TE

Clément MATHIEU <cykl@mAdchAt.org>

Guillaume ERETEO <ereteog@clio.unice.fr>

DRAFT v1.0 16/06/2004

Ce document à été produit dans le cadre d'un travail d'étude au cours de la licence d'informatique 2003/2004 de l'UNSA et présente les concepts et méchanismes fondamentaux de la gestion mémoire dans les systèmes d'exploitation. Les questions actuelle et les évolutions en cours sont aussi abordées afin de montrer un panorama général et actuel de ce domaine.

Table of Contents

1. Introduction 1
2. Généralités 2
    2.1. Historique 2
    2.2. Avantages de la mémoire virtuelle 3
    2.3. Pagination 3
    2.4. Segmentation 7
    2.5. Gestion de la zone d'échange 8
    2.6. Séparation en deux de l'espace d'adressage 9
    2.7. Hiérarchie de mémoires caches 9
    2.8. Exhausion de l'espace d'adressage 11
    2.9. L'évolution du matériel 12
3. BSD 13
4. Linux 13
    4.1. Modèle de pagination 13
    4.2. Espace d'adressage d'un processus 14
    4.3. Gestion des défauts de pages 18
    4.4. Allocateurs mémoire 22
5. Sujets d'actualité 23
    5.1. Chez Linux 23
    5.2. Support NX, No eXecute, par page 28
Glossaire 31
bibliographie 33
A. Comparaisons de différents processeurs 36

1. Introduction

Ce document présente un panorama général de la gestion mémoire, souvent appelée VMM (virtual memory management), MM ou VM, ainsi que l'état actuel des implémentations notamment dans les systèmes de type UNIX.

La première partie sera consacrée à présenter les buts et les principes généraux de la gestion mémoire et ne nécéssite pas de connaissance préalable du domaine. Toutefois une connaissance minimale des principes et du fonctionnement des systèmes d'exploitation peut s'avérer utile pour comprendre le tenant et les aboutissants des concepts présentés. Cette première partie présentera aussi le support offert par le matériel pour la gestion mémoire notamment par le biais de l'architecture la plus répendue actuellement le X86 mais surtout grace à son successeur : l'AMD64

Une seconde partie présentera rapidement l'implémentation des principes présentés grâce à une vue générale de ce qui a été réalisé dans Linux. Il ne s'agit pas d'une description en profondeur ou d'un commentaire de code cette tâche ayant déjà effectuée avec brio par deux fois par des gens bien plus

compétents.

Enfin la dernière partie présentera quelques domaines actifs de la gestion mémoire. Et fera un zoom sur quelques débats et problèmes récents rencontrés dans les système UNIX.

2. Généralités

Les systèmes d'exploitation ont pour mission de fournir une interface avec le matériel aux logiciels voulant s'exécuter sur une plate-forme donnée. Au cours de l'histoire les noyaux n'ont cessé d'évoluer et leurs fonctionnalités n'ont cessé de se modifier en suivant un lien assez étroit avec l'évolution du matériel. De systèmes monoprogrammés sans sécurité nous sommes passés en quelques années à des systèmes multiprogrammés permettant une forte isolation entre les processus.

Ainsi au cours du temps les systèmes d'exploitation en sont venus à fournir une abstraction matérielle très élevée, les processus n'ont plus (ne peuvent) à interagir directement avec tout une partie du matériel mais utilise la couche d'abstraction qu'est le système d'exploitation. Cette couche d'abstraction nécessite que le système fournisse des primitives pour répondre aux besoins des processus. Un des besoins fondamentaux est la mémoire. Tout programme pour s'exécuter à besoin de mémoire pour contenir son code et ses données. Nous allons nous intéresser à comment un système d'exploitation gère la mémoire physique disponible pour l'utiliser et permettre aux processus de l'utiliser. Pour cela nous allons commencer par faire un bref rappel historique retraçant les différentes grandes étapes de la gestion mémoire.

2.1. Historique

Les premiers systèmes n'offraient que très peu d'abstraction avec le matériel et les programmes étaient souvent développés avec une très bonne connaissance du matériel caché derrière. Nous sommes au temps des systèmes monoprogrammés; un seul processus peut s'exécuter à la fois. Un des problèmes majeurs est que la mémoire principale fait bien souvent défaut; un programme et ses données ne peuvent tenir entièrement dedans, le système n'offrant aucune aide. Le processus est alors fortement couplé à la machine sur lequel il est développé. En effet le programmeur doit s'occuper lui même de découper son programme en segments (modules) indépendants et de mettre en place le mécanisme permettant le replacement de ces segments de l'exécution. Cette façon de procéder à beaucoup d'inconvénients en voici quelques uns. Tout d'abord il faut que tout les segments soient plus petits que la mémoire principale disponible, ainsi le programme est développé pour une machine donnée; le découpage en segments complexifie beaucoup le développement puisque les structures de données et les algorithmes doivent être pensés pour ne jamais faire de référence à des données qui ne sont pas dans le segment actuel et surtout chaque programme implémente sa propre façon de charger ses données et de les sauvegarder dans la mémoire de masse.

En plus du caractère fastidieux de la chose on s'aperçoit que tout les programmes réinventent la roue et le résultat final n'est absolument pas portable d'une machine à l'autre. Comble de la chose le programme résultant est deux à trois fois plus gros à cause de la gestion des segments ! Nous sommes alors dans les années 50 et les concepteurs des premiers systèmes d'exploitation rêvent d'une chose : automatiser la gestion de la mémoire pour que les programmes n'aient plus à gérer le *swap in* et le *swap out* des segments.

Ce rêve est réalisé quelques années plus tard au début des années 60 et le résultat se nomme *mémoire virtuelle*; la première implémentation date de 1959 par l'université de Manchester. Ce qui correspond à la mémoire virtuelle est appelé "one-level storage system" l'idée derrière ce nom est simple mais très efficace et consiste en trois points:

- Construire un matériel permettant de transformer une adresse passée par le processeur en la position actuelle des données correspondant à cette adresse.
- Ajouter une mécanisme de *pagination à la demande*. Lorsque le matériel indique que l'adresse demandée n'est pas en mémoire principale ce mécanisme permet charger les données manquantes, par exemple dans la zone d'échange ou encore non chargées. Si la mémoire principale est saturée il faut aussi déporter des pages vers la zone d'échange.

- Développer un *algorithme de remplacement* qui permet de choisir la zone mémoire à sélectionner pour la déporter sur le disque.

Derrière ces trois concepts simples se cache un idée incroyablement puissante; les adresses manipulées par les programmes ne décrivent plus l'emplacement physique des informations mais leur emplacement logique qui est géré par une coopération du système d'exploitation et du matériel. La mémoire est devenue virtuelle pour le processus !

Cette voie sera fortement étudiée par le monde universitaire de 1965 à 1975. De nombreux systèmes reprennent ce principe, les fondeurs de puces incorporent la gestion mémoire dans les processeurs et le milieu de la recherche s'occupe de trouver des algorithmes de remplacement ainsi que de résoudre diverses problème comme le *trashing* tout en étudiant le *principe de localité.*

Dans le même temps les systèmes multiprogrammés font leur apparition et permettent d'exécuter plusieurs processus de manière pseudo concurrente ou concurrente selon les caractéristiques du matériel. Certain systèmes sont de type coopératif (on retrouve des systèmes de ce type jusqu'à assez récemment par exemple avec les Mac OS 7,8,9) ou premptif. Les systèmes multiprogrammés reposent tous sur l'abstraction introduite par la mémoire virtuelle et selon l'usage qui en est fait une réelle isolation des différents processus est atteinte.

Jusqu'à assez récemment les systèmes grand publique étaient incapables d'utiliser correctement les notions introduites des années plus tôt (Win 9x et Mac OS par ne citer qu'eux). Ainsi aucune isolation des espaces d'adressage des différents processus n'était réalisée ce qui pose de graves problèmes de sécurité et de stabilité. Toutefois des années plus tôt ces problèmes étaient parfaitement maîtrisés par de nombreux systèmes, ainsi MULTICS était doté d'une des gestions mémoire les plus évoluées jamais implémentée; les systèmes post MULTICS tel que les UNIX en reprenant généralement les concepts de manière simplifié.

De nos jours les systèmes courants sont de types multitaches préemptifs on peut ainsi noter Linux, *BSD, Solaris, Mac OS X, Win 2k/XP. Sur ces systèmes l'espace d'adressage de chaque processus n'est modifiable que par le processus lui même et le noyau est protégé en lecture et écriture.

2.2. Avantages de la mémoire virtuelle

Parmis les avantages apportés par la mémoire virtuelle on trouve :

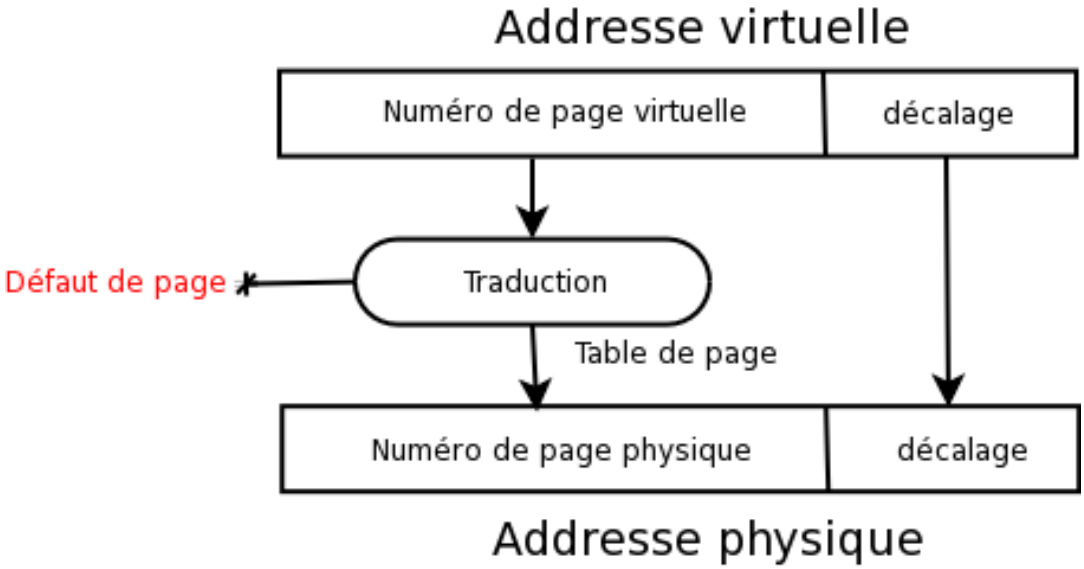
- Contribue à l'écriture de code indépendant de la machine. Portabilité, réutilisabilité.
- Gestion automatique de la zone d'échange. Les programmes peuvent utiliser plus de mémoire qu'il n'y a de mémoire principale
- Optimisation de l'utilisation de la mémoire. Les pages non utiles engendrées par erreurs de gestion mémoire comme les fuites ou les allocation inutiles se retrouverons dans la zone d'échange et libérerons des cadres de page pour des données utiles.
- Possibilité d'utiliser des bibliothèques partagées. Optimisation de l'utilisation mémoire
- Permet d'implémenter des systèmes multiprogrammés de manière sûre.
- Optimisation de l'utilisation mémoire et des performances grâce à des techniques comme la copie à l'écriture et la paginition à la demande.

2.3. Pagination

La pagination est l'une des deux techniques (qui peuvent être combinée) de la mémoire virtuelle. Le principe est de diviser l'espace d'adressage en blocs de taille fixée par le processeur; la taille de ces blocs varie de 4KB à 64KB voir 4MB dans certains cas. Chacun de ces blocs peut être déplacé dans la mémoire physique. Le système d'exploitation avec la coopération du matériel doit maintenir une liste qui permet de localiser l'emplacement de chaque bloc, qui peut éventuellement être déplacé sur le disque.

L'avantage de cette approche est la simplicité des opérations associées à des données dont la taille est connue d'avance. Introduire des blocs de tailles variables impliquerait l'utilisation d'algorithmes de placement coûteux et complexes.

Avec cette adresse la méthode pour retrouver où se situe nos données est facile il suffit de découper une adresse virtuelle en deux parties. Les bits de poids forts indiquent la page à laquelle nous faisons référence alors que les bits de poids faibles représentent le décalage dans la page. Lorsqu'une adresse virtuelle est présentée à la MMU elle extrait les bits de poids fort et commence à rechercher l'adresse réelle de la page. Si cette page n'est pas trouvable alors un défaut de page se produit et est signalé au système d'exploitation par le processus. La tâche de celui-ci est alors de savoir quoi faire selon le contexte associé au défaut de page (voir Section 4, “Linux”). Si la page est trouvée alors les permissions sont vérifiées et si celles-ci sont correctes l'adresse physique est envoyée sur le bus mémoire; si les permissions sont fausses un défaut de page se produit.



Traduction d'adresse par la pagination

Notons que dans ce schéma dès que la MMU rencontre un problème pour faire la traduction d'adresse un défaut de page est remonté au système d'exploitation ce qui lui délègue les tâches suivantes

- Aller chercher la page manquante dans la zone de zone et la mettre en mémoire principale
- Implémenter un algorithme de remplacement de page qui choisi judicieusement les pages à évincer
- Gérer les défauts de page générés volontairement par des mécanismes tel que la *copie à l'écriture* (COW), qui permet de retarder le plus possible la duplication de page.

Ceci laisse une grande liberté aux systèmes d'exploitation pour implémenter des fonctionnalités avancées ou émuler de manière logicielle des fonctionnalités non présentes matériellement.

Le découpage en blocs de taille fixe malgré la simplification des opérations introduit une fragmentation interne. Dans un système paginé les protections portent sur une page ainsi si deux données ne doivent pas être protégées de la même manière celle-ci doivent se trouver dans deux pages différentes ce qui peut mener à des pages partiellement remplies. Plus la taille des pages est petite plus la fragmentation interne sera faible mais d'un autre côté plus les structures de données du noyau seront nombreuses pour gérer la même quantité de mémoire. Actuellement la plupart des processeurs offres des pages de 4KB et l'on se dirige vers des pages un peu plus grande, puisque la mémoire principale ne cesse de grossir et 4KB commence à être problématique pour les noyaux. Certains processeurs tel que les X86 fournissent aussi des pages de 4MB ici la fragmentation interne peut être très importante ! Les processeurs alpha permettent de choisir des pages de 8 à 64K.

Le mécanisme de traduction d'adresse peut reposer sur deux principes. Soit un table des pages multi-niveaux soit une table des pages inversée. Nous allons présenter rapidement ces deux techniques.

2.3.1. Table de pages multi-niveaux

La structure la plus simple pour une table de page est un simple index ou chaque entrée contient son emplacement mémoire et quelques bits pour stocker les informations utiles.

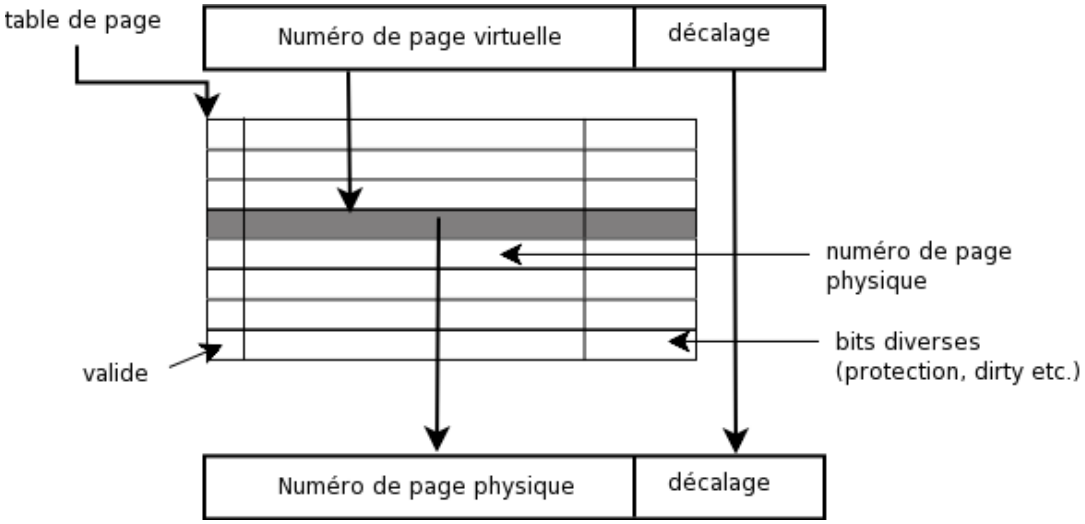


Table de page à un niveau

Trouver l'adresse physique est très simple il suffit alors de remplacer les bits de poids fort décrivant la page virtuelle par la page physique. On peut difficilement faire plus rapide; cependant cette méthode a un énorme inconvénient la taille occupée par la structure. Imaginons des adresses sur 32 bits avec des pages de 4KB. Les 12 bits de poids faibles correspondent donc au décalage dans la page et les 20 bits de poids fort au numéro de la page virtuelle. Ceci donne 2^20 entrées dans la table des pages. Une *entrée de table de page* occupera en moyenne 32 bits (20 bits d'adresse et 12 bits de signification diverse). Ainsi une seule table de page occupe 2^20 * 32 = 32MB; sachant qu'il y a une table de page par processus la consommation mémoire est énorme et sur les machines peu pourvues en mémoire principale la consommation mémoire du telle table de page peut excéder la mémoire physique présente !

Pour contrer ce problème on a introduit les tables de pages multiniveaux. Le but est multiple; tout d'abord seule la table des pages de plus haut niveau à besoin d'être résidente en mémoire les autres peuvent elles même être déplacées vers l'espace d'échange, de plus toutes les tables de pages n'ont pas besoin d'exister tant qu'elles ne sont pas nécessaires. On évite donc d'utiliser de la mémoire inutilement. Les tables de pages rencontrées font généralement deux niveaux (X86)ou trois niveaux (alpha). Plus de niveaux d'indirection commencent a faire sentir une pénalité à l'exécution.

Prenons l'exemple des processeurs X86 qui présentent une table des pages à deux niveaux, un espace d'adressage de 32 bits et des pages de 4KB (nous laissons de côté *PAE* et *PSE*. Chaque entrée de table de page (PTE) fait 4 octets. Il y a donc 2^20 PTEs à référencer. Au lieu d'utiliser une simple table nous utilisons plusieurs niveaux de table. La table de plus haut niveau contient 1024 entrées qui pointent vers 1024 tables de pages de niveau 1. Ainsi chaque entrée de la table de niveau 0 adresse 4MB de mémoire. Les tables de niveau 1 contiennent 1024 PTE qui indiquent la position de la page virtuelle.

On constate que non seulement cette représentation demande moins de mémoire que la représentation à un niveau, 4MB + 4KB contre 20MB, mais en plus les 4MB peuvent être paginés et seule la table racine a besoin d'être résidente. La traduction d'adresse se fait aisément de la manière suivante.

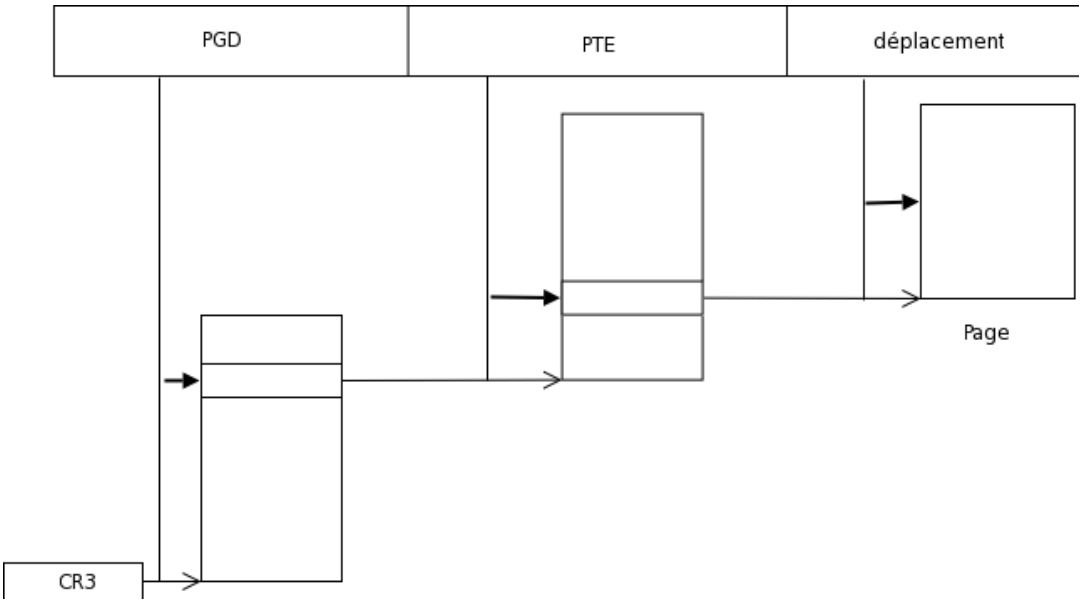


Table de page à deux niveau

On extrait tout d'abord les 10 bits de poids fort, a l'entrée correspondante dans la table de page racine on obtient une adresse qui correspond à la table de page contenant le PTE recherché. On extrait alors les 10 bits de poids fort suivants. À l'indice correspondant de la table des pages de niveau 1 se trouve le PTE associé à notre adresse virtuelle. Il n'y a plus qu'à combiner les 20 bits d'adresse contenus dans le PTE avec les 12 bits de déplacement de l'adresse virtuelle.

Des processeurs tel que le PowerPC ou le PA-Risc implémentent cette recherche de manière un peu différente si bien que dans certain cas on n'est pas obligé de suivre toutes les indirections pour retrouver l'adresse physique. Pour plus de détails on se référera à la documentation de ces processeurs.

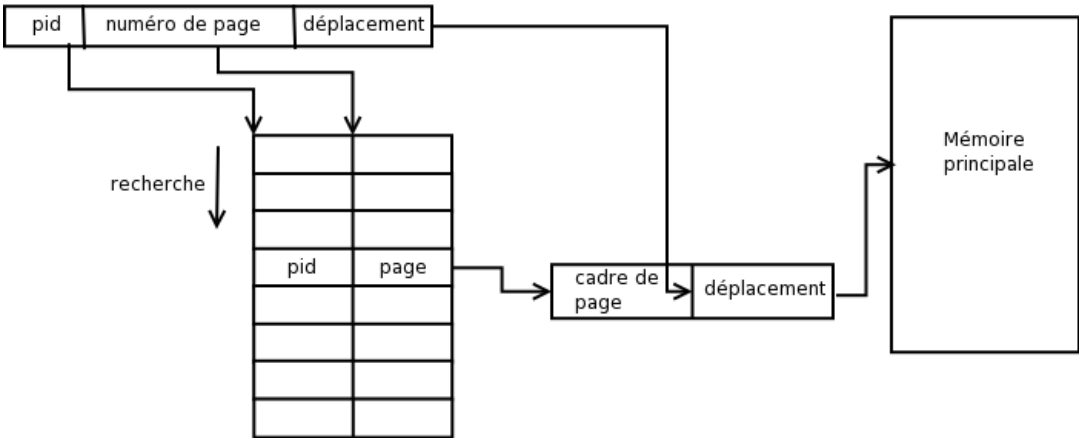
En dépit de sa simplicité cette approche à quelques inconvénients. La taille des données croit avec la taille de l'espace d'adressage et non avec la taille de la mémoire principale. De plus plusieurs niveaux d'indirections sont nécessaires ce qui ralenti la recherche. Enfin ce nombre d'indirections augmente avec des espaces d'adressages conséquents comme ceux de 64 bits. Elle reste toutefois employée dans beaucoup de processeurs.

2.3.2. Table de pages inversée

La table de page multiniveaux est une approche intuitive et puissante cependant elle a l'inconviénent de gaspiller une quantité non négligeable de mémoire, tout d'abord par ce que la taille d'un table de page croit en fonction de la taille de l'espace d'adressage virtuel; mais aussi par ce que chaque processus dispose de sa propre table.

Il existe une deuxième approche nettement moins consommatrice de mémoire, mais moins intuitive et semblant ne pas permettre le partage des pages. Il s'agit des tables de page inversées. Le principe est le suivant. On dispose d'une unique table pour le système qui a pour taille le nombre de cadre du page du système. L'indice de ce tableau correspond au numéro de cadre de page. Une adresse virtuelle est constituée de trois parties <processus, numéro de page, decalage> alors qu'une entrée dans la table des pages contient <processus, numéro de page>.

L'approche la plus basique quand on dispose d'une adresse virtuelle est de parcourir la table des pages à la recherche du couple processus/adresse virtuelle correspondant.



Traduction d'adresse par table de page inversée

Cependant cette méthode n'est pas applicable, on constate qui faut en moyenne nombre_de_cadres_de_page / 2 accès mémoire pour traduire une adresse virtuelle ! Pour résoudre se problème on ajoute une table de hachage pour réduire le nombre d'adresse mémoire à consulter.

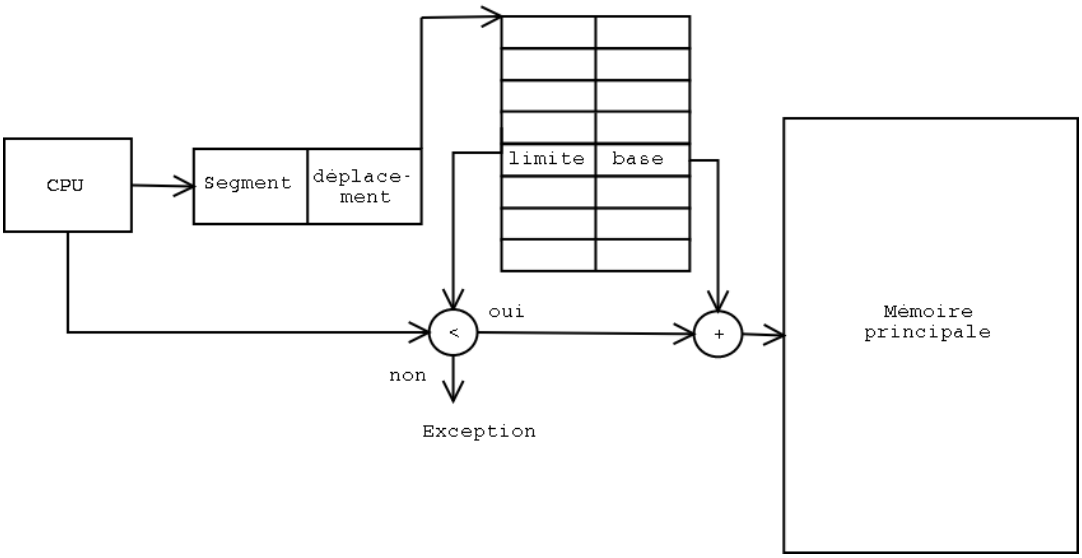
Cette technique est assez peu employée et je ne vois personellement pas comment partager des pages avec cette technique (possible ?).

En plus des deux procédés présentés il existe un petit nombre de variantes. Certaines approches, bien que non implémentées actuellement, ont été étudiées par divers chercheurs; on peut notamment citer les travaux de Jochen Liedtke sur une sorte de table de page multi-niveaux mais non balancée évitant ainsi de gâcher de l'espace pour des espaces d'adressage clairsemés. Son implémentation introduit plus de niveaux d'indirections qu'une table de pages à 3 niveaux classique mais ceci pourrait être compensable à l'aide d'un cache pour stocker les points d'entrer les plus utilisés dans la table des pages. On se reportera notamment à [LIE94] et [LIE] pour plus d'informations sur cette théorie intéressante. On y trouvera aussi une critique des deux autres approches.

2.4. Segmentation

D'un point de vue conceptuel la segmentation est à l'opposé de la pagination. Là où l'on positionnait des données dans des blocs de taille fixe facilement déplacable, la segmentation propose de regrouper un ensemble homogène de données dans un bloc mémoire de taille appropriée. Les informations sont alors accessibles par rapport à l'adresse de début d'un segment donné; une adresse devient donc un couple (numéro de segment, déplacement). La position réelle d'une telle donnée est donc modifiable en changeant l'adresse du début du segment, le programme lui ne voit rien. Cependant le calcul entre des adresses n'appartenant pas à un même segment n'a dès lors aucun sens.

L'avantage apporté par la segmentation est de pas avoir à dupliquer d'informations comme dans la pagination (disposer des protections sur 2^20 pages est plus coûteux que protéger un segment) et de ne pas introduire de fragmentation interne. De même partager un segment entre plusieurs utilisateurs est plus simple que de maintenir des ensembles de pages cohérents. Le calcul de l'adresse physique correspondant à une adresse virtuel est plus simple que pour la pagination et se fait de la manière suivante



Traduction d'adresse par la segmentation

Cependant malgré le fait que le principe de la segmentation s'énonce simplement elle n'est pas la réponse à tout les problèmes et introduit une plus grande complexité dans la conception des systèmes et ne simplifie pas vraiment les algorithmes ! Si la pagination introduisait une fragmentation interne (les blocs sont de taille fixe mais les données peuvent être plus petites) la segmentation introduit un fragmentation externe. Les segments sont de taille variable et peuvent être déplacés dans l'espace d'adressage ceci implique de trouver une zone libre de taille suffisante pour placer les segments ce qui peut être coûteux. De même le déplacement vers la zone d'échange d'un segment s'effectue par segment. L'avantage étant qu'on déplace un ensemble logique servant peu (qui à donc peu de chances de resservir) mais la pagination permettait de déplacer des pages inutiles dans un ensemble donné (erreurs d'allocation par exemple).

Le nombre de segments étant limité (2^13 sur X86) cela peut aussi poser des problèmes si de très nombreux segments doivent être créés. Ceci à d'ailleurs été problématique dans Linux 2.2 et empêchait de créer plus de 2^12 processus.

Certain systèmes comme MULTICS ont implémenté un modèle mémoire basé sur de la segmentation paginée combinant les avantages des deux méthodes. Ceci à permis la mise en place de protections très fines et toujours peu égalées dans les systèmes actuels (les anneaux de protection ont été remplacés par des systèmes plus rudimentaires). Cependant la complexité de la VMM était très importante et les systèmes UNIX ont repris les bonnes idées de MULTICS tout en les simplifiant notamment en mettant de coté la segmentation qui ne sert pratiquement plus aujourd'hui. D'ailleurs l'AMD64 ne dispose plus d'unité de segmentation en mode 64 bits.

2.5. Gestion de la zone d'échange

L'un des rôles du système d'exploitation est de gérer de manière transparente la zone d'échange. L'utilisation du disque dur comme zone de délestage temporaire de la mémoire principale repose sur le principe de localité. Des études sur le comportement des programmes , notament de P. J. Denning [DEN70], [DEN72], [DEN68], [DEN68b] et [DEN96], montrent que généralement les accès mémoire suivent un principe de localité. C'est-à-dire qu'à un instant donné seule une partie de l'espace d'adresse d'un processus est utilisée, cette partie est appelée ensemble de travail. Les études statistiques montre que cet ensemble de travail n'évolue que lentement au cours de l'exécution d'un programme; si une référence mémoire a été fait à l'adresse X à l'instant T il y a de grande chance que cette adresse resserve dans un futur proche. Inversement si une adresse n'a pas été utilisée depuis longtemps, il y a peu de chance que l'on en ait besoin.

Les *algorithmes de remplacement* tirent parti de ce principe pour enlever des pages qui ont le moins de probabilité d'être utilisées dans un futur proche. Il existe un petit nombre d'algorithmes qui sont notamment décrit dans [TAN], nous n'aborderons pas ces algorithmes dans ce document.

2.6. Séparation en deux de l'espace d'adressage

Sur les systèmes utilisant la pagination l'espace d'adressage des processus est généralement séparé en deux parties. Une première partie contient les données du processus et une deuxième partie, identique pour tous les processus, contient la mémoire du noyau.

Le but de cette technique est de minimiser le coût des appels systèmes puisque la mémoire se trouve directement accessible mais protegée par le matériel et il n'y a pas à vider le *TLB* à chaque fois. De même il est alors très facile de transférer des données de l'espace utilisateur vers l'espace noyau.Pour être efficace le noyau à souvent besoin d'avoir dans son espace d'adressage toute la mémoire physique. Cela s'effectue simplement lorsque l'espace d'adressage est assez grand mais pose problème sur les architectures 32 bits. En effet si une machine dispose de 3GB de mémoire principale on ne peut raisonablement pas réserver 3GB de mémoire pour le noyau, les processus ne pourraient alors au maximum adresser qu'un GB de mémoire !

Ainsi généralement on effectue un découpage de taille fixe; courament 3/1 sur les architectures 32 bits. Le noyau ne dispose que d'un GB et ne peut donc pas directement accèder à toute la mémoire. Cela est effectué par une astuce qui consiste à reserver une partie de ce GB pour effectuer des mapping temporaires. On change la table des pages pour pouvoir accèder à l'information intéressante. Cependant une partie des données du noyau doit absolument être adressable directement et doit donc demander moins d'un GB de mémoire.

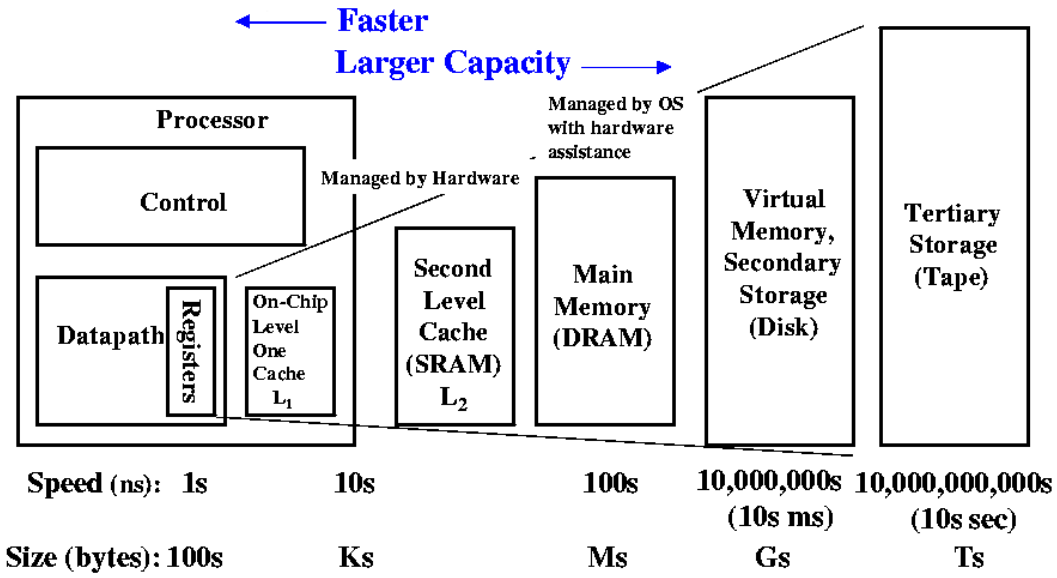
2.7. Hiérarchie de mémoires caches

Jusqu'ici nous n'avons parlé que de trois types de mémoire : la mémoire de masse constituée des disques dur, des lecteurs de bandes ou optiques etc., la mémoire principale et enfin les registres du processeurs.

Ces trois catégories sont séparées par au moins un ordre de grandeur tant au niveau de la taille que des temps d'accès et débit; voici quelques chiffres :

Mémoire de masse: Capacité : de 100GB à quelques TB, Temps d'accès : une dizaine de ms, Débit : 50MB/s

Mémoire principale: Capacité : de 256MB à quelques GB, Temps d'accès : de l'ordre de la dizaine de nanoseconde (133Mhz pour de la SDRAM)



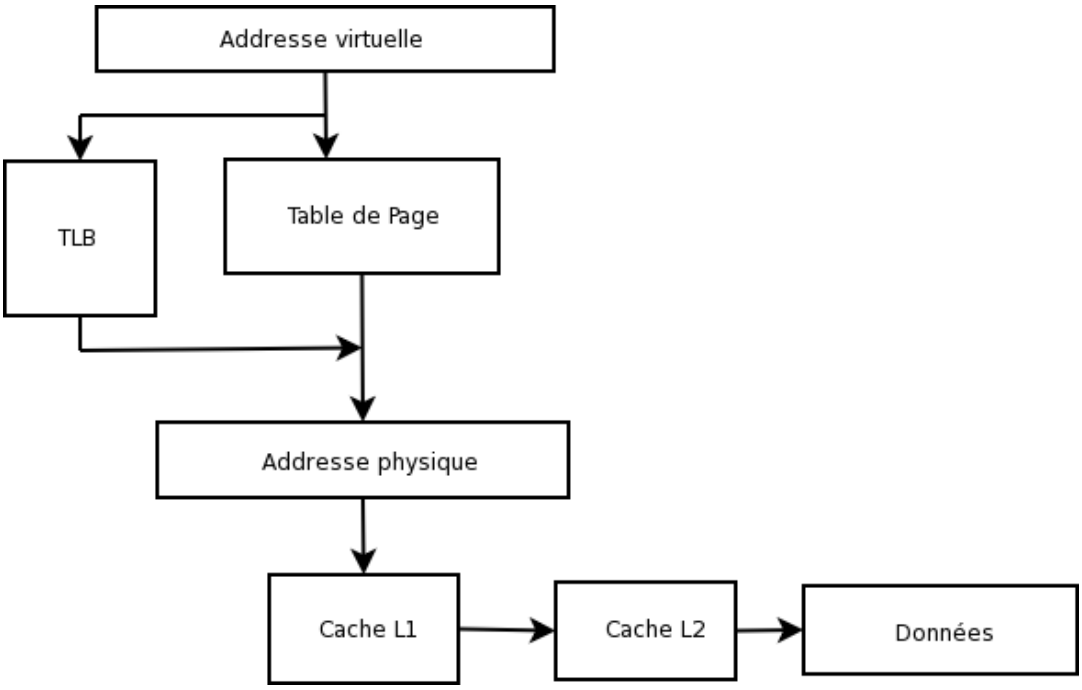
Hiérarchie mémoire des machines actuelles

En comparant ces ordres de grandeur à ceux d'un processeur on peut constater que non seulement al-

ler chercher une donnée sur le disque prend une éternité, n'oublions pas que les processeurs actuels ont une fréquence de l'ordre du Ghz, mais aller chercher une donnée en RAM n'a pas un coût négligeable non plus. Si nous revenons sur le principe de table de page que nous avons introduit on constate que pour faire la traduction d'une adresse virtuelle en une adresse physique il faut consulter la table des pages. Cette table de page se trouve elle même en RAM et pour une table à 2 niveaux il faut 3 accès mémoire pour obtenir l'adresse physique ! Ceci représente un coût gigantesque à l'exécution; l'introduction de la mémoire virtuelle ralentirait alors énormément la machine pourtant cela ne se produit pas pour une raison simple. La mémoire d'un ordinateur actuel est constituée d'une hiérarchie de mémoires. Ainsi on a introduit des caches matériels à diverses endroits qui représentaient des goulots d'étranglement. Ainsi les disques durs sont équipés d'un cache d'une taille de quelques MB qui permet d'éviter d'avoir à déplacer la tête de lecture si la données est encore présente dans le cache.

De même des caches ont été introduits entre la mémoire principale et le processeur. Des caches de niveau L1, L2, voir L3 ont été introduits pour garder un petit nombre de données en mémoire et les rendre accessibles très rapidement. Plus la mémoire est proche du processeur plus elle est rapide mais plus elle est petite car coûteuse. Le but du jeux étant d'essayer de maintenir dans les caches les plus rapides les informations accédées le plus fréquemment. Tout ces caches se basent sur le principe de localité et implémentent diverses méthodes de remplacement dont l'étude sort du cadre de ce document.

Toutefois il existe un cache spécifique qui nous intéresse ici le *TLB*, `translation lookaside buffer`. Le TLB à pour rôle de garder en mémoire un petit ensemble de correspondances adresse virtuelle/adresse physique + détails de la page. Encore une fois le TLB se base sur le principe de localité pour être effectif. Sur les processeurs 80486 celui-ci avait une taille de 32 entrées qui à été élargie à 64 entrées depuis les Pentium III. Ainsi avec une taille de page de 4Ko le taille garde en mémoire 64 * 4 = 256Ko de traduction en mémoire accessible en moins d'un cycle CPU. Ainsi on peut éviter dans la plupart des cas le surcoût entraîné par la pagination en évitant d'avoir à consulter la table de page.



Chemin suivit pour accéder aux données dans la plupart des processeurs modernes. À chaque étape on essai de cours circuiter l'étage du dessous en regardant si la donnée n'est pas déjà dans un cache plus rapide.

Le TLB ne présente que des avantages puisqu'au mieux il permet de faire la traduction en moins d'un cycle CPU et autrement la pénalité pour un "cache missis" est de l'ordre de 50 cycles CPU; la MMU ira alors consulter la table des pages et placera la valeur trouvée dans le TLB.

Il est a noté que chaque processus disposant de son propre espace d'adressage à chaque changement de contexte le TLB doit être vidé et cela à donc un coût à l'exécution. De plus selon les processeurs

la gestion du TLB est matériel ou logiciel et le système d'exploitation à alors la tache de de faire la recherche dans la table des pages et de charger l'information correspondante à l'adresse virtuelle dans le TLB. Ceci à pour avantage de laisser le système choisir la structure de donnée pour ses tables de page le matériel n'imposant rien à l'avance. Cependant le principal inconvénient est qu'il faut alors il faut alors des instructions pour parcourir la table de pages, généralement entre 10 et 100. De plus il est possible que ces instructions ne soient pas dans le cache d'instruction alors la pénalité à l'exécution est plus que conséquente par rapport à une procédure câblée.

Le choix de l'une ou l'autre des approches se fait en observant la fréquence des "cache miss" et donc le coût induit par tel ou tel choix. Nous présenterons en annexe les solutions proposées par tel ou tel processeur.

2.8. Exhausion de l'espace d'adressage

Un des buts et une des conséquences de la mémoire virtuelle est de pouvoir adresser plus d'espace mémoire que de mémoire principale n'est disponible. Bien entendu comme nous l'avons vu cela suppose que l'espace de travail reste lui plus petit ou égal que la mémoire principale le contraire entraînant inévitablement du trashing.

Cependant l'espace constitué de la mémoire principale et des zones de swap sur le disque constitue lui aussi un ensemble de taille borné. Il peut arriver que le système arrive à court de mémoire et ne puisse plus satisfaire les demandes d'allocation mémoire. Cette partie de la gestion mémoire est souvent appelée OOM pour "out of memory management". Un des principaux problèmes qui se pose est comment gérer de manière optimale la situation. À ce problème différentes solutions ont été proposées aucunes n'est satisfaisante nous allons en présenter quelques unes en pointant leurs avantages et leurs inconvénients.

La politique optimale n'est réalisable que part l'administrateur qui sait évaluer la situation et choisir l'action qui aura le moins de conséquence. Évidement un tel algorithme n'est pas concevables ainsi tout les algorithmes présentés présentent au moins un cas pathologique ou il prend la mauvaise décision.

La première solution et la plus simple est de ne rien faire. En fait il suffit de faire échouer les demandes d'allocation mémoire et d'attendre que l'administrateur système intervienne. Aucune décision n'est prise cela à pour avantage de ne jamais prendre de mauvaise décision; le principale inconvénient est que la machine devient très rapidement inutilisable, la charge monte exagérément car tout les processus passent en état exécutables en essayant de se voir allouer de la mémoire. Ainsi au final si l'administrateur n'intervient pas très rapidement la machine va trasher et on arrive souvent à un point que même l'administrateur ne peut reprendre la main pour régler le problème. Cependant cette méthode est utilisée car le système ne fait jamais de faute c'est une sorte de réponse politiquement correcte au problème.

Une autre approche est de tuer la tache qui occupe le plus de place en mémoire. Cette approche peut sembler intéressante car généralement une tache qui s'emballe et qui alloue trop de mémoire et aussi celle qui consomme le plus mémoire. Un exemple classique est la machine de bureau munie de 512MB de RAM et 512MB de swap. Une machine de bureau à généralement un petit nombre d'application consommant une taille moyenne de 100MB de mémoire chacune; un serveur X, un navigateur, un traitement de texte par exemple. Pour arriver dans une situation de pénurie de mémoire il faut généralement qu'un processus s'emballe et alloue toutes la mémoire ici il pourra allouer dans les 500MB avant de rencontrer des problèmes et la tâche problématique sera alors tuée. Tout se passe bien.

Mais il y deux cas où cette politique est problématique la première est un serveur telle une base de donnée. Ces machines font souvent tourner un processus très consommateur en mémoire, souvent plusieurs GB. Si une situation de pénurie survient à cause d'un bug dans un programme annexe, à coup sûr ce processus sera tué pourtant s'il s'agissait du rôle de la machine ce n'était pas forcément un choix très judicieux... Le même problème se produit lorsque l'on était déjà proche de la pénurie la tâche entraînant la pénurie n'a pas forcément le temps de devenir la plus grosse et une tâche est tuée plus ou moins aléatoirement.

La dernière politique est de tuer le processus ayant le taux d'allocation mémoire le plus élevé sur une période donnée. Imaginons qu'une tache alloue 100MB de mémoire par seconde sur une machine

disposant d'1GB de RAM. Une pénurie de mémoire va survenir en mois de 10 secondes. Si l'on tue une tâche qui n'est pas celle qui a ce taux d'allocation à on est sûr que le problème va survenir de nouveau dans les 10 prochaines secondes et ainsi de suite jusqu'à ce que cette tâche soit tuée.

Ainsi un principe simple qui semble marché dans la plupart des cas est que la tache qui a alloué beaucoup de mémoire récemment est responsable du problème et la tuer est une solution facile pour régler le problème. Cependant il est la aussi facile de trouver un cas pathologique ou l'algorithme se comporte très mal.

Cette politique peut être implémentée de deux manières distinctes. Soit en gardant des statistiques sur l'allocation mémoire des processus ce qui à pour avantage de savoir qui tuer à coup sûr mais comme inconvénient de demander des la mémoire et des cycles CPU pour collecter ces statistiques. Une autre solution consiste à tuer la tache demandant de la mémoire. Un bête calcul de probabilité nous montre que l'on a de grande chance de tuer la tache en cause; mais comme tout probabilité... il peut y avoir quelques dommages collatéraux.

La pénurie de mémoire est donc une problématique très délicates à régler et est généralement un sujet de débats passionnant. L'attitude qui semble être de plus en plus adoptée par les systèmes d'exploitation est de fournir ces trois choix à l'utilisateur et de laisser utiliser celui qui lui semble le plus adapté. Aussi performant que puisse être la VMM il n'y a pas de miracle et la quantité de mémoire disponible restera à jamais finie. La solution simple à se problème est donc que l'empreinte mémoire de tout les programmes à exécuter soit inférieure à la mémoire disponible l'"OOM killer" ne devrait intervenir que pour réparer les dégâts causés par des applications défectueuses.

2.9. L'évolution du matériel

Dans le domaine de la gestion mémoire et de la mémoire virtuelle la recherche s'est majoritairement effectuée entre 1965 et 1985. Les grandes étapes étant l'étude des algorithmes de remplacement et du trashing dans un premier temps puis l'implémentation de primitives évoluées par les systèmes UNIX dans le début des années 80. Notamment avec la spécification par le CSRG d'un API consistant et évolué et la décision par les ingénieurs de chez SUN de remplacer la VM de 4.2BSD par l'API proposé pour 4.3BSD.

Depuis peut de chose ont évolué et les efforts se sont majoritairement portés sur l'amélioration des performances et l'abstraction du matériel dans la conception des noyaux. Cependant les techniques sont restées les mêmes et les fonctionnalités offertes par les différents types de matériels n'ont que peu évolué.

Cependant le matériel a radicalement évolué. De l'air des terminaux on est passé à des machines ne reposant qu'assez peu sur le réseau et disposant de la puissance de calcul et mémoire localement. Les ordres de grandeur entre les différents composants matériels ont beaucoup évolué. Voici certains exemples proposés par Rik van Riel sur sa page nommée "WANTED: fundamental CS research"

Temps pour lire tout le disque dur :

1990 Capacité : 10MB, Débit : 180kB/s, Durée : 1 minutes

2003 Capacité : 160GB, Débit : 40MB/s, Durée : 1 heures

2020 Capacité : 2560 TB, Débit : 9GB/s, Durée : 3 jours

Temps pour sauvegarder toute la RAM sur disque :

1990 Capacité : 640kB, Débit : 180kB/s, Durée : 3.5 secondes

2003 Capacité : 2GB, Débit : 40MB/s, Durée : 50 secondes

2020 Capacité : 6TB, Débit : 9GB/s, Durée : 11 minutes

Coût de la lecture de données en RAM :

1990 Inférieur à un cycle d'horloge

2003 200 à 400 cycles d'horloges ce qui correspond à des opérations évoluées sur des nombres flottants.

2020 Si l'on suit l'évolution actuelle 90000 cycles CPU !

On peut constater que depuis 10 ans les ordres de grandeurs se sont creusés et certaines opérations avec la mémoire deviennent de plus en plus coûteuses. Ainsi, par exemple les caches doivent être très performants pour éviter à tout prix d'accéder à des adresses qui ne se trouvent pas dans le TLB. La mémoire principale et surtout la mémoire de masse devient de plus en plus un goulot d'étranglement dans les machines.

De même la recherche sur les algorithmes de remplacement de page à été effectuée il y a plus de 15 ans. Depuis l'utilisation des machines à beaucoup évolué et certaines applications nouvelles sont apparues, on peut notamment citer les serveurs de diffusion de contenu (streaming). L'ensemble de travail de ce type d'applications est assez différent des applications sur lequel les recherches ont été effectuées. Ainsi l'algorithme LRU fréquemment utilisé ne convient pas à de tel serveurs cependant aucune recherche sur l'utilisation d'algorithme déjà utilisés dans les bases de données n'a été effectuée sur leur application en temps qu'algorithme de remplacement de page; il existe encore moins d'implémentations de tels systèmes.

Ainsi on peut constater que si la recherche effectuée il y une dizaine d'année était satisfaisante à l'époque depuis les choses ont bien évoluées et si l'évolution actuelle se poursuit de nombreux mécanismes et algorithmes seront à repenser tant au niveau matériel que logiciel. Les chiffres indiqués plus haut montrent comment la saturation des caches CPU ralentirait très fortement la machine et un swap entraînerait un trashing presque immédiat !

3. BSD

4. Linux

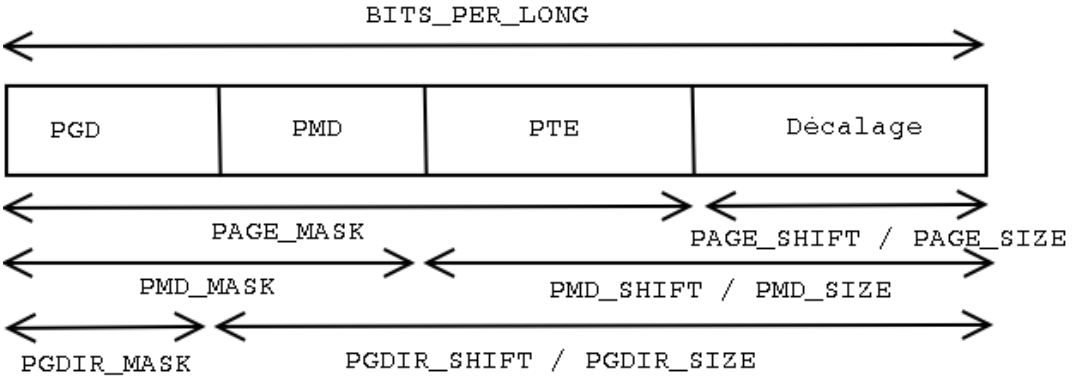
Après avoir présenté les principes généraux de la gestion de la mémoire virtuelle nous allons exposer quelques points d'une implémentation réelle avec l'exemple de Linux. Il ne s'agit pas d'un commentaire de code qui n'aurait que peu d'intérêt puisqu'il faut au préalable disposer d'une bonne vue d'ensemble du noyau pour y trouver un intérêt. Nous nous contenterons donc de faire un focus sur quelques parties, quelques structures de données ou algorithmes importants. Pour plus de renseignements il existe deux ouvrages de référence sur le sujet : [BOV] de BOVET et CESATI ainsi que la thèse de Mel Gorman [GOR]. Notons que cette thèse vient d'être transformée en livre mais les références ne sont pas encore parues à cette heure.

Les points qui ont été retenus sont la gestion des défauts de page, le modèle de pagination pour l'abstraction matériel, les structures de données décrivant la mémoire des processus et enfin les allocateurs mémoire en mode noyau.

4.1. Modèle de pagination

Comme beaucoup d'UNIX, Linux a opté pour une pagination pure sans segmentation. L'un des buts de Linux est aussi d'être portable sur différentes architectures ainsi un modèle abstrait de pagination a été retenu et le seul travail à effectuer pour un nouveau portage est de faire le lien entre le modèle de pagination matériel et l'abstraction du noyau.

Linux a opté pour une pagination à 3 niveaux car elle est très répendue sur les processeurs courants et n'est pas handicapante pour les processeurs 64 bits. L'adresse est donc constituée de 4 parties : le sélecteur d'entrée globale, d'entrée intermédiaire, de pte et enfin le décalage dans la page.



Une adresse est constituée de 4 parties logiques.

Sur les machines utilisant des tables de page de deux ou trois niveaux l'abstraction est réalisée en modifiant le nombre de bits assignés à chaque partie. Ainsi dans le cas d'une architecture 64 bits telle que l'alpha lorsque l'on utilise des pages de 8KB, les 21 bits de poids forts des adresses virtuelles sont toujours nuls, les 10 bits qui suivent servent d'entrée dans le répertoire global, les 10 autres d'entrée dans le répertoire intermédiaire, les 10 suivants d'entrée dans la table selectionnée et les 13 derniers de déplacement dans la page référencée. Par contre, dans le cas d'un X86 sans PAE les adresses ne sont divisées qu'en trois parties représentant les entrées dans le répertoire global, la table et le déplacement dans la page selectionnée qui ne contient plus que 12 bits. La quatrième partie est alors considérée de taille nulle et la table de page intémédiaire est considéré comme ne contenant qu'une entrée.

La taille des différentes structures se définie alors en positionnant un ensemble de macros dépendantes du matériel sous jacent. Ainsi `PAGE_SHIFT`,`PMD_SHIFT` et `PGDIR_SHIFT` définissent le nombre de bits à décaler pour accéder aux diffèrentes parties. Il existe tout un lot de macro permettant entre autre de connaître le nombre d'entrées par répertoire ou par page, `PTRS_PER_PTE`, `PTRS_PER_PMD` et `PTRS_PER_PGD`.

L'espace d'adressage virtuelle est séparé en deux parties; les adresses inférieures à `PAGE_OFFSET` correspondent aux pages appartenant à la mémoire du processus alors que les adresses supérieures correspondent à la mémoire noyau. Tout les processus disposent des mêmes entrées pour ce qui concernent les pages au dessus de `PAGE_OFFSET`. On peut ainsi à tout moment basculer en mode noyau sans pénalité. Les bits de protection (ring level) sont positionés pour que cette mémoire ne soit accessible qu'en mode noyau.

4.2. Espace d'adressage d'un processus

L'espace d'adressage d'un processus, décrit par la structure mm_struct, est constitué de toutes les adresses linéaires que le processus est autorisé à utiliser. Les adresses utilisés par deux processus n'ont aucune relation, elles sont indépendantes. Le noyau peut ajouter ou supprimer dynamiquement des intervalles d'adresses linéaires, représentés par des vma décrites plus loin, dans l'espace d'adressage d'un processus.

La structure mm_struct:

```
struct mm_struct {
        struct vm_area_struct *mmap, *mmap_avl, *mmap_cache;
        pgd_t * pgd;
        atomic_t count;
         int map_count;
        struct semaphore mmap_sem;
        unsigned long context;
        unsigned long start_code, end_code, start_data, end_data;
```

```
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_cnt;
        unsigned long swap_address;
        void * segment;
}
```

mmap	st la liste triée des régions mémoires appartenant au processus.
mmap_avl	est l'arbre AVL des régions mémoire.
mmap_cache	est un pointeur sur la dernière région mémoire référencée par le processus
segment et pgd	pointent respectivement sur la table locale des descripteurs et le répertoire global de pages(LDT et PGD).
count	donne le nombre de processus qui partage le même descripteur.
map_count	défini le nombre de régions mémoire que possède le processus.
mmap_sem	est un sémaphore utilisé pour l'instauration d'une section critique lors de la modification des champs mmap et mmap_avl
start_code, end_code, start_data, et end_data	sont les délimiteurs respectifs de la zone de code et de données.
arg_start, arg_end, env_start, env_end	définissent les adresses de début et de fin des zones contenant respectivement les arguments du processus et l'environnement dans lequel il doit s'exécuter(environnement de compilation ou d'exécution).
start_brk, brk	sont les adresses respectives de début et de fin du tas.
rss	spécifie le nombre de cadres alloués au processus.
total_vm	donne la taille de l'espace d'adressage exprimé en nombre de pages.
locked_vm	donne le nombre de pages vérouillées, c'est à dire celle qui ne peuvent pas être swappées sur le disque.
def_flags	contient les drapeaux par défauts qui seront attribués aux régions mémoire allouées au processus
swap_address	l'adresse de début de la zone de la mémoire swap ou seront déplacées les pages du processus.

Tout processus utilisateur dispose d'une structure mm_struct. Dans le cas général il lui est propre, cependant les threads sont implémentés en utilisant les mêmes fonctions que pour la création d'un processus, `fork()` à l'exception que le clone de l'espace d'adressage n'a pas lieu. Le deux threads dispose de la même reférence sur une mm_struct.

En fait on constate que l'espace d'adressage d'un processus est constitué d'un ensemble de régions mémoire. Une région mémoire est une suite d'adresse possédant les mêmes propriétées logique. Ain-

si la pile est une région mémoire, accessible en lecture et écriture, non exécutable, qui n'est pas relié à un objet dans l'espace de nommage (vm_file est à `NULL`), et extensible vers les adresses basses. Un appel à `mmap()` créer une nouvelle région mémoire pour placer une fichier dans l'espace d'adressage du processus. Les régions mémoire sont représentées par une structure vm_area_struct(vma) définie ainsi:

```
struct vm_area_struct {
        struct mm_struct *vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        struct vm_area_struct *vm_next;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        short vm_avl_heigth;
        struct vm_area_struct *vm_avl_left, *vm_avl_right;
        struct vm_area_struct *vm_next_share, *vm_pprev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct file * vm_file;
        unsigned long vm_pte;
}
```

vm_mm	est un pointeur sur le descripteur de mémoire du processus propriétaire de la région.
vm_start, vm_end	contiennent les adresses de début et de fin de la région.
vm_next	est un pointeur sur la zone mémoire suivante.
vm_avl_left, vm_avl_right	pointent respectivement sur les fils gauches et droites de l'arbre AVL du descripteur de mémoire(vm_area_struct).
vm_avl_heigth	contient la profondeur du sous-arbre dont la racine est la région mémoire elle-même.
vm_flags	contient les drapeaux associés à la région mémoire.
vm_page_prot	contient les valeurs initiales des drapeaux de tables de page.

Notons qu'une VMA est toujours alignée sur la taille des pages du système.

Comme nous l'avons vu Un processus peut possèder plusieurs régions mémoire. Pour lier ces régions entre-elles il dispose de deux solutions. La première est de les lier au moyen d'une liste chaînée. Cette liste est représentée par le champ `mmap` du descripteur de mémoire du processus. `mmap` est un pointeur sur une VMA qui sera le premier élément de la liste, l'élément suivant étant représenté par le champ `next` de cette VMA. Toutefois le parcourt d'une liste s'effectue en O(n), l'utilisation d'une liste n'est donc valable que si le nombre de régions mémoire du processus reste de l'ordre de la dizaine, ce qui est le cas de la majorité des processus . Par contre, pour de grosses applications, telles que des bases de données, pouvant posséder de plusieurs centaines à plusieurs milliers de régions mémoire, l'utilisation d'une liste devient beaucoup trop coûteuse. Dans ce cas là une deuxième solution est utilisée, un arbre AVL. Cette structure de données apporte une recherche d'un élément en O(log(n)) et est donc utilisée pour des processus possèdant un grand nombre de régions mémoire. Cependant les AVL ont leurs inconvénients. En effet, les fonctions qui les gèrent sont bien plus complexes que celles qui gèrent les listes. Donc lorsque le nombre d'éléments est faible, il est bien plus efficace de les placer dans une liste. Le nombre de régions mémoire limite à partir duquel l'utilisation d'un AVL est préférée est représenté par la valeur de la macro AVL_MIN_MAP_COUNT.

Lorsque l'arbre AVL est créer la structure de liste chaînée est toujours maintenue cohérente. On utilise l'arbre pour trouver la VMA recherchée mais lorsque l'on insère ou supprime un élément on

l'ajout dans l'arbre et l'on met à jour les champs `prev` et `next`.

4.2.1. Droit d'accès à une région mémoire

Les droits d'accès associés à une région mémoire sont contenus dans le champ vm_flags du descripteur de VMA. Ils sont représentés par plusieurs drapeaux . Certains de ces drapeaux, donnent au noyau des informations sur les pages de la région mémoire, notamment sur leur contenu ou les droits offerts au processus pour y accèder.

On pourrait se demander l'utilité de stocker les informations sur la protection des pages puisque cette information est déjà présente dans les pte de la région. Cependant il n'y a pas redondance d'informations. En effet pour implémenter la copie à l'écriture on fait en sorte que la MMU signale qu'un accès en écriture à été tentén ceci n'est possible qu'en marquant le pte correspondant à la page en lecture seule. La VMA quand à elle contient les informations de protection réelles sur la zone mémoire. Les pte peuvent eux changer temporairement de permissions (toujours au moins aussi restrictives que celles de la VMA) pour différents besoins.

4.2.2. Allocation d'un intervalle d'adresses

L'allocation de nouvelles régions mémoire s'effectue par l'intermédaire de la fonction `do_mmap()`. Elle reçoit en paramètre l'adresse, `addr`, où doit commencer la recherche d'un intervalle libre, la longueur de l'intervalle d'adresses linéaires, `len`, les droits d'accès, `prot`, des pages contenues dans la région mémoire, et flag, les autres drapeaux de la région mémoire. Un dernier paramètre concernant une région mémoire mappant un fichier est aussi utilisé mais son utilisation sort du cadre de ce sujet. Tout d'abord, cette fonction commence par vérifier si les paramètres correspondent à une requête satisfaisable, notamment si l'intervalle d'adresses linéaires ne dépasse pas PAGE_OFFSET et que le processus ne dépasse pas le nombre de régions mémoire qu'il peut mapper. Si la reqête ne peut pas être satisfaite `do_mmap()` renvoie une valeur négative. Si elle le peut, la fonction tente d'obtenir l'intervalle désiré par l'intermédiaire de la fonction `get_unmapped_area()`, et d'allouer une vma qu'elle initialise avec les bons droits d'accès. Si aucun intervalle assez grand n'est disponible ou que l'intervalle est inclu dans une autre région mémoire du processus, la vma allouée est libérée et un code négatif est renvoyé. En cas de succès de toutes ces étapes, la nouvelle région est insérée dans la liste des régions du processus (et dans l'AVL si nécessaire) et on fusionne les régions qui peuvent l'être.

4.2.3. Libération d'un intervalle d'adresse

La suppression d'un intervalle d'adresses linéaires de l'espace d'adressage d'un processus est effectué par la fonction `do_munmap()`. Cette fonction reçoit deux paramètres addr et len, correspondant respectivement au début et à la longueur de l'intervalle à libérer. Cet intervalle ne décrit génèralement pas une région mémoire du processus, il peut être soit inclu dans une région soit s'étendre sur plusieurs régions. Dans un premier temps, cette fonction commence par vérifier si aucune adresse de l'intervalle à supprimer n'est supérieure à PAGE_OFFSET, sinon elle renvoie un entier négatif. Ensuite elle parcourt la liste des régions mémoire et crée une liste de toutes celles qui chevauchent l'intervalle à supprimer. La création de cette liste amène à des alloctions de vma et si une allocation échoue la fonction renvoie un code négatif. Les régions mémoire de cette liste sont extraites de la liste (si nécessaire de l'AVL) des régions mémoire du processus. Dans un deuxième temps, cette fonction doit mettre à jour la table des pages, et réinsérer une version plus petite des régions mémoire supprimées durant la première phase. Une boucle est effectuée pour parcourir la liste créée dans la première phase. A chaque itération le champ map_count du descripteur de mémoire du processus est décrémenté et les cadres de page de la région mémoire compris dans l'intervalle à supprimer sont libérés par la fonction. Les entrées correspondantes de la table des pages sont mise à jour, tandis que toutes les entrées du tlb sont invalidées. Ensuite si toutes les adresses linéaires de la région mémoire supprimée n'appartiennent pas toutes à l'intervalle, une version plus petite de cette région mémoire doit être réinsérée dans la liste des régions du processus.

4.2.4. Création et destruction de l'espace d'adressage d'un processus

Lors de la création d'un processus la fonction `copy_mm()` est appelée afin de créer l'espace

d'adressage de ce processus. Cette fonction prend un paramètre flag contenant un ensemble de drapeaux. Si le drapeau CLONE_VM de ce paramètre est positionné, alors le processus reçoit l'espace d'adressage de son père, ceci correspond à la création d'un processus léger. Toutefois tout processus doit possèder sa propre LDT, et sa propre entrée de LDT dans la GDT. La mise en place de cette LDT est effectuée par la fonction `copy_segments()`. Par contre si le drapeau CLONE_VM n'est pas positionné, on est dans la création d'un processus normal qui doit possèder son propre espace d'adressage. La fonction alloue alors un nouveau descripteur de mémoire et construit le descripteur de LDT toujours en appelant `copy_segments()`. Un nouveau répertoire global de pages est alloué, les entrées correspondant à celles du noyau sont copiées depuis le répertoire global de pages du processus "swapper" et les autres sont mises à 0. Ensuite la liste des régions mémoire du processus père est dupliquée et insérée dans celle du fils et les entrées de tables de pages sont initialisées. Si le nombre de régions mémoire est supérieur ou égal à AVL_MIN_MAP_COUNT, l'arbre AVL est créé.

La libération d'un processus est gérée par la fonction `exit_mm()`. Le processus entrant dans l'état zombie (TASK_ZOMBIE), la fonction lui affecte l'espace d'adressage du processus "swapper". Les descripteurs de mémoire et les tables référencées sont libérées avant la libération même du descripteur de mémoire.

4.3. Gestion des défauts de pages

Nous avons déjà rapidement parlé de ce qu'était un défaut de page, cependant nous ne sommes pas rentrés dans les détails de la gestion de ceux-ci, cette partie présente une étude de tous les cas pouvant produire un défaut de page ainsi que les traitements associés. Le point d'entrée du traitement d'un défaut de page est la fonction `do_page_fault()`qui reçoit deux paramêtres:

`regs` un pointeur sur struct pt_regs qui représente l'état des registres du processeur au moment de la levée de l'exception.

`error_code` un code d'erreur de trois bits. Le bit 0 détermine si l'exception a été causée par un accès à une page non présente en mémoire centrale, s'il est nul, ou par un accès non autorisé, s'il est positionné. Le bit 1, indique le type d'action désirée sur cette page, écriture s'il est positionné, exécution ou lecture sinon. Le dernier bit indique que l'exéception s'est produite en mode utilisateur s'il est positionné.

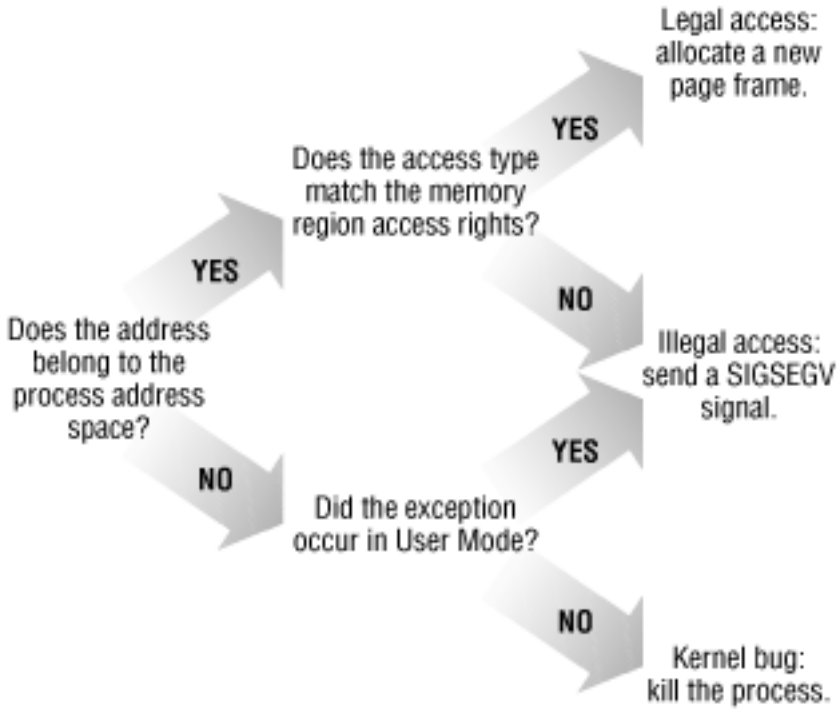


Schéma général de la gestion de défauts de page (bovet).

La prémiere étape est de déterminer si le défaut de page s'est produit en mode utilisateur ou non, le traitement étant radicalement différent.

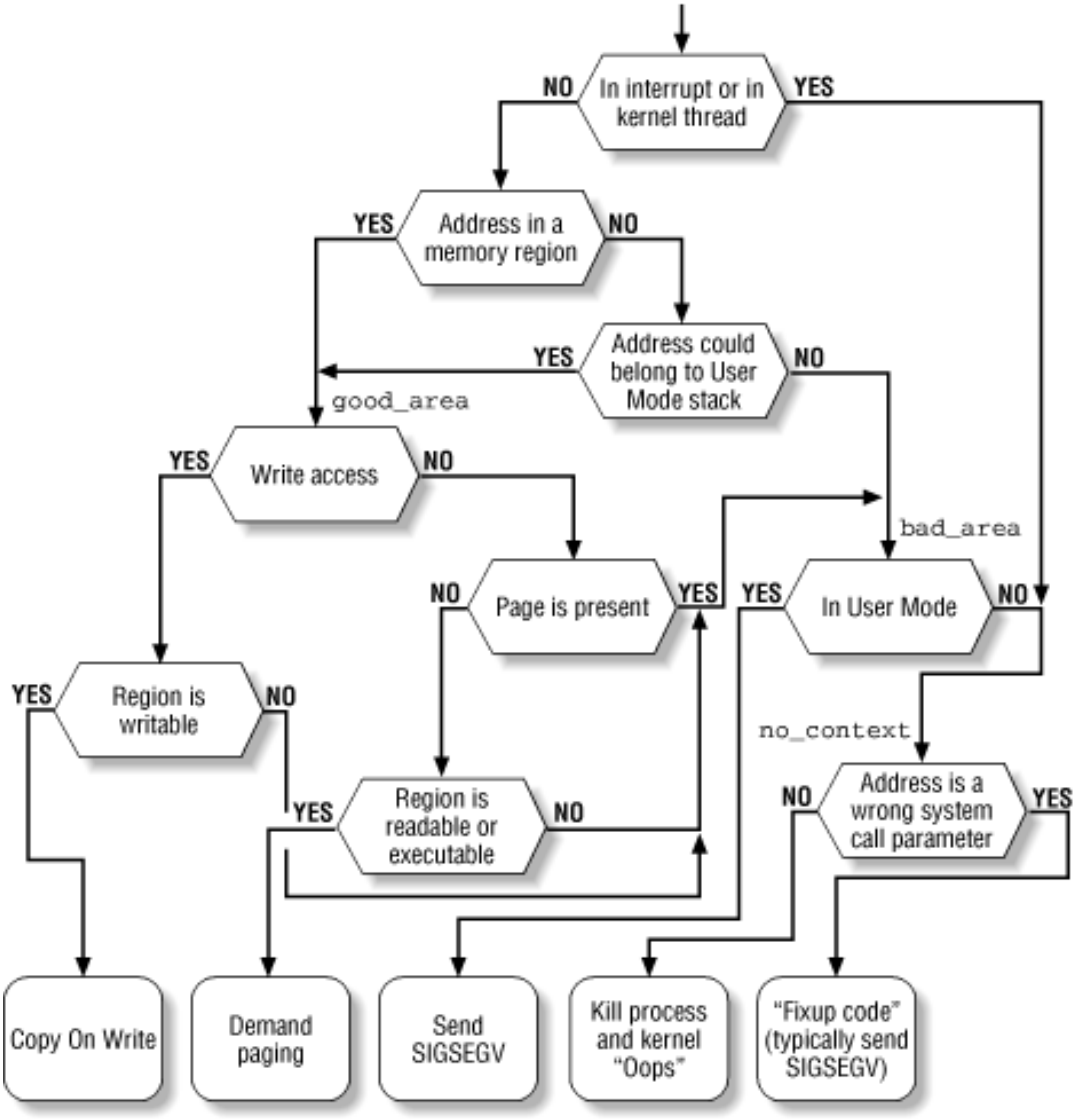


Schéma détaillé de la gestion de défauts de page (bovet).

4.3.1. Défaut de page en mode noyau

En mode noyau il n'y a que deux façons de provoquer un défaut de page. Soit le défaut de page résulte d'une adresse erronée passée en paramètre d'un appel système soit d'une erreur dans le noyau.

Pour ce qui concerne le bug noyau il n'y a pas grand chose à faire, l'état du processus est affiché sur la console (on dit qu'il `Oops`) et on essaye de continuer tant bien que mal.

Pour ce qui est de l'addresse passée en paramètre ceci peut se produire car linux effectue une detection à posteriori des adresses érronée. Dans ses premières versions la vérification de la validité de chaque paramètre était faite à l'entrée dans l'appel système toutefois ceci n'est pas très performant, désormais la seul vérification faite est que l'addresse soit inférieure à `PAGE_OFFSET`. Les autres erreurs sont attrapées à l'exécution.

Il existe deux traitants pour ces erreurs, si l'addresse qui à provoquer le défaut de page n'appartient pas à l'espace d'adressage du processus un `SIGSEV` est envoyé à processus appelant. Si l'adresse appartient à l'espace d'adressage c'est que les permissions étaient érronées un `SIGSEV` est donc aussi envoyé. Cependant comme nous le présenterons dans les défauts de page en mode utilisateur il peut aussi s'agir d'une des conséquences de techniques telles que la copie à l'écriture, se reporter à cette

section puisque les actions seront les mêmes.

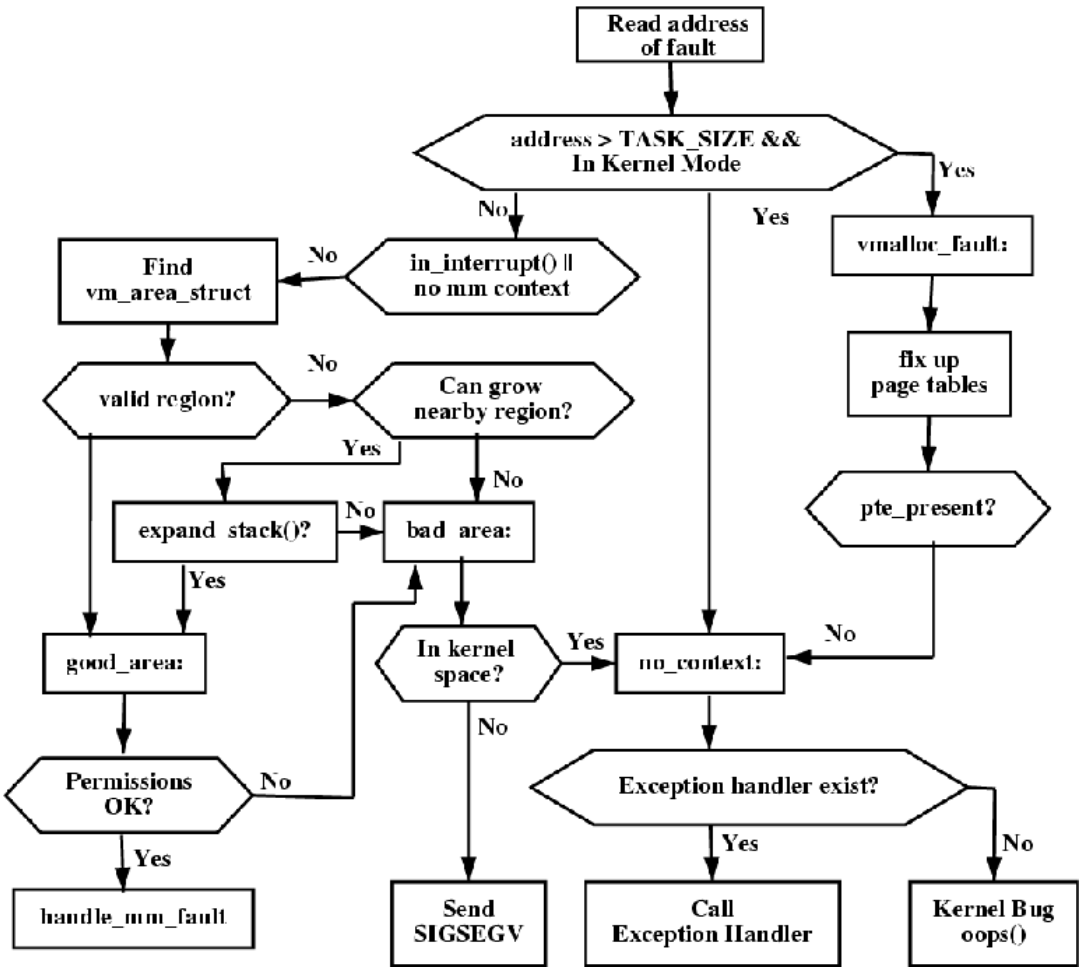
4.3.2. Défaut de page en espace utilisateur

En mode utilisateur les défauts de page sont plus courants voyons pour quel raison ils peuvent se produire. Tout d'abord il peut s'agir d'une référence à une adresse en dehors de l'espace d'adressage; dans ce cas le traitement est simple et consiste à tuer le processus fautif par l'envoie d'une `SIGSEV`. Il existe tout de même un exception à cette règle, la pile en espace utilisateur est extensible automatiquement, il faut alors vérifier que l'adresse n'est pas juste avant la fin de la pile (rappelons que la pile s'étend vers les adresses basses). Comme nous l'avons vu dans la présentation des structures VMA une régions mémoires possède un champ `vm_end` et `vm_start` permettant de la délimiter. Il est verifié si l'adresse fautive n'est pas dans les 32 octets avant le début de pile, et le cas échéant la pile est automatiquement étendue grâce à l'appel de `expand_stack()`.

Il reste alors uniquement le cas où l'adresse se trouvait bien dans l'espace d'adressage. Il peut se produire le condition suivante :

- Le code d'erreur indique une tentative de lecture et la VMA correspondante n'est pas accessible en lecture. Le processus est tué par un `SIGSEV`
- Le code d'erreur indique une tentative de lecture et la VMA est accessible en lecture. Il s'agit d'une pagination à la demande (voir plus bas).
- Le code d'erreur indique une tentative d'écriture et la VMA n'est pas accessible en écriture. Le processus est tué par un `SIGSEV`.
- Le code d'erreur indique une tentative d'écriture et la VMA est accessible en écrite. Il s'agit d'une copie à l'écriture (voir plus bas).

À chaque type d'accès correspond deux possibilités, l'une émanant d'une erreur de l'utilisateur l'autre d'une fonctionalité de la VM. Pour implémenter des fonctionalités avancées telles que la copie à l'écriture ou la pagination à la demande Linux se repose sur un jeu entre les permissions des pages et les permissions des VMAs. Ainsi les permissions des pages peuvent être volontairement plus restrictives que celles des VMAs pour générer un défaut de page et provoquer une action au dernier moment. Par contre si la protection des pages et de la VMA correspondent il s'agit bien d'une erreur de l'utilisateur !



Autre schéma détaillé de la gestion de défauts de page (mel gorman).

La pagination à la demande est une technique d'allocation dynamique qui consiste à n'allouer un cadre à une page que lorsque l'on tente d'y accèder et non par préchargement. La page absente de la mémoire principale entraîne un défaut de page et indique au système que celle-ci est nécéssaire. Le choix de la pagination à la demande plutôt qu'un préchargement implique un surcoût à l'exécution, chaque défaut de page est coûteux; cependant le principe de localité nous permet d'espèrer que le gain sera supérieur au coût. L'un des principaux avantages est que la mémoire correspondante à la page n'est réellement allouée que lorsque celle-ci est référencée et non créée. Si une page n'est jamais référencée alors aucune mémoire ne lui sera associée en réalité. De même une page n'est déplacée de la zone d'échange vers la mémoire principale que lorsque l'on en a effectivement besoin plutôt que de déplacer toutes les pages d'un processus comme le faisaient d'anciens systèmes.

Lorsqu'une page n'a jamais été référencée il faut alors ajouter l'entrée correspondante dans la table des pages et attacher un cadre de page à cette entrée ce qui est effectué par le biais de la fonction `do_no_page()` alors que si la page a simplement été déportée vers la zone d'échange un appel à `do_swap_page()` permettra de la replacer dans la mémoire principale. En réalité encore dans un soucis d'optimisation le cadre de page associé au pte est une page remplie de 0 partagée par tout les processus s'il s'agit de mémoire anonyme, en effet tant que l'on n'a pas écrit dans cette page n'importe quoi y est présent il ne sert a rien d'allouer de la mémoire pour cette page la copie à l'écriture s'en chargera ! Dans le cas d'un fichier la même technique peut être effectuée avec des pages du cache disque mais cela est un peu plus compliqué et sort du cadre de ce document.

L'autre technique est la copie à l'écriture (COW) qui elle aussi permet de rétarder jusqu'au dernier moment l'allocation de cadres de page non nécéssaires. Il est fréquent que l'on doivent dupliquer une vaste ensemble de pages par exemple lors que la création d'un nouveau processus l'espace d'adressage du père est dupliqué. Cependant il n'est pas rare que la duplication soit inutile par exemple pour un `fork()` suivit d'un `exec()` ou c'est clairement une perte de temps. La copie à l'écriture permet de retarder le plus possible la duplication pour ne l'effectuer qu'au moment où elle

est réellement nécessaire, c'est-à-dire lorsque l'un des deux processus tente d'écrire dans la page. À ce moment la duplication est inéluctable. La technique consiste donc à marquer les pages comme non écrivable mais la VMA accorde le droit d'écriture. Lorsqu'une tentative d'écriture est faite, la page est dupliquée et la permission d'écriture est repositionée sur les pte.

4.4. Allocateurs mémoire

Les différente routines du noyaux peuvent avoir besoin d'allouer de la mémoire, par exemple une structure mm_struct lors de la création d'un processus. Des routines noyaux doivent donc fournir des primitives d'allocation mémoire. En général les allocateurs noyaux se doivent d'être simple et rapide, et comme leur équivalent le but est d'avoir une utilisation de la mémoire maximale.

Ainsi les différents noyaux fournissent en général plusieurs allocateurs mémoire qui sont optimisés pour diverses tâche. Les trois UNIX les plus influent actuellement, Solaris, Linux et FreeBSD, ont opté pour les mêmes algorithmes même si les implémentations diffèrent. Tout d'abord il existe un allocateur pour les petites zones de mémoire contigues, l'approche d'allouer un nombre de page égal à une puissance de deux. Nous verrons cette solution avec l'algorithme des zones siamoises (buddy system algorithm). Cet allocateur à été décrit par Marshall Kirk McKusick dans [KUS88] lors de la sortie de 4.3BSD.

En tracant les demandes d'allocation mémoire des noyaux on observe que la plupart des requètes sont d'assez petite taille et sont effectués avec des tailles fixes. Les tampons réseaux font toujours la même taille, une mm_struct fait toujours la même taille etc. Dans ce cas la méthode des zones siamoises n'est pas la plus adaptée. Ceci à été constaté par les concepteurs de Solaris qui ont imaginé un système nommé slab allocator, qui repose sur les zones siamoises mais cache les objets pour éviter les allocations/désallocations succéssives.

Lorsque l'on n'a pas besoin de zone mémoire physiquement contiguës il existe en général un troisième allocateur mémoire. Dans ce cas il n'y a pas de problème de fragmentation.

4.4.1. Algorithme des zones siamoises

Ce système repond à la nécessité de pouvoir allouer des zones de petite taille, physiquement contiguës. Les principales contraintes sont d'être rapide et d'éviter la fragmentation. Une des approche fréquement utilisé dans ce cas est d'arrondir la taille d'allocation à une puissance de deux; le gaspillage étant assez faible et la fragmentation limité. Voyons comme fonctionne les zones siamoises.

Cet algorithme conserve les blocs de cadres de page contigus par le biais de 10 listes. Ces denières contiennent des groupes de 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 cadres de pages contigus.

À partir de ces listes, pour allouer un groupe de 2^n cadres de page contigus, on regarde si la liste contenant les blocs de 2^n cadres n'est pas vide.Le cas échéant on alloue le premier bloc. Dans le cas contraire, on cherche un bloc dans la liste qui contient les blocs de 2^n+1 cadres de page contigus. Lorsqu'on trouve un bloc disponible on alloue les 2^n cadres et on découpe le nombre de cadres restant en des groupes de cadres les plus grands possibles que l'on insère ensuite dans les listes adéquats. Si toutes les listes contenant les tailles 2^n à 512 sont vides, on signale alors une condition d'erreur. L'exemple qui suit permettra de clarifier le principe. Supposons qu'une requête soit émise pour un groupe de 128 cadres. S'il n'en existe pas, l'algorithme cherche le premier bloc de taille supérieure, soit un bloc de la liste des blocs de 256 cadres. Si un tel bloc existe on alloue 128 des 256 cadres pour satisfaire la requête et on insère les 128 cadres restant dans la liste des blocs libres de 128 cadres. S'il n'y a pas de bloc libre de 256 cadres, l'algorithme cherche un bloc de 512 pages. Si un tel bloc existe, il alloue 128 des 512 cadres, il insère les 256 des 384 cadres restant dans la liste des blocs de 256 cadres et les 128 derniers dans la liste des blocs de 128 cadres. Si la liste des blocs libres de 512 cadres est vide l'algorithme abandonne et signale une condition d'erreur.

C'est la technique de libération de blocs de cadres qui jusifie le nom de cette algorithme. Ce dernier essaye de fusionner des paires de blocs contigus siamois, c'est à dire des blocs contigus de mêmes tailles. Cette algorithme est itératif, à la suite de la fusion de blocs de taille n l'algorithme tente de fusionner des blocs de taille 2n.

4.4.2. implémentation

Le noyau Linux utilise un tableau contenant 10 free_area_struct, L'entrée numéro k correspond à la liste des blocs libres de 2^k cadres contigus. La structure free_area_struct est déclarée ainsi:

```
typedef struct free_area_struct{
       struct list_head free_list;
       unsigned long *map;
}free_area_t;
```

La liste des blocs de cadres de page contigus est représentée par le champ free_list et le champ map est un bitmap représentant l'état des paires siamoises. Linux économise de l'espace en utilisant un seul bit pour chaque paire siamoise. Si un bit du bitmap de la kème entrée est à 0, alors les deux blocs sont tous les deux libres ou tous les deux utilisés; s'il est égal à 1 alors un et un seul des deux blocs est utilisé. Lorsque les deux siamois sont libres, le noyau les traite comme un bloc libre unique de taille 2^k+1.

La fonction `_get_free_pages()` implémente la stratégie des zones siamoises en ce qui concerne l'allocation des cadres de page. Cette fonction fait appel a une macro RMQUEUE_TYPE qui reçoit en paramètre le logarithme en base 2 de la taille du bloc voulu, correspondant à l'indice h dans lequel commencer la recherche dans le tableau. Cette macro effectue alors une boucle de recherche dans les listes d'un bloc disponible, en commençant par la liste correspondant à la taille demandée en paramètre et en poursuivant si nécessaire vers des tailles supérieures. Si la boucle se termine, cela signifie qu'aucun bloc n'a été trouvé et `get_free_pages()` retourne NULL. Sinon, un bloc libre convenable a été trouvé; dans ce cas, le descripteur de son premier cadre de page est enlevé de la liste, le bitmap correspondant est mis à jour, et le nombre de pages libres est décrémenté. Si le bloc trouvé provient d'une liste de blocs de taille k plus grande que celle souhaitée, h, la macro alloue les 2^h derniers cadres de page et replace itérativement les 2^k - 2^h, premiers cadres dans les listes du tableau d'index compris entre h et k. Pour finir, cette macro renvoie l'adresse du bloc trouvé près avoir incrémenté le champ count du descripteur de page associé.

La fonction `free_pages_ok()` implémente la libération des cadres de page selon la stratégie des zones siamoises. Elle reçoit en paramètre le numéro de page de l'un des cadres de page contenus dans le bloc à libérer, map_nr, et le logarithme de la taille du bloc, order. Tout d'abord elle convertit map_nr en le numéro du premier cadre de page du bloc et incrémente le nombre de pages libres. La fonction commence alors une boucle chargée de fusionner un bloc avec son siamois, en commençant par la plus petite taille et se pousuivant vers les tailles les plus élevées. Cette fusion est effectuée par la `test_and_change_bit` qui vérifie si la zone siamoise est libre grâce au champ map de la free_area_struct. A la fin de l'itération, la fonction `free_pages_ok()` a obtenu un bloc libre le plus gros possible, et n'a plus qu'a l'insérer dans la bonne liste du tableau.

4.4.3. Slab allocator

Par manque de temps nous ne documenterons pas le slab allocator. Nous l'incluons tout de même ici car il s'agit d'une technique très interessante d'approche objet de l'allocation mémoire avec une utilisation performante de mécanismes de cache. La première documentation provient des concepteurs de Solaris 2.4 qui ont inventé cette approche; on lira [BON94] et [BON01] pour voir l'approche utilisé dans Solaris.

Pour ce qui est de Linux [BOV] et [GOR] en font une très bonne description.

5. Sujets d'actualité

Nous allons présenter ici quelques sujets qui ont fait récemment parler d'eux et expliquer les problèmes rencontré, comment les résoudre et l'implication des modifications.

5.1. Chez Linux

Plus que tout noyau arrivé au stade fonctionnel et largement déployé Linux suit un mode de développement en bazar. Là où les *BSD ne changent que très peu les choses qui fonctionnent, la

VM de FreeBSD n'a été modifiée en profondeur qu'une fois en 10 ans, Linux change constamment d'approche même sur les branches stables sans route bien définie. Ainsi un patch d'AA modifiant profondément la VM avait été appliqué pour la sortie du 2.4.10 en septembre 2001. Depuis la VM reste un sujet particulièrement brûlant surtout en ce moment où plusieurs solutions sont envisagée pour résoudre les problèmes posés par les machine X86 disposant de plusieurs gigas de RAM.

Notons que la plupart des problèmes actuellement rencontrés par Linux sont du à la vieillissante architecture X86. Pour certaines raisons assez obscures certaines personnes s'obstinent à vouloir disposer de grosse quantité de mémoire sur une architecture qui n'a pas été prévu pour. Ceci est d'autant plus absurde que de nos jours les machines 64 bits sont très abordables. Pour répondre à la demande nombre de solutions sont envisagées...

5.1.1. 4K Stack

Récemment un patch nommé "4K Stack" a été inclus dans le noyau. Ce patch a fait beaucoup parlé de lui car il casse un certain nombre de pilotes notamment ceux des cartes radeon ce qui ne pose pas de problème celui-ci étant libre mais surtout celui d'NVidia. Ce patch serait peut être passé presque inaperçu autrement mais puisque l'on en a un petit peu parlé présentons concrètement de quoi il s'agit et les motivations qui ont poussées à casser certain pilotes.

Dans la plupart des systèmes UNIX chaque processus dispose d'un pile en mode utilisateur mais aussi d'une en mode noyau. Lorsqu'un appel système est effectué, la pile de l'utilisateur n'est pas utilisée pour ne pas la perturber. Cette pile est de petite taille, historiquement elle est de 8KB sur la plupart des architectures dans Linux et sert à deux choses. Comme nous l'avons vu elle est utilisée lorsqu'un processus bascule en mode noyau mais si une interruption logicielle ou matérielle survient la pile est aussi utilisée pour le traitement. Contrairement à sa version en mode utilisateur elle n'est pas extensible, tout débordement provoque une corruption mémoire ou un `panic()`. Ceci impose quelques restrictions pour les traitants; bien évidement ils ne doivent pas mettre de grosses structures de données 8K étant très petit mais surtout ceci interdit la récursivité.

Quel était le problème avec cette méthode ? Comme nous l'avons dit à chaque processus est associé une pile noyau, cette pile faisant 8K nécessite 2 pages contiguës en mémoire. Cela peut évidement poser problème lorsqu'il y a une forte pression sur la VM mais aussi lorsque le système est très fragmenté par exemple après une très longue période d'exécution. Il peut être difficile de trouver deux cadres de page contiguës, notamment sur les machines faisant tourner un grand nombre de processus.

L'idée est donc de réduire la taille de la pile noyau à 4KB ce qui élimine le problème. L'inconvénient est qu'il faut s'assurer qu'aucun traitant n'a besoin de plus de 4KB de pile. Pour cela deux nouvelles piles ont déjà été créée pour le traitement des interruptions logicielle et matérielle ce qui permet déjà de réduire les chances de dépassement. Le reste n'est que de la vérification de code notamment grâce à ce script (je ne résiste pas à l'envie de le mettre :-). Son fonctionement est expliqué à l'adresse suivante : http://lkml.org/lkml/2004/5/14/34.

```
objdump - -disassemble "$@" | \
sed -ne '/>:/{s/[<>:]*//g; h; }
  /subl\?.*\$0x[^,][^,][^,].*,%esp/{
  s/.*\$0x\([^,]*\).*/\1/; /^[89a-f].......$/d; G; s/\(.*\)\n.* \(.*\)/\1 \2/; |
  /subl\?.*%.*,%esp/{ G; s/\(.*\)\n\(.*\)/Dynamic \2 \1/; p; }; ' | \
sort | \
perl -e 'while (<>) { if (/^([0-9a-f]+)(.*)/) { $decn = hex("0x" .  $1);\
  if ($decn > 400) { print "$decn $2\n";} } }'
```

Voila un exemple de solution simple permettant de réduire la pression sur la VM et repousser un peu plus loin les limites sur les systèmes chargés.

5.1.2. rmap, objrmp et page clustering

Actuellement un des grands sujets de discutions autour de la VM concerne un problème pour swapper des pages vers le disque. Commençons par voir le problème pour étudier les différentes solu-

tions proposées et leurs conséquences.

Nous avons déjà présenté en quoi consistait le mécanisme de partition d'échange dans la première partie. Le principe en semble fort simple mais il y a cependant plusieurs difficultés cachées. L'une d'elle est notamment qu'un cadre de page peut être référencé par plusieurs pages; ceci se produit lorsque deux pages sont partagées. Lorsqu'une telle page est élue pour être swappée il faut que toutes les références au cadre de page soient éliminées. Le problème vient justement du fait qu'il faut pouvoir localiser rapidement toutes les références.

Dans Linux 2.4 vanilla il n'y a pas de structure de donnée permettant une telle chose. Ainsi l'approche est simple, il faut parcourir toutes les tables de pages du système ce qui peut être très dévastateur. En effet supposons un système avec 100 processus qui partagent 100 Mo de mémoire. Il y a donc approximativement 100 * 1024 * 1024 / 4096 = 21140 entrées de table de pages par processus. Ainsi pour pouvoir déplacer 4K de mémoire vers le disque il faut explorer 100 * 21140 = 2114000 PTEs ce qui représente 2114000 * 8 = 16Mo de mémoire. Ceci peut ne pas sembler énorme mais sur de gros systèmes il n'est pas rare de voir 1000 processus se partager plus d'un giga de mémoire ce qui donne un ordre de grandeur du giga octet à parcourir pour déplacer une page ! Avec un tel système le système va trasher très rapidement.

Plusieurs solutions ont donc été recherchées. La première a été imaginée par Rik Van Riel et est couramment nommée rmap. L'objectif est de maintenir une liste entre les pte faisant référence au même cadre de page. La technique utilisée est de rajouter un champ à la structure page qui pointe vers une liste de pte référençant le même cadre de page. Ainsi lorsque l'on désire supprimer toutes les références à un cadre de page il suffit d'obtenir une page et nous pouvons retrouver les autres avec un coût temporel optimal.

Cependant cette solution pose de sérieux problèmes et principalement deux. Le premier est que la gestion de ces listes coûte relativement cher au système; ainsi en profilant un système avec rmap il est habituel de voir les fonctions gérant les `pte_chain` en haut du tableau. Ainsi les fonctions `page_add_rmap` et `page_remove_rmap` sont particulièrement consommatrices. Mais le plus gros problème n'est pas là. rmap implique de nouveaux champs donc une plus grande consommation mémoire. Ainsi pour chaque page nous avons rajouté 4 octets et 4 ou 8 octets pour chaque pte partagé (en réalité ceci est un peu plus car aligné sur la taille d'une ligne de cache L1). Ceci est d'autant plus problématique sur les grosses machines que les `pte_chain` doivent absolument résider en mémoire basse qui comme nous l'avons vu est limitée à environ 900Mo sur X86. On peut très rapidement se retrouver dans la situation paradoxale ou rmap va consommer toute la mémoire basse est ne résoudra en rien le problème ! Ce problème est moins bloquant sur les architectures 64 bits mais implique quand même une consommation mémoire non négligeable.

rmap est donc une solution fonctionnelle mais peu satisfaisante de par son coût CPU mais surtout mémoire. Notons que sur X86 rmap aura besoin d'être combiné à d'autres solutions comme 4:4 ou le page clustering, que nous verrons plus tard, afin de résoudre le problème de la mémoire basse. rmap a été intégré (rmap_import) à la branche officielle 2.5 il y a deux ans et est donc dans la branche 2.6. De nombreuses améliorations (rmap_improv) ont été effectuées sur les structures de données. L'historique du rmap de Rik Van Riel est disponible (rmap_rik) et s'arrete à la révision 1.40 du fichier `mm/rmap.c`.

Voici un aperçu rapide des changement effectués aux structures de données et leurs relations :

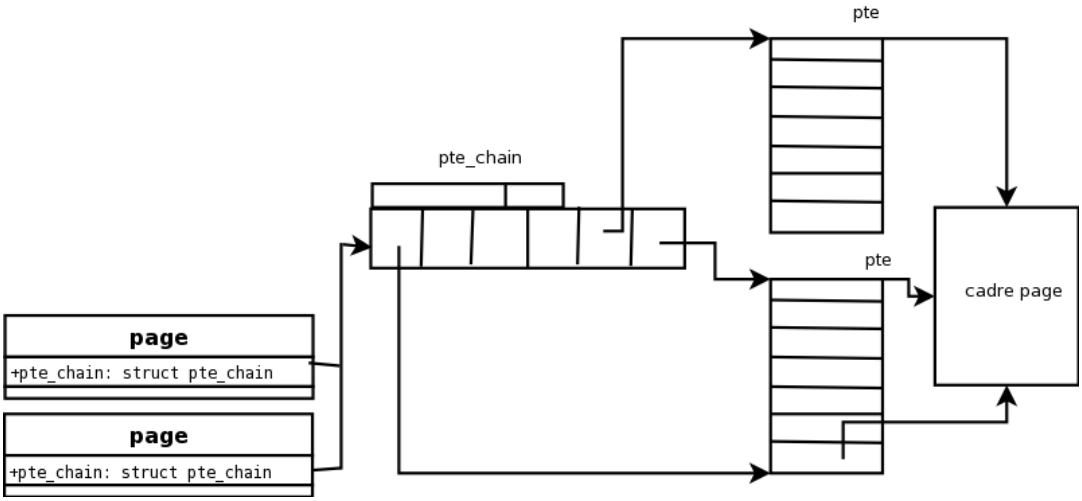
`include/linux/mm.h :`

```
170 struct page {

[...]

179         union {
180                 struct pte_chain *chain;/* Reverse pte mapping pointer.
181                                          * protected by PG_chainlock */
182                 pte_addr_t direct;
183         } pte;

[...]
```

```
200 };
```

mm/rmap.c :

```
55 struct pte_chain {
56         unsigned long next_and_idx;
57         pte_addr_t ptes[NRPTE];
58 } ____cacheline_aligned;
```



Il est facile de retrouver tout les pte référencant le même cadre de page en un temps optimal, il n'y a qu'à parcourir la structure pte_chain

À cause de sa consomation mémoire rmap n'a pas été jugé convenable par un certain nombre de hackeurs Linux qui de plus n'aiment pas la combinaison avec 4:4 qui implique de grosses pertes de performances. Ainsi Dave McCracken, Andrea Arcangeli et Hugh Dickins sont parti dans une voie nommée objrmap. Le principe est qu'il peut exister une voie détournée de trouver tout les pte pointant vers un même cadre de page pour de la mémoire non anonyme. En effet dans une struct page il existe une référence nommée `mapping` qui pointe l'`address_space` de l'objet mappé en mémoire. Nous avons donc accès aux VMA qui comme nous l'avons vu contiennent tout ce qu'il nous faut pour retrouver ce que nous cherchons.

Cette approche simple à tout de même un problème elle ne peut pas fonctionner avec la mémoire anonyme puisqu'il n'y a pas d'objet nommé derrière. Andrea Arcangeli à alors ajouté un patch nommé anon-vma qui permet d'appliquer le même principe à de la mémoire anonyme. L'avantage de objrmap est que le coup mémoire est beaucoup plus limité que rmap puisqu'il y a un surcout par VMA et non par pte. Cependant objrmap à certain problèmes; le principal est qu'il transforme la recherche O(1) d'rmap en O(N) puisque l'on doit effectuer un parcours de toutes les VMA. Or certaines applications pathologiques telles que les bases de données `mmap()` de nombreux petits fichiers ce qui créer beaucoup de VMA. Ainsi ingo Molnar a posté (objrmap_ingo) un petit bout de code qui gèle très facilement la machine à cause de cet algorithme linéaire.

On peut aussi noter que Hugh Dickins est en train d'implémenter sa propre version d'objrmap.

Pour le 2.6.7 linus torvalds vient d'intégrer les fonctions nécessaires à objrmap sans toutefois trancher entre les deux implémentations ni avoir résolu le problème de la complexité introduisant ainsi un déni de service facilement exploitable.

À plus long terme, c'est-à-dire lorsqu'une nouvelle branche de développement sera créée, une troisième solution est envisageable; il s'agit du "page clustering". Sous ce terme un peu barbare se cache le concept de regrouper plusieurs pages physiques en une plus grosse page logique ce qui implique automatiquement un moins de mémoire gaspillée par les structures de données. Il est maintenant assez fréquemment admis que les pages de 4K ne sont plus adaptées pour les quantités de mémoire dont disposent les machines actuelles, le page clustering gérerait de manière intéressante le problème permettant à l'utilisateur de choisir le bon compris entre fragmentation interne et

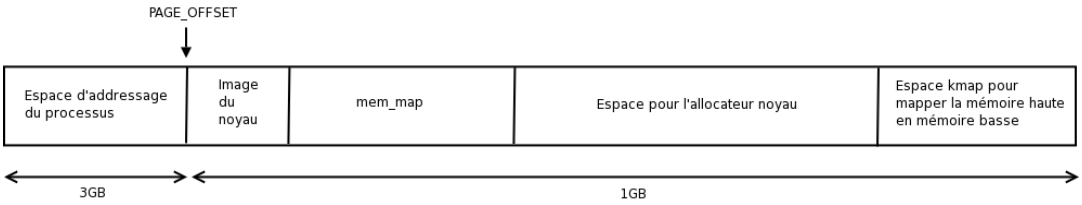
gaspillage de mémoire par le système. Avec un tel système rmap ne poserait plus vraiment de problème puisque l'on pourrait adapter sa consommation mémoire (plus la machine dispose de mémoire plus la taille des pages est importante). Cette solution n'est pour le moment pas envisageable car elle implique trop de modifications en profondeur du noyau.

Au travers de ce problème on peut donc entrevoir la difficulté de l'implémentation de la mémoire virtuelle car elle nécessite de trouver des structures peu consommatrices de mémoire, rapide et donc complexe. À l'heure actuelle il est impossible de dire quelle solution va finalement être intégrée et de quelle manière.

Le lecteur intéressé par ces solutions trouvera, en plus du code source, de bonnes introductions aux différentes solutions dans les articles suivants : mémoire basse (lwn1), rmap (lwn2), objrmap (lwn_objrmap et lwn_objrmap2), page clustering (lwn_pc) et une mine d'information sur la situation actuelle est disponible dans l'interview d'Andrea Arcangeli par kerneltrap (AA).

5.1.3. Séparation 4:4

Dans la présentation de rmap et objrmap nous avons évoqué le problème de la saturation de la mémoire basse par les chaînes des ptes. Cependant nous avons écarté un autre problème qui conduit aux mêmes conséquences. Rappelons nous de l'aspect de la mémoire sous Linux



Vue simplifiée de l'espace d'adresse sous Linux

Le point qui nous intéresse est qu'ici après l'image du noyau se trouve une structure nommée `mem_map` qui est un tableau de struct page de longueur équivalente au nombre pages physiquement présentes au moment du démarrage. Sur une machine dotée de plus de 32GB de mémoire cette structure commence à être problématique par exemple lorsque l'on démarre une machine possédant 64GB de RAM il reste 176MB de mémoire basse utilisable par le noyau ce qui ne permet pas de supporter correctement des charges élevées; une machine avec 64GB est tout de même sensée faire quelque chose ! Si l'on rajoute rmap et les autres structures de données on arrive au fait qu'une telle machine est incapable de démarrer ou de faire quelque chose.

Pour résoudre ce problème il existe deux approches orthogonales. Nous avons déjà rapidement expliqué la première dans la partie précédente il s'agit du page clustering. En utilisant des pages de plus grande taille on diminue le nombre de pages vues par le système et donc le problème se résout de lui même. William Lee Irwin indique qu'avec son patch il reste alors 750MB de mémoire basse disponible ce qui n'est pas énorme mais néanmoins suffisant pour fonctionner correctement. Pour le moment il existe de tel patch pour Linux 2.4, Hugh Dickins, et Linux 2.6, William Lee Irwin. Ceux-ci restent toutefois expérimentaux et peu utilisés.

La deuxième solution, temporaire, a été proposée par Ingo Molnar et est utilisée en production sur la RLE 3.0 par exemple. Cette solution n'est pas très élégante et impose un surcout de 0 à 30% à l'exécution. Voici comment cela fonctionne. Traditionnellement l'espace d'adressage est coupé en deux, du début de la mémoire à PAGE_OFFSET se trouve l'espace d'adressage du processus et de PAGE_OFFSET à la fin correspond l'espace d'adressage du noyau. La seconde partie est mise identiquement dans l'espace d'adressage de tout les processus. PAGE_OFFSET correspond généralement à 0xC0000000 ce qui fait un découpage 3/1. Pour donner plus d'espace au noyau on peut décider de faire un découpage 2/2 en décalant PAGE_OFFSET, cependant ceci n'est utilisé que pour les petites quantités de RAM, typiquement 2GB, puisqu'il réduit l'espace d'adressage des processus à 2GB. L'approche choisi par Ingo Molnar est de faire un découpage 4/4 !

4/4 semble impossible puisque l'espace d'adressage sur X86 est de 32 bits. Toutefois il y a une ruse et le patch modifie en profondeur certain aspects de la VM. Tout d'abord 4/4 n'est pas vraiment le nom vraiment approprié il s'agit plutôt de 4G/16M, désormais l'espace utilisateur occupe entièrement les 4GB (moins 16MB comme nous le verrons) et l'espace d'adressage du noyau fait

aussi 4GB. Puisque les processus ont toujours besoin d'appeler du code noyau; dans les 16MB les plus haut se trouvent déportés un certain nombre de structures et une code trampoline permettant de changer l'espace d'adressage pour passer en mode noyau. Typiquement les structures déplacées sont la GDT, l'IDT, le TSS, la LDT, la page contenant l'adresse des appels système et la fonction permettant la bascule dont j'ai parlé.

Lorsqu'un appel système survient, le code trampoline est exécuté et le système charge l'espace d'adressage du noyau qui peut alors travailler. Ce changement a bien entendu un coût; à chaque appel système le TLB doit être vidé ce qui inclue une double pénalité. L'entrée dans en mode noyau est ralentie à cause de la manipulation du registre CR3, et le vidage du TLB implique des "cache miss" fréquents.

Il reste un certain nombre de problèmes à adresser comme copier des données de l'espace utilisateur vers l'espace noyau ce qui est effectué en modifiant `kmap()` et quelques autres parties de la VM.

Malgré la pénalité à l'exécution ce patch est intéressant car il résout le problème de la mémoire basse saturée mais permet aussi aux processus de disposer d'un espace d'adressage de 4GB. Encore une fois il serait plus judicieux de passer à une architecture 64 bits...

5.2. Support NX, No eXecute, par page

Depuis quelques temps la sécurité commence à devenir un besoin pour les entreprises et les particuliers, depuis trop longtemps des failles très connues sont exploitables bien souvent à cause de programmeurs non formés aux rudiments d'une programmation sécurisée. Bien que la plupart des nouveaux langages de programmation limite les problèmes les failles de type dépassement de tampon (buffer overflow, BO) ne restent que trop nombreuses. L'explication du fonctionnement des BOs sort du cadre de cet article mais pour résumer on peut dire qu'il s'agit d'exploiter une erreur de programmation qui permet d'écrire plus loin que la fin du tampon et ainsi d'aller écrire sur des parties sensibles de la mémoire comme par exemple changé la valeur du registre EIP sur la pile (mais il existe bien d'autre techniques).

Les cas simples de BOs qui consistent à placer du code sur la pile ne devrait tout simplement être possible. Il n'y a pas de raison que cette mémoire soit exécutable et traditionnellement elle ne l'était pas, la pile étant un segment de donnée non exécutable. Cependant le modèle d'espace d'adressage plat d'UNIX à fait que l'on utilise des protections par page et non les protections fournies par les segments. Le problème étant qu'un certain nombre de processeurs fournissent un protection correcte (lecture, écriture, exécution) comme les AMD64, Sparc64, sparc, powerpc, alpha, sh5, hppa alors que d'autre ne fournissent qu'un support partiel i386, powerpc et d'autres aucune protection par page arm, m68k, mips, sparc, vax.

Si sur la première classe en théorie il n'y a aucuns problème pour distinguer les zones exécutables de zone non exécutables une certaine "paresse" à fait que ce support n'a pas été effectué sur bon nombre d'OS ainsi le support dans Windows ne sera effectif qu'à partir de XP SP2 et un patch pour Linux vient d'être proposé par Ingo Molnar.

Sur les autres architectures le problème est un peu plus difficile à maîtriser et différentes approches peuvent être utilisées plus ou moins efficaces et plus ou moins rapides. Actuellement tout les concepteurs de processeurs (non embarqué) ont fait des annonces comme quoi leur processeur supporterait les protections contre l'exécution par page.

Toutefois la protection contre l'exécution est condition nécessaire mais non suffisante il est facile de faire un "return into libc" qui permet d'utiliser une fonction des bibliothèques dans l'espace d'adressage par exemple appeler `system()` ou `exec()`. Il existe des patchs complets qui visent d'une part à supporter le drapeau NX la où il est disponible ou l'émuler si besoin et rendre aléatoire une certain nombre d'adresses pour compliquer l'écriture de "shell code". Nous allons rapidement présenter deux solutions l'une pour Linux, l'autre pour OpenBSD.

5.2.1. PaX

L'auteur de PaX introduit le projet ainsi :

```
The goal of the PaX project is to research various defense mechanisms against
the exploitation of software bugs that give an attacker arbitrary read/write
access to the attacked task's address space. This class of bugs contains among
others various forms of buffer overflow bugs (be they stack or heap based),
user supplied format string bugs, etc.
```

Il ne s'agit donc pas uniquement d'un support NX mais de la recherche d'un ensemble de techniques permettant d'empêcher l'exploitation de failles dans les logiciels utilisateurs.

Pour rendre la pile, le tas et les fichiers `mmap()`és sans PROT_EXEC non exécutable sur X86 deux méthodes ont été implémentée

5.2.1.1. PAGEXEC

PAGEXEC propose d'implémenter une protection par page sur une architecture ne disposant pas matériellement de cette possibilité. Le principe repose dès lors sur une astuce et implique une baisse de performance. Voyons l'astuce utilisée

Les processeurs X86 récents, c'est-à-dire à partir du pentium et du K5, il existe en réalité deux TLB; un pour les données l'autre pour les instructions. La base de PAGEEXEC se repose sur ce fait. Le but est d'empêcher qu'une page non executable ne soit chargée dans le TLB d'instruction. La technique consiste à pouvoir controller logiciellement ce qui sera chargé dans le TLB, cependant les processeurs X86 ne permettent pas un tel remplissage matériel du TLB. Ainsi l'astuce est de force la matériel a passer la main au logiciel; comme nous l'avons vu dans la partie gestion des défauts de page si le matériel n'arrive pas a charger le pte dans le TLB alors il lève une exception. Nous allons donc forcer la levée d'un défaut de page à chaque tentative, ce qui peut être fait facilement en positionant le bit non-présent ou superviseur des pte. Dans PAGEEXEC le bit non présent à été retenu.

Maintenant à chaque fois que le TLB sera rempli il passera la mais à la fonction `do_page_fault()`. L'intêret est que l'on peut vérifier par un moyen détourner si la donnée ira dans le TLB de données ou d'instructions. En effet l'état de la tâche qui à été interompue à été placé dans un segment TSS. On peut donc consulter le registre `EIP`, qui correspond au compteur ordinal sur X86, et vérifier si l'adresse ayant générer le défaut de page est la même qui celle d'EIP. Le cas échéant la donnée sera placé dans le TLB d'instruction.

Il est dès lors facile de permettre le chargement du pte en repositionant le bit présent ou de tuer la tache si une tentative d'execution à été fait sur une page non éxécutable.

Le coût de cette approche est élévée puisque des tests indiquent a peu près 20% de pénalité ce qui fait que cette méthode est rarement utilisée en production. On trouvera dans pageexec une description du procédé par son auteur.

5.2.1.2. SEGMEXEC

Contrairement à PAGEEXEC, SEGMEXEC n'essait pas de mettre en place de protection par page. Il utilise plutôt la protection par segmentation offerte nativement par le processeur. Cela a pour avantage de n'engendrer presque aucune perte de performances.

En temps normal Linux n'utilise pas la pagination et se contente de créer deux segments de 4GB couvrant tout l'espace d'adressage, l'un pour les données l'autre pour les instructions; les deux se recouvrant. Puisqu'il n'est pas possible de désactiver la segmentation cette méthode permet de faire comme s'il n'y en avait pas.

SEGMEXEC propose de revoir ce schéma en découpant l'espace d'adressage d'un processus en deux. Un segment de donnée d'1.5GB est creer et un segment d'instruction d'1.5GB est créé. Ainsi les tentatives de lecture / écriture et d'exécution utilise des adresses linéaire différentes et l'on peut facilement controller leur exécution. Reste qu'il faut que le code executable soit aussi accessible en lecture il faut alors faire de la duplication de VMA entre les deux segments.

SEGMEXEC à l'avantage de proposer une solution de coût presque nul en performance mais impose des restrictions au niveau de l'espace d'adressage. On ne peut plus se servir des 3GB comme bon

nous semble et cela peut être problématique selon l'utilisation de la machine. C'est cependant la solution la plus courament utilisée; on en trouvera un description plus approfondie par son auteur dans segmexec.

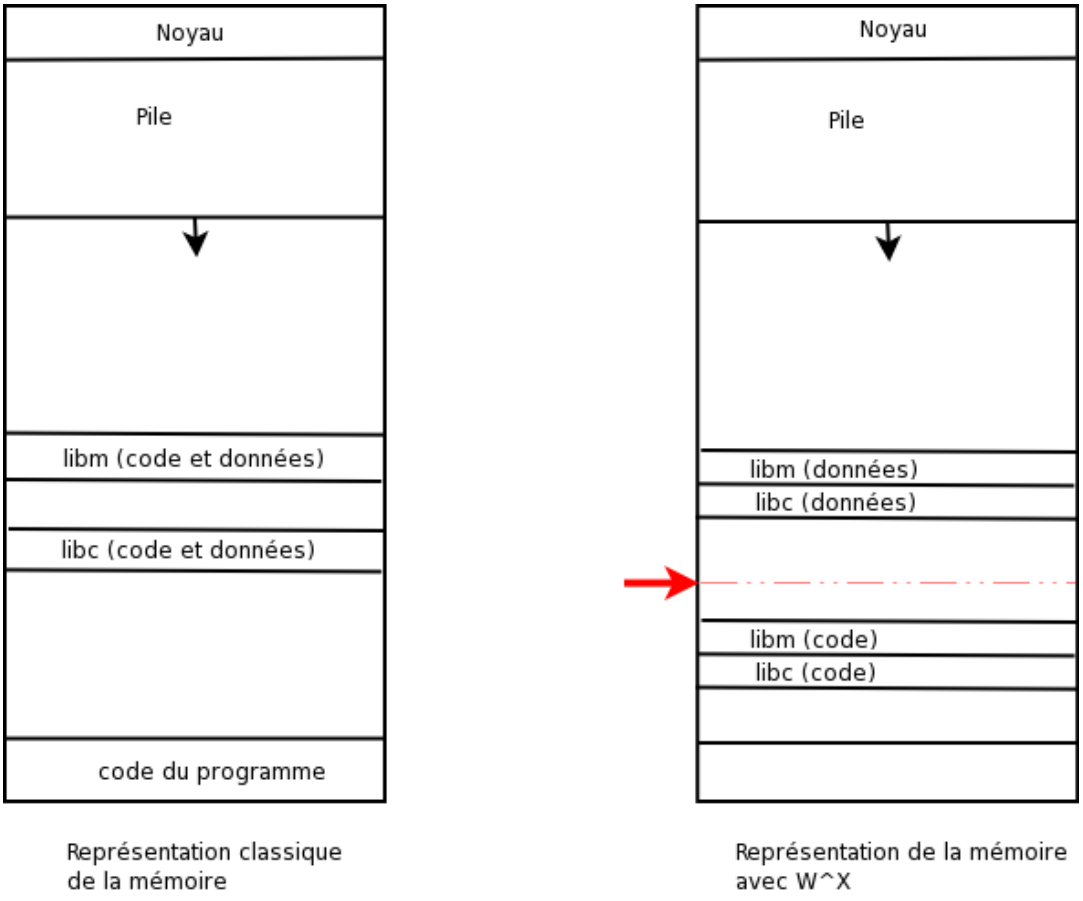
Bien entendu ces protections ne sont pas suffisante et doivent être couplées à d'autres méthodes offertes par PaX pour aboutir à une protection satisfaisante. La description de toutes ces techniques serait trop longue pour ce document et on se reportera à l'excellente documentation, docpax, de PaX pour en apprendre d'avantage.

5.2.2. W^X

W^X, write xor execute, essait de jouer le meme rôle que PaX pour OpenBSD. L'implémentation est plus récente, moins aboutie et peu documentée chose qui semble se répéter éternelement chez OpenBSD... Pour la partie qui nous interesse, équivalente à celle que nous avons documenté pour PaX les résultats sont équivalent cependant la conception générale de la protection semble déffectueuse comme indiqué dans paxvswx.

Mais concentrons nous sur la description de l'implémentation de pages non exécutables sur X86. Hormis le code la seule documentation existante se résume à quelques messages bugtraq ou les listes d'OpenBSD. Nous utiliserons donc ces informations.

Le principal problème rencontré pour implémenter les pages non exécutables sont les bibliothèques partagés; en effet celles-ci sont constituées d'un partie de code et d'une partie de donnée. Il n'est pas évident d'utiliser la segmentation pour résoudre ce problème. PaX à gérer ce problème en utilisant une duplication de VMA pour recopier le code des bibliothèques dans le segment d'instructions. L'approche de R^W est différente puisqu'elle propose de remanier toute la chaîne de compilation/liaison/chargement pour séparer le code des données.



Schématisation de l'aspect de d'adressage. On constate qu'avec W^X le trait rouge sépare l'espace d'adressage en deux zones, l'une exécutable l'autre non.

Avec la nouvelle structure on constate que les données exécutables sont séparées des données en lecture/ecriture. On peut ainsi utiliser la segmentation de manière efficasse. Il suffit de positionner le segment de code du début de l'espace d'adressage jusqu'au trait rouge. L'avantage sur SEGMEXEC est qu'il est facile de modifier les segments et ainsi adapter la répartition code/données dynamiquement.

Cependant il y a une difficulté cachée, une bibliothèque dynamique est un module ELF pré-relié et les données doivent rester à des distances fixées à l'avance. Ainsi la distance entre le code et les données d'une bibliothèque doivent toujours être séparés par la même distance. Ceci pose quelques problèmes pour rendre aléatoire la position des bibliothèques, c'est une des raisons pour laquelle PaX est plus éfficasse sur ce point.

L'approche de R^W modifie moins le noyau, plus la chaîne de compilation/exécution et casse entièrement la compatibilité binaire.

Glossaire

Ici se trouve un rapide glossaire des termes relatifs à la gestion mémoire dans les systèmes d'exploitations. Ce glossaire est très fortement inspiré de diverses sources comme KUS ou l'excellent glossaire de memorymanagement.org [http://www.memorymanagement.org/glossary/].

A

adresse mémoire

: Un nombre spécifiant un emplacement mémoire. Les adresses mémoire sont souvent classées en adresses virtuelles ou physiques, selon qu'elles désignent la mémoire physique ou la mémoire virtuelle.

Algorithme LRU

: Politique de remplacement qui consiste à évincer en premier les objets accédés le moins récement. Ce principe repose souvent sur le principe de localité.

B

Bloc de contrôle de processus (PCB)

: [Process control block] Structures de données contenant le contexte d'un processus. Le PCB défini par le matériel contient la partie matérielle de ce contexte. Le PCB logiciel contient la partie logicielle et se trouve en mémoire, immédiatement après le PCB matériel.

C

Copie à l'écriture (COW)

: [Copy on write] Technique par laquelle des références multiples à un objet commun sont gardées jusqu'à ce que l'objet soit modifié (écrit). Avant l'écriture de l'objet, une copie est faite; la modification est faite sur la copie plutôt que sur l'original. Dans la gestion de la mémoire virtuelle , la copie à l'écriture est une méthode courante que le noyau utilise pour gérer les pages partagées par plusieurs processus. Toutes les entrées des tables de pages mappant une page partagée sont positionée de manière à ce que la première référence à la page provoque un défaut de page. Lorsque le défaut de page est traité, la page est remplacé

avec une copie privée, qu'il est possible d'écrire.

E

Ensemble de travail

[Working set] L'ensemble des pages dans l'espace d'adressage virtuel d'un processus auxquelles des références ont été faites lors des derniÃ¨res secondes écoulés. La plupart des processus-montrent une certaine localité de référence et la taille de leur ensemble de travail est typiquement moins que la moitié de leur taille totale en mémoire virtuelle.

Entrée de table de page (PTE)

[Page Table Entry] Les structures de données dépendantes de la machine qui identifient l'emplacement et l'état d'une page dans l'espace d'adressage virtuel. Lorsqu'une page virtuelle est en mémoire, la PTE contient le numéro de cadre de page dont le matériel à besoin pour mapper la page virtuelle à une page physique.

Espace d'adressage virtuel

[Virtual address space] Un intervalle continu d'emplacements ne mémoire virtuelle.

M

Mémoire virtuelle

Une fonctionnalité permettant à l'intervalle effectif d'emplacements mémoire adressable fourni à un processus d'être indépendant de la taille de la mémoire principale; c'est-à-dire que l'espace d'adressage vbirtuel d'un processus est indépendant de l'espace d'adressage physique du processeur.

P

Page verouillée

La mémoire qui n'est pas remplacée par le démon de pagination. La mémoire phsique est assignée à un intervalle no paginable d'adresses virtuelles lorsque les adresses sont allouées. Les pages verrouilées ne provquent jamais un défaut de page pouvant résulter en une opération bloquante. Loes pages verrouillées sont utilisées typiquement dans l'espace d'adressage du noyau.

Pagination

L'action d'apporter les pages d'un processus exécuté en mémoire principale lorsqu'elles sont référencées et de les enlever de la mémoire loesqu'elles sont remplacées. Loesqu'un processus s'exécute, toutes ses pages résident en mémoire virtuelle; cependant seules les pages en activité ont besoin de résider en mémoire principale. Les pages restantes peuvent être sur disque jusqu'à ce qu'elles soient nécéssaire

Pagination à la demande

Une technique de gestion de la mémoire pour laquelle la mémoire est divisée en pages et les pages fournies aux processus lorsque c'est nécessaire, c'est-à-dire à la demande.

Politique de remplacement

Uune politique utilisée par le système de mémoire virtuelle pour choisir les pages à réutiliser lorsque la mémoire n'est pas disponible autrement.

Politique de ramplacement optimale

Une politique de remplacement qui entraine des performances optimales. Il s'agit de connaitre à l'avance le modèle d'exécution d'un processus de manière à pouvoir minimiser le nombre de défauts de page. Ce type de politique n'est généralement pas applicable et sert de modèle de référence pour pouvoir comparer différents algorithmes.

PAE

PAE est une extension sur les processeurs Intel qui permet d'avoir jusqu'à 64GB de mémoire principale sur des processeurs 32 bits. Cela est possible en étendant le bus d'adresse à 36 bits. Toutefois un processus dispose toujours d'un espace d'adressage de 4GB.

PSE

PSE est une extension sur les processeurs Intel qui permet de disposer de page de 4MB plutôt que 4KB.

T

Translation Lookaside Buffer

bibliographie

Livres

[BOV] Daniel P. BOVET and Marco CESATI. Copyright © 2001 O'reilly & associates. Éditions O'Reilly. *Le noyaux linux*. Des ports E/S à la gestion des processus. 35-63,155-225,445-483.

[KUS] Marshall Kirck McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. Copyright © . Addison Wesley. *Conception et implémentation du système 4.4BSD*. 111-179.

[OSX] Apple Computer. Copyright © 2003 Apple. Apple. *Kernel programming*. Mac OS X. 57-71.

[TAN] Andrew Tanenbaum. Copyright © 2001 Prentice hall. Pearson education. *Systèmes d'exploitation*. seconde édition. 201-285,750-764,858-864.

Papiers

[BEN] A. Bensoussan, C. T. Clingen, and R. C. Daley. Copyright © 1972. ACM. *The Multics Virtual Memory*. Concepts and Design. 308-318.
[http://www.multicians.org/multics-vm.html]

[BON01] Jeff Bonwick. Copyright © 2001. USENIX. *Magazines and Vmem*. Extending the Slab Allocator to Many CPUs and Arbitrary Resources.

[BON94] Jeff Bonwick. Copyright © 1994. USENIX. *The Slab Allocator*. An Object-Caching Kernel Memory Allocator.

[CHA] Howard Chartock and Peter Snyder. *Virtual Swap Space in SunOS*.

[http://ragondin.ath.cx/Operating_Systems_Repository/memory/Sun/chartock91virtual.ps]

[CRA] Charles D. Cranor. Copyright © 1998. *Design and Implementation of the UVM Virtual Memory System*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/BSD/disstalk.ps]

[DEN70] P. J. Denning. Copyright © 1970. ACM. *Virtual Memory*. ACM Computing Surveys vol 2. 153-190.

[DEN72] P. J. Denning and S. C. Schwartz. Copyright © 1972. ACM. *Properties of the Working-set Model*. CACM vol. 15 no. 3. 191-198.

[DEN68] P. J. Denning. Copyright © 1968. ACM. *The working set model for program behavior*. CACM 11. 323-333.

[DEN68b] P. J. Denning. Copyright © 1968. AFIPS. *Trashing: Its Causes and Prevention*. Proceedings AFIPS, 1968 Fall Joint Computer Conference vol. 33. 915-922.

[DEN96] P. J. Denning. Copyright © 1996. George Mason University. *Before the memory was virtual*.

[] S. B.. . Copyright © 1965. ACM. *Segmentation and the design of multiprogrammed computer systems*. J.ACM 12, 4. 589-602.

[FREE] Matthew Dillon. FreeBSD Documentation Project. *FreeBSD developper Handbook*. Chap 15 Virtual Memory System.
[http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/vm.html]

[GIN] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. *Virtual Memory Architectures in SunOS*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/Sun/gingell87virtual.ps]

[GOR] mel Gorman. Copyright © 2003 mel Gorman. *Understanding the Linux Virtual Memory Manager*.

[GOR2] mel Gorman. Copyright © 2003 mel Gorman. *Code commentary of the Linux Virtual Memory Manager*.

[JAC97] Bruce Jacob and Trevor Mudge. Copyright © 1997. University of Michigan. *Memory management Hardware, and its Support for Operating Systems*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/jacob97memory.ps]

[JAC97b] Bruce Jacob and Trevor Mudge. Copyright © 1997. University of Michigan. *Software-Managed Address Translation*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/jacob97softwaremanaged.ps]

[JAC98] Bruce Jacob and Trevor Mudge. Copyright © 1997. *Virtual Memory in contemporary microprocessors*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/jacob98virtual.ps]

[KUS88] Marshall Kirk McKusick. Copyright © 1988. *Design of a general purpose memory allocator for the 4.3BSD UNIX Kernel*.

[LIE94] Jochen Liedtke. Copyright © 1994. *Page Tables Structures For Fine-Grain Virtual Memory*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/liedtke94page.ps]

[LIE] Jochen Liedtke. Copyright © 1994. *Limitation of Page Table Translation Schemes*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/limitations-of-page-table-1.ps]

[MOR] Joseph P. Moran. *SunOS Virtual Memory implementation*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/Sun/moran88sunos.ps]

[OS7] Apple Computer. Copyright © 1998 Apple. Apple. *Technical Note TN1094*. Virtual Memory application compatibility.

[RAS] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, , Robert Baron, David Black, William Bolosky, and Jonathan Chew. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*.
[http://ragondin.ath.cx/Operating_Systems_Repository/memory/Sun/rashid87machineindependent.ps]

Sites

[] Rik van Riel. Copyright © 2003. *Too little, too slow*. An introduction to memory management and Linux memory management today and tomorrow.
[http://www.surriel.com/lectures/mm.html]

[] *Memory management Glossary*.
[http://www.memorymanagement.org/glossary/]

[] *Virtual Memory tutorial*.
[http://cne.gmu.edu/modules/vm/submap.html]

Références sur rmap, objrmap, 4:4

[AA] *Interview d'andrea Arcangali par kerneltrap*.
[http://kerneltrap.org/node/view/3148]

[lwn1] *Virtual Memory I: the problem*.
[http://lwn.net/Articles/74295/]

[lwn2] *Description du rmap de linux*.
[http://lwn.net/2002/0124/kernel.php3]

[lwn_objrmap] *The object-based reverse-mapping VM*.
[http://lwn.net/Articles/23732/]

[objrmap_ingo] *Re: -core-1 (rmap removal for file mappings to avoid 4:4 in >=16G machines)*.
[http://lwn.net/Articles/75225/]

[lwn_objrmap2] *Virtual Memory II: the return of objrmap*.
[http://lwn.net/Articles/75198/]

[lwn_pc] *Page clustering*.
[http://lwn.net/Articles/23785/]

[lwn_44] *Patch: 4G/4G split on x86, 64 GB RAM (and more) support*.
[http://lwn.net/Articles/39283/]

[rmap_import] *Import du patch de RvR*.
[http://linux.bkbits.net:8080/linux-2.5/patch@1.403.147.41]

[rmap_improv] *Ameilloration des structures de données d'rmap*.
[http://linux.bkbits.net:8080/linux-2.5/diffs/mm/rmap.c@1.13?nav=index.html|src/|src/mm|hist/mm/rmap.c]

[rmap_improv2] *Ameilloration des structures de données d'rmap*.
[http://linux.bkbits.net:8080/linux-2.5/diffs/mm/rmap.c@1.25?nav=index.html|src/|src/mm|hist/mm/rmap.c]

[rmap_rik] *Historique de `rmap.h`*.
[http://linux.bkbits.net:8080/linux-2.5/hist/mm/rmap.c?nav=index.html|src/|src/mm]

Références sur PaX

[docpax] *Documentation de PAGEEXEC*.
[http://pax.grsecurity.net/doc/]

[pageexec] *Documentation de PAGEEXEC*.
[http://pax.grsecurity.net/doc/pageexec.txt]

[segmexec] *Documentation de SEGMEXEC*.
[http://pax.grsecurity.net/doc/segmexec.txt]

[paxvswx] *Comparaison de PaX et W^X par spender*.
[http://www.grsecurity.net/PaX-presentation_files/frame.htm]

Comparaisons de différents processeurs

Table A.1. Caractéristiques de différents processeur par rapport à la VMM.

	IA-32	**AMD64**	**Alpha**	**MIPS**	**PowerPC**	**PA-Risc**
Taille des addresses	32 (36 en mode PAE)	48	jusqu'à 64 selon la taille des pages	64	52	96
Remplissage du TLB	matériel	matériel	logiciel	logiciel	matériel	logiciel
Support de la pagination	Oui	Oui	Oui	Oui	Oui	Oui
Support de la segmentation	Oui	Non			Oui	
Proctection contre l'exécution	Par segment	Par page	Par page	pas de protection	Par page	Par page