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.

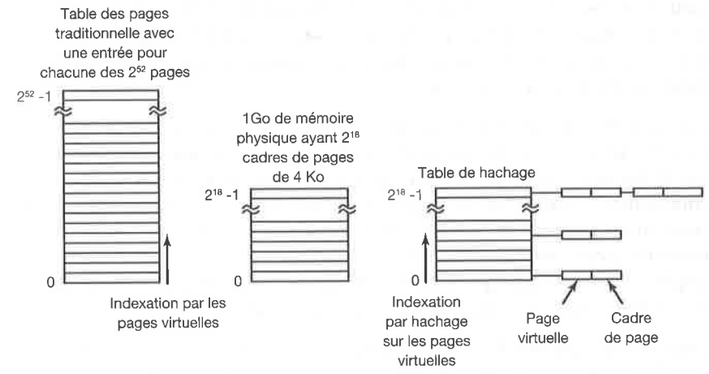

Figure 3.14 • Comparaison entre une table des pages traditionnelle et une table des pages inversée.

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

changé (parce qu'elle contient par exemple le texte d'un programme), la copie disque est déjà à jour et ne nécessite pas de réécriture. La page à lire écrasera simplement la page devant être évincée.

Bien qu'il soit possible de choisir au hasard la page à supprimer à chaque défaut de page, les performances du système seront meilleures si c'est une page peu utilisée qui est choisie. Si une page souvent employée est enlevée, il faudra probablement qu'elle soit rechargée rapidement, ce qui entraînera un dépassement supplémentaire. De nombreux travaux ont été réalisés sur les algorithmes de changements de pages, en théorie comme en pratique. Nous décrivons plus loin certains des algorithmes les plus importants.

Il faut noter que le problème de remplacement de page se rencontre dans d'autres domaines de la conception d'ordinateur. Par exemple, la plupart des ordinateurs ont une ou plusieurs mémoires cache composées de blocs de mémoire de 32 ou 64 bits. Quand le cache est plein, il faut choisir un bloc pour l'enlever. Ce problème est précisément le même que celui du remplacement de page, excepté pour la durée (il doit être réalisé en quelques nanosecondes, et non en quelques millisecondes comme dans le cas du remplacement de page). Cette durée est plus courte parce que les échecs de blocs de cache sont traités depuis la mémoire principale, laquelle n'a pas de temps de recherche, ni de temps d'attente.

Prenons comme deuxième exemple un serveur Web. Le serveur peut conserver un certain nombre de pages Web récemment utilisées dans sa mémoire cache. Toutefois, lorsque la mémoire cache est pleine et qu'une nouvelle page est référencée, une décision doit être prise quant au choix de la page Web à évincer. Cela est également valable concernant les pages de la mémoire virtuelle, excepté pour le fait que les pages Web ne sont jamais modifiées dans le cache, et qu'il existe par conséquent toujours une copie récente sur le disque. Dans un système de mémoire virtuelle, les pages dans la mémoire principale peuvent être soit bonnes (*clean*), soit modifiées (*dirty*).

Avec tous les algorithmes de remplacement que nous allons étudier se pose une question préalable : doit-on supprimer de la mémoire une page appartenant au processus qui en demande une nouvelle, ou peut-on éliminer une page d'un autre processus ? Dans le premier cas, nous limitons le processus à un certain nombre de pages ; pas dans le second. Les deux sont possibles, nous y reviendrons en section 3.5.1.

3.4.1 L'algorithme optimal de remplacement de page

Le meilleur algorithme de remplacement de page imaginable est facile à décrire mais impossible à implanter. Il correspond à la description suivante. Au moment où un défaut de page est généré, plusieurs ensembles de pages sont en mémoire. Une de ces pages sera référencée par une très prochaine instruction (la page qui contient celle-ci). Il se peut que d'autres pages ne soient pas référencées avant la dixième, la centième ou la millième instruction qui suit. Chaque page peut être étiquetée avec le nombre d'instructions qui seront exécutées avant que cette page ne soit référencée.

L'algorithme de page optimal dit simplement que c'est la page dont l'étiquette est la plus grande qui sera enlevée. Si une page n'est pas utilisée avant 8 millions d'instructions

et une autre page avant 6 millions d'instructions, la suppression de la première repousse le défaut de page aussi tard que possible. Les ordinateurs, comme les humains, essayent de retarder autant que possible la gestion des événements désagréables.

Le seul problème avec cet algorithme est qu'il est irréalisable. Au moment où survient le défaut de page, le système d'exploitation n'a aucun moyen de savoir quand chacune de ces pages sera référencée la prochaine fois. (Nous avons vu précédemment une situation similaire avec l'algorithme d'ordonnancement du plus court travail d'abord. Comment le système peut-il dire quel est le travail le plus court ?) Pourtant, en exécutant un programme dans un simulateur et en gardant une trace des références de toutes les pages, il est possible d'implanter l'algorithme optimal de remplacement de page dans une *deuxième* exécution, en employant les informations de références de pages récoltées durant la *première* exécution.

De cette manière, nous pouvons comparer les performances d'algorithmes réalisables avec celles du meilleur algorithme. Par exemple, si un système d'exploitation réalise une performance inférieure de seulement 1 % à celle de l'algorithme optimal, il est inutile de rechercher un meilleur algorithme.

Pour éviter toute confusion, il doit être clair que l'enregistrement des références de pages renvoie à un seul programme mesuré et donc à un seul jeu spécifique d'entrées. L'algorithme de remplacement de page qui en découle est ainsi spécifique à un programme et à des données d'entrée. Bien que cette méthode soit utile pour l'évaluation des algorithmes de remplacements de pages, les systèmes ne s'en servent pratiquement pas. Nous étudions ci-après des algorithmes qui sont utiles dans les systèmes réels.

3.4.2 L'algorithme de remplacement de la page non récemment utilisée

Afin de permettre au système d'exploitation de collecter des statistiques utiles sur le nombre de pages utilisées et le nombre de pages inutilisées, la plupart des ordinateurs avec de la mémoire virtuelle possèdent 2 bits d'état associés à chaque page. Le bit R est mis à 1 à chaque fois que la page est référencée (lue ou écrite) et le bit M est mis à 1 quand la page est réécrite (c'est-à-dire modifiée). Ces bits sont contenus dans chaque entrée de la table des pages, comme l'illustre la figure 3.11. Il est important de réaliser que ces bits doivent être mis à jour à chaque référence mémoire, et qu'il est donc essentiel que cette mise à jour soit matérielle. Lorsqu'un bit a été mis à 1, il reste à 1 jusqu'à ce que le système d'exploitation le remette à 0.

Si le matériel ne dispose pas de ces bits, ils peuvent être simulés de la manière suivante. Quand un processus démarre, toutes les entrées de ses pages sont marquées comme non présentes en mémoire. Dès qu'une page est référencée, un défaut de page est généré. Le système d'exploitation met à 1 le bit R (dans ses tables internes), change l'entrée de la table des pages pour pointer sur la bonne page, avec un mode lecture seule, et recommence l'instruction. Si la page est réécrite par la suite, un autre défaut de page est généré, autorisant le système d'exploitation à mettre à 1 le bit M et à changer le mode d'accès à la page en lecture/écriture.

Les bits R et M peuvent servir à construire un algorithme simple de pagination comme décrit ci-après. Quand un processus démarre, les 2 bits de page pour toutes ses pages sont mis à 0 par le système d'exploitation. Périodiquement (par exemple à chaque interruption d'horloge), le bit R est effacé, afin de distinguer les pages qui n'ont pas été récemment référencées de celles qui l'ont été.

Quand un défaut est généré, le système d'exploitation examine toutes les pages et les sépare en quatre catégories basées sur les valeurs courantes des bits R et M :

- **Classe 0.** Non référencée, non modifiée.
- **Classe 1.** Non référencée, modifiée.
- **Classe 2.** Référencée, non modifiée.
- **Classe 3.** Référencée, modifiée.

Bien que l'existence de pages de classe 1 semble impossible à première vue, celles-ci sont générées lorsque le bit R d'une page de classe 3 est effacé par une interruption d'horloge. Les interruptions d'horloge n'effacent pas le bit M parce que cette information est nécessaire pour savoir si la page a été réécrite sur le disque ou pas. Si le bit R est effacé mais pas le bit M, cela aboutit à une page de classe 1.

L'algorithme de la page non récemment utilisée ou **NRU** (*Not Recently Used*) enlève une page au hasard dans la classe la plus basse non vide. Avec cet algorithme, il est implicite qu'il vaut mieux enlever une page modifiée qui n'a pas été référencée au moins dans un top (typiquement 20 ms) plutôt qu'une page bonne (*clean*) qui est beaucoup utilisée. L'attrait principal de l'algorithme NRU est qu'il est facile à comprendre, relativement simple à implanter, et qu'il donne des performances qui, sans être optimales, sont suffisantes.

3.4.3 L'algorithme de remplacement de page premier entré, premier sorti

Un autre algorithme de pagination est l'algorithme premier entré, premier sorti ou **FIFO** (*First-In, First-Out*, premier entré, premier sorti). Pour illustrer son fonctionnement, considérons un supermarché qui a suffisamment de rayons pour présenter exactement k produits différents. Un jour, une société introduit un nouveau produit alimentaire : un yaourt biologique desséché congelé qui peut être reconstitué dans un four micro-onde. Le succès étant immédiat, notre supermarché doit se débarrasser de l'un de ses vieux produits pour présenter le nouveau.

Une des possibilités consiste à trouver le produit que le supermarché a vendu le plus longtemps (qu'il a par exemple commencé à vendre il y a 120 ans) et de s'en débarrasser au motif qu'il n'intéresse plus personne. En effet, le supermarché maintient une liste chaînée de tous les produits couramment vendus dans l'ordre de leur introduction. Le nouveau arrive en bas de la liste ; celui qui est en tête de liste est abandonné.

La même idée est applicable à un algorithme de remplacement de page. Le système d'exploitation maintient une liste de toutes les pages couramment en mémoire, la

page la plus ancienne étant en tête de liste et la page la plus récente en queue. Dans le cas d'un défaut de page, la page en tête de liste est effacée et la nouvelle page ajoutée en queue de liste. Appliqué au stockage, l'algorithme FIFO peut enlever la paraffine mais le problème est qu'il peut également supprimer la farine, le sel et le beurre. Or ce problème survient aussi quand l'algorithme est appliqué aux ordinateurs. C'est pour cette raison que l'on a rarement recours à l'algorithme FIFO dans cette forme.

3.4.4 L'algorithme de remplacement de page de la seconde chance

On peut apporter une modification simple à l'algorithme FIFO afin d'éviter la suppression d'une page à laquelle on a couramment recours : il s'agit d'inspecter le bit R de la page la plus ancienne. S'il est à 0, la page est à la fois ancienne et inutilisée, elle est donc remplacée immédiatement. Si le bit R est à 1, le bit est effacé, la page est placée à la fin de la liste des pages, et son temps de chargement est mis à jour comme si elle venait d'arriver en mémoire. La recherche continue alors.

L'opération de cet algorithme, appelée **seconde chance**, est illustrée à la figure 3.15. À la figure 3.15(a), nous voyons que les pages de A à H se trouvent dans une liste chaînée, triées par leur temps d'arrivée en mémoire.

Supposons qu'un défaut de page se produise à l'instant 20. La page la plus ancienne est A, qui est arrivée à l'instant 0, quand le processus a démarré. Si le bit R de la page A est à 0, elle est évincée de la mémoire : elle est soit écrite sur le disque (si elle est modifiée), soit simplement abandonnée (si elle est bonne). D'un autre côté, si le bit R est à 1, A est placée à la fin de la liste et son temps de chargement est mis à jour avec le temps courant (20). Le bit R est aussi mis à 0. La recherche d'une page convenable continue alors avec B.

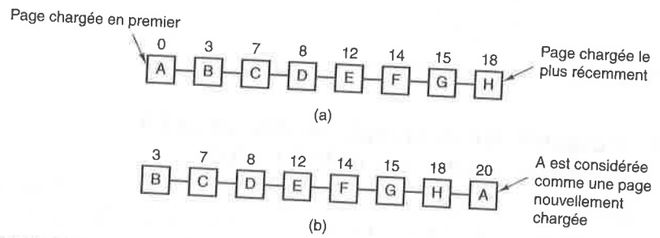

Figure 3.15 • Opération de la seconde chance. (a) Pages triées dans un ordre FIFO. (b) Liste de pages lorsqu'un défaut de page se produit à l'instant 20 et que le bit R de A est à 1. Les nombres au-dessus des pages correspondent à l'instant de leur chargement.

Le travail de l'algorithme de la seconde chance consiste à chercher une ancienne page qui n'a pas été référencée dans le précédent intervalle d'horloge. Si toutes les pages ont été référencées, l'algorithme de la seconde chance dégénère en pur algorithme FIFO. Pour être concrets, imaginons que toutes les pages de la figure 3.15(a) ont leur

bit *R* à 1. Le système d'exploitation déplace une par une les pages à la fin de la liste, effaçant le bit *R* chaque fois qu'il ajoute une page à la fin de la liste. Finalement, il revient à la page *A*, dont le bit *R* est maintenant à 0. À ce moment-là, *A* est évincée. Ainsi, cet algorithme se termine toujours.

3.4.5 L'algorithme de remplacement de page de l'horloge

Bien que l'algorithme de la seconde chance soit un algorithme satisfaisant, il manque d'efficacité parce qu'il déplace constamment des pages dans sa liste. Une meilleure approche consiste à garder tous les cadres de pages dans une liste circulaire formant une horloge, comme l'illustre la figure 3.16. Un pointeur se trouve sur la page la plus ancienne.

Quand un défaut de page survient, la page pointée est examinée. Si son bit *R* est à 0, la page est évincée, la nouvelle page est insérée dans l'horloge à sa place, et le pointeur est avancé d'une position. Si le bit *R* est à 1, il est effacé et le pointeur est avancé sur la prochaine page. Ce processus est répété jusqu'à ce qu'une page avec R = 0 soit trouvée. C'est l'algorithme de l'**horloge**.

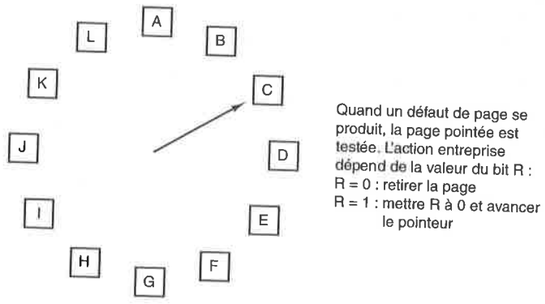

Figure 3.16 • L'algorithme de remplacement de page de l'horloge.

3.4.6 L'algorithme de remplacement de la page la moins récemment utilisée (LRU)

Pour s'approcher au mieux de l'algorithme optimal, il faut se fonder sur l'observation suivante : les pages les plus référencées lors des dernières instructions seront probablement très utilisées dans les prochaines instructions. À l'inverse, des pages qui n'ont pas été employées depuis longtemps ne seront pas demandées avant un long moment. On peut en déduire l'algorithme suivant : quand un défaut de page se produit, c'est la page qui n'a pas été utilisée pendant le plus de temps qui est retirée. Cette méthode est appelée pagination **LRU** (*Least Recently Used*, la moins récemment utilisée).

Cet algorithme LRU est réalisable mais reste coûteux. Pour l'implanter totalement, il est nécessaire de gérer une liste chaînée de toutes les pages en mémoire, avec la page la plus récemment utilisée en tête et la moins utilisée en queue. La difficulté est que cette

liste doit être mise à jour à chaque référence mémoire. La recherche d'une page dans la liste, sa destruction et son déplacement en début de liste sont des opérations coûteuses en temps, même si elles sont réalisées de manière matérielle (en supposant qu'un tel matériel existe).

Cependant, il existe d'autres manières d'implanter cet algorithme avec des composants matériels particuliers. Considérons la plus simple d'entre elle. Cette méthode nécessite d'équiper le matériel avec un compteur 64 bits, C, qui s'incrémente automatiquement après chaque instruction. Par ailleurs, chaque entrée de la table des pages doit avoir un champ assez grand pour contenir ce compteur. Après chaque référence mémoire, la valeur courante de C est enregistrée dans l'entrée de la table des pages pour l'entrée qui vient d'être référencée. Quand un défaut de page se produit, le système d'exploitation examine tous les compteurs dans la table des pages afin de trouver le plus petit d'entre eux. Celui-ci correspond à la page la moins récemment utilisée.

Examinons à présent une deuxième solution matérielle. Pour une machine à *n* cadres, le matériel doit gérer une matrice de $n \times n$ bits, initialement tous nuls. Quand une page *k* est référencée, le matériel commence par mettre à 1 tous les bits de la rangée *k* et tous les bits de la colonne *k* à 0. À chaque instant, la rangée dont la valeur binaire est la plus petite indique la page la moins récemment utilisée.

Le fonctionnement de cet algorithme est donné à la figure 3.17 pour quatre cadres de pages et des références de pages dans l'ordre suivant :

0 1 2 3 2 1 0 3 2 3

Après que la page 0 est référencée, nous avons la situation de la figure 3.17(a). Après que la page la page 1 est référencée, nous avons la situation de la figure 3.17(b), et ainsi de suite.

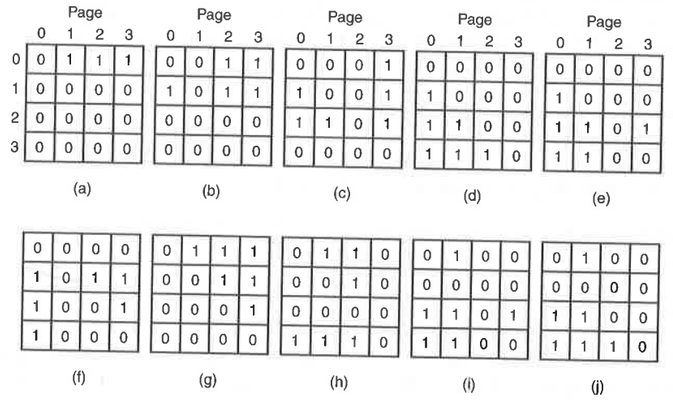

Figure 3.17 • L'algorithme LRU utilisant une matrice dont les pages sont référencées dans l'ordre 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

3.4.7 La simulation logicielle de l'algorithme LRU

Bien que les deux algorithmes LRU précédents soient réalisables, peu de machines disposent du matériel adéquat. À la place, une solution logicielle peut être implantée. Une des méthodes possibles est l'algorithme **NFU** (*Not Frequently Used* , peu utilisé). Il nécessite un compteur logiciel par page, initialement à 0. À chaque interruption d'horloge, le système d'exploitation examine toutes les pages en mémoire. Pour chaque page, le bit *R*, de valeur 0 ou 1, est ajouté à son compteur. Les compteurs mémorisent donc *grosso modo* les différents référencements des pages. Quand un défaut de page se produit, c'est la page avec le plus petit compteur qui est remplacée.

Le problème principal de l'algorithme NFU est qu'il n'oublie rien. Par exemple, dans un compilateur à plusieurs passes, les compteurs des pages très utilisées au cours de la première passe ont des valeurs élevées lors des passes suivantes. Si la première passe a le plus long temps d'exécution, les pages contenant le code des passes suivantes ont toujours un compteur plus faible que les pages de la première passe. Le système d'exploitation aura donc tendance à retirer des pages utiles plutôt que des pages non utilisées.

Il existe néanmoins une petite modification que l'on peut apporter à l'algorithme NFU pour lui permettre de se comporter comme l'algorithme LRU. Cette modification se réalise en deux temps. Tout d'abord, les compteurs sont décalés d'un bit à droite avant d'être ajoutés au bit *R*. Ensuite, le bit *R* est ajouté au bit de poids le plus fort (le bit de gauche) plutôt qu'au bit de poids le plus faible (celui de droite).

La figure 3.18 montre le fonctionnement de cet algorithme modifié, appelé **algorithme de vieillissement** (*aging*). Supposons qu'après le premier top d'horloge, les bits *R* des pages 0 à 5 aient les valeurs 1, 0, 1, 0, 1 et 1 respectivement. Autrement dit, entre les tops d'horloge 0 et 1, les pages 0, 2, 4 et 5 sont référencées et leurs bits *R* sont mis à 1, alors que les autres restent à 0. Après que les 6 compteurs correspondants ont été décalés et que les bits *R* ont été ajoutés à leur poids fort, nous obtenons les valeurs de la figure 3.18(a). Les quatre colonnes suivantes montrent les valeurs des compteurs après les quatre tops d'horloge suivants.

Quand un défaut de page se produit, la page dont le compteur est le plus petit est retirée. Il est évident qu'une page qui n'a pas été référencée pendant 4 tops d'horloge a quatre 0 dans son compteur : de ce fait, elle aura une plus petite valeur que le compteur d'une page non référencée pendant 3 tops d'horloge.

Cet algorithme diffère de l'algorithme LRU sur deux points. Considérons les pages 3 et 5 de la figure 3.18(e). Ni l'une ni l'autre n'ont été référencées depuis 2 tops d'horloge ; elles l'ont été toutes les deux lors du top précédent. Selon l'algorithme LRU, si une page doit être remplacée, c'est l'une des deux qui sera choisie. Le problème est que nous ne savons pas laquelle a été référencée en dernier entre les tops d'horloge 1 et 2. En ne mémorisant qu'un bit par intervalle de temps, nous ne pouvons pas distinguer les références faites au début ou à la fin de cet intervalle. Notre seule option est de retirer la page 3 parce que la page 5 a aussi été référencée 2 tops d'horloge plus tôt, ce qui n'est pas le cas de la page 3.

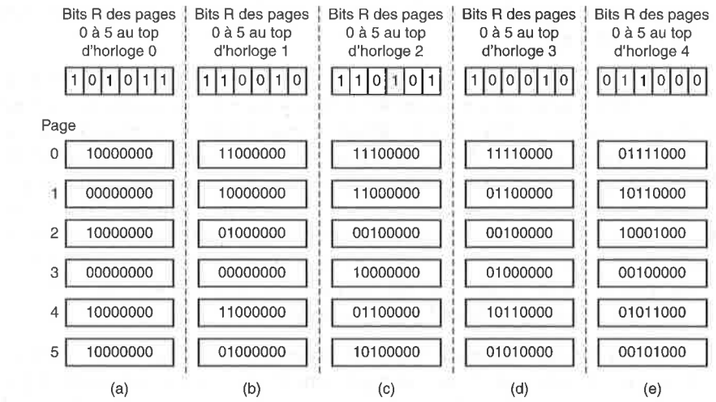

Figure 3.18 • L'algorithme du vieillissement simule de façon logicielle l'algorithme LRU. Six pages sont montrées pour cinq tops d'horloge, de (a) à (e).

La seconde différence entre ces deux algorithmes est que le compteur de l'algorithme du vieillissement a un nombre de bits fini, 8 dans notre exemple. Supposons que nous ayons 2 pages dont le compteur vaut 0. Nous devons en choisir une au hasard. En fait, il est possible que l'une des deux ait été référencée 9 tops d'horloge plus tôt, et que l'autre l'ait été 1 000 tops plus tôt. Nous n'avons aucun moyen de le savoir. Dans la pratique, un compteur 8 bits est suffisant pour des tops d'horloge qui se produisent toutes les 20 ms. Si une page n'a pas été référencée depuis 160 ms, il est probable qu'elle n'est pas très utilisée.

3.4.8 L'algorithme de remplacement de page « ensemble de travail »

Dans un système paginé, les processus sont lancés sans qu'aucune de leurs pages soit en mémoire. Dès que le processeur essaie d'exécuter la première instruction, il se produit un défaut de page, ce qui amène le système d'exploitation à charger la page contenant cette instruction. D'autres défauts de pages vont apparaître rapidement pour des variables globales ou la pile. Au bout d'un certain temps, le processus dispose de la plupart des pages dont il a besoin et son exécution ne générera que peu de défauts de pages. Cette méthode est appelée **pagination à la demande** (*demand paging*) parce que les pages sont chargées uniquement à la demande et non à l'avance.

Naturellement, il est facile d'écrire un programme de test qui lit systématiquement toutes les pages et provoque tellement de défauts de pages que la mémoire n'est pas assez grande pour les charger toutes. Heureusement, la plupart des processus ne travaillent pas de cette manière. Ils font plutôt des références groupées, ce qui signifie que pendant chaque phase d'exécution, les processus référencent uniquement un

nombre restreint de pages. Par exemple, chaque passe d'un compilateur référence une partie différente des pages.

L'ensemble des pages exploitées par le processus courant est son **ensemble de travail** (*working set*). Si la totalité de son ensemble de travail est présente en mémoire, l'exécution de ce processus se déroule sans défaut de page jusqu'à son passage dans un autre état (par exemple, la prochaine passe pour un compilateur). Si la mémoire disponible est trop petite pour charger l'ensemble de travail dans son intégralité, le processus génère de nombreux défauts de pages et s'exécute plus lentement puisqu'une instruction dure quelques nanosecondes alors que le chargement d'une page en mémoire prend 10 ms. Au rythme d'une ou deux instructions toutes les 10 ms, le processus n'est pas prêt de se terminer. Un programme qui génère de nombreux défauts de pages provoque un écroulement du système.

Dans un système de multiprogrammation, les processus sont fréquemment déplacés vers le disque (c'est-à-dire que toutes les pages sont ôtées de la mémoire) afin de permettre à d'autres processus d'avoir recours au processeur. La question qui se pose est la suivante : que se passe-t-il lorsqu'un processus est rechargé en mémoire ? Théoriquement, il n'y a rien à faire. Le processus provoquera des défauts de pages jusqu'au chargement de son ensemble de travail. Le problème est qu'on ne peut accepter 20, 100 ou même 1 000 défauts de pages chaque fois qu'un processus est rechargé en mémoire. Cela serait très lent et gaspillerait énormément de temps processeur puisque le système d'exploitation n'a besoin que de quelques millisecondes pour gérer un défaut de page.

Par conséquent, de nombreux systèmes d'exploitation mémorisent l'ensemble de travail de chaque processus et le charge en mémoire avant d'exécuter les processus. Cette approche est appelée **modèle de l'ensemble de travail** (*working set model*). Elle a été conçue pour réduire le nombre de défauts de pages. Le chargement des pages avant l'exécution est appelé **préchargement** (*prepaging*). Notons que cet ensemble de travail évolue constamment.

Nous savons depuis longtemps que la plupart des programmes ne référencent pas leur espace d'adressage uniformément, mais que les références se font sur un petit nombre de pages. Une référence mémoire peut lire une instruction, une donnée ou enregistrer une information. À chaque instant t, il existe un ensemble constitué de toutes les pages utilisées par les k références mémoire les plus récentes. Cet ensemble, $w(k,t)$, est l'ensemble de travail. La fonction $w(k,t)$ est une fonction monotone croissante. Sa limite en fonction de k est finie parce qu'un programme ne peut pas référencer plus de pages que son espace ne peut en contenir et que peu de programmes auront recours à chaque page. La figure 3.19 décrit la taille de l'ensemble de travail en fonction des k références mémoire les plus récentes.

Le fait que la plupart des programmes accèdent aléatoirement à un petit nombre de pages et que cet ensemble se modifie lentement dans le temps explique la rapide croissance de la courbe à son début, et une croissance lente pour un k plus grand. Par exemple, un programme exécutant une boucle qui occupe 2 pages et utilise 4 pages de données référencera les 6 pages toutes les 1 000 instructions, mais la plus récente référence à une autre page peut avoir été faite un million d'instructions plus tôt,

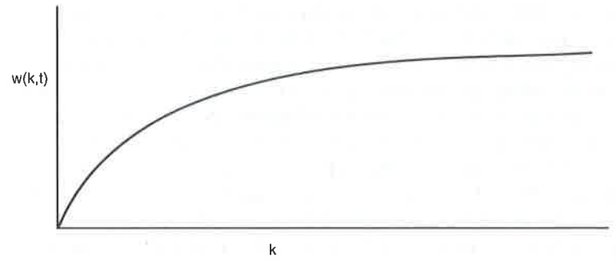

Figure 3.19 • L'ensemble de travail est constitué des pages utilisées par les k références mémoire les plus récentes. La fonction $w(k,t)$ représente la taille de l'ensemble de travail à un instant t.

pendant la phase d'initialisation. Suite à ce comportement asymptotique, le contenu de l'ensemble de travail n'est pas sensible aux valeurs de k choisies. Autrement dit, il existe un intervalle de valeurs de k pour lesquelles l'ensemble de travail est inchangé. Parce que l'ensemble de travail varie lentement avec le temps, nous pouvons faire une hypothèse raisonnable sur les pages nécessaires lors du redémarrage d'un programme stoppé précédemment. Ce préchargement correspond au chargement des pages avant que le processus ne reprenne son exécution.

Pour implanter ce modèle de l'ensemble de travail, il est nécessaire que le système d'exploitation mémorise quelles sont les pages contenues dans celui-ci. Le fait de posséder cette information mène immédiatement à un possible algorithme de remplacement de page : quand un défaut de page se produit, il est facile de trouver une page absente de l'ensemble de travail et de l'évincer. Pour implanter un tel algorithme, nous avons besoin de pouvoir déterminer précisément quelles sont les pages présentes, ou absentes, dans l'ensemble de travail à un instant donné. Par définition, l'ensemble de travail est constitué des pages utilisées dans les k références mémoire les plus récentes (certains auteurs préfèrent les k références de pages les plus récentes, mais le choix est arbitraire). Pour implanter un algorithme de l'ensemble de travail, il faut choisir à l'avance quelques valeurs de k. Lorsque cela est fait, après chaque référence mémoire, l'ensemble des pages employées par les k références mémoire précédentes sont déterminées de manière unique.

Naturellement, le fait d'avoir une définition de l'ensemble de travail n'implique pas qu'il existe un moyen efficace de le gérer en temps réel pendant l'exécution des programmes. Nous pourrions imaginer un registre à décalage de longueur k, qui se décale d'une position à gauche à chaque référence mémoire et insère le numéro de la page la plus récemment référencée à droite. L'ensemble des k numéros de pages du registre à décalage serait cet ensemble de travail. En théorie, pour un défaut de page, le contenu du registre serait lu et trié. Les pages dupliquées seraient ainsi effacées. Le résultat serait donc l'ensemble de travail. Cependant, comme la gestion de ce registre à décalage et l'exécution d'un tel algorithme lors d'un défaut de page seraient bien trop coûteuses, cette solution n'est jamais exploitée.

À la place, on utilise quelques approximations. L'une d'elles consiste à exploiter le temps d'exécution au lieu de compter les k dernières références mémoire. Par exemple, au lieu de définir l'ensemble de travail comme étant les pages utilisées durant les 10 millions de références mémoire précédentes, nous pouvons le définir comme l'ensemble des pages utilisées durant les 100 dernières millisecondes du temps d'exécution. En pratique, une telle définition est une bonne approximation et est simple à exploiter. Il faut noter que pour chaque processus, seul son propre temps d'exécution est comptabilisé. De cette manière, si un processus démarre son exécution au temps `T` et qu'il ait `40` ms de temps processeur dans l'intervalle `T + 100` ms pour s'exécuter, son temps est de `40` ms. Le temps UC qu'un processus a utilisé depuis son démarrage est souvent appelé **temps virtuel courant** (*current virtual time*). Avec cette approximation, l'ensemble de travail d'un processus est constitué de l'ensemble des pages référencées durant les τ dernières secondes du temps virtuel.

Examinons à présent un algorithme de remplacement de page reposant sur un ensemble de travail. L'idée fondamentale est de trouver une page absente de l'ensemble de travail et de l'expulser. À la figure 3.20, nous voyons une partie d'une table des pages d'une machine. Puisque seules les pages présentes en mémoire sont considérées comme des candidates à l'éviction, les pages absentes de la mémoire sont ignorées par cet algorithme. Chaque entrée contient au moins deux types d'information : le temps approximatif de la dernière utilisation de la page et le bit R (page référencée). Le rectangle vide représente d'autres champs non nécessaires à cet algorithme, tels que le numéro du cadre de page, les bits de protection et le bit M (page modifiée).

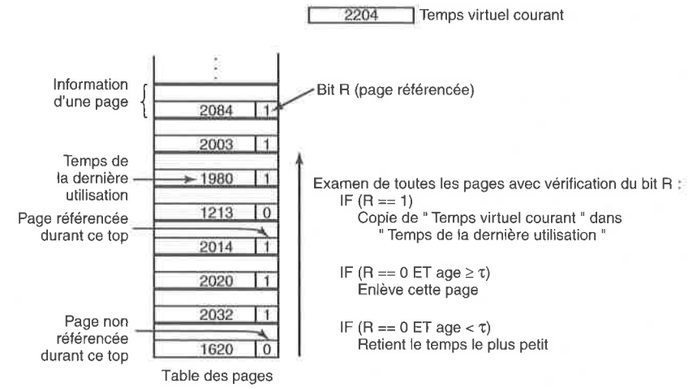

Figure 3.20 • L'algorithme de l'ensemble de travail.

L'algorithme fonctionne de la manière suivante. Le matériel est supposé initialiser les bits R et M, comme nous l'avons vu précédemment. De la même manière, une interruption périodique d'horloge doit générer un programme qui efface le bit R à chaque

interruption. Pour chaque défaut de page, la table des pages est parcourue à la recherche d'une page qui puisse être évincée.

Pour chacune de ces entrées, le bit R est examiné. S'il est égal à 1, la valeur du `Temps virtuel courant` est copiée dans le champ `Temps de la dernière utilisation` dans la table des pages, indiquant que la page était en cours d'utilisation lorsque le défaut de page s'est produit. Puisque la page a été référencée pendant le top courant d'horloge, elle se trouve clairement dans l'ensemble de travail et n'est pas candidate à l'éviction (τ est supposé couvrir de multiples tops d'horloge).

Si R est égal à 0, la page n'a pas été référencée durant le top d'horloge courant et peut être candidate à la suppression. Pour voir si elle doit l'être ou pas, son âge, c'est-à-dire le `Temps virtuel courant` auquel est soustrait son `Temps de dernière utilisation`, est comparé à τ. Si l'âge est supérieur à `?`, la page ne reste pas plus longtemps dans l'ensemble de travail et la nouvelle page la remplace. L'examen continue cependant pour mettre à jour les entrées restantes.

Néanmoins, si le bit R est à 0 et que l'âge soit inférieur ou égal à τ, la page reste dans l'ensemble de travail. La page est momentanément épargnée, mais la page dont l'âge est le plus grand (c'est-à-dire dont le champ `Temps de la dernière utilisation` a la plus petite valeur) est notée. Si toute la table est examinée sans qu'une page candidate à l'éviction soit trouvée, cela signifie que toutes les pages sont dans l'ensemble des pages. Dans ce cas, si l'on trouve une ou plusieurs pages avec le bit R à 0, c'est celle qui a le plus grand âge qui est supprimée. Dans le pire des cas, toutes les pages ont été référencées pendant le top d'horloge courant (et ont toutes leur bit R à 1). L'une d'elles est donc choisie aléatoirement pour être évincée.

3.4.9 L'algorithme de remplacement de page WSClock

L'algorithme de base de l'ensemble de travail est lourd puisque toute la table des pages doit être examinée à chaque défaut de page jusqu'à ce qu'une page adéquate soit localisée. Il existe un meilleur algorithme, appelé **WSClock**, qui utilise l'algorithme de l'horloge et les informations de l'ensemble de travail. Grâce à sa simplicité d'implantation et à ses bonnes performances, il est très employé dans la pratique.

La structure de données nécessaire est une liste circulaire des cadres de pages, comme dans l'algorithme de l'horloge, et tel que décrit à la figure 3.21(a). Initialement, cette liste est vide. Lorsque la première page est chargée, elle est ajoutée à la liste. Au fur et à mesure que des pages sont ajoutées, elles entrent dans la liste et forment un anneau. Chaque entrée contient le champ `Temps de la dernière utilisation` de l'algorithme de base de l'ensemble de travail, en plus du bit R (montré) et du bit M (non montré).

Comme pour l'algorithme de l'horloge, à chaque défaut de page, la page pointée est examinée en premier. Si le bit R est mis à 1, cela signifie que la page a été utilisée pendant le top courant ; il ne convient donc de l'effacer. Le bit R est mis à 0, le pointeur est avancé sur la page suivante et l'algorithme se répète sur cette page. L'état qui intervient après cette séquence d'événements est illustré à la figure 3.21(b).