liste simplement chaînée comme à la figure 3.6(c). Avec cette structure, il est plus facile de trouver l'entrée précédente et de voir si une fusion est possible.

Quand les processus et les trous sont indiqués dans une liste triée par adresse, plusieurs algorithmes peuvent servir à allouer de la mémoire à un processus nouvellement créé (ou un processus existant chargé depuis le disque). L'algorithme le plus simple est l'**algorithme de la première zone libre** (*first fit*). Le gestionnaire de mémoire parcourt la liste des segments jusqu'à trouver un trou qui soit assez grand. Le trou est ensuite divisé en deux parties, l'une destinée au processus et l'autre à la mémoire non utilisée, sauf si le processus et le trou ont la même taille, ce qui est statistiquement peu probable. L'algorithme de la première zone libre est rapide parce qu'il limite ses recherches autant que possible.

Une petite variante de la première zone libre est l'**algorithme de la zone libre suivante** (*next fit*). Il fonctionne de la même façon que le précédent, et mémorise en outre la position de l'espace libre trouvé. Quand il est de nouveau sollicité pour trouver un trou, il débute la recherche dans la liste à partir de l'endroit où il s'est arrêté la fois précédente, au lieu de recommencer au début comme le fait l'algorithme de la première zone libre. Des simulations ont montré que les performances de l'algorithme de la zone libre suivante sont légèrement meilleures que celles de l'algorithme de la première zone libre.

Un autre algorithme bien connu est l'**algorithme du meilleur ajustement** (*best fit*). Il fait une recherche dans toute la liste et prend le plus petit trou adéquat. Plutôt que de casser un gros trou qui peut être nécessaire ultérieurement, l'algorithme du meilleur ajustement essaye de trouver un trou qui correspond à la taille demandée.

Considérons de nouveau la figure 3.6 pour illustrer les algorithmes de la première zone libre et du meilleur ajustement. Si un bloc de taille 2 est demandé, l'algorithme de la première zone libre allouera le trou en 5, mais l'algorithme du meilleur ajustement allouera le trou en 18.

L'algorithme du meilleur ajustement est plus lent que l'algorithme de la première zone libre parce qu'il doit fouiller la liste entière à chaque fois qu'il est sollicité. Plutôt curieusement, il perd aussi plus de place mémoire que les algorithmes de la première zone libre et de la zone libre suivante : en effet, il tend à remplir la mémoire avec de minuscules trous inutiles. En moyenne, l'algorithme de la première zone libre génère des trous plus larges.

Pour régler le problème des cassures et faire concorder de manière quasiment exacte un processus et un trou minuscule, on pourrait penser à l'**algorithme du plus grand résidu** (*worst fit*), qui consisterait à prendre toujours le plus grand trou disponible : ainsi, le trou restant serait assez grand pour être réutilisé. Cependant, des simulations ont montré que cet algorithme n'est pas non plus une solution.

La rapidité des quatre algorithmes que nous avons examinés peut être améliorée si l'on établit des listes séparées pour les processus et les trous. Dans ce cas, les algorithmes consacrent toute leur énergie à chercher des trous, non des processus. Mais il y a une contrepartie inévitable à cette accélération dans l'allocation : une complexité plus

grande, ainsi qu'un ralentissement quand la mémoire est libérée, puisqu'un segment qui est libéré doit être effacé de la liste des processus et inséré dans la liste des trous.

Si l'on procède avec des listes distinctes pour les processus et les trous, la liste des trous doit être triée par taille, afin d'augmenter la rapidité de l'algorithme du meilleur ajustement. Ainsi, l'algorithme parcourt la liste des trous du plus petit au plus grand : dès qu'il en a trouvé un qui convient, il sait que ce trou est le plus petit et par conséquent le meilleur. Aucune autre recherche n'est nécessaire, à la différence du schéma avec une liste unique. Dans le cas d'une liste des trous triée par taille, l'algorithme de la première zone libre et celui du meilleur ajustement sont aussi rapides l'un que l'autre, et l'algorithme de la prochaine zone libre est inutile.

Quand les trous sont conservés dans des listes séparées des processus, une petite optimisation est possible. Au lieu d'avoir un ensemble séparé de structures de données pour gérer la liste des trous [voir figure 3.6(c)], on peut utiliser les trous eux-mêmes. Le premier mot de chaque trou peut être la taille du trou, et le second mot un pointeur sur l'entrée suivante. Les nœuds de la liste de la figure 3.6(c), qui nécessitent trois mots et un bit (P/T), ne sont plus nécessaires.

Il existe un autre algorithme encore, l'**algorithme du placement rapide** (*quick fit*), lequel gère des listes séparées pour certaines des tailles les plus communément demandées. On peut avoir le cas d'une table avec n entrées, dans laquelle la première entrée est un pointeur sur une liste de trous de 4 Ko, une deuxième entrée est un pointeur sur une liste de trous de 8 Ko, une troisième entrée un pointeur sur une liste de trous de 12 Ko, et ainsi de suite. Des trous de 21 Ko, par exemple, seront placés soit dans la liste des 20 Ko, soit dans une liste spéciale de trous de taille impaire. L'algorithme du placement rapide permet de trouver un trou d'une taille donnée de façon extrêmement rapide, mais il présente le même inconvénient que toute méthode qui trie les trous par taille : quand un processus se termine ou est transféré sur disque, le fait de rechercher ses voisins pour voir si une fusion est possible est coûteux. S'il n'y a pas de fusion, la mémoire se fragmente rapidement en un grand nombre de petits trous dans lesquels aucun processus ne peut entrer.

3.3 La mémoire virtuelle

Les registres de base et de limite permettent de créer l'abstraction de l'espace d'adressage, mais un autre problème doit être géré : celui des « obésiciels » (*bloatware*). Si la taille des mémoires augmente, celle des logiciels augmente encore plus vite ! Dans les années 1980, on faisait tourner en temps partagé des dizaines d'utilisateurs (plus ou moins satisfaits) sur des VAX dotés de 4 Mo de mémoire. Aujourd'hui, avec Vista, il faut à un mono-utilisateur 512 Mo minimum et plutôt 1 Go pour faire quelque chose de sérieux. Et avec le développement du multimédia cela ne va pas s'arranger.

En conséquence, on est à peu près sûr d'être à court de mémoire, même si on ne fait pas tourner simultanément de nombreux programmes d'application. Le va-et-vient

n'est pas une bonne option en raison des volumes à transférer : il faut au moins 10 s pour mettre un programme de 1 Go sur un disque SATA au débit crête de 100 Mo/s.

Mais ce problème de programme plus gros que la mémoire est connu depuis bien longtemps. Dans les années 1960, la solution la plus courante consistait à diviser le programme en parties, appelées **segments de recouvrement** (*overlays*). Le segment 0 était exécuté en premier. Quand cela était réalisé, il pouvait appeler un autre segment. Certains des systèmes à base de recouvrement étaient hautement complexes, autorisant de multiples recouvrements simultanés. Les segments de recouvrement étaient conservés sur le disque et chargés en mémoire par le système d'exploitation, dynamiquement à la demande.

Si le va-et-vient des segments était géré par le système, c'est le programmeur qui devait diviser le programme. Or le découpage de grands programmes en petites parties modulaires prenait beaucoup de temps et était en outre ennuyeux. Heureusement, il n'a pas fallu longtemps pour que quelqu'un pense au moyen de faire faire tout le travail par l'ordinateur.

La méthode qui a été imaginée est connue aujourd'hui sous le nom de **mémoire virtuelle**. La mémoire virtuelle repose sur le principe suivant : chaque programme a son propre espace d'adressage découpé en petites entités appelées **pages**, entités formées d'une suite d'adresses contiguës. Ces pages sont mappées sur la mémoire physique, mais il n'est pas obligatoire d'avoir toutes les pages en mémoire physique pour exécuter le programme. Lorsque le programme référence une partie de son espace d'adressage qui est en mémoire physique, le matériel fait la correspondance au vol. Dans le cas contrainte, si la partie de l'espace référencée *n'est pas* en mémoire, le SE prend la main pour aller chercher sur disque ce qui manque, le ranger en mémoire et reprendre ensuite là où l'on s'était arrêté.

En un certain sens, la mémoire virtuelle est une généralisation de l'idée de registres de base et de limite. Le 8088 a des registres de base séparés pour programme et données. Avec la mémoire virtuelle, au lieu d'avoir une réallocation séparée pour le programme et les données, c'est l'espace d'adressage tout entier que l'on met en correspondance avec la mémoire physique au moyen de petites unités.

La mémoire virtuelle fonctionne très bien dans un système multiprogrammé dans lequel bits de données et parties de plusieurs programmes se retrouvent simultanément en mémoire. Quand un programme attend le chargement d'une partie de lui-même, il est en attente d'E/S et ne peut s'exécuter ; l'UC peut par conséquent être donnée à un autre processus.

3.3.1 La pagination

La plupart des systèmes à mémoire virtuelle se servent d'une technique appelée **pagination**, que nous allons à présent étudier. Sur n'importe quel ordinateur, il existe un ensemble d'adresses mémoire que les programmes peuvent générer. Quand un programme utilise une instruction comme

```
MOV REG,1000
```


cela copie le contenu de l'adresse mémoire 1000 dans le registre REG (ou *vice versa*, suivant l'ordinateur). Les adresses peuvent être générées notamment au moyen de l'indexation, des registres de base et des registres de segments.

Les adresses générées par programme sont appelées des **adresses virtuelles** et elles forment l'espace d'adressage virtuel. Dans les ordinateurs sans mémoire virtuelle, l'adresse virtuelle est placée directement sur le bus mémoire et provoque la lecture ou l'écriture du mot de même adresse physique. Dans les ordinateurs avec mémoire virtuelle, elles ne vont pas directement sur le bus mémoire mais dans une **unité de gestion mémoire** (MMU, *Memory Management Unit*) qui fait correspondre les adresses virtuelles à des adresses physiques, comme illustré à la figure 3.8.

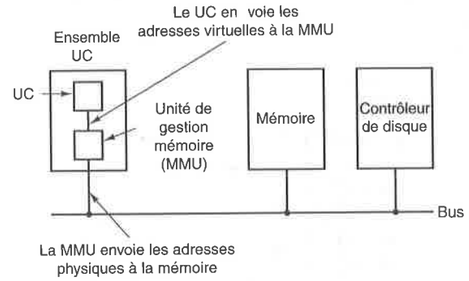

Figure 3.8 • Localisation et fonction d'une MMU. Ici, la MMU est montrée comme une partie intégrante de l'UC parce que cette configuration est usuelle de nos jours. Toutefois, elle pourrait logiquement se trouver dans un composant séparé, comme c'était le cas il y a quelques années.

La figure 3.9 donne un exemple très simple de la manière dont se fait le travail de correspondance. Dans cet exemple, un ordinateur peut produire des adresses sur 16 bits, avec des valeurs comprises entre 0 et 64 Ko : ce sont les adresses virtuelles. Toutefois, cet ordinateur a seulement 32 Ko de mémoire physique ; par conséquent, même s'il permet d'écrire des programmes de 64 Ko, ceux-ci ne peuvent pas être chargés entièrement dans la mémoire et exécutés. Il faut cependant qu'une copie complète de tout un programme – mais qui ne dépasse pas 64 Ko – soit présente sur le disque, afin que les différentes parties puissent être récupérées quand cela est nécessaire.

L'espace d'adressage virtuel est divisé en unités appelées **pages**. Les unités correspondantes dans la mémoire physique sont appelées **cadres de pages** (*page frames*). Les pages et les cadres de pages sont toujours de même taille. Ils sont de 4 Ko dans notre exemple, mais les systèmes réels utilisent des pages dont la taille varie de 512 octets à 64 Ko. Avec 64 Ko d'espace d'adressage virtuel et 32 Ko de mémoire physique, nous avons 16 pages virtuelles et 8 cadres de pages. Les transferts entre la RAM et le disque se font toujours par pages entières.

La figure 3.9 se lit de la façon suivante. L'indication 0K-4K signifie que les adresses virtuelles ou réelles de la page sont situées entre 0 et 4 095, l'indication 4K-8K,

entre 4 096 et 8 191, etc. Chaque page contient 4 096 adresses qui démarrent à un multiple de 4 096 et finissent au multiple suivant (moins un).

Quand le programme essaie par exemple d'accéder à l'adresse 0, à l'aide de l'instruction

```
MOV REG,0
```

l'adresse virtuelle 0 est envoyée à la MMU. La MMU constate que cette adresse virtuelle tombe dans la page 0 (de valeurs comprises entre 0 et 4 095), laquelle correspond au cadre de page 2 (de valeurs comprises entre 8 192 et 12 287). Elle transforme l'adresse en 8 192 et la présente sur le bus. La mémoire ne sait rien de la MMU : elle interprète seulement qu'il s'agit d'une requête de lecture ou d'écriture à l'adresse 8192, et elle la réalise. Ainsi, la MMU a effectivement fait correspondre toutes les adresses virtuelles comprises entre 0 et 4 095 à des adresses physiques comprises entre 8 192 et 12 287.

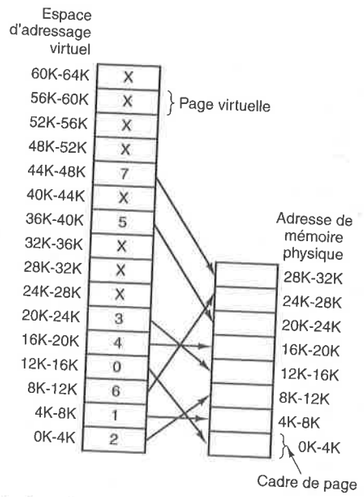

Figure 3.9 • La relation entre les adresses virtuelles et les adresses de la mémoire physique est indiquée dans la table des pages. Chaque page commence à un multiple de 4 096 et finit 4 095 adresses plus haut. 4K-8K signifie donc 4 096 à 8 191 et 8K-12K signifie 8 192 à 12 287.

De la même manière, l'instruction

```
MOV REG,8192
```

est transformée en

```
MOV REG,24576
```

parce que l'adresse virtuelle 8 192 se trouve dans la page virtuelle 2 et que cette page correspond au cadre de page 6. Autre exemple : l'adresse virtuelle 20 500 est située à

20 octets du début de la page virtuelle 5 (adresses virtuelles de 20 480 à 24 575) et correspond à l'adresse physique 12 288 + 20 = 12 308.

Cependant, cette capacité de mettre en correspondance les 16 pages virtuelles sur les 8 cadres de pages en initialisant correctement la MMU ne résout pas le problème posé par un espace d'adressage virtuel plus grand que la mémoire physique. Puisqu'il n'y a que 8 cadres de pages physiques, seules 8 des pages virtuelles de la figure 3.9 sont mises en correspondance avec de la mémoire physique. Les autres, illustrées par des croix dans la figure, ne sont pas mises en correspondance. En termes de matériel, un **bit de présence/absence** conserve la trace des pages qui se trouvent physiquement en mémoire.

Par ailleurs, que se passe-t-il si le programme essaye par exemple de faire appel à une page non présente, avec l'instruction

```
MOV REG,32780
```

qui correspond à l'octet 12 de la page virtuelle 8 (qui commence à 32 768) ? La MMU remarque que la page est absente (ce qui est indiqué par une croix dans la figure) et fait procéder l'UC à un déroutement, c'est-à-dire que le processeur est restitué au système d'exploitation. Ce déroutement, appelé **défaut de page** (*page fault*), est réalisé de la manière suivante : le système d'exploitation sélectionne un cadre de page peu utilisé et écrit son contenu sur le disque ; il transfère ensuite la page qui vient d'être référencée dans le cadre de page libéré, modifie la correspondance et recommence l'instruction déroutée.

Par exemple, si le système d'exploitation décide de déplacer le cadre de page 1, il doit charger la page virtuelle 8 à l'adresse physique 4 K et faire deux changements de correspondance dans la MMU. Tout d'abord, il doit marquer l'entrée de la page virtuelle 1 comme étant non utilisée, afin d'autoriser tout accès ultérieur aux adresses virtuelles comprises entre 4 K et 8 K. Ensuite, il devra remplacer la croix de l'entrée de la page 8 par la valeur 1. Enfin, quand l'instruction déroutée sera réexécutée, il mettra l'adresse virtuelle 32 780 en correspondance avec l'adresse physique 4 108 (4 096 + 12).

Rentrons à présent à l'intérieur d'une MMU afin de voir comment elle fonctionne et pourquoi nous avons choisi d'utiliser des tailles de pages multiples de puissances de 2. La figure 3.10 nous présente un exemple d'adresse virtuelle, 8 196 (0010000000000100 en binaire), mise en correspondance *via* la MMU de la figure 3.9. L'adresse virtuelle de 16 bits qui arrive est divisée en deux parties : un numéro de page sur 4 bits et un décalage sur 12 bits. Avec 4 bits pour le numéro de page, nous pouvons avoir 16 pages, et avec 12 bits pour le décalage, nous pouvons adresser l'ensemble des 4 096 octets d'une page.

Le numéro de page est utilisé comme un index dans la **table des pages** : le numéro du cadre de page correspond à la page virtuelle. Si le bit de présence/absence est à 0, un déroutement vers le système d'exploitation est mis en place. Si le bit est à 1, le numéro du cadre de page trouvé dans la table des pages est copié dans les 3 bits de poids le plus fort du registre de sortie auxquels sont ajoutés les 12 bits de décalage, qui sont

copiés à partir de l'adresse virtuelle entrante sans être modifiés. Ils forment ensemble une adresse physique sur 15 bits. Le registre de sortie est ensuite placé sur le bus mémoire en tant qu'adresse de mémoire physique.

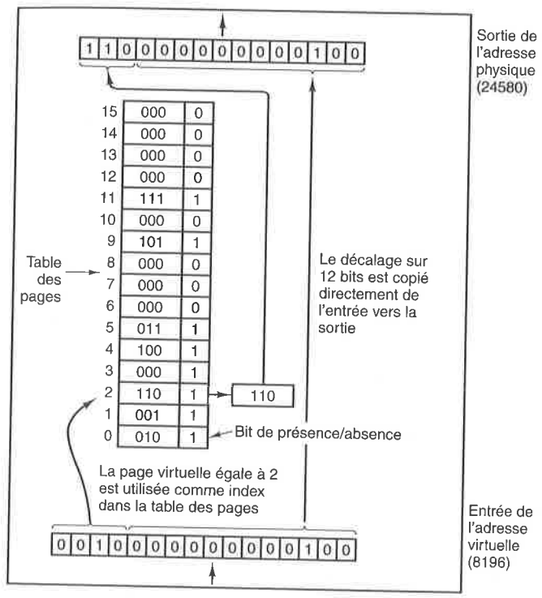

Figure 3.10 • Fonctionnement interne d'une MMU avec 16 pages de 4 Ko.

3.3.2 Les tables des pages

Dans le cas le plus simple, la mise en correspondance des adresses virtuelles et des adresses physiques se déroule comme nous venons de le décrire. L'adresse virtuelle est séparée en deux parties : un numéro de page virtuelle (les bits de poids fort) et un décalage (les bits de poids faible). Par exemple, avec une adresse de 16 bits et une taille de page de 4 Ko, les 4 bits de poids fort peuvent spécifier une des 16 pages virtuelles ; les 12 bits de poids faible indiqueront alors le décalage en octet (de 0 à 4 095) à l'intérieur de la page sélectionnée. La page peut aussi être découpée en 3 ou 5 bits (ou tout autre nombre). Évidemment, la taille des pages diffère en fonction des découpages.

Dans la table des pages, le numéro de page virtuelle sert à trouver l'entrée de la page virtuelle. S'il existe, le numéro du cadre de page est trouvé à partir de cette entrée de la table des pages. Le numéro du cadre de page est attribué aux bits de poids fort suivi

du décalage, remplaçant le numéro de page virtuelle, pour composer une adresse physique qui peut être envoyée à la mémoire.

Le rôle d'une table des pages est ainsi de faire correspondre des pages virtuelles à des cadres de pages. Mathématiquement, une table des pages est une fonction, qui a comme argument le numéro de la page virtuelle et comme résultat le numéro du cadre physique. Le résultat de cette fonction peut servir à remplacer le champ nommé page virtuelle d'une adresse virtuelle par un champ nommé cadre de page et à créer ainsi une adresse mémoire physique.

Structure d'une entrée de table des pages

Laissons maintenant la structure des tables des pages en général pour examiner en détail une seule entrée de table des pages. La composition exacte d'une entrée dépend fortement de la machine, mais le genre d'informations présentes est sensiblement le même d'une machine à l'autre. À la figure 3.11, nous donnons un exemple simple d'une entrée de table. La taille varie d'une machine à l'autre, mais la valeur 32 bits est courante. Le champ le plus important est le `numéro du cadre de page`. En effet, l'objectif est de localiser cette valeur. Après ce champ vient le bit de présence/absence. Si ce bit est à 1, l'entrée est valide et peut être utilisée. S'il est à 0, la page virtuelle à laquelle l'entrée appartient n'est pas actuellement en mémoire. L'accès à l'entrée d'une table des pages dont le bit est à 0 provoque un défaut de page.

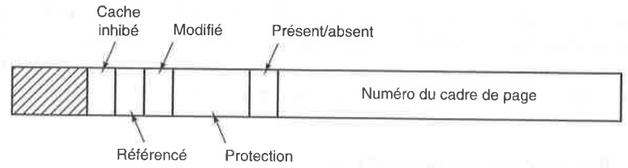

Figure 3.11 • Entrée typique d'une table des pages.

Les bits *de protection* précisent quelles sortes d'accès sont permis. Dans sa forme la plus simple, ce champ contient 1 bit, avec la valeur 0 pour un accès en lecture/écriture et la valeur 1 pour un accès en lecture seule. Un schéma plus sophistiqué proposera 3 bits, c'est-à-dire un bit pour chaque autorisation : un pour la lecture, un pour l'écriture et un pour l'exécution.

Les bits *modifié* et *référencé* conservent une trace de l'utilisation de la page. Quand une page a un accès en écriture, le matériel met automatiquement à 1 le bit modifié. Ce bit est nécessaire quand le système d'exploitation décide de récupérer un numéro de cadre de page. Si cette page a été modifiée (on dit qu'elle est *dirty*), elle doit être écrite sur le disque. Si elle n'a pas été modifiée (on dit qu'elle est *clean*), on peut juste l'abandonner, puisque la copie disque est encore valide. Ce bit est quelquefois appelé **bit dirty**.

Le bit *référencé* est mis à 1 chaque fois qu'une page est consultée, que ce soit en lecture ou en écriture. Cette valeur permet d'aider le système d'exploitation à choisir la page à évincer quand un défaut de page est généré. Les pages qui ne sont pas en cours d'utilisations sont de meilleures candidates que les autres, et ce bit joue un rôle important dans plusieurs algorithmes de remplacements de pages que nous étudierons plus loin dans ce chapitre.

Enfin, le dernier bit permet d'inhiber le cache pour une page. Cette caractéristique est importante pour les pages qui sont mises en correspondance dans des registres matériels plutôt que dans la mémoire. Si le système d'exploitation se trouve dans une boucle en attente de réponse d'un périphérique d'E/S pour une commande qui vient d'être donnée, il est essentiel que le matériel accède au mot donné par le périphérique plutôt que de se servir d'une vieille copie conservée en cache. Avec ce bit, le cache peut être inhibé. Les machines qui ont un espace d'E/S séparé et qui n'utilisent pas de mise en correspondance de la mémoire avec des E/S n'ont pas besoin de ce bit.

Notons que l'adresse disque qui sert à avoir la page quand elle n'est pas en mémoire ne fait pas partie de la table des pages. La raison est simple : la table des pages garde seulement les informations dont le matériel a besoin pour traduire l'adresse virtuelle en adresse physique. Les informations dont le système d'exploitation a besoin pour traiter les défauts de pages sont conservées dans une table logicielle à l'intérieur du système d'exploitation. Le matériel n'en a pas besoin.

Avant d'aller plus loin dans les implémentations, répétons que la mémoire virtuelle crée une abstraction de la mémoire réelle (l'espace d'adressage), un peu comme le processus est une abstraction du processeur physique (l'UC). On peut implémenter la mémoire virtuelle en partitionnant l'espace d'adressage en pages, chaque page étant ensuite mappée (ou pas) sur de la mémoire physique ; ce chapitre concerne donc une abstraction créée par le système d'exploitation et la façon dont il la gère.

3.3.3 Accélérer la pagination

Nous avons maintenant les éléments de base de la pagination. Essayons d'aller plus loin. Tout système de pagination doit considérer deux éléments :

1. La correspondance doit être rapide.
2. La table des pages doit être extrêmement grande.

Le premier point découle du fait que la correspondance entre le virtuel et le physique doit être réalisée à chaque référence mémoire. Une instruction typique a un mot d'instruction et, souvent, un opérande mémoire. Par conséquent, pour chaque instruction, il est nécessaire de faire référence à la table des pages une fois, deux fois et parfois plus. Par exemple, si une instruction prend 1 ns, la table des pages doit être parcourue en moins de 0,2 ns pour éviter de devenir un goulot d'étranglement.

Le second point provient du fait que les ordinateurs modernes utilisent des adresses virtuelles d'au moins 32 bits, une taille de 64 bits devenant même de plus en plus courante. Par exemple, dans le cas d'une page de 4 Ko, un espace d'adressage de

32 bits a 1 million de pages, et un espace d'adressage de 64 bits en a plus que ce que l'on peut imaginer. Avec 1 million de pages dans l'espace d'adressage virtuel, la table des pages doit avoir 1 million d'entrées. En outre, il faut se rappeler que chaque processus a besoin de sa propre table des pages (parce qu'il a son propre espace d'adressage virtuel).

La nécessité que les correspondances de pages soient rapides représente une contrainte importante quant à la manière de construire les ordinateurs. Le schéma le plus simple (au moins du point de vue conceptuel) consiste à avoir une table des pages unique composée d'un tableau de registres matériels rapides, avec une entrée pour chaque page virtuelle, indexée par un numéro de page virtuelle, comme montré à la figure 3.10. Quand un processus démarre, le système d'exploitation charge les registres avec la table des pages des processus à partir d'une copie conservée en mémoire principale. Durant l'exécution du processus, aucune autre référence mémoire n'est nécessaire pour la table des pages. Cette méthode présente l'avantage d'être directe et de ne nécessiter aucune référence durant la correspondance. En revanche, elle a l'inconvénient d'être potentiellement coûteuse (si la table des pages est grande). En outre, le fait de devoir charger la table des pages en totalité à chaque changement de contexte met à mal les performances.

Dans un schéma plus complexe, la table des pages peut être entièrement chargée en mémoire principale. Le matériel a uniquement besoin d'un simple registre qui pointe sur le début de la table des pages. Cette conception permet de modifier la représentation en mémoire lors d'un changement de contexte en rechargeant un registre. Bien entendu, l'inconvénient est qu'elle demande une ou plusieurs références mémoire pour lire les entrées de la table des pages pendant l'exécution de chaque instruction, ce qui ralentit notablement l'exécution.

La mémoire associative

Examinons à présent la façon dont on accélère la pagination et dont on gère les grands espaces virtuels. Partons tout d'abord du principe que la table des pages est en mémoire. Cette hypothèse a un impact considérable sur les performances. Considérons, par exemple, une instruction qui copie un registre dans un autre. En l'absence de pagination, cette instruction fait une seule référence mémoire pour s'exécuter. Avec la pagination, des références mémoire supplémentaires seront nécessaires pour accéder à la table des pages. Puisque la vitesse d'exécution est généralement limitée par la cadence à laquelle l'UC extrait les instructions et les données de la mémoire, lorsqu'il faut faire deux références de table des pages pour une référence mémoire, les performances sont réduites de moitié. Dans de telles conditions, personne ne voudrait se servir d'un tel système.

Les concepteurs d'ordinateurs, qui connaissent ce problème depuis longtemps, ont trouvé une solution qui résulte de l'observation que la plupart des programmes tendent à faire un grand nombre de références à un petit nombre de pages. Ainsi, seule une petite fraction des entrées de la table des pages est réellement lue ; tout le reste sert rarement.

La solution consiste à équiper les ordinateurs avec un petit périphérique matériel qui permet de mettre en correspondance les adresses virtuelles avec les adresses physiques sans passer par la table des pages. Ce périphérique, appelé **TLB** (*Translation Lookaside Buffer*) et parfois **mémoire associative**, est illustré à la figure 3.12. Il se trouve habituellement à l'intérieur de la MMU et consiste en un petit nombre d'entrées (8 dans l'exemple) qui sont rarement plus de 64. Chaque entrée contient les informations d'une page : le numéro de page virtuelle, un bit mis à 1 quand la page est modifiée, le code de protection (permissions en lecture/écriture/exécution) et le cadre de page physique dans lequel la page est située. Ces champs ont une correspondance point à point avec les champs de la table des pages. Un autre bit indique si l'entrée est valide (c'est-à-dire utilisée) ou non.

À titre d'exemple, le TLB de la figure 3.12 pourrait être généré par un processus dans une boucle qui englobe les pages virtuelles 19, 20 et 21 ; ces entrées du TLB ont des codes de protection en lecture et exécution. Les données principales en cours d'utilisation (par exemple, un tableau en cours de traitement) se trouvent dans les pages 129 et 130. La page 140 contient les indices employés dans les calculs du tableau. Enfin, la pile se trouve dans les pages 860 et 861.

Figure 3.12 • TLB qui accélère la pagination.

Valide	Page virtuelle	Modifié	Protection	Cadre de page
1	140	1	RW	31
1	20	0	R-X	38
1	130	1	RW	29
1	129	1	RW-	62
1	19	0	R-X	50
1	21	0	R-X	45
1	860	1	RW	14
1	861	1	RW	75

Voyons maintenant comme le TLB fonctionne. Quand une adresse virtuelle est présentée à la MMU pour être traduite, le matériel vérifie d'abord si ce numéro de page virtuelle est présent dans le TLB en le comparant simultanément (c'est-à-dire en parallèle) à toutes les entrées. Si une correspondance est trouvée et si l'accès n'enfreint pas les bits de protection, le cadre de page est pris directement dans le TLB, sans passer par la table des pages. Si le numéro de page virtuelle est présent dans le TLB mais que l'instruction essaie d'écrire dans une page autorisée en lecture seule, une erreur de protection est générée.

Il est intéressant d'étudier ce qui se passe lorsque le numéro de page virtuelle n'est pas dans le TLB. La MMU détecte l'erreur et parcourt normalement la table des pages.

Elle évince une des entrées du TLB et la remplace par l'entrée de la table des pages trouvée. Ainsi, si cette page est réutilisée rapidement, il en résultera cette fois un accès plutôt qu'une erreur. Quand une entrée est sortie du TLB, le bit modifié est recopié dans la table des pages en mémoire. Les autres valeurs sont déjà présentes. Quand le TLB est chargé depuis la table des pages, tous les champs sont copiés à partir de la mémoire.

Gestion logicielle des TLB

Jusqu'à présent, nous avons admis que chaque machine avec une mémoire virtuelle paginée avait des tables des pages reconnues par le matériel, plus un TLB. Dans cette conception, la gestion du TLB et des erreurs générées par le TLB est entièrement assurée par le matériel de la MMU. Les déroutements vers le système d'exploitation surviennent uniquement lorsqu'une page ne se trouve pas en mémoire.

Dans le passé, cette supposition était avérée. Toutefois, de nombreuses machines modernes de type RISC, telles que les SPARC, MIPS, Alpha et HP PA, assurent quasiment toute la gestion de page de manière logicielle. Sur ces machines, les entrées du TLB sont explicitement chargées par le système d'exploitation. Quand une erreur de TLB se produit, au lieu que la MMU parcoure les tables des pages pour trouver et exécuter la référence de page nécessaire, cela génère simplement une faute de TLB et renvoie le problème au système d'exploitation. Le système doit trouver la page, effacer une entrée du TLB, en entrer une nouvelle et recommencer l'instruction qui a engendré le défaut. Naturellement, tout cela doit se faire en un petit nombre d'instructions parce que les erreurs de TLB se produisent plus fréquemment que les défauts de pages.

Contre toute attente, si le TLB est assez grand (64 entrées, par exemple) pour réduire le taux d'erreur, la gestion logicielle du TLB est d'une efficacité satisfaisante. Le principal avantage est ici une plus grande simplicité de la MMU, laquelle libère une quantité considérable de zone sur le circuit intégré de l'UC, pour des caches ou d'autres fonctions qui peuvent accroître les performances.

Des stratégies diverses ont été développées pour augmenter les performances sur des machines qui gèrent les TLB de manière logicielle. Une de ces approches cherche à réduire à la fois les erreurs de TLB et le coût d'une erreur de TLB quand elle se produit. Pour réduire les erreurs de TLB, le système d'exploitation peut parfois se servir de son intuition pour imaginer quelles pages sont susceptibles d'être les prochaines utilisées et pour précharger les entrées de celles-ci dans le TLB. Par exemple, quand un processus client envoie un message au processus serveur sur la même machine, il est très probable que le serveur devra s'exécuter sous peu. Sachant cela, pendant le traitement du déroutement pour faire l'envoi, le système peut aussi vérifier où se trouvent les pages de code, de données et de pile du serveur, et faire leur mise en correspondance avant qu'elles ne causent des défauts de TLB.

Pour traiter une erreur de TLB, par matériel ou par logiciel, le plus courant est d'aller à la table des pages et d'exécuter les opérations d'indexation pour localiser la page référencée. Le problème, quand on procède par logiciel, est que les pages incluses

dans la table des pages peuvent ne pas se trouver dans le TLB, ce qui engendre des défauts supplémentaires de TLB durant le traitement. On peut limiter ces défauts en gérant un grand cache logiciel (de 4 Ko, par exemple) d'entrées de TLB dans une zone définie dont la page est toujours conservée dans le TLB. En examinant en premier lieu le cache logiciel, le système d'exploitation peut réduire de manière importante les erreurs de TLB.

Il faut bien comprendre, à ce propos, la différence entre deux types d'erreurs. On a une erreur logicielle (*soft miss*) lorsque la page référencée n'est pas dans le TLB mais est tout de même en mémoire. Il faut alors mettre à jour le TLB, ce qui se fait en 10-20 instructions machine et donc en quelques nanosecondes. Dans le cas d'une erreur matérielle (*hard miss*), la page n'est pas en mémoire (et donc évidemment pas dans le TLB) et doit être cherchée sur le disque, ce qui prend plusieurs millisecondes. Le traitement d'une erreur matérielle est ainsi des millions de fois plus lent que celui d'une erreur logicielle.

3.3.4 Les tables des pages des grandes mémoires

On utilise les TLB pour accélérer la traduction d'adresses virtuelles en adresses physiques, mais ce n'est pas le seul problème à traiter. Il faut aussi parfois gérer de très grandes mémoires. Nous allons maintenant voir deux façons de le faire.

Tables des pages multiniveaux

La première façon consiste à se servir d'une table des pages multiniveaux. Un exemple simple est donné à la figure 3.13. À la figure 3.13(a), nous avons une adresse virtuelle 32 bits qui est partitionnée en un champ de 10 bits nommé `PT1`, un champ de 10 bits nommé `PT2` et un champ de 12 bits nommé `Décalage`. Puisque les décalages sont sur 12 bits, les pages font 4 Ko et nous avons un total de 2^{20} pages.

Avec la méthode des tables des pages multiniveaux, le secret consiste à éviter de garder tout le temps en mémoire toutes les tables des pages, en particulier celles dont nous n'avons pas besoin. Supposons, par exemple, que le processus ait besoin de 12 Mo : 4 Mo de mémoire inférieure pour le texte du programme, les 4 Mo suivants pour les données et 4 Mo de mémoire supérieure pour la pile. Entre le haut des données et le bas de la pile existe un gigantesque trou inutilisé.

À la figure 3.13(b), nous voyons comment la table des pages à deux niveaux fonctionne. Sur la gauche, nous avons la table des pages de niveau 1, avec 1 024 entrées, correspondant au champ de 10 bits `PT1`. Quand une adresse virtuelle est présentée à la MMU, elle extrait en premier le champ `PT1` et emploie cette valeur comme index dans la table des pages de niveau 1. Chacune de ces 1 024 entrées représente 4 Mo parce que la totalité des 4 Go (c'est-à-dire 32 bits) de l'espace d'adressage virtuel a été découpée en morceaux de 4 096 octets.

L'entrée localisée par l'indexation de la table des pages de niveau 1 produit l'adresse ou le numéro de cadre de page de la table des pages de second niveau. L'entrée 0 de la table des pages de niveau 1 pointe sur la table des pages du corps du programme,

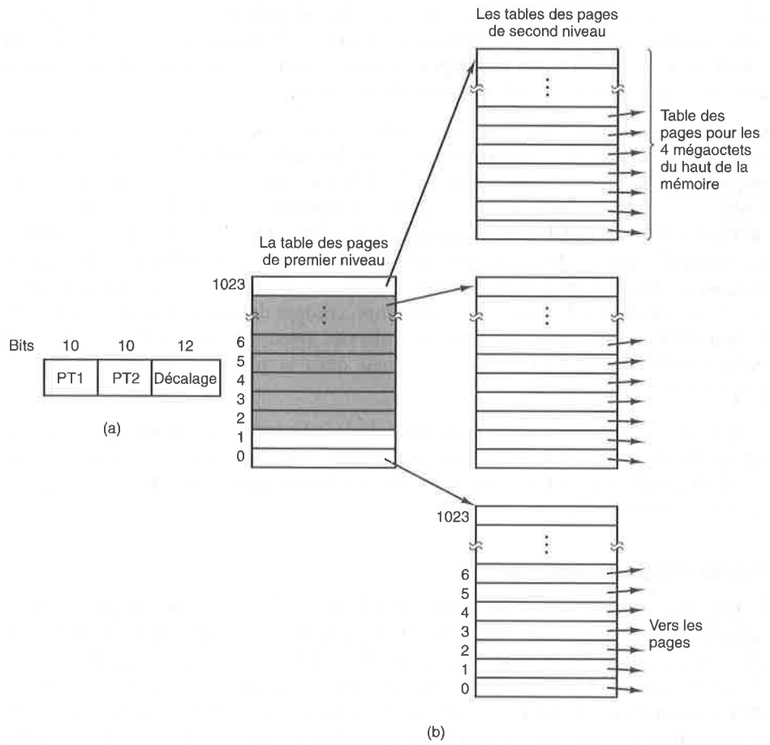

Figure 3.13 • (a) Une adresse 32 bits avec deux champs de table des pages. (b) Des tables des pages à deux niveaux.

l'entrée 1 pointe sur la table des pages des données, et l'entrée 1 023 pointe sur la table des pages de la pile. Les autres entrées (grisées) ne sont pas utilisées. Le champ `PT2` est maintenant employé comme index dans la table des pages de second niveau sélectionnée pour trouver le numéro du cadre de page de la page elle-même.

À titre d'exemple, considérons l'adresse virtuelle sur 32 bits 0x00403004 (soit 4 206 596 en décimal) qui est la valeur 12 292 dans le bloc. Cette adresse virtuelle correspond à `PT1` = 1, `PT2` = 3 et `Décalage` = 4. La MMU se sert d'abord de *PT1* comme index dans la table des pages du premier niveau et récupère l'entrée 1, laquelle correspond aux adresses de 4 Mo à 8 Mo. En employant *PT2* comme index dans la table des pages de second niveau que nous venons de trouver, nous obtenons l'entrée 3, qui correspond aux adresses 12 288 à 16 383 à l'intérieur du bloc de 4 Mo (c'est-à-dire aux adresses absolues de 4 206 592 à 4 210 687). Cette entrée contient le numéro du cadre de page qui comprend l'adresse virtuelle 0x00403004. Si cette page ne se trouve

pas en mémoire, son bit de présence/absence dans la table des pages sera à 0 et provoquera un défaut de page. Si la page est en mémoire, le numéro de cadre de page est combiné avec la valeur de décalage (4) pour construire l'adresse physique. Cette adresse est placée sur le bus et envoyée à la mémoire.

Concernant la figure 3.13, il est intéressant de noter que, bien que l'espace d'adressage contienne plus d'un million de pages, seules quatre tables des pages sont en fait nécessaires : la table de premier niveau, les tables des pages de second niveau – de 0 à 4 Mo et de 4 à 8 Mo – et les 4 Mo du haut de la mémoire. Les bits de présence/absence de 1 021 entrées de la table des pages de premier niveau sont mis à 0, ce qui provoque un défaut de page si elles sont demandées. Le cas échéant, le système d'exploitation remarquerait que le processus essaie de référencer de la mémoire dont il ne dispose pas et prendrait une décision en conséquence, comme de lui envoyer un signal ou de le tuer. Dans cet exemple, nous avons choisi des valeurs arrondies pour les diverses tailles et sélectionné *PT1* égal à *PT2*, mais dans la pratique, on peut également employer d'autres valeurs.

Le système de table des pages à deux niveaux de la figure 3.13 peut être étendu à trois ou quatre niveaux, voire davantage. Les niveaux supplémentaires donnent plus de flexibilité, mais il n'est pas certain que la complexité supplémentaire vaille la peine au-delà de trois niveaux.

Tables des pages inversées

Lorsqu'on a des espaces d'adressage virtuel sur 32 bits, les tables de pages multiniveaux fonctionnent bien. Toutefois, les processeurs 64 bits devenant monnaie courante, la situation est en train de changer. Si l'espace d'adressage est maintenant de 2^{64} octets, avec des pages de 4 Ko, il faut une table des pages avec 2^{52} entrées. Si chaque entrée est sur 8 bits, la taille de la table sera supérieure à 30 millions de gigaoctets. Or, utiliser plus de 30 millions de gigaoctets uniquement pour la table des pages n'est pas faisable aujourd'hui, ni envisageable dans les années à venir. Par conséquent, il faut trouver une autre solution pour les espaces d'adressage virtuel paginés de 64 bits.

Une solution possible est la **table des pages inversée**. Dans cette conception, on trouve une entrée par cadre de page dans la mémoire réelle, plutôt qu'une entrée par page de l'espace d'adressage virtuel. Par exemple, avec des adresses virtuelles de 64 bits, une page de 4 Ko et 1 Go de RAM, une table des pages inversée a besoin de 262 144 entrées. L'entrée conserve la trace de ce qui est localisé (processus, page virtuelle) dans le cadre de page.

Bien que les tables des pages inversées économisent beaucoup d'espace, du moins quand l'espace d'adressage virtuel est bien plus grand que la mémoire physique, elles présentent un sérieux inconvénient : la traduction du virtuel en physique devient largement plus difficile. Quand le processus *n* référence la page virtuelle *p*, le matériel ne peut pas trouver la page physique en se servant de *p* comme index dans la table des pages. Au lieu de cela, il doit chercher dans toute la table des pages inversée une entrée du type (*n*, *p*). De plus, cette recherche doit être effectuée sur chaque référence mémoire, pas seulement sur les défauts de pages. Or, explorer une table de 256 K

entrées à chaque référence mémoire n'est sûrement pas une bonne façon d'exploiter une machine.

Face à ce dilemme, la solution est d'avoir recours au TLB. En effet, s'il peut conserver toutes les pages très souvent utilisées, les traductions peuvent être réalisées aussi rapidement qu'avec des tables des pages normales. En cas d'erreur de TLB, cependant, la table des pages inversée doit être parcourue de façon logicielle. Pour mener cette recherche, une possibilité est d'avoir une table de hachage avec les adresses virtuelles comme clés. Toutes les pages virtuelles actuellement en mémoire qui ont la même valeur de hachage sont chaînées ensemble, comme l'illustre la figure 3.14. Si la table de hachage a autant d'emplacements que la machine a de pages physiques, la chaîne moyenne ne sera longue que d'une entrée, ce qui améliore grandement la mise en correspondance. Une fois le numéro de cadre de page trouvé, la nouvelle paire (virtuelle, physique) est entrée dans le TLB.

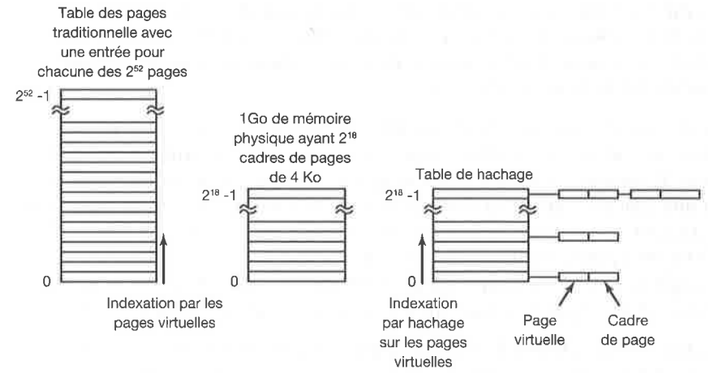

Figure 3.14 • Comparaison entre une table des pages traditionnelle et une table des pages inversée.

Les tables des pages inversées sont couramment utilisées sur les machines 64 bits parce que, même avec une grande taille de page, le nombre d'entrées dans la table des pages est énorme. Par exemple, avec des pages de 4 Mo et des adresses virtuelles sur 64 bits, il faut 2^{42} entrées.

3.4 Les algorithmes de remplacements de pages

Quand un défaut de page se produit, le système d'exploitation doit choisir une page à enlever de la mémoire afin de faire de la place pour la page qui doit être chargée. Si la page qui doit être supprimée a été modifiée quand elle était en mémoire, elle doit être réécrite sur le disque pour mettre à jour la copie disque. Toutefois, si la page n'a pas