index de la PTE associée à l'adresse `vaddr` (voir la section 2.2.3).

Trois commentaires à ce code :

- Cette fonction (`paging_setup_map_helper()`) a le droit d'accéder au répertoire des pages et aux tables des pages par leur adresse physique. Pour le moment en effet, la pagination n'est pas encore initialisée ;
- Au besoin, cette fonction alloue une table des pages si aucune n'est encore initialisée pour l'adresse `vpage` :
- On détourne de son objectif initial (quoique) le compteur de références (voir l'article 3) associé aux tables des pages. On l'utilise en effet pour compter le nombre de PTE qui sont utilisées dans la table des pages. Ceci permettra de libérer automatiquement la table des pages lorsqu'elle ne contiendra plus de PTE utilisée.

3.2.2 Mise en place du mirroring

Une fois le répertoire des pages mis en place pour l'identity mapping, la fonction `hwcore/paging.c:sos_paging_setup()` poursuit en préparant le mirroring (voir la section 3.1) :

```
/** Virtual address where the mirroring takes place */
#define SOS_PAGING_MIRROR_VADDR 0x3fc00000 /* 1GB - 4MB */

  [...]
  pd[virt_to_pd_index(SOS_PAGING_MIRROR_VADDR)].present = TRUE;
  pd[virt_to_pd_index(SOS_PAGING_MIRROR_VADDR)].pt_paddr
    = ((sos_paddr_t)pd)>>12;
  [...]
```

Autrement dit les 4Mo d'adresses linéaires situés à l'adresse linéaire `0x3fc00000` refléteront le contenu de toutes les tables des pages ainsi que celui du répertoire de pages (puisqu'il est considéré comme une table des pages particulière par le mirroring). Cette adresse `0x3fc00000` correspond à la PDE d'index 255 dans le répertoire des pages, *ie* $idxMirror$ = `virt_to_pd_index(SOS_PAGING_MIRROR_VADDR)` = 255.

3.2.3 Activation de la pagination

La fonction `hwcore/paging.c:sos_paging_setup()` termine en :

- Initialisant le registre `cr3` (voir la section 2.4) avec l'adresse physique du répertoire des pages `pd` ;
- Positionnant les bits 31 et 16 du registre `cr0` à 1 pour respectivement activer la pagination dans la MMU et conserver les droits d'accès lecture/écriture y compris pour les accès en mode superviseur sur les pages de type utilisateur [1, section 2.5].

3.2.4 Synthèse

Après les opérations précédentes, la pagination est configurée convenablement et activée conformément à la figure 10. Tout accès dans les zones grisées de la figure provoquera une exception processeur de type `Page Fault` [1, section 5.12].

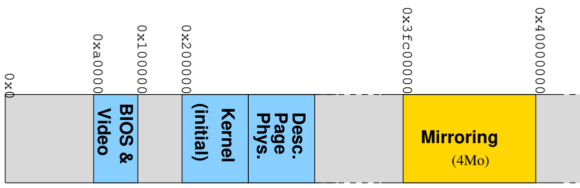


FIG. 10 – Configuration initiale des adresses linéaires dans SOS

Les autres fonctions de `hwcore/paging.c` permettent de modifier cette configuration par la suite, comme nous le décrivons ci-dessous.

3.3 Ajouter une traduction `adresse linéaire → adresse physique`

C'est la fonction `sos_paging_map(ppage_paddr, vpage_vaddr, is_user_page, flags)` qui permet d'ajouter la traduction `adresse linéaire vpage_vaddr → adresse physique ppage_paddr` dans les tables de traduction d'adresses.

Elle ressemble à la fonction `paging_setup_map_helper()` vue précédemment. La différence principale est qu'elle n'accède pas au répertoire et aux tables des pages par leur adresse physique, mais par leur adresse linéaire située dans les 4Mo du mirroring. Elle utilise pour cela les identités arithmétiques énoncées dans la section 3.1.3 :

```
sos_ret_t sos_paging_map(sos_paddr_t ppage_paddr,
                         sos_vaddr_t vpage_vaddr,
                         sos_bool_t is_user_page,
                         int flags)
{
  /* Get the page directory entry and table entry index
     for this address */
  unsigned index_in_pd = virt_to_pd_index(vpage_vaddr);
  unsigned index_in_pt = virt_to_pt_index(vpage_vaddr);

  /* Get the PD of the current context */
  struct x86_pde *pd = (struct x86_pde*)
    (SOS_PAGING_MIRROR_VADDR
     + SOS_PAGE_SIZE*virt_to_pd_index(SOS_PAGING_MIRROR_VADDR));

  /* Address of the PT in the mirroring */
  struct x86_pte * pt
    = (struct x86_pte*) (SOS_PAGING_MIRROR_VADDR
                         + SOS_PAGE_SIZE*index_in_pd);

  /* Map a page for the PT if necessary */
  if (! pd[index_in_pd].present)
    {
      /* No : allocate a new one */
      sos_paddr_t pt_ppage
        = sos_physmem_ref_physpage_new(! (flags & SOS_VM_MAP_ATOMIC));

      pd[index_in_pd].present  = TRUE;
      pd[index_in_pd].pt_paddr = ((sos_paddr_t)pt_ppage) >> 12;

      /* Invalidate TLB for the page we just added */
      invlpg(pt);

      /* Reset this new PT */
      memset((void*)pt, 0x0, SOS_PAGE_SIZE);
    }

  /* If we allocate a new entry in the PT, increase its reference
     count. */
  else if (! pt[index_in_pt].present)
    sos_physmem_ref_physpage_at(pd[index_in_pd].pt_paddr << 12);

  /* Otherwise, that means that a physical page is implicitely
     unmapped */
  else
    sos_physmem_unref_physpage(pt[index_in_pt].paddr << 12);

  /* Map the page in the page table */
  pt[index_in_pt].present = TRUE;
  pt[index_in_pt].paddr   = ppage_paddr >> 12;
  sos_physmem_ref_physpage_at(ppage_paddr);

  /*
   * The page is now mapped in the current address space
   */

  /* Invalidate TLB for the page we just added */
  invlpg(vpage_vaddr);

  return SOS_OK;
}
```

Trois commentaires sont à apporter à cet extrait de code (il ne s'agit pas de la version complète) :

- Dès qu'une entrée quelconque du répertoire des pages ou de la table des pages concernée par la nouvelle traduction est modifiée, on prend soin d'utiliser la macro `invlpg(vaddr)` pour enlever du TLB toute ancienne traduction concernant l'adresse linéaire `vaddr` (voir la section 2.6).
- Si l'ajout de cette traduction ne fait que remplacer une traduction existante, la page physique qui était la destination de cette précédente traduction est libérée par appel de la fonction `sos_physmem_unref_physpage(paddr)`.
- Le paramètre `flags` correspond à un champ de bits qui sert à contrôler le comportement de la fonction.

Le paramètre `flags` peut prendre pour valeur une combinaison arbitraire (établie par des "ou" logiques) des valeurs suivantes (définies dans `hwcore/paging.h`) :

`SOS_VM_MAP_ATOMIC` : permet de contrôler le comportement de la fonction `sos_physmem_ref_physpage_new()` appelée lorsqu'une nouvelle page physique est nécessaire pour stocker une nouvelle table des pages. Ce paramètre permet d'indiquer si tout blocage est interdit lors de l'appel à la fonction. Il a à voir avec le *swap* et le *file mapping* dont nous reparlerons beaucoup plus tard ;

`SOS_VM_MAP_PROT_READ` : le processeur a le droit de lire dans la page située à l'adresse linéaire `vpage_vaddr`. Sur `x86`, dès qu'une traduction est ajoutée dans les tables des pages, ce droit est systématiquement octroyé ;

`SOS_VM_MAP_PROT_WRITE` : le processeur a le droit d'écrire dans la page située à l'adresse linéaire `vpage_vaddr` ;

`SOS_VM_MAP_PROT_EXEC` : le processeur a le droit d'exécuter des instructions dans la page située à l'adresse linéaire `vpage_vaddr`. Sur `x86`, dès qu'une traduction est ajoutée dans les tables des pages, ce droit est systématiquement octroyé.

Pour des raisons d'intérêt et de place, dans l'extrait de code ci-dessus (mais pas dans le vrai code), nous avons omis la prise en charge du paramètre `is_user_page`. Ce booléen indique si la page située à l'adresse linéaire `vpage_vaddr` est accessible uniquement lorsque le processeur est en mode superviseur (valeur : `FALSE`), ou si elle est accessible également s'il est en mode utilisateur (valeur : `TRUE`).

3.4 Enlever une traduction `adresse linéaire → adresse physique`

C'est la fonction `sos_paging_unmap(vpage_vaddr)` qui permet d'enlever la traduction `adresse linéaire vpage_vaddr → page physique associée` dans les tables de traduction d'adresses. Elle s'occupe de :

1. décrémenter le compteur de références de la page physique associée à la traduction. Ceci entraîne la libération de la page si ce compteur passe à 0 (voir l'article 3) ;
2. réinitialiser la PTE concernée pour signifier que plus aucune page physique n'y est associée ;
3. décrémenter le compteur de références associé à la table des pages, puisque celui-ci sert aussi à compter le nombre de PTE utilisées ;
4. si la table des pages n'est plus nécessaire, c'est-à-dire si son compteur de références est passé à

0, c'est-à-dire si la table des pages a été libérée par `sos/physmem.c`, alors il faut signifier au répertoire des pages que la PDE associée ne référence plus cette table des pages.

Si aucune page physique n'est associée à l'adresse linéaire `vpage_vaddr`, cette fonction ne fait rien.

Le code suivant implante cet algorithme :

```
sos_ret_t sos_paging_unmap(sos_vaddr_t vpage_vaddr)
{
  sos_ret_t pt_unref_retval;

  /* Get the page directory entry and table entry index for this
     address */
  unsigned index_in_pd = virt_to_pd_index(vpage_vaddr);
  unsigned index_in_pt = virt_to_pt_index(vpage_vaddr);

  /* Get the PD of the current context */
  struct x86_pde *pd = (struct x86_pde*)
    (SOS_PAGING_MIRROR_VADDR
     + SOS_PAGE_SIZE*virt_to_pd_index(SOS_PAGING_MIRROR_VADDR));

  /* Address of the PT in the mirroring */
  struct x86_pte * pt
    = (struct x86_pte*) (SOS_PAGING_MIRROR_VADDR
                         + SOS_PAGE_SIZE*index_in_pd);

  /* Reclaim the physical page */
  sos_physmem_unref_physpage(pt[index_in_pt].paddr << 12);

  /* Unmap the page in the page table */
  memset(pt + index_in_pt, 0x0, sizeof(struct x86_pte));

  /* Invalidate TLB for the page we just unmapped */
  invlpg(vpage_vaddr);

  /* Reclaim this entry in the PT, which may free the PT */
  pt_unref_retval
    = sos_physmem_unref_physpage(pd[index_in_pd].pt_paddr << 12);
  if (pt_unref_retval > 0)
    /* If the PT is now completely unused... */
    {
      /* Release the PDE */
      memset(pd + index_in_pd, 0x0, sizeof(struct x86_pde));

      /* Update the TLB */
      invlpg(pt);
    }

  return SOS_OK;
}
```

Encore une fois, dès qu'une PDE ou une PTE est modifiée, la fonction veille à invalider l'entrée du TLB concernée (appel à `invlpg`).

3.5 Remarques sur l'utilisation de ces 2 fonctions

Un petit exemple pour illustrer. Pour allouer une page physique et pouvoir y accéder en lecture seule par l'adresse linéaire `0x1000`, il faut faire les 2 étapes qui suivent :

1. `ppage_new = sos_physmem_ref_physpage_new()`
2. `sos_paging_map(ppage_new, 0x1000, FALSE, 0)`

La première étape récupère une page physique marquée libre et fixe le compteur de références associé à cette page `ppage_new` à 1. La deuxième étape le passe à 2.

Pour libérer définitivement la page il faudra faire les 2 opérations symétriques : `sos_paging_unmap(0x1000)` et `sos_physmem_unref_physpage(ppage_new)`. Aucun ordre pour ces deux opérations n'est imposé. Donc on a le choix entre faire (*ref_cnt* étant la valeur du compteur de références associé à la page physique d'adresse `ppage_new`) :

1. `ppage_new = sos_physmem_ref_physpage_new()` $\Rightarrow ref\_cnt = 1$
2. `sos_paging_map(ppage_new, 0x1000, FALSE, 0)` $\Rightarrow ref\_cnt = 2$
3. ***Utilisation de la page à l'adresse linéaire 0x1000***
4. `sos_physmem_unref_physpage(ppage_new)` $\Rightarrow ref\_cnt = 1$
5. `sos_paging_unmap(0x1000)` $\Rightarrow ref\_cnt = 0$, la page est libérée.

Ou faire :

1. `ppage_new = sos_physmem_ref_physpage_new()` $\Rightarrow ref\_cnt = 1$
2. `sos_paging_map(ppage_new, 0x1000, FALSE, 0)` $\Rightarrow ref\_cnt = 2$
3. `sos_physmem_unref_physpage(ppage_new)` $\Rightarrow ref\_cnt = 1$
4. ***Utilisation de la page à l'adresse linéaire 0x1000***
5. `sos_paging_unmap(0x1000)` $\Rightarrow ref\_cnt = 0$, la page est libérée.

La plupart du temps, on utilisera la deuxième manière de procéder puisqu'elle dispense de conserver ou de récupérer l'adresse physique `ppage_new`.

3.6 Autres fonctions de `hwcore/paging.h`

La fonction `sos_paging_get_paddr(vaddr)` renvoie l'adresse physique correspondant à l'adresse linéaire `vaddr`, ou `NULL` si aucune traduction n'est associée. Elle parcourt pour cela les entrées du répertoire des pages et de la table des pages associée en utilisant le mirroring. Elle fait donc l'équivalent (avec des adresses linéaires dans le mirroring) de ce que fait la MMU (avec des adresses physiques).

La fonction `sos_paging_get_prot(vaddr)` renvoie les droits d'accès associés à la page couvrant l'adresse linéaire `vaddr`. Á partir de la PDE et de la PTE associée à la page couvrant `vaddr`, elle construit un champ de bits formé de la combinaison ("ou" logique) des macros `SOS_VM_MAP_PROT_*` vues précédemment. Elle fonctionne d'une manière analogue à la fonction `sos_paging_get_paddr(vaddr)`.

4 Testons !

Pour cet article, la petite démo que nous proposons est très courte et assez "amusante". Nous allons changer au vol l'emplacement physique des instructions que nous sommes en train d'exécuter. Nous testerons aussi le déclenchement de l'exception "`Page Fault`".

Avant de présenter le contenu de cette démo, la première étape est d'initialiser et d'activer la pagination :

```
if (sos_paging_setup(sos_kernel_core_base_paddr,
                     sos_kernel_core_top_paddr))
  sos_bochs_printf("Could not setup paged memory mode\n");
sos_x86_videomem_printf(2, 0,
                        SOS_X86_VIDEO_FG_YELLOW
                        | SOS_X86_VIDEO_BG_BLUE,
                        "Paged-memory mode is activated");
```

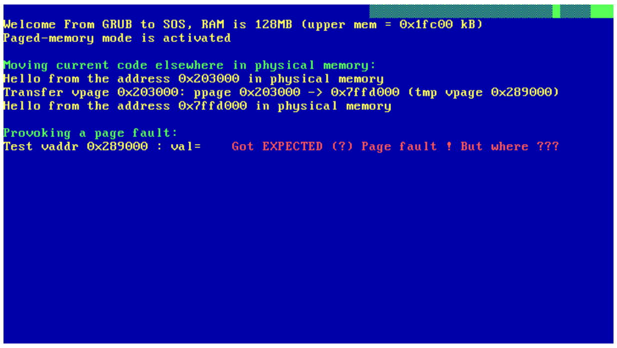
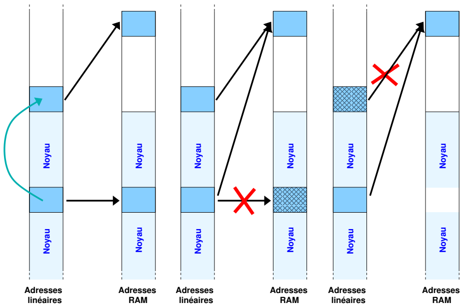


FIG. 11 – Aperçu de la petite démo



(a) Étape 3 (b) Étape 4 (c) Étape 5

FIG. 12 – Déplacement de code en mémoire physique

4.1 Déplacement “au vol” en mémoire physique de la page de code actuelle

Pour déplacer au vol la page physique du code que le processeur est en train d’exécuter, on procède en 5 étapes (voir figure 12) :

1. Récupération de l’adresse linéaire de la page de code que nous sommes en train d’exécuter : `vpage_code`;
2. Allocation d’une nouvelle page physique `ppage_new`;
3. Ajout d’une traduction traduction `vpage_tmp` → `ppage_new` et copie du contenu de la page de code `vpage_code` vers la page `vpage_tmp`. `vpage_tmp` peut être une adresse linéaire inutilisée quelconque (voir ci-dessous) ;
4. Modification de la traduction pour l’adresse `vpage_code` → `ppage_new`. La page située à l’adresse linéaire `vpage_code` ne pointe donc plus vers la même adresse physique, autrement dit le processeur exécute des instructions qui ne se situent pas sur la même page physique qu’en 1, 2 et 3. Mais le contenu de la page, c’est-à-dire les instructions qu’il exécute, est le même puisqu’il a été copié en 3 ;
5. Suppression de la traduction temporaire `vpage_tmp` → `ppage_new` puisque la nouvelle page physique `ppage_new` est maintenant associée à l’adresse linéaire `vpage_code`.

Le code de la démo `sos/main.c:test_paging()` fait exactement ces 5 opérations :

```
static void test_paging(sos_vaddr_t sos_kernel_core_top_vaddr)
{
  sos_vaddr_t vpage_code = SOS_PAGE_ALIGN_INF(test_paging);
  sos_paddr_t ppage_new;
  sos_vaddr_t vpage_tmp = sos_kernel_core_top_vaddr;

  ppage_new = sos_physmem_ref_physpage_new(FALSE);
  sos_paging_map(ppage_new, vpage_tmp,
                 FALSE,
                 SOS_VM_MAP_ATOMIC
                 | SOS_VM_MAP_PROT_READ
                 | SOS_VM_MAP_PROT_WRITE);
  sos_physmem_unref_physpage(ppage_new);

  memcpy((void*)vpage_tmp,
         (void*)vpage_code,
         SOS_PAGE_SIZE);

  sos_paging_map(ppage_new, vpage_code,
                 FALSE,
                 SOS_VM_MAP_ATOMIC
                 | SOS_VM_MAP_PROT_READ
                 | SOS_VM_MAP_PROT_WRITE);

  /* We can safely unmap it from sos_kernel_core_top_vaddr
     while still keeping the vpage_code mapping */
  sos_paging_unmap(vpage_tmp);

[...]
```

Nous avons choisi d’affecter à l’adresse linéaire `vpage_tmp` l’adresse linéaire qui suit immédiatement le code et les données du noyau chargé par Grub (`sos_kernel_core_top_vaddr`). Nous aurions pu l’affecter à n’importe quelle adresse linéaire pourvu qu’elle ne se situe dans aucune zone de mémoire virtuelle déjà utilisée (BIOS/Video, code/données du noyau, 4Mo du mirroring).

4.2 Déclenchement de l’exception “`Page Fault`”

La petite démo comporte une sixième étape pour montrer ce qui se passe lorsqu’on accède à une adresse linéaire dont les tables de traduction n’indiquent pas la traduction. Nous avons choisi de parcourir une partie des adresses linéaires utilisées par le noyau jusqu’à ce qu’on sorte de cette zone, ce qui provoquera l’exception `Page Fault` (voir la section 2.2.1) :

```
  for (i = vpage_code ; /* none */ ; i += SOS_PAGE_SIZE)
    {
      unsigned *pint = (unsigned *)SOS_PAGE_ALIGN_INF(i);
      sos_bochs_printf("Test vaddr 0x%x : val=", (unsigned)pint);
      sos_x86_videomem_printf(10, 0,
                              SOS_X86_VIDEO_FG_YELLOW
                              | SOS_X86_VIDEO_BG_BLUE,
                              "Test vaddr 0x%x : val=      ",
                              (unsigned)pint);
      /* Page Fault should occur HERE: */
```

```
    sos_bochs_printf("0x%x\n", *pint);
    sos_x86_videomem_printf(10, 30,
                            SOS_X86_VIDEO_FG_YELLOW
                            | SOS_X86_VIDEO_BG_BLUE,
                            "0x%x          ", *pint);
  }
```

Afin que le début du parcours ne provoque pas d'exception, nous choisissons de le faire commencer à l'adresse de la page de code qu'on est en train d'exécuter (`vpage_code`). La zone du noyau s'arrêtant juste avant l'adresse linéaire `sos_kernel_core_top_vaddr`, l'exception devrait être levée en accédant à la page située à cette adresse. On remarquera que cette adresse correspond aussi à l'adresse `vpage_tmp` qui nous avait servi au point 3 ci-dessus, mais y accéder maintenant provoque l'exception car la traduction associée avait été enlevée au point 5.

Malheureusement, la routine de traitement de l'exception n'affichera ni l'adresse linéaire de l'instruction fautive, ni l'adresse linéaire accédée et qui a provoqué l'exception. Il faudra attendre l'article 6 pour cela. Le handler se contente donc d'afficher un message et de terminer par une boucle infinie :

```
/* Page fault exception handler */
static void pgflt_ex(int exid)
{
  sos_bochs_printf("Got page fault\n");
  sos_x86_videomem_printf(10, 30,
                          SOS_X86_VIDEO_FG_LTRED
                          | SOS_X86_VIDEO_BG_BLUE,
                          "Got EXPECTED (?) Page fault ! But where ???");
  for (;;) ;
}
```

5 Et dans Linux, comment ça marche ?

5.1 Configuration générale de l'espace des adresses linéaires

Dans le noyau Linux, les applications disposent des adresses virtuelles `0 - PAGE_OFFSET` pour s'exécuter (le processeur est alors en mode "utilisateur", voir l'article 2). Au delà de `PAGE_OFFSET` se trouve la partie réservée au code et aux données du noyau (*ie* réservée au mode "superviseur" du processeur). Sur `x86`, `PAGE_OFFSET` vaut `3Go` en général.

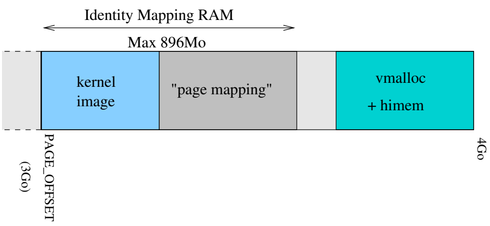


FIG. 13 – Configuration des adresses linéaires dans Linux

Cette partie "noyau" est principalement découpée en 2 zones (voir la figure 13) : le reflet de la mémoire physique et la zone dite `vmalloc`. Le reflet de la mémoire physique possède les propriétés suivantes :

1. La totalité de la mémoire physique, à concurrence de 896 Mo, est accessible dans cette zone.
2. Pour accéder à l'adresse physique pa, il suffit d'accéder à l'adresse linéaire $va = PAGE\_OFFSET + pa$. Dans cette zone, il y a donc une translation constante entre adresses linéaires et adresses physiques.

Cette zone est donc un *identity mapping* (à une translation près) du code et des données originaux du noyau, mais également du reste de la mémoire physique. Cette manière de procéder permet de se dispenser du mirroring puisque le noyau peut manipuler indifféremment (modulo une translation) des adresses physiques et des adresses linéaires. Cela est parfois utile quand on dialogue avec des périphériques matériels auxquels on ne doit fournir que des adresses physiques.

En contrepartie, cela possède deux inconvénients. D'une part la taille de RAM maximale "gérable" simplement par le noyau est limitée à 896 Mo. Au-delà, il faut soit accorder une place plus grande pour l'espace des adresses linéaires réservées au noyau au détriment de l'espace disponible pour les applications (*ie* modifier `PAGE_OFFSET`), soit utiliser les techniques de relais de pages (*bounce buffers*) activées par l'option de configuration `CONFIG_HIMEM` du noyau.

D'autre part, le noyau repose sur des valeurs d'adresses qui correspondent aux adresses physiques (modulo une translation) pour les données et du code qu'il manipule. Ceci empêche de déplacer au vol des pages utilisées par le noyau. En apparence ceci a peu d'impact. Mais en pratique cela peut être gênant car certains périphériques demandent à dialoguer avec l'OS uniquement dans une plage d'adresses physiques bien précise. Par exemple, les accès DMA sur bus ISA ne sont possibles que pour les adresses physiques inférieures à 16 Mo. Or, si du code ou des données du noyau sont dans ces plages d'adresses, celles-ci ne peuvent pas être déplacées ailleurs. Pour cette raison, le noyau définit un allocateur (de type *Buddy System*) de mémoire physique qui distingue des zones de mémoire physique avec des priorités associées afin de limiter les données allouées dans ces plages sensibles, comme nous l'avions vu dans l'article 3.

Quant à la zone `vmalloc`, elle sert aux allocations les plus courantes pour lesquelles une correspondance simple avec les adresses physiques n'est pas nécessaire. L'allocateur pour cette zone est un allocateur de type *Slab* analogue à celui que nous présenterons dans le prochain article. Il travaille avec des adresses linéaires pures, exactement de la même manière que SOS.

5.2 API pour la gestion des tables de traduction

Linux est un noyau qui supporte quelques 20 architectures... Il définit une interface de programmation (API) commune, non pas pour gérer l'espace des adresses linéaires comme nous le faisons dans SOS, mais pour gérer les tables de traduction sous-jacentes d'une manière uniforme.

Cette API commune exporte des fonctions ou des macros pour manipuler des tables de traduction d'adresses à 3 niveaux d'indirection. Sans rentrer dans les détails, nous dirons simplement que cela est très bien adapté à tous les types d'adressage possibles de l'architecture `x86` (32 bits) car on a une correspondance directe avec les caractéristiques de la MMU. Pour les autres architectures, cette couche d'abstraction peut devenir plus "*h-ardue*" à implanter.

Conclusion

Nous venons de présenter les principes de base de la mémoire virtuelle et de proposer une implantation du premier niveau de cette technique : la gestion des tables de traduction d'adresses pour la pagination sur `x86`. Cette implantation a mis l'accent sur la mise en place de l'identity mapping pour faciliter la mise en place de la pagination et sur le mirroring pour permettre et automatiser la modification des tables de traduction d'adresses. En fin de série, nous verrons l'implantation du deuxième niveau de cette technique : le *swap* ou le *file mapping*.

Il est maintenant temps de simplifier notre terminologie pour les articles à venir. D'une part, dans cet article, nous avons parlé explicitement d'adresses linéaires, logiques, etc... Dans SOS, adresses linéaires et logiques sont égales, donc dans le code et dans les articles qui suivront, nous ne ferons plus cette distinction. Nous emploierons le terme générique d'adresse "*virtuelle*" à la place. Ce terme remplacera à l'avenir aussi celui d'adresse "fictive" d'usage peu courant en informatique.

D'autre part, nous avons parlé explicitement d"'ajouter" ou d"'enlever" des traductions d'adresses dans les tables de traduction (sections 3.3 et 3.4). Dans les articles qui suivent, nous adopterons le vocabulaire courant du domaine et dirons respectivement "mapper" et "unmapper" (ou *démapper*) telle page située à telle adresse "virtuelle".

Voilà, cette quatrième étape de notre croisière tire à sa fin. Il reste encore quelques digressions "pour la bonne bouche" en annexe. Cette partie de croisière n'a sûrement pas été une partie de plaisir, nous espérons seulement qu'elle n'aura pas été trop indigeste. Et ce n'est pas fini ! Dans le prochain épisode de la saga, nous parlerons de l'allocateur d'objets pour le noyau, de type *Slab*. Certes, cette prochaine étape oubliera les aspects matériels abondants ici. Mais elle sera fournie en algorithmes où les considérations de récursivité occuperont une bonne place.

A Autres types de pagination

Avec la pagination, on se heurte à l'un des problèmes les plus classiques en algorithmique : comment retrouver une donnée (ici : une adresse physique) associée à une valeur (ici : une adresse fictive), sachant que l'espace des valeurs (ici : l'espace des adresses fictives) est très vaste. La contrainte principale étant de le faire de manière efficace en temps et sans consommer trop d'espace mémoire (en RAM ou dans la MMU).

En matière de gestion de la mémoire virtuelle, plusieurs réponses ont été apportées par les fabriquants de MMU ou de processeurs. Nous allons en évoquer quelques unes sans prétendre être exhaustifs [4, 5]. Toutes font l'observation que les applications et l'OS n'utilisent pas l'intégralité des adresses fictives, mais seulement des portions de cet espace. Par exemple, un programme occupera souvent de l'ordre de 5 morceaux (ensembles de pages contiguës) dans l'espace des adresses fictives : un morceau pour le code, un autre pour les données, un autre pour la pile, un autre pour les données allouées dynamiquement, un autre pour la bibliothèque C ; les "trous" entre ces morceaux n'étant associés à aucune adresse en RAM.

A.1 Pagination à plusieurs niveaux

Une technique est de répartir la traduction entre plusieurs tables par un système d'indirections à 1 ou plusieurs niveaux. C'est la technique que nous venons de voir (voir la section 2) et qui est adoptée dans l'architecture `x86` et `amd64/ia32e`. Elle permet de ne stocker en mémoire que les tables de traduction d'un niveau donné qui couvrent au moins une page physique (en RAM ou sur le disque ou sur tout autre moyen de stockage supporté par le système d'exploitation).

Cette technique assure une diminution remarquable de la consommation mémoire par les tables de traduction d'adresses en RAM. Mais cette consommation est proportionnelle à la quantité d'adresses fictives utilisées par les applications et l'OS.

A.2 Pagination assistée par le système

Le principe de cette technique est simple : on ne définit aucune table de traduction d'adresses pour l'espace fictif entier. Cela revient à laisser le système d'exploitation effectuer toutes les étapes de traduction d'adresses (rappel : une exception est levée lorsqu'une traduction d'adresse est nécessaire). Pour éviter trop d'appels au système, la MMU dispose d'un TLB (voir la section 2.2.4) qui renferme quelques traductions d'adresses. Ce TLB n'est pas *seulement* une optimisation : il est **la** pièce fondamentale. Il faut en effet qu'il contienne au moins les traductions d'adresses utiles à l'OS pour fonctionner et effectuer les traductions demandées, sinon une autre exception est levée par la MMU avant que l'exception précédente n'ait un espoir d'être traitée complètement par l'OS...

Le principal problème de cette technique est que la traduction d'adresses peut introduire une latence importante (appels fréquents au système d'exploitation). Son intérêt principal est que l'OS peut optimiser la manière dont il stocke les traductions : pour une zone d'identity mapping par exemple, il suffit de ne retenir qu'une seule traduction pour toute la zone.

A.3 Table des pages inverse

Cette technique part du principe que l'espace des adresses fictives (par exemple sur 64 bits) est en général largement plus grand que l'espace des adresses en RAM (par exemple sur 30 bits). Dans ces conditions il est plus économique de stocker la traduction (dite "inverse") `adresse en RAM → adresse fictive`.

Le premier problème est que, par définition, la MMU doit quand même effectuer la traduction dans le sens opposé (`adresse fictive → adresse physique`) puisque le processeur travaille avec des adresses fictives. Or chacun sait que la recherche d'un élément dans un tableau non ordonné est lente, surtout quand le tableau devient grand. Ici, il s'agit en effet de rechercher une adresse fictive dans un tableau indéxé par les adresses en RAM. Pour résoudre ce problème, la MMU utilise une table de hachage pour accélérer la recherche, et/ou un cache de traduction d'adresses pour stocker les traductions des adresses fictives les plus fréquemment demandées.

Outre les latences de traduction qui peuvent être relativement importantes, cette méthode nécessite de prendre quelques précautions afin d'être utilisée pour la multi-programmation (une page en RAM appartenant à une application A ne doit pas être accessible par une autre application B). Son intérêt principal est que la quantité de RAM maximale consommée pour stocker les données propres à la MMU reste raisonnable.

B Différences entre segmentation et pagination

Pagination et segmentation sont deux techniques de traduction d'adresses classiques pour implanter la mémoire virtuelle.

Dans un système segmenté, on peut définir un nombre limité de segments de tailles quelconques, plus petit que le nombre de pages possibles dans un système paginé. Il en découle que les segments auront en général une taille plus grande que la taille d'une page. Ceci aura plusieurs conséquences :

- La RAM doit être suffisamment grande pour contenir au moins chaque segment défini dans le système.
- Si la MMU établit qu'une adresse fictive n'est pas en RAM et que l'OS constate qu'il doit aller chercher les données sur disque, alors l'opération de lecture risque d'être longue (par exemple si le segment contient le code de mozilla), même si quelques kilo-octets seulement d'un segment beaucoup plus long sont vraiment nécessaires.
- Les segments occupent en RAM des adresses physiques contiguës et ne sont pas tous de même taille. Ceci risque de conduire à une fragmentation de l'occupation de la mémoire physique : au gré des allocations/libérations de RAM pour contenir les segments, les espaces de RAM entre les segments finiront par être trop petits pour contenir d'autres segments.

À l'inverse, pour les systèmes à base de pagination :

- Il suffit que la RAM puisse contenir au moins une page.
- Si la MMU établit qu'une adresse fictive n'est pas en RAM et que l'OS constate qu'il doit aller chercher les données sur disque, alors l'opération de lecture restera raisonnable (quelques kilo-octets en général) et sera plus "rentable" : la proportion `taille de donnée utile / taille de la page` reste importante.
- Les pages sont toutes de la même taille (dans le cas le plus courant), ce qui garantit qu'aucune fragmentation de la RAM n'est possible.

David Decotigny et Thomas Petazzoni
`Thomas.Petazzoni@enix.org` et `d2@enix.org`

Merci à Nessie pour sa relecture, ses remarques constructives et ses propositions.

Fan Club de Fabrice Bellard : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`
Á Wolfgang Amadeus

Références

[1] Intel Corp. Intel architecture developer's manual, vol 3, 1997.

[2] AMD Corp. AMD64 architecture programmer's manual volume 2 : System programming, Sept 2003.

[3] Intel Corp. Intel extended memory 64 technology software developer's guide, vol 1, 2004.

[4] Andrew Tanenbaum. *Systémes d'exploitation*. Number ISBN 2100045547. InterEditions / Prentice Hall, 1994.

[5] Uresh Vahalia. *UNIX Internals : The New Frontiers*. Number ISSN 0131019082. Prentice Hall, 1995.