le pourcentage de l'espace disque qui sera gaspillé ? Pour un système de fichiers réel, pensez-vous que le pourcentage de gaspillage sera supérieur ou inférieur à cette valeur ? Expliquez votre réponse.

30. La table FAT-16 de MS-DOS contient 64 K entrées. Supposons que l'un de ses bits ait été nécessaire pour autre chose et que la table possède exactement 32 768 entrées. Sans aucune autre modification, quelle serait la taille maximale d'un fichier MS-DOS dans ces conditions ?

31. Les fichiers MS-DOS sont en concurrence pour l'espace dans la table FAT-16 en mémoire. Si un fichier utilise *k* entrées, cela signifie que ces *k* entrées ne sont plus disponibles pour les autres fichiers : comment cela se traduit-il, en termes de contraintes, par rapport à la longueur totale de tous les fichiers combinés ?

32. Un système de fichiers UNIX a des blocs de 1 Ko et des adresses disque sur 4 octets. Quelle est la taille maximale d'un fichier si les i-nodes contiennent 10 entrées directes et une simple indirection, une double indirection et une triple indirection pour chaque fichier ?

33. Combien d'opérations disque sont nécessaires pour charger l'i-node du fichier */usr/ast/courses/os/handout.t* ? Supposez que l'i-node du répertoire racine se trouve en mémoire mais qu'aucun autre élément du chemin d'accès ne s'y trouve. Supposez aussi que tous les répertoires tiennent dans un bloc de disque.

34. Dans de nombreux systèmes UNIX, les i-nodes sont conservés au début du disque. Une alternative consiste à allouer un i-node quand un fichier est créé et à placer l'i-node au début du premier bloc du fichier. Dites les avantages et les inconvénients de cette méthode.

35. Écrivez un programme qui inverse les octets d'un fichier, de sorte que le dernier octet devienne le premier et que le premier devienne le dernier octet. Ce programme doit pouvoir travailler sur des fichiers de longueur quelconque, mais doit être efficace.

36. Écrivez un programme qui démarre dans un répertoire donné et parcourt son arborescence en affichant la taille de tous les fichiers trouvés. Quand il aura terminé, il devra imprimer un histogramme des tailles de fichiers en utilisant une catégorie précisée en paramètre (par exemple, avec 1 024, les tailles de fichiers de 0 à 1 023 sont placées dans une seule catégorie, celles de 1 024 à 2 047 dans une autre, et ainsi de suite…).

37. Écrivez un programme qui scrute tous les répertoires d'un système de fichiers UNIX, trouve tous les i-nodes qui possèdent un compteur de liens physiques de 2 ou plus. Pour chaque fichier de ce type, il rassemble dans une même liste tous les noms de fichiers qui pointent sur le fichier.

38. Écrivez une nouvelle version du programme UNIX *ls*. Cette version prendra comme argument le nom d'un ou de plusieurs répertoires et listera tous les fichiers de chaque répertoire, avec une ligne par fichier. Chaque champ devra être formaté convenablement en fonction de son type. Le cas échéant, donnez l'adresse du premier bloc du disque.

5

Entrées/sorties

En plus des concepts comme les processus (et les threads), l'espace d'adressage et les fichiers, un système d'exploitation contrôle également tous les périphériques d'entrée/sortie (E/S) de l'ordinateur. Il doit émettre des commandes vers les périphériques, intercepter les interruptions et gérer les erreurs. Il fournit également une interface simple et facile à utiliser entre les périphériques et le reste du système. Dans la mesure du possible, l'interface doit être la même pour tous les périphériques (indépendance des périphériques). Le code des E/S représente une part significative du système d'exploitation. Dans ce chapitre, nous étudierons la gestion des E/S par le système d'exploitation.

Ce chapitre est organisé de la manière suivante. Pour commencer, nous étudierons les principes des E/S du point de vue matériel, puis nous examinerons les logiciels d'E/S en général. Les logiciels d'E/S peuvent être structurés en couches, chacune ayant une tâche définie à effectuer. Nous verrons à quoi servent ces couches et comment elles s'adaptent les unes aux autres.

Après cette introduction, nous détaillerons la partie matérielle et logicielle de plusieurs périphériques : disques, horloges, claviers et systèmes d'affichage. Pour finir, nous nous attellerons à la gestion de l'alimentation.

5.1 Les aspects matériels des E/S

Chacun perçoit les aspects matériels des E/S de manière différente. Les ingénieurs en électronique y pensent en termes de puces, câbles, alimentations électriques, moteurs et autres composants physiques qui constituent le matériel. Les programmeurs, quant à eux, s'intéressent davantage à l'interface proposée au logiciel : les commandes acceptées par le matériel, les fonctions qu'il réalise et les erreurs qu'il peut rapporter. Dans ce livre, nous nous intéressons à la programmation des périphériques d'E/S et non à leur conception, à leur construction ou à leur entretien. C'est pourquoi nous nous restreindrons à la manière dont on programme le matériel et non pas à son principe de fonctionnement interne. Pourtant, la programmation de plusieurs

périphériques d'E/S est souvent intimement liée à leur fonctionnement interne. Dans les trois prochaines sections, nous ferons un petit rappel des relations qui existent entre matériel d'E/S et leur programmation. Considérez-le comme une révision et un complément de l'introduction que nous avons faite à la section 1.4.

5.1.1 Les périphériques d'E/S

Les périphériques d'E/S peuvent être divisés en deux catégories du point de vue des informations manipulées : les **périphériques d'E/S par blocs** (*block devices*) et les **périphériques d'E/S de caractères** (*character devices*). Un périphérique d'E/S par blocs stocke les informations dans des blocs de taille fixe, chaque bloc possédant sa propre adresse. La taille courante des blocs s'étend de 512 à 32 768 octets. La propriété essentielle d'un périphérique d'E/S par blocs est qu'il permet d'écrire et de lire chaque bloc indépendamment des autres. Les disques durs, les CD-ROM et les clés USB représentent les périphériques d'E/S par blocs les plus courants.

La limite entre les périphériques dont les blocs sont adressables et ceux qui ne le sont pas n'est pas très bien définie. Chacun s'accorde à dire qu'un disque est un périphérique avec blocs adressables puisque, quel que soit l'emplacement du bras, il est toujours possible de se déplacer vers un autre cylindre puis d'attendre que le bloc demandé passe sous la tête de lecture/écriture. Prenons à présent l'exemple d'un lecteur de bandes destiné aux sauvegardes. Les bandes contiennent une suite de blocs. Si le lecteur de bandes reçoit la commande de lire le bloc *N*, il peut toujours rembobiner la bande et aller en avant jusqu'à trouver le bloc *N*. Cette opération est analogue à un déplacement du bras sur un disque, mais elle est beaucoup plus longue. En outre, on ne peut pas toujours réécrire un bloc qui se trouve au milieu de la bande. Même s'il est possible d'utiliser des bandes comme périphériques par blocs à accès aléatoire, c'est une déformation de leur fonction et on ne les exploite normalement pas de cette manière.

Un périphérique d'E/S de caractères représente l'autre type de périphérique d'E/S. Il reçoit ou fournit un flot de caractères, sans aucune structure de blocs. Il n'est pas adressable et ne possède aucune fonction de recherche. Les imprimantes, les interfaces réseau, les souris et la plupart des autres périphériques que l'on ne peut apparenter à des disques peuvent être considérés comme des périphériques d'E/S de caractères.

Toutefois, cette classification n'est pas parfaite : certains périphériques ne correspondent pas à ce schéma. Les horloges, par exemple, ne sont pas adressables par blocs, pas plus qu'elles ne génèrent ou n'acceptent des flots de caractères. Elles servent simplement à provoquer des interruptions à des intervalles définis. Les écrans à mémoire ne correspondent pas davantage au modèle. Pourtant, la classification en périphériques d'E/S par blocs/périphériques d'E/S de caractères suffit généralement pour servir de base à l'interaction entre le logiciel du système d'exploitation et un périphérique d'E/S autonome. Le système de fichiers, par exemple, traite uniquement avec les périphériques d'E/S par blocs génériques et laisse la partie dépendante du périphérique au logiciel de niveau inférieur.

Les périphériques présentent des vitesses de fonctionnement et des débits de transfert de données très différents, d'où une pression considérable sur le logiciel qui doit gérer avec des performances correctes une grande variété de périphériques. La figure 5.1 montre les débits de quelques périphériques courants. La majorité de ces périphériques devient de plus en plus rapide avec le temps.

Figure 5.1 • Débits de certains périphériques courants, réseaux et bus.

Périphérique	Débit de données
Clavier	10 octets/s
Souris	100 octets/s
Numériseur	400 Ko/s
Caméscope numérique	3,5 Mo/s
Réseau sans fil 802.11g	6,75 Mo/s
CD-ROM 52x	7,8 Mo/s
Ethernet rapide	12,5 Mo/s
Carte mémoire « Compact Flash »	40 Mo/s
FireWire (IEEE 1394)	50 Mo/s
USB 2.0	60 Mo/s
Réseau SONET OC-12	78 Mo/s
SCSI Ultra 2	80 Mo/s
Ethernet Gigabit	125 Mo/s
Disque dur SATA	300 Mo/s
Bande Ultrium	320 Mo/s
Bus PCI	528 Mo/s

5.1.2 Le contrôleur de périphérique

Les périphériques d'E/S sont généralement constitués d'un composant mécanique et d'un composant électronique. Il est souvent possible de séparer les deux parties pour permettre une conception modulaire et générique. Le composant électronique s'appelle un **contrôleur de périphérique** (*device controller*) ou adaptateur. Sur les ordinateurs personnels, il prend souvent la forme d'une puce sur la carte mère ou d'un circuit imprimé que l'on peut insérer dans un connecteur d'extension (carte PCI). Le composant mécanique est le périphérique lui-même, dont la disposition est illustrée par la figure 1.6.

La carte du contrôleur est généralement équipée d'un connecteur sur lequel est branché un câble que l'on connecte au périphérique. De nombreux contrôleurs sont en mesure de gérer 2, 4, voire 8 périphériques identiques. Si l'interface entre le contrôleur et le périphérique est normalisée, soit en respectant une norme internationale (ANSI, IEEE, ISO) ou une norme de fait, des sociétés peuvent fabriquer des contrôleurs ou des périphériques qui respectent cette interface. De nombreuses sociétés, par exemple, fabriquent des disques durs qui utilisent l'interface IDE, SATA, SCSI, USB ou FireWire (IEEE 1394).

L'interface entre le contrôleur et le périphérique est souvent de très bas niveau. Un disque, par exemple, peut être formaté avec 10 000 secteurs de 512 octets par piste. Le lecteur produit en réalité un flot binaire série, qui commence par un **synchroniseur initial** ou **préambule** (*preamble*), suivi des 4 096 bits d'un secteur, et se termine par une somme de contrôle appelée **code de correction d'erreurs** (ECC, *Error-Correcting Code*). Le préambule est écrit au moment du formatage du disque. Il contient le numéro du cylindre et du secteur, la taille du secteur et d'autres données similaires, ainsi que les informations de synchronisation.

Le travail du contrôleur consiste à convertir le flot binaire série en un bloc d'octets et à apporter toute correction qui s'impose en cas d'erreur. Le bloc d'octets est assemblé, bit par bit, dans une mémoire tampon qui se trouve au sein du contrôleur. Une fois que la somme de contrôle a été vérifiée et que le bloc a été déclaré sans erreur, il peut être copié dans la mémoire principale.

Le contrôleur d'écran (par exemple, un moniteur vidéo) fonctionne lui aussi comme un périphérique utilisant un flot binaire en série de bas niveau. Il lit les octets de la mémoire qui contiennent les caractères à afficher et génère les signaux employés pour la modulation du tube cathodique (CRT, *Cathode Ray Tube*) et l'affichage à l'écran. En outre, le contrôleur génère les signaux de balayage horizontal, et vertical, de même que les synchronisations de retour ligne et vertical. Sans contrôleur vidéo, le programmeur du système d'exploitation devrait programmer explicitement le balayage analogique du tube. Grâce au contrôleur vidéo, le système d'exploitation initialise le contrôleur avec quelques paramètres, tels que le nombre de caractères et de pixels par ligne et le nombre de lignes par écran, et il laisse la transmission et la gestion de l'affichage à l'écran sous la responsabilité du contrôleur. Les écrans plats à matrice active (TFT, *Thin Film Transistor*) sont différents mais sont aussi compliqués.

5.1.3 Les E/S projetées en mémoire

Chaque contrôleur possède quelques registres, appelés registres de contrôle ou registres de commande, qui servent à la communication avec le processeur. En écrivant dans ces registres, le système d'exploitation demande au périphérique de lui délivrer des données, d'en accepter, de se mettre sous ou hors tension lui-même ou encore d'effectuer une action donnée. En lisant ces registres, le système d'exploitation peut connaître l'état du périphérique, savoir s'il est prêt à accepter une nouvelle commande, etc.

En plus des registres de contrôle, de nombreux périphériques sont équipés de mémoire tampon pour les données que le système d'exploitation peut lire ou écrire. Par exemple, les pixels affichés à l'écran d'un ordinateur proviennent souvent de la mémoire vidéo, qui n'est autre qu'un tampon de données, dans lequel les programmes ou le système d'exploitation peuvent écrire.

Il faut alors savoir comment le processeur communique avec les registres de contrôle et les mémoires tampon de données du périphérique. Il existe deux possibilités. Dans le premier cas, chaque registre de contrôle se voit assigner un numéro de **port d'E/S** (*I/O port*), un entier de 8 ou 16 bits. L'ensemble de tous les ports d'E/S forme l'**espace d'E/S** (*I/O port space*), qui est un espace protégé c'est-à-dire qu'un programme utilisateur ne peut pas y accéder (seul le système d'exploitation peut y accéder). En utilisant une instruction d'E/S spéciale comme

```
IN REG,PORT,
```

le processeur lit dans le registre de contrôle `PORT` et stocke le résultat dans le registre `REG` du processeur. De même, avec

```
OUT PORT,REG
```

le processeur écrit le contenu du registre `REG` dans un registre du contrôle. La majorité des ordinateurs récents, comme presque tous les mainframes, tels l'IBM 360 et ses successeurs, fonctionnent ainsi.

Dans cette configuration, les espaces d'adressage de la mémoire et des E/S sont distincts, comme l'illustre la figure 5.2(a). Ainsi, les instructions

```
IN R0,4
```

et

```
MOV R0,4
```

sont complètement différentes dans cette conception. La première instruction lit le contenu du port d'E/S 4 et le place en `R0` alors que la seconde lit le contenu du mot 4 de la mémoire et le place dans `R0`. Les valeurs `4` de ces exemples se réfèrent à des espaces d'adressage différents et sans aucun rapport entre eux.

La seconde approche, introduite avec le PDP-11, consiste à projeter tous les registres de contrôle dans l'espace mémoire, comme le montre la figure 5.2(b). Chaque registre de contrôle se voit attribuer une adresse mémoire unique à laquelle aucune mémoire n'est assignée. Ce système est appelé **E/S projetées ou mappées en mémoire** (*memory-mapped I/O*). En général, les adresses assignées se trouvent au sommet de l'espace d'adressage. La figure 5.2(c) montre un schéma hybride, dans lequel les mémoires tampon de données sont projetées en mémoire et des ports d'E/S séparés pour les registres de contrôle. Le Pentium exploite cette architecture. Les adresses de 640 Ko à 1 Mo sont réservées aux mémoires tampon de données des périphériques dans les ordinateurs compatibles IBM PC, en plus des ports d'E/S de 0 à 64 Ko.

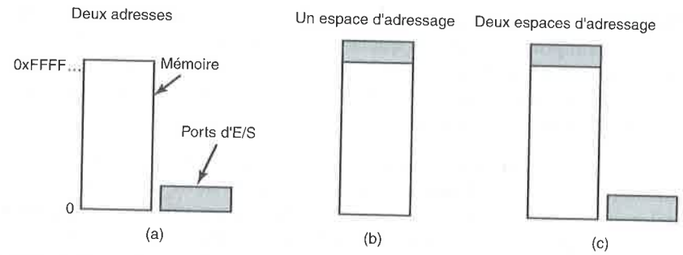

Figure 5.2 • (a) Espaces mémoire et E/S séparés. (b) E/S projetées en mémoire. (c) Système hybride.

Comment fonctionnent ces architectures ? Dans tous les cas, lorsque le processeur veut lire un mot à partir de la mémoire ou d'un port d'E/S, il place l'adresse nécessaire sur les lignes d'adressage du bus, puis revendique un signal READ sur la ligne de contrôle du bus. Un second signal précise s'il s'agit d'une adresse dans l'espace mémoire ou dans l'espace d'E/S. Dans le cas de l'espace mémoire, la mémoire répond à la requête. Dans le cas de l'espace d'E/S, c'est le périphérique d'E/S qui répond à la requête. Si on dispose uniquement de l'espace mémoire [comme dans la figure 5.2(b)], chaque module mémoire et chaque périphérique d'E/S comparent les lignes d'adressage à la plage d'adresses qu'ils desservent. Si l'adresse se trouve dans cette plage, ils répondent à la requête. Dans la mesure où une adresse n'est jamais assignée simultanément à la mémoire et à un périphérique d'E/S, il n'existe aucune ambiguïté ni aucun conflit.

Les deux schémas d'adressage possèdent des avantages et des inconvénients. Commençons par les avantages des E/S projetées en mémoire. Si des instructions d'E/S spéciales sont nécessaires pour lire et écrire dans les registres de contrôle du périphérique, leur accès demande l'usage de code assembleur puisqu'il n'existe aucun moyen d'exécuter une instruction IN ou OUT en C ou C++. L'appel d'une telle procédure surcharge le contrôle des E/S. Par contre, avec les E/S projetées en mémoire, les registres de contrôle sont de simples variables en mémoire et peuvent être adressés en C, à l'instar de toute autre variable. Ainsi, avec les E/S projetées en mémoire, un pilote de périphérique d'E/S peut être entièrement écrit en C. Sans E/S projetées en mémoire, il faut du code assembleur.

De plus, avec les E/S projetées en mémoire, aucun mécanisme de protection spécial n'est nécessaire pour empêcher les processus utilisateur d'effectuer des E/S. Il suffit que le système d'exploitation s'abstienne de placer dans l'espace d'adressage virtuel d'un utilisateur la portion de l'espace d'adressage contenant les registres de contrôle. En outre, si les registres de contrôle de chaque périphérique sont placés dans une page différente de l'espace d'adressage, le système d'exploitation peut donner à un utilisateur le contrôle de certains périphériques et pas à d'autres, en incluant simplement les pages nécessaires dans son tableau de pages. Une telle configuration permet de placer différents pilotes de périphériques dans des espaces d'adressage différents, ce qui réduit la taille du noyau et évite les interférences entre les différents pilotes.

Enfin, avec les E/S projetées en mémoire, chaque instruction qui peut faire référence à la mémoire est également en mesure de faire référence à des registres de contrôle. Par exemple, s'il y a une instruction TEST qui teste si un mot mémoire est égal à 0, elle peut également servir à tester si un registre de contrôle est égal à 0, qui peut être le signal indiquant que le périphérique est inactif et qu'il est prêt à accepter une nouvelle commande. Le code assembleur peut prendre la forme suivante :

```
LOOP:   TEST PORT_4       // vérifie que le port 4 est à 0
        BEQ READY         // si oui, état prêt
        BRANCH LOOP       // sinon, attendre
READY:
```

S'il n'y a pas d'E/S projetées en mémoire, le registre de contrôle doit d'abord être lu dans le processeur, puis testé, ce qui nécessite deux instructions au lieu d'une. Dans le cas de la boucle qui précède, il faut ajouter une quatrième instruction, ce qui réduit légèrement la sensibilité de la détection de l'état inactif d'un périphérique.

Dans un ordinateur, presque tout implique des échanges entre unités fonctionnelles, ce qui est également le cas ici. Les E/S projetées en mémoire possèdent par ailleurs des inconvénients. D'abord, la plupart des ordinateurs actuels disposent d'une forme de mise en cache de mots mémoire. Il serait désastreux de placer dans le cache le registre de contrôle d'un périphérique. Prenons l'exemple de la boucle du code assembleur précédent en présence d'un cache. La première référence à PORT_4 le place en cache. Les références suivantes prennent la valeur du cache sans demander l'état du périphérique. Ensuite, lorsque le périphérique est prêt, le logiciel n'a aucun moyen de le savoir. Et la boucle continue indéfiniment.

Pour éviter ce type de situation avec des E/S projetées en mémoire, le matériel doit pouvoir désactiver la mise en cache sélectivement, par page par exemple. Cette fonctionnalité ajoute une certaine complexité au matériel et au système d'exploitation, qui doit gérer la mise en cache sélective.

Ensuite, s'il n'y a qu'un espace d'adressage, tous les modules mémoire et tous les périphériques d'E/S doivent examiner toutes les adresses mémoire pour savoir qui doit répondre. Si l'ordinateur ne possède qu'un bus, comme dans la figure 5.3(a), la consultation de chaque adresse par tous est simple.

Toutefois, les ordinateurs personnels modernes tendent à posséder un bus mémoire à haut débit dédié [voir figure 5.3(b)], propriété que l'on retrouve aussi dans les mainframes. Ce bus est destiné à optimiser les performances de la mémoire, sans imposer de contraintes particulières aux périphériques d'E/S lents. Les systèmes Pentium, par exemple, peuvent avoir plusieurs bus externes (mémoire, PCI, SCSI, USB, ISA), comme le montre la figure 1.12.

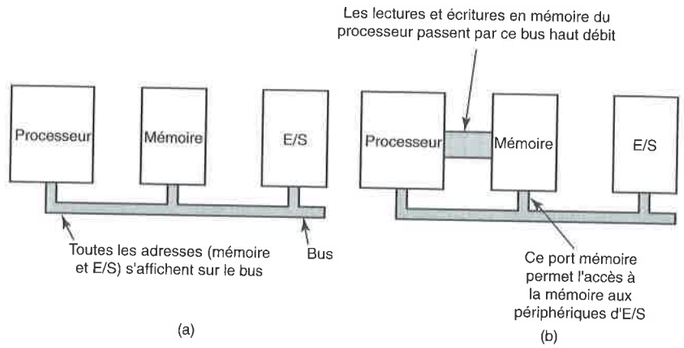

Figure 5.3 • (a) Une architecture à bus unique. (b) Une architecture à deux bus mémoire.

Le problème que pose un bus mémoire séparé sur les ordinateurs ayant des E/S projetées en mémoire est que les périphériques d'E/S ne disposent d'aucun moyen de voir les adresses mémoire lorsqu'elles passent par le bus mémoire ; ils ne peuvent donc pas y répondre. En outre, il faut prendre des mesures spéciales pour faire fonctionner sur un système équipé de plusieurs bus les E/S projetées en mémoire. L'une des solutions consiste à envoyer d'abord toutes les références mémoire à la mémoire. Si celle-ci ne répond pas, le processeur essaie les autres bus. Cette conception peut fonctionner, mais elle augmente la complexité du matériel.

Une deuxième solution consiste à placer un fureteur (*snooping device*) sur le bus mémoire pour transmettre toutes les adresses présentées aux périphériques d'E/S potentiellement intéressés. Le problème, dans ce cas, est que les périphériques d'E/S ne sont pas tous en mesure de traiter les requêtes à la même vitesse que la mémoire.

La troisième possibilité, utilisée dans la configuration du Pentium de la figure 1.12, consiste à filtrer les adresses dans la puce de gestion du bus PCI. Cette puce peut contenir des registres de plages préchargés au moment du démarrage de la machine. Par exemple, les adresses de 640 Ko à 1 Mo peuvent être marquées comme faisant partie de la plage non destinée à la mémoire. Les adresses qui entrent dans l'une des plages marquées comme non destinées à la mémoire sont transférées au bus PCI et non à la mémoire. Cette configuration pose néanmoins un problème : celui de savoir, au moment du démarrage, quelles adresses mémoire ne sont pas réellement des adresses mémoire. En résumé, chaque configuration présente des avantages et des inconvénients, et les compromis sont inévitables.

5.1.4 L'accès direct à la mémoire (DMA)

Que le processeur possède ou non des E/S projetées en mémoire, il doit adresser les contrôleurs de périphérique pour échanger des données avec eux. Le processeur peut demander les données au contrôleur d'E/S octet par octet, mais il gaspille ainsi le

temps du processeur. On fait donc souvent appel à une configuration différente et plus performante, appelée **accès direct à la mémoire** (DMA, *Direct Memory Access*). Le système d'exploitation peut exploiter le DMA uniquement si le matériel est équipé d'un contrôleur DMA, ce qui est le cas de la majorité des systèmes. Il arrive que le contrôleur soit intégré aux contrôleurs de disques ou à d'autres contrôleurs, mais une telle conception demande un contrôleur DMA distinct pour chaque périphérique. En général, il n'existe qu'un seul contrôleur DMA (par exemple, sur la carte mère) pour réguler les transferts, souvent simultanés, vers plusieurs périphériques.

Quel que soit son emplacement, le contrôleur DMA a accès au bus système sans être tributaire du processeur, comme l'illustre la figure 5.4. Il contient plusieurs registres dans lesquels le processeur peut lire et écrire, dont un registre d'adresses mémoire, un registre pour le nombre d'octets et un ou plusieurs registres de contrôle. Ces derniers spécifient le port d'E/S à employer, la direction du transfert (lecture du périphérique d'E/S ou écriture dans le périphérique d'E/S), l'unité de transfert (un octet ou un mot à la fois) et le nombre d'octets à transférer par rafale.

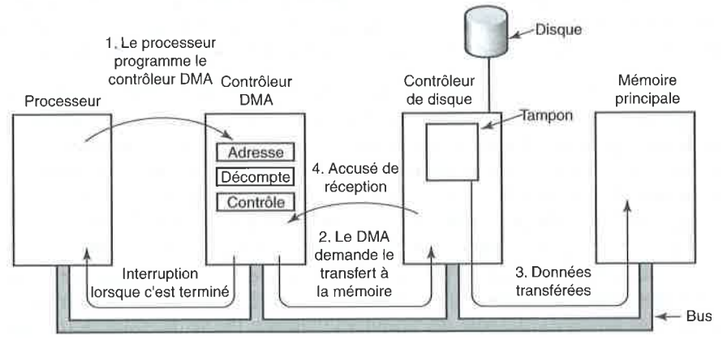

Figure 5.4 • Fonctionnement d'un transfert DMA.

Pour expliquer le fonctionnement du DMA, commençons par étudier de quelle manière se font les lectures de disque quand il n'y a pas de DMA. Tout d'abord, le contrôleur lit le bloc (soit un ou plusieurs secteurs) du disque en série, bit par bit, jusqu'à ce que tout le bloc se trouve dans une mémoire tampon interne. Ensuite, il calcule la somme de contrôle pour vérifier qu'aucune erreur de lecture ne s'est produite. Le contrôleur déclenche alors une interruption. Lorsque le système d'exploitation prend la main, il lit le bloc du disque contenu dans la mémoire tampon du contrôleur, un octet ou un mot à la fois, en exécutant une boucle. À chaque itération de boucle, un octet ou un mot d'un registre du contrôleur est lu et stocké en mémoire principale.

Si l'on fait appel au DMA, la procédure est différente. Pour commencer, le processeur programme le contrôleur DMA en paramétrant ses registres pour qu'il sache quoi

transférer et où (étape 1 de la figure 5.4). Il émet également une commande en direction du contrôleur de disque pour lui demander de lire les données du disque dans sa mémoire tampon interne et de vérifier la somme de contrôle. Quand les données valides se trouvent dans la mémoire tampon du contrôleur de disque, le DMA peut commencer à travailler de manière autonome.

Le contrôleur DMA initie le transfert en émettant une requête de lecture *via* le bus vers le contrôleur de disque (étape 2). Cette requête de lecture est similaire à toute autre requête de lecture et le contrôleur de disque ne sait ni ne se soucie de savoir si elle provient du processeur ou d'un contrôleur DMA. En général, l'adresse mémoire dans laquelle il faut écrire se trouve dans les lignes d'adressage du bus. Ainsi, lorsque le contrôleur de disque récupère le mot suivant dans sa mémoire tampon interne, il sait où l'écrire. L'écriture dans la mémoire est un autre cycle du bus standard (étape 3). Lorsque l'écriture est terminée, le contrôleur de disque envoie un signal de réception au contrôleur DMA, *via* le bus également (étape 4). Le contrôleur DMA incrémente alors l'adresse mémoire à utiliser et décrémente le nombre d'octets. Si ce dernier est toujours supérieur à 0, on répète les étapes 2 à 4 jusqu'à ce qu'il atteigne 0. À ce moment, le contrôleur DMA envoie une interruption au processeur pour lui indiquer que le transfert est terminé. Lorsque le système d'exploitation prend la main, il n'a pas à copier le bloc du disque en mémoire puisqu'il s'y trouve déjà.

La complexité des contrôleurs DMA varie considérablement. Les plus simples gèrent un transfert à la fois, comme décrit précédemment, et les plus complexes peuvent être programmés pour gérer plusieurs transferts simultanés. De tels contrôleurs possèdent plusieurs jeux de registres internes, soit un par canal. Le processeur commence par charger chaque jeu de registres avec les paramètres appropriés au transfert. Chaque transfert exploite un contrôleur de périphérique différent. Une fois que chaque mot a été transféré (étapes 2 à 4 de la figure 5.4), le contrôleur DMA décide du périphérique à servir ensuite. Il peut être configuré de manière à employer un algorithme à **priorité tournante** ou *round robin* ou posséder un schéma de priorité pour favoriser certains périphériques par rapport à d'autres. Plusieurs requêtes adressées à différents contrôleurs de périphériques peuvent se trouver en attente simultanément, à condition qu'il existe un moyen significatif d'indiquer la réception séparément. C'est pourquoi il est fréquent qu'une ligne de confirmation de réception différente du bus serve à chaque canal DMA.

De nombreux bus peuvent fonctionner selon deux modes : le mode un seul mot à la fois ou le mode bloc. Certains contrôleurs DMA peuvent également employer l'un ou l'autre mode. Dans le premier mode, la progression est semblable à celle décrite précédemment : le contrôleur DMA demande le transfert d'un mot et l'obtient. Si le processeur réclame aussi le bus, il doit attendre. Ce procédé est appelé **vol de cycle** (*cycle stealing*) : en effet, le contrôleur du périphérique se faufile et dérobe occasionnellement un cycle de bus au processeur, ce qui le retarde légèrement. En mode bloc, le contrôleur DMA demande au périphérique d'acquérir le bus, il émet une série de transferts puis libère le bus. Il s'agit d'un fonctionnement en **mode rafale** (*burst mode*). Il est plus efficace que le vol de cycle dans la mesure où l'acquisition du bus prend du temps et que plusieurs mots peuvent être transférés au prix d'une acquisition

de bus. L'inconvénient du mode rafale est qu'il peut bloquer le processeur et les autres périphériques pendant une période relativement longue si une rafale importante est transférée.

Dans le modèle dont nous avons parlé, appelé parfois **mode survol** (*fly-by-mode*), le contrôleur DMA commande au contrôleur du périphérique le transfert direct des données à la mémoire principale. Dans un autre mode qu'utilisent certains contrôleurs DMA, le contrôleur du périphérique envoie le mot au contrôleur DMA, qui émet alors une seconde requête de bus pour écrire le mot là où il est supposé aller. Cette configuration nécessite un cycle de bus supplémentaire par mot transféré, mais elle est plus souple en ce qu'elle réalise également des copies de périphérique à périphérique, voire de mémoire à mémoire (en émettant d'abord une lecture de la mémoire puis en émettant une écriture dans la mémoire à une adresse différente).

La plupart des contrôleurs DMA font appel à des adresses mémoire physiques pour les transferts. Pour se servir des adresses physiques, le système d'exploitation doit convertir en adresse physique l'adresse virtuelle de la mémoire tampon prévue, puis écrire cette adresse physique dans le registre d'adressage du contrôleur DMA. Au lieu de cela, certains contrôleurs DMA procèdent selon un autre schéma, qui consiste à écrire les adresses virtuelles dans le contrôleur DMA. Ce dernier doit alors utiliser l'unité de gestion mémoire (MMU, *Memory Management Unit*), afin de réaliser la conversion virtuelle/physique. Dans les rares cas où la MMU fait partie de la mémoire et non du processeur, on peut placer les adresses virtuelles sur le bus.

Nous avons vu précédemment que le disque lit les données dans sa mémoire tampon interne avant que le DMA ne soit lancé. On peut se demander pourquoi le contrôleur ne stocke pas simplement les octets dans la mémoire principale dès qu'il les récupère du disque. Autrement dit, pourquoi faut-il une mémoire tampon interne ? Il y a deux raisons à cela. D'abord, en réalisant une mise en mémoire tampon interne, le contrôleur de disque peut vérifier la somme de contrôle avant de commencer le transfert. Si la somme de contrôle est incorrecte, une erreur est signalée et aucun transfert n'a lieu.

La seconde raison est que lorsqu'un transfert de disque a commencé, les bits continuent d'arriver du disque selon un débit constant, que le contrôleur soit prêt ou non. Si le contrôleur tente d'écrire des données directement dans la mémoire, il doit traverser le bus système à chaque mot transféré. Si le bus est occupé parce qu'un autre périphérique s'en sert (par exemple, en mode rafale), le contrôleur doit attendre. Lorsque le mot suivant du disque arrive avant que le précédent ait été stocké, le contrôleur doit le stocker quelque part. Si le bus est très occupé, le contrôleur peut se retrouver avec plusieurs mots en stock qu'il doit administrer. Or, quand le bloc est placé en mémoire tampon en interne, le bus n'est pas requis avant que le DMA ne commence. Ainsi, la conception du contrôleur est beaucoup plus simple, puisque le transfert DMA vers la mémoire n'est pas lié au temps (certains des anciens contrôleurs se rendaient directement en mémoire avec une faible quantité de mémoire tampon interne mais, lorsque le bus était occupé, le transfert devait être arrêté en raison d'une erreur de saturation).

Les ordinateurs n'exploitent pas tous le DMA. L'argument avancé en défaveur du DMA est que le processeur principal est souvent beaucoup plus rapide que le contrôleur DMA et qu'il peut réaliser l'opération bien plus rapidement (lorsque le facteur limitant n'est pas la vitesse du périphérique d'E/S). S'il n'y a pas d'autre tâche à accomplir, il n'y a aucun sens à ce que le processeur (rapide) attende que le contrôleur DMA (lent) ait terminé. En outre, le fait de se débarrasser du contrôleur DMA et de laisser le processeur effectuer toute l'opération au niveau logiciel permet de réaliser des économies, point important sur les ordinateurs bas de gamme.

5.1.5 Un retour sur les interruptions

Nous avons déjà brièvement abordé les interruptions. Dans un ordinateur personnel classique, la structure des interruptions est analogue à celle de la figure 5.5. Au niveau matériel, les interruptions fonctionnent de la manière suivante : lorsqu'un périphérique d'E/S a terminé la tâche qui lui était dévolue, il déclenche une interruption (en supposant que les interruptions aient été activées par le système d'exploitation). Pour ce faire, il émet un signal sur la ligne du bus auquel il a été assigné. Le signal est détecté par la puce du contrôleur d'interruptions sur la carte mère, qui décide de la suite à donner.

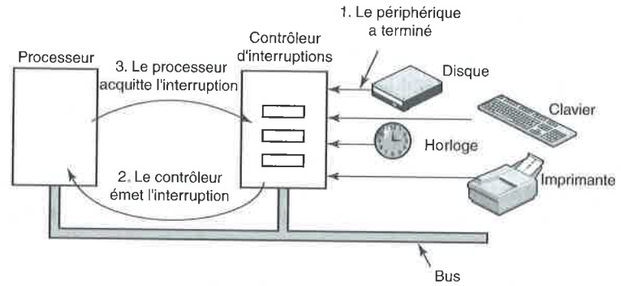

Figure 5.5 • Comment se produit une interruption ? Les connexions entre les périphériques et le contrôleur d'interruptions passent par les lignes d'interruption du bus et non par des câbles dédiés.

Si aucune autre interruption n'est en attente, le contrôleur d'interruptions la traite immédiatement. Si une autre interruption est en cours ou qu'un autre périphérique ait fait une demande simultanée sur une ligne de bus d'une priorité supérieure, le périphérique est ignoré pour l'instant. Dans ce cas, il continue à émettre un signal d'interruption sur le bus jusqu'à ce qu'il soit servi par le processeur.

Pour gérer l'interruption, le contrôleur place un numéro sur les lignes d'adressage afin de désigner le périphérique qui réclame de l'attention et émet un signal qui interrompt le processeur.

À la réception du signal d'interruption, le processeur arrête la tâche en cours et commence une autre tâche. Le numéro qui se trouve sur les lignes d'adressage est utilisé comme index dans une table appelée **vecteur d'interruption**, pour charger le compteur ordinal avec une nouvelle valeur, qui correspond au point de départ de la procédure de service de l'interruption correspondante. En général, les déroutements et les interruptions se servent du même mécanisme et partagent souvent le même vecteur d'interruption. L'emplacement de ce vecteur peut être câblé dans l'ordinateur ou se trouver n'importe où en mémoire, avec un registre du processeur (chargé par le système d'exploitation) pointant vers son origine.

Peu après le début de son exécution, la procédure de service de l'interruption acquitte l'interruption en écrivant une valeur donnée à l'un des ports d'E/S du contrôleur d'interruptions. Cette confirmation indique au contrôleur qu'il est libre d'émettre une autre interruption. Si le processeur diffère cet acquittement jusqu'à ce qu'il soit prêt à gérer la prochaine interruption, on évite les conditions de concurrence impliquant des interruptions presque simultanées. Il est à noter que certains ordinateurs (anciens) ne sont pas équipés d'une puce de contrôleur d'interruptions centralisée ; dans ce cas, chaque contrôleur de périphérique gère ses propres interruptions.

Le matériel sauvegarde toujours certaines informations avant de commencer la procédure de service. Les informations sauvegardées et l'emplacement où elles le sont varient considérablement d'un processeur à l'autre. Il faut au minimum enregistrer le compteur ordinal pour que le processus interrompu puisse redémarrer, mais tous les registres visibles et un grand nombre de registres internes peuvent également être sauvegardés.

L'un des problèmes qui se posent est de savoir où enregistrer ces informations. L'une des options consiste à les placer dans des registres internes que le système d'exploitation peut lire en cas de besoin. Mais cette approche soulève un problème : tant que toutes les informations potentiellement appropriées ne sont pas lues, le contrôleur d'interruptions ne peut pas être confirmé de crainte qu'une seconde interruption n'écrase les registres internes qui conservent l'état. Cette stratégie tend à créer de longs temps morts lorsque les interruptions sont désactivées et entraîne un risque de perte d'interruptions et de données.

Par conséquent, la plupart des processeurs enregistrent les informations sur la pile. Cette approche pose également des problèmes. Pour commencer, quelle pile faut-il employer ? Si on se sert de la pile en cours, il peut fort bien s'agir de la pile d'un processus utilisateur. Il est également possible que le pointeur de la pile ne soit pas initialisé, ce qui provoque une erreur fatale lorsque le matériel essaie d'y écrire des mots. En outre, le pointeur peut désigner la fin d'une page. Après plusieurs écritures en mémoire, la limite de la page peut être atteinte, ce qui génère une erreur de page. Une telle erreur au cours du processus d'interruption matériel soulève une question encore plus importante : où enregistrer l'état pour gérer l'erreur de page ?

Si on utilise la pile du noyau, il y a plus de chances que le pointeur soit initialisé et qu'il pointe vers une page fixe. Toutefois, le basculement en mode noyau peut nécessiter le changement des contextes MMU, ce qui invaliderait probablement une grande

partie du cache et de la mémoire tampon TLB (*Translation Lookaside Buffer*), voire leur totalité. Pour les recharger, on augmente statiquement ou dynamiquement la durée de traitement d'une interruption et, par conséquent, on gaspille du temps processeur.

Les interruptions précises et imprécises

Un autre problème est dû au fait que les processeurs modernes sont fortement à base de pipelines et sont souvent à base d'architectures superscalaires (parallèles). Sur les systèmes plus anciens, après la fin de l'exécution de chaque instruction, le micro-programme ou le matériel vérifiait s'il y avait une interruption en attente. Si tel était le cas, le compteur ordinal et le mot d'état du programme (PSW, *Program Status Word*) étaient placés sur la pile et la séquence d'interruption commençait. Après l'exécution du gestionnaire d'interruptions, le processus inverse avait lieu : l'ancien mot d'état du programme et le compteur ordinal étaient dépilés et le processus précédent continuait.

Ce modèle suppose implicitement que si une interruption se produit immédiatement après une instruction, toutes les instructions jusqu'à cette instruction y compris ont été complètement exécutées et qu'aucune instruction après celle-ci n'a été exécutée. Cette supposition était toujours valide avec les anciens ordinateurs, ce qui n'est pas le cas sur les ordinateurs récents.

Pour commencer, prenons l'exemple du modèle de pipeline de la figure 1.7(a). Que se passe-t-il si une interruption se produit alors que le pipeline est plein (cas le plus courant) ? De nombreuses instructions connaissent des étapes différentes d'exécution. Lorsque l'interruption se produit, la valeur du compteur ordinal peut ne pas refléter le point exact entre les instructions exécutées et non exécutées. En fait, plusieurs instructions peuvent avoir été partiellement exécutées et d'autres peuvent être plus ou moins terminées. Dans cette situation, le compteur ordinal indique plus probablement l'adresse de la prochaine instruction à aller lire et à placer dans le pipeline, et non l'adresse de l'instruction qui vient d'être traitée par l'unité d'exécution.

Si cela n'est pas idéal, la situation est encore pire pour les interruptions sur un ordinateur superscalaire comme celui de la figure 1.7(b). Les instructions peuvent être décomposées en micro-opérations et ces micro-opérations peuvent s'exécuter dans le désordre en fonction de la disponibilité des ressources internes comme les unités fonctionnelles ou les registres. Au moment de l'interruption, certaines instructions commencées il y a longtemps peuvent ne pas être terminées et certaines instructions commencées très récemment peuvent être terminées. Au moment où l'interruption est signalée, il y a de nombreuses instructions dans des états très variés de complétude et avec peu de relations entre elles et le compteur ordinal.

Une interruption qui laisse l'ordinateur dans un état bien défini s'appelle une **interruption précise**. Une telle interruption possède quatre propriétés :

1. Le compteur ordinal, CO en abrégé, (PC, *Program Counter*) est sauvegardé dans un emplacement connu.

2. Toutes les instructions qui précèdent celle pointée par le CO ont été entièrement exécutées.
3. Aucune instruction au-delà de celle pointée par le CO n'a été exécutée.
4. L'état d'exécution de l'instruction pointée par le CO est connu.

Notons qu'il n'est pas interdit que les instructions qui sont au-delà de celle pointée par le CO commencent à s'exécuter. Il faut simplement que toutes les modifications qu'elles font aux registres ou à la mémoire, avant que l'interruption ne se produise, soient annulées. L'instruction pointée peut avoir été exécutée. Il est également permis qu'elle ne l'ait pas été. Toutefois, le cas qui s'applique doit être clairement défini. Souvent, si l'interruption est une interruption d'E/S, l'instruction n'aura pas encore démarré. Cependant, si l'interruption est réellement une erreur (de page par exemple), le CO pointe généralement vers l'instruction qui a provoqué l'erreur pour qu'elle puisse être exécutée à nouveau ultérieurement. La figure 5.6(a) montre l'état d'une interruption précise. Toutes les instructions avant le CO (316) ont été exécutées et aucune au-delà du CO n'a commencé à s'exécuter (ou on est revenu en arrière pour supprimer leurs effets).

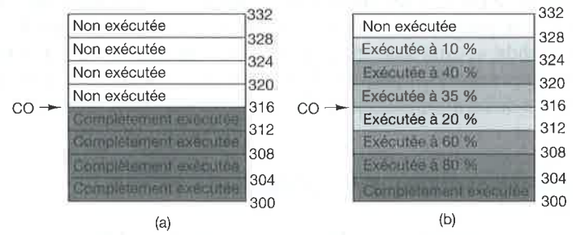

Figure 5.6 • (a) Une interruption précise. (b) Une interruption imprécise.

Une interruption qui ne répond pas à ces exigences est appelée une **interruption imprécise**. Elle complique le travail du développeur du système d'exploitation, qui doit alors savoir ce qui s'est passé et ce qui doit encore se produire. La figure 5.6(b) montre l'état d'une interruption imprécise, où différentes instructions proches du CO sont dans différents états de complétude : les plus anciennes ne sont pas obligatoirement dans un état plus avancé que les nouvelles. Les ordinateurs dont les interruptions sont imprécises déversent généralement une grande partie de l'état interne dans la pile pour permettre au système d'exploitation de comprendre le problème. Le code nécessaire pour la reprise de l'exécution est généralement extrêmement compliqué. La sauvegarde d'une grande quantité d'informations dans la mémoire à chaque interruption ralentit les interruptions et rend la reprise plus difficile. Il en découle une situation ironique dans laquelle les processeurs superscalaires, très rapides, ne sont pas adaptés aux tâches en temps réel à cause de la lenteur des interruptions.

Il existe des ordinateurs conçus pour que certains types d'interruptions et d'erreurs soient précis et d'autres pas. Par exemple, il peut être intéressant que les interruptions d'E/S soient précises, mais que les déroutements provoqués par des erreurs de programmation soient imprécis. En effet, il n'est alors pas nécessaire de redémarrer le processus après une division par zéro. Certains ordinateurs possèdent un bit que l'on peut configurer de manière à obliger toutes les interruptions à être précises. L'inconvénient est que si ce bit est configuré, il oblige le processeur à enregistrer précisément tout ce qu'il fait et à conserver des copies fantômes des registres pour pouvoir générer une interruption précise à tout instant. Toute cette surcharge a un impact important sur les performances.

Certains ordinateurs superscalaires, comme la série des Pentium, prennent en charge les interruptions précises pour permettre aux anciens programmes de fonctionner correctement. La contrepartie de ces interruptions précises est une logique d'interruption extrêmement complexe au sein du processeur. En effet, lorsque le contrôleur d'interruptions signale qu'il va provoquer une interruption, il faut s'assurer que toutes les instructions jusqu'à un certain point sont autorisées à se terminer, et qu'aucune instruction au-delà de ce point ne peut avoir un effet notable sur l'état de l'ordinateur. Dans ce cas, le prix à payer n'est pas le temps que l'on y passe, mais la complexité de conception de la puce. Si les interruptions précises n'étaient pas nécessaires pour des raisons de compatibilité ascendante, cette zone de la puce pourrait servir à des caches plus grands, ce qui rendrait le processeur plus rapide. D'un autre côté, les interruptions imprécises compliquent le système d'exploitation et le ralentissent ; il est donc difficile de savoir quelle approche est réellement la meilleure.

5.2 Les principes des logiciels d'E/S

Nous allons à présent laisser de côté les aspects matériels des E/S pour nous intéresser à leurs aspects logiciels. Nous commencerons par examiner les objectifs des logiciels d'E/S, puis nous verrons les différentes manières d'effectuer des E/S du point de vue du système d'exploitation.

5.2.1 Les objectifs des logiciels d'E/S

L'un des concepts clés des logiciels d'E/S est celui de l'**indépendance par rapport au matériel** (*device independence*). Cela signifie qu'il doit être possible d'écrire des programmes qui accèdent à n'importe quel périphérique d'E/S sans qu'il soit nécessaire de préciser le matériel à l'avance. Par exemple, un programme qui lit un fichier en entrée doit être capable de lire un fichier qui se trouve sur un disque dur, un CD-ROM, un DVD ou une clé USB sans qu'il faille modifier le programme en fonction du périphérique. En outre, on doit pouvoir taper une commande comme

```
sort <input >output
```

et qu'elle puisse fonctionner, que l'entrée provienne de n'importe quel disque ou du clavier, et que la sortie aille sur n'importe quel disque ou sur l'écran. C'est au système d'exploitation de régler les problèmes dus aux différences entre les périphériques et au fait qu'ils nécessitent des séquences de commandes très différentes pour lire ou écrire.

Le principe de **désignation universelle** (*uniform naming*) est étroitement lié à l'indépendance par rapport au matériel. Le nom d'un fichier ou d'un périphérique doit être une chaîne de caractères ou un simple entier et ne doit dépendre en aucune manière du périphérique. Sous UNIX, tous les disques peuvent être intégrés dans la hiérarchie du système de fichiers de manière arbitraire. Ainsi, l'utilisateur n'a pas besoin de connaître le nom qui correspond à un périphérique donné. Par exemple, une clé USB peut être montée (*mounted*) au point de montage */usr/ast/backup*, de manière à ce que le fait de copier un fichier dans */usr/ast/backup/lundi* copie le fichier sur la clé USB. Ainsi, tous les fichiers et périphériques sont adressés de la même manière, c'est-à-dire par leur chemin d'accès.

La **gestion des erreurs** (*error handling*) est un autre point clé lié aux logiciels d'E/S. En général, les erreurs sont gérées aussi près du matériel que possible. Si le contrôleur découvre une erreur de lecture, il essaie de la corriger. S'il ne le peut pas, c'est le pilote de périphérique qui la gère, en essayant par exemple de relire le bloc. De nombreuses erreurs, telles les erreurs de lecture causées par la présence de poussière sur la tête de lecture, sont passagères : elles disparaissent si l'on répète l'opération. C'est uniquement lorsque les couches inférieures ne parviennent pas à résoudre le problème, que les couches supérieures en sont informées. Les erreurs sont souvent récupérées de manière transparente à bas niveau, sans que les niveaux supérieurs en aient connaissance.

Les transferts **synchrones** (bloquants) et les transferts **asynchrones** (pilotés par les interruptions) sont également importants. La plupart des E/S physiques sont asynchrones : le processeur commence le transfert et passe à autre chose en attendant l'interruption. Mais les programmes utilisateur sont bien plus simples à écrire si les opérations d'E/S sont bloquantes : après l'appel système `read`, le programme est automatiquement suspendu jusqu'à ce que les données soient disponibles dans la mémoire tampon. C'est au système d'exploitation de faire en sorte que les opérations pilotées par les interruptions aient l'air bloquantes pour les programmes utilisateur.

La **mise en mémoire tampon** (*buffering*) est un autre principe des logiciels d'E/S. Il arrive souvent que les données qui proviennent d'un périphérique ne puissent pas être stockées directement dans leur emplacement final. Par exemple, lorsqu'un paquet se présente en provenance du réseau, le système d'exploitation ne sait pas où le placer avant de l'avoir rangé quelque part et examiné. En outre, certains périphériques (comme les périphériques de son numérique) sont soumis à des contraintes de temps réel importantes. Par conséquent, les données doivent être placées préalablement dans une mémoire tampon de sortie afin d'établir une séparation entre le débit auquel la mémoire tampon est remplie et le débit auquel elle est vidée, et d'éviter un sous-régime (*underruns*) de la mémoire tampon. La mise en mémoire tampon implique des opérations de copie non négligeables et a souvent un impact important sur les performances des E/S.

Le dernier principe dont nous parlerons concerne le fait qu'un périphérique soit partageable ou non. Certains périphériques d'E/S, tels les disques, peuvent être exploités simultanément par plusieurs utilisateurs. Le fait que plusieurs utilisateurs aient des fichiers ouverts sur le même disque au même moment ne pose aucun problème. En revanche, d'autres périphériques, comme les lecteurs de bandes, doivent être dédiés à un seul utilisateur jusqu'à la fin de l'opération. Il est impossible que deux utilisateurs écrivent des blocs entremêlés aléatoirement sur la même bande. Les périphériques dédiés (non partagés) engendrent une grande diversité de problèmes, comme les interblocages (*deadlocks*). C'est de nouveau au système d'exploitation qu'il incombe de gérer les périphériques partagés et dédiés de manière à éviter les problèmes.

5.2.2 Les E/S programmées

Il existe trois manières fondamentalement différentes de gérer des E/S. Dans cette section, nous étudierons la première (E/S programmées) et examinerons les autres (E/S pilotées par les interruptions et E/S utilisant le DMA) dans les deux sections suivantes. La façon la plus simple de gérer les E/S consiste à laisser le processeur s'occuper de toute l'opération. Cette méthode est celle des **E/S programmées**.

Voyons les E/S programmées par le biais d'un schéma, en prenant l'exemple d'un processus utilisateur qui veut imprimer la chaîne de huit caractères « ABCDEFGH » sur l'imprimante. Il commence par assembler la chaîne dans une mémoire tampon de l'espace utilisateur, comme l'illustre la figure 5.7(a).

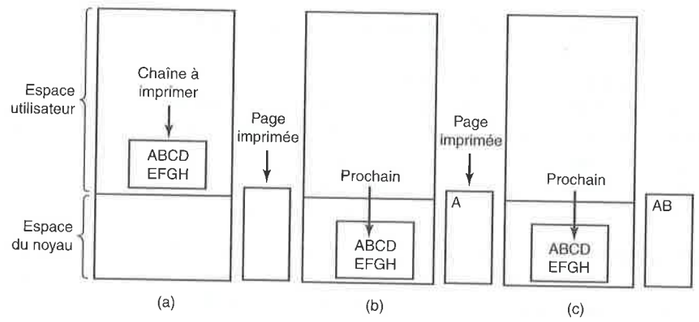

Figure 5.7 • Étapes de l'impression d'une chaîne.

Le processus utilisateur réserve ensuite l'imprimante en émettant un appel système pour l'ouvrir. Si l'imprimante est utilisée par un autre processus, en fonction du système d'exploitation et des paramètres de l'appel, l'appel échoue et retourne un code d'erreur ou se bloque jusqu'à ce que l'imprimante soit libre. Une fois qu'il a obtenu l'imprimante, le processus utilisateur émet un appel système qui commande au système d'exploitation d'imprimer la chaîne sur l'imprimante.

Ensuite, le système d'exploitation copie (généralement) la mémoire tampon contenant la chaîne dans une table, que nous appellerons par exemple *p*, dans l'espace du noyau, où il est plus facilement accessible (parce que le noyau peut avoir à modifier le mappage de la mémoire pour accéder à l'espace utilisateur). Il vérifie alors si l'imprimante est disponible. Si ce n'est pas le cas, il attend jusqu'à ce qu'elle le soit. Dès que l'imprimante est disponible, le système d'exploitation copie le premier caractère dans le registre de données de l'imprimante, en utilisant, dans cet exemple, l'E/S projetée en mémoire. Cette action active l'imprimante, mais le caractère peut ne pas s'afficher immédiatement. En effet, certaines imprimantes placent toute une ligne ou une page en mémoire tampon avant d'imprimer quoi que ce soit. Dans la figure 5.7(b), cependant, nous voyons que le premier caractère a été imprimé et que le système a indiqué le « B » comme prochain caractère à imprimer.

Dès qu'il a copié le premier caractère dans l'imprimante, le système d'exploitation vérifie qu'elle est prête à recevoir un autre caractère. En général, l'imprimante possède un second registre, qui donne son état. Le fait d'écrire dans le registre des données provoque un état non prêt. Lorsque le contrôleur de l'imprimante a traité le caractère en cours, il indique sa disponibilité en positionnant un bit dans son registre d'état ou en y plaçant une valeur donnée.

À ce stade, le système d'exploitation attend que l'imprimante soit à nouveau prête. Lorsque c'est le cas, il imprime le caractère suivant, comme le montre la figure 5.7(c). La boucle continue jusqu'à ce que toute la chaîne soit imprimée. Le contrôle revient alors au processus utilisateur.

Les actions réalisées par le système d'exploitation sont récapitulées à la figure 5.8. Pour commencer, les données sont copiées dans le noyau. Le système d'exploitation entre ensuite dans une petite boucle qui sort un caractère à la fois. L'aspect essentiel de l'E/S programmée, clairement illustré par cette figure, est qu'après avoir émis le caractère, le processeur interroge continuellement le périphérique pour vérifier s'il est prêt à en recevoir un autre. Ce système est souvent appelé **attente active** (*polling* ou *busy waiting*).

Figure 5.8 • Écriture d'une chaîne sur l'imprimante *via* une E/S programmée.

```
copy_from_user(tampon, p, count);                /* p est le tampon du noyau */
for (i = 0; i < count; i++) {                    /* boucle à chaque caractère */
       while (*printer_status_reg != READY) ;    /* boucle jusqu'à l'état prêt*/
       *printer_data_register = p[i];            /* sortie d'un caractère */
}
return_to_user();
```

L'E/S programmée est simple mais a l'inconvénient d'accaparer le processeur jusqu'à ce qu'elle soit complètement réalisée. Cependant, si le temps nécessaire à l'impression d'un caractère est très court (parce que la tâche de l'imprimante se limite à copier le nouveau caractère dans une mémoire tampon interne), l'attente active est parfaite. Dans un système embarqué, où le processeur n'a rien d'autre à faire, l'attente active est possible. Par contre, dans un système plus complexe, où le processeur a d'autres

tâches à effectuer, l'attente active est inefficace. Il faut alors trouver une meilleure méthode pour gérer les E/S.

5.2.3 Les E/S pilotées par les interruptions

Prenons à présent l'exemple d'une impression sur une imprimante qui ne place pas les caractères en mémoire tampon mais imprime chaque caractère dès qu'il se présente. Par exemple, si l'imprimante est en mesure d'imprimer 100 caractères/s, il faut 10 ms pour imprimer chaque caractère. Cela signifie qu'après que chaque caractère est écrit dans le registre des données de l'imprimante, le processeur se trouve dans une boucle inactive pendant 10 ms en attendant de pouvoir sortir le prochain caractère. Cela est plus que suffisant pour procéder à un changement de contexte et exécuter un autre processus durant ces 10 ms, qui sans cela auraient été perdues.

Pour permettre au processeur de réaliser d'autres opérations pendant qu'il attend que l'imprimante soit à nouveau libre, on fait appel aux interruptions. Lorsque l'appel système pour imprimer la chaîne est émis, la mémoire tampon est copiée dans l'espace du noyau, comme nous l'avons vu précédemment, et le premier caractère est copié dans l'imprimante dès qu'elle peut accepter un caractère. Le processeur appelle alors l'ordonnanceur et un autre processus est exécuté. Le processus qui a demandé l'impression de la chaîne est bloqué jusqu'à ce que toute la chaîne soit imprimée. La figure 5.9(a) indique la procédure suivie lors de l'appel système.

Figure 5.9 • Écriture d'une chaîne sur l'imprimante *via* une E/S pilotée par les interruptions. (a) Code exécuté lorsque l'appel système d'impression est émis. (b) Procédure de service de l'interruption.

```
copy_from_user(tampon, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

(a)

```
if (count == 0) {
     unblock_user();
} else {
     *printer_data_register = p[i];
     count = count - 1;
      i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(b)

Lorsque le caractère a été imprimé et que l'imprimante est prête à accepter le suivant, elle génère une interruption. Celle-ci arrête le processus en cours et sauvegarde son état. Ensuite, on exécute la procédure de service de l'interruption de l'imprimante. La figure 5.9(b) montre une version rudimentaire de ce code. S'il n'y a plus de caractère à imprimer, le gestionnaire d'interruptions débloque l'utilisateur. Sinon, il émet le caractère suivant, acquitte l'interruption et retourne au processus qui était en cours d'exécution avant l'interruption, et qui reprend alors où il en était resté.

5.2.4 Les E/S avec DMA

L'inconvénient évident des E/S pilotées par les interruptions est qu'une interruption se produit à chaque caractère. Comme les interruptions prennent du temps, cette méthode consomme une certaine quantité de temps processeur. Une solution consiste à faire appel au DMA. Dans ce cas, l'idée est de laisser le contrôleur DMA fournir les caractères un à un à l'imprimante, sans déranger le processeur. Pour l'essentiel, le DMA fonctionne comme une E/S programmée, à la différence que le contrôleur DMA fait le travail à la place du processeur principal. Une ébauche du code associé à l'E/S avec DMA est illustrée à la figure 5.10.

Figure 5.10 • Impression d'une chaîne *via* le DMA. (a) Code exécuté lorsque l'appel système d'impression est émis. (b) Procédure de service de l'interruption.

```
copy_from_user(tampon, p, count);
set_up_DMA_controller();
scheduler();
```

(a)

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

(b)

Le grand avantage de l'E/S par DMA est de réduire le nombre d'interruptions. On passe ainsi d'une interruption par caractère à une par mémoire tampon imprimée. Si les caractères sont nombreux et les interruptions lentes, l'amélioration peut être considérable. En revanche, le contrôleur DMA est généralement beaucoup plus lent que le processeur principal. Si le contrôleur DMA n'est pas capable de piloter le périphérique à pleine vitesse ou que le processeur n'ait rien d'autre à faire pendant l'impression que d'attendre l'interruption du DMA, les E/S pilotées par les interruptions ou les E/S programmées peuvent être plus appropriées. La plupart du temps, l'utilisation du DMA mérite réflexion.

5.3 La structure en couches des logiciels d'E/S

Les logiciels d'E/S s'organisent typiquement en quatre couches, comme l'illustre la figure 5.11. Chaque couche réalise une fonction précise et présente une interface spécifique aux couches adjacentes. Comme les fonctionnalités des couches et les interfaces diffèrent d'un système à l'autre, notre étude, qui passe en revue toutes les couches à partir de la couche inférieure, n'est pas spécifique à un type d'ordinateur.

5.3.1 Les gestionnaires d'interruptions

Si les E/S programmées sont parfois intéressantes, pour la plupart des E/S, les interruptions sont une réalité déplaisante qui ne peut être évitée. Il faut les cacher, les enfouir dans les entrailles du système d'exploitation, de manière que la partie du système d'exploitation qui en connaît l'existence soit aussi réduite que possible. Le meilleur moyen de les dissimuler est de faire en sorte que le pilote de périphérique qui commence une opération d'E/S se bloque jusqu'à ce qu'elle soit terminée et que

l'interruption survienne. Le pilote peut en effet se bloquer lui-même en effectuant certaines commandes, par exemple, un `down` sur un sémaphore, un `wait` sur une variable conditionnelle et un `receive` sur un message ou tout autre moyen analogue.

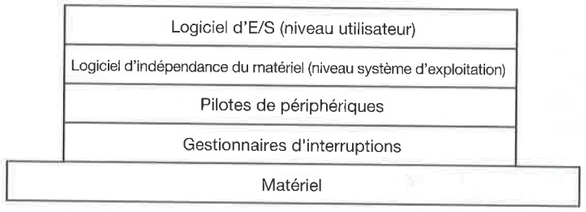

Figure 5.11 • Structure en couches des logiciels d'E/S.

Lorsqu'une interruption se produit, c'est la procédure d'interruption qui la gère et qui sait ce qu'elle doit faire. Elle peut alors débloquer le pilote qui l'a initiée. Dans certains cas, elle exécute simplement un `up` sur un sémaphore. Dans d'autres cas, elle émet un `signal` sur une variable conditionnelle dans un moniteur ou elle envoie un message au pilote bloqué. Dans tous les cas, l'interruption fera que tout pilote précédemment bloqué pourra de nouveau s'exécuter. Ce modèle fonctionne mieux si les pilotes sont structurés sous forme de processus du noyau, possédant leurs propres états, piles et compteur ordinal.

Bien entendu, la réalité n'est pas aussi simple. Le traitement d'une interruption ne se limite pas à intercepter l'interruption, à faire un `up` sur un sémaphore puis à exécuter une instruction `IRET` pour revenir de l'interruption à l'état précédent. Cela représente une masse de travail importante pour le système d'exploitation. Nous allons décrire la procédure sous forme d'étapes qui doivent être mises en œuvre au niveau logiciel une fois que l'interruption matérielle est terminée. Il faut avoir à l'esprit que les paramètres sont fortement dépendants du système et que certaines étapes peuvent ne pas être nécessaires sur un ordinateur donné, alors que des étapes non listées le seront. En outre, ces étapes peuvent se produire dans un ordre différent.

1. Sauvegarder tous les registres (y compris le mot d'état du programme) qui n'ont pas encore été sauvegardés par le matériel d'interruption.
2. Configurer un contexte pour la procédure de service de l'interruption. Cela implique la configuration du TLB, de l'unité de gestion mémoire (MMU) et d'une table des pages.
3. Configurer une pile pour la procédure de service de l'interruption.
4. Acquitter l'interruption auprès du contrôleur d'interruptions. S'il n'existe pas de contrôleur d'interruptions centralisé, autoriser à nouveau les interruptions.
5. Copier les registres depuis l'emplacement où ils ont été enregistrés (si possible dans une pile) vers la table des processus.

6. Exécuter la procédure de service de l'interruption. Cela extraira les informations des registres du contrôleur du périphérique qui a provoqué l'interruption.
7. Choisir le prochain processus à exécuter. Si l'interruption a débloqué un processus à haute priorité qui était bloqué, on peut choisir de l'exécuter maintenant.
8. Configurer le contexte de la MMU pour le prochain processus à exécuter. Il peut être nécessaire de configurer un TLB.
9. Charger les registres du nouveau processus, y compris le mot d'état du programme.
10. Exécuter le nouveau processus.

Comme vous pouvez le constater, le traitement des interruptions est loin d'être insignifiant. Il demande également un nombre considérable d'instructions processeur, en particulier sur les ordinateurs dans lesquels il existe une mémoire virtuelle, et pour lesquels il faut configurer des tables de pages ou sauvegarder l'état de la MMU (par exemple, les bits *R* et *M*). Sur certains ordinateurs, le TLB et le cache du processeur doivent être gérés lorsque l'on bascule entre les modes utilisateur et noyau, ce qui augmente le nombre de cycles de l'ordinateur.

5.3.2 Les pilotes de périphériques

Précédemment dans ce chapitre, nous avons étudié le fonctionnement des contrôleurs de périphériques. Nous avons vu que chaque contrôleur possède des registres de contrôle qui servent à lui adresser des commandes, à lire son état ou les deux. Le nombre de registres de contrôle et la nature des commandes varient radicalement d'un périphérique à l'autre. Par exemple, un pilote de souris doit accepter des informations provenant de la souris et qui lui indiquent la distance parcourue ainsi que le bouton sur lequel on a appuyé. Par contre, un pilote de disque doit connaître les secteurs, pistes, cylindres, têtes, mouvements du bras, mouvements du moteur, temps d'établissement de la tête et tout autre mécanisme permettant au disque de fonctionner correctement. Les pilotes sont, par conséquent, très différents.

En conséquence, chaque périphérique d'E/S connecté à un ordinateur doit disposer d'un programme spécifique au périphérique pour le contrôler. Ce programme, appelé **pilote de périphérique** (*device driver*), est généralement écrit par le fabricant du périphérique et livré avec. Dans la mesure où chaque système d'exploitation a besoin de ses propres pilotes, les fabricants de périphériques fournissent généralement des pilotes pour plusieurs systèmes d'exploitation.

Chaque pilote de périphérique gère normalement un type de périphérique ou tout au plus une classe de périphériques étroitement liés. Par exemple, le pilote d'un disque SCSI peut généralement gérer plusieurs disques SCSI de tailles et de vitesses différentes, voire un CD-ROM SCSI. En revanche, une souris et une manette de jeu sont si différentes que des pilotes distincts sont nécessaires. Il n'existe aucune restriction technique à ce qu'un pilote de périphérique contrôle plusieurs périphériques sans relation entre eux, mais ce n'est tout simplement pas une bonne idée.

Pour accéder au périphérique, autrement dit aux registres du contrôleur, le pilote de périphérique doit normalement faire partie du noyau du système d'exploitation, tout au moins dans les architectures actuelles. Il est effectivement possible de construire des pilotes qui s'exécutent dans l'espace utilisateur, en réalisant des appels système pour lire et écrire les registres des périphériques. Cette architecture isole le noyau des pilotes et isole un pilote de tous les autres éliminant ainsi la principale source de pannes système, dues aux pilotes bogués qui interfèrent avec le noyau d'une manière ou d'une autre. Pour écrire des systèmes avec une haute fiabilité, c'est définitivement le chemin qu'il faut suivre. MINIX 3 est un exemple de système dont les pilotes de périphériques s'exécutent en tant que processus utilisateur. Cependant, la plupart des systèmes d'exploitation actuels attendent des pilotes qu'ils s'exécutent en mode noyau. Nous suivrons donc ce modèle.

Dans la mesure où les concepteurs de systèmes d'exploitation savent que des blocs de code (les pilotes) écrits par d'autres seront installés sur leurs systèmes, ils doivent faire en sorte que cela soit possible dans leur architecture. Ils doivent ainsi proposer un modèle bien défini des actions d'un pilote et de son interaction avec le reste du système d'exploitation. Les pilotes de périphériques interviennent normalement dans une couche située sous le reste du système d'exploitation, comme le montre la figure 5.12.

Les systèmes d'exploitation classent généralement les pilotes par catégories. Les catégories les plus courantes sont les **périphériques d'E/S par blocs** (comme les disques qui contiennent plusieurs blocs de données que l'on peut adresser indépendamment) et les **périphériques d'E/S de caractères** (comme les claviers et les imprimantes, qui génèrent ou acceptent des flots de caractères).

La plupart des systèmes d'exploitation définissent une interface standard, que tous les pilotes de périphériques d'E/S par blocs doivent prendre en charge, et une seconde interface standard qui doit être supportée par tous les pilotes de périphériques d'E/S de caractères. Ces interfaces se composent d'un certain nombre de procédures que le reste du système d'exploitation peut appeler pour pouvoir exploiter le pilote approprié. Parmi les procédures classiques, on trouve celles qui lisent un bloc (périphériques d'E/S par blocs) et celles qui écrivent une chaîne de caractères (périphériques d'E/S de caractères).

Dans certains systèmes, le système d'exploitation est un programme binaire unique qui contient l'ensemble des pilotes dont il aura besoin, qui sont déjà compilés. Ce schéma a été la norme pendant des années sur les systèmes UNIX, car ils étaient exploités dans des centres informatiques où les périphériques changeaient rarement. Quand un nouveau périphérique était ajouté, l'administrateur système recompilait simplement le noyau avec le nouveau pilote pour créer un nouveau code binaire.

Avec l'arrivée des ordinateurs personnels et de leurs myriades de périphériques d'E/S, ce modèle ne fonctionne plus. Peu d'utilisateurs sont capables de recompiler le noyau ou de faire une nouvelle édition des liens du noyau, même s'ils disposent du code source ou des modules d'objets, ce qui n'est pas toujours le cas. Par conséquent, les

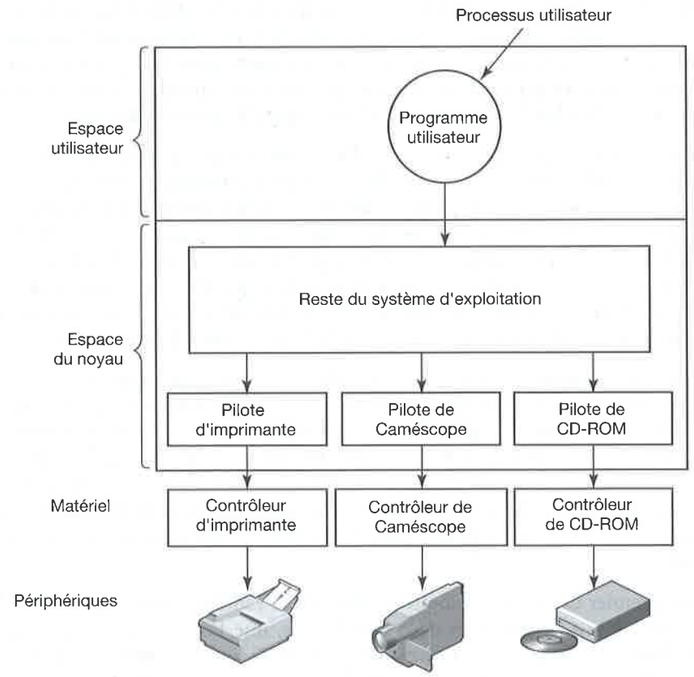

Figure 5.12 • Positionnement logique des pilotes de périphériques. En réalité, toute la communication entre les pilotes et les contrôleurs de périphériques passe par le bus.

systèmes d'exploitation, à commencer par MS-DOS, sont passés à un modèle dans lequel les pilotes sont chargés dynamiquement dans le système pendant l'exécution. Selon le système, le chargement des pilotes s'effectue différemment.

Un pilote de périphérique possède plusieurs fonctions. La plus évidente consiste à accepter les requêtes abstraites de lecture et d'écriture émises par le logiciel d'indépendance du matériel qui se trouve au-dessus de lui et à vérifier qu'elles sont exécutées. Mais il doit aussi prendre en charge d'autres tâches, comme initialiser le périphérique si nécessaire, ou gérer les exigences en matière d'alimentation électrique et enregistrer dans un journal les événements.

De nombreux pilotes de périphériques possèdent une structure similaire. Un pilote classique commence par vérifier que les paramètres d'entrée sont valides. Si ce n'est pas le cas, il retourne une erreur. S'ils sont valides, il peut être nécessaire de convertir les termes abstraits en termes concrets. Pour un pilote de disque, cela implique la conversion d'un numéro de bloc linéaire en numéros de tête, piste, secteurs et cylindre selon la géométrie du disque.

Le pilote vérifie ensuite si le périphérique est utilisé. Le cas échéant, la requête est placée en file d'attente pour un traitement ultérieur. Si le périphérique est inoccupé, son registre d'état est examiné pour vérifier que la requête peut être gérée. Il peut être nécessaire d'allumer le périphérique ou de démarrer un moteur avant de commencer les transferts. Une fois le périphérique allumé et prêt, le contrôle réel débute.

Le contrôle du périphérique consiste en l'émission d'une séquence de commandes. C'est dans le pilote qu'est déterminée la séquence de commandes, selon les actions à entreprendre. Une fois que le pilote connaît les commandes qu'il doit émettre, il les écrit dans les registres du contrôleur. Après que chaque commande est écrite dans le contrôleur, il peut être nécessaire de vérifier qu'il l'a acceptée et qu'il est prêt à accepter la suivante. Cette séquence se poursuit jusqu'à ce que toutes les commandes aient été émises. Certains contrôleurs reçoivent une liste de commandes (en mémoire) qu'ils doivent alors lire et traiter seuls, sans l'aide du système d'exploitation.

Une fois les commandes émises, deux situations peuvent se présenter. Dans la majorité des cas, le pilote de périphérique attend que le contrôleur termine sa tâche et se bloque lui-même jusqu'à ce que l'interruption vienne le débloquer. Dans d'autres cas, l'opération se termine sans attente, auquel cas le pilote n'a pas besoin de se bloquer : à titre d'exemple, pour faire défiler un écran en mode caractères, il est seulement nécessaire d'écrire quelques octets dans les registres du contrôleur. Il n'est besoin d'aucun mouvement mécanique et toute l'opération est exécutée en quelques nanosecondes.

Dans le premier cas, le pilote bloqué est réveillé par l'interruption. Dans le deuxième cas, il ne « dort » jamais. Mais dans tous les cas, une fois l'opération terminée, le pilote doit vérifier la présence d'éventuelles erreurs. Si tout s'est bien déroulé, le pilote dispose de données qu'il doit passer au logiciel d'indépendance du matériel (par exemple, un bloc qu'il vient de lire). Pour finir, il retourne à son appelant des informations d'état pour le rapport d'erreurs. Si d'autres requêtes se trouvent en attente, l'une d'elles est sélectionnée et commence. Quand il n'y a rien en attente, le pilote se bloque en attendant la prochaine requête.

Ce modèle simple n'est qu'une approximation grossière de la réalité. En effet, de nombreux facteurs rendent le code bien plus complexe. Un périphérique qui termine une E/S peut interrompre le pilote en cours d'exécution. Il peut effectivement forcer le pilote actuel à s'exécuter. Par exemple, un paquet peut se présenter alors que le pilote réseau est en train de traiter un paquet entrant. Par conséquent, les pilotes doivent être **réentrants**, c'est-à-dire qu'un pilote en cours d'exécution doit s'attendre à être appelé une seconde fois avant la fin du premier appel.

Dans un système connectable à chaud (*hot pluggable*), on peut ajouter et supprimer des périphériques alors que l'ordinateur est sous tension. Ainsi, tandis que le pilote est occupé par un périphérique, le système peut l'informer que l'utilisateur a soudainement supprimé ce périphérique du système. Non seulement le transfert d'E/S en cours doit être abandonné sans endommager les structures de données du noyau, mais les requêtes en attente du périphérique supprimé doivent également être éliminées du système. Sans compter qu'il faut alors avertir les appelants de la mauvaise

nouvelle. En outre, l'ajout inattendu de nouveaux périphériques oblige le noyau à jongler avec les ressources (par exemple les lignes d'interruptions), afin d'éliminer les anciennes du pilote et lui en attribuer de nouvelles à la place.

Les pilotes ne sont pas autorisés à émettre des appels système, mais il leur faut souvent interagir avec le reste du noyau. Généralement, les appels à certaines procédures du noyau sont autorisés, notamment les appels pour allouer ou désallouer des pages de mémoire physique afin de les exploiter comme mémoire tampon. D'autres appels servent à gérer l'unité de gestion de la mémoire, les horloges, le contrôleur DMA, le contrôleur d'interruptions, etc.

5.3.3 Le logiciel d'indépendance par rapport au matériel

Bien que certains logiciels d'E/S soient spécifiques au périphérique, ce n'est pas le cas de tous. La frontière qui sépare les pilotes du logiciel d'indépendance par rapport au matériel est floue et dépend du système (et du périphérique). En effet, certaines fonctions qui pourraient être réalisées indépendamment du périphérique sont en réalité effectuées par les pilotes, entre autres pour des raisons d'efficacité. Les fonctions de la figure 5.13 sont généralement prises en charge par le logiciel d'indépendance par rapport au matériel.

Figure 5.13 • Fonctions du logiciel d'indépendance par rapport au matériel.

Interface uniforme pour les pilotes de périphériques
Gestion des erreurs
Allocation et libération des périphériques dédiés
Fournir une taille de bloc indépendante du périphérique

La fonction de base du logiciel d'indépendance par rapport au matériel consiste à assurer des fonctions d'E/S communes à tous les périphériques et à fournir une interface uniforme au logiciel utilisateur. Nous allons à présent examiner plus en détail les principales fonctions.

Interface uniforme pour les pilotes de périphériques

L'un des principaux problèmes d'un système d'exploitation est de faire en sorte que tous les périphériques et pilotes d'E/S aient l'air plus ou moins analogues. Si les disques, imprimantes, claviers, etc. ont tous une interface différente, chaque fois qu'un nouveau périphérique se présente, le système d'exploitation doit être modifié en conséquence. Or, il est préférable d'éviter de modifier le système d'exploitation à chaque nouvelle installation de périphérique.

L'interface entre les pilotes de périphériques et le reste du système d'exploitation constitue un aspect important de ce problème. La figure 5.14(a) illustre une situation dans laquelle chaque pilote de périphérique présente une interface différente au

système d'exploitation. Cela signifie que les fonctions mises à disposition par le pilote pour les appels système, ainsi que les fonctions du noyau dont le pilote a besoin, diffèrent d'un pilote à l'autre. Si l'on tient compte de ces deux points, on comprend que l'interface de chaque nouveau pilote demande un nouvel effort de programmation.

En revanche, la figure 5.14(b) illustre une conception différente dans laquelle tous les pilotes présentent la même interface. Il devient alors beaucoup plus simple d'insérer un nouveau périphérique, à condition qu'il se conforme à l'interface d'un pilote. Cela signifie également que les développeurs de pilotes savent ce que l'on attend d'eux. Dans la pratique, les périphériques ne sont pas tous identiques, mais il n'existe qu'un petit nombre de types de périphériques différents qui présentent dans chaque type de nombreuses similitudes.

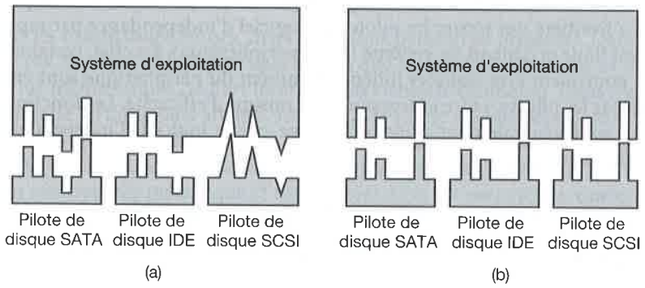

Figure 5.14 • (a) Absence d'une interface de pilote standard. (b) Présence d'une interface de pilote standard.

La manière dont cela fonctionne est la suivante. Pour chaque type de périphériques, tel que les disques ou les imprimantes, le système d'exploitation définit un ensemble de fonctions que le pilote doit apporter. Pour un disque cela inclut naturellement la lecture et l'écriture mais aussi la mise sous ou hors tension, le formatage, ou tout autre chose nécessaire aux disques. Souvent le pilote contient une table avec des pointeurs vers lui-même pour ces fonctions. Lorsque le pilote est chargé, le système d'exploitation enregistre l'adresse de cette table de pointeurs de fonctions ; ainsi, quand il désire appeler l'une de ces fonctions, il peut effectuer un appel indirect *via* cette table. Cette table de pointeurs de fonctions définit l'interface du pilote avec le reste du système d'exploitation. Tous les périphériques d'un type donné (disque, imprimante, etc.) doivent respecter cette interface.

Un autre aspect de l'interface uniforme concerne les noms des périphériques d'E/S. Le logiciel d'indépendance du matériel veille à associer les noms de périphériques symboliques avec le pilote idoine. Par exemple, sous UNIX, un nom de périphérique comme */dev/disque0* indique uniquement l'i-node d'un fichier spécifique ; cet i-node contient également le **numéro de périphérique majeur** (*major device number*), qui sert à localiser le pilote approprié, ainsi que le **numéro de périphérique mineur**

(*minor device number*), passé comme paramètre au pilote pour indiquer l'unité à lire ou dans laquelle écrire. Tous les périphériques possèdent des numéros majeurs et mineurs et on accède à tous les pilotes par le numéro de périphérique majeur pour sélectionner le pilote.

À la dénomination est étroitement liée la protection : comment le système procède-t-il pour interdire l'accès aux périphériques auxquels les utilisateurs ne sont pas autorisés à accéder ? Sous UNIX et Windows, les périphériques se trouvent dans le système de fichiers sous forme d'objets nommés, ce qui signifie que les règles de protection habituelles des fichiers s'appliquent également aux périphériques d'E/S. L'administrateur système peut définir des permissions pour chaque périphérique.

Mise en mémoire tampon

La mise en mémoire tampon (*buffering*) est un autre aspect important, tant pour les périphériques d'E/S par blocs que pour les périphériques d'E/S de caractères, et ce pour plusieurs raisons. Prenons l'exemple d'un processus qui veut lire les données d'un modem. Pour traiter les caractères entrants, le processus utilisateur peut émettre un appel système `read` et se bloquer dans l'attente d'un caractère. Chaque caractère qui se présente provoque une interruption. La procédure de service de l'interruption remet le caractère au processus utilisateur et le débloque. Une fois le caractère placé à un emplacement défini, le processus lit un autre caractère et se bloque à nouveau. Ce modèle est illustré par la figure 5.15(a).

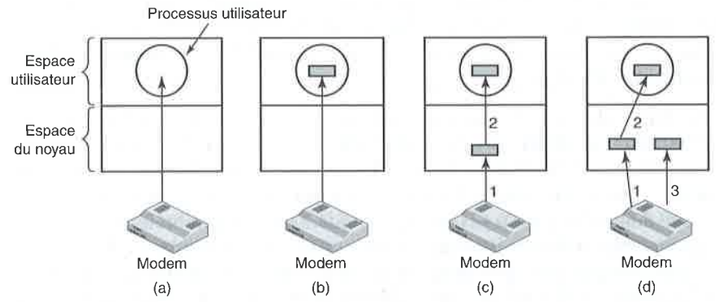

Figure 5.15 • (a) Entrée sans mémoire tampon. (b) Mise en mémoire tampon dans l'espace utilisateur. (c) Mise en mémoire tampon dans le noyau, suivie d'une copie dans l'espace utilisateur. (d) Double mémoire tampon dans le noyau.

Le problème que pose cette méthode est que le processus utilisateur doit être exécuté à chaque caractère entrant. Il est donc préférable d'éviter cette conception, puisque l'opération qui consiste à permettre au processus de s'exécuter de nombreuses fois pour de courtes durées manque d'efficacité.

La figure 5.15(b) montre un modèle amélioré. Dans cet exemple, le processus utilisateur fournit un tampon de *n* caractères dans l'espace utilisateur et réalise une lecture

des n caractères. La procédure de service de l'interruption place les caractères entrants dans le tampon jusqu'à ce qu'il soit plein. Elle réveille ensuite le processus utilisateur. Cette méthode est beaucoup plus efficace que la précédente mais elle présente également un inconvénient : que se passe-t-il si la page de la mémoire tampon est renvoyée lorsqu'un caractère se présente ? La mémoire tampon peut être bloquée en mémoire, mais si plusieurs processus commencent à verrouiller des pages en mémoire, le pool de pages disponibles va se réduire et les performances s'en ressentiront.

Une autre approche consiste à créer une mémoire tampon dans le noyau (*kernel buffer*). Dans ce cas, le gestionnaire d'interruptions y place les caractères, comme l'illustre la figure 5.15(c). Lorsque cette mémoire tampon est pleine, on présente la page contenant la mémoire tampon de l'utilisateur et on y copie la mémoire tampon du noyau en une seule opération. Cette procédure est beaucoup plus efficace.

Cependant, même cette méthode pose problème : que deviennent les caractères qui arrivent pendant que la page qui contient la mémoire tampon de l'utilisateur est chargée à partir du disque ? Comme la mémoire tampon est pleine, il n'y a pas de place pour les y stocker. Une solution consiste à disposer d'une seconde mémoire tampon dans le noyau. Une fois que la première mémoire tampon est pleine, mais avant qu'elle soit vidée, on utilise la seconde, comme le montre la figure 5.15(d). Lorsque la seconde mémoire tampon est pleine, elle peut être copiée dans l'espace de l'utilisateur (en supposant que celui-ci l'ait demandé). Pendant que la seconde mémoire tampon est copiée dans l'espace utilisateur, la première peut servir à recevoir de nouveaux caractères. Ainsi, les deux mémoires tampon s'utilisent à tour de rôle : pendant que l'une est copiée dans l'espace utilisateur, l'autre enregistre les nouvelles entrées. Un tel schéma de mise en mémoire tampon s'appelle **double mémoire tampon** (*double buffering*).

Une autre forme de mise en mémoire tampon qui est largement répandue est la **mémoire tampon circulaire** (*circular buffer*). Elle est composée d'une zone mémoire et de deux pointeurs. Un pointeur désigne le prochain mot disponible, où une nouvelle donnée peut y être placée. Le second pointeur désigne le premier mot qui contient une donnée et qui n'a pas encore été vidé. Dans de nombreuses circonstances, le matériel fait progresser le premier pointeur au fur et à mesure qu'il ajoute des données (par exemple celles qui proviennent du réseau) et le système fait progresser le second pointeur au fur et à mesure qu'il extrait et analyse les données. Les deux pointeurs parcourent toute la mémoire tampon, retournant en bas quand ils atteignent le haut.

La mise en mémoire tampon est également importante en sortie. Voyons, par exemple, comment s'effectue la sortie vers le modem sans mémoire tampon, selon le modèle de la figure 5.15(b). Le processus utilisateur exécute l'appel système `write` pour sortir n caractères. À ce stade, le système dispose de deux possibilités. Il peut bloquer l'utilisateur jusqu'à ce que les caractères soient sortis, mais cela peut prendre beaucoup de temps sur une ligne téléphonique à faible débit. Il peut également libérer immédiatement l'utilisateur et réaliser l'E/S pendant que l'utilisateur continue à travailler, mais cela conduit à une difficulté plus grande encore : comment le processus

utilisateur sait-il que la sortie est terminée et qu'il peut réutiliser le tampon ? Le système pourrait générer un signal ou une interruption logicielle, mais ce style de programmation est périlleux et tend à produire des conditions de concurrence. Il existe une meilleure solution : le noyau copie les données dans une mémoire tampon de noyau, analogue à celle de la figure 5.15(c) (mais dans l'autre sens) et débloque immédiatement l'appelant. Ainsi, le fait que l'E/S soit terminée n'a plus d'importance. L'utilisateur est libre de réutiliser sa mémoire tampon dès qu'il est débloqué.

La mise en mémoire tampon est une technique largement répandue, mais elle possède aussi un inconvénient. Si les données sont trop souvent placées en mémoire tampon, les performances en souffrent. Prenons l'exemple du réseau de la figure 5.16. L'utilisateur émet un appel système pour écrire sur le réseau, et le noyau copie le paquet dans une mémoire tampon du noyau pour permettre à l'utilisateur de continuer sa procédure immédiatement (étape 1). À ce moment, le processus utilisateur peut réutiliser la mémoire tampon.

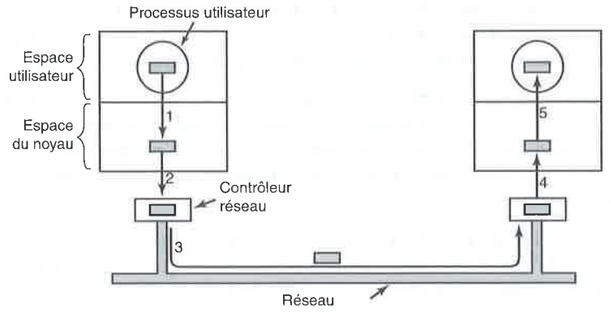

Figure 5.16 • Un réseau peut comprendre de nombreuses copies d'un même paquet.

Lorsque le pilote est appelé, il copie le paquet dans le contrôleur réseau pour la sortie (étape 2). Il ne réalise pas la sortie directement vers le réseau physique depuis la mémoire tampon du noyau. En effet, une fois que la transmission du paquet est commencée, elle doit continuer à une vitesse uniforme. Or le pilote n'est pas en mesure de garantir qu'il pourra récupérer les données de la mémoire à une vitesse uniforme, puisque les canaux DMA et les autres périphériques d'E/S peuvent voler des cycles au processeur. Si un mot n'est pas récupéré à temps, le paquet sera abîmé. Mais en le plaçant dans la mémoire tampon du contrôleur, on évite ce problème.

Une fois que le paquet est copié dans la mémoire tampon interne du contrôleur, il est émis sur le réseau (étape 3). Les bits arrivent au destinataire peu de temps après avoir été envoyés. Ainsi, immédiatement après que le dernier bit est envoyé, celui-ci arrive à son destinataire, où le paquet est placé dans la mémoire tampon du contrôleur. Le paquet est ensuite copié dans la mémoire tampon du noyau du destinataire (étape 4). Pour finir, il est copié dans la mémoire tampon du processus de réception (étape 5).

Généralement, le destinataire envoie une confirmation de réception. Lorsque celle-ci parvient à l'expéditeur, il peut envoyer le paquet suivant. Il est important de noter que ce processus de copie ralentit la vitesse de transmission, dans la mesure où les différentes étapes sont séquentielles.

La gestion des erreurs

Les erreurs sont bien plus courantes dans le contexte des E/S que dans d'autres contextes. Lorsqu'elles se produisent, le système d'exploitation doit les gérer de la meilleure manière possible. De nombreuses erreurs sont spécifiques au périphérique et doivent être gérées par le pilote approprié, mais la structure de la gestion des erreurs est indépendante du périphérique.

Les erreurs de programmation, par exemple, sont un type d'erreurs d'E/S. Elles se produisent quand un processus demande une opération impossible, comme écrire dans un périphérique d'entrée (clavier, souris, scanner, etc.) ou lire à partir d'un périphérique de sortie (imprimante, table traçante, etc.). On trouve également des erreurs telles qu'une adresse de mémoire tampon incorrecte ou un périphérique non valide (par exemple disque 3 alors que le système ne contient que deux disques). Pour prendre ces erreurs en charge, il suffit de retourner à l'appelant le code de l'erreur.

Les erreurs d'E/S sont un autre type d'erreur. Elles se produisent par exemple si on essaie d'écrire dans un bloc de disque endommagé ou que l'on tente de lire des données provenant d'un caméscope éteint. Dans de tels cas, c'est au pilote de déterminer ce qu'il convient de faire. S'il ne le sait pas, il retourne le problème au logiciel d'indépendance du matériel.

L'action entreprise par ce logiciel dépend alors de l'environnement et de la nature de l'erreur. S'il s'agit d'une simple erreur de lecture et qu'il existe un utilisateur en mode interactif, le logiciel peut afficher une boîte de dialogue qui demande à l'utilisateur ce qu'il veut faire : recommencer un certain nombre de fois, ignorer l'erreur ou arrêter le processus appelant. S'il n'y a pas d'utilisateur, la seule option réellement disponible est l'échec de l'appel système avec un code d'erreur.

Certaines erreurs ne peuvent toutefois pas être traitées de cette manière. Par exemple, une structure de données décisive, comme le répertoire racine ou une liste de blocs libres, peut avoir été détruite. Dans ce cas, le système doit afficher un message d'erreur et s'arrêter.

Allocation et libération des périphériques dédiés

Certains périphériques, comme les graveurs de CD-ROM, ne peuvent être exploités que par un seul processus à un instant donné. C'est au système d'exploitation d'examiner les requêtes d'utilisation du périphérique et de les accepter ou de les rejeter, selon que le périphérique demandé est disponible ou non. Une manière simple de gérer ces requêtes consiste à demander que les processus fassent directement un appel système `open` sur les fichiers spéciaux pour les périphériques. Si le périphérique n'est pas disponible, l'appel système `open` échoue. Le périphérique dédié est libéré lorsqu'on le ferme.

On peut également mettre en place des mécanismes spéciaux qui demandent et libèrent les périphériques dédiés. Si l'on essaie d'acquérir un périphérique qui n'est pas disponible, l'appel est bloqué au lieu d'échouer. Les processus bloqués sont placés en file d'attente. Tôt ou tard, le périphérique demandé se libère et le premier processus de la file d'attente peut l'utiliser et continuer son exécution.

Fournir une taille de bloc indépendante du périphérique

Les tailles de secteurs varient d'un disque à l'autre. C'est au logiciel d'indépendance du matériel de masquer cela et de fournir une taille de bloc uniforme aux couches supérieures. Pour ce faire, il peut par exemple traiter plusieurs secteurs comme un bloc logique unique. Ainsi, les couches supérieures ne communiquent qu'avec des périphériques abstraits qui exploitent tous la même taille de bloc logique, indépendante de la taille des secteurs physiques. En outre, certains périphériques d'E/S de caractères (les modems, par exemple) livrent leurs données un octet à la fois, alors que d'autres (comme les interfaces réseau) les délivrent en unités plus importantes. Ces différences peuvent également être masquées.

5.3.4 Les logiciels d'E/S de l'espace utilisateur

Bien que la grande majorité du logiciel d'E/S soit à l'intérieur du système d'exploitation, une petite partie est composée de bibliothèques liées avec le programme utilisateur, voire de programmes s'exécutant en dehors du noyau. Les appels système, y compris les appels système d'E/S, sont normalement émis par des procédures de bibliothèque. Quand un programme C contient l'appel

```
count = write(fd, tampon, nbytes);
```

la procédure de bibliothèque *write* est liée au programme et se trouve dans le programme binaire présent en mémoire au moment de l'exécution. La collection de toutes ces procédures de bibliothèque fait clairement partie du système d'E/S.

Si ces procédures se limitent souvent à placer leurs paramètres à l'emplacement approprié pour le système, d'autres procédures d'E/S accomplissent un travail réel. Elles réalisent, par exemple, le formatage de l'entrée et de la sortie. Un exemple qui vient du C est la fonction *printf*. Elle prend un format de chaîne et éventuellement quelques variables comme entrée, construit une chaîne ASCII, puis appelle *write* pour sortir la chaîne. L'instruction suivante est un exemple de *printf* :

```
printf("Le carré de %3d est %6d\n", i, i*i);
```

Elle formate une chaîne qui se compose des douze caractères « Le carré de », suivie de la valeur *i* sous forme de chaîne de trois caractères, puis de la chaîne de cinq caractères « est », de i^2 sous la forme de six caractères et d'un saut de ligne pour finir.

La fonction *scanf* est un exemple de procédure similaire pour l'entrée. Elle lit l'entrée et la stocke dans des variables décrites dans un format de chaîne en utilisant la même syntaxe que *printf*. La bibliothèque d'E/S standard contient un certain nombre de procédures qui concernent les E/S et qui s'exécutent dans des programmes utilisateur.

Les logiciels d'E/S ne se composent pas uniquement de procédures de bibliothèques. Le système de spoule en est une autre partie importante. Le **spoule** ou **désynchronisation des E/S** (*spool, Simultaneous Peripheral Operations Online*) est une manière de traiter les périphériques d'E/S dédiés dans un système en multiprogrammation. Prenons l'exemple d'un périphérique spoulé classique : l'imprimante. Du point de vue technique, il semble simple de laisser un processus utilisateur ouvrir le fichier spécial de l'imprimante, mais à supposer qu'un processus l'ait ouvert et qu'il ne fasse rien pendant des heures, aucun autre processus ne pourrait imprimer quoi que ce soit.

Au lieu de cela, on crée un processus spécial appelé **démon** (*daemon*) et un répertoire spécial appelé **répertoire de spoule** (*spooling directory*). Pour imprimer un fichier, un processus avant toute chose génère le fichier à imprimer et le place dans le répertoire de spoule. C'est à la charge du démon, qui est le seul processus à disposer de l'autorisation d'utiliser le fichier spécial de l'imprimante, d'imprimer les fichiers du répertoire de spoule. En protégeant le fichier spécial de l'accès direct par les utilisateurs, on évite qu'il reste inutilement ouvert pendant une durée indéterminée.

Le spoule ne sert pas uniquement aux imprimantes. Par exemple, le transfert de fichiers en réseau exploite souvent un démon de réseau. Pour envoyer un fichier quelque part, l'utilisateur le place dans un répertoire de spoule du réseau. Plus tard, le démon réseau les récupère et les transmet. Par exemple, sur l'internet, il existe des milliers de groupes de discussion abordant de nombreux sujets. Dans ce cadre, le système Usenet fait un usage particulier de la transmission de fichiers spoulés. Ce réseau se compose de millions d'ordinateurs de par le monde qui communiquent *via* l'internet. Pour publier un article de forum (*news*), l'utilisateur invoque un programme, qui accepte de publier l'article et de le déposer dans un répertoire de spoule pour une transmission ultérieure aux autres ordinateurs. L'ensemble du système Usenet s'exécute en dehors du système d'exploitation.

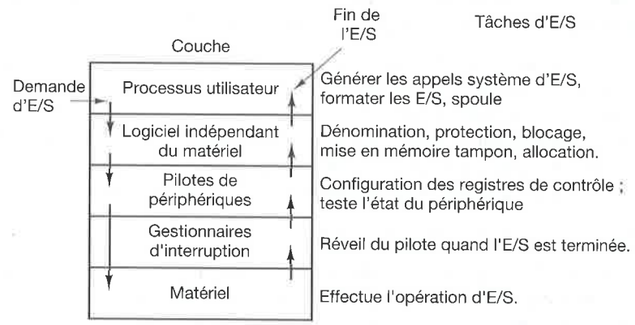

Figure 5.17 • Les couches d'un système d'E/S et les principales fonctions de chaque couche.

La figure 5.17 récapitule le système d'E/S. Elle représente toutes les couches et les tâches qu'elles accomplissent. En commençant par la couche inférieure, les couches

sont : le matériel, les gestionnaires d'interruptions, les pilotes de périphériques, le logiciel d'indépendance du matériel et, pour finir, les processus utilisateur.

Les flèches de la figure 5.17 matérialisent le flot de contrôle. Lorsqu'un programme utilisateur tente de lire un bloc d'un fichier, par exemple, le système d'exploitation est invoqué pour effectuer l'appel. Le logiciel d'indépendance du matériel le recherche, par exemple, dans le cache de la mémoire tampon. Si le bloc demandé ne s'y trouve pas, le logiciel appelle le pilote du périphérique pour qu'il envoie la requête au matériel afin de le récupérer sur le disque. Le processus est alors bloqué jusqu'à ce que l'opération de disque soit terminée.

Quand le disque a terminé, le matériel génère une interruption. Le gestionnaire d'interruptions s'exécute pour découvrir ce qui s'est produit, autrement dit pour connaître le périphérique qui requiert son attention. Il récupère alors l'état du périphérique et réveille le processus dormant pour terminer la demande d'E/S et permettre au processus utilisateur de poursuivre.

5.4 Les disques

Nous allons maintenant aborder l'étude de certains périphériques d'E/S, en commençant par les disques. Nous nous pencherons ensuite sur les horloges, les claviers et les systèmes d'affichage.

5.4.1 Les différents types de disques

Il existe une grande variété de types de disques. Les plus courants sont les disques magnétiques (disques durs et disquettes). Ils se caractérisent par le fait qu'ils lisent et écrivent à la même vitesse, ce qui en fait des mémoires secondaires idéales (pagination, système de fichiers, etc.). Des ensembles de disques servent parfois de moyen de stockage à haute fiabilité. Pour la distribution des programmes, des données et des films, on fait appel à différents types de disques optiques (CD-ROM, CD inscriptibles et DVD). Dans les prochaines sections, nous commencerons par décrire le matériel, puis nous verrons les logiciels associés à ces périphériques.

Les disques magnétiques

Les disques magnétiques s'organisent en cylindres, chacun contenant autant de pistes qu'il y a de têtes empilées verticalement. Les pistes sont divisées en secteurs, dont le nombre, sur la circonférence, se situe généralement entre 8 et 32 pour les disquettes et peut atteindre jusqu'à plusieurs centaines pour les disques durs. Quant aux têtes, on en compte entre 1 et 16.

On trouve peu d'électronique sur les anciens disques magnétiques, qui ne délivrent alors qu'un simple flot de bits série. Sur ces disques, le contrôleur réalise la plus grande part du travail. D'autres disques, et en particulier les disques **IDE** (*Integrated Drive Electronics*) et **SATA** (*Serial ATA*), contiennent un microcontrôleur qui accomplit