- Vide.
- Occupée par les filles.
- Occupée par les garçons.

Utilisez votre langage de programmation favori pour développer les procédures suivantes : `fille_veut_entrer`, `garçon_veut_entrer`, `fille_quitte`, `garçon_quitte`. Vous avez le choix des compteurs et des techniques de synchronisation.

52. Réécrivez le programme de la figure 2.23 afin de prendre en charge plus de deux processus.

53. Écrivez un problème du type producteur-consommateur utilisant des threads et partageant un tampon commun. En revanche, n'utilisez ni sémaphores ni tout autre technique de synchronisation pour protéger les structures de données partagées. Laissez simplement chaque thread y accéder lorsqu'il le souhaite. Utilisez `sleep` et `wakeup` pour gérer les conditions `full` et `empty`. Voyez combien de temps va s'écouler avant qu'une condition de concurrence fatale ne se produise. Par exemple, vous pouvez programmer un producteur qui affiche un nombre de temps en temps. N'affichez pas plus d'un nombre par minute, sinon les E/S pourraient avoir une incidence sur les conditions de concurrence.

# 3

# La gestion de la mémoire

La mémoire principale (RAM) est une ressource importante qui doit être gérée avec attention. Tandis qu'aujourd'hui la quantité moyenne de mémoire d'un ordinateur personnel est mille fois plus importante que celle de l'IBM 7094, le plus grand ordinateur du monde au début des années 1960, la taille des programmes s'accroît aussi vite que celle des mémoires. Si l'on paraphrase la loi de Parkinson, « les programmes s'accroissent pour remplir la mémoire disponible qui leur est réservée ». Dans ce chapitre, nous étudierons comment les systèmes d'exploitation gèrent la mémoire.

Dans l'absolu, les programmeurs aimeraient disposer d'une mémoire infiniment grande, infiniment rapide et qui serait aussi non volatile, c'est-à-dire qui ne perdrait pas son contenu quand l'électricité est coupée. Tant que nous y sommes, pourquoi ne pas demander aussi qu'elle soit bon marché ? Malheureusement, la technologie ne permet pas actuellement de fabriquer de telles mémoires. C'est pourquoi, la plupart des ordinateurs disposent d'une **hiérarchisation de la mémoire** : quelques mégaoctets de mémoire cache volatile, rapide et chère, quelques gigaoctets de mémoire de rapidité d'accès et de prix moyens, et quelques téraoctets de mémoire plus lente et non volatile servant de mémoire de masse. Le système d'exploitation a pour rôle de coordonner la manière dont ces différentes mémoires sont utilisées.

L'entité du système d'exploitation qui gère la hiérarchie de la mémoire est appelée le **gestionnaire de mémoire**. Son rôle est de garder trace de la partie de la mémoire qui est en cours d'utilisation et de celle qui ne l'est pas, d'allouer cette mémoire aux processus qui en ont besoin et de la libérer quand ils ont fait leur travail.

Dans ce chapitre, nous étudierons un certain nombre de gestionnaires de mémoire, du plus simple au plus sophistiqué. La gestion du plus bas niveau de mémoire (le cache) étant en principe réalisée par le matériel et le stockage permanent (sur disque) étant traité au chapitre suivant, nous nous attacherons ici à décrire la gestion de la mémoire principale telle que peut la voir le programmeur.

## 3.1 Abstraction de la mémoire ?

L'abstraction la plus simple consiste à ne pas faire d'abstraction du tout. Les ordinateurs d'avant 1960, les mini-ordinateurs d'avant 1970 et les micro-ordinateurs d'avant 1980 ne disposaient pas d'abstraction de la mémoire. Chaque programme voyait tout simplement la mémoire physique. Face à une instruction comme

```
MOV REGISTER1, 1000
```

le système faisait passer le contenu de l'emplacement mémoire 1000 dans le registre 1. Le modèle de mémoire présenté au programmeur correspondait tout simplement à la mémoire physique : un ensemble d'adresses numérotées de 0 à une valeur maximale, chaque adresse désignant une cellule contenant un certain nombre de bits (huit en général).

Il n'était pas possible d'avoir deux programmes s'exécutant en même temps. En effet, si un programme écrit une certaine valeur à l'emplacement 2000, par exemple, on ne peut admettre qu'un autre vienne modifier cette valeur à sa guise. En fait, dans ce cas, les deux programmes se planteraient immédiatement.

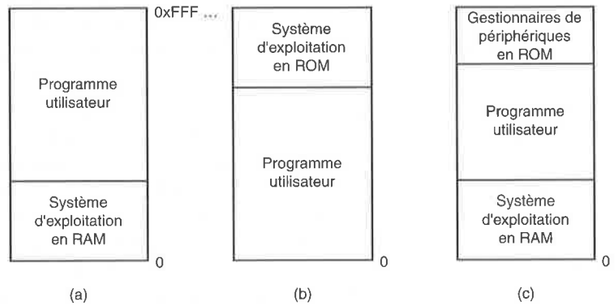


**Figure 3.1** • Trois possibilités simples d'organiser la mémoire avec un système d'exploitation et un seul processus utilisateur.

Mais même si le modèle ne reflète que la mémoire physique, il peut exister plusieurs variantes comme illustré à la figure 3.1. Le système d'exploitation peut se trouver en bas de la mémoire vive (RAM), comme illustré à la figure 3.1(a), ou bien être en mémoire morte (ROM) en haut de la mémoire, comme à la figure 3.1(b) ; les gestionnaires de périphériques peuvent aussi se trouver en haut de la mémoire en ROM tandis que le reste du système en RAM se situe tout en bas de la mémoire, comme à la figure 3.1(c). Autrefois utilisé dans les mainframes et les mini-ordinateurs, le premier modèle est aujourd'hui rare. Le deuxième modèle équipe de nombreux ordinateurs de poche et des systèmes embarqués. Enfin, le troisième modèle était exploité sur les premiers ordinateurs personnels (ceux qui faisaient fonctionner MS-DOS), dans lesquels la partie du système d'exploitation en ROM s'appelait le **BIOS**. Les modèles (a) et (c) présentent un inconvénient majeur : un bogue dans un programme utilisateur peut engendrer des conséquences désastreuses sur le système d'exploitation.



Quand le système est organisé de cette manière, un seul processus peut s'exécuter à la fois. Dès que l'utilisateur tape une commande, le système d'exploitation copie le programme demandé depuis le disque vers la mémoire et l'exécute. Lorsque le processus se termine, le système d'exploitation affiche une suite de caractères particulière, appelé invite (*prompt*), et attend une nouvelle commande. Quand il reçoit la commande, il charge un nouveau programme en mémoire, en écrasant le précédent.

On peut introduire un peu de parallélisme dans ces systèmes en utilisant les threads. Puisque tous les threads d'un processus ont la même image mémoire, ce n'est pas un problème qu'ils y soient contraints. Si, en théorie, cette idée fonctionne, elle est en réalité d'un intérêt limité : ce que veulent les gens c'est exécuter simultanément des programmes *indépendants*, ce qui ne correspond pas à l'idée des threads. De plus, si le système est si rudimentaire qu'il ne fournit pas d'abstraction mémoire, il y a peu de chances qu'on dispose d'une gestion de threads.

### L'exécution de plusieurs programmes sans abstraction de mémoire

Même sans abstraction de la mémoire, il est cependant possible d'exécuter quasi simultanément plusieurs programmes. Le système d'exploitation doit alors recopier sur disque l'intégralité du contenu de la mémoire avant de charger et d'exécuter le programme suivant. Tant qu'à un instant donné un seul programme est en mémoire, il ne peut y avoir de conflit. Cette idée (va-et-vient ou *swapping*) est présentée un peu plus loin.

En ajoutant un peu de matériel spécialisé, on peut aller plus loin. Sur les premiers modèles d'IBM 360, la mémoire était divisée en blocs de 2 Ko à chacun desquels était affectée une clé de protection de 4 bits contenue dans des registres spéciaux de l'UC. Pour une machine dotée de 1 Mo, il fallait 512 registres de 4 bits et donc 256 octets de stockage. Le mot d'état du programme contenait lui aussi cette clé et, dès qu'un processus cherchait à accéder à la mémoire avec une clé d'accès différente, le matériel du 360 provoquait un déroutement. Comme seul le système d'exploitation pouvait changer les clés de protection, les utilisateurs ne risquaient pas d'interférer les uns avec les autres ou avec le SE.

Cette solution présente néanmoins un inconvénient illustré à la figure 3.2. Nous avons deux programmes, chacun d'eux de 16 Ko, comme indiqué aux figures 3.2(a) et 3.2(b). Le grisé indique que la première zone mémoire a une clé différente de la seconde. Le premier programme démarre avec un saut à l'adresse 24 qui contient une instruction `MOV`. Le second fait un saut à l'adresse 28 qui contient une instruction de comparaison `CMP`. Les instructions qui ne nous concernent pas ici ne sont pas indiquées. Lorsque les deux programmes sont chargés consécutivement en mémoire en partant de l'adresse 0, nous obtenons la situation de la figure 3.2(c). Le SE, quant à lui, est quelque part ailleurs, probablement dans les adresses hautes de la mémoire.

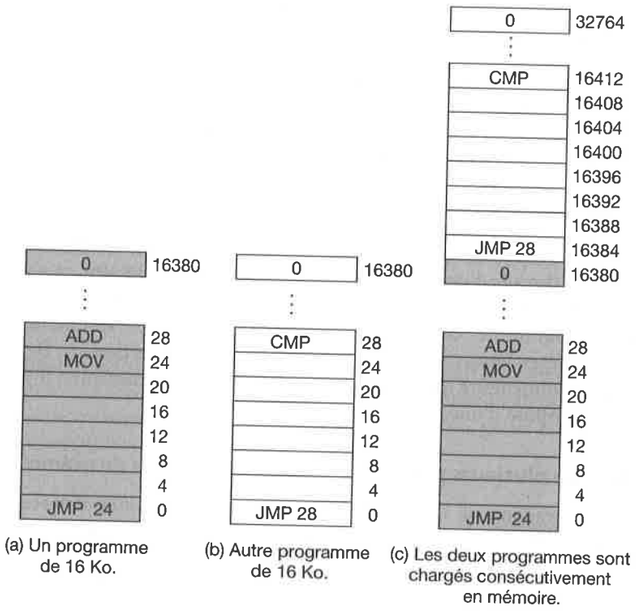


**Figure 3.2** • Le problème de la réallocation. (a) Un programme de 16 Ko. (b) Autre programme de 16 Ko. (c) Les deux programmes sont chargés consécutivement en mémoire.

Une fois que les programmes ont été chargés, on peut passer à l'exécution. Puisqu'ils ont des clés différentes, les deux programmes ne risquent pas de se gêner mais un problème va tout de même se poser. Lorsque le premier programme s'exécute, il fait le saut à l'adresse 24 où il trouve l'instruction attendue. Ce programme fonctionne donc normalement.

Mais après quelques instants, voici que le SE décide de passer au second programme, celui qui a été chargé juste derrière le premier à l'adresse 16 384. La première instruction est un saut à l'adresse 28, ce qui nous conduit à exécuter l'instruction `ADD` du premier programme et non le `CMP` qu'on voulait faire. Le programme va donc probablement se planter très vite.

Le problème vient du fait que les deux programmes référencent la mémoire physique : ce n'est pas ce que nous voulons. Nous voulons que chaque programme référence son propre ensemble d'adresses. L'IBM 360 utilisait pour résoudre ce problème une technique appelée **réallocation statique** qui consistait à modifier au vol, pendant le chargement en mémoire, toutes les adresses manipulées par le programme en leur ajoutant l'adresse à partir de laquelle il est stocké. C'est une solution peu générale, qui ralentit le chargement mais elle marche bien. Elle nécessite néanmoins



de l'information supplémentaire dans les programmes pour distinguer une adresse réallouable (à modifier) d'une constante numérique (qui doit rester telle quelle) comme dans

```
MOV REGISTER1,28
```

qui place le nombre 28 dans le registre 1.

L'adressage direct est de l'histoire ancienne pour les mainframes, les PC et les assistants personnels, mais il est encore d'actualité pour les systèmes embarqués et les cartes à puce. Les machines à laver, les postes de radio et les fours à micro-ondes contiennent du logiciel (en ROM) qui, dans la plupart des cas, utilisent de l'adressage direct. Cela fonctionne, car le comportement des programmes est complètement défini, tout en interdisant aux utilisateurs de faire tourner leurs propres programmes sur leur grille-pain.

Les systèmes embarqués les plus élaborés (comme ceux des téléphones mobiles) disposent de SE complexes mais d'autres peuvent n'avoir, en guise de système d'exploitation, qu'une bibliothèque liée au programme d'application et qui gère les appels systèmes destinés aux E/S. C'est le cas de **e-cos**, un système assez connu.

## 3.2 Une abstraction de la mémoire : les espaces d'adressage

Finalement, faire voir aux processus la mémoire physique présente plusieurs inconvénients. D'abord, si les programmes de l'utilisateur peuvent adresser chaque octet de la mémoire, on prend le risque d'un plantage général (sauf mise en place d'un matériel spécial comme avec l'IBM 360). Ce problème existe même avec un seul programme qui s'exécute. Ensuite, si l'on veut exécuter plusieurs programmes (chacun à leur tour puisqu'il n'y a qu'une UC), ce n'est pas très facile. Sur les PC, il est courant que plusieurs programmes soient ouverts en même temps (traitement de texte, courrier électronique, navigateur Web…) : un seul programme est actif à un instant donné mais les autres peuvent être réactivés d'un clic. Sans abstraction de la mémoire, on voit mal comment faire.

### 3.2.1 La notion d'espace d'adressage

Pour avoir plusieurs applications en mémoire simultanément, il faut savoir résoudre les problèmes de protection et de réallocation. Nous avons vu la solution proposée sur l'IBM 360, qui est une solution lente et complexe.

Il vaut mieux créer une nouvelle abstraction de la mémoire : l'espace d'adressage. De la même façon que le concept de processus crée une sorte d'UC abstraite exécutant un programme, l'**espace d'adressage** est une sorte de mémoire abstraite d'un programme prêt à l'exécution. C'est l'ensemble des adresses qu'un processus peut utiliser pour adresser la mémoire. Chaque processus a son propre espace d'adressage, indépendant de celui des autres (sauf cas où l'on souhaite partager la mémoire).

En fait, ce concept d'espace d'adressage est très général. Considérons les numéros de téléphone. En France, les numéros de téléphone étaient composés de huit chiffres jusqu'en 1996 avec un espace d'adressage de 00000000 à 99999999 (certaines combinaisons étant réservées). Avec l'expansion du nombre d'objets communicants (et surtout des téléphones mobiles), il a fallu passer à 10 chiffres. L'espace d'adressage pour les ports d'E/S du Pentium va de 0 à 16 383. Le protocole IPv4 dispose d'adresses sur 32 bits, donc son espace d'adressage va de 0 à $2^{32} - 1$ (avec, là encore, certaines combinaisons réservées).

Les espaces d'adressage ne sont pas forcément numériques. L'ensemble des domaines *.com* de l'internet est également un espace d'adressage qui comprend toutes les combinaisons de 2 à 63 caractères (lettres, chiffres, tirets) suivies de *.com*. Finalement, un espace d'adressage est quelque chose d'assez simple.

Il est plus difficile de donner à chaque programme son propre espace d'adressage, c'est-à-dire de faire en sorte que l'adresse 28 de l'un corresponde à un emplacement différent de l'adresse 28 de l'autre. Il existe une solution assez simple, qui a été fort utilisée mais est tombée en désuétude en raison des nouvelles capacités des UC modernes.

### Les registres de base et de limite

La solution la plus simple consiste à faire de la **réallocation dynamique** en mappant l'espace d'adressage de chaque processus sur une partie différente de la mémoire physique. Cette solution classique, qui a été utilisée sur des machines allant du CDC 6600 (le premier super-ordinateur) à l'Intel 8088 (le microprocesseur qui équipait les premiers PC d'IBM), consiste à équiper l'UC de deux registres matériels : les registres de **base** et de **limite**. Les programmes sont alors chargés en mémoire sans réallocation, là où il y a de la place pour les ranger dans des mots mémoire consécutifs [voir figure 3.2(c)]. À l'exécution du programme, le SE range dans le registre de base l'adresse physique de début de programme et dans le registre de limite celle de fin. À la figure 3.2(c), les valeurs de base et de limite pour le premier programme sont 0 et 16 384, alors que celles du second sont 16 384 et 32 768. Si on chargeait en mémoire un troisième programme de 16 Ko, les valeurs des registres passeraient à 32 768 et 16 384.

Chaque fois qu'un processus référence la mémoire, que ce soit pour charger une instruction ou bien lire ou écrire une donnée, le matériel de l'UC ajoute automatiquement à l'adresse engendrée par le programme la valeur du registre de base et vérifie que la nouvelle adresse ainsi créée reste bien inférieure ou égale à la valeur du registre de limite. Dans le cas de la première instruction du second programme de la figure 3.2(c), le processus exécute une instruction

```
JMP 28
```

mais le matériel la traite comme si c'était

```
JMP 16412
```



ce qui aboutit à l'instruction `CMP` désirée. Les valeurs des registres de base et de limite durant l'exécution du second programme de la figure 3.2(c) apparaissent à la figure 3.3.

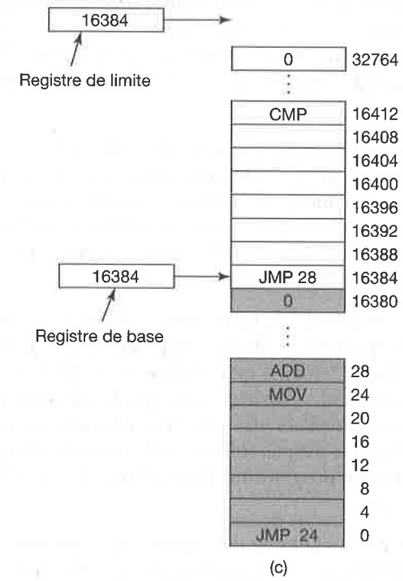


**Figure 3.3** • Registres de base et de limite utilisés pour attribuer à chaque processus un espace d'adressage.

Cette utilisation des registres de base et de limite est une façon simple d'attribuer à chaque processus son espace d'adressage. Dans nombre d'implémentations (comme celle du CDC 6600), ces registres ne sont accessibles qu'au SE. Le 8088, cependant, ne disposait pas de cette sécurité (il n'avait d'ailleurs même pas de registre de limite) alors qu'il disposait de plusieurs registres de base, permettant ainsi de réallouer indépendamment code du programme et données.

L'inconvénient majeur de cette méthode est qu'elle exige une addition et une comparaison à chaque référence mémoire. Les comparaisons se font très rapidement, mais les additions sont lentes en raison du mécanisme de propagation de la retenue (à moins d'utiliser des circuits d'addition spéciaux).

## 3.2.2 Le va-et-vient

Si la mémoire physique de l'ordinateur est suffisamment grande pour contenir tous les processus, les méthodes vues jusqu'ici conviennent. Mais, en pratique, c'est rarement

le cas. Sur un système Linux ou Windows, quelque chose comme 40 à 60 processus sont lancés au démarrage de la machine. Par exemple, lorsqu'on installe une application sous Windows, elle s'arrange souvent pour faire en sorte qu'à chaque démarrage on vérifie automatiquement si une mise à jour est disponible. Ce genre de processus occupe facilement 5 à 10 Mo de mémoire. Sans parler de ceux qui vérifient l'arrivée du courrier, les connexions réseau entrantes, etc. Tout cela avant même qu'on ait lancé le moindre programme d'application. Un programme d'application, de nos jours, consomme facilement 50 à 200 Mo, si ce n'est plus. On se trouve donc très vite à court de mémoire.

Deux approches de gestion mémoire peuvent être utilisées. La stratégie la plus simple, appelée **va-et-vient** (*swapping*), consiste à considérer chaque processus dans son intégralité (exécution puis placement sur le disque). L'autre stratégie, appelée **mémoire virtuelle**, permet aux programmes de s'exécuter même quand ils sont partiellement en mémoire principale. Nous allons étudier le va-et-vient avant de détailler le principe de la mémoire virtuelle, à la section 3.3.

Le fonctionnement d'un système de va-et-vient est illustré à la figure 3.4. Au départ, seul le processus *A* est en mémoire. Ensuite, les processus *B* et *C* sont créés ou chargés depuis le disque. À la figure 3.4(d), *A* est transféré sur le disque. Ensuite, *D* arrive tandis que *B* s'en va. Finalement, *A* revient. Bien que *A* soit maintenant localisé différemment, les adresses contenues doivent être relocalisées, soit par un logiciel quand *A* est chargé depuis le disque, soit – c'est le cas le plus fréquent – par le matériel pendant l'exécution du programme. Les registres de base et de limite fonctionneraient bien dans ce cas.

Quand l'opération de va-et-vient crée de multiples trous dans la mémoire, il est possible de tous les recombiner en une seule zone plus grande en déplaçant tous les processus vers le bas de la mémoire aussi vite que possible. Cette technique, connue sous le nom de **compactage de mémoire**, n'est pas souvent mise en œuvre car elle requiert énormément de temps UC. Par exemple, sur une machine avec 1 Go de RAM qui peut copier 4 octets en 20 ns, environ 5 s sont nécessaires pour compacter toute la mémoire.

Un point mérite d'être souligné ici, concernant la manière dont la mémoire doit être allouée pour la création d'un processus ou lors de son va-et-vient sur le disque. Si des processus sont créés avec une taille fixe qui ne change jamais, l'allocation est alors simple : le système d'exploitation alloue exactement la mémoire nécessaire, pas plus, pas moins.

Mais lorsque les segments de données des processus doivent croître (par exemple par allocation dynamique de mémoire à partir du tas, comme de nombreux langages de programmation l'autorisent), un problème surgit toutes les fois qu'un processus essaye de s'accroître. Si un trou est adjacent au processus, ce trou peut lui être alloué et le processus est autorisé à s'étendre dedans. Par ailleurs, si le processus est adjacent à un autre processus, le processus croissant devra être déplacé dans un trou suffisamment grand pour lui. Enfin, si un processus ne peut pas croître en mémoire et que la zone de va-et-vient sur le disque soit pleine, le processus devra attendre ou être tué.



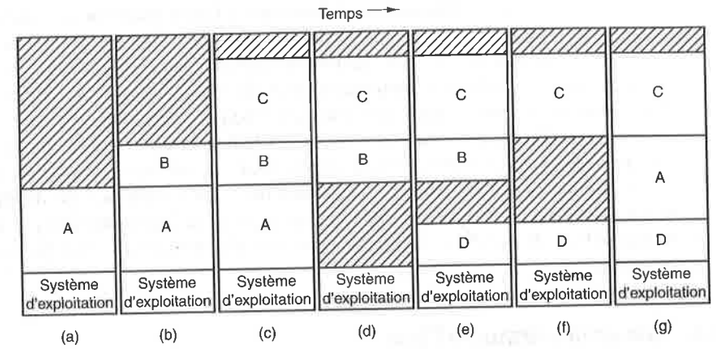


**Figure 3.4** • L'allocation mémoire change au gré des processus qui viennent en mémoire et qui la quittent. Les zones grisées indiquent que la mémoire est inutilisée.

S'il est à prévoir que la plupart des processus s'agrandiront lors de leur exécution, il est généralement bon d'allouer un peu de mémoire supplémentaire chaque fois qu'un processus est chargé ou déplacé. Cela permet de réduire le temps système associé aux processus déplacés ou chargés dont la taille n'est pas appropriée à la mémoire qui leur est allouée. Cependant, lorsque l'on transfère des processus sur le disque, seule la mémoire véritablement utilisée doit être recopiée ; il est inutile de recopier aussi la mémoire supplémentaire. La figure 3.5(a) présente une configuration mémoire dans laquelle un espace destiné à l'accroissement a été attribué à deux processus.




**Figure 3.5** • (a) Allocation d'espace pour l'accroissement d'un segment de données. (b) Allocation d'espace pour l'accroissement de la pile et d'un segment de données.

Prenons le cas de processus qui peuvent avoir deux zones d'accroissement : par exemple, un segment de données qui sert à des variables allouées et libérées dynamiquement, et un segment de pile destiné aux variables locales normales et aux adresses de retour. Une organisation différente s'impose alors d'elle-même, comme illustré à la figure 3.5(b). Dans cette figure, nous observons que chaque processus a une pile en haut de sa mémoire allouée qui peut s'accroître vers le bas, et une zone de données adjacente au programme qui peut s'accroître vers le haut. La mémoire comprise entre ces deux zones peut être utilisée par chacun des segments. Si la mémoire est insuffisante, chaque processus devra être déplacé dans un espace vide suffisamment grand, ou bien transféré hors de la mémoire jusqu'à la création d'une zone vide assez grande, ou bien encore détruit.

### 3.2.3 Gérer la mémoire libre

Quand la mémoire est attribuée dynamiquement, le système d'exploitation doit la gérer. Il y a deux façons de conserver une trace de l'utilisation de la mémoire : à l'aide des tables de bits et à l'aide des listes. Nous présentons le fonctionnement de ces deux méthodes ci-après.

#### Gérer la mémoire avec une table de bits

Avec une table de bits, la mémoire est répartie en unités d'allocation dont la taille peut varier de quelques mots à plusieurs kilo-octets. Chaque unité d'allocation correspond à un bit du tableau de bits, lequel est 0 si l'unité correspondante est vide et 1 si elle est occupée (ou *vice versa*). Nous présentons à la figure 3.6 une partie de mémoire et le tableau de bits qui lui est associé.

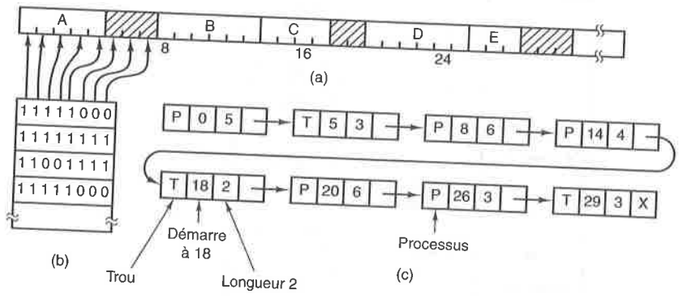


**Figure 3.6** • (a) Une partie de mémoire avec 5 processus et 3 trous. Les petites marques verticales indiquent les unités d'allocation mémoire. Les zones grisées (valeur 0 dans le tableau de bits) sont des zones libres. (b) Le tableau de bits correspondant. (c) Les mêmes informations sous forme d'une liste chaînée.

La taille de l'unité d'allocation est un élément fondamental dans la configuration : plus l'unité est petite, plus le tableau de bits est important. Toutefois, même avec une



petite unité de 4 octets, 32 bits de mémoire n'auront besoin que de 1 bit du tableau de bits. Une mémoire de $32n$ bits utilisera une table de $n$ bits, qui ne prendra pas plus de 1/33 de la mémoire. Lorsque l'unité d'allocation est plus grande, le tableau de bits est plus petit, mais une quantité non négligeable de mémoire peut être gaspillée dans la dernière unité allouée à un processus si la taille de celui-ci n'est pas un multiple exact de l'unité d'allocation.

Le tableau de bits offre un moyen simple de garder une trace des mots mémoire dans une quantité fixe de mémoire : en effet, la taille de la table dépend seulement de celle de la mémoire et de celle de l'unité d'allocation. Le tableau de bits présente cependant un inconvénient : lorsqu'un processus de $k$ unités est chargé en mémoire, le gestionnaire de mémoire doit parcourir le tableau de bits pour trouver une séquence de $k$ bits consécutifs dont la valeur est 0. Or cette recherche est une opération lente (parce que cette zone peut enjamber des mots frontières dans la table), et cela constitue un argument contre les tables de bits.

#### Gérer la mémoire avec des listes chaînées

Une autre manière de conserver une trace de la mémoire est de maintenir une liste chaînée des segments de mémoire alloués et libres ; dans cette liste, un segment est soit un processus, soit un trou entre deux processus. La mémoire de la figure 3.6(a) est représentée à la figure 3.6(c) comme une liste chaînée de segments. Chaque entrée de cette liste indique un trou (T) ou un processus (P), l'adresse à laquelle il débute et sa longueur, et elle spécifie un pointeur sur la prochaine entrée.

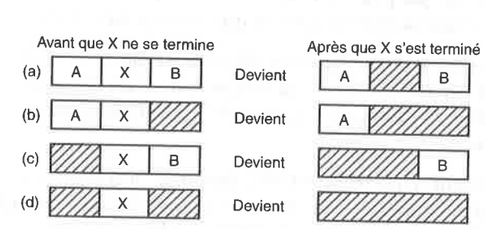


**Figure 3.7** • Quatre combinaisons de voisinage pour un processus *X* qui se termine.

Dans cet exemple, la liste des segments est triée par adresse, ce qui est pratique puisque lorsqu'un processus se termine ou est transféré sur disque, la mise à jour est directe. Un processus qui se termine a normalement deux voisins (excepté s'il est placé tout en haut ou tout en bas de la mémoire), qui peuvent être des processus ou des trous, ce qui conduit aux quatre combinaisons de la figure 3.7. Dans la figure 3.7(a), la mise à jour de la liste oblige à remplacer un P par un T. Dans les figures 3.7(b) et 3.7(c), deux entrées sont réunies en une seule, et la liste devient plus courte d'une unité. Dans la figure 3.7(d), trois entrées sont fusionnées et deux items sont retirés de la liste. Puisque la table des processus intègre les processus qui se terminent, elle doit normalement pointer sur l'entrée de la liste des processus eux-mêmes, il serait plus judicieux d'avoir une liste doublement chaînée, plutôt qu'une

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