Systèmes d'exploitation

Gestion de la mémoire

Pilot Systems - Gaël Le Mignot

INSIA SRT - 2007

Table des matières

1 Principes de la mémoire virtuelle **3**
1.1 Introduction . . . 3
1.1.1 Les différents problèmes à résoudre . . . 3
1.1.2 Quelques points de théorie . . . 3
1.2 La segmentation . . . 5
1.2.1 Principes de la segmentation . . . 5
1.2.2 Utilisations de la segmentation . . . 5
1.2.3 Avantages et limites de la segmentation . . . 6
1.3 La pagination . . . 7
1.3.1 Principes de la pagination . . . 7
1.3.2 Utilisations de la pagination . . . 7
1.3.3 Implémentation de la pagination . . . 9
1.3.4 Avantages et limites de la pagination . . . 11

2 Algorithmes de pagination **12**
2.1 Les différents enjeux . . . 12
2.2 L'algorithme NRU (*Not recently used*) . . . 12
2.3 L'algorithme d'horloge . . . 13
2.3.1 L'algorithme FIFO (*First-in first-out*) . . . 13
2.3.2 L'algorithme deuxième chance . . . 13
2.3.3 L'algorithme d'horloge . . . 13
2.4 L'algorithme LRU « Least Recently Used » . . . 13
2.4.1 Le LRU théorique . . . 13
2.4.2 L'algorithme NFU . . . 13
2.4.3 Une approximation de LRU : *Aging* . . . 13
2.5 Introduction à la pagination sous Mach . . . 14
2.5.1 Le fonctionnement des pagers externes . . . 14
2.5.2 Quelques exemples d'application . . . 15

Table des figures

1.1 Hiérarchie des mémoires 4
1.2 Exemple de fragmentation 6
1.3 Exemple de pagination 8
1.4 Exemple d'entrée de table de pages 9
1.5 Pagination à deux niveaux 11

2.1 Éviction d'une page sous Mach 14
2.2 Restauration d'une page sous Mach 15

Chapitre 1

Principes de la mémoire virtuelle

1.1 Introduction

1.1.1 Les différents problèmes à résoudre

Avant de considérer le modèle théorique général de la mémoire virtuelle, puis les manières de le mettre en pratique, voyons les différents problèmes qu'elle doit résoudre.

La relocation

Un programme contient un certain nombre de références à des variables, fonctions ou zones du code. Ces références sont parfois en relatif, mais souvent en absolu. Dans le cadre de la multiprogrammation, si le programme est chargé à un endroit différent de la mémoire, ces références absolues ne sont plus valables.

Il existe plusieurs solutions pour résoudre ce problème, une consiste à faire de la relocation dynamique au démarrage du programme : chaque adresse se voit réécrite suivant l'endroit exact où le programme a été chargé. Il est aussi possible, avec un coût non négligeable, de faire de la relocation dynamique lors de chaque accès.

Le swap

Le deuxième problème à adresser est la possibilité de mettre sur le disque dur un certain nombre de zones mémoires, lorsque la mémoire centrale est pleine. Cette fonctionnalité est indispensable dans les systèmes d'exploitation généralistes, même si elle peut être inutile dans des cas précis.

La protection mémoire

Le troisième problème est celui de la protection mémoire, il doit être nécessaire de faire en sorte qu'un processus ni puis pas lire, en encore moins écrire, dans la mémoire d'une autre application ou du système lui-même.

La mémoire partagée

Le quatrième problème à adresser est celui de la mémoire partagée, que l'on a évoqué dans le chapitre sur l'IPC. Afin d'avoir de la communication rapide entre deux processus, il est nécessaire de prévoir un mécanisme pour avoir de la mémoire partagée, sans pour autant détruire la protection mémoire.

Les fichiers mmapés

Le dernier problème, moins critique, est de fournir aux programmes la capacité de projeter un fichier en mémoire, et de l'utiliser directement comme s'il s'agissait de données en mémoire. Ce mécanisme, nommé "mmap" permet à un programme d'avoir la flexibilité et la rapidité de données en mémoire, tout en ayant la persistance.

1.1.2 Quelques points de théorie

La hiérarchie des mémoires

Tout d'abord, un petit rappel sur l'architecture du matériel. Dans un monde parfait, la mémoire est rapide d'accès, disponible en grande quantité, ne coûte pas cher, et est rendue persistante à volonté. Hélas, un tel

monde n'existe pas. Dans la réalité, la mémoire est soit rapide et très chère, soit bon marché mais très lente. Et la mémoire rapide n'est jamais persistante.

Afin d'obtenir des performances correctes sur un ordinateur, nous sommes donc obligés d'utiliser toute une série de mémoires, certaines très rapides mais en quantité très faible, d'autres lentes mais en quantité très élevée.

Type de mémoire	Taille typique	Vitesse typique
1. Registres	256 octets	0.5 ns
2. Cache L1	16 Ko	1 ns
3. Cache L2	2 Mo	2 ns
4. Mémoire centrale	2 Go	10 ns
5. Disque dur	300Go	10 ms

Fig. 1.1 – Hiérarchie des mémoires

Une manière de considérer toute cette hiérarchie de mémoire est de la voir comme une hiérarchie géante de caches. Les registres servent de cache au cache de niveau un, qui lui-même sert de cache au cache de niveau 2, qui lui-même cache la mémoire centrale elle-même n'étant qu'un cache pour le disque dur.

Le système d'exploitation n'a que peu de contrôle sur le fonctionnement des caches gérés automatiquement par la matériel (L1 et L2), ni sur les registres (qui eux sont gérés par le compilateur lors de la compilation du programme). Un concepteur de système (ou une personne tentant d'en optimiser un) doit cependant être conscient de leur existence, puisqu'un changement de contexte trop fréquent, par exemple, peut nuire fortement à leur efficacité.

Là où le système d'exploitation intervient à plein est sur la gestion de la mémoire centrale, c'est en effet à lui qu'incombe (en général) de décider ce qui doit être gardé en mémoire et ce qui doit être uniquement sur le disque dur.

La notion de *backing store*

Il est bien important de comprendre que pour tout système de gestion de la mémoire suffisamment évolué, les notion de swap et de cache disque sont fusionnées. Les deux consistent à utiliser la mémoire comme un cache pour le disque : dans le premier cas, on peut libérer de la mémoire en l'écrivant sur le disque, dans le second cas, on peut utiliser la mémoire disponible pour éviter des accès disques par la suite.

Il existe bien sûr quelques différences, par exemple, une modification sur un fichier doit être écrite sur le disque tôt ou tard pour éviter une perte de données en cas de redémarrage ou coupure de courant, tandis qu'une zone de données d'un programme n'a jamais besoin d'être écrite. Mais sur le fond, les deux peuvent en effet être gérés de manière unifiée.

Tout ceci se réduit à la notion de *backing store*. La mémoire n'est cache pour le disque dur. Chaque zone mémoire, dépendant de son utilisation (bibliothèque partagée, exécutable, données, cache disque) est associée à un *backing store* qui est la zone du disque qui permet de la stocker si nécessaire.

Il peut arriver qu'une zone mémoire soit plus grande que son backing store (par exemple une machine équipée de 2Go de mémoire mais dont le swap a été limité à 1Go), le système doit être capable de gérer ces cas, mais ils ne modifient pas le modèle théorique.

Enfin il est à noter que l'utilisation d'une partition brute comme backing store est souvent bien plus efficace que l'utilisation d'un fichier (avec tous les soucis de fragmentation, . . .associés). Mais bien sûr, ce n'est souvent pas possible (en pratique, seul le swap est sur une partition externe).

La MMU

Pour les solutions que nous allons voir par la suite (segmentation et pagination), il est nécessaire d'obtenir de l'aide du matériel. En effet, effectuer des traitements logiciels sur chaque accès mémoire serait bien trop coûteux en temps, et difficile à réaliser (puisque le logiciel a besoin lui-même d'accéder à la mémoire).

Il existe donc une entité matérielle spéciale nommé *Memory Management Unit* ou MMU en abrégé, qui se situe théoriquement entre le CPU et la mémoire. Cette unité effectue une traduction entre l'adresse virtuelle demandée par la CPU et l'adresse physique comprise par le matériel.

En pratique, la MMU est quasiment toujours intégrée au CPU et accède directement aux registres du CPU, par exemple.

1.2 La segmentation

1.2.1 Principes de la segmentation

Segmentation simple

La segmentation simple, comme celle utilisée dans le mode réel des processeurs x86, consiste à spécifier, pour tout accès mémoire, un registre de segment. Ce registre contient une valeur qui détermine à partir d'où l'adresse mémoire est spécifiée.

Par exemple, si on indique un segment commençant à l'adresse physique 0x1000, et qu'on demande l'adresse 0x42, on utilisera en réalité l'adresse 0x1042. L'adresse spécifiée en complément du segment est souvent appelée *offset*.

Le registre de segment à utiliser est parfois déterminé automatiquement, en effet, le x86 possède un registre spécifique à la pile et un autre spécifique au code (là où sont chargées les instructions).

Segmentation avancée

Dans la segmentation avancée, le registre de segment ne contient pas directement l'adresse physique de base du segment, mais indique une entrée dans une table de *descripteurs de segments*.

Cette table contient, pour chaque segment, l'adresse de début, l'adresse de fin du segment, ainsi que des informations de protection (par exemple, lecture seule, ou alors, utilisable uniquement par le noyau).

Lors d'un accès mémoire, les opérations suivantes sont réalisées, dans l'ordre :

1. le descripteur de segment est chargé depuis la table ;
2. les modes de protection sont vérifiées ;
3. l'adresse de début (nommée la *base*) du segment est ajoutée à l'offset ;
4. une comparaison est effectuée pour vérifier que cette somme ne dépasse pas le sommet du segment.

Sur certaines architectures, comme ia32 (i386 et supérieur), il existe plusieurs tables de descripteurs de segment. Sous ia32, il y a une GDT (Global Descriptor Table), qui contient les segments communs à tous les processus (par exemple, ceux du noyau), et une LDT (Local Descriptor Table) spécifique à chaque processus.

L'adresse de la (ou des) tables de descripteurs de segments doit être renseignée dans des registres spécifiques (en adresse absolue, bien sûr) et ne peut en général être changée que par le noyau.

Par la suite, c'est ce mode de segmentation qui sera considéré, sauf mention explicite du contraire.

1.2.2 Utilisations de la segmentation

La relocation

La première utilité de la segmentation est de régler le problème de la relocation. En effet, il suffit de positionner les registres de segments aux bonnes valeurs, et le programme peut être chargé à n'importe quel endroit de la mémoire.

Le swap

Il est possible d'utiliser la segmentation pour implémenter du swap. En effet, il est possible de transférer le contenu complet d'un segment sur le disque dur, et de marquer ce segment comme invalide.

Lorsqu'un accès sera fait sur ce segment, la MMU signalera l'erreur au système d'exploitation, qui devra recharger le segment.

La protection

La segmentation permet une première forme de protection mémoire (si seul le noyau peut spécifier les descripteurs de segments) : toute zone mémoire qui ne se trouve pas dans un segment configuré pour l'application ne pourra pas être utilisée.

Mais les descripteurs de segments contiennent en général des informations spécifiques sur les modes de protection. Un segment peut être déclaré en lecture seule, par exemple (c'est souvent le cas des segments où se trouvent le code de l'application). Mais surtout, la protection entre espace noyau et espace utilisateur peut être configurée au niveau des segments : un segment marqué "noyau" ne pourra être utilisé que par le noyau, tandis qu'un segment marqué "utilisateur" le sera par les deux.

La mémoire partagée

Il est enfin possible d'utiliser la segmentation pour implémenter de la mémoire partagée, en créant un segment spécifique qui sera disponible pour plusieurs processus.

Les autres utilités

Il est à noter aussi que la segmentation, surtout historiquement (mais ce cas peut encore se présenter dans l'embarqué, ou au contraire, sur les serveurs équipés de très grandes quantités de mémoire), permet aussi d'avoir plus de mémoire que la taille des registres ne le permettrait.

En effet, si on regarde le 8086, il possède des registres de 16-bits. Une adresse mémoire donc tenir sur 16-bits, ce qui est 65536 octets de mémoire maximum. L'utilisation de segments permet au 8086 d'adresser jusqu'à 1Mo de mémoire,

1.2.3 Avantages et limites de la segmentation

La segmentation a l'énorme avantage d'être très simple et efficace à implémenter au niveau matériel (il suffit d'effectuer une addition et une comparaison, choses que le matériel sait faire très rapidement).

Mais elle possède de nombreux désavantages, dont trois en particulier.

Faible granularité

Le premier problème de la segmentation est la faible granularité qu'elle fourni. Il est nécessaire pour le swap, par exemple, de déplacer un segment entier vers le disque, ce qui peut coûter très cher.

Fragmentation

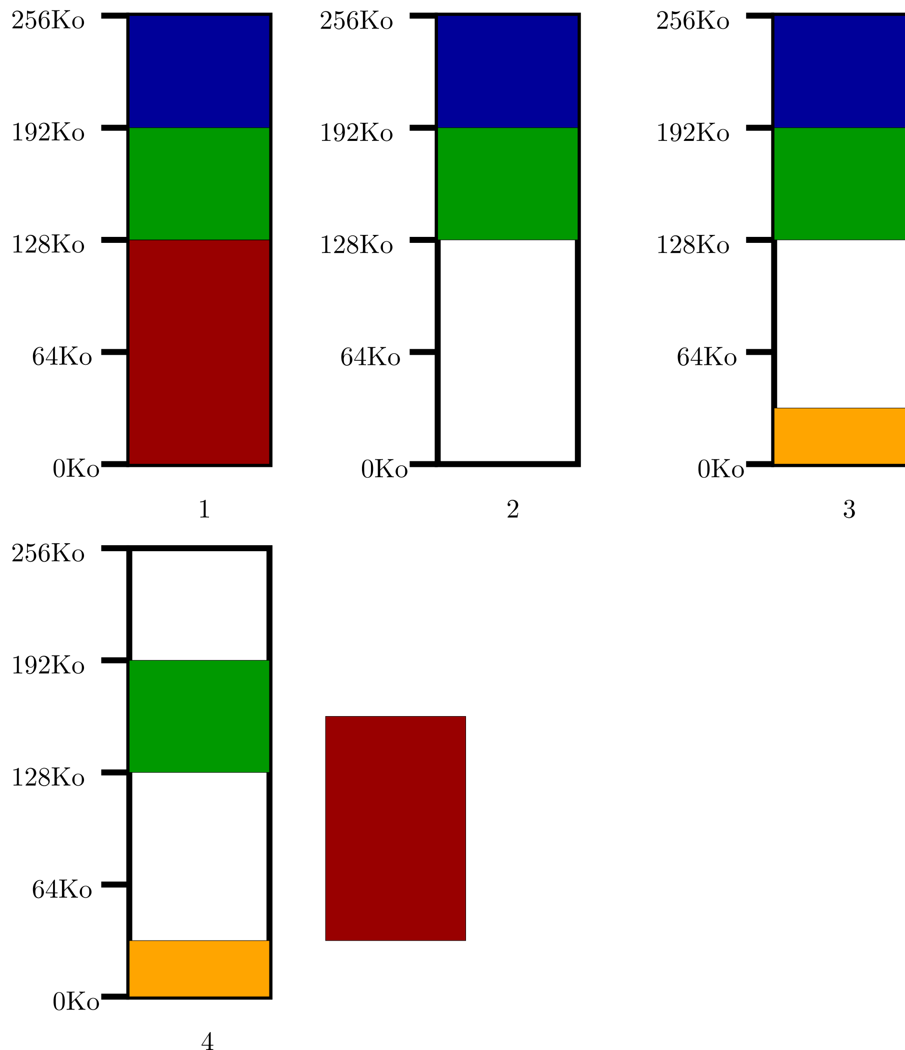

Fig. 1.2 – Exemple de fragmentation

Le deuxième problème est le risque de fragmentation, vu qu'un segment doit être continu en mémoire physique. S'il y a un espace disponible entre 32Ko et 128Ko (soit 96Ko) et un autre espace disponible entre 192Ko et 256Ko (soit 64Ko), il ne sera pas possible de loger un segment de 128Ko, alors que pourtant, il y a bien 160Ko de mémoire disponible.

Regardons la figure 1.2 : initialement (étape 1), trois segments sont présents en mémoire. Un quatrième segment est nécessaire, pour un nouveau programme. Le segment rouge est enlevé de la mémoire et mis sur le disque (étape 2). Le nouveau segment est créé (étape 3).

Puis le segment bleu est libéré (étape 4), puisqu'il n'est plus nécessaire (par exemple, le programme qui l'utilisait a terminé son exécution). Il y aurait alors assez de mémoire libre pour charger le segment rouge de nouveau, mais à cause de la fragmentation, il sera nécessaire de mettre sur le swap le segment orange ou le segment vert, afin d'avoir suffisamment de place continue.

Non transparence

Le dernier problème est que la segmentation n'est que partiellement transparente pour le programme l'utilisant. En particulier lorsqu'on utilise de la mémoire partagée, il est nécessaire pour le programme d'être conscient du fait qu'il utilise un segment, or, les langages de plus haut niveau que l'assembleur (comme le C) n'ont pas le concept de segments.

1.3 La pagination

1.3.1 Principes de la pagination

La pagination vise à régler les principaux problèmes de la segmentation.

Son principe est de diviser la mémoire en zones de taille fixe (de 2Ko à 16Ko en général, 4Ko étant le plus fréquent), appelées *page*. La mémoire physique elle est découpée en cadres de pages (*page frames* en anglais), de la même taille.

Pour chaque page, une table indique son cadre de page correspondant, ainsi que des informations sur son état.

En quelque sorte, une page est un segment de taille fixe et réduite. Mais la différence principale est que les pages sont transparentes pour le programme, qui lui-même voit sa mémoire de manière continue. Le matériel et le système d'exploitation gèrent cette pagination de manière transparente.

Par exemple, avec des pages de 4096 octets, l'adresse mémoire 4196 correspond à 100 octets à l'intérieur de la page numéro 1 (en informatique, la numérotation commence toujours à 0, ou presque). Si la table de correspondance indique qu'elle se trouve dans le cadre numéro 2, l'adresse physique sera donc 8192 (début du cadre numéro 2) + 100, soit 8292.

1.3.2 Utilisations de la pagination

En réalité, chaque page ne correspond pas forcément à un cadre de page. Chaque page peut être soit présente dans la mémoire physique (donc, assignée à un cadre de page), soit absente de la mémoire physique (marquée comme invalide). Une page peut être absente de la mémoire physique soit parce qu'elle est sur un support annexe (comme le swap), soit parce qu'elle est réellement invalide (zone non allouée). La figure 1.3 montre un exemple de ces différents cas.

La relocation

La relocation est gérée tout simplement en fixant l'adresse dans la mémoire virtuelle du code. Il peut ensuite se trouver n'importe où en mémoire physique.

Le swap

Le swap peut être géré avec une granularité d'une page. Le système d'exploitation peut sauvegarder une page (donc, 4Ko par exemple) sur le disque, et marquer cette page comme invalide. En cas d'accès à des données de cette page, la MMU va lever une erreur, et donner le contrôle au système d'exploitation.

Celui-ci peut alors relire les données du disque, les mettre dans un cadre de page libre, et indiquer le nouveau cadre de page dans la table.

Il est ainsi possible d'effectuer du swap de manière fine, en ne transférant qu'une partie des données d'un programme (celles qu'il utilise peu souvent, si possible).

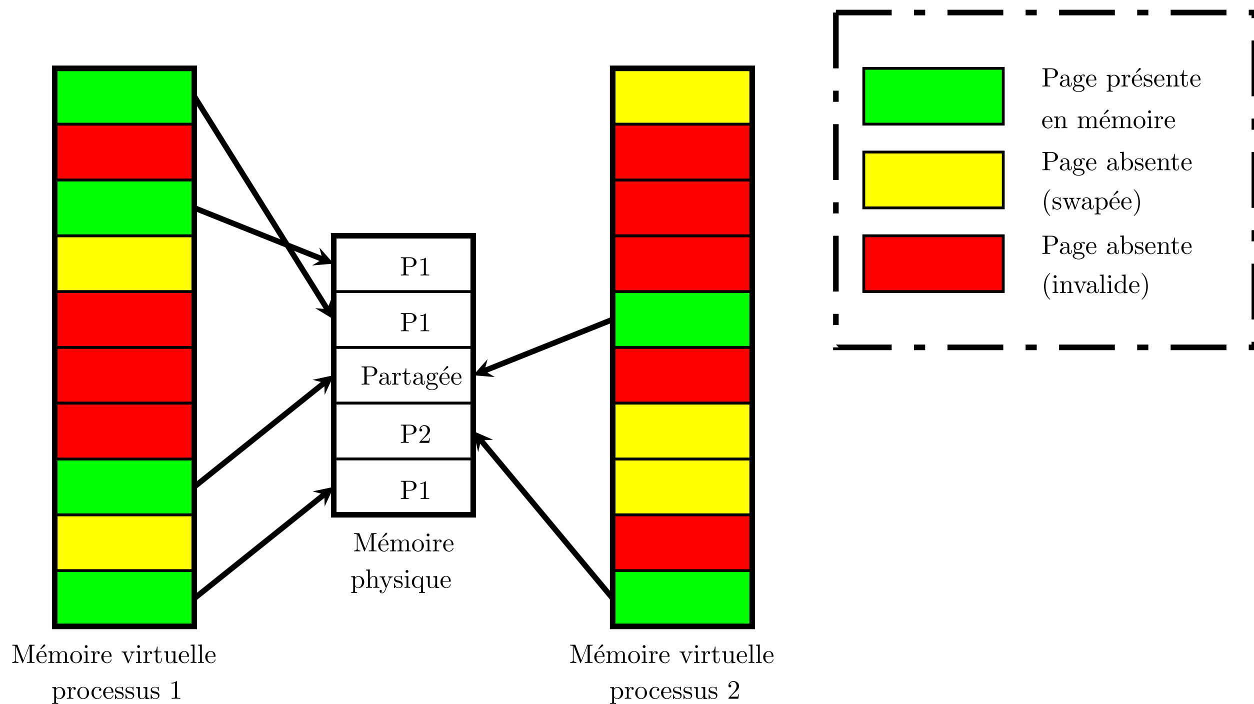

Fig. 1.3 – Exemple de pagination

La protection

La protection mémoire est, comme pour la segmentation, implémentée de deux façons : tout d'abord, tout cadre de page qui ne peut être atteint depuis aucune page ne peut pas être vue par l'application.

Ensuite, chaque page possède des protections, comme pour la segmentation : lecture seule ou non, accessible pour le noyau uniquement ou non.

Chaque processus possède donc, bien sûr, sa propre table de pages. Cette table de page locale d'un processus est nommée son *espace d'adressage*.

La mémoire partagée

La mémoire partagée peut être implémentée facilement avec la pagination, tout simplement en indiquant les mêmes cadres de pages pour des pages de deux espaces d'adressage différents. Ces pages peuvent elles se trouver à des adresses virtuelles différentes.

Les fichiers et « mmap »

Il est possible d'utiliser la pagination pour implémenter les fichiers projetés en mémoire (`mmap`), comme suit :

1. Une zone de la taille du fichier est allouée dans l'espace d'adressage virtuel du processus.
2. Chaque page de cette zone est marquée comme invalide.
3. Lorsqu'un accès en lecture est fait sur une page absente, elle est lue depuis le fichier et mise dans un cadre de page, en lecture seule.
4. Lorsqu'un accès en lecture est fait sur une page présente, le résultat est instantané, sans transition vers l'espace noyau (contrairement à un `read`).
5. Lorsqu'un accès est fait en écriture sur une page en lecture seule, celle-ci est modifiée pour autoriser l'écriture.
6. Régulièrement, l'ensemble des pages qui sont autorisées en écriture sont écrites sur le disque, et les pages sont marquées de nouveau comme étant en lecture seule.

L'astuce avec la lecture seule permet de ne transférer sur le disque que les pages qui ont été modifiées.

Le « copy on write »

Une autre utilisation, fondamentale sur les systèmes Unix avec l'appel système `fork` est ce que l'on appelle le « copy on write », c'est à dire « copie lors de l'écriture ». Ce mécanisme permet d'éviter d'avoir à copier l'ensemble des données d'un processus lors d'un `fork`.

Il fonctionne comme suit :

1. Lors du `fork`, la table de pages est recopiée, mais pointe les mêmes cadres de pages.
2. L'ensemble des pages sont marquées en lecture seule.
3. Lorsque l'un des deux processus essaie d'écrire sur une page, celle-ci est dupliquée physiquement, et chaque processus se voit attribué un cadre de page différent. Les deux sont maintenant autorisés en lecture et en écriture.
4. Les accès en écriture suivant sur la même page se font normalement.

1.3.3 Implémentation de la pagination

Nous allons maintenant voir comment la pagination peut être implémentée au niveau du matériel. Pour des soucis de concision et de clarté, nous considérerons toujours par la suite des pages de 4Ko et un espace d'adressage, virtuel comme physique, de 32 bits (donc, 4Go).

Implémentation simple au niveau du matériel

Dans l'implémentation simple, la table de pages contient une entrée de 32-bits par page. La page numéro 21 sera donc à l'octet numéro 84 de la table de pages. La table de pages, elle, est présente dans la mémoire physique de manière continue. Son adresse physique est chargée dans un registre spécial.

Sur 32-bits, on peut faire tenir l'adresse physique du début de la page. Mais où stocker les informations de protection ? En fait, l'adresse physique complète n'est pas nécessaire. En effet, les cadres de page étant tous de 4Ko, ils commenceront tous sur un multiple de 4096. Les 12 derniers bits ne sont donc pas nécessaires ; il est beaucoup plus efficace de stocker le numéro du cadre, et de garder les bits restant pour des informations complémentaires.

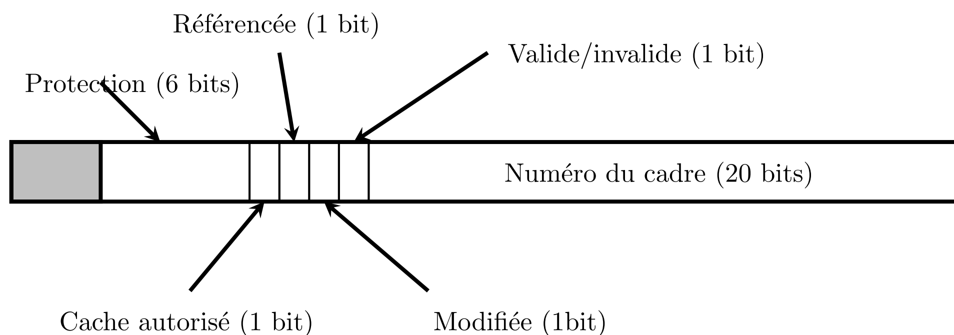

Fig. 1.4 – Exemple d'entrée de table de pages

Sur la figure 1.4 on peut voir un exemple d'entrée de table de pages. Les fonctionnalités exactes disponibles dépendent du matériel, mais voici la signification des champs de cet exemple :

Numéro du cadre Information principale, le numéro du cadre de page associé à la page.

Valide/invalide Bit à 0 sur la page est invalide (swapée ou réellement invalide), à 1 si elle est valide.

Modifiée Bit mis à 1 par le matériel à chaque accès en écriture, et qui peut être remis à 0 par le système d'exploitation (permet d'éviter l'astuce « lecture seule » pour l'implémentation de `mmap`, par exemple).

Référencée Bit mis à 1 par le matériel à chaque accès en lecture ou en écriture, et qui peut être remis à 0 par le système d'exploitation (permet d'implémenter des algorithmes plus efficace, voir plus loin).

Cache autorisé Ce bit peut permettre d'interdire l'utilisation des caches L1 et L2, par exemple dans le cas d'une zone mémoire dédiée à communiquer avec un périphérique externe.

Protection La protection peut être simple : un bit qui autorise ou non les accès en écriture, et un qui autorise ou non les accès par le code utilisateur (= non noyau). Dans ce exemple, 6 bits ont été réservés, par exemple pour les trois permissions classiques `rwx`, pour l'espace noyau et pour l'espace utilisateur.

Dans la suite, les bits “référencée” et “modifiée” seront nommés R et M, respectivement.

Le matériel (la MMU) fonctionne ainsi :

1. L'adresse virtuelle est découpée en deux, une partie contenant le numéro de la page, et l'autre partie le décalage à l'intérieur.
2. L'entrée de la table de page correspondante est chargée.
3. Les vérifications sont effectuées sur les bits de validité et de protection.
4. Éventuellement, les bits R et M sont mis à jour.
5. Le numéro du cadre de page est multiplié par 4096 (décalage de bits) et l'offset y est ajouté.
6. Cette dernière valeur est envoyée sur le bus mémoire.

La pagination à deux niveaux

Ce mode de fonctionnement comporte un gros problème : la taille des tables de pages. Avec 32-bits (donc 4 octets) par page, et 2^{20} pages, chaque table de pages fait 4Mo. Le système doit donc allouer 4Mo, en mémoire physique, par processus présent. Pour 100 processus (chiffre très raisonnable, pour information la machine où ce texte est rédigée a 175 processus actuellement), cela fait 400Mo de mémoire consommée. Ce coût est jugé exorbitant.

Les MMU implémentent donc, dans la pratique, des mécanismes un peu plus compliqués comme la pagination à deux niveaux. En effet, la plupart des processus n'utilisent qu'une petite partie de leur espace d'adressage virtuel. Le principe, au lieu d'utiliser une table globale, est de regrouper les pages dans un catalogue de page. Le catalogue de pages contient des entrées sur des tables de pages (mais certaines entrées du catalogue peuvent bien sûr être invalides).

Comme on le voit sur la figure 1.5, l'adresse virtuelle de 32-bits est découpée en trois parties : le numéro de la table de pages dans le catalogue de pages sur 10-bits, puis le numéro de la page dans la table de pages, et enfin le décalage dans la page. L'avantage de ce découpage (10/10/12) est que le catalogue, comme chacune des tables de pages, font chacun 4Ko, c'est à dire exactement une page.

La MMU effectue l'algorithme suivant :

1. L'adresse de base du catalogue de page (en nombre de cadres de pages, donc en multiples de 4096) est dans un registre spécial, `CR3` sur ia32.
2. L'adresse virtuelle est découpée en trois parties : `PT1`, `PT2` et `Offset`.
3. L'entrée `PT1` du catalogue de page est chargée (adresse mémoire physique $CR3 \times 4096 + PT1$), dans un registre interne que nous appellerons `PTB`.
4. Cette entrée est vérifiée, si elle est invalide une erreur est signalée.
5. L'entrée `PT2` de la table de pages indiquée par `PTB` (adresse mémoire physique $PTB \times 4096 + PT2$) est chargée dans un registre interne, que nous nommerons `Base`.
6. Cette entrée est vérifiée, et éventuellement mise à jour, comme dans la pagination simple.
7. L'adresse mémoire physique finale $Base \times 4096 + Offset$ est envoyée sur le bus mémoire.

L'importance de la TLB

La pagination à deux niveaux est parfaitement utilisable en pratique. La surcharge mémoire reste faible (un catalogue de 4Ko, plus ensuite une table de pages de 4Ko pour chaque plage de 4Mo de mémoire utilisée). Cependant, si on regarde bien l'algorithme précédant, pour chaque accès mémoire on a deux accès mémoires (dans le catalogue et dans la table), plus une série de calculs (même si de simples décalages de bits et additions sans retenue) et de vérifications. Effectuer toutes ces opérations à chaque fois coûterait bien trop cher.

Un cache spécial est donc présent au niveau du matériel : la TLB, pour *Translation Lookaside Buffer*. Ce cache couvre les opérations 1 à 6 de l'algorithme précédant, et permet de retrouver instantanément, à partir des 20 premiers bits d'une adresse virtuelle, les 20 premiers bits de l'adresse physique à laquelle ils correspondent. La taille de la TLB varie entre 32 et 1024 entrées, en général.

Cette TLB doit être vidée à chaque changement de contexte (passage d'un processus à un autre), et c'est ce qui explique en grande partie le coût élevé de ces changement de contexte.

Il est aussi à noter que si la plupart des architectures remplissent la TLB au niveau du matériel, sur certaines la MMU ne gère que les entrées présentes dans la TLB. En cas d'entrée non présente dans la TLB, le système d'exploitation est invoqué, et c'est lui qui doit peupler la TLB, avec l'algorithme et la méthode de pagination de son choix.

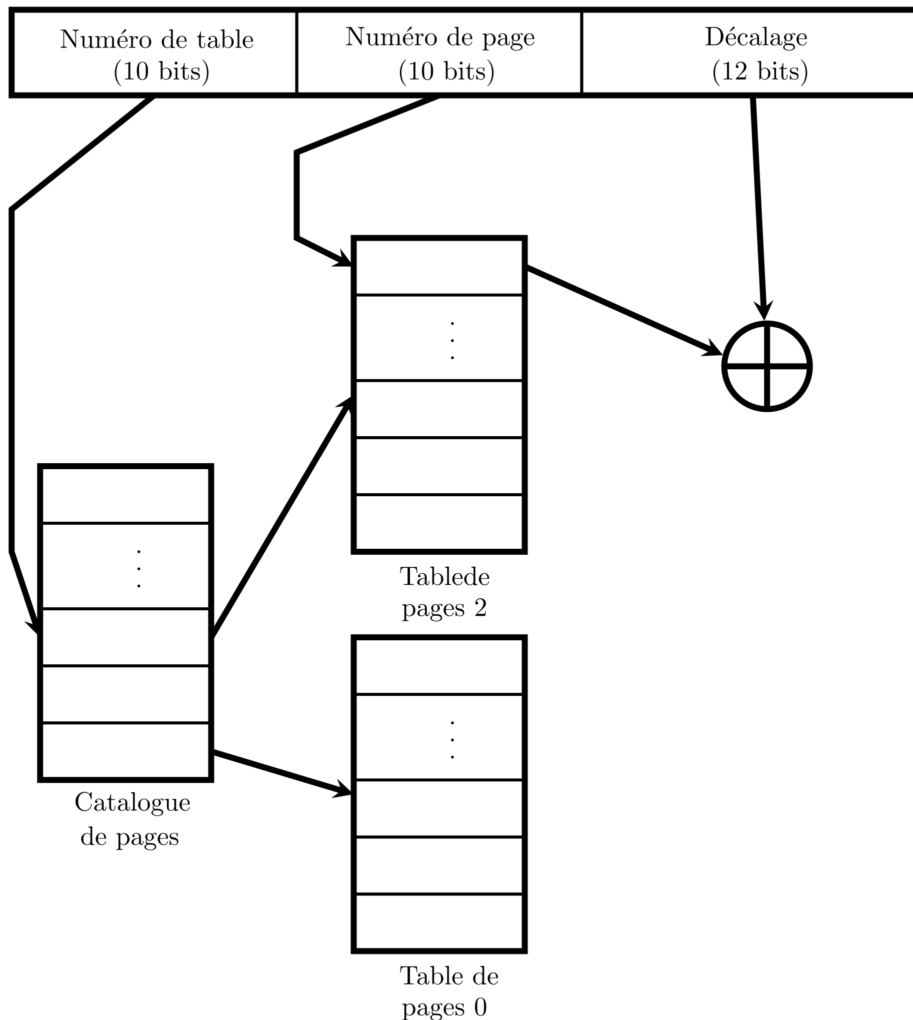

Fig. 1.5 – Pagination à deux niveaux

Implémentation au niveau du système

Au niveau du système, la pagination peut être implémentée de multiples façons. Il est nécessaire de maintenir, pour chaque processus, l'état de ses pages de mémoire virtuelle, pour gérer les différents systèmes comme le swap ou le « copy on write ». En effet, si le matériel permet de connaître le cadre de page associé à une page présente en mémoire, il ne dit rien pour les pages absentes de la mémoire (endroit du swap où elles sont, entrée valide ou non, ...).

Un point important à noter est l'utilisation du bit "modifiée", s'il est présent dans le matériel, pour savoir si une page doit être écrite sur le disque ou non. Dans le cas du swap par exemple, lorsqu'une page est restaurée depuis le swap, ce bit est mis à 0, et la page est laissée sur le swap. Plus tard, si cette même page doit être de nouveau mise sur le swap, mais que le bit "modifiée" n'est pas à 1, il n'est pas nécessaire de la réécrire, et on gagne ainsi une opération d'écriture sur le disque.

1.3.4 Avantages et limites de la pagination

La pagination permet de contourner toutes les limites de la segmentation. Le principal inconvénient est qu'elle est plus complexe, et implique donc une MMU plus compliquée (et donc, plus chère) ainsi que des coûts en performances plus élevés (en particulier lors d'un changement de processus, mais aussi en consommation mémoire pour stocker les tables de pages).

Chapitre 2

Algorithmes de pagination

2.1 Les différents enjeux

Maintenant que nous avons vu le mécanisme de la pagination, il reste une grande question à résoudre : que faire quand la mémoire physique disponible n'est pas suffisante. On peut utiliser du swap, on peut libérer une page du cache disque ou d'un fichier projeté en mémoire. Mais il faut décider quelle page libérer.

Un bon algorithme de pagination doit essayer de limiter au maximum la fréquence à laquelle des pages sont ajoutées ou enlevées de la mémoire, car ces opérations sont coûteuses.

Une décision importante à prendre est de considérer si l'algorithme doit être global ou local. Un algorithme global considère l'ensemble des pages du système, et peut prendre un cadre de page à un processus pour le donner à un autre. Un algorithme local ne regarde que les cadres de page du processus qui nécessite plus de mémoire. Un algorithme global sera plus efficace (puisque dans son cycle de vie un même programme varie grandement sur ses besoins de mémoire), mais moins juste. En général, les systèmes utilisent des algorithmes globaux.

Le concept important à comprendre, pour la pagination comme pour tous les autres cache (L1, L2, TLB, ...) est le concept de « *working set* ». Dans le courant de son exécution, un processus travaille en général pendant un temps assez long sur la même série de pages mémoires. La taille et la durée sont bien sûr variable, mais le concept reste presque toujours présent.

Ce concept est associé à un autre, celui de « *trahsing* ». Ce terme est employé lorsque les pages du *working set* sont régulièrement mises sur le swap et rechargées (soit à cause d'un mauvais algorithme, soit parce qu'il n'y a pas assez de mémoire physique). Il désigne une situation où les performances sont très fortement dégradées.

De manière générale, sur une machine donnée, c'est d'ailleurs plus la fréquence à laquelle les pages sont lues et écrites sur le swap (ou autre *backing store*), plutôt que la quantité de swap utilisée qui dénote un problème. Cette information est accessible sous la plupart des Unix avec la commande `vmstat`.

2.2 L'algorithme NRU (*Not recently used*)

L'algorithme NRU consiste à éliminer une page qui n'a pas été utilisée récemment. Pour cela, il repose sur le bit R mis à jour par le matériel. Régulièrement, le bit est mis à 0 sur toutes les pages actives. Lorsqu'une page doit être libérée, une page ayant encore son bit à 0 est choisie (et si aucune n'est disponible, une page ayant son bit à 1 est alors choisie).

Une amélioration de cet algorithme consiste à définir quatre classes de pages comme suit :

1. Non référencées, non modifiées.
2. Non référencées, modifiées.
3. Référencées, non modifiées.
4. Référencées, modifiées.

Les pages de la classe 1 sont choisies en priorité (elles ne nécessitent pas d'écriture sur le disque, et n'ont pas été utilisées), puis celles de la classe 2, et ainsi de suite.

À noter que la classe 2 n'est pas inexistante, si une page est modifiée, puis que sont bit R a vidé par l'opération régulière sans qu'elle soit écrite sur le disque.

Cet algorithme est modérément efficace, mais nécessite de remettre à 0 les bits de toutes les pages régulièrement, ce qui peut coûter cher.

2.3 L'algorithme d'horloge

2.3.1 L'algorithme FIFO (*First-in first-out*)

Un autre algorithme classique et simple à implémenter est l'algorithme FIFO : la page qui est évincée est celle qui avait mise en mémoire il y a le plus longtemps. Une simple liste chaînée suffit pour l'implémenter.

Mais cet algorithme est peu efficace : en effet, certaines pages sont utilisées très souvent (comme certaines pages qui contiennent le code de la libc, par exemple) même si elles sont présentes depuis longtemps.

2.3.2 L'algorithme deuxième chance

Une amélioration de l'algorithme FIFO est l'algorithme de deuxième chance. Dans ce cas, on garde la liste chaînée de pages de l'algorithme FIFO, mais on regarde aussi le bit R de la page.

Si la page la plus vieille (qui devrait être utilisée) a son bit R à 1, alors on la déplace à la fin de la liste (on considère que c'est la page la plus récente), mais on remet son bit R à 0. On s'arrête à la première page qu'on trouve et qui a son bit R à 0, qui est la page la plus ancienne qui n'a pas été utilisée récemment (ou éventuellement, la page la plus ancienne tout court, après avoir parcourue toutes les pages, si jamais toutes ont été utilisées).

Cet algorithme est raisonnablement efficace, et ne crée pas de sur coût trop élevé (au pire, la liste sera parcourue complètement une fois lors de la libération d'une page, mais les cas où toutes les pages ont été utilisées sont rares).

2.3.3 L'algorithme d'horloge

L'algorithme d'horloge est une amélioration de l'implémentation de l'algorithme précédant, pour éviter de devoir déplacer sans cesse des maillons dans la chaîne (opération peu coûteuse, mais coûteuse tout de même).

La solution consiste à utiliser une liste chaînée circulaire, avec une aiguille qui pointe au départ sur la première page créée (donc, la plus ancienne). Lorsqu'il faut libérer une page, la page pointée par l'aiguille est regardée. Si son bit R est à 0, elle est libérée et l'aiguille est avancée. Si son bit R est à 1, il est mis à 0, l'aiguille est avancée et l'algorithme recommence, jusqu'à trouver une page dont le bit R est à 0.

Cet algorithme, sous une forme ou sous une autre, est l'un de ceux qui sont utilisés en pratique dans plusieurs systèmes d'exploitation réels.

2.4 L'algorithme LRU « Least Recently Used »

2.4.1 Le LRU théorique

Le LRU consiste à évincer de la mémoire non pas la page la plus vieille (même avec une deuxième chance), mais la page qui a été utilisée il y a le plus longtemps.

Hélas, le matériel ne donne en général pas les informations suffisantes pour implémenter cet algorithme, pour des raisons économiques (il faudrait ajouter, par exemple, un compteur de temps sur chaque entrée de la page de tables, et le mettre à jour à chaque accès, chose coûteuse en mémoire comme en temps).

2.4.2 L'algorithme NFU

Une première approximation, nommée NFU (« Not Frequently Used ») consiste à utiliser un compteur, maintenu au niveau logiciel, associé à chaque page, initialement mis à zéro.

De manière régulière, le système va scanner la liste de toutes les pages présentes en mémoire, ajouter la valeur du bit R (0 ou 1) au compteur et le remettre à 0.

Le compteur contient donc le nombre de cycles où la page a été référencée. Lorsqu'il faut évincer une page, c'est celle où le compteur est le plus faible qui sera choisie. Cet algorithme a l'inconvénient de ne jamais rien oublier, une page qui a été utilisée souvent pendant un moment mais qui ne l'est plus restera longtemps en mémoire.

2.4.3 Une approximation de LRU : *Aging*

Une solution à ce dernier problème consiste à effectuer les opérations suivantes, au lieu d'une addition : le compteur est décalé vers la droite d'un bit (c'est à dire, divisé par deux), puis la valeur du bit R est ajouté dans son bit le plus à gauche (donc, celui qui a la plus grande valeur).

Ainsi, le temps est pris en compte, et l'historique ne sert qu'à départager les pages ayant été utilisées récemment.

Cet algorithme est souvent plus efficace que l'algorithme d'horloge en terme de bon choix des pages à garder en mémoire, mais il est plus coûteux puisqu'il nécessite de parcourir régulièrement toutes les pages.

Il est cependant le plus utilisé, sous une forme ou sous une autre.

2.5 Introduction à la pagination sous Mach

2.5.1 Le fonctionnement des pagers externes

Le fonctionnement général

Sous Mach, l'algorithme de pagination est des plus classiques : une approximation de LRU implémentée dans le micro-noyau. Par contre, le concept intéressant et novateur est l'existence de pagers externes.

Le principe est de permettre à des programmes utilisateurs de gérer la manière dont les pages sont stockées (sur le swap, sur le réseau, . . .). Chaque application (plus exactement, chaque zone mémoire d'une application) est gérée par un programme nommé « *pager* ».

Lorsqu'une page doit être évincée, le fonctionnement est le suivant (voir figure 2.1) :

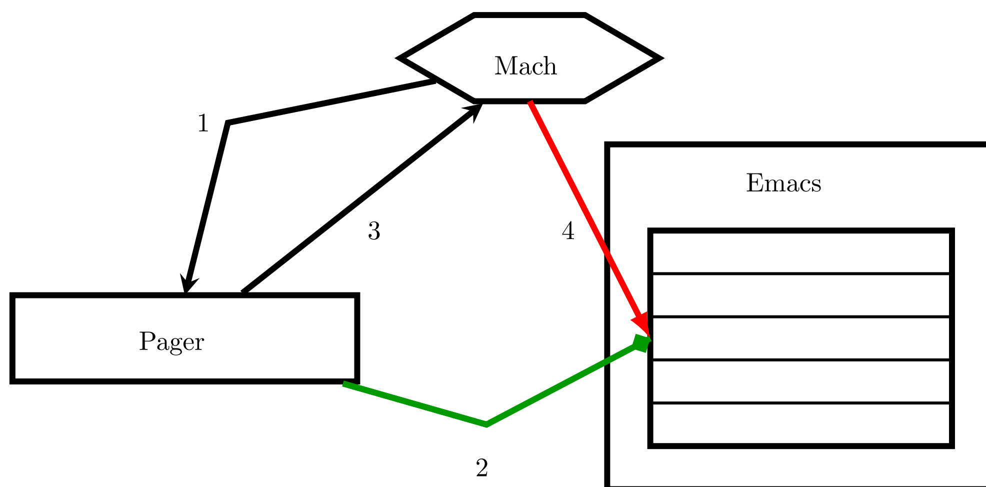

Fig. 2.1 – Éviction d'une page sous Mach

1. Le micro-noyau Mach envoie un message au pager, lui indiquant quelle page doit être éliminée.
2. Le pager lit le contenu de la page, et la stocke quelque part.
3. Le pager répond au micro-noyau pour l'informer que la page est désormais sauvegardée.
4. Le mico-noyau modifie alors l'entrée de la table de pages, et peut réutiliser le cadre de page.

Lorsqu'une page doit être restaurée en mémoire, le fonctionnement est le suivant (voir figure 2.2) :

1. Le processus utilisateur (GNU Emacs, par exemple) essaie d'accéder à une page qui n'est plus présente en mémoire.
2. La MMU lève une erreur (exception) et invoque le micro-noyau.
3. Le micro-noyau affecte un cadre de page libre (éventuellement en en libérant un autre) à la page.
4. Le micro-noyau envoie un message au pager, pour lui demander de restaurer la page.
5. Le pager restaure les données dans la page concernée.
6. Le pager signale que la page est prête au micro-noyau.
7. Le micro-noyau redémarre le processus qui avait été arrêté.

Le pager par défaut

Ce mécanisme se heurte à un problème de récursion : le pager étant une application comme une autre, il est lui-même pagé par un pager... Pour contourner ce problème, un pager spécial est lancé au démarrage du système, le pager par défaut, qui lui ne peut pas être pagé. Il gère en général le paging vers un swap fixe.

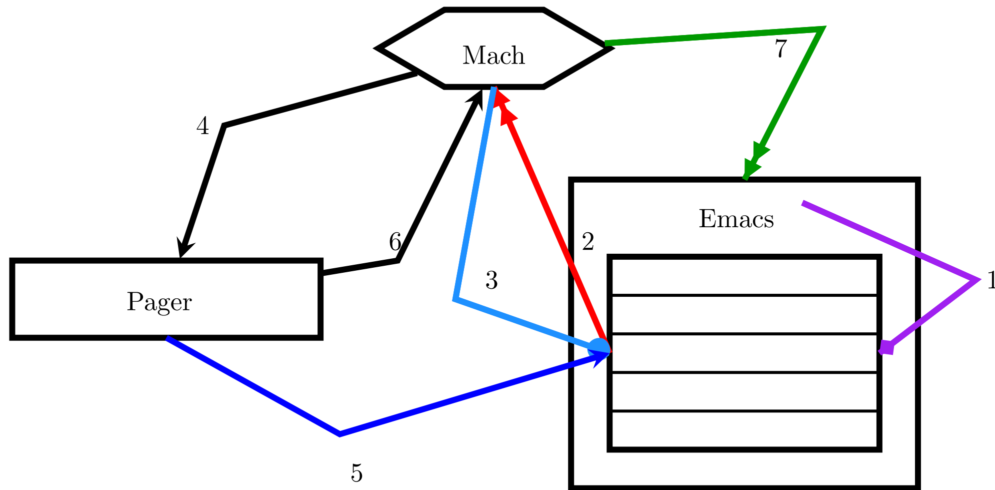

Fig. 2.2 – Restauration d'une page sous Mach

De plus, il est possible qu'un pager externe se plante et ne réponde plus aux requêtes, ou alors, qu'il mette très longtemps à y répondre. Pour éviter un dénis de service ou une paralysie du système, si un pager spécifique mais trop longtemps à répondre, alors Mach demande au pager par défaut (qui lui est lancé au démarrage et considéré comme fiable) de s'occuper de la page.

2.5.2 Quelques exemples d'application

Voici quelques exemples d'application des pagers externes :

- Pour des machines de faibles capacités, utiliser un serveur réseau externe pour stocker le swap, et non le disque local.
- Compresser les pages en mémoire, au lieu de les mettre sur le disque dur, ce qui peut être très efficace dans le cas de données faciles à compresser, comme des images ou du son en haute résolution.
- Ne pas sauvegarder les données, quand elles peuvent être régénérées facilement, par exemple, les données décompressées d'une image dans un navigateur peuvent être facilement recréées à partir de l'image compressée, elle déjà présente dans le cache du navigateur.

Informations légales

Ce document est Copyright(c) Pilot Systems 2007. Il est disponible selon les termes de la GNU General Public License, version 3 ou supérieure.

La version PDF et le code source LaTeX sont disponibles sur `http://insia.pilotsystems.net`.